Oracle® Developer Studio 12.5: Debugging a Program with dbx

Part No: E60748
June 2016

ORACLE®

Oracle Developer Studio 12.5: Debugging a Program with dbx

Part No: E60748

Référence: E60748

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Using This Documentation 25

1 Getting Started With dbx 27

- Compiling Your Code for Debugging 27
- Starting dbx or dbxtool and Loading Your Program 28
- Running Your Program in dbx 29
- Debugging Your Program With dbx 30
  - Examining a Core File 31
  - Setting Breakpoints 32
  - Stepping Through Your Program 33
  - Looking at the Call Stack 34
  - Examining Variables 35
  - Finding Memory Access Problems and Memory Leaks 35
- Quitting dbx 36
- Accessing dbx Online Help 37

2 Starting dbx 39

- Starting a Debugging Session 39
- Debugging a Core File 40
  - Debugging a Core File in the Same Operating Environment 40
  - If Your Core File Is Truncated 41
  - Debugging a Mismatched Core File 42
- Using the Process ID 44
- dbx Startup Sequence 45
- Setting Startup Properties 45
  - Mapping the Compile-Time Directory to the Debug-Time Directory 46
  - Setting dbx Environment Variables 46
  - Creating Your Own dbx Commands 47

Compiling a Program for Debugging 47
Compiling With the -g Option 47
Using a Separate Debug File 47
Compressed Debug Sections (Oracle Solaris Only) 50
Debugging Optimized Code 51
Parameters and Variables 52
Inlined Functions 52
Code Compiled Without the -g Option 53
Shared Libraries Require the -g Option for Full dbx Support 53
Completely Stripped Programs 53
Quitting Debugging 54
Stopping a Process Execution 54
Detaching a Process From dbx 54
Killing a Program Without Terminating the Session 54
Saving and Restoring a Debugging Run 55
Using the save Command 55
Saving a Series of Debugging Runs as Checkpoints 57
Restoring a Saved Run 57
Saving and Restoring Using replay 58

3 Customizing dbx 59
Using the dbx Initialization File 59
Creating a .dbxrc File 60
Initialization File Sample 60
Setting dbxenv Variables 60
dbxenv Variables and the Korn Shell 66

4 Viewing and Navigating To Code 67
Navigating To Code 67
Navigating To a File 68
Navigating To Functions 68
Printing a Source Listing 69
Walking the Call Stack to Navigate To Code 69
Types of Program Locations 70
Program Scope 70
Variables That Reflect the Current Scope 70

Visiting Scope 71
Qualifying Symbols With Scope Resolution Operators 72
Backquote Operator 73
C++ Double-Colon Scope Resolution Operator 73
Block Local Operator 74
Linker Names 75
Locating Symbols 75
Printing a List of Occurrences of a Symbol 75
Determining Which Symbol `dbx` Uses 76
Scope Resolution Search Path 77
Relaxing the Scope Lookup Rules 77
Viewing Variables, Members, Types, and Classes 78
Looking Up Definitions of Variables, Members, and Functions 78
Looking Up Definitions of Types and Classes 79
Debugging Information in Object Files and Executables 81
Object File Loading 82
Compiler and Linker Options to Support Debugging 82
Listing Debugging Information for Modules 84
Listing Modules 85
Finding Source and Object Files 86

5 Controlling Program Execution 87
Running a Program 87
Attaching `dbx` to a Running Process 88
Detaching `dbx` From a Process 89
Stepping Through a Program 90
Controlling Single Stepping Behavior 90
Stepping Into a Specific or Last Function 91
Continuing Execution of a Program 91
Calling a Function 92
Call Safety 93
Using Ctrl+C to Stop a Process 94
Event Management 94

6 Setting Breakpoints and Traces 97
Setting Breakpoints 97

Setting a Breakpoint at a Line of Source Code 98
Setting a Breakpoint in a Function 99
Setting Multiple Breakpoints in C++ Programs 100
Setting Data Change Breakpoints (Watchpoints) 102
Setting Filters on Breakpoints 105
Qualifying Breakpoints With Conditional Filters 105
Qualifying Breakpoints With Caller Filters 106
Filters and Multithreading 107
Tracing Execution 108
Setting a Trace 108
Controlling the Speed of a Trace 108
Directing Trace Output to a File 108
Executing `dbx` Commands at a Line 109
Setting Breakpoints in Dynamically Loaded Libraries 109
Listing and Deleting Breakpoints 110
Listing Breakpoints and Traces 110
Deleting Specific Breakpoints Using Handler ID Numbers 110
Enabling and Disabling Breakpoints 111
Efficiency Considerations 111

7 Using the Call Stack 113
Finding Your Place on the Stack 113
Walking the Stack and Returning Home 114
Moving Up and Down the Stack 114
Moving Up the Stack 114
Moving Down the Stack 114
Moving to a Specific Frame 115
Popping the Call Stack 115
Hiding Stack Frames 116
Displaying and Reading a Stack Trace 116

8 Evaluating and Displaying Data 119
Evaluating Variables and Expressions 119
Verifying Which Variable `dbx` Uses 119
Variables Outside the Scope of the Current Function 120
Printing the Value of a Variable, Expression, or Identifier 120
Printing C++ Pointers 120

Evaluating Unnamed Arguments in C++ Programs 121
Dereferencing Pointers 122
Monitoring Expressions 122
Stop the Display (Undisplaying) 123
Assigning a Value to a Variable 123
Evaluating Arrays 123
Array Slicing 124
Using Slices 127
Using Strides 127
Using Pretty-Printing 128
Invoking Pretty-Printing 129
Call-Based Pretty-Printing 129
Python Pretty-Print Filters (Oracle Solaris) 131

9 Using Runtime Checking 135
Capabilities of Runtime Checking 135
When to Use Runtime Checking 136
Runtime Checking Requirements 136
Using Runtime Checking 137
Enabling Memory Use and Memory Leak Checking 137
Enabling Memory Access Checking 137
Enabling All Runtime Checking 137
Disabling Runtime Checking 137
Running Your Program 138
Using Access Checking 140
Understanding the Memory Access Error Report 141
Memory Access Errors 142
Using Memory Leak Checking 143
Detecting Memory Leak Errors 144
Possible Leaks 144
Checking for Leaks 145
Understanding the Memory Leak Report 145
Fixing Memory Leaks 148
Using Memory Use Checking 148
Suppressing Errors 149
Types of Suppression 150
Suppressing Error Examples 151

Default Suppressions 151
Using Suppression to Manage Errors 152
Using Runtime Checking on a Child Process 152
Using Runtime Checking on an Attached Process 156
Attached Process on a System Running Oracle Solaris 156
Attached Process on a System Running Linux 157
Using Fix and Continue With Runtime Checking 157
Runtime Checking Application Programming Interface 159
Using Runtime Checking in Batch Mode 160
`bcheck` Syntax 160
`bcheck` Examples 160
Enabling Batch Mode Directly From `dbx` 161
Troubleshooting Tips 161
Runtime Checking Limitations 162
Performance Improves With More Symbols and Debug Information 162
`SIGSEGV` and `SIGALTSTACK` Signals Are Restricted on x86 Platforms 162
Performance Improves When Sufficient Patch Area Is Available Within 8 MB of All Existing Code (SPARC Platforms Only). 163
Runtime Checking Errors 165
Access Errors 165
Memory Leak Errors 169

10 Debugging Multithreaded Applications 171
Understanding Multithreaded Debugging 171
Thread Information 172
Viewing the Context of Another Thread 173
Viewing the Threads List 174
Resuming Execution 174
Understanding Thread Creation Activity 175
Understanding LWP Information 176

11 Debugging Child Processes 179
Attaching to Child Processes 179
Following the `exec` Function 180
Following the `fork` Function 180
Interacting With Events 180

12 Debugging OpenMP Programs 181
How Compilers Transform OpenMP Code 181
`dbx` Functionality Available for OpenMP Code 182
Single-Stepping Into a Parallel Region 182
Printing Variables and Expressions 183
Printing Region and Thread Information 183
Serializing the Execution of a Parallel Region 186
Using Stack Traces 186
Using the `dump` Command 187
Using Events 188
Execution Sequence of OpenMP Code 189

13 Working With Signals 191
Understanding Signal Events 191
Catching Signals 192
Changing the Default Signal Lists 193
Trapping the FPE Signal (Oracle Solaris Only) 193
Sending a Signal to a Program 196
Automatically Handling Signals 196

14 Debugging C++ With `dbx` 197
Using `dbx` With C++ 197
Exception Handling in `dbx` 198
Commands for Handling Exceptions 198
Examples of Exception Handling 201
Debugging With C++ Templates 202
Template Example 203
Commands for C++ Templates 204

15 Debugging Fortran Using `dbx` 209
Debugging Fortran 209
Current Procedure and File 209
Uppercase Letters 210
Sample `dbx` Session 210
Debugging Segmentation Faults 213
Using `dbx` to Locate Problems 213

Locating Exceptions 214
Tracing Calls 214
Working With Arrays 215
Fortran Allocatable Arrays 216
Showing Intrinsic Functions 217
Showing Complex Expressions 218
Showing Interval Expressions 218
Showing Logical Operators 219
Viewing Fortran Derived Types 220
Pointer to Fortran Derived Type 221
Object Oriented Fortran 223
Allocatable Scalar Type 223

16 Debugging a Java Application With dbx 225
Using dbx With Java Code 225
Capabilities of dbx With Java Code 225
Limitations of dbx With Java Code 225
Environment Variables for Java Debugging 226
Starting to Debug a Java Application 226
Debugging a Class File 227
Debugging a JAR File 227
Debugging a Java Application That Has a Wrapper 228
Attaching dbx to a Running Java Application 228
Debugging a C Application or C++ Application That Embeds a Java Application 229
Passing Arguments to the JVM Software 230
Specifying the Location of Your Java Source Files 230
Specifying the Location of Your C Source Files or C++ Source Files 230
Specifying a Path for Class Files That Use Custom Class Loaders 230
Setting Breakpoints on Java Methods 231
Setting Breakpoints in Native (JNI) Code 231
Customizing Startup of the JVM Software 231
Specifying a Path Name for the JVM Software 232
Passing Run Arguments to the JVM Software 232
Specifying a Custom Wrapper for Your Java Application 233
Specifying 64-bit JVM Software 234
dbx Modes for Debugging Java Code 235

Switching From Java or JNI Mode to Native Mode 235
Switching Modes When You Interrupt Execution 236
Using `dbx` Commands in Java Mode 236
Java Expression Evaluation in `dbx` Commands 236
Static and Dynamic Information Used by `dbx` Commands 237
Commands With Identical Syntax and Functionality in Java Mode and Native Mode 238
Commands With Different Syntax in Java Mode 239
Commands Valid Only in Java Mode 240

17 Debugging at the Machine-Instruction Level 241
Using `dbx` at the Machine-Instruction Level 241
Examining the Contents of Memory 241
Using the `examine` or `x` Command 242
Using the `dis` Command 244
Using the `listi` Command 245
Stepping and Tracing at Machine-Instruction Level 246
Single-Stepping at the Machine-Instruction Level 246
Tracing at the Machine-Instruction Level 247
Setting Breakpoints at the Machine-Instruction Level 248
Setting a Breakpoint at an Address 248
Using the `regs` Command 248
Platform-Specific Registers 251

18 Using `dbx` With the Korn Shell 259
`ksh`-88 Features Not Implemented 259
Extensions to `ksh`-88 260
Renamed Commands 260
Rebinding of Editing Functions 260

19 Debugging Shared Libraries 263
Dynamic Linker 263
Link Map 263
Startup Sequence and `.init` Sections 264
Procedure Linkage Tables 264
Setting Breakpoints in Shared Libraries 264

Setting a Breakpoint in an Explicitly Loaded Library 265

A Modifying a Program State 267
Impacts of Running a Program Under dbx 267
Commands That Alter the State of the Program 268
assign Command 268
pop Command 268
call Command 269
print Command 269
when Command 269
fix Command 270
cont at Command 270

B Event Management 271
Event Handlers 271
Creating Event Handlers 272
Manipulating Event Handlers 272
Using Event Counters 273
Event Safety 273
Setting Event Specifications 274
Breakpoint Event Specifications 274
Data Change Event Specifications 277
System Event Specifications 279
Execution Progress Event Specifications 283
Tracked Thread Event Specifications 285
Other Event Specifications 286
Event Specification Modifiers 289
-if Modifier 289
-resumeone Modifier 289
-in Modifier 290
-disable Modifier 290
-count *n*, -count infinity Modifier 290
-temp Modifier 290
-instr Modifier 291
-thread Modifier 291
-lwp Modifier 291

`-hidden` Modifier 291
`-perm` Modifier 292
Parsing and Ambiguity 292
Using Predefined Variables 292
Variables Valid for `when` Command 294
Variables Valid for `when` Command and Specific Events 294
Event Handler Examples 296
Setting a Breakpoint for Store to an Array Member 296
Implementing a Simple Trace 296
Enabling a Handler While Within a Function 296
Determining the Number of Lines Executed 297
Determining the Number of Instructions Executed by a Source Line 297
Enabling a Breakpoint After an Event Occurs 297
Resetting Application Files for `replay` 298
Checking Program Status 298
Catch Floating-Point Exceptions 298

C Macros 301
Additional Uses of Macro Expansion 301
Macro Definitions 302
Compiler and Compiler Options 303
Tradeoffs in Functionality 303
Limitations 304
Skimming Errors 304
Using the `pathmap` Command to Improve Skimming 305

D Command Reference 307
`adi assign` Command 307
Native Mode Syntax 307
`adi examine` Command 308
Native Mode Syntax 308
`assign` Command 308
Native Mode Syntax 308
Java Mode Syntax 309
`attach` Command 309
Syntax 309

bsearch Command 310
Syntax 310
call Command 310
Native Mode Syntax 311
Java Mode Syntax 311
cancel Command 312
catch Command 312
Syntax 312
check Command 313
Syntax 313
clear Command 316
Syntax 316
collector Command 316
Syntax 317
collector archive Command 318
collector dbxsample Command 318
collector disable Command 318
collector enable Command 319
collector heaptrace Command 319
collector hwprofile Command 319
collector limit Command 320
collector pause Command 321
collector profile Command 321
collector resume Command 321
collector sample Command 321
collector show Command 322
collector status Command 323
collector store Command 323
collector synctrace Command 324
collector tha Command 324
collector version Command 324
cont Command 325
Syntax 325
dalias Command 325
Syntax 325
dbx Command 326

- Native Mode Syntax 326
- Java Mode Syntax 327
- Options 328

`dbxenv` Command 328
- Syntax 329

`debug` Command 329
- Native Mode Syntax 329
- Java Mode Syntax 330
- Options 331

`delete` Command 332
- Syntax 332

`detach` Command 332
- Native Mode Syntax 333
- Java Mode Syntax 333

`dis` Command 333
- Syntax 333
- Options 334

`display` Command 334
- Native Mode Syntax 334
- Java Mode Syntax 335

`down` Command 336
- Syntax 336

`dump` Command 336
- Syntax 336

`edit` Command 337
- Syntax 337

`examine` Command 337
- Syntax 337

`exception` Command 339
- Syntax 339

`exists` Command 339
- Syntax 340

`file` Command 340
- Syntax 340

`files` Command 340
- Native Mode Syntax 340

Java Mode Syntax 341
fix Command 341
Syntax 341
fixed Command 342
fortran_modules Command 342
Syntax 342
frame Command 342
Syntax 343
func Command 343
Native Mode Syntax 343
Java Mode Syntax 343
funcs Command 344
Syntax 344
gdb Command 345
Syntax 345
handler Command 346
Syntax 346
hide Command 346
Syntax 346
ignore Command 347
Syntax 347
import Command 347
Syntax 347
intercept Command 348
Syntax 348
java Command 349
Syntax 349
jclasses Command 349
Syntax 349
joff Command 350
jon Command 350
jpkgs Command 350
kill Command 350
Syntax 350
language Command 351
Syntax 351

line Command 351
Syntax 352
Examples 352
list Command 352
Syntax 352
listi Command 354
loadobject Command 354
Syntax 355
loadobject -dumpelf Command 355
loadobject -exclude Command 356
loadobject -hide Command 356
loadobject -list Command 357
loadobject -load Command 358
loadobject -unload Command 358
loadobject -use Command 359
lwp Command 359
Syntax 359
lwps Command 360
macro Command 360
Syntax 360
mmapfile Command 361
Syntax 361
Example 361
module Command 362
Syntax 362
modules Command 362
Syntax 362
native Command 363
Syntax 363
next Command 363
Native Mode Syntax 364
Java Mode Syntax 364
nexti Command 365
Syntax 365
omp_loop Command 366
omp_pr Command 366

Syntax ... 366
omp_serialize Command ... 367
Syntax ... 367
omp_team Command ... 367
Syntax ... 367
omp_tr Command ... 367
Syntax ... 368
pathmap Command ... 368
Syntax ... 368
Examples ... 369
pop Command ... 370
Syntax ... 370
print Command ... 370
Native Mode Syntax ... 370
Java Mode Syntax ... 373
proc Command ... 374
Syntax ... 374
prog Command ... 374
Syntax ... 374
quit Command ... 375
Syntax ... 375
regs Command ... 375
Syntax ... 375
Example (SPARC platform) ... 376
replay Command ... 376
Syntax ... 376
rerun Command ... 377
Syntax ... 377
restore Command ... 377
Syntax ... 377
rprint Command ... 377
Syntax ... 378
rtc showmap Command ... 378
rtc skippatch Command ... 378
Syntax ... 379
run Command ... 379

Native Mode Syntax 379
Java Mode Syntax 380
`runargs` Command 380
Syntax 380
`save` Command 381
Syntax 381
`scopes` Command 381
`search` Command 381
Syntax 381
`showblock` Command 382
Syntax 382
`showleaks` Command 382
Syntax 382
`showmemuse` Command 383
Syntax 383
`source` Command 383
Syntax 384
`status` Command 384
Syntax 384
Example 384
`step` Command 384
Native Mode Syntax 385
Java Mode Syntax 386
`stepi` Command 386
Syntax 387
`stop` Command 387
Syntax 387
`stopi` Command 392
Syntax 392
`suppress` Command 393
Syntax 393
`sync` Command 395
Syntax 396
`syncs` Command 396
`thread` Command 396
Native Mode Syntax 396

Java Mode Syntax 397
`threads` Command 398
Native Mode Syntax 398
Java Mode Syntax 399
`trace` Command 400
Syntax 400
`tracei` Command 404
Syntax 404
`uncheck` Command 405
Syntax 405
`undisplay` Command 406
Native Mode Syntax 406
Java Mode Syntax 406
`unhide` Command 406
Syntax 407
`unintercept` Command 407
Syntax 407
`unsuppress` Command 408
Syntax 408
`unwatch` Command 409
Syntax 409
`up` Command 409
Syntax 410
`use` Command 410
`watch` Command 410
Syntax 410
`whatis` Command 411
Native Mode Syntax 411
Java Mode Syntax 412
`when` Command 413
Syntax 413
`wheni` Command 415
Syntax 415
`where` Command 415
Native Mode Syntax 415
Java Mode Syntax 416

`whereami` Command ... 417
Syntax ... 417
`whereis` Command ... 417
Syntax ... 417
`which` Command ... 418
Syntax ... 418
`whocatches` Command ... 418
Syntax ... 419

Index ... 421

Using This Documentation

- **Overview** – Describes how to use the `dbx` command-line debugger, an interactive source level debugging tool
- **Audience –** Application developers, system developers, architects, support engineers
- **Required knowledge** – Familiarity with the Fortran, C, C++, or Java programming language and some understanding of the Oracle Solaris operating system, or the Linux operating system, and UNIX® commands

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E60778-01`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ CHAPTER 1

Getting Started With dbx

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

You can use dbx to debug an application written in C, C++, including the C++11 and C11 standard, or Fortran. You can also, with some limitations (see "Limitations of dbx With Java Code" on page 225), debug an application that is a mixture of Java™ code and C JNI (Java Native Interface) code or C++ JNI code.

dbxtool provides a graphical user interface for dbx.

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

- "Compiling Your Code for Debugging" on page 27
- "Starting dbx or dbxtool and Loading Your Program" on page 28
- "Running Your Program in dbx" on page 29
- "Debugging Your Program With dbx" on page 30
- "Quitting dbx" on page 36
- "Accessing dbx Online Help" on page 37

Compiling Your Code for Debugging

You must prepare your program for source-level debugging with dbx by compiling it with the -g option, which is accepted by the C compiler, C++ compiler, Fortran compiler, and Java compiler. dbx also supports code written in the C++11 and C11 standard. For more information, see "Compiling a Program for Debugging" on page 47.

Starting dbx or dbxtool and Loading Your Program

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

```
$ dbx
```

To start dbxtool, type the dbxtool command in a shell prompt:

```
$ dbxtool
```

To start dbx and load the program to be debugged:

```
$ dbx program-name
```

To start dbxtool and load the program to be debugged:

```
$ dbxtool program-name
```

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

```
$ dbx program-name {.class | .jar}
```

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

```
$ dbx - process-ID
```

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

```
$ dbxtool - process-ID
```

If you don’t know the process ID of the process, include the pgrep command in the dbx command to find and attach to the process. For example:

```
$ dbx - `pgrep Freeway`
Reading -
Reading ld.so.1
Reading libXm.so.4
Reading libgen.so.1
Reading libXt.so.4
Reading libX11.so.4
Reading libce.so.0
Reading libsocket.so.1
Reading libm.so.1
Reading libw.so.1
Reading libc.so.1
Reading libSM.so.6
Reading libICE.so.6
```

```
Reading libXext.so.0
Reading libnsl.so.1
Reading libdl.so.1
Reading libmp.so.2
Reading libc_psr.so.1
Attached to process 1855
stopped in _libc_poll at 0xfef9437c
0xfef9437c: _libc_poll+0x0004:   ta     0x8
Current function is main
   48   XtAppMainLoop(app_context);
(dbx)
```

For more information about the dbx command and startup options, see "dbx Command" on page 326 and the dbx(1) man page, or type `dbx -h`.

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

(dbx) **debug** *program-name*

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

(dbx) **debug** *program-name*{.class | .jar

dbx debug program-name{.class | .jar

}

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

(dbx) **debug** *program-name process-ID*

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

(dbx) **debug** *program-name*{.class | .jar} *process-ID*

For more information, see "debug Command" on page 329.

Running Your Program in dbx

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

```
run [ arguments ] [ < inputfile ] [ > output-file ]
```

For example:

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

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

If you repeat the `run` command without arguments, the program restarts using the arguments or redirection from the previous `run` command. You can reset the options using the `rerun` command. For more information about the `run` command, see "`run` Command" on page 379. For more information about the `rerun` command, see "`rerun` Command" on page 377.

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

Debugging Your Program With `dbx`

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

- To determine where and why it is crashing. Strategies for locating the cause of a crash include:
  - Running the program in `dbx`. `dbx` reports the location of the crash when it occurs.
  - Examining the core file and looking at a stack trace. See "Examining a Core File" on page 31 and "Looking at the Call Stack" on page 34.
- To determine why your program is returning incorrect results. Strategies include:
  - Setting breakpoints to stop execution so that you can check your program's state and look at the values of variables. See "Setting Breakpoints" on page 32 and "Examining Variables" on page 35.
  - Stepping through your code one source line at a time to monitor how the program state changes. See "Stepping Through Your Program" on page 33.
- To find a memory leak or memory management problem. Runtime checking (RTC) lets you detect runtime errors such as memory access errors and memory leak errors and enables you to monitor memory usage. See "Finding Memory Access Problems and Memory Leaks" on page 35.

Examining a Core File

To determine where your program is crashing, you might want to examine the core file, which is the memory image of your program when it crashed. You can use the `where` command to determine where the program was executing when it dumped core. See "`where` Command" on page 415

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

To debug a core file, type:

```
$ dbx program-name core
```

or

```
$ dbx - core
```

In the following example, the program has crashed with a segmentation fault and dumped core. First, `dbx` started with the core file loaded. Then, the `where` command displays a stack trace, which shows that the crash occurred at line 9 of the file `foo.c`.

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

For more information about debugging core files, see "Debugging a Core File" on page 40. For more information about using the call stack, see "Looking at the Call Stack" on page 34.

Note - If your program is dynamically linked with any shared libraries, debug the core file in the same operating environment in which it was created. For information on debugging a core file that was created in a different operating environment, see "Debugging a Mismatched Core File" on page 42.

Setting Breakpoints

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

When your program stops at a breakpoint, you can then examine the state of program and the values of variables. dbx enables you to set many types of breakpoints "Using Ctrl+C to Stop a Process" on page 94.

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

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

For more information about the stop in command, see Chapter 6, "Setting Breakpoints and Traces" and "stop Command" on page 387.

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

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

For more information about the stop at command, see "Setting a Breakpoint at a Line of Source Code" on page 98 and "stop Command" on page 387.

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

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

```
14    }
(dbx) stop at 13
(4) stop at “t.c”:13
```

To continue execution of your program after it has stopped at a breakpoint, use the cont command (see “Continuing Execution of a Program” on page 91 and “cont Command” on page 325).

To display a list of all current breakpoints, use the status command:

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

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

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

Stepping Through Your Program

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

The step up command continues execution until the current function returns control to the function that called it.

The step to command attempts to step into a specified function in the current source line, or if no function is specified, into the last function called as determined by the assembly code for the current source line.

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

The following example shows the use of the step and next commands as well as the breakpoint set in “Setting Breakpoints” on page 32.

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

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

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

For more information about stepping through your program, see "Stepping Through a Program" on page 90. For more information about the `step` and `next` commands, see "`step` Command" on page 384 and "`next` Command" on page 363.

Looking at the Call Stack

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

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

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

For functions that were compiled with the `-g` option, the argument names and their types are known so accurate values are displayed. For functions without debugging information, hexadecimal numbers are displayed for the arguments. These numbers are not necessarily meaningful. For example, in the stack trace above, frame 1 shows the contents of the SPARC input registers `$i0` through `$i5`.Only the contents of registers `$i0` through `$i1` are meaningful because only two arguments were passed to `printf` in the example shown in "Stepping Through Your Program" on page 33.

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

For more information about the call stack, see "Efficiency Considerations" on page 111. For more information about the current scope, see "Program Scope" on page 70.

Examining Variables

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

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

You can track when the values of variables and expressions change using data change breakpoints (see "Setting Data Change Breakpoints (Watchpoints)" on page 102). For example, to stop execution when the value of the variable count changes, type:

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

Finding Memory Access Problems and Memory Leaks

Runtime checking consists of two parts: memory access checking, and memory use and leak checking. *Access checking* checks for improper use of memory by the debugged application.

Memory use and leak checking involves keeping track of all the outstanding heap space and then on demand or at termination of the program, scanning the available data spaces and identifying the space that has no references.

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

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

To enable memory use and memory leak checking:

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

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

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

For detailed information about using runtime checking, see Chapter 9, "Using Runtime Checking".

Quitting dbx

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

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

```
(dbx) quit
```

When you start `dbx` and attach it to a running process by providing the process ID, the process survives and continues when you quit the debugging session. `dbx` performs an implicit `detach` before quitting the session.

For more information about quitting `dbx`, see "Quitting Debugging" on page 54.

Accessing dbx Online Help

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

```
(dbx) help
```

♦ ♦ ♦ CHAPTER 2

Starting dbx

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

- "Starting a Debugging Session" on page 39
- "Debugging a Core File" on page 40
- "Using the Process ID" on page 44
- "dbx Startup Sequence" on page 45
- "Setting Startup Properties" on page 45
- "Compiling a Program for Debugging" on page 47
- "Debugging Optimized Code" on page 51
- "Quitting Debugging" on page 54
- "Saving and Restoring a Debugging Run" on page 55

Starting a Debugging Session

How you start dbx depends on what you are debugging, where you are, what you need dbx to do, how familiar you are with dbx, and whether you have set up any dbxenv variables.

You can use dbx entirely from the command line in a terminal window, or run dbxtool, a graphical user interface for dbx. For information about dbxtool, see the dbxtool man page and the online help in dbxtool.

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

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

```
$ dbx program-name
```

or

```
$ dbxtool program-name
```

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

```
$ dbx program-name{.class | .jar}
```

The Oracle Developer Studio software includes two dbx binaries: a 32-bit dbx that can debug 32-bit programs only and a 64-bit dbx that can debug both 32-bit and 64-bit programs. When you start dbx, it determines which of its binaries to execute. On 64-bit operating systems, the 64-bit dbx is the default.

Note - On the Linux OS, the 64-bit dbx cannot debug 32-bit programs. To debug a 32-bit program on the Linux OS, you must start the 32-bit dbx with the dbx command option -xexec32 or set the DBX_EXEC_32 environment variable.

When using the 32-bit dbx on a 64-bit Linux OS, do not use the debug command or set the follow_fork_mode environment variable to *child* if the result will be execution of a 64-bit program. Exit dbx and start the 64-bit dbx to debug a 64-bit program.

For more information about the dbx command and startup options, see "dbx Command" on page 326 and the dbx(1) man page.

Debugging a Core File

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

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

Debugging a Core File in the Same Operating Environment

To debug a core file, use the following command:

```
$ dbx program-name core
```

or

```
$ dbxtool program-name core
```

If you issue the following command, `dbx` determines the program name from the core file:

```
$ dbx - core
```

or

```
$ dbxtool - core
```

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

```
(dbx) debug -c core program-name
```

If you substitute `-` for the program name,`dbx` will attempt to extract the program name from the core file. `dbx` might not find the executable if its full path name is not available in the core file. If `dbx` does not find the executable, specify the complete path name of the binary when you tell `dbx` to load the core file.

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

Use the `where` command to determine where the program was executing when it dumped core.

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

If Your Core File Is Truncated

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

If the core file is incomplete, and the stack segment is missing, then stack trace information is not available. If the runtime linker information is missing, then the list of load objects is not available. In this case, you get an error message about `librtld_db.so` not being initialized. If the list of light weight processes (LWPs) is missing, then no thread information, LWP information, or stack trace information is available. If you run the `where` command, you get an error saying the program was not active.

Debugging a Mismatched Core File

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

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

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

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

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

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

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

  ```
  pathmap / /net/core-host
  ```
- The single-argument mode for the `pathmap` command does not work with load object path names, so use the two argument from-path to-path mode.

- Debugging the core file is likely to work better if the `dbx`-host has either the same or a more recent version of the Oracle Solaris operating system than the core-host, though this setup is not always necessary.
- The system libraries that you might need are as follows:
  - For the runtime linker:

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

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

    `/usr/lib/64/ld.so.1`

    `/usr/lib/64/librtld_db.so.1`
  - For the threads library, depending on which implementation of `libthread` you are using:

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

    `/usr/lib/64/libthread_db.so.1`

    You will need the 64-bit versions of the *xxx*`_db.so` libraries if `dbx` is running on a 64-bit capable version of the Oracle Solaris OS since these system libraries are loaded and used as part of `dbx`, not as part of the target program.

    The `ld.so.1` libraries are part of the core file image like `libc.so` or any other library, so you need the 32-bit `ld.so.1` library or 64-bit `ld.so.1` library that matches the program that created the core file.
- If you are looking at a core file from a threaded program and the `where` command does not display a stack, try using `lwp` commands. For example:.

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

The `-setfp` and `-resetfp` options of the `lwp` command are useful when the frame pointer (`fp`) of the LWP is corrupted. These options work when debugging a core file, where `assign $fp=...` is unavailable.

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

▼ To Eliminate Shared Library Problems and Debug a Mismatched Core File

1. **Set the `dbxenv` variable `core_lo_pathmap` to `on`.**

2. **Use the `pathmap` command to indicate where the correct libraries for the core file are located.**

3. **Use the `debug` command to load the program and the core file.**

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

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

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

Using the Process ID

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

$ **dbx** *programname process-ID*

or

dbxtool *program-name processD*

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

$ **dbx** *program-name*`{.class | .jar}` *process-ID*

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

$ **dbx -** *processID*

or

$ **dbxtool -** *processID*

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

For more information, see "Attaching dbx to a Running Process" on page 88.

dbx Startup Sequence

When you start dbx, if you do not specify the -S option, dbx looks for the installed startup file, dbxrc, in the directory /*install-dir*/lib. The default installation directory is /opt/solstudio12.4 on Oracle Solaris platforms and /opt/oracle/solstudio12.4 on Linux platforms. If your Oracle Developer Studio software is not installed in the default directory, dbx derives the path to the dbxrc file from the path to the dbx executable.

Then dbx searches for a .dbxrc file in the current directory, then in $HOME. You can specify a different startup file than .dbxrc explicitly by specifying the file path using the -s option. For more information, see "Using the dbx Initialization File" on page 59.

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

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

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

Setting Startup Properties

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

Mapping the Compile-Time Directory to the Debug-Time Directory

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

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

Add common pathmaps to your `.dbxrc` file.

The following command establishes a new mapping from the directory *from* to the directory *to*

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

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

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

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

For more information, see "`pathmap` Command" on page 368.

Setting dbx Environment Variables

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

You can also set `dbxenv` variables. See "Saving and Restoring Using `replay`" on page 58 for more information about the `.dbxrc` file and about setting these variables.

For more information, see "Setting `dbxenv` Variables" on page 60 and "`dbxenv` Command" on page 328.

Creating Your Own dbx Commands

You can create your own dbx commands using the kalias or dalias commands. For more information, see "dalias Command" on page 325.

Compiling a Program for Debugging

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

Compiling With the -g Option

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

For example, to compile using the C++ compiler:

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

For the C++ compiler:

- The -g option alone, with no optimization level specified, enables capturing debugging information and disables inlining of functions.
- The -g option used with the -O option or the -xO*level* option turns on debugging information and does not disable inlining of functions. This set of options produces limited debugging information and inlined functions.
- The -g0 (zero) option turns on debugging information and does not affect inlining of functions. You cannot debug inline functions in code compiled with the -g0 option. The -g0 option can significantly decrease link time and dbx startup time, depending on the use of inlined functions by the program.

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

Using a Separate Debug File

dbx enables you to use options in the objcopy command on Linux platforms and the gobjcopy command on Oracle Solaris platforms to copy the debugging information from an executable to a separate debug file, strip that information from the executable, and create a link between these two files.

`dbx` searches for the separate debug file in the following order and reads the debugging information from the first file it finds:

1. The directory that contains the executable file.
2. A subdirectory named `debug` in the directory that contains the executable file.
3. A subdirectory of the global debug file directory, which you can view or change if the `dbxenv` variable `debug_file_directory` is set to the path name of the directory. The default value of the environment variable is `/usr/lib/debug`.

For example, the following procedure describes how to create a separate debug file for executable `a.out`.

▼ How to Create a Separate Debug File

1. **Create a separate debug file named `a.out.debug` containing the debugging information.**

```
objcopy --only-keep-debug a.out a.out.debug
```

2. **Strip the debugging information from `a.out`.**

```
objcopy --strip-debug a.out
```

3. **Establish the link between the two files.**

```
objcopy --add-gnu-debuglink=a.out.debug a.out
```

On Oracle Solaris platforms, use the `gobjcopy` command. On Linux platforms, use the `objcopy` command.

On a Linux platform, you can use the command `objcopy -help` to find out whether the `-add-gnu-debuglink` option is supported on the platform. You can replace the `-only-keep-debug` option of the `objcopy` command with the command `cp a.out a.out.debug` to make `a.out.debug` a fully executable file.

Ancillary Files (Oracle Solaris Only)

By default, load objects contain both allocable and non-allocable sections. Allocable sections are the sections that contain executable code and the data needed by that code at runtime. Non-allocable sections contain supplemental information that is not required to execute a file at runtime. These sections support the operation of debuggers and other observability tools. The non-allocable sections in an object are not loaded into memory at runtime by the operating system, and so, they have no impact on memory use or other aspects of runtime performance no matter their size.

For convenience, both allocable and non-allocable sections are normally maintained in the same file. However, there are situations in which it can be useful to separate these sections. Specifically, to support fine grained debugging of highly optimized code requires considerable debug data. In modern systems, the debugging data can easily be larger than the code it describes. The size of a 32-bit object is limited to 4GB. In very large 32-bit objects, the debug data can cause this limit to be exceeded and prevent the creation of the object.

Traditionally, load objects have been stripped of non-allocable sections in order to address these issues. Stripping is effective, but destroys data that might be needed later. The Oracle Solaris link-editor can instead write non-allocable sections to an ancillary file. This feature is enabled via the `-z ancillary` command line option.

```
% ld ... -z ancillary[=outfile] ...
/* Your file is separated into a.out and b.out, where
a.out: ELF 32-bit LSB executable 80386 Version 1 [FPU], dynamically linked, not
 stripped, ancillary object b.out
b.out: ELF 32-bit LSB ancillary 80386 Version 1, primary object a.out */
```

By default, the ancillary file is given the same name as the primary output object, with a `.anc` file extension. However, a different name can be provided by providing an *outfile* value to the `-z ancillary` option.

Note - The ELF definition of ancillary files provides for a single primary file, and an arbitrary number of ancillary file. At this time, the Oracle Solaris link-editor only produces a single ancillary file containing all non-allocable sections. This might change in the future.

When `-z ancillary` is specified, the link-editor does the following.

- All allocable sections are written to the primary file. In addition, all non-allocable sections containing one or more input sections that have the `SHF_SUNW_PRIMARY` section header flag set are written to the primary file.
- All remaining non-allocable sections are written to the ancillary file.
- Both output files receive full identical copies of the following well known non-allocable sections:

.shstrtab	Section name string table.
.symtab	The full non-dynamic symbol table.

.symtab	The symbol table extended index section associated with `.symtab`.
.strtab	The non-dynamic string table associated with `.symtab`.
.SUNW_ancillary	Contains the information required to identify the primary object, and all of the ancillary objects, and to identify the object being examined.

- The primary file and all ancillary files contain the same array of sections headers. Each section has the same section index in every file.
- Although the primary and ancillary files all define the same section headers, the data for most sections will be written to a single file as described above. If the data for a section is not present in a given file, the `SHF_SUNW_ABSENT` section header flag will be set, and `sh_size` field will be 0.

This organization makes it possible to acquire a full list of section headers, a complete symbol table, and a complete list of the primary and ancillary files, all from examining a single file.

`dbx` can then use these ancillary files just as `dbx` uses a separate debug file, by looking for ancillary files in your executable. Use the `-z ancillary` option when compiling as follows:

```
%CC -g -z ancillary=a.out demo.cpp //"a.out" contains the ancillary object
```

The primary load object, and all associated ancillary files, contain a .SUNW_ancillary section that allows all the load objects to be identified and related together.

For more information, see Chapter 2, “Link-Editor” in *Oracle Solaris 11.3 Linkers and Libraries Guide*.

Note - This feature is currently only available for Oracle Solaris 11.1.

Compressed Debug Sections (Oracle Solaris Only)

In addition to “Ancillary Files (Oracle Solaris Only)” on page 48, `dbx` also supports debugging of compressed debug sections. Compressed debug sections are useful in compacting debugging data that can sometimes be bigger than the code itself. This issue is sometimes referred to as the "DWARF bloat" problem.

Compressed debug sections are non-allocable sections that are reduced in size with the industry standard ZLIB compression library. Documentation for ZLIB can be found at `http://www.zlib.net/`. The debugger recognizes compressed debug sections within input objects and automatically decompresses these sections. This operation is transparent to the user of the debugger and requires no special action.

Use the `-z compress-debug-sections` option to enable the compression of debug sections in the output file.

```
$ cc -z compress-sections[=cmp-type] demo.cc
```

The following lists acceptable values for *cmp-type*:

zlib
: Compress candidates using ZLIB compression. The resulting output sections have the `SHF_COMPRESSED` section flag set to identify the use of compression. This is the default *cmd-type*, if one is not specified. For more information on the `SHF_COMPRESSED` section flag, see "Section Compression" in *Oracle Solaris 11.3 Linkers and Libraries Guide*.

zlib-gnu
: Compress all candidate sections using ZLIB compression, using the GNU section compression format. This format requires candidate sections to have a name that begins with `.debug`. The resulting output sections are renamed to start with `.zdebug` to identify the use of compression.

For more information on compressed debug sections and for an example of using compressed debug sections, see "Compressed Debug Sections" in *Oracle Solaris 11.3 Linkers and Libraries Guide*.

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 do the following:

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

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

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

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

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

For OpenMP programs, compiling with the `-xopenmp=noopt` option instructs the compiler not to apply any optimizations. However, the optimizer still processes the code in order to implement the OpenMP directives, so some of the problems described might occur in programs compiled with `-xopenmp=noopt`.

Parameters and Variables

Generally, symbolic information for parameters, local variables, and global variables is available for optimized programs. Type information about structs, unions, C++ classes, and the types and names of local variables, global variables, and parameters should be available.

Information about the location of parameters and local variables is sometimes missing for optimized code. If `dbx` cannot locate a value, it reports that it cannot. Sometimes the value might disappear temporarily, so try to single-step and print again.

The Oracle Developer Studio 12.2 compilers and later Oracle Developer Studio updates for SPARC based systems and x86 based systems provide the information for locating parameters and local variables. Newer versions of the GNU compilers also provide this information.

You can print global variables and assign values to them, although they might have inaccurate values if the final register-to-memory store has not happened yet.

Inlined Functions

`dbx` allows you to set breakpoints on inlined functions. Control stops at the first instruction from the inlined function in the caller. You can perform the same `dbx` operations (for example, `step`, `next`, and `list` commands) on inlined functions as you can perform on non-inlined functions.

The `where` command shows the call stack with the inlined function and the parameters if location information for the inlined parameters is available.

The `up` and `down` commands for moving up and down the call stack are also supported for inlined functions.

Local variables from the caller are not available in the inline frame.

Registers, if shown, are those from the caller's window.

Functions that the compilers might inline include the C++ inline functions, the C functions with the C99 inline keyword, and any other functions that the compiler deems profitable for performance.

The *Oracle Developer Studio 12.5: Performance Analyzer* contains information that might be helpful when debugging an optimized 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 restriction also applies to code that has had `strip -x` applied to it.

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

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

Completely Stripped Programs

`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. Some runtime checking works on stripped programs or load objects. For example, memory use checking works and access checking works with code stripped with `strip -x`, but not with code stripped with `strip`.

Quitting Debugging

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

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

```
(dbx) quit
```

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

Stopping a Process Execution

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

Detaching a Process From dbx

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

You can detach a process and leave it in a stopped state while you temporarily apply other `/proc`-based debugging tools that might be blocked when `dbx` has exclusive access. For more information, see "Detaching `dbx` From a Process" on page 89.

For more information, see "`detach` Command" on page 332.

Killing a Program Without Terminating the Session

The `dbx` `kill` command terminates debugging of the current process as well as killing the process. However, the `kill` command 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`.For more information, see "`kill` Command" on page 350.

Saving and Restoring a Debugging Run

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

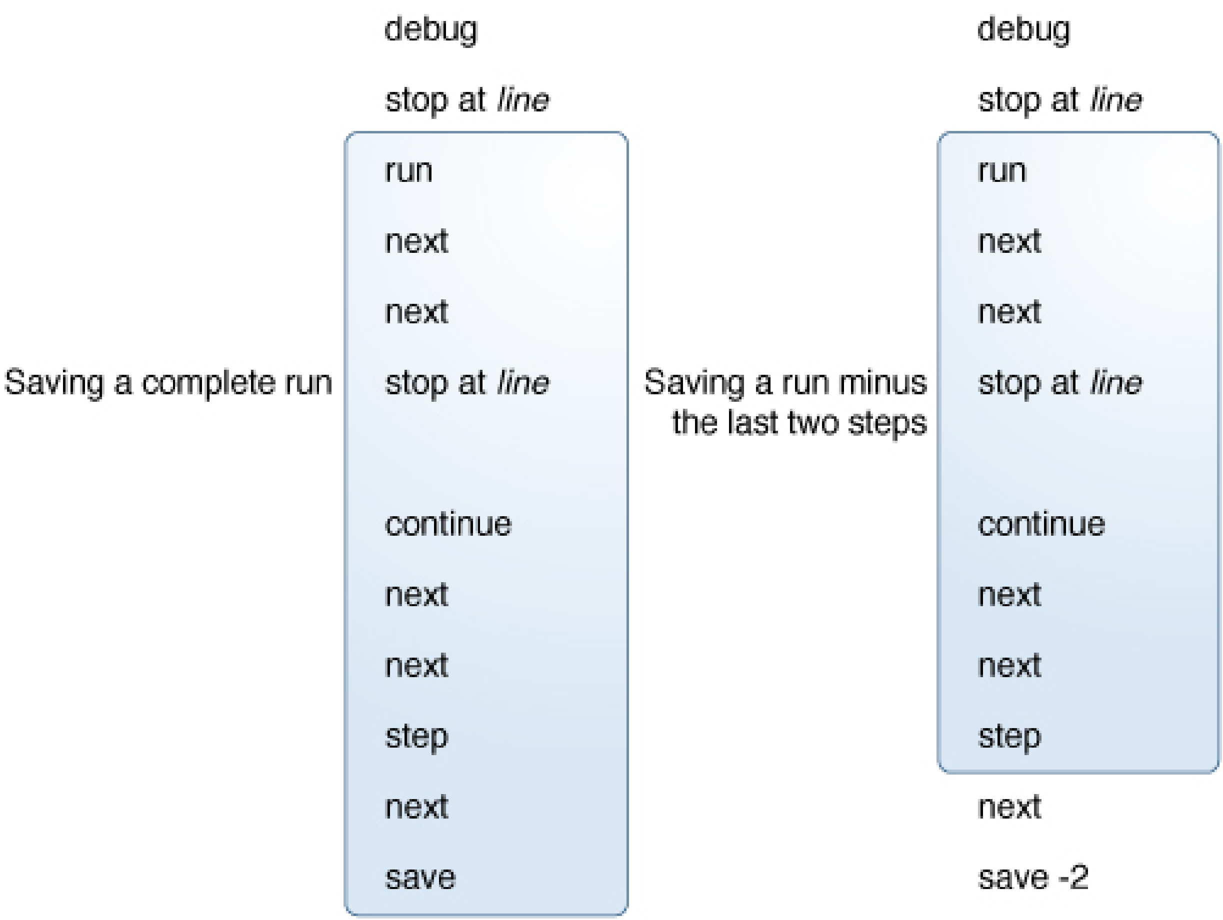
- `save [-`*number*`] [`*filename*`]`
- `restore [`*filename*`]`
- `replay [-`*number*`]`

Using the `save` Command

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

In addition to the list of debugging commands issued, the `save` command 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.



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

Note - By default, the `save` command writes information to a special file. If you want to save a debugging run to a file you can restore later, you can specify a file name with the `save` command. See "Saving a Series of Debugging Runs as Checkpoints" on page 57.

Issue the `save` command at the point at which you want to save an entire debugging.

```
(dbx) save
```

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

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

Saving a Series of Debugging Runs as Checkpoints

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

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

To save a debugging run to a file other than the default, include the file name:

```
(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 its save file. When you restore a run, `dbx` first resets the internal state to what it was at the start of the run, then reissues each of the debugging commands in the saved run.

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

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

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

To restore a saved debugging run

```
(dbx) restore
```

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

```
(dbx) restore filename
```

Saving and Restoring Using `replay`

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

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

```
(dbx) replay
```

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

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

♦ ♦ ♦ CHAPTER 3

Customizing dbx

This chapter describes the dbxenv 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 contains the following sections:

- "Using the dbx Initialization File" on page 59
- "Setting dbxenv Variables" on page 60
- "dbxenv Variables and the Korn Shell" on page 66

Using the dbx Initialization File

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

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

During startup, the search order is:

1. Installation directory (unless you specify the -S option to the dbx command) /*install--dir* \/lib/dbxrc. The default installation directory is /opt/solstudio12.4 on Oracle Solaris platforms and /opt/oracle/solstudio12.4 on Linux platforms. If your Oracle Developer Studio software is not installed in the default *install-dir*, dbx derives the path to the dbxrc file from the path to the dbx executable.
2. Current directory ./.dbxrc
3. Home directory $HOME/.dbxrc

Creating a `.dbxrc` File

To create a `.dbxrc` file that contains common customizations and aliases

```
(dbx) help .dbxrc>$HOME/.dbxrc
```

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

Initialization File Sample

The following example shows a sample `.dbxrc` file:

```
dbxenv input_case_sensitive false
catch FPE
```

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

- `dbxenv` is the command used to set `dbxenv` variables. For a complete list of `dbxenv` variables, see "Setting `dbxenv` Variables" on page 60.
- `input_case_sensitive` is the `dbxenv` variable that controls case sensitivity.
- `false` is the setting for `input_case_sensitive`.

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

Setting `dbxenv` Variables

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

To display the value of a specific variable:

(dbx) **dbxenv** *variable*

To show all variables and their values

(dbx) **dbxenv**

To set the value of a variable:

(dbx) **dbxenv** *variable value*

Table 1, "dbx Environment Variables," on page 61 consists all of the dbxenv variables that you can set.

TABLE 1 dbx Environment Variables

dbx Environment Variable	What the Variable Does
array_bounds_check on|off	If set to on, dbx checks the array bounds. Default: on.
c_array_op on | off	Allows array operations for C and C++. For example, if a and b are arrays, you can use the command print a+b. Default: off.
CLASSPATHX	Specifies to dbx a path for Java class files that are loaded by custom class loaders.
core_lo_pathmap on|off	Controls whether dbx uses pathmap settings to locate the correct libraries for a ismatchedcore file. Default: off.
debug_file_directory	Sets the global debug file directory. Default: /usr/lib/debug.
disassembler_versionautodetect|v8|v9|x86_32|x86_64	SPARC platform: Sets the version of dbx's built-in disassembler for SPARC V8 or V9. Default is autodetect, which sets the mode dynamically depending on the type of the machine a.out is running on. x86 platforms: Sets the version of dbx's built-in disassembler for x86_32 or x86_64. Default is autodetect, which sets the mode dynamically depending on the type of the machine a.out is running on.
event_safety on | off	Protects dbx against unsafe use of events. Default: on.
filter_max_length *num*	Sets the maximum length of sequences converted to arrays by pretty-printing filters to *num*.
fix_verbose on|off	Governs the printing of compilation line during a fix. Default: off.
follow_fork_inherit on|off	When following a child, determines whether to inherit breakpoints. Default: off.
follow_fork_mode parent|child|both|ask	Determines which process is followed after a fork; that is, when the current process executes a fork, vfork, or fork1. If set to parent, the process follows the parent. If set to child, it follows the child. If set to both, it follows the child, but the parent process remains active. If set to ask, you are asked which process to follow whenever a fork is detected. Default: parent.
follow_fork_mode_inner unset|parent|child|both	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. Default: unset.
input_case_sensitive autodetect| true|false	If set to autodetect, dbx automatically selects case sensitivity based on the language of the file: false for Fortran files; otherwise true. If true, case matters in variable and function names; otherwise, case is not significant. Default: autodetect.
JAVASRCPATH	Specifies the directories in which dbx should look for Java source files.

dbx Environment Variable	What the Variable Does
jdbx_mode java| jni| native	Stores the current dbx mode. Valid settings are java, jni, or native.
jvm_invocation	The jvm_invocation environment variable enables you to customize the way the JVM™ software is started. (The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java™ platform.) For more information, see "Customizing Startup of the JVM Software" on page 231.
language_mode autodetect|main|c| c++| fortran|fortran90	Governs the language used for parsing and evaluating expressions. ■ autodetect sets the expression language to the language of the current file. Useful if debugging programs with mixed languages (default). ■ main sets the expression language to the language of the main routine in the program. Useful if debugging homogeneous programs. ■ c, c++, c++, fortran, or fortran90 sets the expression language to the selected language.
macro_expand on | off	When set to on, globally enables macro expansion for selected expressions. Default: on.
macro_source none | compiler | skim | skim_unless_compiler	Governs where dbx gets macro information. See "Skimming Errors" on page 304 for more information. Default: skim_unless_compiler.
mt_resume_one on | off | auto	When set to off, all threads are resumed when stepping over calls with the next command in order to avoid deadlocks. When set to on, only the current thread is resumed when stepping over calls with the next command. When set to auto, behavior is the same as when set to off unless the program is a transaction management application and you are stepping within a transaction, in which case only the current thread is resumed. Default: auto.
mt_scalable on|off	When enabled, dbx is more conservative in its resource usage and will be able to debug processes with upwards of 300 LWPs. However, this setting can result in significant slowdown. Default: off.
mt_sync_tracking on | off	Determines whether dbx enables tracking of sync objects when it starts a process. Default: off.
output_auto_flush on|off	Automatically calls fflush() after each call. Default: on
output_base 8|10|16|automatic	Default base for printing integer constants. Default: automatic (pointers in hexadecimal characters, all else in decimal).
output_class_prefix on | off	Used to cause a class member to be prefixed with one or more classnames when its value or declaration is printed. If set to on, it causes the class member to be prefixed. Default: on.
output_data_member_only on|off	Used to display data members only when printing definition of a class (dbx command whatis -t -a). When set to on, the -a option is the default (dbx command whatis -t will display data members only). Default: off.

dbx Environment Variable	What the Variable Does
output_dynamic_type on|off	When set to on, -d is the default for printing watches and displaying. Default: off.
output_inherited_members on|off	When set to on, -r is the default for printing, displaying, and inspecting. Default: off.
output_list_size *num*	Governs the default number of lines to print in the list command. Default: 10.
output_log_file_name *filename*	Name of the command log file. Default: /tmp/dbx.log.*unique-ID* .
output_max_object_size *number*	Sets maximum number of bytes for printing variable; if variable size larger than this number, specifying the -L flag is required. This dbxenv variable applies to commands print, display. and watch. Default: 4096.
output_max_string_length *number*	Sets *number* of characters printed for char *s. Default: 096.
output_no_literal on|off	When enabled, if the expression is a string (char *), print the address only, do not print the literal. Default: off.
output_pretty_print on|off	Sets -p as the default for printing watches and displaying. Default: on.
output_pretty_print_fallback on|off	By default, pretty-printing reverts to regular printing if problems occur. If you want to diagnose a pretty-printing problem, set this variable to off to prevent the fallback. Default: on.
output_pretty_print_mode call | filter | filter_unless_call	Determines which pretty-printing mechanism is used. If set to call. uses call-style pretty-printers. If set to filter, uses python-based pretty-printers. If set to filter_unless_call, uses call-style pretty-printers first.
output_short_file_name on|off	Displays short path names for files. Default: on.
overload_function on|off	For C++, if set to on, does automatic function overload resolution. Default: on.
overload_operator on|off	For C++, if set to on, does automatic operator overload resolution. Default: on.
pop_auto_destruct on|off	If set to on, automatically calls appropriate destructors for locals when popping a frame. Default: on.
proc_exclusive_attach on|off	If set to on, keeps dbx from attaching to a process if another tool is already attached. Caution: If more than one tool attaches to a process and tries to control it unexpected results can occur. Default: on.
rtc_auto_continue on|off	Logs errors to rtc_error_log_file_name and continues. Default: off.
rtc_auto_suppress on|off	If set to on, an RTC error at a given location is reported only once. Default: on.

dbx Environment Variable	What the Variable Does
rtc_biu_at_exit on|off|verbose	Used when memory use checking is on explicitly or because of check -all. If the value is on, a non-verbose memory use (blocks in use) report is produced at program exit. If the value is verbose, a verbose memory use report is produced at program exit. The value off causes no output. Default: on.
rtc_error_limit *number*	The number of RTC access errors to be reported. Default: 1000.
rtc_error_log_file_name *filename*	Name of file to which RTC errors are logged if rtc_auto_continue is set. Default:/tmp/dbx.errlog.
rtc_error_stack on|off	If set to on, stack traces show frames corresponding to RTC internal mechanisms. Default: off.
rtc_inherit on|off	If set to on, enables runtime checking on child processes that are executed from the debugged program and causes the LD_PRELOAD environment variable to be inherited. Default: off.
rtc_mel_at_exit on|off|verbose	Used when memory leak checking is on. If the value is on, a non-verbose memory leak report is produced at program exit. If the value is verbose, a verbose memory leak report is produced at program exit. The value off causes no output. Default: on.
run_autostart on|off	If set to on with no active program, step, next, stepi, and nexti implicitly run the program and stop at the language-dependent main routine. If set to on, cont implies run when necessary. Default: off.
run_iostdio|pty	Governs whether the user program's input/output is redirected to dbx's stdio or a specific pty. The pty is provided by run_pty. Default: stdio.
run_pty *ptyname*	Sets the name of the pty to use when run_io is set to pty. ptys are used by graphical user interface wrappers.
run_quick on|off	If set to on, no symbolic information is loaded. The symbolic information can be loaded on demand using prog -readsysms. Until then, dbx behaves as if the program being debugged is stripped. Default: off.
run_savetty on | off	Multiplexes TTY settings, process group, and keyboard settings (if -kbd was used on the command line) between dbx and the program being debugged. Useful when debugging editors and shells. Set to on if dbx gets SIGTTIN or SIGTTOU and pops back into the shell. Set to off to gain a slight speed advantage. The setting is irrelevant if dbx is attached to the program being debugged or is running in the Oracle Developer Studio IDE. Default: off.
run_setpgrp on | off	If set to on, when a program is run, setpgrp(2)is called right after the fork. Default: off.
scope_global_enums on | off	If set to on, enumerators are put in global scope and not in file scope. Set before debugging information is processed (~/.dbxrc). Default: off.
scope_look_aside on | off	If set to on, finds file static symbols, in scopes other than the current scope. Default: on.

dbx Environment Variable	What the Variable Does
session_log_file_name *filename*	Name of the file where dbx logs all commands and their output. Output is appended to the file. Default: " "(no session logging).
show_static_members	When set to on, -S is the default for printing, watches, and displaying. Default: on.
stack_find_source on | off	When set to on, dbx attempts to find and automatically make active the first stack frame with source when the program being debugged comes to a stop in a function that is not compiled with -g. Default: on.
stack_max_size *number*	Sets the default size for the where command. Default: 100.
stack_verbose on | off	Governs the printing of arguments and line information in where. Default: on.
step_abflow stop|ignore	When set to stop, dbx stops in longjmp(), siglongjmp(), and throw statements when single stepping. When set to ignore, dbx does not detect abnormal control flow changes for longjmp() and siglongjmp(). Default: stop.
step_events on |off	When set to on, allows breakpoints while using step and next commands to step through code. Default: off.
step_granularity statement | line	Controls granularity of source line-stepping. When set to statement the following code: `a(); b();` takes the two next commands to execute. When set to line, a single next command executes the code. The granularity of line is particularly useful when dealing with multi-line macros. Default: statement.
suppress_startup_message *number*	Sets the release level below which the startup message is not printed. Default: 3.01.
symbol_info_compression on|off	When set to on, reads debugging information for each include file only once. Default: on.
trace_speed *number*	Sets the speed of tracing execution. Value is the number of seconds to pause between steps. Default: 0.50.
track_process_cwd on|off	When set to on and the GUI is attached to a running process, the current working directory changes to the working directory of the running process. Default: off.
vdl_mode classic | lisp | xml	Value Description Language (VDL) is used to communicate data structures to the graphical user interface (GUI) for dbx. classic mode was used for the Sun WorkShop™ IDE. lisp mode is used by the IDE in Sun Studio and Oracle Developer Studio releases. xml mode is experimental and unsupported. Default: value is set by the GUI.

dbxenv Variables and the Korn Shell

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

♦ ♦ ♦ CHAPTER 4

Viewing and Navigating To Code

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

This chapter contains the following sections

- "Navigating To Code" on page 67
- "Types of Program Locations" on page 70
- "Program Scope" on page 70
- "Qualifying Symbols With Scope Resolution Operators" on page 72
- "Locating Symbols" on page 75
- "Viewing Variables, Members, Types, and Classes" on page 78
- "Debugging Information in Object Files and Executables" on page 81
- "Finding Source and Object Files" on page 86

Navigating To Code

Each time the program you are debugging stops, `dbx` prints the source line associated with the stop location. At each program stop, `dbx` resets the value of the current function to the function in which the program is stopped. Before the program starts running and when it is stopped, you can move to, or navigate through, functions and files elsewhere in the program. You can navigate to any function or file that is part of the program. Navigating sets the current scope (see "Program Scope" on page 70). It is useful for determining when and at what source line you want to set a `stop at` breakpoint.

Navigating To a File

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

```
(dbx) file filename
```

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

```
(dbx) file
```

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

```
(dbx) file filename ; list line-number
```

For more information, see "Setting a Breakpoint at a Line of Source Code" on page 98.

Navigating To Functions

You can use the func command to navigate to a function. Type the command func followed by the function name. For example:

```
(dbx) func adjust-speed
```

The func command by itself echoes the current function.

For more information, see "func Command" on page 343

Selecting From a List of C++ Ambiguous Function Names

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

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

Choosing Among Multiple Occurrences

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

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

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

Printing a Source Listing

Use the `list` command to print the source listing for a file or function. Once you navigate through a file, the `list` command prints *number* lines from the top. The default is 10 lines. Once you navigate through a function, the `list` command prints its lines.

For detailed information, see "`list` Command" on page 352.

Walking the Call Stack to Navigate To Code

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

For more information see "Walking the Stack and Returning Home" on page 114.

Types of Program Locations

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

- The current address, which is used and updated by the `dis` command and the `examine` command .
- The current source code line, which is used and updated by the `list` command This line number is reset by some commands that alter the visiting scope. For more information, see "Changing the Visiting Scope" on page 71.
- The current visiting scope, which is a compound variable described in "Visiting Scope" on page 71. The visiting scope is used during expression evaluation. It is updated by the `line` command, the `func` command, the `file` command,and the `list` command.

Program Scope

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

Variables That Reflect the Current Scope

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

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

These variables are only useful during a live process.

Visiting Scope

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

```
(dbx) print i
```

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

Components of the Visiting Scope

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

`$vscope`	Current visiting scope
`$vloadobj`	Current visiting load object
`$vfile`	Current visiting file
`$vlineno`	Current visiting line number
`$vclass`	Class to which `$vfunc` belongs
`$vfunc`	Current visiting function

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

Changing the Visiting Scope

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

- `func`

- `file`
- `up`
- `down`
- `frame` *number*
- `pop`
- `list` *procedure*

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

When you hit a breakpoint, `dbx` sets the visiting scope to the current location. If the `stack-find-source` environment variable set to `on`, `dbx` attempts to find and make active a stack frame that has source code.

When you use the `up` command, the `down` command, the `frame` command, or the `pop` command to change the current stack frame, `dbx` sets the visiting scope according to the program counter from the new stack frame.

The line number location used by the `list` command changes the visiting scope only if you use the `list` command. When the visiting scope is set, the line number location for the `list` command is set to the first line number of the visiting scope. When you subsequently use the `list` command, the current line number location for the `list` command is updated, but as long as you are listing lines in the current file, the visiting scope does not change. For example, the following command causes `dbx` to list the start of the source for `my-func` and change the visiting scope to `my-func`.

```
(dbx) list my-func
```

The following command causes `dbx` to list line 127 in the current source file and does not change the visiting scope.

```
(dbx) list 127
```

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

Qualifying Symbols With Scope Resolution Operators

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

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

In addition to qualifying file and function names when navigating through code, symbol name qualifying is also necessary for printing and displaying out-of-scope variables and expressions, and for displaying type and class declarations (using the `whatis` command).

This section covers the rules for all types of symbol name qualifying. The symbol qualifying rules are the same in all cases.

Backquote Operator

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

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

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

```
(dbx) func`filename`function-name
```

C++ Double-Colon Scope Resolution Operator

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

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

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

```
(dbx) func class::function-name (args)
```

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

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

Block Local Operator

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

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

```
(dbx) stop in `animate.o`change-glyph:230`item
```

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

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

```
i = 3
(dbx) which i
`a.out`t.c`main:10`i
(dbx) print `main:7`i
`a.out`t.c`main`i = 1
(dbx) print `main:8`i
`a.out`t.c`main:8`i = 2
(dbx) print `main:10`i
`a.out`t.c`main:10`i = 3
(dbx) print `main:14`i
`a.out`t.c`main:8`i = 2
(dbx) print `main:15`i
`a.out`t.c`main`i = 1
```

Linker Names

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

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

Locating Symbols

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

Printing a List of Occurrences of a Symbol

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

```
(dbx) whereis table
forward: `Blocks`block-draw.cc`table
```

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

The output includes the name of the loadable objects where the program defines *symbol*, as well as its entity type: class, function, or variable.

Because information from the `dbx` symbol table is read in as it is needed, the `whereis` command registers only occurrences of a symbol that are already loaded. As a debugging session gets longer, the list of occurrences can grow. For more information, see "Debugging Information in Object Files and Executables" on page 81.

Determining Which Symbol `dbx` Uses

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

```
(dbx) func
wedge::wedge(char*, int, int, const point&, load-bearing-block*)
(dbx) which draw
`block-draw.cc`wedge::draw(unsigned long)
```

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

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

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

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

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

Scope Resolution Search Path

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

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

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

Relaxing the Scope Lookup Rules

To relax the scope lookup rules for static symbols and C++ member functions, set the `dbxenv` variable `scope-look-aside` to `on`:

```
dbxenv scope-look-aside on
```

You can also use the “double backquote” prefix:

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

If the dbxenv variable scope-look-aside is set to on, dbx looks for the following:

- Static variables defined in other files if not found in current scope. Files from libraries in /usr/lib are not searched.
- C++ member functions without class qualification.
- Instantiations of C++ inline member functions in other files if a member function is not instantiated in current file.

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

Viewing Variables, Members, Types, and Classes

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

For more information, see "whatis Command" on page 411.

Looking Up Definitions of Variables, Members, and Functions

Use the whatis command to print out the declaration of an identifier:

(dbx) **whatis** *identifier*

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

For C++ programs, whatis lists function template instantiations. Template definitions are displayed with whatis -t See "Looking Up Definitions of Types and Classes" on page 79.

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

To print out the member function, you would type the following commands:

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

To print out the data member

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

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

```
(dbx) whatis the-table
class table *the-table;
.
```

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

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

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

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

Looking Up Definitions of Types and Classes

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

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

```
(dbx) whatis -t class-name
```

To view data members only, use the `whatis` command along with the -a option. This option only prints the list of data members for a specific class (and not its members' functions). It displays this information in the same order as the `-r` option, starting from the base class first.

```
(dbx) whatis -t -a class-name
```

To view members in base classes, the `whatis` command takes an `-r` option (for recursive). This displays the declaration of a specified class, as well as the members it inherits from the base classes.

```
(dbx) whatis -t -r  class-name
```

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

To see the root of a class' inherited members, the `whatis` command takes a `-u` option that displays the root of the type definition. Without the `-u` option, the `whatis` command will display the last value in the value history. This is similar to the `ptype` command used in `gdb`.

The following two examples us 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);
};
```

The following examples show the results when `whatis -r` is used on a child class to see members it inherits.

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

```
    char *block::name();
    int block::is-movable();
//  deleted several members from example protected:
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load-bearing-block *supported-by;
    Panel-item panel-item;
    /* from base class table::load-bearing-block */
public:
    load-bearing-block::load-bearing-block();
    load-bearing-block::load-bearing-block(char *name, int w, int h,
        const class point &pos, class load-bearing-block *blk);
    virtual int load-bearing-block::is-load-bearing();
    virtual class list *load-bearing-block::supported-blocks();
    void load-bearing-block::add-supported-block(class block &b);
    void load-bearing-block::remove-supported-block(class block &b);
    virtual void load-bearing-block::print-supported-blocks();
    virtual void load-bearing-block::clear-top();
    virtual void load-bearing-block::put-on(class block &object);
    class point load-bearing-block::get-space(class block &object);
    class point load-bearing-block::find-space(class block &object);
    class point load-bearing-block::make-space(class block &object);
protected:
    class list *support-for;
    /* from class table */
public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

Debugging Information in Object Files and Executables

For the best results, compile your source files with the `-g` option to make your program more debuggable. The `-g` option causes the compilers to record debugging information in stabs or DWARF format into the object files along with the code and data for the program.

`dbx` parses and loads debugging information for each object file (module) on demand when the information is needed. You can use the `module` command to ask `dbx` to load debug information for any specific module, or for all modules. See also “Finding Source and Object Files” on page 86.

Object File Loading

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

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

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

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

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

Object files can be created with DWARF using the -xs option. For more information, see "Index DWARF (-xs[={yes|no}])" on page 84.

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

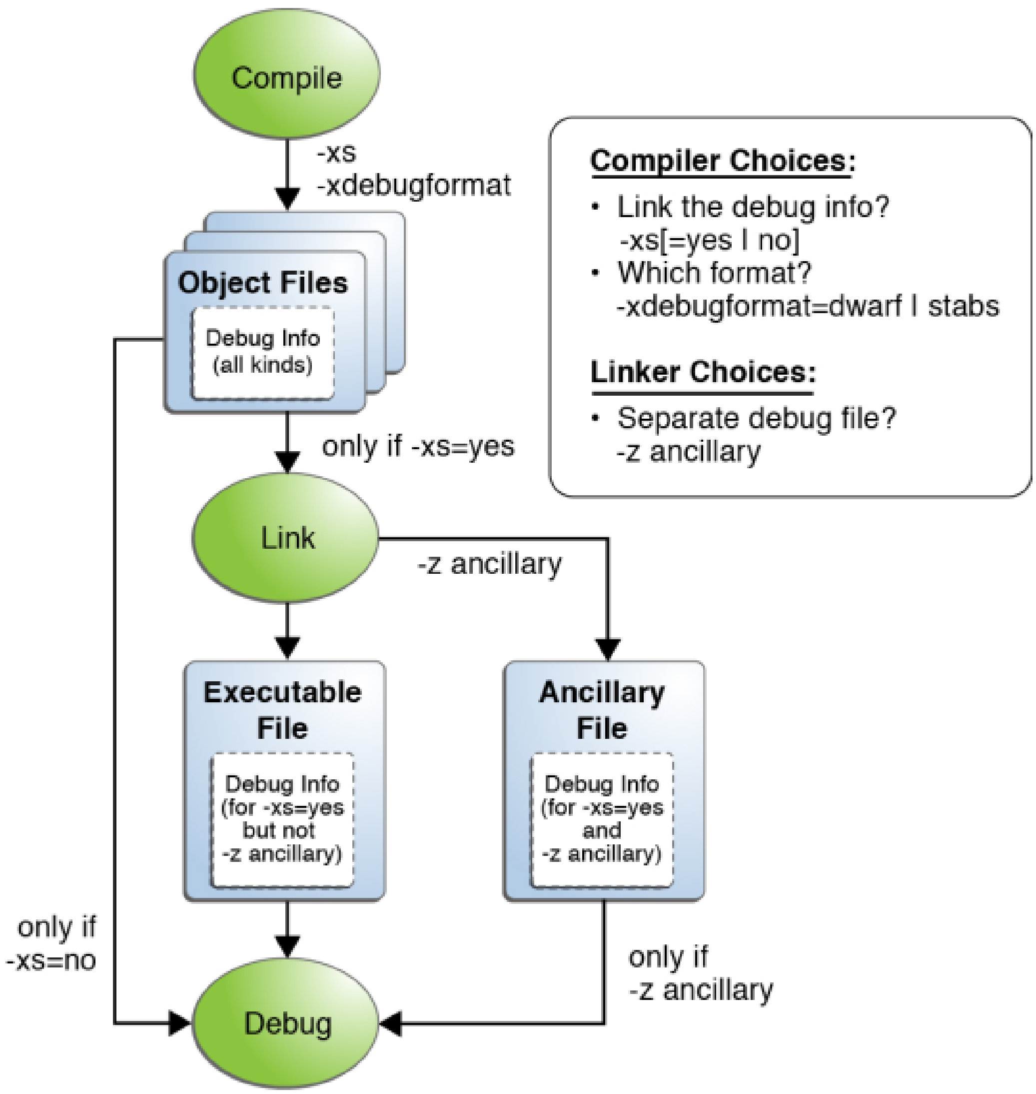
For more information about the stabs index, see the Stabs Interface guide found with the path *install-dir*/solarisstudio12.4/READMEs/stabs.pdf.

Compiler and Linker Options to Support Debugging

Compiler and linker options give users more freedom to generate and use debug information. Compilers generate an Index for DWARF, similar to index stabs The index is always present and results in faster dbx start-up time, as well as other improvements when debugging with DWARF.

The following is a diagram of the different kinds and locations of debug information, specifically highlighting where the debug data resides:

FIGURE 1 Flow of Debug Information



Index DWARF (-xs[={yes|no}])

DWARF by default is loaded into the executable file. The new index makes it possible to leave the DWARF in the object files with the `-xs=no` option. This results in a smaller executable size and a faster link. The object files must be retained in order to debug. This is similar to how stabs works.

Separate Debug File (-z ancillary[=*outfile*])

The Oracle Solaris 11.1 linker can send debug information to a separate ancillary file while building the executable. A separate debug file is useful for environments where all the debug information must be moved, installed, or archived. An executable can be run independently, but can also be debugged by people with a copy of its separate debug file.

`dbx` continues to support the use of the GNU utility `objcopy` to extract debug information into a separate file, but using the Oracle Solaris linker has the following advantages over `objcopy`:

- The separate debug file is produced as a by-product of the link
- A program which was too large to be linked as one file links as two files

For more information, see "Ancillary Files (Oracle Solaris Only)" on page 48.

Minimizing Debug Information

The `-g1` compiler option is intended for minimal debuggability of deployed applications. Compiling your application with this option produces the file and line number, as well as simple parameter information that is considered crucial during postmortem debugging. For more information, see the compiler man pages and the compiler user guides.

Listing Debugging Information for Modules

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

To read in debugging information for a module:

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

To read in debugging information for all modules:

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

where:

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

To print the name of the current module, type:

```
(dbx) module
```

Listing Modules

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

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

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

To list the names of all program modules regardless of whether they contain debugging information:

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

To list all program modules that contain debugging information:

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

where:

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

Finding Source and Object Files

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

Under the stabs format used by `dbx` in Sun Studio 11 and earlier releases, debugging information in `dbx` sometimes uses object files to load additional debugging information. Source files are used when `dbx` displays source code.

Symbolic information, including paths to source files, is contained within the executable file. When `dbx` needs to display source lines, it reads as much symbolic information as necessary to locate the source file, and read and display the lines from it.

The symbolic information includes the full path name of a source file, but when you type `dbx` commands, you typically use only the basename of a file. For example:

```
stop at test.cc:34
```

`dbx` searches for a matching file in the symbolic information.

If you have removed source files, `dbx` cannot show you source lines from those files, but you can display stack traces, print variable values, and even determine the source line you are on.

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

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

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

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

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

The mapping of `/tmp-mnt` to `/` exists by default.

♦ ♦ ♦ CHAPTER 5

Controlling Program Execution

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

This chapter contains the following sections:

- "Running a Program" on page 87
- "Attaching `dbx` to a Running Process" on page 88
- "Detaching `dbx` From a Process" on page 89
- "Stepping Through a Program" on page 90
- "Using Ctrl+C to Stop a Process" on page 94
- "Event Management" on page 94

Running a Program

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

You can set breakpoints in the program before running it.

Note - When debugging an application that is a mixture of Java™ code and C JNI (Java Native Interface) code or C++ JNI code, you might want to set breakpoints in code that has not yet been loaded. For more information, see "Setting Breakpoints in Native (JNI) Code" on page 231.

Use the `run` command to start program execution.

You can optionally add command-line arguments and redirection of input and output, using < for input and > or >> for output. Using >> will append contents to the existing output file.

(dbx) **run** [*arguments*][< *input-file*] [> *output-file*]

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

Note - Output from the run command overwrites an existing file even if you have set noclobber for the shell in which you are running dbx, unless you used >>, in which case, the command appends to the existing file.

The run command without arguments 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 might need to debug a program that is already running. You would attach to a running process in the following situations:

- You want to debug a running server, and you do not want to stop or kill it.
- You want to debug a running program that has a graphical user interface, and you do not 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 process ID number as an argument to the dbx debug command.

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

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

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

(dbx) **debug** *program-name process-ID*

or

(dbx) **attach** *process-ID*

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

For more information, see "`debug` Command" on page 329 and "`attach` Command" on page 309.

If `dbx` is not running, start `dbx` by typing:

```
% dbx program-name process-id
```

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

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

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

If the process to which you are attaching `dbx` is stopped due to a `SIGSTOP` signal, `SIGTSTOP` signal, `SIGTTIN` signal, or `SIGTTOUT` signal, the attach succeeds with a message like the following:

```
dbx76: warning: Process is stopped due to signal SIGSTOP
```

The process is inspectable, but to resume it you need to send it a `SIGCONT` signal with the `cont` command:.

```
(dbx) cont -sig cont
```

You can use runtime checking on an attached process with certain exceptions. See "Using Runtime Checking on an Attached Process" on page 156.

Detaching dbx From a Process

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

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

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

For more information, see "detach Command" on page 332.

Stepping Through a Program

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

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

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

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

For more information on the next and step commands, see "next Command" on page 363 and "step Command" on page 384.

Controlling Single Stepping Behavior

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

```
(dbx) next n
```

or

```
(dbx) step n
```

The `step_granularity` dbxenv variable determines the unit by which the `step` command and `next` command step through your code. The unit can be either `statement` or `line`.

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

The `step_abflow` environment variable controls whether `dbx` stops when it detects that an abnormal control flow change is about to happen. This type of control flow change can be caused by a call to `siglongjmp()` or `longjmp()` or an exception throw.

For more information, see "Setting `dbxenv` Variables" on page 60.

Stepping Into a Specific or Last Function

To step into a function called from the current source code line, use the `step to` command.

```
(dbx) step to function
```

To step into the last function called:

```
(dbx) step to
```

For the following two examples, using `step to` by itself will step into `foo`:

```
foo(bar(baz(4)));
```

```
baz()->bar()-> foo()
```

Continuing Execution of a Program

To continue a program after it has hit a breakpoint or some event, use the `cont` command.

```
(dbx) cont
```

A variant, `cont at` *line-number*, enables you to specify a line other than the current program location line at which to resume program execution. This option enables you to skip over one or more lines of code that you know are causing problems, without having to recompile.

To continue a program at a specified line, type:

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

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

Using the `cont at` *line-number* `command` with the `assign` command, you can avoid executing a line of code that contains a call to a function that might be incorrectly computing the value of some variable. To quickly adjust incorrectly computed values, use the `assign` command to give the variable a correct value. Use `cont at` *line-number* to skip the line that contains the function call that would have computed the value incorrectly.

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

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

For more information, see the following:

- "Setting a Breakpoint at a Line of Source Code" on page 98
- "Setting Breakpoints in Member Functions of Different Classes" on page 100
- "Setting Breakpoints in All Member Functions of a Class" on page 100
- "Setting Multiple Breakpoints in Nonmember Functions" on page 101
- "`when` Command" on page 413

Calling a Function

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

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

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

While the parameters are optional, you must type the parentheses after the function name. For example:

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

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

A C++ virtual function can be called like any other function using the `print` command or `call` command , or any other command that executes a function call.

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

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

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

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

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

Call Safety

Making calls into the process you are debugging, either by using the `call` command or by printing expressions that contain calls, has the potential for causing severe non-obvious disruptions. For example:

- A call might go into an infinite loop, which you can interrupt, or cause a segmentation fault. In many cases, you can use a `pop -c` command to return to the site of the call.
- When you make a call in a multithreaded application, all threads are resumed in order to avoid deadlocks, so you might see side-effects on threads other than the one on which you made the call.

- Calls used in breakpoint conditionals might confuse event management (see "Resuming Execution" on page 174).

Some calls made by dbx are performed safely. If a problem, typically a segmentation fault, is encountered instead of the usual `Stopped with call to ...`, dbx does one of the following actions:

- Ignores any `stop` commands including those caused by detection of memory access errors
- Automatically issues a `pop -c` command to return to the site of the call
- Proceeds with execution

dbx uses safe calls for the following situations:

- Calls occurring within an expression printed by the `display` command. A failed call appears as: `ic0->get _data() = <call failed>`

  To diagnose such a failure, try printing the expression with the `print` command.
- Calls to the `db_pretty_print()` function, except when the `print -p` command is used.
- Calls used in event condition expressions. A condition with a failed call evaluates to false.
- Calls made to invoke destructors during a `pop` command.
- All internal calls.

Using Ctrl+C to Stop a Process

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

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

Event Management

An event is an occurrence in the debugging process that causes dbx to be notified. Event management refers to the capability of dbx to perform actions when events take place in the program being debugged. When an event occurs, dbx enables you to stop a process, execute arbitrary commands, or print information. The simplest example of an event is a breakpoint. Examples of other events are faults, signals, system calls, calls to `dlopen()`, and data changes (see "Qualifying Breakpoints With Caller Filters" on page 106).

For more in-depth information about event management, such as event handlers, event safety, creating events, event specifications, and other event management topics, see Appendix B, "Event Management".

♦ ♦ ♦ CHAPTER 6

Setting Breakpoints and Traces

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

This chapter describes how to set, clear, and list breakpoints and traces. For complete information on the event specifications you can use in setting breakpoints and traces, see "Setting Event Specifications" on page 274.

This chapter contains the following sections:

- "Setting Breakpoints" on page 97
- "Setting Filters on Breakpoints" on page 105
- "Tracing Execution" on page 108
- "Executing `dbx` Commands at a Line" on page 109
- "Setting Breakpoints in Dynamically Loaded Libraries" on page 109
- "Listing and Deleting Breakpoints" on page 110
- "Enabling and Disabling Breakpoints" on page 111
- "Efficiency Considerations" on page 111

Setting Breakpoints

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

- `stop` – 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` – If the program arrives at a breakpoint created with a `when` command, the program halts and `dbx` executes one or more debugging commands, then the program continues unless one of the executed commands is `stop`.

- `trace` – A trace displays information about an event in your program, such as a change in the value of a variable. Although a trace's behavior is different from that of a breakpoint, traces and breakpoints share similar event handlers. If a program arrives at a breakpoint created with a `trace` command, the program halts and an event-specific `trace` information line is emitted, then the program continues.

The `stop`, `when`, and `trace` commands all take as an argument an event specification, which describes the event on which the breakpoint is based. Event specifications are discussed in detail in "Setting Event Specifications" on page 274.

To set machine-level breakpoints, use the `stopi`, `wheni`, and `tracei` commands. For more information, see Chapter 17, "Debugging at the Machine-Instruction Level".

Note - When debugging an application that is a mixture of Java™ code and C JNI (Java Native Interface) code or C++ JNI code, you might want to set breakpoints in code that has not yet been loaded. For information on setting breakpoints on such code, see "Setting Breakpoints in Native (JNI) Code" on page 231.

Setting a Breakpoint at a Line of Source Code

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

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

For example:

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

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

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

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

```
(dbx) stop at 13
```

For more information on specifying at an location event, see "at Event Specification" on page 275.

Setting a Breakpoint in a Function

You can set a breakpoint in a function by using the `stop in` command.

(dbx) **stop in** *function*

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

`dbx` should be able to determine which function you are referring to except in the following situations:

- You reference an overloaded function by name only.
- You reference a function with a leading `` ` ``.
- You reference a function by its linker name (mangled name in C++). In this case, `dbx` accepts the name if you prefix it with a `#`. For more information, see "Linker Names" on page 75.

Consider the following set of declarations:

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

To stop at a non-member function, the following command sets a breakpoint at the global `foo (int)`:

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

To set a breakpoint at the member function:

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

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

```
stop in foo
```

If you type:

```
stop in `bar
```

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

Note - If a member name is unique, for example unique_member, using stop in unique_member is sufficient. If a member name is not unique, you can use the stop in command and answer the overload menu to specify which member you mean.

For more information about specifying an in-function event, see "in Event Specification" on page 275.

Setting Multiple Breakpoints in C++ Programs

You can check for problems related to calls to members of different classes, calls to any members of a given class, or calls to overloaded top-level functions. You can use the keywords, 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 class-specific variants of a particular member function (same member function name, different classes), use stop inmember.

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

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

For more information about specifying an inmember or inmethod event, see "inmember Event Specification" on page 276.

Setting Breakpoints in All Member Functions of a Class

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

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

The following command sets a breakpoint in all member functions defined in the class `shape`:

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

The following command sets a breakpoint in all member functions defined in the class, and also in functions inherited from the class:

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

For more information on specifying an `inclass` event, see "`inclass` Event Specification" on page 277 and "`stop` Command" on page 387.

Due to the large number of breakpoints that might be inserted by `stop inclass` and other breakpoint selections, be sure to set the `dbxenv` variable `step_events` to `on` to speed up the `step` and `next` commands. For more information,see "Efficiency Considerations" on page 111.

Setting Multiple Breakpoints in Nonmember Functions

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

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

```
(dbx) stop infunction sort
```

For more information on specifying an `infunction` event, see "`infunction` Event Specification" on page 276.

Setting Breakpoints in Objects

Set an in-object breakpoint to check the operations applied to a specific object instance.

Use in-object breakpoints to stop program execution when any method is called on a specific object instance. For example, the following code will only cause a stop when `f1->printit()` is called:

```
Foo *f1 = new Foo();
Foo *f2 = new Foo();
 f1->printit();
 f2->printit();
```

```
(dbx) stop inobject f1
```

The address stored in `f1` identifies the objects you put a breakpoint on. This implies that this breakpoint can only be created after the object in `f1` has been instantiated.

By default, an in-object breakpoint suspends program execution in all nonstatic member functions of the object's class, including inherited ones. To restrict breakpoints only to the objects class, specify the `-norecurse` option.

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

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

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

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

For more information on specifying an `inobject` event, see "`inobject` Event Specification" on page 277 and "`stop` Command" on page 387

Setting Data Change Breakpoints (Watchpoints)

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

Stopping Execution When an Address Is Accessed

Use the `stop access` command to stop execution when a memory address has been accessed:

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

mode specifies how the memory was accessed. The valid mode options are:

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

`x` The memory has been executed.

mode can also contain either of the following:

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

`b` Stops the process before the access.

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

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

In the following example, the command will stop execution after any of the four bytes after the memory address `0x4762` has been read.

```
(dbx) stop access r 0x4762, 4
```

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

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

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

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

For more information on specifying an access event, see “`access` Event Specification” on page 277 and “`stop` Command” on page 387.

Stopping Execution When Variables Change

Use the `stop change` command to stop program execution if the value of a specified variable has changed:

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

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

- `dbx` stops the program at the line *after* the line that caused a change in the value of the specified variable.

- If *variable* is local to a function, the variable is considered to have changed when the function is first entered and storage for *variable* is allocated. The same is true with respect to parameters.
- The command does not work with multithreaded applications.

For more information on specifying a change event, see "change Event Specification" on page 278 and "stop Command" on page 387.

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

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

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

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

Note - Watching data changes is faster using the `access` event than the `change` event. Instead of automatically single-stepping the program, the `access` event uses hardware or OS services that are much faster.

Stopping Execution on a Condition

Use the`stop cond` command to stop program execution if a conditional statement evaluates to true:

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

The program stops executing when the condition occurs.

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

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

For more information about specifying a condition event, see "cond Event Specification" on page 279 and "stop Command" on page 387.

Setting Filters on Breakpoints

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

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

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

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

With a location-based breakpoint like `in` or `at`, the scope for parsing the condition is that of the breakpoint location. Otherwise, the scope of the condition is the scope at the time of entry, not at the time of the event. You might have to use the backquote operator (see "Backquote Operator" on page 73) to specify the scope precisely.

The following two filters are not the same:

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

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

Qualifying Breakpoints With Conditional Filters

To set a breakpoint that includes a 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.

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

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

You can use the `-if` option with function calls:

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

The following is an example of using a conditional filter with a watchpoint:

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

Qualifying Breakpoints With Caller Filters

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

Consider this example:

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

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

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

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

Consider a simple example, in which you have code like the following:

```
44:     if(open(filename, ...) == -1)
45:          return "Error";
```

You can stop on a specific failure, for example `ENOENT` of `open()` with the following command:

```
(dbx) stop at 45 -if errno == 2
```

Filters can be convenient when you are placing a data change breakpoint on a local variable. In the following example, the current scope is in function `foo()`, while `index`, the variable of interest, is in function `bar()`.

```
(dbx) stop access w &bar`index -in bar
```

``bar`index`` ensures that the `index` variable in function `bar()` is picked up, instead of the `index` variable in function `foo` or a global variable named `index`.

`-in bar` implies the following:

- The breakpoint is automatically enabled when function `bar()` is entered.
- The breakpoint remains enabled for the duration of `bar()` including any functions it calls.

- The breakpoint is automatically disabled upon return from `bar()`.

The stack location corresponding to `index` might be reused by some other local variable of some other function. `-in` ensures that the breakpoint is triggered only when ``bar`index`` is accessed.

Filters and Multithreading

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

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

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

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

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

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

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

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

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

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

For detailed information, see “Event Specification Modifiers” on page 289.

Tracing Execution

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

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

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

Setting a Trace

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

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

For the complete syntax of the `trace` command, see "`trace` Command" on page 400.

The information a trace provides depends on the type of *event* associated with it (see "Setting Event Specifications" on page 274).

Controlling the Speed of a Trace

Often trace output goes by too quickly. The `dbxenv` variable `trace_speed` enables you to control the delay after each trace is printed. The default delay is `0.5` seconds.

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

`dbxenv` **`trace_speed`** *number*

Directing Trace Output to a File

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

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

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

Executing dbx Commands at a Line

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

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

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

For the complete syntax of the `when` command, see "when Command" on page 413. For detailed information on event modifiers, see "Event Specification Modifiers" on page 289.

Setting Breakpoints in Dynamically Loaded Libraries

`dbx` interacts with the following types of shared libraries:

- Libraries that are implicitly loaded at the beginning of a program's execution.
- Libraries that are explicitly (dynamically) loaded using `dlopen(2)`. The names in such libraries are known only after the library has been loaded during a run, so you cannot place breakpoints in them after starting a debugging session with a `debug` or `attach` command.
- Filter libraries that are explicitly loaded using `dlopen(2)`. The names in such libraries are known only after the library has been loaded and the first function in it has been called.

You can set breakpoints in explicitly (dynamically) loaded libraries in two ways:

- If you have a library, for example `mylibrary.so`, which contains a function `myfunc()`, you could preload the library's symbol tale into `dbx` and set a breakpoint on the function as follows:

  ```
  (dbx) loadobject -load fullpathto/mylibrary.so
  ```

```
(dbx) stop in myfunc
```

- A much easier way is to run your program under `dbx` to completion. `dbx` records and remembers all shared libraries that are loaded with `dlopen(2)`, even if they are closed with `dlclose()`. So after the first run of the program, you will be able to set breakpoints successfully.

```
(dbx) run
execution completed, exit code is 0
(dbx) loadobject -list
u   myprogram (primary)
u   /lib/libc.so.1
u p /platform/sun4u-us3/lib/libc_psr.so.1
u   fullpathto/mylibrary.so
(dbx) stop in myfunc
```

Listing and Deleting Breakpoints

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

Listing Breakpoints and Traces

To display a list of all active breakpoints, use the `status` command to display ID numbers in parentheses or brackets, which can then be used by other commands. If ID numbers are in brackets, these breakpoints are disabled. Additionally, an asterisk (*) might appear before the parentheses or brackets to indicate if the program is stopped due to that event.

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

Deleting Specific Breakpoints Using Handler ID Numbers

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

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

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

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

```
(dbx) delete all
```

For more information, see "`delete` Command" on page 332.

Enabling and Disabling Breakpoints

Each event management command (`stop`, `trace`, `when`) that you use to set a breakpoint creates an event handler. Each of these commands returns a number known as the handler ID (*hid*). You can use the handler ID as an argument to the `handler` command to enable or disable the breakpoint. For example:

```
(dbx) handler -disable 5
(dbx) handler -enable 5
```

For more information, see "Event Handlers" on page 271.

Efficiency Considerations

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

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

In the case of the `step` command and `next` command, 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 the `step` command and `next` command slows down considerably. Use the `dbx step_events` environment variable to control whether breakpoints are taken out and reinserted after each `step` command or `next` command.

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

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

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

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

◆ ◆ ◆ CHAPTER 7

Using the Call Stack

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

In a multithreaded program, these commands operate on the call stack of the current thread. See "`thread` Command" on page 396 for information on how to change the current thread.

This chapter contains the following sections:

- "Finding Your Place on the Stack" on page 113
- "Walking the Stack and Returning Home" on page 114
- "Moving Up and Down the Stack" on page 114
- "Popping the Call Stack" on page 115
- "Hiding Stack Frames" on page 116
- "Displaying and Reading a Stack Trace" on page 116

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

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

Finding Your Place on the Stack

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

`where [-f] [-h] [-l] [-q] [-v]` *number-ID*

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

`where [-f] [-q] [-v] [` *thread_id* `]` *number-ID*

The `where` command is also useful for learning about the state of a program that has crashed and produced a core file. When this occurs, you can load the core file into `dbx` (see "Debugging a Core File" on page 40).

For more information, see "`where` Command" on page 415.

Walking the Stack and Returning Home

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

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

Moving Up and Down the Stack

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

Moving Up the Stack

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

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

If you do not specify *number*, the default is one level. For more information, see "`up` Command" on page 409.

Moving Down the Stack

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

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

If you do not specify *number*, the default is one level. For more information, see "down Command" on page 336.

Moving to a Specific Frame

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

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

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

You can also move to a specific frame using the `pop` command.

Popping the Call Stack

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

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

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

```
pop 5
```

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

```
pop -f 5
```

For more information, see "pop Command" on page 370.

Hiding Stack Frames

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

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

hide [*regular-expression* **]**

The regular-expression matches either the function name or the name of the load object and uses `sh` or `ksh` syntax for file matching.

Use the `unhide` command to delete all stack frame filters.

```
unhide 0
```

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

`unhide [` *number* `|` *regular-expression* `]`

Displaying and Reading a Stack Trace

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

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

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

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

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

In the following example, the program has halted with a segmentation fault. The cause is most likely the null argument to strlen() in SPARC input register $i0.

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

CD Library Statistics:

 Titles:         1

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

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

For more examples of stack traces, see “Looking at the Call Stack” on page 34 and “Tracing Calls” on page 214.

CHAPTER 8

Evaluating and Displaying Data

This chapter describes two types of data checking: evaluating data and displaying data.

This chapter contains the following sections:

- "Evaluating Variables and Expressions" on page 119
- "Assigning a Value to a Variable" on page 123
- "Evaluating Arrays" on page 123
- "Using Pretty-Printing" on page 128

Evaluating Variables and Expressions

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

Verifying Which Variable `dbx` Uses

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

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

For information on the commands, see "`which` Command" on page 418 and "`whereis` Command" on page 417.

Variables Outside the Scope of the Current Function

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

- Qualify the name of the function. See "Qualifying Symbols With Scope Resolution Operators" on page 72. For example:

  ```
  (dbx) print 'item
  ```
- Visit the function by changing the current function. See "Navigating To Code" on page 67.

Printing the Value of a Variable, Expression, or Identifier

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

Use the print command to evaluate a variable or expression in native code:

`print` *expression*

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

For more information, see "`print` Command" on page 370.

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

Printing C++ Pointers

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

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

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

These commands also take a `-d` or `+d` option that toggles the default behavior of the `dbxenv` variable `output_dynamic_type`.

Using the `-d` flag or setting the `dbxenv` variable `output_dynamic_type` to `on` when no process is running generates a `program is not active` error message. As when you are debugging a core file, accessing dynamic information is not possible when there is no process. An `illegal cast on class pointers` error message is generated if you try to find a dynamic type through a virtual inheritance. Casting from a virtual base class to a derived class is not legal in C++.

Evaluating Unnamed Arguments in C++ Programs

You can define functions in C++ with unnamed arguments. For example:

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

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

```
_ARG%n
```

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

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

For more information, see "`whatis` Command" on page 411.

To evaluate (or display) an unnamed function argument:

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

Dereferencing Pointers

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

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

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

Monitoring Expressions

Monitoring the value of an expression each time the program stops is an effective technique for learning how and when a particular expression or variable changes. The `display` command instructs `dbx` to monitor one or more specified expressions or variables. Monitoring continues until you stop it with the `undisplay` command. The `watch` command evaluates and prints expressions at every stopping point in the scope current at that stop point.

Use the `display` 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.

For more information, see "`display` Command" on page 334.

Use the `watch` command to watch the value of the expression at every stopping point:

watch *expression*, ...

For more information, see "`watch` Command" on page 410.

Stop the Display (Undisplaying)

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

To stop the display of a particular variable or expression:

`undisplay` *expression*

To stop the display of all currently monitored variables:

`undisplay 0`

For more information, see "`undisplay` Command" on page 406.

Assigning a Value to a Variable

Use the `assign` command to assign a value to a variable:

`assign` *variable* **`=`** *expression*

Evaluating Arrays

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

The following example is a sample Fortran array:

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

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

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

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

- Array slicing – Prints any rectangular, *n*-dimensional box of a multidimensional array.
- Array striding – Prints certain elements only, in a fixed pattern, within the specified slice, which might be an entire array.

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

Array Slicing

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

Array Slicing Syntax for C and C++

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

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

where:

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

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

For example:

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

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

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

Array Slicing Syntax for Fortran

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

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

where:

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

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

For example:

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

```
(6) 6

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

To specify rows and columns:

```
demo% f95 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
    1         INTEGER*4 a(3,4), col, row
    2         DO row = 1,3
    3             DO col = 1,4
    4               a(row,col) = (row*10) + col
    5             END DO
    6         END DO
    7         DO row = 1, 3
    8                WRITE(*,'(4I3)') (a(row,col),col=1,4)
    9        END DO
    10        END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
    7          DO row = 1, 3
```

To print row 3:

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

To print column 4:

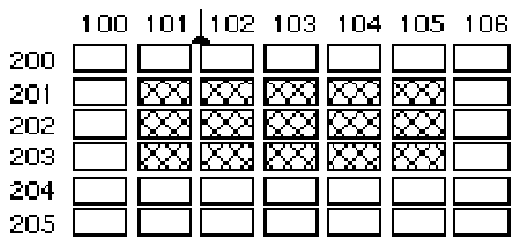
```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(1:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

Using Slices

The following example is a two-dimensional, rectangular slice of a C++ array, with the default stride of 1 omitted.

```
print arr[201..203][101..105]
```

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



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

Using Strides

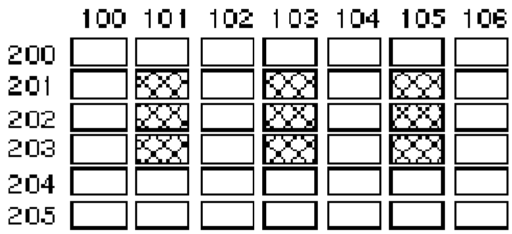
When you instruct `print` to *stride* across a slice of an array, `dbx` evaluates certain elements in the slice only, skipping over a fixed number of elements between each one it evaluates.

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

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

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

As shown in the diagram, a stride of 2 prints every second element, skipping every other element.



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

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

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

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

```
print arr
```

The following command prints every third element in the second dimension of a two-dimensional array:

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

Using Pretty-Printing

Pretty-printing enables your program to provide its own rendition of an expression's value through a function call. `dbx` supports two mechanisms for pretty-printing : call-based pretty-printing and filter-based pretty-printing. The older, call-based mechanism works by calling functions defined in the debuggee, which conform to a certain pattern. The current version of dbx now supports Python-based filters, allowing the user to create filters that transform values from one form to another.

- "Call-Based Pretty-Printing" on page 129
- "Python Pretty-Print Filters (Oracle Solaris)" on page 131

dbx determines which mechanism to use with the dbxenv variable output_pretty_print_mode. If set to call, call-based pretty-printers are sought. If set to filter, Python-based pretty-printers are sought. If set to filter_unless_call, call-based pretty-printers take precedence over filters.

Pretty-printers, regardless of type,are invoked if you specify the -p option to the print command, rprint command, display command, or watch command. For more about invocation of pretty-printers, see "Invoking Pretty-Printing" on page 129.

If the dbxenv variable output_pretty_print is set to on, -p is passed to the print command, rprint command, or display command as the default. Use +p to override this behavior. In addition, output_pretty_print controls pretty-printing for IDE locals, balloon evaluation, and watches.

Invoking Pretty-Printing

Pretty-print functions are invoked for the following:

- print -p or if the dbxenv variable output_pretty_print is set to on.
- display -p or if the dbxenv variable output_pretty_print is set to on.
- watch -p or if the dbxenv variable output_pretty_print is set to on.
- Balloon evaluation if the dbxenv variable output_pretty_print is set to on.
- Local variable if the dbxenv variable output_pretty_print is set to on.

Pretty-print functions are not invoked for the following:

- $[]. $[] is intended to be used in scripts, therefore the scripts should be predictable.
- The dump command. dump uses the same simplified formatting as the where command, which might be converted to use pretty-printing in later releases. This limitation does not apply to the Local Variables window in the IDE.

Call-Based Pretty-Printing

Call-based pretty-printing enables an application to provide its own rendition of an expression's value through a function call. If you specify the -p option to the print command, rprint command, display command, or watch command, dbx searches for a function of the form const chars *db_pretty_print(const T *, int flags, const char *fmt) and calls it, substituting the returned value for print or display.

The value passed in the flags argument of the function is bit-wise or one of the following:

FVERBOSE	0x1	Not currently implemented, always set
FDYNAMIC	0x2	-d
FRECURSE	0x4	-r
FFORMAT	0x8	-f (if set, fmt is the format part)
FLITERAL	0x10	-l

The db_pretty_print() function can be either a static member function or a standalone function.

When pretty-printing, also consider the following information:

- "Possible Failures" on page 131
- "Pretty-Printing Function Considerations" on page 130
- Prior to dbx version 8.0 pretty-printing was based on a ksh implementation of prettyprint. While this ksh function (and its pre-defined alias pp) still exist, most of the semantics have been reimplemented inside dbx with the following results:
  - For the IDE, watches, local variables, and balloon evaluation can use pretty-printing.
  - In the print command, display command, and watch command, the -p option uses the native route.
  - Better scalability, especially now that pretty-printing can be called quite often, especially for watches and local variables.
  - Better opportunity to derive addresses from expressions.
  - Better error recovery.
- Nested values will not be pretty-printed because dbx does not have the infrastructure to calculate the addresses of nested fields.
- The dbxenv variable output_pretty_print_fallback is set by default to on, which means that dbx will fall back on regular formatting if pretty-printing fails. If pretty-printing fails while i the environment variable is set to off, dbx will still issue an error message.

Pretty-Printing Function Considerations

When using the pretty-printing functions, you will need to consider the following:

- For const/volatile unqualified types, in general, functions such as db_pretty_print(int *, ...()) and db_pretty_print(const int *, ...)() are considered distinct. The overload resolution approach of dbx is discerning but non-enforcing:
  - Discerning – If you have defined variables declared both int and const int, each will be routed to the appropriate function.

  - Non-enforcing – If you have only one `int` or `const int` variable defined, they will match with both functions. This behavior is not specific to pretty-printing and applies to any calls.
- The `db_pretty_print()` function must be compiled with the `-g` option because `dbx` needs access to parameter signatures.
- The `db_pretty_print()` function is allowed to return `NULL`.
- The main pointer passed to the `db_pretty_print()` function is guaranteed to be non-`NULL` but otherwise it might still point to a poorly initialized object.
- The `db_pretty_print()` function needs to be disambiguated based on the type of its first parameter. In C, you can overload functions by writing them as file statics.

Possible Failures

Pretty-printing might fail for one of these detectable and recoverable reasons:

- No pretty-print function found.
- The expression to be pretty-printed cannot have its address taken.
- The function call did not immediately return, which would imply a segmentation fault resulting when the pretty-print function is not robust when encountering bad objects. It could also imply a user breakpoint.
- The pretty-print function returned `NULL`.
- The pretty-print function returned a pointer that `dbx` fails to indirect through.
- A core file is being debugged.

For all cases except the function call not immediately returning, these failures are silent and `dbx` falls back on regular formatting. But if the `output_pretty_print_fallback` `dbxenv` variable is set to `off`, `dbx` will issue an error message if pretty-printing fails.

If you use the `print -p` command rather than setting the `dbxenv` variable `output_pretty_print` to `on`, `dbx` stops in the broken function to enable you to diagnose the cause of failure. You can then use the `pop -c` command to clean up the call.

Python Pretty-Print Filters (Oracle Solaris)

The pretty-printing filter feature enables you to write filters in Python which can transform a Value from one form to another. Python-based pretty-printers are only available on Oracle Solaris.

Note - Python pretty-print filters can only be used in C and C++ code, not Fortran.

Filters are built in for select classes in 4 implementations of the C++ Standard Template Library. The following table specifies the library name and the compiler option for that library:

Compiler option for Library	Library Name
`-library=Cstd` (default)	`libCstd.so.1`
`-library=stlport4`	`libstlport.so.1`
`-library=stdcxx4`	`libstdcxx4.so.4.**`
`-library=stdcpp` (default when using the `-std=c++11` option)	`libstdc++.so.6.*`

The following table specifies which classes the pretty-print filters can be used for in the C++ Standard Template Library and whether index and slice can be printed:

Classes	Index and Slice Availability	C++ Compatibility
`string`	No	Yes
`pair`	No	Yes
`vector`	Yes	Yes
`list`	Yes	Yes
`set`	Yes	Yes
`bitset`	Yes	Yes
`map`	Yes	Yes
`stack`	Yes	Yes
`priority queue`	Yes	Yes
`queue`	Yes	Yes
`multimap`	Yes	Yes
`tuple`	No	C++11 only
`unique_ptr`	No	C++11 only
`forward_list`	Yes	C++11 only
`unordered_map`	Yes	C++11 only
`unordered_multimap`	Yes	C++11 only
`unordered_set`	Yes	C++11 only
`unordered_multiset`	Yes	C++11 only
`array`	Yes	C++11 only

Classes	Index and Slice Availability	C++ Compatibility
initializer_list	Yes	C++11 only

EXAMPLE 1 Pretty-Printing with Filters

The following output is an example of printing a list using the `print` command in `dbx`:

```
(dbx) dbxenv output_pretty_print off
(dbx) print list10
list10 = {
   __buffer-size = 32U
   __buffer-list = {
     __data_ = 0x654a8
   }
   __free-list   = (nil)
   __next-avail  = 0x67334
   __last     = 0x67448
   __node     = 0x48830
   __length   = 10U
   }
```

The following is the same list printed in `dbx`, but using pretty-printing filters:

```
(dbx) print -p list10
list10 = (200, 201, 202, 203, 204, 205, 206, 207, 208, 209)

(dbx) print -p list10[5]
list10[5] = 205

(dbx) print -p list10[1..100:2]
list10[1..100:2] =
[1] = 202
[3] = 204
[5] = 206
[7] = 208
```

Using Python on Oracle Solaris

Python pretty-print filters and the `python` command is available only on Oracle Solaris. To start the built-in Python interpreter, type `python`. To evaluate your Python code, type `python` *python-code*. A nascent Python plugin API is available. However, its primary purpose is for the writing of pretty-printer filters which that get invoked as callbacks. Therefore the `python` command mainly serves testing and diagnostic purposes.

Python Pretty-Print API Documentation

To generate the Python pretty-print API documentation, use the `python-docs` command. This command is only available on Oracle Solaris.

♦ ♦ ♦ CHAPTER 9

Using Runtime Checking

Runtime checking (RTC) enables you to automatically detect runtime errors such as memory access errors and memory leak, in a native code application during the development phase. It also enables you to monitor memory usage.

The following topics are covered in this chapter:

- "Capabilities of Runtime Checking" on page 135
- "Using Runtime Checking" on page 137
- "Using Access Checking" on page 140
- "Using Memory Leak Checking" on page 143
- "Using Memory Use Checking" on page 148
- "Suppressing Errors" on page 149
- "Using Runtime Checking on a Child Process" on page 152
- "Using Runtime Checking on an Attached Process" on page 156
- "Using Fix and Continue With Runtime Checking" on page 157
- "Runtime Checking Application Programming Interface" on page 159
- "Using Runtime Checking in Batch Mode" on page 160
- "Troubleshooting Tips" on page 161
- "Runtime Checking Limitations" on page 162
- "Runtime Checking Errors" on page 165

Capabilities of Runtime Checking

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

Note - You cannot use runtime checking on Java code.

Runtime checking provides the following capabilities:

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

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

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

When to Use Runtime Checking

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

Runtime Checking Requirements

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

- Dynamic linking with `libc`.
- Use of the standard `libc malloc`, `free`, and `realloc` functions or allocators based on those functions. Runtime checking provides an application programming interface (API) to handle other allocators. See "Runtime Checking Application Programming Interface" on page 159.
- Programs that are not fully stripped; programs stripped with `strip -x` are acceptable.

For information about the limitations of runtime checking, see "Runtime Checking Limitations" on page 162.

Using Runtime Checking

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

Enabling Memory Use and Memory Leak Checking

Use the following command to enable memory use and memory leak checking:

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

When memory use checking or memory leak checking is enabled, the `showblock` command shows the details about the heap block at a given address. The details include the location of the block's allocation and its size. For more information, see "`showblock` Command" on page 382.

Enabling Memory Access Checking

Use the following command to enablememory access checking only:

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

Enabling All Runtime Checking

Use the following command to enable memory leak, memory use, and memory access checking:

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

For more information, see "`check` Command" on page 313.

Disabling Runtime Checking

Use the following command to disable runtime checking entirely:

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

For detailed information, see "`uncheck` Command" on page 405.

Running Your Program

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

The program runs normally but slowly because each memory access is checked for validity just before it occurs. If `dbx` detects invalid access, it displays the type and location of the error. Control returns to you unless the `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. For detailed information, see "`cont` Command" on page 325.

If the `rtc_auto_continue` `dbxenv` variable is set to `on`, runtime checking continues to find errors and keeps running automatically. It redirects errors to the file named by the `dbxenv` variable `rtc_error_log_file_name.` The default log file name is `/tmp/dbx.errlog.`*unique-ID*.

You can limit the reporting of runtime checking errors using the `suppress` command. For detailed information, see "`suppress` Command" on page 393.

The following simple example shows how to enable memory access and memory use checking for a program called `hello.c`.

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

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

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

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

 Total      Num of  Leaked     Allocation call stack
 Size       Blocks  Block
                    Address
```

```
==========  ====== ==========  ======================================
        32       1    0x21aa8  memory_leak < main

Possible leaks report  (possible leaks:       0  total size:      0 bytes)

Checking for memory use...
Blocks in use report   (blocks in use:        2  total size:      44 bytes

 Total     % of Num of  Avg    Allocation call stack
 Size      All  Blocks  Size
========== ==== ====== ======  ======================================
        32  72%      1     32  memory_use < main
        12  27%      1     12  memory_use < main

execution completed, exit code is 0
```

The function `access_error()` reads variable `j` before it is initialized. Runtime checking reports this access error as a Read from uninitialized (`rui`) error.

The function `memory_leak()` does not free the variable `local` before it returns. When `memory_leak()` returns, this variable goes out of scope and the block allocated at line 20 becomes a leak.

The program uses the global variables `hello1` and `hello2,` which are in scope all the time. They both point to dynamically allocated memory, which is reported as Blocks in use (`biu`).

Using Access Checking

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

Programs might incorrectly read or write memory in a variety of ways, which are called memory access errors. For example, the program might reference a block of memory that has been deallocated through a `free()`call for a heap block. Or a function might return a pointer to a local variable and when that pointer is accessed, an error would result. Access errors might result in wild pointers in the program and can cause incorrect program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to find.

Runtime checking maintains a table that tracks the state of each block of memory being used by the program. Runtime checking checks each memory operation against the state of the block of memory it involves and then determines whether the operation is valid. The possible memory states are:

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

Using runtime checking to find memory access errors is not unlike using a compiler to find syntax errors in your program. In both cases, a list of errors is produced, with each error message giving the cause of the error and the location in the program where the error occurred. In both cases, you should fix the errors in your program starting at the top of the error list and working your way down. One error can cause other errors in a chain reaction. The first error in the chain is, therefore, the "first cause" and fixing that error might also fix some subsequent errors.

For example, a read from an uninitialized section of memory can create an incorrect pointer, which when dereferenced can cause another invalid read or write, which can in turn lead to yet another error.

Understanding the Memory Access Error Report

Runtime checking provides the following information for memory access errors:

`type`	Type of error.
`access`	Type of access attempted (read or write).
`size`	Size of attempted access.
`address`	Address of attempted access.
`size`	Size of leaked block.
`detail`	More detailed information about address. For example, if the address is in the vicinity of the stack, then its position relative to the current stack pointer is given. If the address 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 the address 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, runtime checking provides function and address.

The following example shows a typical access error.

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xefffee50
    which is 96 bytes above the current stack pointer
Variable is ”j’
Current function is rui
   12           i = j;
```

Memory Access Errors

Runtime checking detects the following memory access errors:

- `rui` – See “Read From Uninitialized Memory (`rui`) Error” on page 167
- `rua` – See “Read From Unallocated Memory (`rua`) Error” on page 167
- `rob` – See “Read From Array Out-of-Bounds (`rob`) Error” on page 167
- `wua` – See “Write to Unallocated Memory (`wua`) Error” on page 168
- `wro` – See “Write to Read-Only Memory (`wro`) Error” on page 168
- `wob` – See“Write to Array Out-of-Bounds Memory (`wob`) Error” on page 168
- `mar` – See “Misaligned Read (`mar`) Error” on page 166
- `maw` – See “Misaligned Write (`maw`) Error” on page 166
- `duf` – See “Duplicate Free (`duf`) Error” on page 165
- `baf` – See “Bad Free (`baf`) Error” on page 165
- `maf` – See “Misaligned Free (`maf`) Error” on page 166
- `oom` – See “Out of Memory (`oom`) Error” on page 167

Note - On SPARC platforms, runtime checking does not perform array bounds checking and therefore does not report array bound violations as access errors.

Using Memory Leak Checking

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because no pointers are pointing to the blocks, programs cannot reference them, much less free them. Runtime checking finds and reports such blocks.

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

Typically, memory leaks occur because allocated memory is not freed and you lose a pointer to the allocated block. Here are some examples of memory leaks:

```
void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

    return; /* no free of s. Once foo returns, there is no     */
            /* pointer pointing to the malloc’ed block,         */
            /* so that block is leaked.                         */
}
```

A leak can result from incorrect use of an API.

```
void
printcwd()
{

    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));

    return; /* libc function getcwd() returns a pointer to     */
            /* malloc’ed area when the first argument is NULL, */
            /* program should remember to free this. In this   */
            /* case the block is not freed and results in leak.*/
}
```

You can avoid memory leaks by always freeing memory when it is no longer needed and paying close attention to library functions that return allocated memory. If you use such functions, remember to free up the memory appropriately.

Sometimes the term *memory leak* is used to refer to any block that has not been freed. This definition is much less useful, because it is a common programming practice not to free memory if the program will terminate shortly. Runtime checking does not report a block as a leak if the program still retains one or more pointers to it.

Detecting Memory Leak Errors

Runtime checking detects the following memory leak errors:

- `mel` – See "Memory Leak (`mel`) Error" on page 170
- `air` – See "Address in Register (`air`) Error" on page 169
- `aib` – See "Address in Block (`aib`) Error" on page 169

Note - Runtime checking only finds leaks of `malloc` memory. If your program does not use `malloc`, runtime checking cannot find memory leaks.

Possible Leaks

Runtime checking can report a "possible" leak in two cases. The first case is when no pointers are found pointing to the beginning of the block but a pointer is found pointing to the *interior* of the block. This case is reported as an Address in block (`aib`) error. A stray pointer pointing into the block 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. This case would not be a memory leak. Because runtime checking cannot distinguish between these two cases, it reports both of them as possible leaks, letting you determine which are real memory leaks.

The second type of possible leak occurs when no pointers to a block are found in the data space but a pointer is found in a register. This case is reported as an Address in register (`air`) error. If the register points to the block accidentally or if it is an old copy of a memory pointer that has since been lost, then this is a real leak. However, the compiler can optimize references and place the only pointer to a block in a register without ever writing the pointer to memory. Such a case would not be a real leak. Hence, if the program has been optimized and the report was the result of the `showleaks` command, it is likely not to be a real leak. In all other cases, it is likely to be a real leak. For more information, see "`showleaks` Command" on page 382.

Note - Runtime leak checking requires the use of the standard `libc malloc/free/realloc` functions or allocators based on those functions. For other allocators, see "Runtime Checking Application Programming Interface" on page 159.

Checking for Leaks

If memory leak checking is enabled, 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. The following example is a typical memory leak error message:

```
Memory leak (mel):
Found leaked block of size 6 at address 0x21718
At time of allocation, the call stack was:
    [1] foo() at line 63 in test.c
    [2] main() at line 47 in test.c
```

A UNIX program has a `main` procedure (called `MAIN` in f77) that is the top-level user function for the program. Normally, a program terminates either by calling `exit(3)` or by returning from `main`. In the latter case, all variables local to `main` go out of scope after the return, and any heap blocks they pointed to are reported as leaks unless global variables point to those same blocks.

A common programming practice is not to free heap blocks allocated to local variables in `main`, because the program is about to terminate and return from `main` without calling `exit()`. To prevent runtime checking from reporting such blocks as memory leaks, stop the program just before `main` returns by setting a breakpoint on the last executable source line in `main`. When the program halts there, use the `showleaks` command to report all the true leaks, omitting the leaks that would result merely from variables in `main` going out of scope.

For more information, see "`showleaks` Command" on page 382.

Understanding the Memory Leak Report

With leak checking enabled, you receive an automatic leak report when the program exits. All possible leaks are reported, provided the program has not been killed using the `kill` command. The level of detail in the report is controlled by the `dbxenv` variable `rtc_mel_at_exit`. By default, a non-verbose leak report is generated.

Reports are sorted according to the combined size of the leaks. Actual memory leaks are reported first, followed by possible leaks. The verbose report contains detailed stack trace information, including line numbers and source files whenever they are available.

Both reports include the following information for memory leak errors:

Size	Size of leaked block

Location	Location where leaked block was allocated
Address	Address of leaked block
Stack	Call stack at time of allocation, as constrained by `check -frames`

The following 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
```

The following example shows a typical verbose leak report.

```
Actual leaks report    (actual leaks:         3  total size:    2427 bytes)

Memory Leak (mel):
Found 2 leaked blocks with total size 1852 bytes
At time of each allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] true_leak() at line 224 in "leaks.c"

Memory Leak (mel):
Found leaked block of size 575 bytes at address 0x22150
At time of allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] main() at line 87 in "leaks.c"

Possible leaks report  (possible leaks:       1  total size:       8 bytes)

Possible memory leak -- address in block (aib):
Found leaked block of size 8 bytes at address 0x219b0
At time of allocation, the call stack was:
    [1] in_block() at line 177 in "leaks.c"
    [2] main() at line 100 in "leaks.c"
```

Generating a Leak Report

You can ask for a leak report at any time using the `showleaks` command, which reports new memory leaks since the last `showleaks` command. For more information, see "`showleaks` Command" on page 382.

Combining Leaks

Because the number of individual leaks can be very large, runtime checking automatically combines leaks allocated at the same place into a single combined leak report. The decision to combine leaks, or report them individually, is controlled by the `number-of-frames-to-match` parameter specified by the `-match` *m* option on a `check -leaks` or the `-m` option of the `showleaks` command. If the call stack at the time of allocation for two or more leaks matches to *m* frames to the exact program counter level, these leaks are reported in a single combined leak report.

Consider the following three call sequences:

Block 1	Block 2	Block 3
`[1] malloc`	`[1] malloc`	`[1] malloc`
`[2] d() at 0x20000`	`[2] d() at 0x20000`	`[2] d() at 0x20000`
`[3] c() at 0x30000`	`[3] c() at 0x30000`	`[3] c() at 0x31000`
`[4] b() at 0x40000`	`[4] b() at 0x41000`	`[4] b() at 0x40000`
`[5] a() at 0x50000`	`[5] a() at 0x50000`	`[5] a() at 0x50000`

If all of these blocks lead to memory leaks, the value of *m* determines whether the leaks are reported as separate leaks or as one repeated leak. If *m* is 2, Blocks 1 and 2 are reported as one repeated leak because the 2 stack frames above `malloc()` are common to both call sequences. Block 3 will be reported as a separate leak because the trace for `c()` does not match the other blocks. For *m* greater than 2, runtime checking reports all leaks as separate leaks. The `malloc` is not shown on the leak report.

In general, the smaller the value of *m*, the fewer individual leak reports and the more combined leak reports are generated. The greater the value of *m*, the fewer combined leak reports and the more individual leak reports are generated.

Fixing Memory Leaks

Once you have obtained a memory leak report, follow these guidelines for fixing the memory leaks:

- Most importantly, determine where the leak is. The leak report tells you the allocation trace of the leaked block, the place where the leaked block was allocated.
- You can then look at the execution flow of your program and see how the block was used. If it is obvious where the pointer was lost, the job is easy; otherwise you can use `showleaks` to narrow your leak window. By default, the `showleaks` command lists the new leaks created only since the last `showleaks` command. You can run `showleaks` repeatedly while stepping through your program to narrow the window where the block was leaked.

For more information, see "`showleaks` Command" on page 382.

Using Memory Use Checking

Memory use checking enables you to see all the heap memory in use. You can use this information to get a sense of where memory is allocated in your program or which program sections are using the most dynamic memory. This information can also be useful in reducing the dynamic memory consumption of your program and might help in performance tuning.

Memory use checking is useful during performance tuning or to control virtual memory use. When the program exits, a memory use report can be generated. Memory usage information can also be obtained at any time during program execution with the `showmemuse` command, which causes memory usage to be displayed. For information, see "`showmemuse` Command" on page 383.

Enabling memory use checking also enables 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. The level of detail in the memory use report is controlled by the `dbxenv` variable `rtc_biu_at_exit`.

The following example shows 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

Runtime checking includes a powerful error suppression facility that provides great flexibility in limiting the number and types of errors reported. If an error occurs that you have suppressed, then no report is given, and the program continues as if no error had occurred.

You can suppress errors using the `suppress` command.

You can undo error suppression using the `unsuppress` command.

Suppression is persistent across `run` commands within the same debug session, but not across `debug` commands.

Types of Suppression

This section describes thetypes of suppression that 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, or the main program
File	Applies to all functions in a particular file
Function	Applies to a particular function
Line	Applies to a particular source line
Address	Applies to a particular instruction at an address

Suppression of Last Error

By default, runtime checking suppresses the most recent error to prevent repeated reports of the same error. This setting is controlled by the `dbx` variable `rtc_auto_suppress`. When `rtc_auto_suppress` is set to `on` (the default), a particular access error at a particular location is reported only the first time it is encountered and suppressed thereafter. This setting is useful, for example, for preventing multiple copies of the same error report when an error occurs in a loop that is executed many times.

Limiting the Number of Errors Reported

You can use the `dbxenv` variable `rtc_error_limit` to limit the number of errors that will be reported. The error limit is used separately for access errors and leak errors. For example, if the error limit is set to 5, then a maximum of five access errors and five memory leaks are shown in both the leak report at the end of the run and for each `showleaks` command you issue. The default is 1000.

Suppressing Error Examples

In the following examples, `main.cc` is a file name, `foo` and `bar` are functions, and `a.out` is the name of an executable.

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

```
suppress mel in foo
```

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

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

Suppress read from uninitialized in all functions in `a.out`:

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

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

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

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

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

Suppress reporting of all errors in function `bar`:

```
suppress all in bar
```

For more information, see "`suppress` Command" on page 393.

Default Suppressions

To detect all errors, runtime checking does not require the program be compiled using the `-g` option (symbolic). However, symbolic information is sometimes needed to guarantee the correctness of certain errors, mostly `rui` errors. For this reason, certain errors (`rui` for `a.out` and `rui`, `aib`, and `air` for shared libraries) are suppressed by default if no symbolic information is available. This behavior can be changed using the `-d` option of the `suppress` command and `unsuppress` command.

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

```
unsuppress -d rui
```

For more information, see "unsuppress Command" on page 408.

Using Suppression to Manage Errors

For the initial run on a large program, the large number of errors might be overwhelming. Consider taking a phased approach. You can do so using the suppress command to reduce the reported errors to a manageable number, fixing just those errors, and repeating the cycle. This enables you to suppress fewer and fewer errors with each iteration.

For example, you could focus on a few error types at one time. The most common error types typically encountered are rui, rua, and wua, usually in that order. rui errors are less serious, although they can cause more serious errors to happen later. Often a program might still work correctly with these errors. rua and wua errors are more serious because they are accesses to or from invalid memory addresses and always indicate a coding error.

You can start by suppressing rui and rua errors. After fixing all the wua errors that occur, run the program again, suppressing only rui errors. After fixing all the rua errors that occur, run the program again with no errors suppressed. Fix all the rui errors. Lastly, run the program a final time to ensure that no errors are left.

If you want to suppress the last reported error, use suppress -last.

Using Runtime Checking on a Child Process

To use runtime checking on a child process, you must have the dbxenv variable rtc_inherit set to on. By default, it is set to off.

dbx supports runtime checking of a child process if runtime checking is enabled for the parent and the dbxenv variable follow_fork_mode is set to child.

When a fork happens, dbx automatically performs runtime checking on the child. If the program calls exec(), the runtime checking settings of the program calling exec() are passed on to the program.

At any given time, only one process can be under runtime checking control, as shown in the following example.

```
% cat -n program1.c
     1 #include <sys/types.h>
     2 #include <unistd.h>
     3 #include <stdio.h>
     4
     5 int
     6 main()
     7 {
     8      pid_t child_pid;
     9      int parent_i, parent_j;
    10
    11      parent_i = parent_j;
    12
    13      child_pid = fork();
    14
    15      if (child_pid == -1) {
    16          printf("parent: Fork failed\n");
    17          return 1;
    18      } else if (child_pid == 0) {
    19          int child_i, child_j;
    20
    21          printf("child: In child\n");
    22          child_i = child_j;
    23          if (execl("./program2", NULL) == -1) {
    24              printf("child: exec of program2 failed\n");
    25              exit(1);
    26          }
    27      } else {
    28          printf("parent: child's pid = %d\n", child_pid);
    29      }
    30      return 0;
    31 }
```

```
 % cat -n program2.c
     1
     2 #include <stdio.h>
     3
     4 main()
     5 {
     6      int program2_i, program2_j;
     7
     8      printf ("program2: pid = %d\n", getpid());
     9      program2_i = program2_j;
    10
    11      malloc(8);
    12
```

```
    13      return 0;
    14 }
%

 % cc -g -o program1 program1.c
 % cc -g -o program2 program2.c
 % dbx -C program1
 Reading symbolic information for program1
 Reading symbolic information for rtld /usr/lib/ld.so.1
 Reading symbolic information for librtc.so
 Reading symbolic information for libc.so.1
 Reading symbolic information for libdl.so.1
 Reading symbolic information for libc_psr.so.1
 (dbx) check -all
 access checking - ON
 memuse checking - ON
 (dbx) dbxenv rtc_inherit on
 (dbx) dbxenv follow_fork_mode child
 (dbx) run
 Running: program1
 (process id 3885)
 Enabling Error Checking... done
```

RTC reports first error in the parent, `program1`

```
 Read from uninitialized (rui):
 Attempting to read 4 bytes at address 0xeffff110
     which is 104 bytes above the current stack pointer
 Variable is ’parent_j’
 Current function is main
   11       parent_i = parent_j;
(dbx) cont
 dbx: warning: Fork occurred; error checking disabled in parent
 detaching from process 3885
 Attached to process 3886
```

Because `follow_fork_mode` *is set to* `child`*, when the fork occurs error checking is switched from the parent*
to the child process

```
 stopped in _fork at 0xef6b6040
 0xef6b6040: _fork+0x0008:    bgeu    _fork+0x30
 Current function is main
    13       child_pid = fork();
 parent: child’s pid = 3886
 (dbx) cont
 child: In child
 Read from uninitialized (rui):
 Attempting to read 4 bytes at address 0xeffff108
     which is 96 bytes above the current stack pointer
```

RTC reports an error in the child

```
 Variable is ’child_j’
```

```
 Current function is main
    22       child_i = child_j;
 (dbx) cont
 dbx: process 3886 about to exec("./program2")
 dbx: program "./program2" just exec'ed
 dbx: to go back to the original program use "debug $oprog"
 Reading symbolic information for program2
 Skipping ld.so.1, already read
 Skipping librtc.so, already read
 Skipping libc.so.1, already read
 Skipping libdl.so.1, already read
 Skipping libc_psr.so.1, already read
```

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

```
 Enabling Error Checking... done
 stopped in main at line 8 in file "program2.c"
     8       printf ("program2: pid = %d\n", getpid());
(dbx) cont
 program2: pid = 3886
 Read from uninitialized (rui):
 Attempting to read 4 bytes at address 0xeffff13c
     which is 100 bytes above the current stack pointer
```

RTC reports an access error in the executed program, `program2`

```
 Variable is 'program2_j'
 Current function is main
     9       program2_i = program2_j;
 (dbx) cont
 Checking for memory leaks...
```

RTC prints a memory use and memory leak report for the process that exited while under RTC control, `program2`

```
Actual leaks report   (actual leaks:      1  total size:   8
 bytes)

 Total      Num of  Leaked     Allocation call stack
 Size       Blocks  Block
                    Address
==========  ====== ========== ====================================
        8       1    0x20c50  main
 Possible leaks report  (possible leaks:   0  total size:   0
 bytes)

 execution completed, exit code is 0
```

Using Runtime Checking on an Attached Process

Runtime checking works on an attached process, with the exception that `rui` cannot be detected if the affected memory has already been allocated.

Attached Process on a System Running Oracle Solaris

On a system running the Oracle Solaris operating system, the process must have `rtcaudit.so` preloaded when it starts. If the process to which you are attaching is a 64-bit process, use the appropriate 64-bit `rtcaudit.so`, which is located at:

64-bit SPARC platforms: `/`*install-dir*`/lib/dbx/sparcv9/runtime/rtcaudit.so`

AMD64 platforms: `/`*install-dir*`/lib/dbx/amd64/runtime/rtcaudit.so`

32-bit platforms: `/`*install-dir*`/lib/dbx/runtime/rtcaudit.so`

To preload `rtcaudit.so`:

```
% setenv LD_AUDIT path-to-rtcaudit/rtcaudit.so
```

Set the `LD_AUDIT` environment variable to preload `rtcaudit.so` only when needed. Do not keep it loaded all the time. For example:

```
% setenv LD_AUDIT...
% start_your_application
% unsetenv LD_AUDIT
```

Once you attach to the process, you can enable runtime checking.

If the program you want to attach to is forked or executed from some other program, you must set `LD_AUDIT` for the main program, which will fork. The setting of `LD_AUDIT` is inherited across forks and execution. This solution might not work if a 32–bit program forks or executes a 64–bit program, or a 64–bit program forks or executes a 32–bit program.

The `LC_AUDIT` environment variable applies to both 32-bit programs and 64-bit programs, which makes it difficult to select the correct library for a 32-bit program that runs a 64-bit program, or a 64-bit program that runs a 32-bit program. Some versions of the Oracle Solaris OS support the `LD_AUDIT_32` environment variable and the `LD_AUDIT_64` environment variable, which affect only 32-bit programs and 64-bit programs, respectively. See the *Linker and Libraries Guide* for the version of Oracle Solaris you are running to determine if these variables are supported.

Attached Process on a System Running Linux

On a system running the Linux operating system, the process must have `librtc.so` preloaded when it starts. If the process to which you are attaching is a 64-bit process running on an AMD64 processor, use the appropriate 64-bit `librtc.so`, which is located at:

64-bit AMD64 platforms: `/`*install-dir*`/lib/dbx/amd64/runtime/librtc.so`

32-bit AMD64 platforms`/`*install-dir*`/lib/dbx/runtime/librtc.so`

To preload `librtc.so`:

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

Set the `LD_PRELOAD` environment variable to preload `librtc.so` only when needed. Do not keep it loaded all the time. For example:

```
% setenv LD_PRELOAD...
% start_your_application
% unsetenv LD_PRELOAD
```

Once you attach to the process, you can enable runtime checking.

If the program you want to attach to is forked or executed from some other program, you must set `LD_PRELOAD` for the main program, which will fork. The setting of `LD_PRELOAD` is inherited across forks and execution. This solution might not work if a 32-bit program forks or executes a 64-bit program, or a 64-bit program forks or executes a 32-bit program.

The `LC_PRELOAD` environment variable applies to both 32-bit programs and 64-bit programs, which makes it difficult to select the correct library for a 32-bit program that runs a 64-bit program, or a 64-bit program that runs a 32-bit program. Some versions of Linux support the `LD_PRELOAD_32` environment variable and the `LD_PRELOAD_64` environment variable, which affect only 32-bit programs and 64-bit programs, respectively. See the *Linker and Libraries Guide* for the version of Linux you are running to determine if these variables are supported.

Using Fix and Continue With Runtime Checking

You can use runtime checking along with the `fix` and `cont` commands to isolate and fix programming errors rapidly. Fix and continue provide a powerful combination that can save you a lot of debugging time. For 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 \Qa.out' have been changed (fixed):
bug.c
Remember to remake program.
```

For more information about using fix and continue, see "Memory Leak (mel) Error" on page 170.

Runtime Checking Application Programming Interface

Both leak detection and access checking require that the standard heap management routines in the shared library libc.so be used so that runtime checking can keep track of all the allocations and deallocations in the program. Many applications write their own memory management routines either on top of the malloc() or free() function or stand alone. When you use your own allocators (referred to as *private allocators*), runtime checking cannot automatically track them. Therefore, you do not learn of leak and memory access errors resulting from their improper use.

However, runtime checking provides an API for the use of private allocators. This API allows the private allocators the same treatment as the standard heap allocators. The API itself is provided in the header file rtc-api.h and is distributed as a part of Oracle Developer Studiosoftware. The man page rtc_api(3x) details the runtime checking API entry points.

Some minor differences might exist with runtime checking access error reporting when private allocators do not use the program heap. When a memory access error referring to a standard heap block occurs, the error report typically includes the location of the heap block allocation. When private allocators do not use the program heap, the error report might not include the allocation item.

Using the runtime checking API to track memory allocators in `libumem` is not required. Runtime checking interposes `libumem` heap management routines and redirects them to the corresponding `libc` functions.

Using Runtime Checking in Batch Mode

The `bcheck` utility is a convenient batch interface to the runtime checking feature of `dbx`. It runs a program under `dbx` and, by default, places the runtime checking error output in the default file `program.errs`.

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

Note - Before running the `bcheck` utility on a system running the 64-bit Linux OS, you must set the `_DBX_EXEC_32` environment variable.

`bcheck` Syntax

The syntax for `bcheck` is:

```
bcheck [-V] [-access | -all | -leaks | -memuse] [-xexec32] [-o logfile] [-q]
[-s script] program [args]
```

Use the `-o` *logfile* option to specify a different name for the logfile. Use the `-s` *script* option before executing the program to read in the `dbx` commands contained in the file *script*. The *script* file typically contains commands like `suppress` and `dbxenv` to tailor the error output of the `bcheck` utility.

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

`bcheck` Examples

To perform only leak checking on `hello`:

```
bcheck hello
```

To perform only access checking on mach with the argument 5:

```
bcheck -access mach 5
```

To 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. 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.

Enabling Batch Mode Directly From dbx

You can also enable a batch-like mode directly from dbx by setting the dbxenv variables rtc_auto_continue and rtc_error_log_file_name.

If rtc_auto_continue is set to on, runtime checking continues to find errors and keeps running automatically. It redirects errors to the file named by the dbxenv variable rtc_error_log_file_name. The default log file name is /tmp/dbx.errlog.*unique-ID*. To redirect all errors to the terminal, set the rtc_error_log_file_name environment variable to /dev/tty.

By default, rtc_auto_continue is set to off.

Troubleshooting Tips

After error checking has been enabled for a program and the program is run, one of the following errors might be detected:

```
librtc.so and dbx version mismatch; Error checking disabled
```

This error can occur if you are using runtime checking on an attached process and have set `LD_AUDIT` to a version of `rtcaudit.so` other than the one shipped with your Oracle Developer Studio `dbx` image. To fix this, change the setting of `LD_AUDIT`.

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

Runtime checking was unable to find patch space close enough to a load object for access checking to be enabled. See "Runtime Checking Limitations" on page 162.

Runtime Checking Limitations

This section describes the limitations of runtime checking.

Performance Improves With More Symbols and Debug Information

Access checking requires some symbol information in the load objects. When a load object is fully stripped, runtime checking might not catch all of the errors. Read from uninitialized (`rui`) memory errors might be incorrect and therefore are suppressed. You can override the suppression with the `unsuppress rui` command. To retain the symbol table in the load object, use the `-x` option when stripping a load object.

Runtime checking cannot catch all array out-of-bounds errors. Bounds checking for static and stack memory is not available without debug information.

`SIGSEGV` and `SIGALTSTACK` Signals Are Restricted on x86 Platforms

Runtime checking instruments memory access instructions for access checking. These instructions are handled by a `SIGSEGV` handler at runtime. Because runtime checking requires its own `SIGSEGV` handler and signal alternate stack, an attempt to install a `SIGSEGV` handler or `SIGALTSTACK` handler results in an `EINVAL` error or ignoring the attempt.

`SIGSEGV` handler calls cannot be nested. Doing so results in the error `terminating signal 11 SEGSEGV`. If you receive this error, use the `rtc skippatch` command to skip instrumentation of the affected function.

Performance Improves When Sufficient Patch Area Is Available Within 8 MB of All Existing Code (SPARC Platforms Only).

Two problems might arise if a sufficient patch area is not available within 8 megabytes of all existing code.

- Slowness

  When access checking is enabled, `dbx` replaces each load and store instruction with a branch instruction that branches to a patch area. This branch instruction has an 8-megabyte range. If the debugged program has used all the of address space within 8 megabytes of the particular load or store instruction being replaced, no place exists to put the patch area. In this case, `dbx` invokes a trap handler instead of using a branch. The transfer of control to a trap handler is significantly slower (up to 10 times), but does not suffer from the 8 megabyte limit.
- Out register override problem in V8+ mode

  The trap handler limitation affects access checking if both of the following conditions apply:

  - The process being debugged is instrumented using traps.
  - The process uses the V8+ instruction set.

  The problem occurs because the sizes of out registers and in registers on V8+ architecture are different. Out registers are 64 bits long, while in registers are only 32 bits long. When a trap handler is invoked, out registers are copied into in registers and the higher 32 bits are lost. Therefore, if the process being debugged uses the higher 32 bits of out registers, the process might run incorrectly when access checking is enabled.

  The compilers use the V8+ architecture by default when creating 32-bit SPARC based binaries, but you can tell the compilers to use the V8 architecture with the `-xarch` option. Unfortunately, system runtime libraries are unaffected by recompiling your application.

  `dbx` automatically skips instrumentation of the following functions and libraries that are known not to work correctly when instrumented with traps:

  - `server/libjvm.so`
  - `client/libjvm.so`
  - `` `libfsu_isa.so`__f_cvt_real ``
  - `` `libfsu_isa.so`__f90_slw_c4 ``

  However, skipping instrumentation might result in incorrect RTC error reports.

If either of the above conditions applies to your program and the program starts to behave differently when you enable access checking, the trap handler limitation probably affects your program. To work around the limitation, you can do the following:

- Use the `rtc skippatch` command to skip instrumentation of the code in your program that uses the functions and libraries listed above. Generally, tracking the problem to a specific function is difficult, so you might want to skip instrumentation of an entire load object. The `rtc showmap` command displays a map of instrument types sorted by address.
- Try using 64-bit SPARC-V9 instead of 32-bit SPARC-V8.

  If possible, recompile your program for V9 architecture, in which all of the registers are 64 bits long.
- Try adding patch area object files.

  You can use the `rtc_patch_area` shell script to create special `.o` files that can be linked into the middle of a large executable or shared library to provide more patch space. For more information, see the `rtc_patch_area`(1) man page.

  When `dbx` reaches the 8-megabyte limit, it tells you which load object was too large (the main program or a shared library) and displays the total patch space needed for that load object.

  For the best results, the special patch object files should be evenly spaced throughout the executable or shared library, and the default size (8 megabytes) or smaller should be used. Also, do not add more than 10-20% more patch space than `dbx` says it requires. For example, if dbx says that it needs 31 megabytes for `a.out`, then add four object files created with the `rtc_patch_area` script, each one 8 megabytes in size, and space them approximately evenly throughout the executable.

  When `dbx` finds explicit patch areas in an executable, it prints the address ranges spanned by the patch areas, which can help you to place them correctly on the link line.
- Try dividing the large load object into smaller load objects.

  Split up the object files in your executable or your large library into smaller groups of object files, then link them into smaller parts. If the large file is the executable, then divide it into a smaller executable and a series of shared libraries. If the large file is a shared library, then rearrange it into a set of smaller libraries.

  This technique enables `dbx` to find space for patch code in between the different shared objects.
- Try adding a “pad” `.so` file.

  This solution should be necessary only if you are attaching to a process after it has started up.

  The runtime linker might place libraries so close together that patch space cannot be created in the gaps between the libraries. When `dbx` starts the executable with runtime checking enabled, it asks the runtime linker to place an extra gap between the shared libraries. However, when attaching to a process that was not started by `dbx` with runtime checking enabled, the libraries might be too close together.

If the runtime libraries are too close together and if you cannot start the program using dbx, then you can try creating a shared library using the rtc_patch_area script and linking it into your program between the other shared libraries. See the rtc_patch_area(1) man page for more details.

Runtime Checking Errors

Errors reported by runtime checking generally fall in two categories: access errors and leaks.

Access Errors

When access checking is enabled, runtime checking detects and reports the types of errors described in this section.

Bad Free (baf) Error

Problem: Attempt to free memory that has never been allocated.

Possible causes: Passing a non-heap data pointer to free() or realloc().

Example:

```
char a[4];
char *b = &a[0];

free(b);                    /* Bad free (baf) */
```

Duplicate Free (duf) Error

Problem: Attempt to free a heap block that has already been freed.

Possible causes: Calling free() more than once with the same pointer. In C++, using the delete operator more than once on the same pointer.

Example:

```
char *a = (char *)malloc(1);
```

```
free(a);
free(a);                    /* Duplicate free (duf) */
```

Misaligned Free (`maf`) Error

Problem: Attempt to free a misaligned heap block.

Possible causes: Passing an improperly aligned pointer to `free()` or `realloc()`; changing the pointer returned by `malloc`.

Example:

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

Misaligned Read (`mar`) Error

Problem: Attempt to read data from an address without proper alignment.

Possible causes: Reading 2, 4, or 8 bytes from an address that is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:

```
char *s = "hello world";
int *i = (int *)&s[1];
int j;

j = *i;                    /* Misaligned read (mar) */
```

Misaligned Write (`maw`) Error

Problem: Attempt to write data to an address without proper alignment.

Possible causes: Writing 2, 4, or 8 bytes to an address that is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:

```
char *s = "hello world";
int *i = (int *)&s[1];
```

```
*i = 0;                    /* Misaligned write (maw) */
```

Out of Memory (`oom`) Error

Problem: Attempt to allocate memory beyond physical memory available.

Cause: Program cannot obtain more memory from the system. Useful in locating problems that occur when the return value from `malloc()` is not checked for `NULL`, which is a common programming mistake.

Example:

```
char *ptr = (char *)malloc(0x7fffffff);
/* Out of Memory (oom), ptr == NULL */
```

Read From Array Out-of-Bounds (`rob`) Error

Problem: Attempt to read from array out-of-bounds memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block.

Example:

```
char *cp = malloc (10);
char ch = cp[10];
```

Read From Unallocated Memory (`rua`) Error

Problem: Attempt to read from nonexistent, unallocated, or unmapped memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block or accessing a heap block that has already been freed.

Example:

```
char *cp = malloc (10);
free (cp);
cp[0] = 0;
```

Read From Uninitialized Memory (`rui`) Error

Problem: Attempt to read from uninitialized memory.

Possible causes: Reading local or heap data that has not been initialized.

Example:

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

Write to Array Out-of-Bounds Memory (`wob`) Error

Problem: Attempt to write to array out-of-bounds memory.

Possible causes: A stray pointer or overflowing the bounds of a heap block.

Example:

```
char *cp = malloc (10);
cp[10] = 'a';
```

Write to Read-Only Memory (`wro`) Error

Problem: Attempt to write to read-only memory.

Possible causes: Writing to a text address, writing to a read-only data section (`.rodata`), or writing to a page that `mmap` has made read-only.

Example:

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

Write to Unallocated Memory (`wua`) Error

Problem: Attempt to write to nonexistent, unallocated, or unmapped memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:

```
char *cp = malloc (10);
free (cp);
cp[0] = 0;
```

Memory Leak Errors

With leak checking enabled, runtime checking reports the following types of errors.

Address in Block (`aib`) Error

Problem: A possible memory leak. There is no reference to the start of an allocated block, but there is at least one reference to an address within the block.

Possible causes: The only pointer to the start of the block is incremented.

Example:

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

Address in Register (`air`) Error

Problem: A possible memory leak. An allocated block has not been freed and no reference to the block exists anywhere in program memory but a reference exists in a register.

Possible causes: This situation can occur legitimately if the compiler keeps a program variable only in a register instead of in memory. The compiler often does this for local variables and function parameters when optimization is enabled. If this error occurs when optimization has not been enabled, it is likely to be an actual memory leak. This situation can occur if the only pointer to an allocated block goes out of scope before the block is freed.

Example:

```
if (i == 0) {
       char *ptr = (char *)malloc(4);
       /* ptr is going out of scope */
}
```

```
  /* Memory Leak or Address in Register */
```

Memory Leak (`mel`) Error

Problem: An allocated block has not been freed and no reference to the block exists anywhere in the program.

Possible causes: Program failed to free a block no longer used.

Example:

```
char *ptr;

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

♦ ♦ ♦ CHAPTER 10

Debugging Multithreaded Applications

dbx can debug multithreaded applications that use either Oracle Solaris threads or POSIX threads. With dbx, you can examine stack traces of each thread, resume all threads, step or next a specific thread, and navigate between threads.

This chapter describes how to find information about and debug threads using the dbx thread commands. It contains the following sections:

- "Understanding Multithreaded Debugging" on page 171
- "Understanding Thread Creation Activity" on page 175
- "Understanding LWP Information" on page 176

Understanding Multithreaded Debugging

dbx recognizes a multithreaded program by detecting whether it utilizes libthread.so. The program uses libthread.so either by explicitly being compiled with -lthread or -mt, or implicitly by being compiled with -lpthread.

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

dbx is synchronous, so 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 Oracle Solaris *Multithreaded Programming Guide*.

Thread Information

The thread information shown in the following example is available in dbx.

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

For native code, each line of information is composed of the following:

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

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

For Java code, each line of information is composed of the following:

- t@*number*, a dbx-style thread ID
- The thread state
- The thread name in single quotation marks
- A number indicating the thread priority

Thread and LWP States

suspended	The thread has been explicitly suspended.
runnable	The thread is runnable and is waiting for an LWP as a computational resource.

zombie	When a detached thread exits (`thr_exit()`), it is in a zombie state until it has rejoined through the use of `thr_join(). THR_DETACHED` is a flag specified at thread creation time (`thr_create()`). A non-detached thread that exits is in a zombie state until it has been reaped.
asleep on *syncobj*	Thread is blocked on the given synchronization object. Depending on what level of support `libthread` and `libthread_db` provide, *syncobj* might be as simple as a hexadecimal address or something with more information content.
active	The thread is active on an LWP but `dbx` cannot access the LWP.
unknown	`dbx` cannot determine the state.
lwpstate	A bound or active thread state has the state of the LWP associated with it.
running	LWP was running but was stopped in synchrony with some other LWP.
syscall *num*	LWP stopped on an entry into the given system call #.
syscall return *num*	LWP stopped on an exit from the given system call #.
job control	LWP stopped due to job control.
LWP suspended	LWP is blocked in the kernel.
single stepped	LWP has just completed a single step.
breakpoint	LWP has just hit a breakpoint.
fault *num*	LWP has incurred the given fault #.
signal *name*	LWP has incurred the given signal.
process sync	The process to which this LWP belongs has just started executing.
LWP death	LWP is in the process of exiting.

Viewing the Context of Another Thread

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

```
thread [-blocks] [-blockedby] [-info] [-hide] [-unhide] [-suspend] [-resume] thread_id
```

To display the current thread:

`thread`

To switch to thread *thread-ID*:

`thread` *thread-ID*

For more information, see “`thread` Command” on page 396.

Viewing the Threads List

To view the threads list, use the `threads` command. The syntax is:

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

To print the list of all known threads:

`threads`

To print threads normally not printed (zombies):

`threads -all`

For an explanation of the threads list, see “Thread Information” on page 172.

For more information on the `threads` command, see “`threads` Command” on page 398.

Resuming Execution

Use the `cont` command to resume program execution. Currently, threads use synchronous breakpoints, so all threads resume execution. However, you can resume a single thread using the `call` command with the `-resumeone` option.

Consider the following two scenarios when debugging a multithreaded application where many threads call the function `lookup()`:

- You set a conditional breakpoint:

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

  When `t@1` stops at the call to `lookup()`, `dbx` attempts to evaluate the condition and calls `strcmp()`.

- You set a breakpoint:

  ```
  stop in lookup
  ```

  When `t@1` stops at the call to `lookup()`, you issue the command:

  ```
  call strcmp(name, "troublesome")
  ```

When calling `strcmp()`, `dbx` would resume all threads for the duration of the call, which is similar to what `dbx` does when you are single-stepping with the `next` command. It does so because resuming only `t@1` has the potential to cause a deadlock if `strcmp()` tries to grab a lock that is owned by another thread.

A drawback to resuming all threads in this case is that `dbx` cannot handle another thread, such as `t@2`, hitting the breakpoint at `lookup()` while`strcmp()` is being called. It emits a warning like one of the following:

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

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

In such cases, if you can ascertain that the function called in the conditional expression will not grab a mutex, you can use the `-resumeone` event modifier to force `dbx` to resume only `t@1`:

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

Only the thread that hit the breakpoint in `lookup()` would be resumed in order to evaluate `strcmp()`.

This approach does not help in cases such as the following examples:

- If the second breakpoint on `lookup()` happens in the same thread because the conditional recursively calls `lookup()`
- If the thread on which the conditional runs yields, sleeps, or in some manner relinquishes control to another thread

Understanding Thread Creation Activity

You can get an idea of how often your application creates and destroys threads by using the `thr_create` event and `thr_exit` event, as in the following example:

```
(dbx) trace thr_create
(dbx) trace thr_exit
(dbx) run

trace: thread created t@2 on l@2
trace: thread created t@3 on l@3
trace: thread created t@4 on l@4
trace: thr_exit t@4
trace: thr_exit t@3
trace: thr_exit t@2
```

The application created three threads. Note how the threads exited in reverse order from their creation, which might indicate that had the application had more threads, the threads would accumulate and consume resources.

To get more extensive information, you could try the following example in a different session:

```
(dbx) when thr_create { echo "XXX thread $newthread created by $thread"; }
XXX thread t@2 created by t@1
XXX thread t@3 created by t@1
XXX thread t@4 created by t@1
```

The output shows that all three threads were created by thread `t@1`, which is a common multithreading pattern.

Suppose you want to debug thread `t@3` from its outset. You could stop the application at the point that thread `t@3` is created as follows:

```
(dbx) stop thr_create t@3
(dbx) run
t@1 (l@1) stopped in tdb_event_create at 0xff38409c
0xff38409c: tdb_event_create       :    retl
Current function is main
216       stat = (int) thr_create(NULL, 0, consumer, q, tflags, &tid_cons2);
(dbx)
```

If your application occasionally spawns a new thread from thread `t@5` instead of thread `t@1`, you could capture that event as follows:

```
(dbx) stop thr_create -thread t@5
```

Understanding LWP Information

Normally, you need not be aware of LWPs. However, sometimes thread level queries cannot be completed. In these cases, use the `lwps` command to show information about LWPs.

```
(dbx) lwps
    l@1 running in main()
    l@2 running in sigwait()
    l@3 running in _lwp_sema_wait()
  *>l@4 breakpoint in Queue_dequeue()
    l@5 running in _thread_start()
(dbx)
```

Each line of the LWP list contains the following:

- The * (asterisk) indicates that an event requiring user attention has occurred in this LWP.
- The > (arrow) denotes the current LWP.
- `l@`*number* refers to a particular LWP.
- The LWP state.
- The name of the function that the LWP is currently executing.

Use the `lwp` command to list or change the current LWP.

CHAPTER 11

Debugging Child Processes

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

This chapter contains the following sections:

- "Attaching to Child Processes" on page 179
- "Following the `exec` Function" on page 180
- "Following the `fork` Function" on page 180
- "Interacting With Events" on page 180

Attaching to Child Processes

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

- When starting `dbx`:

  ```
  $ dbx program-name process-ID
  ```
- From the `dbx` command line:

  ```
  (dbx) debug program-name  process-ID
  ```

If you include a `-` (minus sign) rather than a program name, `dbx` automatically finds the executable associated with the given process ID. After using a `-`, a subsequent `run` command or `rerun` command does not work because `dbx` does not know the full path name of the executable.

You can also attach to a running child process in the Oracle Developer StudioIDE. For more information, see the online help fro the IDE and for `dbxtool`.

Following the exec Function

If a child process executes a new program using the `exec(2)`function or one of its variations, the process ID does not change but the process image does. `dbx` automatically takes note of a call to the `exec()` function and does an implicit reload of the newly executed program.

The original name of the executable is saved in `$oprog`. To return to it, use `debug $oprog`.

Following the fork Function

If a child process calls the `vfork(2)`, `fork1(2)`, or `fork(2)` function, the process ID changes, but the process image stays the same, The behavior of `dbx` depends on how the `dbxenv` variable `follow_fork_mode` is set.

parent	In the traditional behavior, `dbx` ignores the fork and follows the parent.
child	`dbx` automatically switches to the forked child using the new process ID. All connection to and awareness of the original parent is lost.
both	This mode is available only when using `dbx` through the Oracle Developer Studio IDE or `dbxtool`.
ask	You are prompted to choose `parent`, `child`, `both`, or `stop to investigate` whenever `dbx` detects a fork. If you choose `stop`, you can examine the state of the program, then type `cont` to continue. You will be prompted again to select which way to proceed. `both` is supported only in the Oracle Developer Studio IDE and `dbxtool`.

Interacting With Events

All breakpoints and other events are deleted for any `exec()` or `fork()` process.
You can override the deletion for forked processes by setting the `dbxenv` variable `follow_fork_inherit` to `on`, or make the events permanent using the `-perm eventspec` modifier. For more information about using event specification modifiers, see "`cont at` Command" on page 270.

♦ ♦ ♦ CHAPTER 12

Debugging OpenMP Programs

The OpenMP™ application programming interface (API) is a portable, parallel programming model for shared memory multiprocessor architectures, developed in collaboration with a number of computer vendors. Support for debugging Fortran, C++, and C OpenMP programs with `dbx` is based on the general multithreaded debugging features of `dbx`.

This chapter contains the following sections:

- "How Compilers Transform OpenMP Code" on page 181
- "`dbx` Functionality Available for OpenMP Code" on page 182
- "Execution Sequence of OpenMP Code" on page 189

See the*Oracle Developer Studio 12.5: OpenMP API User's Guide* for information on the directives, runtime library routines, and environment variables comprising the OpenMP Version 4.0 Application Program Interfaces, as implemented by the Oracle Developer Studio Fortran and C compilers.

How Compilers Transform OpenMP Code

To better describe OpenMP debugging, it is helpful to understand how OpenMP code is transformed by the compilers. Consider the following Fortran example:

```
program example
    integer i, n
    parameter (n = 1000000)
    real sum, a(n)

    do i = 1, n
    a(i) = i*i
    end do

     sum = 0

 !$OMP PARALLEL DO DEFAULT(PRIVATE), SHARED(a, sum)
```

```
13
14        do i = 1, n
15        sum = sum + a(i)
16        end do
17
18    !$OMP END PARALLEL DO
19
20        print*, sum
21        end program example
```

The code in line 12 through line 18 is a parallel region. The f95 compiler converts this section of code to an outlined subroutine that will be called from the OpenMP runtime library. This outlined subroutine has an internally generated name, in this case `_$d1A12.MAIN_`. The f95 compiler then replaces the code for the parallel region with a call to the OpenMP runtime library and passes the outlined subroutine as one of its arguments. The OpenMP runtime library handles all the thread-related issues and dispatches slave threads that execute the outlined subroutine in parallel. The C compiler works in the same way.

When debugging an OpenMP program, the outlined subroutine is treated by `dbx` as any other function, with the exception that you cannot explicitly set a breakpoint in that function by using its internally generated name.

`dbx` Functionality Available for OpenMP Code

In addition to the usual functionality for debugging multithreaded programs, `dbx` provides functionality for debugging an OpenMP program. All of the `dbx` commands that operate on threads and LWPs can be used for OpenMP debugging. `dbx` does not support asynchronous thread control in OpenMP debugging.

Single-Stepping Into a Parallel Region

`dbx` can single-step into a parallel region. Because a parallel region is outlined and called from the OpenMP runtime library, a single step of execution actually involves several layers of runtime library calls that are executed by threads created for this purpose. When you single-step into the parallel region, the first thread that reaches the breakpoint causes the program to stop. This thread might be a slave thread rather than the master thread that initiated the stepping.

For example, refer to the Fortran code in"How Compilers Transform OpenMP Code" on page 181, and assume that master thread `t@1` is at line 10. You single-step into line 12, and slave threads `t@2`, `t@3`, and `t@4` are created to execute the runtime library calls. Thread `t@3` reaches the breakpoint first and causes the program execution to stop. The single step that was initiated by thread t@1 therefore ends on thread t@3. This behavior is different from normal stepping in which you are usually on the same thread after the single step as before.

Printing Variables and Expressions

dbx can print all shared, private, and thread-private variables. If you try to print a thread private variable outside of a parallel region, the master thread's copy is printed. The whatis command prints data sharing attributes for shared and private variables within a parallel construction. It prints data sharing attributes for thread-private variables regardless of whether they are within a parallel construction. For example:

```
(dbx) whatis p_a
# OpenMP first and last private variable
int p_a;
```

The print -s command prints the value of an expression *expression* for each thread in the current OpenMP parallel region if the expression contains private or thread private variables. For example:

```
(dbx) print -s p_a
thread t@3: p_a = 3
thread t@4: p_a = 3
```

If the expression does not contain any private or thread private variables, only one value is printed.

Printing Region and Thread Information

Use the omp_pr command to print a description of the current parallel region or a specified parallel region, including the parent region, parallel region ID, team size (number of threads), and program location (program counter address). For example:

```
(dbx) omp_pr
parallel region 127283434369843201
    team size = 4
    source location = test.c:103
    parent = 127283430568755201
```

You can also print descriptions of all the parallel regions along the path from the current parallel region or specified parallel region to its root. For example:

```
(dbx) omp_pr -ancestors
parallel region 127283434369843201
```

```
    team size = 4
    source location = test.c:103
    parent = 127283430568755201

    parallel region 127283430568755201
        team size = 4
        source location = test.c:95
        parent = <no parent>
```

You can also print the whole parallel region tree. For example:

```
(dbx) omp_pr -tree
parallel region 127283430568755201
    team size = 4
    source location = test.c:95
    parent = <no parent>

    parallel region 127283434369843201
        team size = 4
        source location = test.c:103
        parent = 127283430568755201
```

For more information, see "omp_pr Command" on page 366.

Use the omp_tr command to print a description of the current task region or a specified task region, including the task region ID, state (spawned, executing, waiting), executing thread, program location (program counter address), unfinished children, and parent. For example:

```
(dbx) omp_tr
task region 65540
    type = implicit
    state = executing
    executing thread = t@4
    source location == test.c:46
    unfinished children = 0
    parent = <no parent>
```

You can also print descriptions of all the task regions along the path from the current task region or specified task region to its root.

```
(dbx) omp_tr -ancestors
task region 196611
    type = implicit
    state = executing
    executing thread = t@3
    source location - test.c:103
    unfinished children = 0
    parent = 131075
```

```
    task region 131075
        type = implicit
        state = executing
        executing thread = t@3
        unfinished children = 0
        parent = <no parent>
```

And you can print the whole task region tree. For example:

```
(dbx) omp_tr -tree
task region 10
    type = implicit
    state = executing
    executing thread = t@10
    source location = test.c:103
    unfinished children = 0
    parent = <no parent>
task region 7
    type = implicit
    state = executing
    executing thread = t@7
    source location = test.c:103
    unfinished children = 0
    parent = <no parent>
task region 6
    type implicit
    state = executing
    executing thread = t@6
    source location = test.c:103
    unfinished children = 0
    parent = <o parent>
task region 196609
    type = implicit
    state = executing
    executing thread = t@1
    source location = test.c:95
    unfinished children = 0
    parent = <no parent>

    task region 262145
        type = implicit
        state = executing
        executing thread = t@1
        source location = test.c:103
        unfinished children - 0
        parent = 196609
```

For more information, see "omp_tr Command" on page 367.

Use the omp_loop command to print a description of the current loop, including the scheduling type (static, dynamic, guided, auto, or runtime), ordered or not, bounds, steps or strides, and number of iterations. For example:

```
(dbx) omp_loop
    ordered loop: no
    lower bound: 0
    upper bound: 3
    step: 1
    chunk: 1
    schedule type: static
    source location: test.c:49
```

For more information, see "omp_loop Command" on page 366.

Use the omp_team command to print all the threads on the current team or the team of a specified parallel region. For example:

```
(dbx) omp_team
team members:
    0: t@1 state = in implicit barrier, task region = 262145
    1: t@6 state = in implicit barrier, task region = 6
    2: t@7 state = working, task region = 7
    3: t@10 state = in implicit barrier, task region = 10
```

For more information, see "omp_team Command" on page 367.

When you are debugging OpenMP code, the thread -info prints the OpenMP thread ID, parallel region ID, task region ID, and OpenMP thread state, in addition to the usual information about the current or specified thread. For more information, see "thread Command" on page 396.

Serializing the Execution of a Parallel Region

Use the omp_serialize command to serialize the execution of the next encountered parallel region for the current thread or for all threads in the current team. For more information, see "omp_serialize Command" on page 367.

Using Stack Traces

When execution is stopped in a parallel region, a where command shows a stack trace that contains the outlined subroutine.

```
(dbx) where
current thread: t@4
=>[1] _$d1E48.main(), line 52 in "test.c"
  [2] _$p1I46.main(), line 48 in "test.c"

--- frames from parent thread ---
current thread: t@1
  [7] main(argc = 1, argv = 0xffffffff7fffec98), line 46 in "test.c"
```

The top frame on the stack is the frame of the outlined function. Even though the code is outlined, the source line number still maps back to 15.

When execution is stopped in a parallel region, a `where` command from a slave thread prints the master thread's stack trace if the relevant frames are still active. A `where` command from the master thread has a full traceback.

You can also determine how execution reached the breakpoint in a slave thread by first using the `omp_team` command to list all the threads in the current team, and then switching to the master thread (the thread with the OpenMP thread ID `0`) and getting a stack trace from that thread.

Using the dump Command

When execution is stopped in a parallel region, a `dump` command might print more than one copy of private variables. In the following example, the `dump` command prints two copies of the variable `i`:

```
[t@1 l@1]: dump
i = 1
sum = 0.0
a = ARRAY
i = 1000001
```

Two copies of variable `i` are printed because the outlined routine is implemented as a nested function of the hosting routine, and private variables are implemented as local variables of the outlined routine. Because a `dump` command prints all the variables in scope, both the `i` in the hosting routine and the `i` in the outlined routine are displayed.

Using Events

dbx provides events you can use with the stop, when, and trace commands on your OpenMP code. For information about using events with these commands, see "Setting Event Specifications" on page 274.

Synchronization Events

`omp_barrier [type] [state]`

Tracks the event of a thread entering a barrier.

type valid values are:

- `explicit` – Track explicit barriers
- `implicit` – Track implicit barriers

If you do not specify *type*, then only explicit barriers are tracked.

state valid values are:

- `enter` – Report the event when any thread enters a barrier
- `exit` – Report the event when any thread exits a barrier
- `all_entered` – Report the event when all threads have entered a barrier

If you do not specify *state*, the default is `all_entered`.

If you specify `enter` or `exit`, you can include a thread ID to specify tracking only for that thread.

`omp_taskwait [state]`

Tracks the event of a thread entering a taskwait.

state valid values are:

- `enter` – Report the event when a thread enters a taskwait
- `exit` – Report the event when all child tasks have finished

If you do not specify *state*, then `exit` is the default.

`omp_ordered [state]`

Tracks the event of a thread entering an ordered region.

state valid values are:

- `begin` – Report the event when an ordered region begins
- `enter` – Report the event when a thread enters an ordered region
- `exit` – Report the event when a thread exits an ordered region

If you do not specify *state*, then the default is `enter`.

`omp_critical` Tracks the event of a thread entering a critical region.

`omp_atomic [`*state*`]` Tracks the event of a thread entering an atomic region.

state valid values are:

- `begin` – Report the event when an atomic region begins
- `exit` – Report the event when a thread exits an atomic region

If you do not specify *state*, then the default is `begin`.

`omp_flush` Tracks the event of a thread executing a flush.

Other Events

`omp_task [`*state*`]` Tracks the creation and termination of tasks.

state valid values are:

- `create` – Report the event when a task has just been created and before its execution begins
- `start` – Report the event when a task starts its execution
- `finish` – Report the event when a task has finished its execution and is about to be terminated

If you do not specify *state*, the default is `start`.

`omp_master` Tracks the event of the master thread entering the master region.

`omp_single` Tracks the event of a thread entering a single region.

Execution Sequence of OpenMP Code

When you are single-stepping inside a parallel region in an OpenMP program, the execution sequence might not be the same as the source code sequence. This difference in sequence occurs because the code in the parallel region is usually transformed and rearranged by the compiler. Single-stepping in OpenMP code is similar to single-stepping in optimized code where the optimizer has usually moved code around.

♦ ♦ ♦ CHAPTER 13

Working With Signals

This chapter describes how to use `dbx` to work with signals.

This chapter contains the following sections.

- "Understanding Signal Events" on page 191
- "Catching Signals" on page 192
- "Sending a Signal to a Program" on page 196
- "Automatically Handling Signals" on page 196

Understanding Signal Events

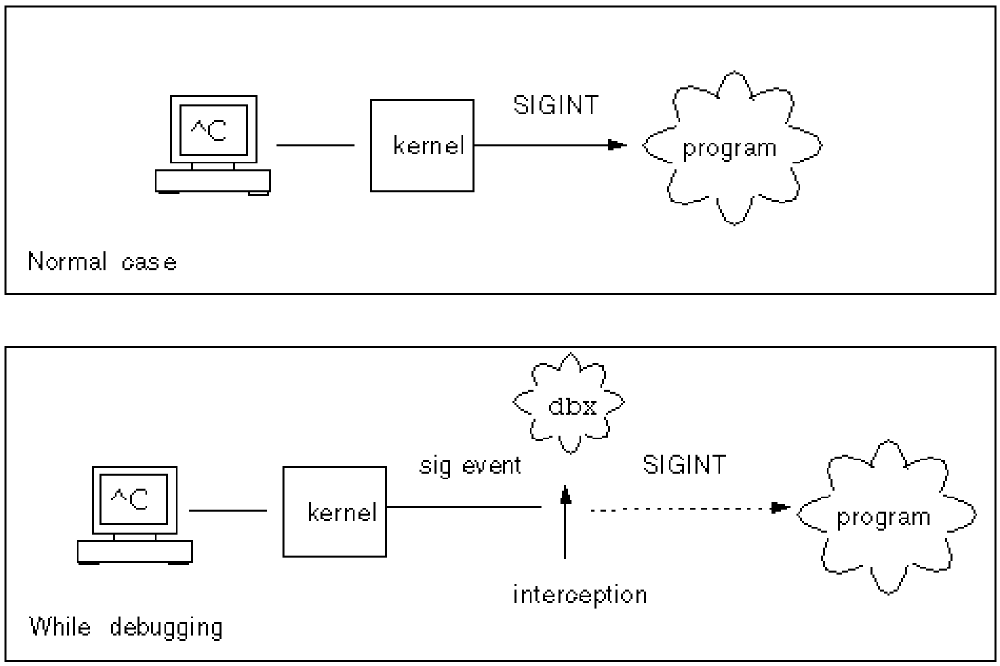
When a signal is to be delivered to a process that is being debugged, the signal is redirected to `dbx` by the kernel. When this happens, you usually receive a prompt. You then have two choices:

- Cancel the signal when the program is resumed (the default behavior of the `cont` command), facilitating easy interruption and resumption with `SIGINT` (Control-C), as shown in Figure 2, "Intercepting and Cancelling the `SIGINT` Signal," on page 192.
- Forward the signal to the process using the following command:

  `cont -sig` *signal*

  signal can be either a signal name or a signal number.

FIGURE 2 Intercepting and Cancelling the `SIGINT` Signal



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

`ignore` *signal*

However, the signal is still forwarded to the process. A default set of signals is automatically forwarded in this manner (see "`ignore` Command" on page 347).

Catching Signals

`dbx` supports the `catch` command, which instructs `dbx` to stop a program when `dbx` detects any of the signals appearing on the catch list.

By default, the catch list contains many of the more than 33 detectable signals. (The numbers depend upon the operating system and version.) You can change the default catch list by adding signals to or removing them from the default catch list.

Note - The list of signal names that `dbx` accepts includes all of those supported by the versions of the Oracle Solaris operating environment that `dbx` supports. So `dbx` might accept a signal that is not supported by the version of the Oracle Solaris operating environment you are running. For example, `dbx` might accept a signal that is supported by the Oracle Solaris 9 OS even through you are running the Oracle Solaris 7 OS. For a list of the signals supported by the Oracle Solaris OS that you are running, see the `signal`(3head) man page.

To see the list of signals currently being trapped, type **`catch`** with no *signal* argument.

```
(dbx) catch
```

To see a list of the signals currently being *ignored* by `dbx` when the program detects them, type **`ignore`** with no *signal* argument.

```
(dbx) ignore
```

Changing the Default Signal Lists

You control which signals cause the program to stop by moving the signal names from one list to the other. To move signal names, supply a signal name that currently appears on one list as an argument to the other list.

For example, to move the `QUIT` and `ABRT` signals from the catch list to the ignore list:

```
(dbx) ignore QUIT ABRT
```

Trapping the FPE Signal (Oracle Solaris Only)

Floating-point and integer arithmetic operations can cause exceptions like overflow or divide by 0. Such exceptions are often silent such that the system returns a reasonable answer (e.g. NaN) as the result for the operation that caused the exception. Therefore these exceptions are not visible to `dbx`.

You can arrange for the exception to not be silent and instead cause a trap. Then the operating system will convert the trap to a `SIGFPE` and deliver it to the process and `dbx` can intercept this signal delivery. Note the following:

- `F77` by default does not trap on any floating-point exception.

- F95 by default traps on invalid operand, divide-by-zero, and overflow exceptions, but not underflow and inexact exceptions.
- C and C++ do not trap on floating-point exceptions by default.
- There is no provision for integer overflow to implicitly trigger a SIGFPE. On SPARC, you can use the TVS (trap-on-overflow-set) assembly instruction. On SPARC or Intel, you can use analogous branch-on-overflow-set instructions.

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.

You can enable a trap using the following:

- fpsetmask – This function strictly controls the enabling of traps. See the fpsetmask(3C) man page.

  Example:

  ```
  #include <ieeefp.h>
      int main() {
      fpsetmask(FP_X_INV|FP_X_OFL|FP_X_UFL|FP_X_DZ|FP_X_IMP);
      ...
  ```
- ieee_handler – There is no exact analog of psetmask(3c) for Fortran. Instead, you can enable traps by establishing the default behavior as follows.

  Example:

  ```
      integer*4 ieeer
      ieeeer  = ieee_handler('set', 'common', SIGFPE_DEFAULT)
  ```

  See the ieee_environment(3f) and ieee_handler(3m) man pages for more information.
- -ftrap compiler flag – This tag, like fpsetmask()(), strictly controls the enabling of traps. For Fortran 95, see the f95(1) man page.

When you enable a floating-point trap handler using one of the previously mentioned methods, the trap enable mask in the hardware floating-point status register is set. This trap enable mask causes the exception to raise the SIGFPE signal at run time.

Once you have inserted a call to fpsetmask()() or ieee_handler()() or compiled the program with the trap handler, load the program into dbx. SIGFPE is caught by default as of Oracle Developer Studio 12.5. With older versions of dbx, ensure that the signal is still in the catch list.

```
(dbx) catch FPE
```

You can further tailor which specific exceptions you see by tweaking the parameters of fpsetmask() and ieee_handler() by using an alternative to the dbx catch command which acts like catch FPE, similar to the following.

```
(dbx) stop sig FPE
(dbx) ignore SIGFPE #don't catch it twice
```

You can use the following code for finer control:

`stop sig FPE` *subcode*

where *subcode* can be one of the following:

FPE_INTDIV	Integer divide by zero.
FPE_INTOVF	Integer overflow.
FPE_FLTDIV	Floating-point divide by zero.
FPE_FLTOVF	Floating-point overflow.
FPE_FLTUND	Floating-point underflow.
FPE_FLTRES	Floating-point inexact result.
FPE_FLTINV	Invalid floating-point operation,
FPE_FLTSUB	Subscript out of range.

Determining Where the Exception Occurred

After adding `FPE` to the catch list, run the program in `dbx`. When the exception that you are trapping occurs, the `SIGFPE` signal is raised and `dbx` stops the program. Then you can trace the call stack using the `dbx` `where` command to help find the specific line number of the program where the exception occurs.

Determining the Cause of the Exception

To determine the cause of the exception on SPARC, 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

On Intel, the floating-point status register is `fstat` for x87 and `mxcsr` for SSE.

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 *Oracle Developer Studio 12.5: Numerical Computation Guide*.

Sending a Signal to a Program

The `dbx` `cont` command supports the `-sig` option, which enables you to resume execution of a program with the program behaving as if it had received the system signal *signal*.

For example, if a program has an interrupt handler for `SIGINT` (`^C`), you can type `^C` to stop the application and return control to `dbx`. If you issue a `cont` command by itself to continue program execution, the interrupt handler never executes. To execute the interrupt handler, send the signal, `SIGINT`, to the program:

```
(dbx) cont -sig int
```

The `step` command, `next` command, and `detach` command also accept the `-sig` option.

Automatically Handling Signals

The event management commands can also deal with signals as events. The following 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 14

Debugging C++ With dbx

This chapter describes how dbx handles C++ exceptions and debugging C++ templates, including a summary of commands used when completing these tasks and examples with code samples. You can debug C++ with dbx normally, with the exceptions that are explained in this chapter.

This chapter contains the following sections:

- "Using dbx With C++" on page 197
- "Exception Handling in dbx" on page 198
- "Debugging With C++ Templates" on page 202

For information about compiling C++ programs, see "Compiling a Program for Debugging" on page 47.

Using dbx With C++

Although this chapter concentrates on two specific aspects of debugging C++, dbx provides full functionality when debugging your C++ programs. You can still do the following tasks with your C++ program:

Note - All the following tasks have been explored in previous chapters.

Find out about class and type definitions	See "Looking Up Definitions of Types and Classes" on page 79
Print or display inherited data members	See "Printing C++ Pointers" on page 120
Find out dynamic information about an object pointer	See "Printing C++ Pointers" on page 120
Debug virtual functions	See "Calling a Function" on page 92

Using runtime type information	See "Printing the Value of a Variable, Expression, or Identifier" on page 120
Set breakpoints on all member functions of a class	See "Setting Breakpoints in All Member Functions of a Class" on page 100
Set breakpoints on all overloaded member functions	See "Setting Breakpoints in Member Functions of Different Classes" on page 100
Set breakpoints on all overloaded nonmember functions	See "Setting Multiple Breakpoints in Nonmember Functions" on page 101
Set breakpoints on all member functions of a particular object	See "Setting Breakpoints in Objects" on page 101
Deal with overloaded functions or data members	See "Setting a Breakpoint in a Function" on page 99

The rest of this chapter concentrates on two specific aspects of debugging 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. You can set up blocks to catch exceptions raised by expressions elsewhere in the code.

While debugging a program, `dbx` enables you to do the following:

- Catch unhandled exceptions before stack unwinding
- Catch unexpected exceptions
- Catch specific exceptions regardless of whether they are handled before stack unwinding
- Determine where a specific exception would be caught if it occurred at a particular point in the program

If you issue a `step` command after stopping at a point where an exception is thrown, control is returned at the start of the first destructor executed during stack unwinding. If you `step` out of a destructor executed during stack unwinding, control is returned at the start of the next destructor. When all destructors have been executed, a `step` command brings you to the catch block handling the throwing of the exception.

Commands for Handling Exceptions

This sections describes the `dbx` commands for handling exceptions.

`exception` Command

The syntax for the `exception` command is as follows:

```
exception [-d | +d]
```

Use the `exception` command to display an exception's type at any time during debugging. If you use the `exception` command without an option, the type shown is determined by the setting of the `dbxenv` variable `output_dynamic_type`:

- If it is set to `on`, the derived type is shown.
- If it is set to `off` (the default), the static type is shown.

Specifying the `-d` or `+d` option overrides the setting of the environment variable.

- If you specify `-d`, the derived type is shown.
- If you specify `+d`, the static type is shown.

For more information, see "`exception` Command" on page 339.

`intercept` Command

The syntax for the intercept command is as follows:

intercept [-all] [-x] [-set] [*typename*]

You can intercept, or catch, exceptions of a specific type before the stack has been unwound.

- Use the `intercept` command with no arguments to list the types that are being intercepted.
- Use `-all` to intercept all exceptions. Use *typename* to add a type to the intercept list.
- Use `-x` to exclude a particular type to the excluded list to keep it from being intercepted.
- Use `-set` to clear both the intercept list and the excluded list, and set the lists to intercept or exclude only throws of the specified types.

For example, to intercept all types except `int`:

```
(dbx) intercept -all -x int
```

To intercept exceptions of type `Error`:

```
(dbx) intercept Error
```

After intercepting too many `CommonError` exceptions with the following command:

```
(dbx) intercept -x CommonError
```

Typing the `intercept` command with no arguments would then show that the intercept list includes unhandled exceptions and unexpected exceptions, which are intercepted by default, plus exceptions of class `Error` except for those of class `CommonError`.

```
(dbx) intercept
-unhandled   -unexpected   class Error -x class CommonError
```

If you then realize that `Error` is not the class of exceptions that interests you, but you do not know the name of the exception class you are looking for, you could try intercepting all exceptions except those of class `Error` by typing:

```
(dbx) intercept -all -x Error
```

For more information, see "`intercept` Command" on page 348.

`unintercept` Command

The syntax for the `unintercept` command is as follows:

`unintercept [-all] [-x] [`*typename*`]`

- Use the `unintercept` command to remove exception types from the intercept list or the excluded list.
- Use the command with no arguments to list the types that are being intercepted (same as the `intercept` command).
- Use `-all` to remove all types from the intercept list. Use *typename* to remove a type from the intercept list. Use `-x` to remove a type from the excluded list.

For more information, see "`unintercept` Command" on page 200.

`whocatches` Command

The `whocatches` command reports where an exception of *typename* would be caught if thrown at the current point of execution. Use this command to find out what would happen if an exception were thrown from the top frame of the stack.

The line number, function name, and frame number of the catch clause that would catch *typename* are displayed. The command returns "*type* `is unhandled`" if the catch point is in the same function that is doing the throw.

For more information, see "whocatches Command" on page 200.

Examples of Exception Handling

This example demonstrates exception handling in `dbx` by 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.

```
#include <stdio.h>

class c {
    int x;
  public:
    c(int i) { x = i; }
    ~c() {
              printf("destructor for c(%d)\n", x);
         }
};

void bar() {
     c c1(3);
     throw(99);
}

int main() {
     try {
         c c2(5);
         bar();
         return 0;
     }
     catch (int i) {
         printf("caught exception %d\n", i);
     }
}
```

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

```
(dbx) intercept
-unhandled -unexpected
(dbx) intercept int
<dbx> intercept
-unhandled -unexpected int
(dbx) stop in bar
(2) stop in bar()
(dbx)run
```

```
Running: a.out
(process id 304)
Stopped in bar at line 13 in file “foo.cc”
   13      c c1(3);
(dbx) whocatches int
int is caught at line 24, in function main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or caught)
(dbx) cont
Exception of type int is caught at line 24, in function main (frame number 4)
stopped in _exdbg_notify_of_throw at 0xef731494
0xef731494: _exdbg_notify_of_throw         :        jmp     %o7 + 0x8
Current function is bar
   14       throw(99);
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8         printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
    9       }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
    9       )
(dbx) step
stopped in main at line 24 in file "foo.cc"
   24          printf("caught exception %d\n", i);
(dbx) step
caught exception 99
stopped in main at line 26 in file "foo.cc"
   26   }
```

Note - The examples used in this section were built with the Oracle Developer Studio compilers. The examples would differ if compiling the code with `gcc`.

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:

Setting breakpoints at class or function template instantiations	See "`stop inclass` Command" on page 206, "`stop inclass` Command" on page 206, and "`stop in` Command" on page 207.
Printing a list of all class and function template instantiations	See "`whereis` Command" on page 205
Displaying the definitions of templates and instances	See "`whatis` Command" on page 205
Calling member template functions and function template instantiations	See "`call` Command" on page 207
Printing values of function template instantiations	See"`print` Expressions" on page 208
Displaying the source code for function template instantiations	See "`list` Expressions" on page 208

Template Example

The following code example shows the class template `Array` and its instantiations and the function template square and its instantiations.

```
template<class C> void square(C num, C *result)
{
     *result = num * num;
}

template<class T> class Array
{
public:
      int getlength(void)
     {
            return length;
      }

      T & operator[](int i)
      {
            return array[i];
     }

      Array(int l)
      {
            length = l;
            array = new T[length];
      }

      ~Array(void)
      {
            delete [] array;
      }
```

```

private:
      int length;
      T *array;
};

int main(void)
{
      int i, j = 3;
     square(j, &i);

     double d, e = 4.1;
      square(e, &d);

      Array<int> iarray(5);
      for (i = 0; i < iarray.getlength(); ++i)
      {
           iarray[i] = i;
     }

      Array<double> darray(5);
      for (i = 0; i < darray.getlength(); ++i)
     {
            darray[i] = i * 2.1;
      }

      return 0;
}
```

In the example:

- `Array` is a class template
- `square` is a function template
- `Array<int>` is a class template instantiation (template class)
- `Array<int>::getlength` is a member function of a template class
- `square(int, int*)` and `square(double, double*)` are function template instantiations (template functions)

Commands for C++ Templates

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

whereis Command

Use the `whereis` command to print a list of all occurrences of function or class instantiations for a function or class template.

For a class template:

```
(dbx) whereis Array
member function: `Array<int>::Array(int)
member function: `Array<double>::Array(int)
class template instance: `Array<int>
class template instance: `Array<double>
class template: `a.out`template_doc_2.cc`Array
```

For a function template:

```
(dbx) whereis square
function template instance: `square<int>(__type_0,__type_0*)
function template instance: `square<double>(__type_0,__type_0*)
function template: `a.out`template_doc_2.cc`square
```

The `__type_0` parameter refers to the 0th template parameter. A `__type_1` would refer to the next template parameter.

For more information, see "whereis Command" on page 205.

whatis Command

Use the `whatis` command to print the definitions of function and class templates and instantiated functions and classes.

For a class template:

```
(dbx) whatis -t Array
template<class T> class Array
To get the full template declaration, try `whatis -t Array<int>';
```

For the class template's constructors:

```
(dbx) whatis Array
More than one identifier 'Array'.
Select one of the following:
 0) Cancel
```

```
 1) Array<int>::Array(int)
 2) Array<double>::Array(int)
> 1
Array<int>::Array(int 1);
```

For a function template:

```
(dbx) whatis square
More than one identifier 'square'.
Select one of the following:
 0) Cancel
 1) square<int(__type_0,__type_0*)
 2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);
```

For a class template instantiation:

```
(dbx) whatis -t Array<double>
class Array<double> {
public:
    int Array<double>::getlength()
    double &Array<double>::operator [](int i);
    Array<double>::Array<double>(int l);
    Array<double>::~Array();
private:
    int length;
    double *array;
};
```

For a function template instantiation:

```
(dbx) whatis square(int, int*)
void square(int num, int *result);
```

For more information, see "whatis Command" on page 205.

stop inclass Command

To stop in all member functions of a template class:

```
(dbx)stop inclass Array
(2) stop inclass Array
```

Use the `stop inclass` command to set breakpoints at all member functions of a particular template class:

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

For more information, see “stop in Command” on page 207 and “inclass Event Specification” on page 277.

stop infunction Command

Use the `stop infunction` command to set breakpoints at all instances of the specified function template:

```
(dbx) stop infunction square
(9) stop infunction square
```

For more information, see “stop infunction Command” on page 207 and “infunction Event Specification” on page 276.

stop in Command

Use the `stop in` command to set a breakpoint at a member function of a template class or at a template function.

For a member of a class template instantiation:

```
(dbx) stop in Array<int>::Array(int l)
(2) stop in Array<int>::Array(int)
```

For a function instantiation:

```
(dbx) stop in square(double, double*)
(6) stop in square<double>(__type_0,__type_0*)
```

For more information, “stop in Command” on page 207 and “in Event Specification” on page 275.

call Command

Use the `call` command to explicitly call a function instantiation or a member function of a class template when you are stopped in scope. If `dbx` is unable to determine the correct instance, it displays a numbered list of instances from which you can choose.

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

For more information, see "call Command" on page 310.

print Expressions

Use the print command to evaluate a function instantiation or a member function of a class template.

```
(dbx) print iarray.getlength()
iarray.getlength() = 5
```

Use print to evaluate the this pointer.

```
(dbx) whatis this
class Array<int> *this;
(dbx) print *this
*this = {
    length = 5
    array   = 0x21608
}
```

For more information, see "print Command" on page 370.

list Expressions

Use the list command to print the source listing for the specified function instantiation.

```
(dbx) list square(int, int*)
```

For more information, see "list Command" on page 352.

CHAPTER 15

Debugging Fortran Using dbx

This chapter introduces dbx features you might use with Fortran. Sample requests to dbx are also included to provide you with assistance when debugging Fortran code using dbx.

This chapter includes the following topics:

- "Debugging Fortran" on page 209
- "Debugging Segmentation Faults" on page 213
- "Locating Exceptions" on page 214
- "Tracing Calls" on page 214
- "Working With Arrays" on page 215
- "Showing Intrinsic Functions" on page 217
- "Showing Complex Expressions" on page 218
- "Showing Logical Operators" on page 219
- "Viewing Fortran Derived Types" on page 220
- "Pointer to Fortran Derived Type" on page 221

Debugging Fortran

The following tips and general concepts are provided to help you while debugging Fortran programs. For information about debugging Fortran OpenMP code with dbx, see "Interacting With Events" on page 180.

Current Procedure and File

During a debug session, dbx defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, `stop at 5` sets different breakpoints, depending on which file is current.

Uppercase Letters

If your program has uppercase letters in any identifiers, dbx recognizes them. You need not provide case-sensitive or case-insensitive commands, as in some earlier versions.

Fortran and dbx must be in the same case-sensitive or case-insensitive mode:

- Compile and debug in case-insensitive mode without the -U option. The default value of the dbx input_case_sensitive environment variable is then false.

  If the source has a variable named LAST, then in dbx, both the print LAST or print last commands work. Fortran and dbx consider LAST and last to be the same, as requested.
- Compile and debug in case-sensitive mode using -U. The default value of the dbx input_case_sensitive environment variable is then true.

  If the source has a variable named LAST and one named last, then in dbx, print last works but print LAST does not work. Fortran and dbx distinguish between LAST and last, as requested.

Note - File or directory names are always case-sensitive in dbx, even if you have set the dbx input_case_sensitive environment variable to false.

Sample dbx Session

The following examples use a sample program called my_program.

Main program for debugging, a1.f:

```
PARAMETER ( n=2 )
REAL twobytwo(2,2) / 4 *-1 /
CALL mkidentity( twobytwo, n )
PRINT *, determinant( twobytwo )
END
```

Subroutine for debugging, a2.f:

```
SUBROUTINE mkidentity ( array, m )
REAL array(m,m)
DO 90 i = 1, m
    DO 20 j = 1, m
        IF ( i .EQ. j ) THEN
        array(i,j) = 1.
```

```
            ELSE
            array(i,j) = 0.
            END IF
20        CONTINUE
90    CONTINUE
    RETURN
    END
```

Function for debugging, `a3.f`:

```
    REAL FUNCTION determinant ( a )
    REAL a(2,2)
    determinant = a(1,1) * a(2,2) - a(1,2) * a(2,1)
    RETURN
    END
```

How to Run the Sample dbx Session

1. **Compile and link with the -g option.**

   You can do this in one or two steps.

   - **To compile and link in one step:**

   ```
   demo% f95 -o my_program -g a1.f a2.f a3.f
   ```

   - **To compile and link in separate steps:**

   ```
   demo% f95 -c -g a1.f a2.f a3.f
   demo% f95 -o my_program a1.o a2.o a3.o
   ```

2. **Start dbx on the executable named my_program.**

   ```
   demo% dbx my_program
   Reading symbolic information…
   ```

3. **Set a simple breakpoint.**

   To stop at the first executable statement in a main program.

   ```
   (dbx) stop in MAIN
   (2) stop in MAIN
   ```

   Although the main program `MAIN` must be all uppercase, the names of subroutines, functions, or block data subprogramas can be uppercase or lowercase.

4. **Run the program in the executable files named when you started dbx.**

```
(dbx) run
Running: my_program
stopped in MAIN at line 3 in file "a1.f"
    3         call mkidentity( twobytwo, n )
```

When the breakpoint is reached, `dbx` displays a message showing where it stopped, in this case, at line 3 of the `a1.f` file.

5. **Print a value.**

   Print the value of `n`:

```
(dbx) print n
n = 2
```

To print the matrix `twobytwo`, the format might vary:

```
(dbx) print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
```

Note that you cannot print the matrix array because `array` is not defined here, only in `mkidentity`.

6. **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.

7. **Quit `dbx`.**

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

Debugging Segmentation Faults

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

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a `REAL` argument, which the called routine has as `INTEGER`.
- An array index is miscalculated.
- The calling routine has fewer arguments than required.
- A pointer is used before it has been defined.

Using `dbx` to Locate Problems

Use `dbx` to find the source code line where a segmentation fault has occurred.

Use a program to generate a segmentation fault.

```
demo% cat WhereSEGV.f
    INTEGER a(5)
    j = 2000000
    DO 9 i = 1,5
        a(j) = (i * 10)
9    CONTINUE
    PRINT *, a
    END
demo%
```

Use `dbx` to find the line number of a `dbx` segmentation fault.

```
demo% f95 -g -silent WhereSEGV.f
demo% a.out
Segmentation fault
demo% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

Locating Exceptions

A program can throw an exception for many possible reasons. One approach to locating the problem is to find the line number in the source program where the exception occurred, and then examine that location.

Compiling with `-ftrap=common` forces trapping on all common exceptions.

To find where an exception occurred:

```
demo% cat wh.f
                call joe(r, s)
                print *, r/s
                end
                subroutine joe(r,s)
                r = 12.
                s = 0.
                return
                end
demo% f95 -g -o wh -ftrap=common wh.f
demo% dbx wh
Reading symbolic information for wh
(dbx) catch FPE
(dbx) run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in file “wh.f”
   2                    print *, r/s
(dbx)
```

Tracing Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that led it there. This sequence is called a *stack trace*.

The `where` command shows where in the program flow execution stopped and how execution reached this point, a *stack trace* of the called routines.

`ShowTrace.f` is a program written to get a core dump a few levels deep in the call sequence, to show a stack trace.

Note the reverse order:

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
```

```
MAIN called calc, calc called calcb.
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Execution stopped, line 23
Reading symbolic information for a.out
...
(dbx) run
calcB called from calc, line 9
Running: a.out
(process id 1089)
calc called from MAIN, line 3
signal SEGV (no mapping at the fault address) in calcb at line 23 in file "ShowTrace.f"
   23                  v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
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
```

For more information, see "Array Slicing Syntax for Fortran" on page 125.

Fortran Allocatable Arrays

The following example shows how to work with change to allocatable arrays in `dbx`.

```
demo% f95 -g Alloc.f95
  demo% dbx a.out
  (dbx) list 1,99
      1   PROGRAM TestAllocate
      2   INTEGER n, status
      3   INTEGER, ALLOCATABLE :: buffer(:)
      4           PRINT *, 'Size?'
      5           READ *, n
      6           ALLOCATE( buffer(n), STAT=status )
      7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
      8           buffer(n) = n
      9           PRINT *, buffer(n)
     10           DEALLOCATE( buffer, STAT=status)
     11   END
 (dbx) stop at 6
 (2) stop at "alloc.f95":6
 (dbx) stop at 9
 (3) stop at "alloc.f95":9
 (dbx) run
 Running: a.out
 (process id 10749)
```

```
 Size?
 1000
 stopped in main at line 6 in file "alloc.f95"
     6          ALLOCATE( buffer(n), STAT=status )
 (dbx) whatis buffer
 integer*4 , allocatable::buffer(:)
 (dbx) next
 continuing
 stopped in main at line 7 in file "alloc.f95"
     7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
 (dbx) whatis buffer
 integer*4 buffer(1:1000)
 (dbx) cont
 stopped in main at line 9 in file "alloc.f95"
     9          PRINT *, buffer(n)
 (dbx) print n
buffer(1000) holds 1000
 n = 1000
 (dbx) print buffer(n)
 buffer(n) = 1000
```

Showing Intrinsic Functions

`dbx` recognizes Fortran intrinsic functions (SPARC platforms and x86 platforms only).

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(i)
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% f95 -g ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

Showing Interval Expressions

To show an interval expression in dbx:

```
demo% cat ShowInterval.f95
```

```
   INTERVAL v
   v = [ 37.1, 38.6 ]
   END
demo% f95 -g -xia ShowInterval.f95
demo% dbx a.out
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: a.out
(process id 5217)
stopped in MAIN at line 2 in file "ShowInterval.f95"
    2      v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16  v
(dbx) print v
v = [0.0,0.0]
(dbx) next
stopped in MAIN at line 3 in file "ShowInterval.f95"
    3      END
(dbx) print v
v = [37.1,38.6]
(dbx) print v+[0.99,1.01]
v+[0.99,1.01] = [38.09,39.61]
(dbx) quit
demo%
```

Showing Logical Operators

`dbx` can locate Fortran logical operators and print them.

To show logical operators in `dbx`:

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo% f95 -g ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
```

```
    5           z = .false.
    6           END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4 y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

Viewing Fortran Derived Types

You can show structures, Fortran derived types, with `dbx`.

```
demo% f95 -g DebStruc.f95
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM Struct ! Debug a Structure
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7 REAL price
    8      END TYPE product
    9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Coffee Cup"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
(dbx) stop at 17
(2) stop at "Struct.f95":17
```

```
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f95"
   17      WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
    integer*4 id
    character*16 name
    character*8 model
    real*4 cost
    real*4 price
end type product
(dbx) n
(dbx) print prod1
    prod1 = (
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

Pointer to Fortran Derived Type

You can show structures, Fortran derived types, and pointers with `dbx`.

```
demo% f95 -o debstr -g DebStruc.f95
 demo% dbx debstr
 (dbx) stop in MAIN
 (2) stop in MAIN
 (dbx) list 1,99
     1   PROGRAM DebStruPtr! Debug structures & pointers
Declare a derived type.
     2      TYPE product
     3         INTEGER        id
     4         CHARACTER*16   name
     5         CHARACTER*8    model
     6         REAL           cost
     7         REAL           price
     8      END TYPE product
     9
Declare prod1  and prod2 targets.
    10      TYPE(product), TARGET :: prod1, prod2
```

Declare `curr` *and* `prior` *pointers.*

```
    11      TYPE(product), POINTER :: curr, prior
    12
```

Make `curr` *point to* `prod2`.

```
    13      curr => prod2
```

Make `prior` *point to* `prod1`.

```
    14      prior => prod1
```

Initialize `prior`.

```
    15      prior%id = 82
    16      prior%name = "Coffee Cup"
    17      prior%model = "XL"
    18      prior%cost = 24.0
    19      prior%price = 104.0
```

Set `curr` *to* `prior`.

```
    20      curr = prior
```

Print `name` *from* `curr` *and* `prior`.

```
    21      WRITE ( *, * ) curr%name, " ", prior%name
    22   END PROGRAM DebStruPtr
 (dbx) stop at 21
 (1) stop at "DebStruc.f95":21
 (dbx) run
 Running: debstr
(process id 10972)
stopped in main at line 21 in file "DebStruc.f95"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) print prod1
 prod1 = (
    id = 82
    name = "Coffee Cup"
    model = "XL"
    cost = 24.0
    price = 104.0
)
```

In the previous example, `dbx` displays all fields of the derived type, including field names.

You can use structures and inquire about an item of a Fortran derived type.

Ask about the variable

```
(dbx) whatis prod1
 product prod1
```

Ask about the type (`-t`*)*

```
 (dbx) whatis -t product
 type product
    integer*4 id
    character*16 name
    character*8 model
    real cost
    real price
```

```
 end type product
```

To print a pointer:

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
 )
```

Object Oriented Fortran

The Object Oriented Fortran features supported in `dbx` are type extension and polymorphic pointers, which is consistent with C++ support.

The `dbxenv` variables `output_dynamic_type` and `output_inherited_members` work with Fortran.

You can use the `-r`, `+r`, `-d`, and `+d` options with the `print` and `whatis` commands to get information about the inherited (parent) types and the dynamic types in Object Oriented Fortran code.

Allocatable Scalar Type

`dbx` supports the Fortran allocatable scalar type.

♦ ♦ ♦ CHAPTER 16

Debugging a Java Application With dbx

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

The chapter contains the following sections:

- "Using dbx With Java Code" on page 225
- "Environment Variables for Java Debugging" on page 226
- "Starting to Debug a Java Application" on page 226
- "Customizing Startup of the JVM Software" on page 231
- "dbx Modes for Debugging Java Code" on page 235
- "Using dbx Commands in Java Mode" on page 236

Using dbx With Java Code

You can use Oracle Developer Studio dbx to debug mixed code (Java code and C code or C++ code) running under the Oracle Solaris™ OS and the Linux OS.

Capabilities of dbx With Java Code

You can debug several types of Java applications with dbx. Most dbx commands operate similarly on native code and Java code.

Limitations of dbx With Java Code

dbx has the following limitations when debugging Java code:

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

- dbx cannot tell you the state of a Java application if the application is hung for some reason and dbx is not able to make procedure calls.
- Fix and continue, and runtime checking, do not apply to Java applications.

Environment Variables for Java Debugging

The following dbxenv variables are specific to debugging a Java application with dbx. You can set the JAVASRCPATH, CLASSPATHX, and jvm_invocation environment variables at a shell prompt before starting dbx or from the dbx command line. The setting of the jdbx_mode environment variable changes as you are debugging your application. You can change its setting with the jon command and the joff command.

jdbx_mode	The jdbx_mode dbxenv variable can have the following settings: java, jni, or native. For descriptions of the Java, JNI, and native modes, and how and when the mode changes, see "dbx Modes for Debugging Java Code" on page 235. Default: java.
JAVASRCPATH	You can use the JAVASRCPATH dbxenv variable to specify the directories in which dbx should look for Java source files. This variable is useful when the Java sources files are not in the same directory as the .class or .jar files. See "Specifying the Location of Your Java Source Files" on page 230 for more information.
CLASSPATHX	The CLASSPATHX dbxenv variable lets you specify to dbx a path for Java class files that are loaded by custom class loaders. For more information, see "Specifying a Path for Class Files That Use Custom Class Loaders" on page 230.
jvm_invocation	The jvm_invocation dbxenv variable lets you customize the way the JVM™ software is started. (The terms "Java virtual machine" and "JVM" mean a virtual machine for the Java platform.) For more information, see "Customizing Startup of the JVM Software" on page 231.

Starting to Debug a Java Application

You can use dbx to debug the following types of Java applications:

- A file with a file name that ends in .class
- A file with a file name that ends in .jar
- A Java application that is started using a wrapper
- A running Java application that was started in debug mode to which you attach dbx
- A C application or C++ application that embeds a Java application using the JNI_CreateJavaVM interface

`dbx` recognizes that it is debugging a Java application in all of these cases.

Debugging a Class File

If the class that defines the application is defined in a package, you need to include the package path just as when running the application under the JVM software, as in the following example.

```
(dbx) debug java.pkg.Toy.class
```

You can debug a file that uses the `.class` file name extension using `dbx`.You can also use a full path name for the class file. `dbx` automatically determines the package portion of the class path by looking in the `.class` file and adds the remaining portion of the full path name to the class path. For example, given the following path name, `dbx` determines that `pkg/Toy.class` is the main class name and adds `/home/user/java` to the class path.

```
(dbx) debug /home/user/java/pkg/Toy.class
```

Debugging a JAR File

A Java application can be bundled in a JAR (Java Archive) file. You can debug a JAR file using `dbx`. When you start debugging a file that has a file name ending in `.jar`, `dbx` uses the `Main-Class` attribute specified in the manifest of this JAR file to determine the main class. (The main class is the class within the JAR file that is your application's entry point.) If you use a full path name or relative path name to specify the JAR file, `dbx` uses the directory name and prefixes it to the class path in the `Main-Class` attribute.

If you debug a JAR file that does not have the `Main-Class` attribute, you can use the JAR URL syntax `jar:<url>!/{entry}` that is specified in the class `JarURLConnection` of the Java 2 Platform, Standard Edition to specify the name of the main class, as shown in the following examples.

```
(dbx) debug jar:myjar.jar!/myclass.class
(dbx) debug jar:/a/b/c/d/e.jar!/x/y/z.class
(dbx) debug jar:file:/a/b/c/d.jar!/myclass.class
```

For each of these examples `dbx` would do the following:

- Treat the class path specified after the `!` character as the main class (for example, `/myclass.class` or `/x/y/z.class`)

- Add the name of the JAR file (`./myjar.jar`, `/a/b/c/d/e.jar`, or `/a/b/c/d.jar`) to the class path
- Begin debugging the main class

Note - If you have specified a custom startup of the JVM software using the `jvm_invocation` environment variable (see "Customizing Startup of the JVM Software" on page 231), the file name of the JAR file is not automatically added to the class path. In this case, you must add the file name of the JAR file to the class path when you start debugging.

Debugging a Java Application That Has a Wrapper

A Java application usually has a wrapper to set environment variables. If your Java application has a wrapper, you need to tell `dbx` that a wrapper script is being used by setting the `jvm_invocation` environment variable. For more information, see "Customizing Startup of the JVM Software" on page 231.

Attaching `dbx` to a Running Java Application

You can attach `dbx` to a running Java application if you specified the options shown in the following example when you started the application. After starting the application, you would use the `dbx` command with the process ID of the running Java process to start debugging.

```
$ java -agentlib:dbx_agent myclass.class
$ dbx - 2345
```

For the JVM software to locate `libdbx_agent.so`, you need to add the appropriate path to `LD_LIBRARY_PATH` before running the Java application:

- 32-bit version of the JVM software on a system running the Oracle Solaris OS: add /*install-dir*/`SUNWspro/lib/libdbx_agent.so`
- 64-bit version of the JVM software on a SPARC based system running the Oracle Solaris OS: add /*install-dir*/`SUNWspro/lib/v9/libdbx_agent.so` to `LD_LIBRARY_PATH`
- 64-bit version of the JVM software on an x64 based system running the Linux OS: add/*install-dir*/`sunstudio12/lib/amd64/libdbx_agent.so` to `LD_LIBRARY_PATH`

install-dir is the location where the Oracle Developer Studio is installed.

When you attach `dbx` to the running application, `dbx` starts debugging the application in Java mode.

If your Java application requires 64-bit object libraries, include the `-d64` option when you start the application. Then when you attach `dbx` to the application, `dbx` will use the 64-bit JVM software on which the application is running.

```
$ java -agentlib:dbx_agent
$ dbx - 2345
```

The following task explains how to attach `dbx` to a specific Java process using a process ID.

▼ To Attach to a Running Java Process

1. **Ensure that the JVM™ software can find `libdbx_agent.so` by adding `libdbx_agent.so` to your `LD_LIBRARY_PATH` as explained in the previous section.**
2. **Start your Java application by typing:**

   `java -agentlib:dbx_agent` *myclass*`.class`
3. **Then you can attach to the process by starting `dbx` with the process ID:**

   `dbx` -*process-ID*

Debugging a C Application or C++ Application That Embeds a Java Application

You can debug a C application or C++ application that embeds a Java application using the `JNI_CreateJavaVM` interface. The C application or C++ application must start the Java application by specifying the following option to the JVM software:

```
-agentlib:dbx_agent
```

For the JVM software to locate `libdbx_agent.so`, you need to add the appropriate path to `LD_LIBRARY_PATH` before running the Java application. See "Attaching `dbx` to a Running Java Application" on page 228.

The *install-dir* is the location where the Oracle Developer Studio software is installed.

Passing Arguments to the JVM Software

When you use the `run` command in Java mode, the arguments you give are passed to the application and not to the JVM software. To pass arguments to the JVM software, see "Customizing Startup of the JVM Software" on page 231.

Specifying the Location of Your Java Source Files

Sometimes your Java source files are not in the same directory as the `.class` or `.jar` files. You can use the `$JAVASRCPATH` environment variable to specify the directories in which `dbx` should look for Java source files. The following example causes `dbx` to look in the listed directories for source files that correspond to the class files being debugged.

```
JAVASRCPATH=.:/mydir/mysrc:/mydir/mylibsrc:/mydir/myutils
```

Specifying the Location of Your C Source Files or C++ Source Files

`dbx` might not be able to find your C source files or C++ source files in the following circumstances:

- If your source files are not in the same location as they were when you compiled them
- If you compiled your source files on a different system than the one on which you are running `dbx` and the compile directory does not have the same path name

In such cases, use the `pathmap` command (see "`pathmap` Command" on page 368) to map one path name to another so that `dbx` can find your files.

Specifying a Path for Class Files That Use Custom Class Loaders

An application can have custom class loaders that load class files from locations that might not be part of the regular class path. In such situations `dbx` cannot locate the class files. The `CLASSPATHX` environment variable lets you specify to `dbx` a path for the class files that are loaded by their custom class loaders. For example, `CLASSPATHX=.:/myloader/myclass:/`

mydir/mycustom causes dbx to look in the listed directories when it is trying to locate a class file.

Setting Breakpoints on Java Methods

Unlike native applications, Java applications do not contain an easily accessible index of names. So, for example, you cannot simply specify a method name:

```
(dbx) stop in myMethod #This will not work
```

Instead, you need to use the full path to the method.

```
(dbx) stop in com.any.library.MyClass.myMethod
```

An exception is the case where you are stopped with some method of MyClass in which myMethod should be enough.

One way to avoid including the full path to the method is to use stop inmethod.

```
(dbx) stop inmethod myMethod
```

However, this command might cause stops in multiple methods name myMethod.

Setting Breakpoints in Native (JNI) Code

The shared libraries that contain JNI C or C++ code are dynamically loaded by the JVM and setting breakpoints in them requires some additional steps. For more information, see "Setting Breakpoints in Dynamically Loaded Libraries" on page 109.

Customizing Startup of the JVM Software

You might need to customize startup of the JVM software from dbx to do certain tasks. Common tasks involving customization include the following::

- "Specifying a Path Name for the JVM Software" on page 232
- "Passing Run Arguments to the JVM Software" on page 232
- "Specifying a Custom Wrapper for Your Java Application" on page 233
- "Specifying 64-bit JVM Software" on page 234

You can customize startup of the JVM software using the `jvm_invocation` environment variable. By default, when the `jvm_invocation` environment variable is not defined, `dbx` starts the JVM software as follows

```
java -agentlib:dbx_agent=sync=process-ID
```

When the `jvm_invocation` environment variable is defined, `dbx` uses the value of the variable to start the JVM software.

You must include the `-Xdebug` option in the definition of the `jvm_invocation` environment variable. `dbx` expands `-Xdebug` into the internal options `-Xdebug- Xnoagent -Xrundbxagent: sync`.

If you do not include the `-Xdebug` option in the definition, as in the following example, `dbx` issues an error message.

```
jvm_invocation="/set/java/javasoft/sparc-S2/jdk1.2/bin/java"
```

```
dbx: Value of `$jvm_invocation' must include an option to invoke the VM in debug mode
```

Specifying a Path Name for the JVM Software

By default, `dbx` starts the JVM software in your path if you do not specify a path name for the JVM software.

To specify a path name for the JVM software, set the `jvm_invocation` environment variable to the appropriate path name, as shown in the following example.

```
jvm_invocation="/myjava/java -Xdebug"
```

This setting causes `dbx` to start the JVM software as follows:

```
/myjava/java -agentlib:dbx_agent=sync
```

Passing Run Arguments to the JVM Software

To pass run arguments to the JVM software, set the `jvm_invocation` environment variable to start the JVM software with those arguments, as in the following example.

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

This example causes `dbx` to start the JVM software as follows:

```
java -agentlib:dbx_agent=sync= -Xms512 -Xmx1024 -Xcheck:jni
```

Specifying a Custom Wrapper for Your Java Application

A Java application can use a custom wrapper for startup. If your application uses a custom wrapper, you can use the `jvm_invocation` environment variable to specify the wrapper to be used, as shown in the following example.

```
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"
```

This example causes `dbx` to start the JVM software as follows:

```
/export/siva-a/forte4j/bin/forte4j.sh - -agentlib:dbx_agent=sync=process-ID
```

Using a Custom Wrapper That Accepts Command-Line Options

The following wrapper script (`xyz`) sets a few environment variables and accepts command line options.

```
#!/bin/sh
CPATH=/mydir/myclass:/mydir/myjar.jar; export CPATH
JARGS="-verbose:gc -verbose:jni -DXYZ=/mydir/xyz"
ARGS=
while [ $# -gt 0 ] ; do
    case "$1" in
        -userdir) shift; if [ $# -gt 0 ]
; then userdir=$1; fi;;
        -J*) jopt=`expr $1 : '-J<.*>'`
; JARGS="$JARGS '$jopt'";;
        *) ARGS="$ARGS '$1'" ;;
    esac
    shift
done
java $JARGS -cp $CPATH $ARGS
```

This script accepts some command-line options for the JVM software and the user application. For wrapper scripts of this form, you would set the `jvm_invocation` environment variable and start `dbx` as follows:

```
% jvm_invocation="xyz -J-Xdebug -J other-java-options"
% dbx myclass.class -Dide=visual
```

Using a Custom Wrapper That Does Not Accept Command-Line Options

The following wrapper script (xyz) sets a few environment variables and starts the JVM software, but does not accept any command-line options or a class name.

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
java <options> myclass
```

You could use such a script to debug a wrapper using dbx in one of two ways:

- Modify the script to start dbx from inside the wrapper script itself by adding the definition of the jvm_invocation variable to the script and starting dbx.

  ```
  #!/bin/sh
  CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
  ABC=/mydir/abc; export ABC
  jvm_invocation="java -Xdebug <options>"; export jvm_invocation
  dbx myclass.class
  ```

  Once you have made this modification, you could start the debugging session by running the script.
- Modify the script slightly to accept some command-line options as follows:

  ```
  #!/bin/sh
  CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
  ABC=/mydir/abc; export ABC
  JAVA_OPTIONS="$1 <options>"
  java $JAVA_OPTIONS $2
  ```

  Once you make this modification, you would set the jvm_invocation environment variable and start dbx as follows:

  ```
  % jvm_invocation="xyz -Xdebug"; export jvm_invocation
  % dbx myclass.class
  ```

Specifying 64-bit JVM Software

If you want dbx to start 64-bit JVM software to debug an application that requires 64-bit object libraries, include the -d64 option when you set the jvm_invocation environment variable.

```
jvm_invocation="/myjava/java -Xdebug -d64"
```

dbx Modes for Debugging Java Code

When debugging a Java application, dbx is in one of three modes:

- Java mode
- JNI mode
- Native mode

When dbx is in Java mode or JNI (Java Native Interface) mode, you can inspect the state of your Java application, including JNI code, and control execution of the code. When dbx is in native mode, you can inspect the state of your C or C++ JNI code. The current mode (java, jni, or native) is stored in the environment variable jdbx_mode.

In Java mode, you interact with dbx using Java syntax and dbx uses Java syntax to present information to you. This mode is used for debugging pure Java code, or the Java code in an application that is a mixture of Java code and C JNI code or C++ JNI code.

In JNI mode, dbx commands use native syntax and affect native code, but the output of commands shows Java-related status as well as native status, so JNI mode is a “mixed” mode. This mode is used for debugging the native parts of an application that is a mixture of Java code and C JNI code or C++ JNI code.

In native mode, dbx commands affect only a native program, and all features related to Java are disabled. This mode is used for debugging non-Java related programs.

As you execute your Java application, dbx switches automatically between Java mode and JNI mode as appropriate. For example, when it encounters a Java breakpoint, dbx switches into Java mode, and when you step from Java code into JNI code, it switches into JNI mode.

Switching From Java or JNI Mode to Native Mode

dbx does not switch automatically into native mode. You can switch explicitly from Java or JNI Mode to native mode with the joff command, and from native mode to Java mode with the jon command.

Switching Modes When You Interrupt Execution

If you interrupt execution of your Java application (for example, by typing control-C), `dbx` tries to set the mode automatically to Java/JNI mode by bringing the application to a safe state and suspending all threads.

If `dbx` cannot suspend the application and switch to Java/JNI mode, `dbx` switches to native mode. You can then use the `jon` command to switch to Java mode so that you can inspect the state of the program.

Using `dbx` Commands in Java Mode

When you are using `dbx` to debug a mixture of Java and native code, `dbx` commands fall into several categories:

- Commands that accept the same arguments and operate the same way in Java mode or JNI mode as in native mode. See "Commands With Identical Syntax and Functionality in Java Mode and Native Mode" on page 238.
- Commands that have arguments that are valid only in Java mode or JNI mode, as well as arguments that are valid only in native mode. See "Commands With Different Syntax in Java Mode" on page 239.
- Commands that are valid only in Java mode or JNI mode. See "Commands Valid Only in Java Mode" on page 240.

Any commands not included in one of these categories work only in native mode.

Java Expression Evaluation in `dbx` Commands

The Java expression evaluator used in most `dbx` commands supports the following constructs:

- All literals
- All names and field accesses
- `this` and `super`
- Array accesses
- Casts

- Conditional binary operations
- Method calls
- Other unary/binary operations
- Assignment to variables or fields
- `instanceof` operator
- Array length operator

The Java expression evaluator does not support the following constructs:

- Qualified `this`, for example, <*ClassName*>`.this`
- Class instance creation expressions
- Array creation expressions
- String concatenation operator
- Conditional operator ? :
- Compound assignment operators, for example `x += 3`

A particularly useful way of inspecting the state of your Java application is using the watch facility in the IDE or `dbxtool`.

Do not depend on precise value semantics in expressions that do more than just inspect data.

Static and Dynamic Information Used by `dbx` Commands

Much of the information about a Java application is normally available only after the JVM software has started, and is unavailable after the Java application has finished executing. However, when you debug a Java application with `dbx`, `dbx` gleans some of the information it needs from class files and JAR files that are part of the system class path and user class path before it starts the JVM software. This information enables `dbx` to do better error checking on breakpoints before you run the application.

Some Java classes and their attributes might not be accessible through the class path. `dbx` can inspect and step through these classes, and the expression parser can access them once they are loaded at runtime. However, the information it gathers is temporary and is no longer available after the JVM software terminates.

Some information that `dbx` needs to debug your Java application is not recorded anywhere so `dbx` skims Java source files to derive this information as it is debugging your code.

Commands With Identical Syntax and Functionality in Java Mode and Native Mode

The dbx commands listed in the following table have the same syntax and perform the same operations in Java mode as in native mode.

Command	Functionality
attach	Attaches dbx to a running process, stopping execution and putting the program under debugging control
cont	Causes the process to continue execution
dbxenv	List or set dbxenv variables
delete	Deletes breakpoints and other events
down	Moves down the call stack (away from main)
dump	Prints all variables local to a procedure or method
file	Lists or changes the current file
frame	Lists or changes the current stack frame number
handler	Modifies event handlers (breakpoints)
import	Imports commands from a dbx command library
line	Lists or changes the current line number
list	Displays lines of a source file
next	Steps one source line (steps over calls)
pathmap	Maps one path name to another for finding source files and the like
proc	Displays the status of the current process
prog	Manages programs being debugged and their attributes
quit	Exits dbx
rerun	Runs the program with no arguments
runargs	Changes the arguments of the target process
status	Lists the event handlers (breakpoints)
step up	Steps up and out of the current function or method
stepi	Steps one machine instruction (steps into calls)
up	Moves up the call stack (toward main)
whereami	Displays the current source line

Commands With Different Syntax in Java Mode

The dbx commands listed in the following table have different syntax for Java debugging than for native code debugging and operate differently in Java mode than in native mode.

Command	Native Mode Functionality	Java Mode Functionality
assign	Assigns a new value to a program variable	Assigns a new value to a local variable or parameter
call	Calls a procedure	Calls a method
dbx	Starts dbx	Starts dbx
debug	Loads the specified application and begins debugging the application	Loads the specified Java application, checks for the existence of the class file, and begins debugging the application
detach	Releases the target process from dbx's control	Releases the target process from dbx's control
display	Evaluates and prints expressions at every stopping point	Evaluates and prints expressions, local variables, or parameters at every stopping point
files	Lists file names that match a regular expression	Lists all of the Java source files known to dbx
func	Lists or changes the current function	Lists or changes the current method
next	Steps one source line (stepping over calls)	Steps one source line (stepping over calls)
print	Prints the value of an expression	Prints the value of an expression, local variable, or parameter
run	Runs the program with arguments	Runs the program with arguments
step	Steps one source line or statement (stepping into calls)	Steps one source line or statement (stepping into calls)
stop	Sets a source-level breakpoint	Sets a source-level breakpoint
thread	Lists or changes the current thread	Lists or changes the current thread
threads	Lists all threads	Lists all threads
trace	Shows executed source lines, function calls, or variable changes	Shows executed source lines, function calls, or variable changes
undisplay	Undoes display commands	Undoes display commands
whatis	Prints the type of expression or declaration of type	Prints the declaration of an identifier
when	Executes commands when a specified event occurs	Executes commands when a specified event occurs
where	Prints the call stack	Prints the call stack

Commands Valid Only in Java Mode

The dbx commands listed in the following table are valid only in Java mode or JNI mode.

Command	Functionality
java	Used when dbx is in JNI mode to indicate that the Java version of a specified command is to be executed
jclasses	Prints the names of all Java classes known to dbx when you give the command
joff	Switches dbx from Java mode or JNI mode to native mode
jon	Switches dbx from native mode to Java mode
jpkgs	Prints the names of all Java packages known to dbx when you give the command
native	Used when dbx is in Java mode to indicate that the native version of a specified command is to be executed

♦ ♦ ♦ CHAPTER 17

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.

This chapter contains the following sections:

- "Using `dbx` at the Machine-Instruction Level" on page 241
- "Examining the Contents of Memory" on page 241
- "Stepping and Tracing at Machine-Instruction Level" on page 246
- "Setting Breakpoints at the Machine-Instruction Level" on page 248
- "Using the `regs` Command" on page 248

Using `dbx` at the Machine-Instruction Level

The `next` command, `step` command, `stop` command, and `trace` command each support a machine-instruction level variant: the `nexti` command, `stepi` command, `stopi` command, and `tracei` command. Use the `regs` command to print out the contents of machine registers or the `print` command to print out individual registers.

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 following:

- The *address*, using the `=` (equal sign) character
- The *contents* stored at an address, using the `/` (slash) character

You can print the assembly commands using the `dis` command and the `listi` command.

Using the `examine` or `x` Command

Use the `examine` command, or its alias `x`, to display memory contents or addresses.

Use the following syntax to display the contents of memory starting at *address* for *count* items in format *format*. The default *address* is the next one after the last address previously displayed. The default *count* is 1. The default *format* is the same as was used in the previous `examine` command, or `X` if this is the first command given.

The syntax for the `examine` command is:

```
examine [address] [/ [count] [format]]
```

To display the contents of memory from *address1* through *address2* inclusive in format *format*:

```
examine address1, address2 [/ [format]]
```

To display the address, instead of the contents of the address in the given format:

```
examine address = [format]
```

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, provide the expression as an address.

```
examine address=format
examine address=
```

Using Addresses

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

The following examples are valid addresses:

`0xff00`	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 ($).

Using Formats

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

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

i	Display as an assembly instruction
d	Display as 16 bits (2 bytes) in decimal
D	Display as 32 bits (4 bytes) in decimal
o	Display as 16 bits (2 bytes) in octal
O	Display as 32 bits (4 bytes) in octal
x	Display as 16 bits (2 bytes) in hexadecimal
X	Display as 32 bits (4 bytes) in hexadecimal (default format)
b	Display as a byte in octal
c	Display as a character
n	Display as a decimal (1 byte).
w	Display as a wide character
s	Display as a string of characters terminated by a null byte
W	Display as a wide character string
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

Using Count

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

Examples of Using an Address

The following examples show how to use an address with and format options to display five successive disassembled instructions starting from the current stopping point.

For SPARC based systems:

```
(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 x86 based systems:

```
(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.

The syntax for the `dis` command is:

`dis [`<*address*>`] [/`<*count*>`] | `<*address1*>`, `<*address1*>

The `dis` command operates as follows:

- Without arguments displays 10 instructions starting at *address*

- With the *address* argument only, disassembles 10 instructions starting at *address*
- With the *address* argument and a *count*, disassembles *count* instructions starting at *address*
- With the *address1* and *address2* arguments, disassembles instructions from *address1* through *address2*
- With only a *count*, displays *count* instructions starting at +
- With the option -a, disassembles entire function or, when used without parameters, disassembles remains of current visiting function

Using the `listi` Command

To display source lines with their corresponding assembly instructions, use the `listi` command, which is equivalent to the command `list -i`. See the discussion of `list -i` in "Printing a Source Listing" on page 69.

SPARC based systems example:

```
(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]
```

x86 based systems example:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
```

```
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

Stepping and Tracing at Machine-Instruction Level

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

Single-Stepping at the Machine-Instruction Level

To single-step from one machine instruction to the next machine instruction, use the `nexti` command or the `stepi` command

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

The output from the `nexti` command and the `stepi` command differ from the corresponding source level commands in two ways:

- The output includes the address of the instruction at which the program is stopped (instead of the source code line number).
- The default output contains the disassembled instruction instead of the source code line.

For example:

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

For more information, see "`nexti` Command" on page 365 and "`stepi` Command" on page 386.

Tracing at the Machine-Instruction Level

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

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

For more information on `trace` and its event specifications and modifiers, see "Tracing Execution" on page 108 and "`tracei` Command" on page 404.

The general syntax for the `tracei` command is:

`tracei` *event-specification* `[`*modifier*`]`

Commonly used forms of the `tracei` command are:

`tracei step`	Trace each instruction
`tracei next`	Trace each instruction, but skip over calls
`tracei at` *address*	Trace the given code address.

For more information, see "`tracei` Command" on page 404.

For SPARC:

```
(dbx) tracei next -in main
(dbx) cont
0x00010814: main+0x0004:  clr     %l0
0x00010818: main+0x0008:  st      %l0, [%fp - 0x8]
0x0001081c: main+0x000c:  call    foo
0x00010820: main+0x0010:  nop
0x00010824: main+0x0014:  clr     %l0
....
....
(dbx) (dbx) tracei step -in foo -if glob == 0
(dbx) cont
0x000107dc: foo+0x0004:  mov     0x2, %l1
0x000107e0: foo+0x0008:  sethi   %hi(0x20800), %l0
0x000107e4: foo+0x000c:  or      %l0, 0x1f4, %l0     ! glob
```

```
0x000107e8: foo+0x0010:  st      %l1, [%l0]
0x000107ec: foo+0x0014:  ba      foo+0x1c
....
....
```

Setting Breakpoints at the Machine-Instruction Level

To set a breakpoint at the machine-instruction level, use the `stopi` command. The command accepts any event specification. The syntax for the `stopi` command is:

`stopi` *event-specification* `[`*modifier*`]`

Commonly used forms of the `stopi` command are:

`stopi [at` *address*`] [-if` *cond*`]`
`stopi in` *function* `[-if` *cond*`]`

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

Setting a Breakpoint at an Address

Use the `stopi` command 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 `regs` Command

The `regs` command enables you to print the value of all the registers.

The syntax for the `regs` command is:

```
regs [-f][-F]
```

`-f` includes floating-point registers (single precision). `-F` includes floating-point registers (double precision).

For more information, see "regs Command" on page 375.

SPARC based systems example:

```
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
...
```

For x64 based systems example:

```
(dbx) regs
current frame:  [1]
r15     0x0000000000000000
r14     0x0000000000000000
r13     0x0000000000000000
r12     0x0000000000000000
r11     0x0000000000401b58
r10     0x0000000000000000
r9      0x0000000000401c30
r8      0x0000000000416cf0
rdi     0x0000000000416cf0
rsi     0x0000000000401c18
rbp     0xfffffd7fffdff820
rbx     0xfffffd7fff3fb190
rdx     0x0000000000401b50
rcx     0x0000000000401b54
rax     0x0000000000416cf0
trapno  0x0000000000000003
err     0x0000000000000000
```

```
rip     0x0000000000401709:main+0xf9    movl $0x0000000000000000,0xfffffffffffffffc(%
rbp)
cs      0x000000000000004b
eflags  0x0000000000000206
rsp     0xfffffd7fffdff7b0
ss      0x0000000000000043
fs      0x00000000000001bb
gs      0x0000000000000000
es      0x0000000000000000
ds      0x0000000000000000
fs_base  0xfffffd7fff3a2000
gsbase  0xffffffff80000000
(dbx) regs -F
current frame:  [1]
r15     0x0000000000000000
r14     0x0000000000000000
r13     0x0000000000000000
r12     0x0000000000000000
r11     0x0000000000401b58
r10     0x0000000000000000
r9      0x0000000000401c30
r8      0x0000000000416cf0
rdi     0x0000000000416cf0
rsi     0x0000000000401c18
rbp     0xfffffd7fffdff820
rbx     0xfffffd7fff3fb190
rdx     0x0000000000401b50
rcx     0x0000000000401b54
rax     0x0000000000416cf0
trapno  0x0000000000000003
err     0x0000000000000000
rip     0x0000000000401709:main+0xf9    movl     $0x0000000000000000,0xfffffffffffffffc
(%rbp)
cs      0x000000000000004b
eflags  0x0000000000000206
rsp     0xfffffd7fffdff7b0
ss      0x0000000000000043
fs      0x00000000000001bb
gs      0x0000000000000000
es      0x0000000000000000
ds      0x0000000000000000
fs_base  0xfffffd7fff3a2000
gsbase  0xffffffff80000000
st0     +0.00000000000000000000e+00
st1     +0.00000000000000000000e+00
st2     +0.00000000000000000000e+00
st3     +0.00000000000000000000e+00
st4     +0.00000000000000000000e+00
```

```
st5     +0.00000000000000000000e+00
st6     +0.00000000000000000000e+00
st7     +NaN
xmm0a-xmm0d     0x00000000 0xfff80000 0x00000000 0x00000000
xmm1a-xmm1d     0x00000000 0x00000000 0x00000000 0x00000000
xmm2a-xmm2d     0x00000000 0x00000000 0x00000000 0x00000000
xmm3a-xmm3d     0x00000000 0x00000000 0x00000000 0x00000000
xmm4a-xmm4d     0x00000000 0x00000000 0x00000000 0x00000000
xmm5a-xmm5d     0x00000000 0x00000000 0x00000000 0x00000000
xmm6a-xmm6d     0x00000000 0x00000000 0x00000000 0x00000000
xmm7a-xmm7d     0x00000000 0x00000000 0x00000000 0x00000000
xmm8a-xmm8d     0x00000000 0x00000000 0x00000000 0x00000000
xmm9a-xmm9d     0x00000000 0x00000000 0x00000000 0x00000000
xmm10a-xmm10d   0x00000000 0x00000000 0x00000000 0x00000000
xmm11a-xmm11d   0x00000000 0x00000000 0x00000000 0x00000000
xmm12a-xmm12d   0x00000000 0x00000000 0x00000000 0x00000000
xmm13a-xmm13d   0x00000000 0x00000000 0x00000000 0x00000000
xmm14a-xmm14d   0x00000000 0x00000000 0x00000000 0x00000000
xmm15a-xmm15d   0x00000000 0x00000000 0x00000000 0x00000000
fcw-fsw  0x137f 0x0000
fctw-fop        0x0000 0x0000
frip     0x0000000000000000
frdp     0x0000000000000000
mxcsr    0x00001f80
mxcr_mask       0x0000ffff
(dbx)
```

Platform-Specific Registers

The tables in this section list platform-specific register names for SPARC architecture, x86 architecture, and AMD64 architecture that can be used in expressions.

SPARC Register Information

The following table lists register information for SPARC architecture.

Register	Description
`$g0 through $g7`	Global registers
`$o0 through $o7`	“out” registers
`$l0 through $l7`	“local” registers
`$i0 through $i7`	“in” registers

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 quad floating-point registers are treated as having C long double type, They are available on SPARC V9 hardware:

```
$q0 $q4 through $q60
```

The following pairs of registers, which combine the least significant 32 bits of two registers, are available on SPARC V8+ hardware:

```
$g0g1 through $g6g7
$o0o1 through $o6o7
```

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

```
$xg0 through $xg7
$xo0 through $xo7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

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

x86 Register Information

The following table lists register information for x86 architecture.

Register	Description
$gs	Alternate data segment register
$fs	Alternate data segment register
$es	Alternate data segment register
$ds	Data segment register
$edi	Destination index register
$esi	Source index register
$ebp	Frame pointer
$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
$ss	Stack segment register

Commonly used registers are also aliased to their machine independent names.

Register	Description
$sp	Stack pointer; equivalent of $uesp
$pc	Program counter; equivalent of $eip
$fp	Frame pointer; equivalent of $ebp
$ps	Processor status register

The following table lists registers for the 80386 lower halves (16 bits).

Register	Description
$ax	General register
$cx	General register
$dx	General register
$bx	General register

Register	Description
`$si`	Source index register
`$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, as shown in the following table:

Register	Description
`$al`	Lower (right) half of register `$ax`
`$ah`	Higher (left) half of register `$ax`
`$cl`	Lower (right) half of register `$cx`
`$ch`	Higher (left) half of register `$cx`
`$dl`	Lower (right) half of register `$dx`
`$dh`	Higher (left) half of register `$dx`
`$bl`	Lower (right) half of register `$bx`
`$bh`	Higher (left) half of register `$bx`

The following table lists registers for 80387 halves:.

Register	Description
`$fctrl`	Control register
`$fstat`	Status register
`$ftag`	Tag register
`$fip`	Instruction pointer offset
`$fcs`	Code segment selector
`$fopoff`	Operand pointer offset
`$fopsel`	Operand pointer selector
`$st0 through $st7`	Data registers

AMD64 Register Information

The following table lists register information for AMD64 architecture:

Register	Description
rax	General purpose register - argument passing for function calls
rbp	General purpose register - stack management/frame pointer
rbx	General purpose register - callee-saved
rcx	General purpose register - argument passing for function calls
rdx	General purpose register - argument passing for function calls
rsi	General purpose register - argument passing for function calls
rdi	General purpose register - argument passing for function calls
rsp	General purpose register - stack management/stack pointer
r8	General purpose register - argument passing for function calls
r9	General purpose register - argument passing for function calls
r10	General purpose register - temporary
r11	General purpose register - temporary
r12	General purpose register - callee-saved
r13	General purpose register - callee-saved
r14	General purpose register - callee-saved
r15	General purpose register - callee-saved
rflags	Flags register
rip	Instruction pointer
mmx0/st0	64-bit media and floating-point register
mmx1/st1	64-bit media and floating-point register
mmx2/st2	64-bit media and floating-point register
mmx3/st3	64-bit media and floating-point register
mmx4/st4	64-bit media and floating-point register
mmx5/st5	64-bit media and floating-point register
mmx6/st6	64-bit media and floating-point register
mmx7/st7	64-bit media and floating-point register
xmm0	128-bit media register
xmm1	128-bit media register
xmm2	128-bit media register
xmm3	128-bit media register
xmm4	128-bit media register
xmm5	128-bit media register
xmm6	128-bit media register
xmm7	128-bit media register

Register	Description
xmm8	128-bit media register
xmm9	128-bit media register
xmm10	128-bit media register
xmm11	128-bit media register
xmm12	128-bit media register
xmm13	128-bit media register
xmm14	128-bit media register
xmm15	128-bit media register
cs	Segment register
es	Segment register
fs	Segment register
gs	Segment register
os	Segment register
ss	Segment register
fcw	fxsave and fxstor memory image control word
fsw	fxsave and fxstor memory image status word
fctw	fxsave and fxstor memory image tag word
fop	fxsave and fxstor memory image last x87 op code
frdp	fxsave and fxstor memory image 64-bit offset into the date segment
frip	fxsave and fxstor memory image 64-bit offset into the code segment
mxcsr_mask	set bits in mxcsr_mask indicate supported feature bits in mxcsr
ymmo	256–bit advanced vector register
ymm1	256–bit advanced vector register
ymm2	256–bit advanced vector register
ymm3	256–bit advanced vector register
ymm4	256–bit advanced vector register
ymm5	256–bit advanced vector register
ymm6	256–bit advanced vector register
ymm7	256–bit advanced vector register
ymm8	256–bit advanced vector register
ymm9	256–bit advanced vector register
ymm10	256–bit advanced vector register
ymm11	256–bit advanced vector register
ymm12	256–bit advanced vector register

Register	Description
ymm13	256–bit advanced vector register
ymm14	256–bit advanced vector register
ymm15	256–bit advanced vector register

The fields of an advanced vector (AVX) register (ymm0 through ymm15) can be treated as having C int, float, or double types.

CHAPTER 18

18

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 describes the differences between ksh-88 and dbx command language.

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

This chapter contains the following sections:

- "ksh-88 Features Not Implemented" on page 259
- "Extensions to ksh-88" on page 260
- "Renamed Commands" on page 260

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` options: `allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset -l -u -L -R -H` attributes
- Backquote (`\Q…\Q`) for command substitution (use `$(...)` instead)
- [[*expression*]] compound command for expression evaluation
- `@(pattern[|pattern] …)` extended pattern matching
- Co-processes (command or pipeline running in the background that communicates with your program)

Extensions to ksh-88

dbx adds the following features as extensions:

- `$[ p– > flags ]` language expression
- `typeset -q` enables special quoting for user-defined functions
- C shell-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 `dbxkill` command.
- The `alias` command is the `ksh` `alias` command, unless in `dbx` compatibility mode.
- *address/format* is now `examine` *address/format*.
- /*pattern* is now `search` *pattern*.
- ?*pattern* is now `bsearch` *pattern*.

Rebinding of Editing Functions

The `bind` command enables you to rebind editing functions. You can use the command to display or modify the key bindings for EMacs-style editors and vi-style editors. The syntax of the `bind` command is:

`bind`	Display the current editing key bindings

`bind` *key*=*definition*	Bind *key* to *definition*
`bind` *key*	Display the current definition for *key*
`bind` *key*=	Remove binding of *key*
`bind -m` *key*=*definition*	Define *key* to be a macro with *definition*
`bind -m`	Same as `bind`

where:

key is the name of a key.

definition is the definition of the macro to be bound to the key.

Some of the more important default key bindings for EMacs-style editors are:

^A = beginning-of-line	^B = backward-char
^D = eot-or-delete	^E = end-of-line
^F = forward-char	^G = abort
^K = kill-to-eo	^L = redraw
^N = down-history	^P = up-history
^R = search-history	^^ = quote
^? = delete-char-backward	^H = delete-char-backward
^[b = backward-word	^[d = delete-word-forward
^[f = forward-word	^[^H = delete-word-backward
^[^[= complete	^[? = list-command

Some of the more important default key bindings for vi-style editors are:

a = append	A = append at EOL
c = change	d = delete
G = go to line	h = backward character
i = insert	I = insert at BOL
j = next line	k = previous line
l = forward line	n = next match
N = prev match	p = put after
P = put before	r = repeat
R = replace	s = substitute
u = undo	x = delete character

X = delete previous character	y = yank
~ = transpose case	_ = last argument
* = expand	= = list expansion
- = previous line	+ = next line
sp = forward char	# = comment out command
? = search history from beginning	
/ = search history from current	

In insert mode, the following keystrokes are special:

^? = delete character	^H = delete character
^U = kill line	^W = delete word

♦ ♦ ♦ CHAPTER 19

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 contains the following sections:

- "Dynamic Linker" on page 263
- "Setting Breakpoints in Shared Libraries" on page 264
- "Setting a Breakpoint in an Explicitly Loaded Library" on page 265

Dynamic Linker

The dynamic linker, also known as rtld, Runtime ld, or ld.so, arranges to bring shared objects (load objects) into an executing application. The two primary areas where rtld is active are:

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

dbx uses the term *load object* to refer to a shared object (.so) or executable (a.out). You can use the loadobject command to list and manage symbolic information from load objects.

Link Map

The dynamic linker maintains a list of all loaded objects in a list called a *link map*. The link map is maintained in the memory of the program being debugged, and is indirectly accessed through librtld_db.so, a special system library for use by debuggers.

Startup Sequence and .init Sections

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

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. For more information, see "syncrtld Event Specification" on page 288.

Procedure Linkage Tables

Procedure linkage tables (PLTs) are structures used by the rtld to facilitate calls across shared object boundaries. For instance, calls to printf go through this indirect table. For details, see the generic and processor-specific SVR4 ABI reference manuals.

For dbx to handle step and next commands across PLTs, it has to keep track of the PLT table of each load object. The table information is acquired at the same time as the rtld handshake.

Setting Breakpoints in Shared Libraries

To set a breakpoint in a shared library, dbx needs to confirm that a program will use that library when it runs, and dbx needs to load the symbol table for the library. To determine which libraries a newly loaded program will use when it runs, dbx executes the program just long enough for the runtime linker to load all of the starting libraries. dbx then reads the list of loaded libraries and kills the process. The libraries remain loaded and you can set breakpoints in them before rerunning the program for debugging.

dbx follows the same procedure for loading the libraries regardless of whether the program is loaded from the command line with the dbx command, from the dbx prompt with the debug command, or in the IDE.

Setting a Breakpoint in an Explicitly Loaded Library

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

If a shared object is unloaded using dlclose(), dbx remembers the breakpoints placed in it and replaces them if the shared object is again loaded with dlopen(), even if the application is run again.

However, you do not need to wait for the loading of a shared object with dlopen() to place a breakpoint in it, or to navigate its functions and source code. If you know the name of the shared object that the program being debugged will be loading with dlopen(), you can request that dbx preload its symbol table by using the loadobject -load command:

```
loadobject -load /usr/java1.1/lib/libjava_g.so
```

You can now navigate the modules and functions in this load object and place breakpoints in it before it has been loaded with dlopen(). Once the load object is loaded by your program, dbx automatically places the breakpoints.

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

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

♦ ♦ ♦ APPENDIX A

Modifying a Program State

This appendix focuses on dbx usage and commands that change your program or change the behavior of your program when you run it under dbx, as compared to running it without dbx. Understanding which commands might make modifications to your program is important.

The chapter contains the following sections:

- "Impacts of Running a Program Under dbx" on page 267
- "Commands That Alter the State of the Program" on page 268

Impacts of Running a Program Under dbx

You use dbx to observe a process, and the observation should not affect the process. However, on occasion, you might drastically modify the state of the process. Sometimes plain observation can affect execution and cause intermittent bug symptoms.

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

- You might have forgotten to take out a -C or disable RTC. Having the RTC support library librtc.so loaded into a program can cause the program to behave differently.
- Your dbx initialization scripts might have some environment variables set that you have forgotten. The stack base starts at a different address when running under dbx. The address might also different based on your environment and the contents of argv[], forcing local variables to be allocated differently. If the variables are not initialized, they will produce different random numbers. This problem can be detected using runtime checking.
- The program does not initialize memory allocated with malloc() before use. This problem can be detected using runtime checking.
- dbx has to catch LWP creation and dlopen events, which might affect timing-sensitive multithreaded applications.

- dbx does context switching on signals so if your application makes heavy use of signals, things might work differently.
- Your program might be expecting that mmap() always returns the same base address for mapped segments. Running under dbx affects the address space sufficiently that mmap() is unlikely to return the same address as when the program is run without dbx. To determine if this is a problem, look at all uses of mmap() and ensure that the address returned is used by the program, rather than a hard-coded address.
- If the program is multithreaded, it might contain data races or be otherwise dependent upon thread scheduling. Running under dbx perturbs thread scheduling and might cause the program to execute threads in a different order than normal. To detect such conditions, use lock_lint.

Otherwise, determine whether running with adb or truss causes the same problems.

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

Commands That Alter the State of the Program

The commands described in this section might make modifications to your program.

assign Command

The assign command assigns the value of *expression* to *variable*. Using it in dbx permanently alters the value of *variable*.

assign *variable* = *expression*

pop Command

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

pop	Pop current frame.
pop *number*	Pop *number* frames.
pop -f *number*	Pop frames until specified frame *number*.

Any calls popped are re-executed upon resumption, which might result in unwanted program changes. `pop` also calls destructors for objects local to the popped functions.

For more information, see "`pop` Command" on page 370.

`call` Command

When you use the `call` command in `dbx`, you call a procedure and the procedure performs as specified:

`call` *proc*`([params])`

The procedure could modify your program. `dbx` makes the call as if you had written it into your program source.

For more information, see "`call` Command" on page 310.

`print` Command

To print the value of the expressions, type:

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

If an expression has a function call, printing the expression causes the call command to execute. Therefore, the same considerations apply as with the "`call` Command" on page 310. With C ++, you should also be careful of unexpected side effects caused by overloaded operators.

For more information, see "`print` Command" on page 370.

`when` Command

The general syntax of the `when` command is as follows:

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

When the event occurs, the commands are executed. Depending upon which command is issued, this action could alter your program state.

For more information, see "`when` Command" on page 413.

`fix` Command

You can use the `fix` command to make immediate changes to your program.

Although it is a very useful tool, the `fix` command recompiles modified source files and dynamically links the modified functions into the application.

Note that the fix and continue feature isn't supported on Intel Linux or SPARC Linux. Make sure to check for other restrictions for fix and continue. See "Memory Leak (`mel`) Error" on page 170.

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

`cont at` Command

The `cont at` command alters the order in which the program runs. Execution is continued at line *line*. The ID is required if the program is multithreaded.

`cont at` *line* `[` *ID* `]`

This command could change the outcome of the program.

♦ ♦ ♦ APPENDIX B

Event Management

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

This appendix contains the following sections:

- "Event Handlers" on page 271
- "Creating Event Handlers" on page 272
- "Manipulating Event Handlers" on page 272
- "Using Event Counters" on page 273
- "Event Safety" on page 273
- "Setting Event Specifications" on page 274
- "Event Specification Modifiers" on page 289
- "Parsing and Ambiguity" on page 292
- "Using Predefined Variables" on page 292
- "Event Handler Examples" on page 296

Event Handlers

Event management is based on the concept of a *handler*. The name comes from an analogy with hardware interrupt handlers. Each event management command typically creates a handler, which consists of an *event specification* and a series of side-effect actions. (See "Setting Event Specifications" on page 274.) The event specification specifies the event that will trigger the handler.

When the event occurs and the handler is triggered, the handler evaluates the event according to any modifiers included in the event specification. (See "Event Specification Modifiers" on page 289.) If the event meets the conditions imposed by the modifiers, the handler's side-effect actions are performed (that is, the handler "fires").

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

The most generic form of creating a handler is by using the `when` command.

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

Examples in this chapter show how you can write a command (like `stop`, `step`, or `ignore`) in terms of `when`. These examples are meant to illustrate the flexibility of the `when` command and the underlying *handler* mechanism, but they are not always exact replacements.

Creating Event Handlers

Use the `when` command, `stop` command, and `trace` command to create event handlers. (For detailed information, see "`when` Command" on page 413, "`stop` Command" on page 387, and "`trace` Command" on page 400.)

`stop` is shorthand for a common `when` idiom.

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

An *event-specification* is used by the event management commands `stop`, `when`, and `trace` to specify an event of interest. (see "Setting Event Specifications" on page 274).

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

Every command returns a number known as a handler id (*hid*). You can access this number using the predefined variable `$newhandlerid`.

Manipulating Event Handlers

You can use the following commands to manipulate event handlers. For more information on each command, see the cited section.

TABLE 2 Manipulating Event Handlers

Command	Description	For More Information
`status`	Lists handlers	See "`status` Command" on page 384

Command	Description	For More Information
delete	Deletes all handlers including temporary handlers	See "delete Command" on page 332
clear	Deletes handlers based on breakpoint position	See "clear Command" on page 316
handler -enable	Enables handlers	See "handler Command" on page 346
handler -disable	Disables handlers	See "handler Command" on page 346
cancel	Cancels signals and enables the process to continue	See "cancel Command" on page 312

Using Event Counters

An event handler has a trip counter, which has a count limit. Whenever the specified event occurs, the counter is incremented. The action associated with the handler is performed only if the count reaches the limit, at which point the counter is automatically reset to 0. The default limit is 1. Whenever a process is rerun, all event counters are reset.

You can set the count limit using the `-count` modifier with a `stop` command, `when` command, or `trace` command. Otherwise, use the `handler` command to individually manipulate event handlers.

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

Event Safety

While `dbx` provides you with a rich set of breakpoint types through the event mechanism, it also uses many events internally. By stopping on some of these internal events you can easily disrupt the internal workings of `dbx`. If you modify the process state in these cases the chance of disruption is even higher. See Appendix A, "Modifying a Program State" and "Call Safety" on page 93.

`dbx` can protect itself from disruption in some cases but not all cases. Some events are implemented in terms of lower level events. For example, all stepping is based on the `fault FLTTRACE` event. So, issuing the command `stop fault FLTTRACE` disrupts stepping.

During the following phases of debugging, `dbx` is unable to handle user events because they interfere with some careful internal orchestration. These phases include:

- When `rtld` runs at program startup (see "Dynamic Linker" on page 263)
- The beginning and end of processes
- Following the `fork()` function and the `exec()` function (see "Following the fork Function" on page 180 and "Following the exec Function" on page 180
- During calls when `dbx` needs to initialize a head in the user process (`proc_heap_init()`)
- During calls when `dbx` needs to ensure availability of mapped pages on the stack (`ensure_stack_memory()`)

In many cases you can use the `when` command instead of the `stop` command, and echo the information you would have otherwise acquired interactively.

`dbx` protects itself by:

- Disallowing the `stop` command for the `sync`, `syncrtld`, and `prog_new` events
- Ignoring the `stop` command during the `rtld` handshake and the other phases mentioned above

For example:

```
...SolBook linebreakstopped in munmap at 0xff3d503c 0xff3d503c: munmap+0x0004: ta %icc,
0x00000008SolBook linebreak dbx76: warning: 'stop' ignored -- while doing rtld handshake
```

Only the stoppage effect, including recording in the `$firedhandlers` variable, is ignored. Counts or filters are still active. To stop in such a case, set the `event_safety` environment variable to `off`.

Setting Event Specifications

Event specifications are used by the `stop` command, `stopi` command, `when` command, `wheni` command, `trace` command, and `tracei` command to denote event types and parameters. The format consists of a keyword representing the event type and optional parameters. The meaning of an event specification is generally identical for all three commands. Exceptions are documented in the command descriptions in Appendix D.

Breakpoint Event Specifications

A breakpoint is a location where an action occurs, at which point the program stops executing. This section describes event specifications for breakpoint events.

`in` Event Specification

The syntax for the `in` event specification is:

`in` *function*

The function has been entered, and the first line is about to be executed. The first executable code after the prolog is used as the actual breakpoint location. This might be a line where a local variable is being initialized. In the case of C++ constructors, execution stops after all base class constructors have executed. If the `-instr` modifier is used, it is the first instruction of the function about to be executed. The *function* specification can take a formal parameter signature to help with overloaded function names or template instance specification. For example:

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

Note - Do not confuse `in` *function* with the-`in` *function* modifier.

`at` Event Specification

The syntax for the `at` event specification is:

`at [`*filename:*`]`*line-number*

The designated line is about to be executed. If you specify *filename*, then the designated line in the specified file is about to be executed. The file name can be the name of a source file or an object file. Although quotation marks are not required, they might be necessary if the file name contains special characters. If the designated line is in template code, a breakpoint is placed on all instances of that template.

You can also use specify a specific address:

`at` *address-expression*

The instruction at the given address is about to be executed. This event is available only with the `stopi` command or with the `-instr` event modifier

`infile` Event Specification

The syntax for the `infile` event specification is:

`infile` *filename*

This event puts a breakpoint on every function defined in a file. The `stop infile` command iterates through the same list of functions as the `funcs -f` *filename* command.

Method definitions in `.h` files, template files, or plain C code in `.h` files, such as the kind used by the `regexp` command, might contribute function definitions to a file, but these definitions are excluded.

If the specified filename is the name of an object file (that is, it ends in `.o`). breakpoints are put on every function that occurs in that object file.

The `stop infile list.h` command does not put breakpoints on all instances of methods defined in the `list.h` file. Use events like `inclass` or `inmethod` to do so.

The `fix` command might eliminate or add a function to a file. The `stop infile` command puts breakpoints on all old versions of function in a file as well as any functions that might be added in the future.

No breakpoints are put on nested functions or subroutines in Fortran files.

You can use the `clear` command to disable a single breakpoint in the set created by the `infile` event.

`infunction` Event Specification

The syntax for the `infunction` event specification is:

`infunction` *function*

This specification is equivalent to `in` *function* for all overloaded functions named *function* or all template instantiations thereof.

`inmember` Event Specification

The syntax for the `inmember` event specification is:

`inmember` *function*

This specification is an alias for the `inmethod` event specification.

`inmethod` Event Specification

The syntax for the `inmethod` event specification is:

`inmethod` *function*

This specification is equivalent to the `in` *function* or the member method named *function* for every class.

`inclass` Event Specification

The syntax for the `inclass` event specification is:

`inclass` *classname* `[-recurse | -norecurse]`

This specification is equivalent to `in` *function* for all member functions that are members of *classname*, but not any of the bases of *classname*. `-norecurse` is the default. If `-recurse` is specified, the base classes are included.

`inobject` Event Specification

The syntax for the `inobject` event specification is:

`inobject` *object-expression* `[-recurse | -norecurse]`

A member function called on the specific object at the address denoted by *object-expression* has been called. `stop inobject` *ox* is roughly equivalent to the following, but unlike `inclass`, bases of the dynamic type of *ox* are included. `-recurse` is the default. If `-norecurse` is specified, the base classes are not included.

```
stop inclass dynamic_type(ox) -if this==ox
```

Data Change Event Specifications

This section describes event specifications for events that involve access or change to the contents of a memory address.

`access` Event Specification

The syntax for the `access` event specification is:

`access` *mode address-expression* `[,`*byte-size-expression*`]`

The memory specified by *address-expression* has been accessed.

mode specifies how the memory was accessed. Valid values are one or all of the following letters:

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

mode can also contain either of the following:

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

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

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

```
stop access w 0x5678, sizeof(Complex)
```

The `access` command has the limitation that no two matched regions can overlap.

Note - The `access` event specification is a replacement for the `modify` event specification.

`change` Event Specification

The syntax for the `change` event specification is:

`change` *variable*

The value of *variable* has changed. The `change` event is roughly equivalent to:

```
when step { if [ $last_value !=$[variable]]
```

```
    then
         stop
    else
         last_value=$[variable]
    fi
  }
```

This event is implemented using single-stepping. For faster performance, use the `access` event.

The first time *variable* is checked causes one event, even though no change is detected. This first event provides access to the initial value of *variable*. Subsequent detected changes in the value of *variable* trigger additional events.

`cond` Event Specification

The syntax for the `cond` event specification is:

`cond` *condition-expression*

The condition denoted by *condition-expression* evaluates to true. You can specify any expression for *condition-expression*, but it must evaluate to an integral type. The `cond` event is roughly equivalent to the following `stop` command:

`stop step -if` *conditional-expression*

System Event Specifications

This section describes event specifications for system events.

`dlopen` and `dlclose` Event Specification

The syntax for the `dlopen()` and `dlopen()` event specifications is:

`dlopen [` *lib-path* `]`

`dlclose [` *lib-path* `]`

System events occur after a `dlopen()` call or a `dlclose()` call succeeds. A `dlopen()` call or `dlclose()` call can cause more than one library to be loaded. The list of these libraries is always available in the predefined variable `$dllist`. The first shell word in `$dllist` is a + (plus sign) or a - (minus sign), indicating whether the list of libraries is being added or deleted.

lib-path is the name of a shared library. If it is specified, the event occurs only if the given library was loaded or unloaded. In that case, `$dlobj` contains the name of the library. `$dllist` is still available.

If *lib-path* begins with a `/`, a full string match is performed. Otherwise, only the tails of the paths are compared.

If *lib-path* is not specified, then the events always occur whenever there is any `dl`-activity. In this case, `$dlobj` is empty but `$dllist` is valid.

`fault` Event Specification

The syntax for the `fault` event specification is:

`fault` *fault*

The `fault` event occurs when the specified fault is encountered. The faults are architecture-dependent. The set of faults known to `dbx` is listed in the following list and defined in the `proc`(4) man page.

`FLTILL`	Illegal instruction
`FLTPRIV`	Privileged instruction
`FLTBPT`*	Breakpoint trap
`FLTTRACE`*	Trace trap (single step)
`FLTACCESS`	Memory access (such as alignment)
`FLTBOUNDS`	Memory bounds (invalid address)
`FLTIOVF`	Integer overflow
`FLTIZDIV`	Integer zero divide
`FLTPE`	Floating-point exception
`FLTSTACK`	Irrecoverable stack fault

`FLTPAGE`	Recoverable page fault
`FLTWATCH`*	Watchpoint trap
`FLTCPCOVF`	CPU performance counter overflow

Note - `FLTBPT`, `FLTTRACE`, and `FLTWATCH` are not handled because they are used by `dbx` to implement breakpoints, single-stepping, and watchpoints.

These faults are taken from `/usr/include/sys/fault.h`. *fault* can be any of those listed above, in uppercase or lowercase, with or without the `FLT`- prefix, or the actual numerical code.

Note - The `fault` event is not available on Linux platforms.

`lwp_exit` Event Specification

The syntax for the `lwp_exit` event specification is:

```
lwp_exit
```

The `lwp_exit` event occurs when `lwp` has been exited. `$lwp` contains the ID of the exited LWP (lightweight process) for the duration of the event handler.

Note - The `lwp_exit` event is not available on Linux platforms.

`sig` Event Specification

The syntax for the `sig` event specification is:

`sig` *signal*

The `sig` *signal* event occurs when the signal is first delivered to the program being debugged. *signal* can be either a decimal number or the signal name in uppercase or lowercase. The prefix is optional. This event is completely independent of the `catch` command and `ignore` command, although the `catch` command can be implemented as follows:

```
function simple_catch {
    when sig $1 {
            stop;
            echo Stopped due to $sigstr $sig
```

```
            whereami
    }
}
```

Note - When the `sig` event is received, the process has not seen it yet. Only if you continue the process with the specified signal is the signal forwarded to it.

Alternatively, you can specify a signal with a sub-code. The syntax for this option of the `sig` event specification is:

`sig` *signal sub-code*

When the specified signal with the specified *sub-code* is first delivered to the child, the `sig` *signal sub-code* event occurs. As with signals, you can provide the *sub-code* as a decimal number, in uppercase or lowercase. The prefix is optional.

`sysin` Event Specification

The syntax for the `sysin` event specification is:

`sysin` *code*|*name*

The specified system call has just been initiated, and the process has entered kernel mode.

The concept of system call supported by `dbx` is that provided by traps into the kernel as enumerated in `/usr/include/sys/syscall.h`.

This concept is not the same as the ABI notion of system calls. Some ABI system calls are partially implemented in user mode and use non-ABI kernel traps. However, most of the generic system calls (the main exception being signal handling) are the same between `syscall.h` and the ABI.

Note - The `sysin` event is not available on Linux platforms.

The list of kernel system call traps in `/usr/include/sys/syscall.h` is part of a private interface in the Oracle Solaris OS that changes from release to release. The list of trap names (codes) and trap numbers that `dbx` accepts includes all of those supported by any of the versions of the Oracle Solaris OS that `dbx` supports. The names supported by `dbx` are unlikely to exactly match those of any particular release of the Oracle Solaris OS, and some of the names in `syscall.h` might not be available. Any trap number (code) is accepted by `dbx` and works as expected, but a warning is issued if it does not correspond to a known system call trap.

`sysout` Event Specification

The syntax for the `sysout` event specification is:

`sysout` *code*|*name*

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

Note - The `sysout` event is not available on Linux platforms.

`sysin | sysout` Event Specifications

Without arguments, all system calls are traced. Certain `dbx` features, for example, the `modify` event and runtime checking, cause the child to execute system calls for its own purposes and show up if traced.

Execution Progress Event Specifications

This section describes event specifications for events pertaining to execution progress.

`exit` Event Specification

The syntax for the `exit` event specification is:

`exit` *exitcode*

The `exit` event occurs when the process has exited.

`next` Event Specification

The `next` event is similar to the `step` event except that functions are not stepped into.

`returns` Event Specification

The `returns` event is a breakpoint at the return point of the current *visited* function. The visited function is used so that you can use the `returns` event specification after giving a number of step up commands. The returns event is always -temp and can only be created in the presence of a live process.

The syntax for the returns event specification is:

returns *function*

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

- SPARC based systems – $o0
- x86 based systems – $eax
- x64 based systems – $rax, $rdx

The event is roughly equivalent to:

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

step Event Specification

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

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

When enabling a step event, you instruct dbx to single step automatically the next time cont command is used.

Note - The step (and next) events do not occur upon the termination of the step command. The step command is implemented in terms of the step event roughly as follows: alias step="when step -temp { whereami; stop; }; cont"

throw Event Specification

The syntax for the throw event is:

throw [*type* | -unhandled | -unexpected]

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

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

If the `-unhandled` option is specified, a special exception type signifying an exception is thrown but for which there is no handler.

The `-unexpected` option is specified, a special exception type signifying an exception does not satisfy the exception specification of the function that threw it.

Tracked Thread Event Specifications

The following section describes event specifications for tracked threads.

`omp_barrier` Event Specification

The `omp_barrier` event specification is when the tracked thread enters or exits a barrier. You can specify a *type*, which can be `explicit` or `implicit`, and a *state*, which can be `enter`, `exit`, or `all_entered`. The default is `explicit all_entered`.

`omp_taskwait` Event Specification

The `omp_taskwait` event specification is when the tracked thread enters or exists a taskwait. You can specify a *state,* which can be `enter` or `exit`. The default is `exit`.

`omp_ordered` Event Specification

The `omp_ordered` event specification is when the tracked thread enters or exists an ordered region. You can specify a *state*, which can be `begin`, `enter` or `exit`. The default is `enter`.

`omp_critical` Event Specification

The `omp_critical` event specification is when the tracked thread enters a critical region.

`omp_atomic` Event Specification

The `omp_atomic` event specification is when the tracked thread enters or exists an atomic region. You can specify a *state*, which can be `begin` or `exit`. The default is `begin`.

`omp_flush` Event Specification

The `omp_flush` event specification is when the tracked thread enters a explicit flush region.

`omp_task` Event Specification

The `omp_task` event specification is when the tracked thread enters or exists a task region. You can specify a *state*, which can be `create`, `start` or `finish`. The default is `start`.

`omp_master` Event Specification

The `omp_master` event specification is when the tracked thread enters a master region.

`omp_single` Event Specification

The `omp_single` event specification is when the tracked thread enters a single region.

Other Event Specifications

This section describes event specifications for other types of events.

`attach` Event Specification

The `attach` event is when `dbx` has successfully attached to a process.

detach Event Specification

The detach event is when dbx has successfully detached from the program being debugged.

lastrites Event Specification

The lastrites event is when process being debugged is about to expire, which can happen for the following reasons:

- The _exit(2) system call has been called, either through an explicit call or when main() returns.
- A terminating signal is about to be delivered.
- The process is being killed by the kill command.

The final state of the process is usually, but not always, available when this event is triggered, giving you your last opportunity to examine the state of the process. Resuming execution after this event terminates the process.

Note - The lastrites event is not available on Linux platforms.

---

proc_gone Event Specification

The proc_gone event occurs when dbx is no longer associated with a debugged process. The predefined variable $reason can be signal, exit, kill, or detach.

prog_new Event Specification

The prog_new event occurs when a new program has been loaded as a result of follow exec.

Note - Handlers for this event are always permanent.

---

stop Event Specification

The stop event occurs whenever the process stops such that the user receives a prompt, particularly in response to a stop handler. For example, the following commands are equivalent:

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

`sync` Event Specification

The `sync` event occurs when the process being debugged has just been executed with `exec()`. All memory specified in `a.out` is valid and present, but preloaded shared libraries have not been loaded. For example, `printf`, although available to `dbx`, has not been mapped into memory.

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

Note - The `sync` event is not available on Linux platforms.

`syncrtld` Event Specification

The `syncrtld` event occurs after a `sync` or an `attach` if the process being debugged has not yet processed shared libraries. It executes after the dynamic linker startup code has executed and the symbol tables of all preloaded shared libraries have been loaded but before any code in the `.init` section has run.

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

`thr_create [`*thread-ID*`]` Event Specification

The `thr_create` event occurs when a thread, or a thread with the specified thread ID, has been created. For example, in the following `stop` command, the thread ID `t@1` refers to creating thread, while the thread ID `t@5` refers to the created thread.

```
stop thr_create t@5 -thread t@1
```

`thr_exit` Event Specification

The `thr_exit` event occurs when a thread has exited. To capture the exit of a specific thread, use the `-thread` option of the `stop` command as follows:

```
stop thr_exit -thread t@5
```

`timer` Event Specification

The syntax for the `timer` event is:

`timer` *seconds*

The `timer` event occurs when the program being debugged has been running for *seconds*. The timer used with this event is shared with `collector` command. The resolution is in milliseconds, so a floating point value for *seconds*, for example `0.001`, is acceptable.

Event Specification Modifiers

An event specification modifier sets additional attributes of a handler, the most common kind being event filters. Modifiers must appear after the keyword portion of an event specification. A modifier begins with a dash (`-`). The following are the valid event specification modifiers.

`-if` Modifier

The syntax for the `-if` modifier is:

`-if` *condition*

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

If the `-if` modifier is used with an event that has an associated singular source location, such as `in` or `at`, *condition* is evaluated in the scope corresponding to that location. Otherwise, qualify it with the desired scope.

Macro expansion is performed on the condition according to same conventions as with the `print` command.

`-resumeone` Modifier

The `-resumeone` modifier can be used with the `-if` modifier in an event specification for a multithreaded program, and causes only one thread to be resumed if the condition contains function calls. For more information, see "Qualifying Breakpoints With Conditional Filters" on page 105.

`-in` Modifier

The syntax for the `-in` modifier is:

`-in` *function*

The event triggers only if it occurs between the time the first instruction of the given function is reached and the time the function returns. Recursion on the function are ignored.

`-disable` Modifier

The-`disable` modifier creates the handler in the disabled state.

`-count` *n*, `-count infinity` Modifier

The syntax for the `-count` modifier is:

`-count` *n*

or

`-count infinity`

The `-count` *n* and `-count infinity` modifiers have the handler count from 0 (see "Using Event Counters" on page 273). Each time the event occurs, the count is incremented until it reaches *n*. Once that happens, the handler fires and the counter is reset to zero.

Counts of all enabled handlers are reset when a program is run or rerun. More specifically, they are reset when the `sync` event occurs.

The count is reset when you begin debugging a new program with the `debug -r` command (see "`debug` Command" on page 329) or the `attach -r` command (see "`attach` Command" on page 309).

`-temp` Modifier

The `-temp` modifier creates a temporary handler. Once the event has occurred it is automatically deleted. By default, handlers are not temporary. If the handler is a counting handler, it is automatically deleted only when the count reaches 0 (zero).

Use the `delete -temp` command to delete all temporary handlers.

`-instr` Modifier

The `-instr` modifier makes the handler act at an instruction level. This event replaces the traditional ’`i`’ suffix of most commands. It usually modifies two aspects of the event handler:

- Any message prints assembly-level rather than source-level information.
- The granularity of the event becomes instruction level. For instance, `step -instr` implies instruction-level stepping.

`-thread` Modifier

The syntax for the `-thread` modifier is:

`-thread` *thread-ID*

The `-thread` modifier means the action is executed only if the thread that caused the event matches a different thread ID. The specific thread you have in mind might be assigned a different thread ID from one execution of the program to the next.

`-lwp` Modifier

The syntax for the `-lwp` modifier is:

`-lwp` *lwp-ID*

The `-lwp` modifier means the action is executed only if the `-lwp` that caused the event matches *lwp-ID*. The specific `-lwp` you have in mind might be assigned a different *lwp-ID* from one execution of the program to the next.

`-hidden` Modifier

The `-hidden` modifier hides the handler in a regular `status` command. Use `status -h` to see hidden handlers.

`-perm` Modifier

Normally all handlers are thrown away when a new program is loaded. Using the `-perm` modifier retains the handler across debugging sessions. A plain `delete` command does not delete a permanent handler. Use `delete -p` to delete a permanent handler.

Parsing and Ambiguity

The syntax for event specifications and modifiers is keyword driven and based on `ksh` conventions. Everything is split into words delimited by spaces.

Expressions can have spaces embedded in them, causing ambiguous situations. For example, consider the following two commands:

```
when a -temp
when a-temp
```

In the first example, even though the application might have a variable named *temp*, the `dbx` parser resolves the event specification in favor of `-temp` being a modifier. In the second example, `a-temp` is collectively passed to a language-specific expression parser. If no variables are named *a* and *temp*, an error occurs. Use parentheses to force parsing.

Using Predefined Variables

Certain read-only `ksh` predefined variables are provided. The variables listed in the following table are always valid.

Variable	Definition
`$ins`	Disassembly of the current instruction.
`$lineno`	Current line number in decimal.
`$vlineno`	Current “visiting” line number in decimal.
`$line`	Contents of the current line.
`$func`	Name of the current function.
`$vfunc`	Name of the current “visiting” function.
`$class`	Name of the class to which `$func` belongs.
`$vclass`	Name of the class to which `$vfunc` belongs.
`$file`	Name of the current file.

Variable	Definition
$vfile	Name of the current file being visited.
$loadobj	Name of the current loadable object.
$vloadobj	Name of the current loadable object being visited.
$scope	Scope of the current PC in back-quote notation.
$vscope	Scope of the visited PC in back-quote notation.
$funcaddr	Address of $func in hex.
$caller	Name of the function calling $func.
$dllist	After a dlopen or dlclose event, contains the list of load objects just loaded or unloaded. The first word of dllist is a + (plus sign) or a - (minus sign) depending on whether a dlopen or a dlclose has occurred.
$newhandlerid	ID of the most recently created handler. This variable has an undefined value after any command that deletes handlers. Use the variable immediately after creating a handler. dbx cannot capture all of the handler IDs for a command that creates multiple handlers.
$firedhandlers	List of handler ids that caused the most recent stoppage. The handlers on the list are marked with *(an asterisk) in the output of the status command.
$proc	Process ID of the current process being debugged.
$lwp	ID of the current LWP.
$thread	Thread ID of the current thread.
$newlwp	ID of a newly created LWP.
$newthread	ID of a newly created thread.
$prog	Full path name of the program being debugged.
$oprog	Previous value of $prog, which is used to get back to what you were debugging following an exec(), when the full path name of the program reverts to - (dash). While $prog is expanded to a full path name, $oprog contains the program path as specified on the command line or to the debug command. If exec() is called more than once, there is no way to return to the original program.
$exec32	True if the dbx binary is 32-bit.
$exitcode	Exit status from the last run of the program. The value is an empty string if the process has not exited.
$booting	Set to true if the event occurs during the boot process. Whenever a new program is debugged, it is first booted so that the list and location of shared libraries can be ascertained. The process is then killed. This sequence is termed “booting”. While booting is occurring, all events are still available. Use this variable to distinguish, for example, the sync and syncrtld events occurring during a debugging run and the ones occurring during a normal run.
$machtype	If a program is loaded, returns its machine type: sparcv8, sparcv8+, sparcv9, x86, or x86_64. Otherwise, returns unknown.

Variable	Definition

`$datamodel`	If a program is loaded, returns its data model: `ilp32` or `lp64`. Otherwise, returns `unknown`. To find the model of the program you've just loaded, use the following in your `.dbxrc` file:
	`when prog_new -perm {`
	`     echo machine: $machtype $datamodel;`
	`}`

The following example shows that `whereami` can be implemented:

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

Variables Valid for `when` Command

The variables described in this section are valid only within the body of a `when` command.

`$handlerid`

During the execution of the body, `$handlerid` is the ID of the `when` command to which the body belongs. The following commands are equivalent:

```
when X -temp { do_stuff; }
when X  { do_stuff; delete $handlerid; }
```

Variables Valid for `when` Command and Specific Events

Certain variables are valid only within the body of a `when` command and for specific events, as shown in the following tables.

TABLE 3 Variables Valid for `sig` Event

Variable	Description
`$sig`	Signal number that caused the event
`$sigstr`	Name of `$sig`

Variable	Description
`$sigcode`	Subcode of `$sig` if applicable
`$sigcodestr`	Name of `$sigcode`
`$sigsender`	Process ID of sender of the signal, if appropriate

TABLE 4 Variable Valid for `exit` Event

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

TABLE 5 Variable Valid for `dlopen` and `dlclose` Events

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

TABLE 6 Variables Valid for `sysin` and `sysout` Events

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

TABLE 7 Variable Valid for `proc_gone` Events

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

TABLE 8 Variables Valid for `thr_create` Event

Variable	Description
`$newthread`	ID of the newly created thread, for example, `t@5`
`$newlwp`	ID of the newly created LWP, for example, `l@4`

TABLE 9 Variables Valid for `access` Event

Variable	Description
`$watchaddr`	The address being written to, read from, or executed
`$watchmode`	One of the following: r for read, w for write, x for execute; followed by one of the following: a for after, b for before

Event Handler Examples

This section provides some examples of setting event handlers.

Setting a Breakpoint for Store to an Array Member

This example shows how to set a data change breakpoint 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;
```

Implementing a Simple Trace

This example shows how to implement a simple trace:

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

Enabling a Handler While Within a Function

The following example shows how to enable a handler while within a function:

```
<dbx> trace step -in foo
```

This command is equivalent to the following:

```
    # create handler in disabled state
    when step -disable { echo Stepped to $line; }
    t=$newhandlerid    # remember handler id
    when in foo {
    # when entered foo enable the trace
    handler -enable "$t"
    # arrange so that upon returning from foo,
    # the trace is disabled.
    when returns { handler -disable "$t"; };
    }
```

Determining the Number of Lines Executed

This example shows how to see how many lines have been executed in a small program, type:

```
(dbx) stop step -count infinity     # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

The program never stops, and then the program terminates. The number of lines executed is 133. This process is very slow. It is most useful with breakpoints on functions that are called many times.

Determining the Number of Instructions Executed by a Source Line

This example shows how 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, the lines in the function are counted as well. You can use the `next` event instead of `step` to count instructions, excluding called functions.

Enabling a Breakpoint After an Event Occurs

Enable a breakpoint only after another event has occurred. For example, you would use the following breakpoint if your program begins to execute incorrectly in function `hash`, but only after the 1300th 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.

Resetting Application Files for `replay`

In this example, if your application processes files that need to be reset during a `replay`, you can write a handler to do that each time you run the program.

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database...    # during which database gets clobbered
(dbx) save
...              # implies a RUN, which implies the SYNC event which
(dbx) restore      # causes regen to run
```

Checking Program Status

This example shows how to see quickly where the program is while it is running, type:

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

You would then issue `^C` to see a stack trace of the program without stopping it.

This example is basically what the collector hand sample mode does (and more). Use `SIGQUIT (^\)` to interrupt the program because `^C` is now used.

Catch Floating-Point Exceptions

The following example shows how to catch only specific floating-point exceptions, for example, IEEE underflow:

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

For more information about enabling ieee handlers, see "Trapping the FPE Signal (Oracle Solaris Only)" on page 193.

♦ ♦ ♦ APPENDIX C

Macros

By default, selected expressions are macro expanded before being evaluated, including expressions you specify with the `print`, `display`, and `watch` commands; the `-if` option of the `stop`, `trace`, and `when` commands; and the `$[]` construct. Macro expansion is also applied to balloon evaluation and watches in the IDE or `dbxtool`.

Additional Uses of Macro Expansion

Macro expansion is applied to both the variable and the expression in an `assign` command.

In the `call` command, macro expansion is applied to the name of the function being called as well as to the parameters being passed.

The `macro` commandtakes any expression and macro and expands the macro. For example:

```
(dbx) macro D(1, 2)
    Expansion of: D(1, 2)
              is: d(1,2)
```

If you give the `whatis` command a macro, it shows the macro's definition. For example:

```
(dbx) whatis B
    #define B(x) b(x)
```

If you give the `which` command a macro, it shows where the macro that is currently active in the scope is defined. For example:

```
(dbx) which B2
    `a.out`macro_wh.c`B2    # defined at defs2.h:3
            # included from defs1.h:3
            # included from macro_wh.c:23
```

If you give the `whereis` command a macro, it shows all of the places where the macro has been defined. The list is limited to modules for which `dbx` has already read debugging information. For example:

```
(dbx) whereis U
    macro:          U       # defined at macro_wh.c:21
    macro:          U       # undefined at defs1.h:5
```

The `dbxenv` variable `macro_expand` controls whether these commands expand macros. It is set to `on` by default.

In general, the `+m` option in `dbx` commands causes the commands to bypass macro expansion. The `-m` option forces macro expansion even if the `dbxenv` variable `macro_expand` is set to `off`. An exception is the `-m` option within the `$[]` construct, where `-m` only causes macros to be expanded, with no evaluation taking place. This exception facilitates macro expansion in shell scripts.

Macro Definitions

`dbx` can recognize macro definitions in two ways:

- Definitions are provided by the compilers when you compile with the `-g3` option if you use the default DWARF format for debugging information. They are not provided if you specify the `-xdebugformat=stabs` option when compiling.
- `dbx` can re-create definitions by skimming the source file and its include files. Accurate re-creation depends on access to the original sources and include files. It also depends on the availability of the path name to the compiler used, and on compiler options like `-D` and `-I`. This information is available in both DWARF and stabs formats from Oracle Developer Studio compiler, but not from GNU compilers. See "Skimming Errors" on page 304 and "Using the `pathmap` Command to Improve Skimming" on page 305 for information about ensuring successful skimming.

The `dbxenv` variable `macro_source` (see Table 1, "`dbx` Environment Variables," on page 61 in Chapter 3, "Customizing `dbx`") controls which one of the two methods `dbx` uses to recognize macro definitions.

There are several factors to take into account in choosing which method you want `dbx` to use.

Compiler and Compiler Options

One factor in choosing a macro definition method is the availability of various types of information that depend on which compiler and compiler options you used to build your code. The following table shows which methods you can choose depending on the compiler and debugging information options.

TABLE 10 Macro Definition Methods Available for Various Build Options

Compiler	`-g` option	Debug Information Format	Methods That Work
Oracle Developer Studio	`-g`	DWARF	Skimming
Oracle Developer Studio	`-g`	stabs	Skimming
Oracle Developer Studio	`-g3`	DWARF	Skimming and from compiler
Oracle Developer Studio	`-g3`	stabs	Skimming (`-g3` option with `-xdebugformat=stabs` option is not supported)
GNU	`-g`	DWARF	Neither
GNU	`-g`	stabs	N/A
GNU	`-g3`	DWARF	From compiler
GNU	`-g3`	stabs	N/A

Tradeoffs in Functionality

Another factor to take into account in choosing a macro definition method is the tradeoffs in functionality depending on which method you choose:

- **Size of executable**. The main advantage of the skimming method is that it does not require compilation with the `-g3` option because it works with the smaller executables produced by compiling with the `-g` option.
- **Debugging format**. Skimming works with both DWARF and stabs. Compiling with the `-g3` option to get the definitions from the compiler works only with DWARF.
- **Speed**. Skimming takes up to one second the first time an expression is evaluated for a module for which `dbx` has not yet read the debugging information.
- **Accuracy**. Information provided by the compilers when you compile with the `-g3` option is more stable and accurate than information provided by skimming.
- **Availability of the build environment**. Skimming requires that the compilers, source code files, and include files be available during debugging. `dbx` does not check for these

items becoming out of date, so if they are likely to change, accuracy might deteriorate and compiling with the `-g3` option might be better than depending on skimming.

- **Debugging on a different system from the one where the code was compiled**. If you compiled the code on system A and are debugging it on system B, `dbx` accesses files on system A using NFS with some help from the `pathmap` command.

  The `pathmap` command also helps facilitate file access during skimming. Although it works for your program's source files and include files, it might not work for system include files because `/usr/include` is not usually available through NFS. Macro definitions therefore are extracted from `/usr/include` on the debugging system instead of on the build system.

  You can choose to be aware of and tolerant of possible discrepancies between system include files, or choose to compile with the `-g3` option.

Limitations

- Although Fortran compilers support macros through the `cpp(1)` function or the `fpp(1)` function, `dbx` does not support macro expansion for Fortran.
- `dbx` ignores macro information generated by compiling with the `-g3` option and the `-xdebugformat=stabs` option.

  For more information about the stabs index, see the Stab Interface guide,found with the path *install-dir*`/solarisstudio12.4/READMEs/stabs.pdf`.
- Skimming works with code compiled with the `-g` option and the `-xdebugformat=stabs` option.

Skimming Errors

You are depending on macro skimming if you did not compile your code with the `-g3` option and have the `macro_source` `dbxenv` variable set to `skim_unless_compiler` or `skim`.

For skimming to succeed for a module, the following conditions need to be true:

- The module must have been compiled with a Oracle Developer Studio compiler using the `-g` option.
- The compiler used to compile the module must be accessible by `dbx`.
- The source file for the module must be accessible by `dbx`.
- Files included by the source code of the module must be available, that is, the path given to the `-I` options when the module was compiled must be accessible by `dbx`.

- The source code must be lexically sound. For example, it cannot contain unterminated strings of comments or be missing #endifs.

If the source code or include files are not accessible by dbx, you can use the pathmap command to make them accessible.

Using the pathmap Command to Improve Skimming

If you move your source files after compiling, build on one machine and debug on another, or are in one of the other situations described in "Finding Source and Object Files" on page 86, macro skimming might not be able to find include files in the file it is skimming. The solution, as with other cases of files not being found, is to use the pathmap command to help the macro skimmer locate include directories. Imagine, for example, that you compile with the option -I/export/home/proj1/include and have the statement #include "module1/api.h" in your code. Then, if you rename proj1 to proj2, the following pathmap command will help the macro skimmer locate your files:

```
pathmap /export/home/proj1 /export/home/proj2
```

The pathmap is not applied to the compilers used to compile the original code.

When you are working with macros, you must reload your application in order to have pathmaps take effect, unlike other situations when a file is not found and you can use the pathmap command to make changes in a pathmapping that are immediately effective.

The pathmap command helps dbx find the correct files when you build on one machine and debug on another. However, system include files such as /usr/include/stdio.h are typically not exported from the build machine, so the macro skimmer is likely to use the files on the debug machine. In some cases, a system include file might not be available on the debug machine. The value of system-specific and system-dependent macros also might not be the same on the debug machine as on the build machine.

If the pathmap command does not solve your skimming problems, consider compiling your code with the -g3 option and setting the macro_source dbxenv variable to skim_unless_compiler or compiler.

♦ ♦ ♦ APPENDIX D

Command Reference

This appendix provides detailed syntax and functional descriptions of all of the `dbx` commands.

`adi assign` Command

The `adi assign` command assigns a new ADI version to an address. This command is supported only in native mode on Oracle Solaris SPARC systems.

Native Mode Syntax

`adi assign` <*addr*> [/ <*count*>] = <*ver*>	Assigns a new ADI version to an address starting at *addr* for *count* addresses. The default count is 1. *ver* must be between 0 and 15.
`adi assign` <*&object*> [/ <*count*>] = <*ver*>	Assigns a new ADI version to the address span of the object, or *count* bytes; whichever is less. The default count is the entire object.
`adi assign` *<addr1>*, *<addr2>* = <*ver*>	Assigns a new ADI version to addresses from *addr1* through *addr2*. *ver* must be between 0 and 15.

where:

addr is an address in the debuggee's address space.

count is the number of bytes.

ver is the newly assigned ADI version.

`adi examine` Command

The `adi examine` command displays one ADI version per cacheline. All groups of output lines that would be identical to the immediately preceding output line (except for byte offsets) will be replaced with a line containing only an asterisk (`*`). The last line is always printed. This command is supported only in native mode on Oracle Solaris SPARC systems.

Native Mode Syntax

`adi examine` | x <addr> [/ <count>]	Displays the ADI version, one per cache line, starting at *addr* for *count* addresses. The default count is 1.
`adi examine` | x *&object* [/ <*count*>]	Displays the ADI version for an object for *count* bytes, up to the address span of the object. The default count is the entire object. Only one version per cacheline is printed.
`adi examine` | x <*addr1*> , <*addr2*>	Displays the ADI version, one per cacheline, from *addr1* through *addr2*, inclusive.

where:

addr is an address in the debuggee's address space.

count is the number of bytes.

`assign` Command

In native mode, the `assign` command assigns a new value to a program variable. In Java mode, the `assign` command assigns a new value to a local variable or parameter.

Native Mode Syntax

`assign` *variable* `=` *expression*

where:

expression is the value to be assigned to *variable*.

Java Mode Syntax

`assign` *identifier* `=` *expression*

where:

expression is a valid Java expression, which can include any of the following:

- *class-name* is the name of a Java class. You can use either of the following:
  - The package path using a period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`
  - The full path name preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers. For example, `#test1/extra/T1$Inner`. Enclose *class-name* in quotation marks if you use the `$` qualifier.
- *field-name* is the name of a field in the class.
- *identifier* is a local variable or parameter, including `this`, the current class instance variable (*object-name*.*field-name*) or a class (static) variable (*class-name*.*field-name*).
- *object-name* is the name of a Java object.

`attach` Command

The `attach` command attaches `dbx` to a running process, stopping execution and putting the program under debugging control. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`attach` *process-ID*	Begin debugging the program with process ID *process-ID*. `dbx` finds the program using `/proc`.
`attach -p` *process-ID* *program-name*	Begin debugging *program-name* with process ID *process-ID*.
`attach` *program-name process-ID*	Begin debugging *program-name* with process ID *process-ID*. *program-name* can be `-`. `dbx` finds it using `/proc`.

`attach -r ...`	The `-r` option causes `dbx` to retain all `watch` commands, `display` commands, `when` commands, and `stop` commands. With no `-r` option, an implicit `delete all` command and `undisplay 0` command are performed.

where:

process-ID is the process ID of a running process.

program-name is the path name of the running program.

For information on how to attach `dbx` to a running Java process, see "Attaching `dbx` to a Running Java Application" on page 228.

`bsearch` Command

The `bsearch` command searches backward in the current source file. It is valid only in native mode.

Syntax

`bsearch` *string*	Search backward for *string* in the current file.
`bsearch`	Repeat search, using the last search string.

where:

string is a character string.

`call` Command

In native mode, the `call` command calls a procedure. In Java mode, the `call` command calls a method.

You can also use the `call` command to call a function. To display the return value use the `print` command.

Occasionally the called function hits a breakpoint. You can choose to continue using the `cont` command or abort the call by using `pop -c`. The latter method is useful also if the called function causes a segmentation fault.

Native Mode Syntax

`call [-lang` *language*`] [-resumeone] [-m] [+m]` *procedure* `([`*parameters*`])`

where:

language is the language of the called procedure.

procedure is the name of the procedure.

parameters are the procedure's parameters.

`-lang` specifies the language of the called procedure and tells `dbx` to use the calling conventions of the specified language. This option is useful when the procedure being called was compiled without debugging information and `dbx` does not know how to pass parameters.

`-resumeone` resumes only one thread when the procedure is called. For more information, see "Resuming Execution" on page 174.

`-m` specifies that macro expansion be applied to the procedure and parameters when the `dbxenv` variable `macro_expand` is set to `off`.

`+m` specifies that macro expansion be skipped when the `dbxenv` variable `macro_expand` is set to `on`.

Java Mode Syntax

`call [`*class-name*`.|`*object-name*`.]` *method-name* `([`*parameters*`])`

where:

class-name is the name of a Java class. You can use either of the following:

- The package path using a period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`
- The full path name preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers. For example, `#test1/extra/T1$Inner`. Enclose *class-name* in quotation marks if you use the `$` qualifier.

object-name is the name of a Java object.

method-name is the name of a Java method.

parameters are the method's parameters.

`cancel` Command

The `cancel` command cancels the current signal. It is primarily used within the body of a `when` command (see "when Command" on page 413). It is valid only in native mode.

Signals are normally cancelled when `dbx` stops because of a signal. If a `when` command is attached to a signal event, the signal is not automatically cancelled. The `cancel` command can be used to explicitly cancel the signal.

`catch` Command

The `catch` command catches the given signals. It is valid only in native mode.

Catching a given signal causes `dbx` to stop the program when the process receives that signal. If you continue the program at that point, the signal is not processed by the program.

Syntax

`catch`	Print a list of the caught signals.
`catch` *number number* ...	Catch signals numbered *number.*
`catch` *signal signal* ...	Catch signals named by *signal*. `SIGKILL` cannot be caught or ignored.
`catch $(ignore)`	Catch all signals.

where:

number is the number of a signal.

signal is the name of a signal.

check Command

The check command enables checking of memory access, leaks, or usage and prints the current status of runtime checking (RTC). It is valid only in native mode.

The features of runtime checking that are enabled by this command are reset to their initial state by the debug command.

Syntax

This section provides information about the options for the check command.

check [*functions*] [*files*] [*loadobjects*]

Equivalent to check -all; suppress all; unsuppress all in *functions*, *files*, and *loadobjects*

where:

functions is one or more function names.

files is one or more file names.

loadobjects is one or more load object names.

You can use this to focus runtime checking on places of interest.

Note - To detect all errors, RTC does not require the program be compiled with -g. However, symbolic (-g) information is sometimes needed to guarantee the correctness of certain errors (mostly read from uninitialized memory). For this reason certain errors (rui for a.out and rui + aib + air for shared libraries) are suppressed if no symbolic information is available. This behavior can be changed by using suppress and unsuppress.

-access Option

The -access option enables checking. RTC reports the following errors:

baf Bad free

duf	Duplicate free
maf	Misaligned free
mar	Misaligned read
maw	Misaligned write
oom	Out of memory
rob	Read from array out-of-bounds memory
rua	Read from unallocated memory
rui	Read from uninitialized memory
wob	Write to array out-of-bounds memory
wro	Write to read-only memory
wua	Write to unallocated memory

The default behavior is to stop the process after detecting each access error, which can be changed using the rtc_auto_continue dbxenv variable. When set to on, access errors are logged to a file. The log file name is controlled by the dbxenv variable rtc_error_log_file_name.

By default, each unique access error is only reported the first time it happens. You can change this behavior using the dbxenv variable rtc_auto_suppress. The default setting of this variable is on.

-leaks Option

The syntax for the -leaks option is:

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

Enable leak checking. RTC reports the following errors:

aib	Possible memory leak – The only pointer points in the middle of the block
air	Possible memory leak – Pointer to the block exists only in register

mel Memory leak – No pointers to the block

With leak checking enabled, an automatic leak report is generated when the program exits. All leaks including possible leaks are reported at that time. By default, a non-verbose report is generated, which can be changed through the dbxenv variable rtc_mel_at_exit. However, you can ask for a leak report at any time (see "showleaks Command" on page 382).

-frames *n* implies that up to *n* distinct stack frames are displayed when reporting leaks. -match *m* is used for combining leaks; if the call stack at the time of allocation for two or more leaks matches *n* frames, then these leaks are reported in a single combined leak report.

The default value of *n* is 8 or the value of *m* (whichever is larger). Maximum value of *n* is 16. The default value of *m* is 8.

-memuse Option

The syntax for the -memuse option is:

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

The -memuse option behaves similarly to the -leaks option and also enables a blocks-in-use report (biu) when the program exits. By default, a non-verbose blocks in use report is generated, which can be changed through the dbxenv variable rtc_biu_at_exit. At any time during program execution you can see where the memory in your program has been allocated (see "showmemuse Command" on page 383).

-frames *n* implies that up to *n* distinct stack frames will be displayed while reporting memory use and leaks. Use -match *m* to combine these reports. 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 memory leak report.

The default value of *n* is 8 or the value of *m*, whichever is larger. The maximum value of *n* is 16. The default value of *m* is 8.

-all Option

The syntax for the -all option is:

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

Equivalent to:

`check -access` and `check -memuse [-frames` *n*`] [-match` *m*`]`

The value of the `dbxenv` variable `rtc_biu_at_exit` is not changed with `check -all`, so by default no memory use report is generated at exit. See "`dbx` Command" on page 326 for the description of the `rtc_biu_at_exit` environment variable.

`clear` Command

The `clear` command clears breakpoints. It is valid only in native mode.

Event handlers created using the `stop` command, `trace` command, or `when` command with the `inclass` argument, `inmethod` argument, `infile` argument, or `infunction` argument create sets of breakpoints. If the *line* you specify in the `clear` command matches one of these breakpoints, only that breakpoint is cleared. Once cleared in this manner, an individual breakpoint belonging to a set cannot be enabled again. However, disabling and then enabling the relevant event handler re-establishes all the breakpoints.

Syntax

`clear [`*filename: line*`]`

where:

line is the number of a source code line, such that all breakpoints are cleared at the specified line.

filename is the name of a source code file, such that all breakpoints at line *line* are cleared in the specified file.

If no file or line is specified, all breakpoints are cleared at the current stopping point.

`collector` Command

The `collector` command collects performance data for analysis by the Performance Analyzer. It is valid only in native mode.

This section lists the collector commands and provides details about them.

Syntax

`collector archive` *options*	Specify the mode for archiving an experiment when it terminates.
`collector dbxsample` *options*	Control the collection of samples when `dbx` stops the target process.
`collector disable`	Stop data collection and close the current experiment.
`collector enable`	Enable the collector and open a new experiment .
`collector heaptrace` *options*	Enable or disable collection of heap tracing data.
`collector hwprofile` *options*	Specify hardware counter profiling settings.
`collector limit` *options*	Limit the amount of profiling data recorded.
`collector pause`	Stop collecting performance data but leave experiment open.
`collector profile` *options*	Specify settings for collecting callstack profiling data.
`collector resume`	Start performance data collection after pause.
`collector sample` *options*	Specify sampling settings.
`collector show` *options*	Show current collector settings.
`collector status`	Inquire status about current experiment.
`collector store` *options*	Experiment file control and settings.
`collector synctrace` *options*	Specify settings for collecting thread synchronization wait tracing data.
`collector tha` *options*	Specify settings for collecting thread analyzer data.

`collector version`	Report the version of `libcollector.so` that would be used to collect data.

where:

To start collecting data, type `collector enable`.

To stop data collection, type `collector disable`.

collector archive Command

The `collector archive` command specifies the archiving mode to be used when the experiment terminates.

Syntax

`collector archive on|off|copy`	By default, normal archiving is used. For no archiving, specify `off`. To copy load objects into the experiment for portability, specify `copy`.

collector dbxsample Command

The `collector dbxsample` command specifies whether to record a sample when the process is stopped by `dbx`.

Syntax

`collector dbxsample on|off`	By default, a sample is collected when the process is stopped by `dbx`. To indicate not to collect a sample at this time, specify `off`.

collector disable Command

The `collector disable` command causes the data collection to stop and the current experiment to be closed.

collector enable Command

The collector enable command enables the collector and opens a new experiment.

collector heaptrace Command

The collector heaptrace command specifies options for collecting heap tracing (memory allocation) data.

Syntax

collector heaptrace on|off	By default, heap tracing data is not collected. To collect this data, specify on.

collector hwprofile Command

The collector hwprofile command specifies options for collecting hardware-counter overflow profiling data.

Syntax

collector hwprofile on|off	By default, hardware-counter overflow profile data is not collected. To collect this data, specify on.
collector hwprofile list	Print out the list of available counters.
collector hwprofile counter on|hi|high|lo|low|off	By default, hardware-counter overflow profile data is not collected. To collect this data, specify on. You can set the resolution of the counters to high or low. If you do not specify a resolution, it is set to normal. These options are similar to the collect command options. See the collect(1) man page for more information.

`collector hwprofile addcounter on|off`	Add additional counters for hardware counter overflow profiles.
`collector hwprofile counter` *name interval* `[`*name2 interval2*`]`	Specify hardware counter names and intervals.

where:

name is the name of a hardware counter.

interval is the collection interval in milliseconds.

name2 is the name of a second hardware counter.

interval2 is the collection interval in milliseconds.

Hardware counters are system-specific, so the choice of counters available depends on the system you are using. Many systems do not support hardware-counter overflow profiling. On these machines, the feature is disabled.

`collector limit` Command

The `collector limit` command specifies the experiment file size limit.

Syntax

`collector limit` *value* `| unlimited | none`

where:

value, in megabytes, limits the amount of profiling data recorded and must be a positive number.When the limit is reached, no more profiling data is recorded but the experiment remains open and sample points continue to be recorded. By default, there is no limit on the amount of data recorded.

If you have set a limit, specify `unlimited` or `none` to remove the limit.

collector pause Command

The collector pause command causes the data collection to stop but leaves the current experiment open. Sample points are not recorded while the Collector is paused. A sample is generated prior to a pause, and another sample is generated immediately following a resume. Data collection can be resumed with the collector resume command.

collector profile Command

The collector profile command specifies options for collecting profile data.

Syntax

collector profile on|off	Specify the profile data collection mode.
collector profile timer *interval*	Specify profile timer period, fixed or floating point, with an optional trailing m for milliseconds or u for microseconds.

collector resume Command

The collector resume command causes the data collection to resume after a pause created by the collector pause command (see "collector pause Command" on page 321).

collector sample Command

The collector sample command specifies the sampling mode and the sampling interval.

Syntax

collector sample periodic|manual	Specify sampling mode.

`collector sample period` *seconds*	Specify sampling interval in *seconds.*
`collector sample record [`*name*`]`	Record a sample with an optional *name*.

where:

seconds is the length of the sampling interval.

name is the name of the sample.

`collector show` Command

The `collector show` command shows the settings of one or more categories of options.

Syntax

`collector show`	Show all settings
`collector show all`	Show all settings
`collector show archive`	Show archive setting
collector show duration	Show duration setting
`collector show hwprofile`	Show hardware counter data settings
`collector show heaptrace`	Show heap tracing data settings
`collector show limit`	Show experiment size limits
collector show pausesig	Show pause and resume signal

`collector show profile`	Show call stack profiling settings
`collector show sample`	Show sample settings
collector show samplesig	Show sample signal
`collector show store`	Show store settings
`collector show synctrace`	Show thread synchronization wait tracing settings
`collector show tha`	Show thread analyzer data settings

`collector status` Command

The `collector status` command inquires about the status of the current experiment. It returns the working directory and the experiment name.

`collector store` Command

The `collector store` command specifies the directory and file name where an experiment is stored.

Syntax

`collector store {-directory` *pathname* `| -filename` *filename* `| -group` *string*`}`

where:

pathname is the pathname of the directory where an experiment is to be stored.

filename is the name of the experiment file.

string is the name of an experiment group.

collector synctrace Command

The collector synctrace command specifies options for collecting synchronization wait tracing data.

Syntax

collector synctrace on|off	By default, thread synchronization wait tracing data is not collected. To collect this data, specify on.
collector synctrace threshold {*microseconds*|calibrate}	Specify threshold in microseconds. The default value is 1000. If calibrate is specified, the threshold value will be calculated automatically.

where:

microseconds is the threshold below which synchronization wait events are discarded.

collector tha Command

The collector tha command specifies options for collecting thread analyzer data.

Syntax

collector tha on|off	By default, thread analyzer data is not collected. To collect this data, specify on.

collector version Command

The collector version command reports the version of libcollector.so that would be used to collect data.

Syntax

```
collector version
```

cont Command

The `cont` command causes the process to continue execution. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`cont`	Continue execution. In a multithreaded process, all threads are resumed. Use Control-C to stop executing the program.
`cont ... -sig` *signal*	Continue execution with signal *signal*.
`cont ...` *ID*	The *id* specifies which thread or LWP to continue.
`cont at` *line* `[`*ID*`]`	Continue execution at line *line*. *ID* is required if the application is multithreaded.
`cont ... -follow parent|child|both`	If the `dbx` `follow_fork_mode` environment variable is set to `ask` and you have chosen `stop`, use this option to choose which process to follow. `both` is only applicable in the Oracle Developer Studio IDE.

dalias Command

The `dalias` command defines a `dbx`-style (`csh`-style) alias. It is valid only in native mode.

Syntax

`dalias [`*name* `[`*definition*`]]`	(`dbx` alias) List all currently defined aliases.

If a name is specified, list the definition, if any, of alias *name*.

If a definition is also specified, define *name* to be an alias for *definition*. *definition* can contain white space. A semicolon or newline terminates the definition.

where:

name is the name of an alias

definition is the definition of an alias.

`dbx` accepts the following `csh` history substitution meta-syntax, which is commonly used in aliases:

`!:<n>`

`!-<n>`

`!^`

`!$`

`!*`

The `!` usually needs to be preceded by a backslash. For example:

```
dalias goto "stop at \!:1; cont; clear"
```

For more information, see the `csh`(1) man page.

`dbx` Command

The `dbx` command starts `dbx`.

Native Mode Syntax

`dbx` *options program-name* [*core* | *process-ID*]	Debug *program-name*. If *core* is specified, debug *program-name* with corefile *core*. If *process-ID* is specified, debug *program-name* with process ID *process-ID*.

`dbx` *options* `-` {*process-ID*|*core*}	If *process ID* is specified, debug process ID *process-ID*; `dbx` finds the program using `/proc`. If *core* is specified, debug with corefile *core*.
`dbx` *options* `-` *core*	Debug using corefile *core*.
`dbx` *options* `-r` *program-name arguments*	Run *program-name* with arguments *arguments*. If abnormal termination, start debugging *program-name*, else just exit.

where:

program-name is the name of the program to be debugged.

process-ID is the process ID of a running process.

arguments are the arguments to be passed to the program.

options are the options listed in "Options" on page 328.

Java Mode Syntax

`dbx` *options program-name*`{.class	.jar}`	Debug *program-name*.
`dbx` *options program-name*`{.class	.jar}` *process-ID*	Debug *program-name* with process ID *process ID*.
`dbx` *options* `-` *process-ID*	Debug process ID *process ID*; `dbx` finds the program using `/proc`.
`dbx` *options* `{ -r |-a}` *program-name*`{.class	.jar}` *arguments*	Run *program-name* with arguments *arguments*. If abnormal termination, start debugging *program-name*, else, just exit.

where:

program-name is the name of the program to be debugged.

process-id is the process ID of a running process.

arguments are the arguments to be passed to the program (not to the JVM software).

options are the options listed in "Options" on page 328.

Options

The following table lists the options of the dbx command for both native mode and Java mode:

--a *arguments*	Load program with program arguments *arguments*.
--B	Suppress all messages; return with exit code of program being debugged.
-c *commands*	Execute *commands* before prompting for input.
-C	Preload the Runtime Checking library (see "check Command" on page 313).
-d	Used with -s, removes *file* after reading.
-e	Echo input commands.
-f	Force loading of core file, even if it does not match.
-h	Print the usage help on dbx.
-I *dir*	Add *dir* to pathmap set (see "pathmap Command" on page 368).
-k	Save and restore keyboard translation state.
-q	Suppress messages about reading stabs.
-r	Run program; if program exits normally, exit.
-R	Print the README file on dbx.
-s *file*	Use *file* instead of /*current-dir*/.dbxrc or $HOME/.dbxrc as the startup file
-S	Suppress reading of initialization file /*install-dir*/lib/dbxrc.
-V	Print the version of dbx.
-w *n*	Skip *n* frames on where command.
-x exec32	Run the 32-bit dbx binary instead of the 64-bit dbx binary that runs by default on systems running a 64-bit OS.
--	Marks the end of the option list; use this if the program name starts with a dash.

dbxenv Command

The dbxenv command is used to list or set dbxenv variables. It has the same syntax and functionality in native mode and Java mode.

Syntax

`dbxenv [`*environment-variable setting*`]`	Display the current settings of the `dbxenv` variables. If a `dbxenv` variable is specified, set the `dbxenv` variable to *setting*.

where:

environment-variable is a `dbxenv` variable.

setting is a valid setting for that variable.

`debug` Command

The `debug` command lists or changes the program being debugged. In native mode, it loads the specified application and begins debugging the application. In Java mode, it loads the specified Java application, checks for the existence of the class file, and begins debugging the application.

Native Mode Syntax

`debug`	Print the name and arguments of the program being debugged.
`debug` *program-name*	Begin debugging *program-name* with no process or core.
`debug -c` *core program-name*	Begin debugging *program-name* with core file *core*.
`debug -p` *process-ID program-name*	Begin debugging *program-name* with process ID *process-ID*.
`debug` *program-name core*	Begin debugging *program* with core file *core*. *program-name* can be `-`. `dbx` will attempt to extract the name of the executable from the core file. For details, see "Debugging a Core File" on page 40.
`debug` *program-name process-ID*	Begin debugging *program-name* with process ID *process-ID*. *program-name* can be `-`; `dbx` finds it using `/proc`.
`debug -f ...`	Force loading of a core file, even if it does not match.

`debug -r ...`	The `-r` option causes `dbx` to retain all `display`, `trace`, `when`, and `stop` commands. With no `-r` option, an implicit `delete all` and `undisplay 0` are performed.
`debug -clone ...`	The `-clone` option causes another `dbx` process to begin execution, permitting debugging of more than one process at a time. Valid only if running in the Oracle Developer Studio IDE.
`debug -clone`	Starts another `dbx` process debugging nothing. Valid only if running in the Oracle Developer Studio IDE.
`debug [`*options*`] -- `*program-name*	Start debugging *program-name* even if *program-name* begins with a dash.

where:

core is the name of a core file.

options are the options listed in"Options" on page 331.

process-ID is the process ID of a running process.

program-name is the path name of the program.

Leaks checking and access checking are disabled when a program is loaded with the `debug` command. You can enable them with the `check` command.

Java Mode Syntax

`debug`	Print the name and arguments of the program being debugged.
`debug `*program-name*` [.class | .jar]`	Begin debugging *program-name* with no process.
`debug -p `*process-ID program-name*` [.class | .jar]`	Begin debugging *program-name* with process ID *process-ID*.
`debug `*program-name*` [.class | .jar] `*process-ID*	Begin debugging *program-name* with process ID *process-ID*. *program-name* can be `-`; `dbx` finds it using `/proc`.
`debug -r`	The `-r` option causes `dbx` to retain all `watch` commands, `display` commands, `trace` commands, `when` commands, and `stop` commands.

	With no `-r` option, an implicit `delete all` command and `undisplay 0` command are performed.
`debug -clone ...`	The `-clone` option causes another `dbx` process to begin execution, permitting debugging of more than one process at a time. Valid only if running in the Oracle Developer Studio IDE.
`debug -clone`	Starts another `dbx` process debugging nothing. Valid only if running in the Oracle Developer Studio IDE.
`debug [`*options*`] --` *program-name*`{.class | .jar}`	Start debugging *program-name* even if *program-name* begins with a dash.

where:

options are the options listed in"Options" on page 331.

process-ID is the process ID of a running process.

program-name is the path name of the program.

Options

`-c` *commands*	Execute *commands* before prompting for input.
`-d`	Used with `-s`, removes
`-e`	Echo input commands.
`-I` *directory_name*	Add *directory_name* to `pathmap` set (see "`pathmap` Command" on page 368.
`-k`	Save and restore keyboard translation state.
`-q`	Suppress messages about reading stabs.
`-r`	Run program; if program exits normally, then exit.
`-R`	Print the readme file for `dbx`.
`-s` *file*	Use *file* instead of *current_directory*`/.dbxrc` or `$HOME/.dbxrc` as the startup file

-S	Suppress reading of initialization file /*install-dir*/lib/dbxrc.
-V	Print the version of dbx.
-w *n*	Skip *n* frames on where command.
--	Marks the end of the option list; use this if the program name starts with a dash.

delete Command

The delete command deletes breakpoints and other events. It has the same syntax and functionality in native mode and Java mode.

Syntax

delete [-h] *handler-ID* ...	Remove trace commands, when commands, or stop commands of given *handler-ID*s. To remove hidden handlers, you must include the -h option.
delete [-h] 0 | all | -all	Remove all trace commands, when commands, and stop commands excluding permanent and hidden handlers. Specifying -h removes hidden handlers as well.
delete -temp	Remove all temporary handlers.
delete $firedhandlers	Delete all the handlers that caused the latest stoppage.

where:

handler-ID is the identifier of a handler.

detach Command

The detach command releases the target process from dbx's control.

Native Mode Syntax

detach [-sig *signal* | -stop]	Detach dbx from the target, and cancel any pending signals. If the -sig option is specified, detach while forwarding the given *signal*. If the -stop option is specified, detach dbx from the target and leave the process in a stopped state. This option allows temporary application of other/proc-based debugging tools that might be blocked due to exclusive access. For an example, "Detaching dbx From a Process" on page 89.

where:

signal is the name of a signal.

Java Mode Syntax

detach	Detach dbx from the target, and cancel any pending signals.

dis Command

The dis command disassembles machine instructions. It is valid only in native mode.

Syntax

dis [-a] *address* [/*count*]	Disassemble *count* instructions (default is 10), starting at address *address*.
dis *address1*, *address2*	Disassemble instructions from *address1* through *address2*.
dis	Disassemble 10 instructions, starting at the value of +.

where:

address is the address at which to start disassembling. The default value of *address* is the address after the last address previously assembled. This value is shared by the examine command.

address1 is the address at which to start disassembling.

address2 is the address at which to stop disassembling.

count is the number of instructions to disassemble. The default value of *count* is 10.

Options

`-a`	When used with a function address, disassembles the entire function. When used without parameters, disassembles the remains of the current visiting function, if any.

`display` Command

In native mode, the `display` command re-evaluates and prints expressions at every stopping point. In Java mode, the `display` command evaluates and prints expressions, local variables, or parameters at every stopping point. Object references are expanded to one level and arrays are printed itemwise.

The expression is parsed for the current scope at the time you type the command and reevaluated at every stopping point. Because the expression is parsed at entry time, the correctness of the expression can be immediately verified.

If you are running `dbx` in the IDE or `dbxtool` in the Sun Studio 12 release, the Sun Studio 12 Update 1 release, the Oracle Solaris Studio 12.2 release, or later updated releases, the `display` *expression* command effectively behaves like a `watch $(which` *expression*`)` command.

Native Mode Syntax

`display`	Print the list of expressions being displayed.
`display` *expression*`, ...`	Display the value of expressions *expression*, ... at every stopping point. Because *expression* is parsed at entry time, the correctness of the expression is immediately verified.
`display [-r|+r|-d|+d|-S|+S|-p|`	See “`print` Command” on page 370 for the meaning of these flags.

+p|-L|-f*format*|-
F*format*|-m|+m|--]
expression, ...

where:

expression is a valid expression.

format is the output format you want used to print the expression. For information on valid formats, see "print Command" on page 370.

Java Mode Syntax

display	Print the list of variables and parameters being displayed.
display *expression*|*identifier*, ...	Display the value of variables and parameters of *identifier*, ... at every stopping point.
display [-r|+r|-d|+d|-p|+p|-f*format*|-F*format*|-F*format*|--] *expression*|*identifier*, ...	See "print Command" on page 370 for the meaning of these flags.

where:

class-name is the name of a Java class. You can use either of the following:

- The package path using a period (.) as a qualifier; for example, test1.extra.T1.Inner
- The full path name preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers. For example, #test1/extra/T1$Inner. Enclose *class-name* in quotation marks if you use the $ qualifier.

expression is a valid Java expression.

field-name is the name of a field in the class.

format is the output format you want used to print the expression. For information about valid formats, see "print Command" on page 370.

identifier is a local variable or parameter, including this, the current class instance variable (*object-name*.*field-name*) or a class (static) variable (*class-name*.*field-name*).

object-name is the name of a Java object.

down Command

The down command moves down the call stack (away from main). It has the same syntax and functionality in native mode and Java mode.

Syntax

down	Move down the call stack one level.
down *number*	Move down the call stack *number* levels.
down -h [*number*]	Move down the call stack, but do not skip hidden frames.

where:

number is a number of call stack levels.

dump Command

The dump command prints all variables local to a procedure. It has the same syntax and functionality in native mode and Java mode.

Syntax

dump [*procedure*]	Print all variables local to the current procedure. If a procedure is specified, print all variables local to *procedure*.

where:

procedure is the name of a procedure.

`edit` Command

The `edit` command invokes `$EDITOR` on a source file. It is valid only in native mode.

The `edit` command uses `$EDITOR` if `dbx` is not running in the Oracle Developer Studio IDE. Otherwise, it sends a message to the IDE to display the appropriate file.

Syntax

`edit [`*filename* `|` *procedure*`]`	Edit the current file. If a file name is specified, edit the specified file *filename*. If a procedure is specified, edit the file containing function or procedure *procedure*.

where:

filename is the name of a file.

procedure is the name of a function or procedure.

`examine` Command

The `examine` command shows memory contents. It is valid only in native mode.

The `x` command is an alias for the `examine` command.

Syntax

`examine [`*address*`] [ / [`*count*`] [`*format*`]]`	Display the contents of memory starting at *address* for *count* items in format *format*.
`examine` *address1*`,` *address2* `[ / [`*format*`]]`	Display the contents of memory from *address1* through *address2* inclusive, in format *format*.

`examine` *address*`=`[*format*]	Display the address (instead of the contents of the address) in the given format. The *address* can be +, which indicates the address just after the last one previously displayed (the same as omitting it). x is a predefined alias for `examine`.

where:

address is the address at which to start displaying memory contents. The default value of *address* is the address after the address whose contents were last displayed. This value is shared by the `dis` command.

address1 is the address at which to start displaying memory contents.

address2 is the address at which to stop displaying memory contents.

count is the number of addresses from which to display memory contents. The default value of *count* is 1.

format is the format in which to display the contents of memory addresses. The default format is X (hexadecimal) for the first `examine` command, and the format specified in the previous `examine` command for subsequent `examine` commands. The following values are valid for *format*:

`o,O`	octal (2 or 4 bytes)
`x,X`	hexadecimal (2 or 4 bytes)
`b`	octal (1 byte)
`c`	character
`w`	wide character
`s`	string
`W`	wide character string
`f`	hexadecimal and floating point (4 bytes, 6-digit precision)
`F`	hexadecimal and floating point (8 bytes, 14-digit precision)
`g`	same as `F`

`E`	hexadecimal and floating point (16 bytes, 14-digit precision)
`ld,lD`	decimal (4 bytes, same as `D`)
`lo,lO`	octal 94 bytes, same as `O`
`lx,lX`	hexadecimal (4 bytes, same as `X`)
`Ld,LD`	decimal (8 bytes)
`Lo,LO`	octal (8 bytes)
`Lx,LX`	hexadecimal (8 bytes)

`exception` Command

The `exception` command prints the value of the current C++ exception. It is valid only in native mode.

Syntax

`exception [-d | +d]`	Prints the value of the current C++ exception, if any.

where:

`-d` enables showing dynamic exceptions.

`+d` disables showing dynamic exceptions.

`exists` Command

The `exists` command checks for the existence of a symbol name. It is valid only in native mode.

Syntax

`exists` *name*	Returns 0 if *name* is found in the current program, 1 if *name* is not found.

where:

name is the name of a symbol.

`file` Command

The `file` command lists or changes the current file. It has the same syntax and functionality in native mode and in Java mode.

Syntax

`file` *filename*	Print the name of the current file. If a file name is specified, change the current file.

where:

filename is the name of a file.

`files` Command

In native mode, the `files` command lists file names that match a regular expression. In Java mode, the `files` command lists all of the Java source files known to `dbx`. If your Java source files are not in the same directory as the `.class` or `.jar` files, `dbx` might not find them unless you have set the `$JAVASRCPATH` environment variable. For more information, see "Specifying the Location of Your Java Source Files" on page 230.

Native Mode Syntax

`files`	List the names of all files that contributed debugging information to the current program (those that were compiled with `-g`).

`files` *regular-expression*	List the names of all files compiled with `-g` that match the given regular expression.

where:

regular-expression is a regular expression.

For example:

```
(dbx) files ^r
myprog:
retregs.cc
reg_sorts.cc
reg_errmsgs.cc
rhosts.cc
```

Java Mode Syntax

`files`	List the names of all of the Java source files known to `dbx`.

`fix` Command

The `fix` command recompiles modified source files and dynamically links the modified functions into the application. It is valid only in native mode. It is not valid on Linux platforms.

Syntax

`fix [`*file-name file-name* ...`]` `[-options]`	Fix the current file. If file names are listed, fix files in list.

where:

`-options` are the following valid options.

`-f`	Force fixing the file, even if source has not been modified.
`-a`	Fix all modified files.

-g	Strip -O flags and add -g flag.
-c	Print compilation line (can include some options added internally for use by dbx).
-n	Do not execute compile/link commands (use with -v).
v	Verbose mode (overrides dbx fix_verbose environment variable setting).
+v	Non-verbose mode (overrides dbx fix_verbose environment variable setting).

fixed Command

The fixed command lists the names of all fixed files. It is valid only in native mode.

fortran_modules Command

The fortran_modules command lists the Fortran modules in the current program, or the functions or variables in one of the modules.

Syntax

fortran_modules [-f *module-name* | -v *module-name*]	Lists all Fortran modules in the current program. If the -f option is specified, list all functions in the specified module. If the -v option is specified, lists all variables in the specified module.

frame Command

The frame command lists or changes the current stack frame number. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

`frame`	Display the frame number of the current frame.
`frame [-h]` *number*	Set the current frame to frame *number*.
`frame [-h]` `+[`*number*`]`	Go *number* frames up the stack; default is 1.
`frame [-h] -` `[`*number*`]`	Go *number* frames down the stack; default is 1.
`-h`	Go to frame, even if frame is hidden.

where:

number is the number of a frame in the call stack.

`func` Command

In native mode, the `func` command lists or changes the current function. In Java mode, the `func` command lists or changes the current method.

Native Mode Syntax

`func [`*procedure*`]`	Print the name of the current function. If a procedure is specified, change the current function to the function or procedure *procedure*.

where:

procedure is the name of a function or procedure.

Java Mode Syntax

`func`	Print the name of the current method.

`func [`*class-name*`.]`*method-name* `[(`*parameters*`)]`	Change the current function to the method *method-name*.

where:

class-name is the name of a Java class. You can use either of the following:

- The package path using a period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`
- The full path name preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers. For example, `#test1/extra/T1$Inner`. Enclose *class-name* in quotation marks if you use the `$` qualifier.

method-name is the name of a Java method.

parameters are the method's parameters.

`funcs` Command

The `funcs` command lists all function names that match a regular expression. It is valid only in native mode.

Syntax

`funcs [-f` *filename*`] [-g] [`*regular-expression*`]`	List all functions in the current program, If `-f` *filename* is specified, list all functions in the file. If `-g` is specified, list all functions with debugging information. If *filename* ends in `.o`, then all functions, including those created automatically by the compiler, are listed. Otherwise, only functions appearing in the source code are listed. If *regular-expression* is specified, list all functions that match the regular expression.

where:

filename is the name of the file for which you wish to list all the functions.

regular-expression is the regular expression for which you wish to list all the matching functions.

For example:

```
(dbx) funcs [vs]print
"libc.so.1"isprint
"libc.so.1"wsprintf
"libc.so.1"sprintf
"libc.so.1"vprintf
"libc.so.1"vsprintf
```

gdb Command

The `gdb` command supports the GDB command set. It is valid only in native mode.

Syntax

`gdb on | off`	Use `gbd on` to enter the GDB command mode under which `dbx` understands and accepts GDB commands. To exit the GDB command mode and return to the `dbx` command mode, type `gdb off`. `dbx` commands are not accepted while in GDB command mode and GDB commands are not accepted while in `dbx` mode. All debugging settings such as breakpoints are preserved across different command modes.

The following GDB commands are not supported in this release:

- `commands`
- `define`
- `handle`
- `hbreak`
- `interrupt`
- `maintenance`
- `printf`
- `rbreak`
- `return`
- `signal`
- `tcatch`
- `until`

`handler` Command

The `handler` command modifies event handlers (enable, disable, and such). It has identical syntax and identical functionality in native mode and in Java mode.

A handler is created for each event that needs to be managed in a debugging session. The commands `trace`, `stop`, and `when` create handlers. Each of these commands returns a number known as the *handler ID* (*handler-ID*). The `handler`, `status`, and `delete` commands manipulate or provide information about handlers in a generic fashion.

Syntax

`handler [-enable | -disable]` *handler-ID* `...`	Either enable or disable given handlers, specify *handler-ID* as all for all handlers. Use `$firedhandlers` instead of *handler-ID* to disable the handlers that caused the most recent stoppage.
`handler -count` *handler-ID new-limit*	Print value of trip counter for given handler. If a new limit parameter is specified, set new count limit for given event.
`handler -reset` *handler-ID*	Reset trip counter for given handler.

where:

handler-ID is the identifier of a handler.

`hide` Command

The `hide` command hides stack frames that match a regular expression. It is valid only in native mode.

Syntax

`hide` *regular-expression*	List the stack frame filters currently in effect. If regular-expression is specified, hide stack frames matching *regular-expression*. The regular expression matches either the function name, or

the name of the load object, and is a sh or ksh file matching style regular expression.

where:

regular-expression is a regular expression.

ignore Command

The ignore command tells the dbx process not to catch the given signals. It is valid only in native mode.

Ignoring a signal causes dbx not to stop when the process receives that kind of signal.

Syntax

ignore [*number* ... | *signal* ...]	Print a list of the ignored signals. If a signal number is specified, ignore signal numbered *number*. If a signal is specified, ignore signals named by *signal*. SIGKILL cannot be caught or ignored.

where:

number is the number of a signal.

signal is the name of a signal.

import Command

The import command imports commands from a dbx command library. It has the same syntax and functionality in native mode and in Java mode.

Syntax

import *path-name*	Import commands from the dbx command library *path-name*.

where:

path-name is the path name of a dbx command library.

intercept Command

The intercept command throws (C++ exceptions) of the given type (C++ only). It is valid only in native mode.

dbx stops when the type of a thrown exception matches a type on the intercept list unless the type of the exception also matches a type on the excluded list. A thrown exception for which there is no matching catch is called an "unhandled" throw. A thrown exception that does not match the exception specification of the function it is thrown from is called an "unexpected" throw.

Unhandled and unexpected throws are intercepted by default.

Syntax

intercept -x *excluded-typename* [, *excluded-typename* ...]	Add throws of *excluded-typename* to the excluded list.
intercept -a[ll] -x *excluded-typename* [, *excluded-typename*...]	Add all types except *excluded-typename* to the intercept list.
intercept -s [et] [*intercepted-typename* [, *intercepted-typename* ...]] [-x *excluded-typename* [, *excluded-typename*]]	Clear both the intercept list and the excluded list, and set the lists to intercept or exclude only throws of the specified types.

`intercept`	List intercepted types.

where:

included-typename and *excluded-typename* are exception type specifications such as `List <int>` or `unsigned short`.

`java` Command

The `java` command is used when `dbx` is in JNI mode to indicate that the Java version of a specified command is to be executed. It causes the specified command to use the Java expression evaluator, and when relevant, to display Java threads and stack frames.

Syntax

`java` *command*

where:

command is the command name and arguments of the command to be executed.

`jclasses` Command

The `jclasses` command prints the names of all Java classes known to `dbx` at the time you issue the command. It is valid only in Java mode.

Classes in your program that have not yet been loaded are not printed.

Syntax

`jclasses [-a]`	Print the names of all Java classes known to `dbx`. If the `-a` option is specified, print system classes as well as other known Java classes.

joff Command

The joffcommand switches dbx from Java mode or JNI mode to native mode.

jon Command

The jon command switches dbx from native mode to Java mode.

jpkgs Command

The jpkgs command prints the names of all Java packages known to dbx at the time you issue the command. It is valid only in Java mode.

Packages in your program that have not yet been loaded are not printed.

kill Command

The kill command sends a signal to a process. It is valid only in native mode.

Syntax

kill -l	List all known signal numbers, names, and descriptions.
kill	Kill the controlled process.
kill [*signal*]*job* ...	Send the SIGTERM signal to the listed jobs. If the -signal option is specified, send the given signal to the listed jobs.

where:

job can be a process ID or can be specified in any of the following ways:

%+	Kill the current job.
%-	Kill the previous job.
%*number*	Kill job number *number*.
%*string*	Kill the job that begins with *string*.
%?*string*	Kill the job that contains *string*.

where:

signal is the name of a signal.

`language` Command

The `language` command lists or changes the current source language. It is valid only in native mode.

Syntax

`language`	Print the current language mode set by the `dbx` `language_mode` environment variable. If the language mode is set to `autodetect` or `main`, the command also prints the name of the current language used for parsing and evaluating expressions.

where:

language is `c`, `c++`, `fortran`, or `fortran90`.

Note - `c` is an alias for `ansic`.

`line` Command

The `line` command lists or changes the current line number. It has the same syntax and functionality in native mode and in Java mode.

Syntax

`line [["`*file-name*`":]` `[`*number*`]]`	Display the current line number. If a number is specified, set the current line number to *number*. If a file name is specified, set current line number to line 1 in *filename*. If both are specified, set current line number to line *number* in *file-name*.

where:

filename is the name of the file in which to change the line number. The "" quotation marks around the file name are optional. They are useful when your file name contains spaces.

number is the number of a line in the file.

Examples

```
line 100
line "/root/test/test.cc":100
```

`list` Command

The `list` command displays lines of a source file. It has the same syntax and functionality in native mode and in Java mode.

The default number of lines listed, N, is controlled by the `dbx` `output_list_size` environment variable.

Syntax

`list`	List N lines.
`list` *number*	List line number *number*.
`list +`	List next N lines.

`list +`*n*	List next *n* lines.
`list -`	List previous N lines.
`list -`*n*	List previous *n* lines.
`list` *n1*`,` *n2*	List lines from *n1* to *n2*.
`list` *n1*`, +`	List from *n1* to *n1* + N.
`list` *n1*`, +`*n2*	List from *n1* to *n1* + *n2*.
`list` *n1*`, -`	List from *n1*-N to *n1*.
`list` *n1*`, -`*n2*	List from *n1*-*n2* to *n1*.
`list` *function*	List the start of the source for *function*. `list` *function* changes the current scope. See “Program Scope” on page 70 for more information.
`list` *filename*	List the start of the file *filename*.
`list` *filename*`:`*n*	List file *filename* from line *n*.

where:

filename is the file name of a source code file.

function is the name of a function to display.

number is the number of a line in the source file.

n is a number of lines to display.

n1 is the number of the first line to display.

n2 is the number of the last line to display. Where appropriate, the line number can be ”$” which denotes the last line of the file. Comma is optional.

Options

`-i` or `-instr`	Intermix source lines and assembly code.

-w or -w*n*	List N (or *n*) lines (window) around line or function. This option is not allowed in combination with the plus sign (+) or minus sign (-) syntax or when two line numbers are specified.
-a	When used with a function name, lists the entire function. When used without parameters, lists the remains of the current visiting function, if any.

Examples

```
list                       // list N lines starting at current line
list +5                       // list next 5 lines starting at current line
list -                     // list previous N lines
list -20                   // list previous 20 lines
list 1000                  // list line 1000
list 1000,$                // list from line 1000 to last line
list 2737 +24              // list line 2737 and next 24 lines
list 1000 -20              // list line 980 to 1000
list test.cc:33            // list source line 33 in file test.cc
list -w                    // list N lines around current line
list -w8 ”test.cc”func1    // list 8 lines around function func1
list -i 500 +10                 // list source and assembly code for line
                500 to line 510
```

listi Command

The listi command displays source and disassembled instructions. It is valid only in native mode. This command is the same as using list -i.

See “list Command” on page 352 for details.

loadobject Command

The loadobject command lists and manages symbolic information from load objects. It is valid only in native mode.

This section lists the loadobject options and provides details about them.

Syntax

`loadobject -list [`*regexp*`] [-a]`	Show currently loaded load objects.
`loadobject -load` *loadobject*	Load symbols for specified load object.
`loadobject -unload [`*regexp*`]`	Unload specified load objects.
`loadobject -hide [`*regexp*`]`	Remove load object from `dbx`'s search algorithm.
`loadobject -use [`*regexp*`]`	Add load object to `dbx`'s search algorithm.
`loadobject -dumpelf [`*regexp*`]`	Show various ELF details of the load object.
`loadobject -exclude` *ex-regexp*	Don't automatically load loadobjects matching *ex-regexp*.
`loadobject exclude -clear`	Clear the exclude list of patterns.

where:

regexp is a regular expression. If it is not specified, the command applies to all load objects.

ex-regexp is not optional, it must be specified.

This command has a default alias `lo`.

`loadobject -dumpelf` Command

The `loadobject -dumpelf` command shows various ELF details of the load object. It is valid only in native mode.

Syntax

`loadobject -dumpelf [`*regexp*`]`

where:

regexp is a regular expression. If it is not specified, the command applies to all load objects.

This command dumps out information related to the ELF structure of the load object file on disk. The details of this output are highly subject to change. If you want to parse this output, use the Oracle Solaris OS commands `dump` or `elfdump`.

`loadobject -exclude` Command

The `loadobject -exclude` command tells `dbx` not to automatically load loadobjects matching the specified regular expression.

Syntax

`loadobject -exclude` *ex-regexp* `[-clear]`

where:

ex-regexp is a regular expression.

This command prevents `dbx` from automatically loading symbols for load objects that match the specified regular expression. Unlike *regexp* in other `loadobject` subcommands, if *ex-regexp* is not specified, it does not default to all. If you do not specify *ex-regexp*, the command lists the excluded patterns that have been specified by previous `loadobject -exclude` commands.

If you specify `-clear`, the list of excluded patterns is deleted.

Currently this functionality cannot be used to prevent loading of the main program or the runtime linker. Also, using it to prevent loading of C++ runtime libraries could cause the failure of some C++ functionality.

This option should not be used with runtime checking (RTC).

`loadobject -hide` Command

The `loadobject -hide` command removes load objects from `dbx`'s search algorithm.

Syntax

`loadobject -hide [`*regexp*`]`

where:

regexp is a regular expression. If it is not specified, the command applies to all load objects.

This command removes a load object from the program scope, and hides its functions and symbols from `dbx`. This command also resets the “preload” bit. For more information, refer to the `dbx` help file by typing the following into the `dbx` prompt.

```
(dbx) help loadobject preloading
```

`loadobject -list` Command

The `loadobject -list` command shows currently loaded loadobjects. It is valid only in native mode.

Syntax

`loadobject -list [`*regexp*`] [-a]`

where:

regexp is a regular expression. If it is not specified, the command applies to all load objects.

The full path name for each load object is shown along with letters in the margin to show status. Load objects that are hidden are listed only if you specify the `-a` option.

`h`	This letter means “hidden” (the symbols are not found by symbolic queries like `whatis` or `stop in`).
`u`	If there is an active process, `u` means “unmapped.”
`p`	This letter indicates a load object that is preloaded, that is, the result of a `loadobject -load` command or a `dlopen` event in the program.

For example:

```
(dbx) lo -list libm
/usr/lib/64/libm.so.1
/usr/lib/64/libmp.so.2
(dbx) lo -list ld.so
h  /usr/lib/sparcv9/ld.so.1 (rtld)
```

This last example shows that the symbols for the runtime linker are hidden by default. To use those symbols in `dbx`commands, see "`loadobject -use` Command" on page 359.

`loadobject -load` Command

The `loadobject -load` command loads symbols for specified load objects. It is valid only in native mode.

Syntax

`loadobject -load` *load-object*

where:

load-object can be a full path name or a library in `/usr/lib`, `/usr/lib/sparcv9` or `/usr/lib/amd64`. If a program is being debugged, then only the proper ABI library directory will be searched.

`loadobject -unload` Command

The `loadobject -unload` command unloads specified load objects. It is valid only in native mode.

Syntax

`loadobject -unload [`*regexp*`]`

where:

regexp is a regular expression. If it is not specified, the command applies to all load objects.

This command unloads the symbols for any load objects matching the *regexp* supplied on the command line. The main program loaded with the `debug` command cannot be unloaded. `dbx` might also refuse to unload other load objects that might be currently in use or critical to the proper functioning of `dbx`.

`loadobject -use` Command

The `loadobject -use` command adds load objects from dbx's search algorithm. It is valid only in native mode.

Syntax

`loadobject -use [`*regexp*`]`

where:

regexp is a regular expression. If it is not specified, the command applies to all load objects.

`lwp` Command

The `lwp` command lists or changes the current LWP (lightweight process). It is valid only in native mode.

Note - The `lwp` command is available only on Oracle Solaris platforms.

Syntax

`lwp`	Display current LWP.
`lwp` *lwp-ID*	Switch to LWP *lwp-ID*.
`lwp -info`	Displays the name, home, and masked signals of the current LWP.

`lwp [`*lwp-ID*`] -setfp` *address-expression*	Tells `dbx` that the `fp` register has the value *address-expression*. The state of the program being debugged is not changed. A frame pointer set with the `-setfp` option is reset to its original value upon resuming execution.
`lwp [`*lwp-ID*`] -resetfp`	Sets the frame pointer logical value from the register value in the current process or core file, undoing the effect of a previous `lwp -setfp` command.

where:

lwp-ID is the identifier of a lightweight process.

If the command is used with both an LWP ID and an option, the corresponding action is applied to LWP specified by the *lwp-ID*, but the current LWP is not changed.

The `-setfp` and `-resetfp` options are useful when the frame pointer (`fp`) of the LWP is corrupted. In this event, `dbx` cannot reconstruct the call stack properly and evaluate local variables. These options work when debugging a core file, where `assign $fp=...` is unavailable.

To make changes to the fp register visible to the application being debugged, use the `assign $fp=`*address-expression* command.

`lwps` Command

The `lwps` command lists all LWPs (lightweight processes) in the process. It is valid only in native mode.

Note - The `lwps` command is available only on Oracle Solaris platforms.

`macro` Command

The macro command prints the macro expansion of an expression.

Syntax

`macro` *expression*`, ...`

mmapfile Command

The mmapfile command views the contents of memory mapped files that are missing from a core dump. It is valid only in native mode.

Oracle Solaris core files do not contain any memory segments that are read-only. Executable read-only segments (that is, text) are dealt with automatically and dbx resolves memory accesses against these by looking into the executable and the relevant shared objects.

Syntax

mmapfile *mmapped-file address offset length*	View contents of memory mapped files missing from core dump.

where:

mmapped-file is the file name of a file that was memory mapped during a core dump.

address is the starting address of the address space of the process.

length is length in bytes of the address space to be viewed.

offset is the offset in bytes to the starting address in *mmapped-file*.

Example

Read-only data segments typically occur when an application memory maps a database. For example:

```
caddr_t vaddr = NULL;
off_t offset = 0;
size_t = 10 * 1024;
int fd;
fd = open("../DATABASE", ...)
vaddr = mmap(vaddr, size, PROT_READ, MAP_SHARED, fd, offset);
index = (DBIndex *) vaddr;
```

The following command enables access to the database through the debugger as memory:

```
mmapfile ../DATABASE $[vaddr] $[offset] $[size]
```

Then, to look at your database contents in a structured way:

```
print *index
```

module Command

The `module` command reads debugging information for one or more modules. It is valid only in native mode.

Syntax

`module [-v]`	Print the name of the current module.
`module [-f] [-v] [-q] {`*name*`	-a}`	If *name* is specified, read in debugging information for the module called *name*. If `-a` is specified, read in debugging information for all modules.

where:

name is the name of a module for which to read debugging information.

`-a` specifies all modules.

`-f` forces reading of debugging information, even if the file is newer than the executable. Use this option with caution!

`-v` specifies verbose mode, which prints language, file names, and such.

`-q` specifies quiet mode.

modules Command

The `modules` command lists module names. It is valid only in native mode.

Syntax

`modules [-v] [-debug |-read]`	List all modules.

If `-debug` is specified, list all modules containing debugging information.

If `-read` is specified, list names of modules containing debugging information that have been read in already.

where:

`-v` specifies verbose mode, which prints language, file names, and such.

`native` Command

The `native` command is used when `dbx` is in Java mode to indicate that the native version of a specified command is to be executed. Preceding a command with native results in `dbx` executing the command in native mode. This means that expressions are interpreted and displayed as C expressions or C++ expressions, and certain other commands produce different output than they do in Java mode.

This command is useful when you are debugging Java code but you want to examine the native environment.

Syntax

`native` *command*

where:

command is the command name and arguments of the command to be executed.

`next` Command

The `next` command steps one source line (stepping over calls).

The `dbx` `step_events` environment variable (see "Setting `dbxenv` Variables" on page 60) controls whether breakpoints are enabled during a step.

Native Mode Syntax

`next`	Step one line (step over calls). With multithreaded programs when a function call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.
`next` *n*	Step *n* lines (step over calls).
`next ... -sig` *signal*	Deliver the specified signal while stepping.
`next ...` *thread-ID*	Step the specified thread.
`next ...` *lwp-ID*	Step the given LWP. Will not implicitly resume all LWPs when stepping over a function.

where:

n is the number of lines to step.

signal is the name of a signal.

thread-ID is a thread ID.

lwp-ID is an LWP ID.

When an explicit *thread-id* or *lwp-ID* is included, the deadlock avoidance measure of the generic `next` command is defeated.

See also "`nexti` Command" on page 365 for machine-level stepping over calls.

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

Java Mode Syntax

`next`	Step one line (step over calls). With multithreaded programs when a function call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.

`next` *n*	Step *n* lines (step over calls).
`next ...` *thread-ID*	Step the given thread.
`next ...` *lwp-ID*	Step the given LWP. Will not implicitly resume all LWPs when stepping over a function.

where:

n is the number of lines to step.

thread-ID is a thread identifier.

lwp-ID is an LWP identifier.

When an explicit *thread-ID* or *lwp-ID* is included, the deadlock avoidance measure of the generic `next` command is defeated.

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

`nexti` Command

The `nexti` command steps one machine instruction (stepping over calls). It is valid only in native mode.

Syntax

`nexti`	Step one machine instruction (step over calls).
`nexti` *n*	Step *n* machine instructions (step over calls).
`nexti -sig` *signal*	Deliver the given signal while stepping.
`nexti ...` *lwp-ID*	Step the given LWP.
`nexti ...` *thread-ID*	Step the LWP on which the given thread is active. Will not implicitly resume all LWPs when stepping over a function.

where:

n is the number of instructions to step.

signal is the name of a signal.

thread-ID is a thread ID.

lwp-ID is an LWP ID.

omp_loop Command

The omp_loop command prints a description of the current loop, including scheduling (static, dynamic, guided, auto, or runtime), ordered or not, bounds, steps or strides, and number of iterations. You can issue the command only from the thread that is currently executing a loop.

omp_pr Command

The omp_pr command prints a description of the current or specified parallel region, including the parent region, parallel region id, team size (number of threads), and program location (program counter address).

Syntax

omp_pr	Print a description of the current parallel region.
omp_pr *parallel-region-ID*	Print a description of the specified parallel region. This command does not cause dbx to switch the current parallel region to the specified region.
omp_pr -ancestors	Print descriptions of all the parallel regions along the path from the current parallel region to the root of the current parallel region tree.
omp_pr *parallel-region-ID* -ancestors	Print descriptions of all the parallel regions along the path from the specified parallel region to its root.
omp_pr -tree	Print a description of the whole parallel region tree.
omp_pr -v	Print a description of the current parallel region with team member information.

omp_serialize Command

The omp_serialize command serializes the execution of the next encountered parallel region for the current thread or for all threads in the current team. The serialization applies only to that one trip into the parallel region and does not persist.

Be sure you are in the right place in the program when you use this command. A logical place is just before a parallel directive.

Syntax

omp_serialize [-team]	Serialize the execution of the next encountered parallel region for the current thread. If -team is specified, do this for all threads in the current team.

omp_team Command

The omp_team command prints all the threads in the current team.

Syntax

omp_team [*parallel-region-ID*]	Print all the threads in the current team. If a parallel region ID is specified, print all the threads in the team for the specified parallel region.

omp_tr Command

The omp_tr command prints a description of the current task region, including the task region ID, type (implicit or explicit), state (spawned, executing, or waiting), executing thread, program location (program counter address), unfinished children, and parent.

Syntax

`omp_tr`	Print a description of the current task region.
`omp_tr` *task-region-ID*	Print a description of the specified task region. This command does not cause `dbx` to switch the current task region to the specified task region.
`omp_tr -ancestors`	Print descriptions of all the task regions along the path from the current task region to the root of the current task region tree.
`omp_tr` *task-region-ID* `-ancestors`	Print descriptions of all the task regions along the path from the specified task region to its root.
`omp_tr -tree`	Print a description of the whole task region tree.

`pathmap` Command

The `pathmap` command maps one path name to another for finding source files and such. The mapping is applied to source paths, object file paths, and the current working directory (if you specify -c). During macro skimming, it is also applied to include directory paths. The `pathmap` command has the same syntax and functionality in native mode and in Java mode.

The `pathmap` command is useful for dealing with automounted and explicit NFS mounted filesystems with different paths on differing hosts. Current working directories are inaccurate on automounted filesystems. Specify `-c` when you are trying to correct problems arising due to the automounter. The `pathmap` command is also useful if source or build trees are moved.

`pathmap /tmp_mnt /` exists by default.

The `pathmap` command is used to find load objects for core files when the `dbxenv` variable `core_lo_pathmap` is set to `on`. Other than this case, the `pathmap` command has no effect on finding load objects (shared libraries). For more information, see "Debugging a Mismatched Core File" on page 42.

Syntax

`pathmap [ -c ] [ -`*index* `]` *from to*	Establish a new mapping from *from* to *to*.

`pathmap [ -c ]` `[ -`*index* `]` *to*	Map all paths to *to*.
`pathmap`	List all existing path mappings (by index).
`pathmap -s`	The same, but the output can be read by `dbx`.
`pathmap -d` *from1* *from2* `...`	Delete the given mappings by path.
`pathmap -d` *index1* *index2* `...`	Delete the given mappings by index.

where:

from and *to* are path prefixes. *from* refers to the path compiled into the executable or object file and *to* refers to the path at debug time.

from1 is the path of the first mapping to be deleted.

from2 is the path of the last mapping to be deleted.

index specifies the index with which the mapping is to be inserted in the list. If you do not specify an index, the mapping is added to the end of the list.

index1 is the index of the first mapping to be deleted.

index2 is the index of the last mapping to be deleted.

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

If you specify `-s`, the existing mappings are listed in an output format that `dbx` can read.

If you specify `-d`, the specified mappings are deleted.

Examples

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2
  # maps /export/home/work1/abc/test.c to /net/mmm/export/home/work2/abc/test.c
(dbx) pathmap /export/home/newproject
  # maps /export/home/work1/abc/test.c to /export/home/newproject/test.c
(dbx) pathmap
(1) -c /tmp_mnt /
(2) /export/home/work1 /net/mmm/export/home/work2
(3) /export/home/newproject
```

`pop` Command

The `pop` command removes one or more frames from the call stack. It is valid only in native mode.

You can pop only to a frame for a function that was compiled with `-g`. The program counter is reset to the beginning of the source line at the call site. You cannot pop past a function call made by the debugger; but must use `pop -c`.

Normally, a `pop` command calls all the C++ destructors associated with the popped frames. You can override this behavior by setting the `dbx` `pop_auto_destruct` environment variable to `off`.

Syntax

`pop`	Pop the current top frame from stack.
`pop` *number*	Pop *number* frames from stack.
`pop -f` *number*	Pop frames from stack until specified frame *number*.
`pop -c`	Pop the last call made from the debugger.

where:

number is the number of frames to pop from the stack.

`print` Command

In native mode, the `print` command prints the value of an expression. In Java mode, the `print` command prints the value of an expression, local variable, or parameter.

Native Mode Syntax

`print` *expression*`, ...`	Print the value of the expression *expression*, ...

`print -r` *expression*	Print the value of the expression *expression* including its inherited members.
`print +r` *expression*	Do not print inherited members when the `dbx` `output_inherited_members` environment variable is set to `on`.
`print -d [-r]` *expression*	Show dynamic type of expression *expression* instead of static type.
`print +d [-r]` *expression*	Don't use dynamic type of expression *expression* when the `dbx` `output_dynamic_type` environment variable is set to `on`.
`print -s` *expression*	Print the value of expression *expression* for each thread in the current OpenMP parallel region if the expression contains private or thread-private variables.
`print -S [-r] [-d]` *expression*	Print the value of expression *expression* including its static members (C++ only)
`print +S [-r] [-d]` *expression*	Don't print static members when the `dbxenv` variable `show_static_members` is set to `on` (C++ only).
`print -p` *expression*	Call the `prettyprint` function.
`print +p` *expression*	Do not call the `prettyprint` function when the `dbx` `output_pretty_print` environment variable is on.
`print -L` *expression*	If the printing object *expression* is larger than 4K, enforce the printing.
`print +l` *expression*	If the expression is a string (`char *`), print the address only, do not print the literal.
`print -l` *expression*	('Literal') Do not print the left side. If the expression is a string (`char *`), do not print the address, just print the raw characters of the string, without quotes.
`print -f`*format* *expression*	Use *format* as the format for integers, strings, or floating-point expressions.
`print -F`*format* *expression*	Use the given format but do not print the left hand side (the variable name or expression).

`print -o` *expression*	Print the value of *expression*, which must be an enumeration as an ordinal value. You can also use a format string here (`-f`*format*). This option is ignored for non-enumeration expressions.
`print –m` *expression*	Apply macro expansion to *expression* when the `dbxenv` variable `macro_expand` is set to `off`.
`print +m` *expression*	Skip macro expansion of expression when the `dbxenv` variable `macro_expand` is set to `on`.
`print --` *expression*	"--" signals the end of flag arguments. This is useful if *expression* can start with a plus or minus. See"Program Scope" on page 70 for scope resolution rules.

where:

expression is the expression whose value you want to print.

format is the output format you want used to print the expression. If the format does not apply to the given type, the format string is silently ignored and `dbx` uses its built-in printing mechanism.

The allowed formats are a subset of those used by the `printf`(3S) command. The following restrictions apply:

- No `n` conversion.
- No `*` for field width or precision.
- No `%<digits>$` argument selection.
- Only one conversion specification per format string.

  The allowed forms are defined by the following simple grammar:

  ```
  FORMAT ::= CHARS % FLAGS WIDTH PREC MOD SPEC CHARS

  CHARS ::= <any character sequence not containing a %>

  | %%

  | <empty>

  | CHARS CHARS

  FLAGS ::= + | - | <space> | # | 0 | <empty>

  WIDTH ::= <decimal_number> | <empty>

  PREC ::= . | . <decimal_number> | <empty>

  MOD ::= h | l | L | ll | <empty>
  ```

SPEC ::= d | i | o | u | x | X | f | e | E | g | G |

c | wc | s | ws | p

If the given format string does not contain a %, dbx automatically prepends one. If the format string contains spaces, semicolons, or tabs, the entire format string must be surrounded by double quotes.

Java Mode Syntax

print *expression*, ... | ...	Print the values of the expressions *expression*, ... or identifier *identifier*, ...
print -r *expression* | *identifier*	Print the value of *expression* or *identifier* including its inherited members.
print +r *expression* | *identifier*	Do not print inherited members when the dbx output_inherited_members environment variable is set to on.
print -d [-r] *expression* | *identifier*	Show dynamic type of *expression* or *identifier* instead of static type.
print +d [-r] *expression* | *identifier*	Do not use dynamic type of *expression* or *identifier* when the dbx output_dynamic_type environment variable is set to on.
print -- *expression* | *identifier*	”--’ signals the end of flag arguments. This is useful if *expression* can start with a plus or minus. See “Program Scope” on page 70 for scope resolution rules.

where:

class-name is the name of a Java class. You can use either of the following:

- The package path using a period (.) as a qualifier; for example, test1.extra.T1.Inner
- The full path name preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers. For example, #test1/extra/T1$Inner. Enclose *class-name* in quotation marks if you use the $ qualifier.

expression is the Java expression whose value you want to print.

field-name is the name of a field in the class.

identifier is a local variable or parameter, including `this`, the current class instance variable (*object-name*.*field-name*) or a class (static) variable (*class-name*.*field-name*).

object-name is the name of a Java object.

proc Command

The `proc` command displays the status of the current process. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

`proc {-cwd | -map | -pid}`	If `-cwd` is specified, show the current working directory of the current process. If `-map` is specified, show the list of load objects with addresses. If `-process-ID` is specified, show current process ID (process-ID).

prog Command

The `prog` command manages programs being debugged and their attributes. It has the same syntax and functionality in native mode and Java mode.

Syntax

`prog -readsyms`	Read symbolic information which was postponed by having set the `dbx` `run_quick` environment variable to `on`.
`prog -executable`	Prints the full path of the executable, `-` if the program was attached to using `-`.
`prog -argv`	Prints the whole `argv`, including `argv[0]`.

`prog -args`	Prints the `argv`, excluding `argv[0]`.
`prog -stdin`	Prints < *filename* or empty if `stdin` is used.
`prog -stdout`	Prints > *filename* or >> *filename* or empty if `stdout` is used. The outputs of `-args`, `-stdin`, `-stdout` are designed so that the strings can be combined and reused with the `run` command.

`quit` Command

The `quit` command exits `dbx`. It has the same syntax and functionality in native mode and Java mode.

If `dbx` is attached to a process, the process is detached from before exiting. If there are pending signals, they are cancelled. Use the `detach` command for fine control.

Syntax

`quit`	Exit `dbx` with return code 0. Same as `exit`.
`quit` *n*	Exit with return code *n*. Same as `exit` *n*.

where:

n is a return code.

`regs` Command

The `regs` command prints the current value of registers. It is valid only in native mode.

Syntax

```
regs [-f] [-F]
```

where:

-f includes floating-point registers (single precision) (SPARC platform only)

-F includes floating-point registers (double precision) (SPARC platform only)

Example (SPARC platform)

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3              0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7              0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3              0x00000003 0x00000014 0xef7562b4 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
```

replay Command

The replay command replays debugging commands since the last run, rerun, or debug command. It is valid only in native mode.

Syntax

replay [-*number*]	Replay all or all minus *number* commands since last run command, rerun command, or debug command.

where:

number is the number of commands not to replay.

rerun Command

The rerun command runs the program with no arguments. It has the same syntax and functionality in native mode and Java mode.

Syntax

rerun	Begin executing the program with no arguments.
rerun *arguments*	Begin executing the program with new arguments by the save command (see “save Command” on page 381).

restore Command

The restore command restores dbx to a previously saved state. It is valid only in native mode.

Syntax

restore [*filename*]

where:

filename is the name of the file to which the dbx commands executed since the last run, rerun, or debug command were saved.

rprint Command

The rprint command prints an expression using shell quoting rules. It is valid only in native mode.

Syntax

`rprint [-r|+r|-d|+d|-S|+S|-p|+p|-L|-l|-f` *format* `| -F`*format* `| -- ]` *expression*	Print the value of the expression. No special quoting rules apply, so `rprint a > b` puts the value of `a` (if it exists) into file `b`. See "`print` Command" on page 370 for the meanings of the flags.

where:

expression is the expression whose value you want to print.

format is the output format you want used to print the expression. For information about valid formats, see "`print` Command" on page 370.

`rtc showmap` Command

The `rtc showmap` command reports the address range of program text categorized by instrumentation type (branches and traps). It is valid only in native mode.

This command is intended for expert users. Runtime checking instruments program text for access checking. The instrumentation type can be a branch or a trap instruction based on available resources. The `rtc showmap` command reports the address range of program text categorized by instrumentation type. This map can be used to find an optimal location for adding patch area object files and to avoid the automatic use of traps. See"Runtime Checking Limitations" on page 162 for details.

`rtc skippatch` Command

The `rtc skippatch` command excludes load objects, object files, and functions from being instrumented by runtime checking. The effect of the command is permanent to each `dbx` session unless the load object is unloaded explicitly.

Because `dbx` does not track memory access in load objects, object files, and functions affected by this command, incorrect `rui` errors might be reported for functions that were not skipped. `dbx` cannot determine whether an `rui` error was introduced by this command, so this type error was not suppressed automatically.

Syntax

`rtc skippatch` *load-object* `[-o` *object-file* `...]` `[-f` *function* `...]`	Exclude the specified object files and functions in the specified load object from being instrumented.

where:

load-object is the name of a load object or the path to the name of a load object.

object-file is the name of an object file.

function is the name of a function.

`run` Command

The `run` command runs the program with arguments.

Use Control-C to stop executing the program.

Native Mode Syntax

`run`	Begin executing the program with the current arguments.
`run` *arguments*	Begin executing the program with new arguments.
`run ... >|>>` *output-file*	Set the output redirection.
`run ... < input-file`	Set the input redirection.

where:

arguments are the arguments to be used in running the target process.

input-file is the file name of the file from which input is to be redirected.

output-file is the file name of the file to which output is to be redirected.

Note - There is currently no way to redirect `stderr` using the `run` or `runargs` command.

Java Mode Syntax

`run`	Begin executing the program with the current arguments.
`run` *arguments*	Begin executing the program with new arguments.

where:

arguments are the arguments to be used in running the target process. They are passed to the Java application, not to the JVM software. Do not include the main class name as an argument.

You cannot redirect the input or output of a Java application with the `run` command.

Breakpoints you set in one run persist in subsequent runs.

`runargs` Command

The `runargs` command changes the arguments of the target process. It has identical syntax and identical functionality in native mode and Java mode.

Use the `debug` command with no arguments to inspect the current arguments of the target process.

Syntax

`runargs` *arguments*	Set the current arguments, to be used by the `run` command (see "`run` Command" on page 379).
`runargs ... >|` >>*file*	Set the output redirection to be used by the `run` command.
`runargs ...` <*file*	Set the input redirection to be used by the `run` command.
`runargs`	Clear the current arguments.

where:

arguments are the arguments to be used in running the target process.

file is the file to which output from the target process or input to the target process is to be redirected.

save Command

The save command saves commands to a file. It is valid only in native mode.

Syntax

save [-*number*] [*filename*]	Save all or all minus *number* commands since last run command, rerun command, or debug command to the default file or *filename*.

where:

number is the number of commands not to save.

filename is the name of the file to save the dbx commands executed since the last run, rerun, or debug command.

scopes Command

The scopes command prints a list of active scopes. It is valid only in native mode.

search Command

The search command searches forward in the current source file. It is valid only in native mode.

Syntax

search *string*	Search forward for *string* in the current file.

search	Repeat search, using last search string.

where:

string is the character string for which you wish to search.

showblock Command

The showblock command shows where the particular heap block was allocated from runtime checking. It is valid only in native mode.

When runtime checking is turned on, the showblock command shows the details about the heap block at the specified address. The details include the location of the blocks' allocation and its size.

Syntax

showblock -a *address*

where:

address is the address of a heap block.

showleaks Command

Note - The showleaks command is available only on Oracle Solaris platforms.

In the default non-verbose case, a one-line report per leak record is printed. Actual leaks are reported followed by the possible leaks. Reports are sorted according to the combined size of the leaks.

Syntax

showleaks [-a] [-m *m*] [-n *number*] [-v]

where:

`-a` shows all the leaks generated so far, not just the leaks since the last `showleaks` command.

`-m` *m* combines leaks; if the call stack at the time of allocation for two or more leaks matches *m* frames, then these leaks are reported in a single combined leak report. If the `-m` option is given, it overrides the global value of *m* specified with the `check` command.

`-n` *number* shows up to *number* records in the report. The default is to show all records.

`-v` Generate verbose output. The default is to show non-verbose output.

`showmemuse` Command

A one-line report per block-in-use record is printed. The commands sorts the reports according to the combined size of the blocks. Any leaked blocks since the last `showleaks` command are also included in the report.

Syntax

`showmemuse [-a] [-m` *m*`] [-n` *number*`] [-v]`

where:

`-a` shows all the blocks in use (not just the blocks since the last `showmemuse` command).

`-m` *m* combines the blocks-in-use reports. The default value of *m* is 8 or the global value last given with the `check` command. If the call stack at the time of allocation for two or more blocks matches *m* frames, then these blocks are reported in a single combined report. If the `-m` option is given, it overrides the global value of *m*.

`-n` *number* shows up to *number* records in the report. The default is 20.

`-v` generates verbose output. The default is to show non-verbose output.

`source` Command

The `source` command executes commands from a given file. It is valid only in native mode.

Syntax

`source` *filename*	Execute commands from file *filename*. `$PATH` is not searched.

`status` Command

The `status` command lists event handlers (breakpoints and such). It has identical syntax and identical functionality in native mode and Java mode.

Syntax

`status`	Print `trace`, `when`, and `stop` breakpoints in effect.
`status` *handler-ID*	Print status for handler *handler-ID*.
`status -h`	Print `trace`, `when`, and `stop` breakpoints in effect including the hidden ones.
`status -s`	The same, but the output can be read by `dbx`.

where:

handler-ID is the identifier of an event handler.

Example

```
(dbx) status -s > bpts
...
(dbx) source bpts
```

`step` Command

The `step` command steps one source line or statement, stepping into calls that were compiled with the `-g` option.

The dbx step_events environment variable controls whether breakpoints are enabled during a step.

The dbx step_granularity environment variable controls granularity of source line stepping.

The dbx step_abflow environment variable controls whether dbx stops when it detects that abnormal control flow change is about to occur. This type of control flow change can be caused by a call to siglongjmp() or longjmp() or an exception throw.

Native Mode Syntax

`step`	Single-step one line (step into calls). With multithreaded programs when a function call is stepped over, all threads are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.
`step` *n*	Single-step *n* lines (step into calls).
`step up`	Step up and out of the current function.
`step ... -sig` *signal*	Deliver the specified signal while stepping. If a signal handler exists for the signal, step into it if the signal handler was compiled with the `-g` option.
`step ...`*thread-ID*	Step the specified thread. Does not apply to `step up`.
`step ...`*lwp-ID*	Step the specified LWP. Does not implicitly resume all LWPs when stepping over a function.

`step to [` *function* `]`	Attempts to step into *function* called from the current source code line. If *function* is not specified, steps into the last function called, helping to avoid long sequences of `step` commands and `step up` commands. Examples of the last function are:
	`f()->s()-t()->last();`
	`last(a() + b(c()->d()));`

where:

n is the number of lines to step.

signal is the name of a signal.

thread-ID is a thread ID.

lip-ID is an LWP ID.

function is a function name.

When an explicit *lwp*ID is specified, the deadlock avoidance measure of the generic `step` command is defeated.

When executing the `step to` command, while an attempt is made to step into the last assembly call instruction or step into a function (if specified) in the current source code line, the call might not be taken due to a conditional branch. In a case where the call is not taken or no function call is in the current source code line, the `step to` command steps over the current source code line. Take special consideration on user-defined operators when using the `step to` command.

See also "`stepi` Command" on page 386 for machine-level stepping.

Java Mode Syntax

`step`	Single-step one line (step into calls). With multithreaded programs when a method call is stepped over, all threads are implicitly resumed for the duration of that method call in order to avoid deadlock. Non-active threads cannot be stepped.
`step` *n*	Single-step *n* lines (step into calls).
`step up`	Step up and out of the current method.
`step ...`*thread-ID*	Step the specified thread. Does not apply to `step up`.
`step ...`*lwp-ID*	Step the specified LWP. Does not implicitly resume all LWPs when stepping over a method.

`stepi` Command

The `stepi` command steps one machine instruction (stepping into calls). It is valid only in native mode.

Syntax

`stepi`	Single-step one machine instruction (step into calls).
`stepi` *n*	Single step *n* machine instructions (step into calls).
`stepi -sig` *signal*	Step and deliver the specified signal.
`stepi ...`*lwp-ID*	Step the given LWP.
`stepi ...`*thread-ID*	Step the LWP on which the specified thread is active.

where:

n is the number of instructions to step.

signal is the name of a signal.

lwp-ID is an LWP ID.

thread-ID is a thread ID.

`stop` Command

The `stop` command sets a source-level breakpoint.

Syntax

The `stop` command has the following general syntax:

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

When the specified event occurs, the process is stopped.

Native Mode Syntax

This section describes some of the more important syntaxes that are valid in native mode. For information about additional events, see "Setting Event Specifications" on page 274.

`stop [ -update ]`	Stop execution now. Only valid within the body of a `when` command.
`stop -noupdate`	Stop execution now but do not update the Oracle Developer Studio IDE Debugger windows.
`stop access` *mode address-expression* `[,`*byte-size-expression* `]`	Stop execution when the memory specified by *address-expression* has been accessed. See also "Stopping Execution When an Address Is Accessed" on page 102.
`stop at` *line-number*	Stop execution at *line-number*. See "Setting a Breakpoint at a Line of Source Code" on page 98.
`stop change` *variable*	Stop execution when the value of *variable* has changed.
`stop cond` *condition-expression*	Stop execution when the condition denoted by *condition-expression* evaluates to true.
`stop in` *function*	Stop execution when *function* is called. See "Setting a Breakpoint in a Function" on page 99.
`stop inclass` *class-name* `[ -recurse | -norecurse ]`	C++ only: Set breakpoints on all member functions of a class, struct, union, or template class. `-norecurse` is the default. If `-recurse` is specified, the base classes are included. See also "Setting Breakpoints in All Member Functions of a Class" on page 100.
`stop infile` *file-name*	Stop execution when any function in *filename* is called.
`stop infunction` *name*	C++ only: Set breakpoints on all non-member functions *name*.
`stop inmember` *name*	C++ only: set breakpoints on all member functions *name*. See "Setting Breakpoints in Member Functions of Different Classes" on page 100.
`stop inobject` *object-expression* `[ -recurse | -norecurse ]`	C++ only: set breakpoint on entry into any non-static method of the class and all its base classes when called from the object *object-expression*. `-recurse` is the default. If `-norecurse` is specified, the base classes are not included. See "Setting Breakpoints in Objects" on page 101.

line-number is the number of a source code line.

function is the name of a function.

class-name is the name of a C++ class, struct, union, or template class.

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

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

mode can also contain the following:

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

name is the name of a C++ function.

object-expression identifies a C++ object.

variable is the name of a variable.

The following modifiers are valid in native mode.

`-if` *condition-expression*	The specified event occurs only when *condition-expression* evaluates to true.
`-in` *function*	Execution stops only if the specified event occurs during the extent of *function*.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs.
`-disable`	Create the breakpoint in a disabled state.
`-instr`	Do instruction-level variation. For example, `step` becomes instruction level stepping, and `at` takes a text address for an argument instead of a line number.

`-perm`	Make this event permanent across `debug`. Certain events (like breakpoints) are not appropriate to be made permanent. `delete all` will not delete permanent handlers. To delete permanent handlers, use `delete` *hid*.
`-hidden`	Hide the event from the `status` command. Some import modules might choose to use this. Use `status -h` to see them.
`-lwp` *lwp-ID*	Execution stops only if the specified event occurs in the specified LWP.
`-thread` *thread-ID*	Execution stops only if the specified event occurs in the specified thread.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

`stop access` *mode class-name.field-name*	Stop execution when the memory specified by *class-name.field-name* has been accessed.
`stop at` *line-number*	Stop execution at *line-number*.
`stop at` *filename:line-number*	Stop execution at *line-number* in *filename*.
`stop change` *class-name.field-name*	Stop execution when the value of *field-name* in *class-name* has changed.
`stop classload`	Stop execution when any class is loaded.
`stop classload` *class-name*	Stop execution when *class-name* is loaded.
`stop classunload`	Stop execution when any class is unloaded.
`stop classunload` *class-name*	Stop execution when *class-name* is unloaded.
`stop cond` *condition-expression*	Stop execution when the condition denoted by *condition-expression* evaluates to true.

`stop in` *class-name.method-name*	Stop execution when *class-name.method-name* has been entered, and the first line is about to be executed. If no parameters are specified and the method is overloaded, a list of methods is displayed.
`stop in` *class-name.method-name*`([`*parameters*`])`	Stop execution when *class-name.method-name* has been entered, and the first line is about to be executed.
`stop inmethod` *class-name. method-name*	Set breakpoints on all non-member methods *class-name*`.`*method-name*.
`stop inmethod` *class-name. method-name* `([`*parameters*`])`	Set breakpoints on all non-member methods *class-name*`.`*method-name*.
`stop throw`	Stop execution when a Java exception has been thrown.
`stop throw` *type*	Stop execution when a Java exception of *type* has been thrown.

where:

class-name is the name of a Java class.. You can use either of the following:

- The package path using a period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`
- The full path name preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers. For example, `#test1/extra/T1$Inner`. Enclose *class-name* in quotation marks if you use the `$` qualifier.

condition-expression can be any expression, but it must evaluate to an integral type.

field-name is the name of a field in the class.

filename is the name of a file.

line-number is the number of a source code line.

method-name is the name of a Java method.

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

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

mode can also contain the following:

`b`	Stops the process before the access.

The program counter will point at the offending instruction.

parameters are the method's parameters.

type is a type of Java exception. `-unhandled` or `-unexpected` are valid values for *type*.

The following modifiers are valid in Java mode:

`-if` *condition-expression*	The specified event occurs only when *condition-expression* evaluates to true.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs.
`-disable`	Create the breakpoint in a disabled state.

See "`stopi` Command" on page 392 for information about setting a machine-level breakpoint.

For a list and the syntax of all events, see "Setting Event Specifications" on page 274.

`stopi` Command

The `stopi` command sets a machine-level breakpoint. It is valid only in native mode.

Syntax

The `stopi` command has the following general syntax:

`stopi` *event-specification* `[`*modifier*`]`

When the specified event occurs, the process is stopped.

The following specific syntaxes are valid:

`stopi at` *address-expression*	Stop execution at location *address-expression*.
`stopi in` *function*	Stop execution when *function* is called.

where:

address-expression is any expression resulting in or usable as an address.

function is the name of a function.

For a list and the syntax of all events, see "Setting Event Specifications" on page 274.

`suppress` Command

The `suppress` command suppresses reporting of memory errors during runtime checking. It is valid only in native mode.

If the `dbx` `rtc_auto_suppress` environment variable is set to `on`, the memory error at a given location is reported only once.

Syntax

`suppress`	History of `suppress` and `unsuppress` commands, not including those specifying the `-d` and `-reset` options.
`suppress -d`	List of errors being suppressed in functions not compiled for debugging (default suppression). This list is per load object. These errors can be unsuppressed only by using the `unsuppress` command with the `-d` option.
`suppress -d` *errors*	Modify the default suppressions for all load objects by further suppressing *errors*.
`suppress -d` *errors* `in` *load-objects*	Modify the default suppressions in the *load-objects* by further suppressing *errors*.

`suppress -last`	At error location suppress present error.
`suppress -reset`	Set the default suppression to the original value (startup time).
`suppress -r` *ID*`...`	Remove the unsuppress events as specified by the IDs, which can be obtained with the `unsuppress` command.
`suppress -r 0 | all | -all`	Remove all the unsuppress events as specified by the `unsuppress` command.
`suppress` *errors*	Suppress *errors* everywhere.
`suppress` *errors* `in [` *functions* `] [` *files* `] [` *load-objects* `]`	Suppress *errors* in list of *functions*, list of *files*, and list of *load-objects*.
`suppress` *errors* `at` *line*	Suppress *errors* at *line*.
`suppress` *errors* `at "`*file*`":`*line*	Suppress *errors* at *line* in *file*.
`suppress` *errors* `addr` *address*	Suppress *errors* at location *address*.

where:

address is a memory address.

errors are blank separated and can be any combination of the following:

all	All errors
`aib`	Possible memory leak - address in block
`air`	Possible memory leak - address in register
`baf`	Bad free
`duf`	Duplicate free
`mel`	Memory leak
`maf`	Misaligned free

`mar`	Misaligned read
`maw`	Misaligned write
`oom`	Out of memory
`rob`	Read from array out-of-bounds memory
`rua`	Read from unallocated memory
`rui`	Read from uninitialized memory
`wob`	Write to array out-of-bounds memory
`wro`	Write to read-only memory
`wua`	Write to unallocated memory
`biu`	Block in use (allocated memory). Though not an error, you can use `biu` just like *errors* in the `suppress` commands.

file is the name of a file.

files is the names of one or more files.

functions is one or more function names.

line is the number of a source code line.

load-objects is one or more load object names.

See "Suppressing Errors" on page 149 for more information about suppressing errors.

See "`unsuppress` Command" on page 408 for information about unsuppressing errors.

`sync` Command

The `sync` command shows information about a specified synchronization object. It is valid only in native mode.

Note - The `sync` command is available only on Oracle Solaris platforms.

Syntax

`sync -info` *address*	Show information about the synchronization object at *address*.

where:

address is the address of the synchronization object.

`syncs` Command

The `syncs` command lists all synchronization objects (locks). It is valid only in native mode.

Note - The `syncs` command is available only on Oracle Solaris platforms.

`thread` Command

The `thread` command lists or changes the current thread.

Native Mode Syntax

`thread`	Display current thread.
`thread` *thread-ID*	Switch to thread *thread-ID*.

In the following variations, the current thread is assumed if a thread ID is not specified.

`thread -info` [*thread-ID*]	Print everything known about the specified thread. For OpenMP threads, the information includes the OpenMP thread ID, parallel region ID, task region ID, and thread state.
`thread -hide` [*thread-ID*]	Hide the specified (or current) thread. It will not show up in the generic threads listing.
`thread -unhide` [*thread-ID*]	Unhide the specified (or current) thread.

`thread -unhide all`	Unhide all threads.
`thread -suspend` *thread-ID*	Keep the specified thread from ever running. A suspended thread shows up with an “S” in the threads list.
`thread -resume` *thread-ID*	Undo the effect of `-suspend`.
`thread -blocks [`*thread-ID*`]`	List all locks held by the specified thread blocking other threads.
`thread -blockedby [`*thread-ID*`]`	Show which synchronization object the specified thread is blocked by, if any.

where:

thread-ID is a thread ID.

Java Mode Syntax

`thread`	Display current thread.
`thread` *thread-ID*	Switch to thread *thread-ID*.

In the following variations, the current thread is assumed if a thread ID is not specified.

`thread -info [`*thread-ID*`]`	Print everything known about the specified thread.
`thread -hide [`*thread-ID*`]`	Hide the specified (or current) thread. It will not show up in the generic threads listing.
`thread -unhide [`*thread-ID*`]`	Unhide the specified (or current) thread.
`thread -unhide all`	Unhide all threads.
`thread -suspend` *thread-ID*	Keep the specified thread from ever running. A suspended thread shows up with an “S” in the threads list.

`thread -resume` *thread-ID*	Undo the effect of `-suspend`.
`thread -blocks` [*thread-ID*]	Lists the Java monitor owned by *thread-ID*.
`thread -blockedby` [*thread-id*]	Lists the Java monitor on which *thread-ID* is blocked.

where:

thread-ID is a `dbx`-style thread ID of the form `t@`*number* or the Java thread name specified for the thread.

`threads` Command

The `threads` command lists all threads.

Native Mode Syntax

`threads`	Print the list of all known threads.
`threads -all`	Print threads normally not printed (zombies).
`threads -mode all|filter`	Controls whether all threads are printed or threads are filtered. The default is to filter threads. When filtering is on, threads that have been hidden by the `thread -hide` command are not listed.
`threads -mode auto|manual`	Under the IDE, enables automatic updating of the thread listing.
`threads -mode`	Echo the current modes.

Each line of information is composed of the following:

- An * (asterisk) indicating that an event requiring user attention has occurred in this thread. Usually this is a breakpoint.

  An 'o’ instead of an asterisk indicates that a `dbx` internal event has occurred.

- An > (arrow) denoting the current thread.
- `t@`*num*, the thread ID, referring to a particular thread. The *number* is the `thread_t` value passed back by `thr_create`.
- `b l@`*num* meaning the thread is bound (currently assigned to the designated LWP), or `a l@`*num* meaning the thread is active (currently scheduled to run).
- The "Start function" of the thread as passed to `thr_create`. A `?()` means that the start function is not known.
- The thread state, which is one of the following:
  - `monitor`
  - `running`
  - `sleeping`
  - `unknown`
  - `wait`
  - `zombie`

  The function that the thread is currently executing.

Java Mode Syntax

`threads`	Print the list of all known threads.
`threads -all`	Print threads normally not printed (zombies).
`threads -mode all|filter`	Controls whether all threads are printed or threads are filtered. The default is to filter threads.
`threads -mode auto|manual`	Under the IDE, enables automatic updating of the thread listing.
`threads -mode`	Echo the current modes.

Each line of information in the listing is composed of the following:

- An > (arrow) denoting the current thread
- `t@`*number*, a `dbx`-style thread ID
- The thread state, which is one of the following:
  - `monitor`
  - `running`
  - `sleeping`

- unknown
- wait
- zombie
- The thread name in single quotation marks
- A number indicating the thread priority

trace Command

The trace command shows executed source lines, function calls, or variable changes.

The speed of a trace is set using the dbx trace_speed environment variable.

If dbx is in Java mode and you want to set a trace breakpoint in native code, switch to Native mode using the joff command or prefix the trace command with native.

If dbx is in JNI mode and you want to set a trace breakpoint in Java code, prefix the trace command with java.

Syntax

The trace command has the following general syntax:

trace *event-specification* [*modifier*]

When the specified event occurs, a trace is printed.

Native Mode Syntax

The following specific syntaxes are valid in native mode:

trace -file *filename*	Direct all trace output to the specified file name. To revert trace output to standard output use - for *filename*. Trace output is always appended to *filename*. It is flushed whenever dbx prompts and when the application has exited. The file is always re-opened on a new run or resumption after an attach.
trace step	Trace each source line, function call, and return.

`trace next -in` *function*	Trace each source line while in the specified function.
`trace at` *line-number*	Trace given source *line*.
`trace in` *function*	Trace calls to and returns from the specified function.
`trace infile` *filename*	Trace calls to and returns from any function in *filename*.
`trace inmember` *function*	Trace calls to any member function named *function*.
`trace infunction` *function*	Trace when any function named *function* is called.
`trace inclass` *class*	Trace calls to any member function of *class.*
`trace change` *variable*	Trace changes to the *variable.*

where:

filename is the name of the file to which you want trace output sent.

function is the name of a function.

line-number is the number of a source code line.

class is the name of a class.

variable is the name of a variable.

The following modifiers are valid in native mode.

`-if` *condition-expression*	The specified event occurs only when *condition-expression* evaluates to true.
`-in` *function*	Execution stops only if the specified event occurs in *function*.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.

`-temp`	Create a temporary breakpoint that is deleted when the event occurs.
`-disable`	Create the breakpoint in a disabled state.
`-instr`	Do instruction-level variation. For example, `step` becomes instruction-level stepping, and `at` takes a text address for an argument instead of a line number.
`-perm`	Make this event permanent across `debug`. Certain events like breakpoints are not appropriate to be made permanent. `delete all` will not delete permanent handlers. To delete permanent handlers, use `delete` *hid*.
`-hidden`	Hide the event from the `status` command. Some import modules might choose to use this. Use `status -h` to see them.
`-lwp` *lwp-ID*	Execution stops only if the specified event occurs in the given LWP.
`-thread` *thread-ID*	Execution stops only if the specified event occurs in the given thread.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

`trace -file` *filename*	Direct all trace output to the specified *filename*. To revert trace output to standard output use - for *filename*. Trace output is always appended to *filename*. It is flushed whenever `dbx`prompts and when the application has exited. The file is always re-opened on a new run or resumption after an attach.
`trace at` *line-number*	Trace *line-number*.
`trace at` *filename.line-number*	Trace specified source *filename.line-number*.
`trace in` *class-name.method-name*	Trace calls to and returns from *class-name. method-name*.
`trace in` *class-name.method-name*`([`*parameters*`]).`	Trace calls to and returns from *class-name.method-name*([*parameters*]).

`trace inmethod` *class-name.* *method-name*	Trace when any method named *class-name.method-name* is called.
`trace inmethod` *class-name.* *method-* *name*`[(`*parameters*`)]`	Trace when any method named *class-name.method-name* [(*parameters*)] is called.

where:

class_name is the name of a Java class. You can use either of the following:

- The package path using a period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`
- The full path name preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers. For example, `#test1/extra/T1$Inner`. Enclose *class-name* in quotation marks if you use the `$` qualifier.

filename is the name of a file.

line-number is the number of a source code line.

method-name is the name of a Java method.

parameters are the method's parameters

The following modifiers are valid in Java mode.

`-if` *condition-expression*	The specified event occurs and the trace is printed only when *condition-expression* evaluates to true.
`-count` *number*	Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, the trace is printed and the counter is reset to 0.
`-count infinity`	Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.
`-temp`	Create a temporary breakpoint that is deleted when the event occurs and the trace is printed. If `-temp` is used with `-count`, the breakpoint is deleted only when the counter is reset to 0.
`-disable`	Create the breakpoint in a disabled state.

For a list and the syntax of all events see "Setting Event Specifications" on page 274.

`tracei` Command

The `tracei` command shows machine instructions, function calls, or variable changes. It is valid only in native mode.

`tracei` is really a shorthand for `trace` *event-specification* `-instr` where the `-instr` modifier causes tracing to happen at instruction granularity instead of source-line granularity. When the event occurs, the printed information is in disassembly format instead of source-line format.

Syntax

`tracei step`	Trace each machine instruction.
`tracei next -in` *function*	Trace each instruction while in the specified function.
`tracei at` *address*	Trace the instruction at *address*.
`tracei in` *function*	Trace calls to and returns from the specified function.
`tracei inmember` *function*	Trace calls to any member function named *function.*
`tracei infunction` *function*	Trace when any function named *function* is called.
`tracei inclass` *class*	Trace calls to any member function of *class.*
`tracei change` *variable*	Trace changes to the variable.

where:

address is any expression resulting in or usable as an address.

filename is the name of the file to which you want trace output sent.

function is the name of a function.

line is the number of a source code line.

class is the name of a class.

variable is the name of a variable.

See "trace Command" on page 400 for more information.

uncheck Command

The uncheck command disables checking of memory access, leaks, or usage. It is valid only in native mode.

Syntax

uncheck	Print the current status of checking.
uncheck -access	Disable access checking.
uncheck -leaks	Disable leak checking.
uncheck -memuse	Disable memory use checking (leak checking is disabled as well).
uncheck -all	Equivalent to uncheck -access; uncheck -memuse.
uncheck [*functions*] [*files*] [*load-objects*]	Equivalent to suppress all in *functions files load-objects.*

where:

functions is one or more function names.

files is one or more file names.

load-objects is one or more load object names

See "check Command" on page 313 for information about enabling checking.

See "suppress Command" on page 393 for information about suppressing errors.

See "Capabilities of Runtime Checking" on page 135 for an introduction to runtime checking.

`undisplay` Command

The `undisplay` command undoes `display` commands.

Native Mode Syntax

`undisplay {`*expression*`, ... |` *n* `...}`	Undo a `display` *expression* command or all the `display` commands numbered *n*, ... If *n* is set to zero (0), then undo all display commands.

where:

expression is a valid expression.

Java Mode Syntax

`undisplay` *expression*`, ... |` *identifier*`, ...`	Undo a `display` *expression*`, ... or display` *identifier*`, ...` command.
`undisplay` *n*`, ...`	Undo the `display` commands numbered *n*, ...
`undisplay 0` do all `display` commands.	Undo all `display` commands.

where:

expression is a valid Java expression.

field-name is the name of a field in the class.

identifier is a local variable or parameter, including `this`, the current class instance variable (*object-name*.*field-name*), or a class (static) variable (*class-name*.*field-name*).

`unhide` Command

The `unhide` command undoes `hide` commands. It is valid only in native mode.

Syntax

`unhide {`*regular-expression* `|` *number*`}`	Delete stack frame filter *regular-expression* or delete stack frame filter number *number* If *number* is set to zero (0), delete all stack frame filters.

where:

regular-expression is a regular expression.

number is the number of a stack frame filter.

The `hide` command lists the filters with numbers.

`unintercept` Command

The `unintercept` command undoes `intercept` commands (C++ only). It is valid only in native mode.

Syntax

`unintercept` *intercepted-typename* `[,` *intercepted-typename* `... ]`	Delete throws of type *intercepted-typename* from the `intercept` list.
`unintercept -a [ll]`	Delete all throws of all types from `intercept` list.
`unintercept -x` *excluded-typename* `[,` *excluded-typename* `... ]`	Delete *excluded-typename* from `excluded` list.
`unintercept -x -a[ll]`	Delete all throws of all types from the `excluded` list.
`unintercept`	List intercepted types.

where:

included-typename and *excluded-typename* are exception type specifications such as `List <int>` or `unsigned short.`

`unsuppress` Command

The `unsuppress` command undoes `suppress` commands. It is valid only in native mode.

Syntax

`unsuppress`	History of `suppress` and `unsuppress` commands (not those specifying the `-d` and `-reset` options).
`unsuppress -d`	List of errors being unsuppressed in functions that are not compiled for debugging. This list is per load object. Any other errors can be suppressed only by using the suppress command with the `-d` option.
`unsuppress -d` *errors*	Modify the default suppressions for all load objects by further unsuppressing *errors*.
`unsuppress -d` *errors* `in` *load-objects*	Modify the default suppressions in the *load-objects* by further unsuppressing *errors*.
`unsuppress -last`	At error location unsuppress present error.
`unsuppress -reset`	Set the default suppression mask to the original value (startup time).
`unsuppress` *errors*	Unsuppress *errors* everywhere.
`unsuppress` *errors* `in [`*functions*`] [`*filename ...*`] [`*load-objects*`]`	Suppress *errors* in a list of functions, a list of files, and a list of load objects.
`unsuppress` *errors* `at` *line*	Unsuppress *errors* at *line*.

`unsuppress` *errors* `at "`*filenames*`"` *line*	Unsuppress *errors* at *line* in *filenames*.
`unsuppress` *errors* `addr` *address*	Unsuppress *errors* at location *address*.

where:

errors is one or more error names.

functions is one or more function names.

filenames is one or more file names.

line is a line number.

load-objects is one or more load object names

`unwatch` Command

The `unwatch` command undoes a `watch` command. It is valid only in native mode.

Syntax

`unwatch` `{`*expression* `|` *n*`}`	Undo a `watch` *expression* command or the `watch` commands numbered *n* If *n* is set to zero (0), then undo all `watch` commands.

where:

expression is a valid expression.

`up` Command

The `up` command moves up the call stack toward `main`. It has the same syntax and functionality in native mode and in Java mode.

Syntax

`up [-h [`*number*`]]`	Move up the call stack one level. If *number* is specified, move up the call stack *number* levels. If `-h` is specified, move up the call stack, but do not skip hidden frames.

where:

number is a number of call stack levels.

`use` Command

The `use` command lists or changes the directory search path. It is valid only in native mode.

This command is an anachronism and usage of this command is mapped to the following `pathmap` commands:

`use` is equivalent to `pathmap -s`

`use` *directory* is equivalent to `pathmap` *directory*.

`watch` Command

The `watch` command evaluates and prints expressions at every stopping point in the scope current at that stop point. Because the expression is not parsed at entry time, the correctness of the expression cannot be immediately verified. The `watch` command is valid only in native mode.

Syntax

`watch`	Print the list of expressions being displayed.
`watch [-r|+r|-d|+d|-S|+S|-p|`	Watch the value of expression *expression* at every stopping point. See "`print` Command" on page 370 for the meaning of these flags.

+p|-L|-f*format*|-F*format*|-m|+m|--] *expression*

where:

expression is a valid expression.

format is the output format you want used to print the expression. For information about valid formats, see "print Command" on page 370.

whatis Command

In native mode, the whatis command prints the type of expression or declaration of type, or the definition of a macro. It also prints OpenMP data-sharing attribute information when applicable.

In Java mode, the whatis command prints the declaration of an identifier. If the identifier is a class, it prints method information for the class, including all inherited methods.

Native Mode Syntax

whatis [-n] [-r] [-m] [+m] *name*	Print the declaration of the non-type *name*, or the definition if *name* is a macro.
whatis -t [-a] [-r] [-u] *type*	Print the declaration of the type *type*.
whatis -e [-r] [-u] [-d] *expression*	Print the type of the expression *expression*.

where:

name is the name of a non-type or macro.

type is the name of a type.

expression is a valid expression.

macro is the name of a macro.

`-a` prints only data members for a specified class.

`-d` shows dynamic type instead of static type.

`-e` displays the type of an expression.

`-n` displays the declaration of a non-type. It is not necessary to specify `-n`; this is the default if you type the `whatis` command with no options.

`-r` prints information about base classes and types.

`-t` displays the declaration of a type.

`-u` displays the root definition of a type.

`-m` forces macro expansion even if the `dbxenv` variable `macro_expand` is set to `off`.

`+m` defeats macro lookup so that any symbols that might have been shadowed by macros are found instead.

The `whatis` command, when run on a C++ class or structure, provides you with a list of all the defined member functions, the static data members, the class friends, and the data members that are defined explicitly within that class. Undefined member functions are not listed.

Specifying the `-r` (recursive) option adds information from the inherited classes.

The`-d` flag, when used with the `-e` flag, uses the dynamic type of the expression.

For C++, template-related identifiers are displayed as follows:

- All template definitions are listed with `whatis -t`.
- Function template instantiations are listed with `whatis`.
- Class template instantiations are listed with `whatis -t`.

Java Mode Syntax

`whatis` *identifier*	Print the declaration of *identifier*.

where:

identifier is a class, a method in the current class, a local variable in the current frame, or a field in the current class.

when Command

The when command executes commands when a specified event occurs.

If dbx is in Java mode and you want to set a when breakpoint in native code, switch to Native mode using the joff command or prefix the when command with native.

If dbx is in JNI mode and you want to set a when breakpoint in Java code, prefix the when command with java.

Syntax

The when command has the following general syntax:

when *event-specification* [*modifier*]{*command*; ... }

When the specified event occurs, the commands are executed. The following commands are forbidden in the when command:

- attach
- debug
- next
- replay
- rerun
- restore
- run
- save
- step

A cont command with no options is ignored.

Native Mode Syntax

The following specific syntaxes are valid in native mode:

when at *line-number* { *command;* }	Execute *command* when *line-number* is reached.

`when in` *procedure* `{` *command;* `}`	Execute *command* when *procedure* is called.

where:

line-number is the number of a source code line.

command is the name of a command.

procedure is the name of a procedure.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

`when at` *line-number*	Execute command when source *line-number* is reached.
`when at` *filename.line-number*	Execute command when *filename*.*line-number* is reached.
`when in` *class-name.method-name*	Execute command when *class-name*.*method-name* is called.
`when in` *class-name.method-name*`([`*parameters*`])`	Execute command when *class-name*.*method-name*([*parameters*]) is called.

class-name is the name of a Java class. You can use either of the following:

- The package path using a period (`.`) as a qualifier; for example, `test1.extra.T1.Inner`
- The full path name preceded by a pound sign (`#`) and using slash (`/`) and dollar sign (`$`) as qualifiers. For example, `#test1/extra/T1$Inner`. Enclose *class-name* in quotation marks if you use the `$` qualifier.

filename is the name of a file.

line-number is the number of a source code line.

method-name is the name of a Java method.

parameters are the method's parameters.

For a list and the syntax of all events, see "Setting Event Specifications" on page 274.

See "wheni Command" on page 415 for information about executing commands on a specified low-level event.

wheni Command

The wheni command executes commands when a specified low-level event occurs. It is valid only in native mode.

Syntax

wheni *event-specification* [*modifier*]{*command*... ; }

When the specified event occurs, the commands are executed.

The following specific syntax is valid:

wheni at *address* { *command;* }	Execute *command* when *address* is reached.

where:

address is any expression resulting in or usable as an address.

command is the name of a command.

For a list and the syntax of all events see "Setting Event Specifications" on page 274.

where Command

The where command prints the call stack. For OpenMP slave threads, the command also prints the master thread's stack trace if the relevant frames are still active.

Native Mode Syntax

where	Print a procedure traceback.

`where` *number*	Print the *number* top frames in the traceback.
`where -f` *number*	Start traceback from frame *number*.
`where -fp` *address-expression*	Print traceback as if `fp` register had *address-expression* value.
`where -h`	Include hidden frames.
`where -l`	Include library name with function name.
`where -q`	Quick traceback (only function names).
`where -v`	Verbose traceback, which includes the function arguments and line information.

where:

address-expression is any expression resulting in or usable as an address.

number is a number of call stack frames.

Any of these options can be combined with a thread or LWP ID to obtain the traceback for the specified entity.

The `-fp` option is useful when the `fp` (frame pointer) register is corrupted, in which event `dbx` cannot reconstruct call stack properly. This option provides a shortcut for testing a value for being the correct `fp` register value. Once you have identified that the correct value has been identified, you can set it with an `assign` command or `lwp` command.

Java Mode Syntax

`where [`*thread-ID*`]`	Print a method traceback.
`where -f [`*thread-ID*`]` *number*	Print the *number* top frames in the traceback. If f is specified, start traceback from frame *number*.
`where -q [`*thread-ID*`]`	Quick trace back (only method names).
`where -v [`*thread-ID*`]`	Verbose traceback, which includes the method arguments and line information.

where:

number is a number of call stack frames.

thread-ID is a dbx-style thread ID or the Java thread name specified for the thread.

whereami Command

The whereami command displays the current source line. It is valid only in native mode.

Syntax

whereami	Display the source line corresponding to the current location (top of the stack), and the source line corresponding to the current frame, if different.
whereami -instr	Same as previous, except that the current disassembled instruction is printed instead of the source line.

whereis Command

The whereis command prints all uses of a specified name, or symbolic name of an address. It is valid only in native mode.

The whereis -a command can print the location of an address-expression if the address is from a heap or stack. Note that dbx cannot print the location if the process is not alive or if dbx is working with a core file.

Syntax

whereis *name*	Print all declarations of *name*.
whereis -a *address-expression*	Print location of an *address-expression*.

where:

name is the name of a loadable object that is in scope, for example, a variable, function, class template, or function template.

address is any expression resulting in or usable as an address.

`which` Command

The `which` command prints the full qualification of a specified name. It is valid only in native mode.

Syntax

`which [-n] [-m] [+m]` *name*	Print full qualification of *name*.
`which -t` *type*	Print full qualification of *type*.

where:

name is the name of loadable object that is in scope, for example, a variable, function, class template, or function template.

type is the name of a type.

`-n` displays the full qualification of a non-type. It is not necessary to specify `-n`; this is the default if you type the `which` command with no options.

`-t` displays the full qualification of a type.

`-m` forces macro lookup even if the `dbxenv` variable `macro_expand` is set to `off`.

`+m` defeats macro lookup so that any symbols that might have been shadowed by macros are found instead.

`whocatches` Command

The `whocatches` command tells where a C++ exception would be caught. It is valid only in native mode.

Syntax

`whocatches` *type*	Tell where (if at all) an exception of type *type* would be caught if thrown at the current point of execution. Assume the next statement to be executed is a `throw` *x* where *x* is of type *type*, and display the line number, function name, and frame number of the `catch` clause that would catch it. Will return `"`*type* `is unhandled"` if the catch point is in the same function that is doing the throw.

where:

type is the type of an exception.

Index

Numbers and Symbols

:: (double-colon) C++ operator, 73

A

access checking, 140
access event, 277
address
  current, 70
  display format, 243
  examining contents at, 241
  examples, 242
  examples of using, 244
adi assign command
  native mode syntax, 307
adi examine command
  native mode syntax, 308
adjusting default dbx settings, 59
alias command, 47
AMD64 registers, 254
ancillary objects, 48
array_bounds_check dbxenv variable, 61
arrays
  bounds, exceeding, 213
  evaluating, 123, 123
  Fortran, 215
  Fortran allocatable, 216
  slicing, 124, 127
    syntax for C and C++, 124
    syntax for Fortran, 125
  striding, 124, 127
assembly language debugging, 241
assign command
  syntax, 308
  using to assign a value to a variable, 123, 268
assigning a value to a variable, 123, 268
at event, 275
attach command, 72, 88, 309
attach event, 286
attached process, using runtime checking on, 156
attaching
  dbx to a new process while debugging an existing process, 89
  dbx to a running child process, 179
  dbx to a running Java process, 229
  dbx to a running process, 44, 88
    when dbx is not already running, 89

B

backquote operator, 73
bcheck command, 160
  examples, 160
  syntax, 160
bind command, 260
block local operator, 74
breakpoints
  clearing, 110
  defined, 32, 97
  deleting, using handler ID, 110
  disabling, 111
  enabling, 111
  enabling after event occurs, 297
  event efficiency, 111
  event specifications, 274
  filters, 105

using return value of a function call, 105
In Function, 99
In Object, 101
listing, 110, 110
multiple, setting in nonmember functions, 101
On Value Change, 103
overview, 97
setting
at a line, 32, 98
at a member function of a template class or at a template function, 207
at all instances of a function template, 207
at an address, 248
at class template instantiations, 203, 206
at function template instantiations, 203, 206
filters on, 105
in a function, 32, 99
in all member functions of a class, 100
in an explicitly loaded library, 265
in dynamically loaded libraries, 109
in member functions of different classes, 100
in native (JNI) code, 231
in objects, 101
in shared libraries, 264
machine level, 248
multiple breaks in C++ code, 100
on Java methods, 231
with filters that contain function calls, 107
`stop` type, 97
determining when to set, 67
`trace` type, 98
`when` type, 97
setting at a line, 109
`bsearch` command, 310

C

`-count` event specification modifier, 290
C
debugging application that embeds a Java application, 229
source files, specifying the location of, 230
C++
ambiguous or overloaded functions, 68
backquote operator, 73
class
declarations, looking up, 78
definition, looking up, 79
displaying all the data members directly defined by, 121
displaying all the data members inherited from, 121
printing the declaration of, 79
seeing inherited members, 80
viewing, 78
compiling with the -g option, 47
compiling with the -g0 option, 47
debugging application that embeds a Java application, 229
double-colon scope resolution operator, 73
exception handling, 198
function template instantiations, listing, 78
inherited members, 80
mangled names, 75
object pointer types, 120
printing, 120
setting multiple breakpoints, 100, 100
source files, specifying the location of, 230
template debugging, 202
template definitions
displaying, 78
tracing member functions, 108
unnamed arguments, 121
using dbx with, 197
`call` command
safety, 93
syntax, 310
using to call a function, 92
using to call a function explicitly, 93
using to call a procedure, 92, 269
using to explicitly call a function instantiation or member function of a class template, 207
call safety, 93
call stack, 113
deleting
all frame filters, 116

frames, 116
finding your place on, 113
frame, defined, 113
hiding frames, 116
looking at, 34
moving
down, 114
to a specific frame in, 115
up, 114
popping, 115, 268
removing the stopped-in function from, 115
walking, 69, 114
calling
function, 92, 93
function instantiation or member function of a class template, 207
member template functions, 203
procedure, 269
`cancel` command, 312
catch blocks, 198
`catch` command, 193, 194, 312
catch signal list, 193
catching exceptions of a specific type, 199
`change` event, 278
changing
default signal lists, 193
`check` command, 36, 137, 137, 313
`-access` option, 313
`-all` option, 315
combining leaks, 147
`-leaks` option, 314
`-memuse` option, 315
checkpoints, saving a series of debugging runs as, 57
child process
attaching `dbx` to, 179
debugging, 179
interacting with events, 180
using runtime checking on, 152
choosing among multiple occurrences of a symbol, 69
class template instantiations, printing a list of, 203, 205
classes
displaying all the data members directly defined by, 121
displaying all the data members inherited from, 121
looking up declarations of, 78
looking up definitions of, 79
printing the declarations of, 79
seeing inherited members, 80
viewing, 78
`CLASSPATHX` `dbxenv` variable, 61, 226
`clear` command, 316
clearing breakpoints, 110
`collector archive` command, 318
`collector` command, 316
`collector dbxsample` command, 318
`collector disable` command, 318
`collector enable` command, 319
`collector heaptrace` command, 319
`collector hwprofile` command, 319
`collector limit` command, 320
`collector pause` command, 321
`collector profile` command, 321
`collector resume` command, 321
`collector sample` command, 321
`collector show` command, 322
`collector status` command, 323
`collector store` command, 323
`collector synctrace` command, 324
`collector tha` command, 324
`collector version` command, 324
commands, 307
`dbxenv`, 60
`debug`
using to attach to a child process, 179
handling exceptions, 198
`kill`, 145
`print`
using to dereference a pointer, 122
process control, 87
setting startup properties, 45
`stop`

using to set breakpoint at all member functions of a C++ template class, 206
that alter the state of your program, 268
thread, 173
when, 272
compiling
code for debugging, 27
optimized code, 47
with the -g option, 47
with the -g0 option, 47
cond event, 279
cont command
continuing execution of your program with, 91, 138
syntax, 325
using to continue execution of your program from a different line, 91, 270
using to resume execution of a multithreaded program, 174
continuing execution of a program, 91
at a specified line, 91, 270
core file
core file truncation, 41
debugging, 31, 40
debugging mismatched, 42
examining, 31
using debug command to debug a core file, 41
core_lo_pathmap dbxenv variable, 61
count
using, 244
creating
a .dbxrc file, 60
event handlers, 272
creating a separate debug file, 48
current address, 70
current procedure and file, 209
customizing dbx, 59

D

-disable event specification modifier, 290
.dbxrc file, 59
creating, 60
sample, 60
use at dbx startup, 45, 59
dalias command, 325
data change event specifications, 277
data member, printing, 79
dbx
attaching to a process, 88
customizing, 59
detaching a process from, 54
detaching from a process, 89
quitting, 36, 54
starting, 28, 39
startup options, 328
with core file name, 40
with process ID only, 45
dbx command, 39, 44, 326
dbx commands
at the machine-instruction level, 241
creating your own, 47
differences between Korn shell and, 259
Java expression evaluation in, 236
process control, 87
setting startup properties, 45
static and dynamic information used by when debugging Java code, 237
that alter the state of your program, 268
using in Java mode, 236
valid only in Java mode, 240
with different syntax in Java mode, 239
with identical syntax and functionality in Java mode and native mode, 238
dbx dbxenv variables
output_pretty_print_fallback, 63
output_pretty_print_mode, 63
dbx modes for debugging Java code, 234
switching from Java or JNI to native mode, 235
switching modes when you interrupt execution, 236
dbx online help, 37
dbxenv command, 46, 60, 328
dbxenv variable, 61
dbxenv variables, 60, 61

descriptions of, 61
follow_fork_mode, 180
for Java debugging, 226
Korn shell, and, 66
setting, 60
setting with the dbxenv command, 60
dbxrc file, use at dbx startup, 45, 59
dbxtool, 27, 39, 39
debug command, 72
syntax, 329
using to attach dbx to a running process, 88
using to attach to a child process, 179
using to debug a core file, 41
debug_file_directory dbxenv variable, 61
debugging
assembly language, 241
child processes, 179
code compiled without -g option, 53
core file, 31, 40
creating a separate debug file, 48
machine-instruction level, 241, 246
mismatched core file, 42
multithreaded programs, 171
optimized code, 51
replaying a saved debugging run, 58
saving a run, 55
using a separate debug file, 47
ancillary objects, 48
debugging application that embeds a Java application
C, 229
C++, 229
debugging information
reading in, 84, 85
debugging run
saved
replaying, 58
restoring, 57
saving, 55
declarations, looking up (displaying), 78
delete command, 332
deleting
all call stack frame filters, 116
call stack frames, 116
specific breakpoints using handler IDs, 110
dereferencing a pointer, 122
detach command, 54, 332
detach event, 287
detaching
a process from dbx, 54, 89
a process from dbx and leaving it in a stopped state, 90
determining
at symbol dbx uses, 76
cause of floating-point exception (FPE), 195
granularity of source line stepping, 91
location of floating-point exception (FPE), 195
number of instructions executed, 297
number of lines executed, 297
where your program is crashing, 31
dis command, 70, 244, 333
disabling
runtime checking, 137
disassembler_version dbxenv variable, 61
display command, 122, 122, 334
displaying
all the data members directly defined by a class, 121
all the data members inherited from a base class, 121
declarations, 78
definitions of templates and instances, 203, 205
inherited members, 80
source code for function template instantiations, 203
stack trace, 116
symbols, occurrences of, 75
template definitions, 78
type of an exception, 199
unnamed function argument, 122
variable type, 79
variables and expressions, 122
dlclose event
valid variables, 295
dlopen event, 279
valid variables, 295
down command, 72, 114, 336

dump command, 336
  using on OpenMP code, 187
dynamic linker, 263

E

edit command, 337
enabling
  a breakpoint after an event occurs, 297
  memory access checking, 36, 137, 137
  memory leak checking, 36, 137, 137
  memory use checking, 36, 137, 137
error suppression, 149, 150
  default, 151
  examples, 151
  scope, 150
  types, 150
establishing a new mapping from directory to directory, 46, 86
evaluating
  arrays, 123, 123
  function instantiation or member function of a class template, 208
  unnamed function argument, 122
event counters, 273, 273
event handler
  hiding, 291
  retaining across debugging sessions, 292
event handlers
  creating, 272
  manipulating, 272
  setting, examples, 296
event management, 94, 271
event specification modifiers, descriptions of, 289
event specifications, 271, 272, 274
  for breakpoint events, 274
  for data change events, 277
  for execution progress events, 283
  for OpenMP code, 188
    for synchronization, 188
    other, 189
  for other types of events, 286
  for synchronization, 188
  for system events, 279
  for thread tracking, 285
  keywords, defined, 274
  machine-instruction level, 248
  modifiers, 289
  setting, 274
  using predefined variables, 292
event-specific variables, 294
event_safety dbxenv variable, 61
events
  ambiguity, 292
  child process interaction with, 180
  parsing, 292
examine command, 70, 242, 337
examining the contents of memory, 241
exception command, 199, 339
exception handling, 198
  examples, 201
exceptions
  floating point, determining cause of, 195
  floating point, determining location of, 195
  in Fortran programs, locating, 214
  of a specific type, catching, 199
  removing types from intercept list, 200
  reporting where type would be caught, 200
  type of, displaying, 199
exec function, following, 180
executables
  separate debugging information, 81
execution progress event specifications, 283
exists command, 339
exit event, 283
  valid variables, 295
experiments
  limiting the size of, 320
expressions
  displaying, 122
  Fortran
    complex, 218
    interval, 218
  monitoring changes, 122
  monitoring the value of, 122
  printing the value of, 120, 269

stop the display of, 123

F

fault event, 280
fflush(stdout), after dbx calls, 93
field type
displaying, 79
printing, 79
file command, 68, 70, 72, 340
files
finding, 46, 86
location of, 86
navigating to, 68
qualifying name, 72
files command, 340
filter_max_length dbxenv variable, 61
finding
object files, 46
place on the call stack, 113
source files, 46, 86
fix and continue
using with runtime checking, 157
fix command, 270, 341
fix_verbose dbxenv variable, 61
fixed command, 342
fixing
program, 270
floating-point exception (FPE)
catching, 298
determining cause of, 195
determining location of, 195
follow_fork_inherit dbxenv variable, 61, 180
follow_fork_mode dbxenv variable, 61, 152, 180
follow_fork_mode_inner dbxenv variable, 61
fork function, following, 180
Fortran
allocatable arrays, 216
allocatable scalar type, 223
array slicing syntax for, 125
case sensitivity, 210
complex expressions, 218
derived types, 220
interval expressions, 218
intrinsic functions, 217
logical operators, 219
Object Oriented, 223
sample dbx session, 210
structures, 220
fortran_modules command, 342
FPE signal, trapping, 193
frame command, 72, 115, 342
frame, defined, 113
func command, 68, 70, 72, 343
funcs command, 344
function argument, unnamed, 122, 122
function template instantiations
displaying the source code for, 203
printing a list of, 203, 205
printing the values of, 203
functions
ambiguous or overloaded, 68
calling, 92, 93
inlined, in optimized code, 52
instantiation
calling, 207
evaluating, 208
printing source listing for, 208
intrinsic, Fortran, 217
looking up definitions of, 78
member of a class template, calling, 207
member of class template, evaluating, 208
navigating to, 68
obtaining names assigned by the compiler, 121
qualifying name, 72
setting breakpoints in, 99
setting breakpoints in C++ code, 100

G

-g compiler option, 47
gdb command, 345

H

-hidden event specification modifier, 291
handler command, 273, 346
handler id, defined, 272
handlers, 271
  creating, 272, 272
  enabling while within a function, 296
hide command, 116, 346
hiding call stack frames, 116

I

-if event specification modifier, 289
-in event specification modifier, 290
-instr event specification modifier, 291
ignore command, 192, 193, 347
ignore signal list, 193
import command, 347
in event, 275
In Function breakpoint, 99
In Object breakpoint, 101
inclass event, 277
infile event, 275
infunction event, 276
inherited members
  displaying, 80
  seeing, 80
inmember event, 276
inmethod event, 276, 276
inobject event, 277
input_case_sensitive dbxenv variable, 61
input_case_sensitive environment variable, 210, 210
instances, displaying the definitions of, 203, 205
Intel registers, 252
intercept command, 199, 348
invocation options, 328

J

JAR file, debugging, 227
Java applications
  attaching dbx to, 228
  specifying custom wrappers for, 233
  starting to debug, 226
  that require 64-bit libraries, 229
  types you can debug with dbx, 226
  with wrappers, debugging, 228
Java class file, debugging, 227
Java code
  capabilities of dbx with, 225
  dbx modes for debugging, 234
  limitations of dbx with, 225
  static and dynamic information used by dbx commands, 237
  using dbx with, 225
java command, 349
Java debugging, environment variables for, 226
Java mode, 235
  dbx commands valid only in, 240
  different syntax than dbx commands, 239
  identical syntax and functionality for dbx commands, 238
  switching from Java or JNI to native mode, 235
  using dbx commands in, 236
Java source files, specifying the location of, 230
JAVASRCPATH dbxenv variable, 61, 226
jclasses command, 349
jdbx_mode dbxenv variable, 62, 226
joff command, 350
jon command, 350
jpkgs command, 350
JVM software
  customizing startup of, 231
  passing run arguments to, 230, 232
  specifying 64-bit, 234
  specifying a path name for, 232
jvm_invocation dbxenv variable, 62, 226

K

key bindings for editors, displaying or modifying, 260
kill command, 54, 145, 350
killing

program, 54
Korn shell
differences from dbx, 259
extensions, 260
features not implemented, 259
renamed commands, 260

L

-lwp event specification modifier, 291
language command, 351
language_mode dbxenv variable, 62
lastrites event, 287
LD_AUDIT, 156
LD_PRELOAD, 157
leak checking, 137
libraries
dynamically loaded, setting breakpoints in, 109
shared, compiling for dbx, 53
librtc.so, preloading, 157
librtld_db.so, 263
libthread_db.so, 171
limiting the experiment size, 320
line command, 70, 351
link map, 263
linker names, 75
list command, 70, 72
syntax, 352
using to print a source listing for a file or function, 69
using to print the source listing for a function instantiation, 208
listi command, 245, 354
listing
all program modules that contain debugging information, 85
breakpoints, 110, 110
C++ function template instantiations, 78
debugging information for modules, 84
names of all program modules, 85
names of modules containing debugging information that have already been read into dbx, 85
signals currently being ignored, 193
signals currently being trapped, 193
traces, 110
load object, defined, 263
loading your program, 28
loadobject command, 354
-dumpelf flag, 355
-exclude flag, 356
-hide flag, 356
-list flag, 357
-load flag, 358
-unload flag, 358
-use flag, 359
looking up
definitions of classes, 79
definitions of functions, 78
definitions of members, 78
definitions of types, 79
definitions of variables, 78
this pointer, 79
lwp command, 359
lwp_exit event, 281
LWPs (lightweight processes), 171
information displayed for, 176
showing information about, 176
states, 172
lwps command, 176, 360

M

machine-instruction level
address, setting breakpoint at, 248
AMD64 registers, 254
debugging, 241
Intel registers, 252
printing the value of all the registers, 248
setting breakpoint at address, 248
single stepping, 246
SPARC registers, 251

tracing, 247
using dbx, 241
macro
compiler and compiler options, 303
definition method, 302, 303
limitations, 304
skimming, 304
tradeoffs in functionality, 303
definitions, 302
expansion, 301
skimming, 304
macro command, 301, 360
macro_expand dbxenv variable, 62, 302
macro_source dbxenv variable, 62, 302
manipulating event handlers, 272
member functions
printing, 79
setting multiple breakpoints in, 100
tracing, 108
member template functions, 203
members
declarations, looking up, 78
looking up declarations of, 78
looking up definitions of, 78
viewing, 78
memory
address display formats, 243
display modes, 241
examining contents at address, 241
states, 140
memory access
checking
enabling, 36
error report, 141
errors, 142, 165
memory access checking, 140
enabling, 137, 137
memory leak
checking, 145
enabling, 36
errors, 144, 169
fixing, 148
report, 145
memory leak checking, 143
enabling, 137, 137
memory use checking, 148
enabling, 36, 137, 137
mmapfile command, 361
module command, 84, 362
modules
all program, listing, 85
containing debugging information that have already been read into dbx, listing, 85
containing debugging information, listing, 85
current, printing the name of, 85
listing debugging information for, 84
modules command, 84, 85, 362
monitoring the value of an expression, 122
moving
down the call stack, 114
to a specific frame in the call stack, 115
up the call stack, 114
mt_resume_one dbxenv variable, 62
mt_scalable dbxenv variable, 62
mt_sync_tracking dbxenv variable, 62
multithreaded programs, debugging, 171

N

native command, 363
navigating
through functions by walking the call stack, 69
to a file, 68
to functions, 68
next command, 90, 363
next event, 283
nexti command, 246, 365

O

object files
finding, 46
loading, 82
separate debugging information, 81
object pointer types, 120

obtaining the function name assigned by the compiler, 121
omp_atomic event, 286
omp_barrier event, 285
omp_critical event, 285
omp_flush event, 286
omp_loop command, 366
omp_master event, 286
omp_ordered event, 285
omp_pr command, 366
omp_serialize command, 367
omp_single event, 286
omp_task event, 286
omp_taskwait event, 285
omp_team command, 367
omp_tr command, 367
online help, accessing, 37
OpenMP application programming interface, 181
OpenMP code
    dbx functionality available for, 182
    events for, other, 189
    events for, synchronization, 188
    execution sequence of, 189
    printing a description of the current loop, 186
    printing a description of the current parallel region, 183
    printing a description of the current task region, 184
    printing all the threads on the current tea, 186
    printing shared, private, and thread private variables in, 183
    serializing the execution of the next encountered parallel region, 186
    single-stepping in, 182
    transformation by compilers, 181
    using stack traces with, 186
    using the dump command on, 187
operators
    backquote, 73
    block local, 74
    C++ double colon scope resolution, 73
optimized code
    about parameters and variables, 52
    compiling, 47
    debugging, 51
    inlined functions, 52
output_auto_flush dbxenv variable, 62
output_base dbxenv variable, 62
output_class_prefix dbxenv variable, 62
output_data_member_only dbxenv variable, 62
output_dynamic_type dbxenv variable, 63, 199
output_dynamic_type environment variable, 121
output_inherited_members dbxenv variable, 63
output_list_size dbxenv variable, 63
output_log_file_name dbxenv variable, 63
output_max_object_size dbxenv variable, 63
output_max_string_length dbxenv variable, 63
output_no_literal dbxenv variable, 63
output_pretty_print dbxenv variable, 63
output_pretty_print environment variable, 129
output_pretty_print_mode environment variable, 129
output_short_file_name dbxenv variable, 63
overload_function dbxenv variable, 63
overload_operator dbxenv variable, 63

P

-perm event specification modifier, 292
pathmap command, 86
    skimming, 305
    syntax, 368
    using to map the compile-time directory to the debug-time directory, 46
pointers
    dereferencing, 122
    printing, 223
pop command
    syntax, 370
    using to change the current stack frame, 72
    using to pop frames from the call stack, 268
    using to remove frames from the call stack, 115
pop_auto_destruct dbxenv variable, 63
popping

the call stack, 115, 268
predefined variables for event specification, 292
preloading
librtc.so, 157
rtcaudit.so, 156
pretty-printing, 128
call-based, 129
failures, 131
function considerations, 130
filters, 131
invoking, 129
Python, 131, 133
API, 134
Python Docs, 134
print command
syntax, 370
syntax to slice a C or C++ array, 124
syntax to slice a Fortran array, 125
using to dereference a pointer, 122
using to evaluate a function instantiation or a member function of a class template, 208
using to evaluate a variable or expression, 120
using to print the value of an expression, 269
printing
all the threads on the current team, 186
arrays, 123
data member, 79
declaration of a type or C++ class, 79
description of the current loop, 186
description of the current parallel region, 183
description of the current task region, 184
field type, 79
list of all class and function template instantiations, 203, 205
list of all known threads, 174
list of occurrences of a symbol, 75
list of threads normally not printed (zombies), 174
member functions, 79
name of the current module, 85
pointer, 223
shared, private, and thread private variables in OpenMP code, 183
source listing, 69
source listing for the specified function instantiation, 208
tvalue of a variable or expression, 120
value of all the machine-level registers, 248
value of an expression, 269
values of function template instantiations, 203
variable type, 79
proc command, 374
proc_exclusive_attach dbxenv variable, 63
proc_gone event, 287
valid variables, 295
procedure linkage tables, 264
procedure, calling, 269
process
attached, using runtime checking on, 156
attaching dbx using process ID, 44
child
attaching dbx to, 179
using runtime checking on, 152
detaching from dbx, 54, 89
detaching from dbx and leaving in a stopped state, 90
running, attaching dbx to, 88, 89
stopping execution, 54
stopping with Ctrl+C, 94
process control commands, definition, 87
prog command, 374
prog_new event, 287
program
continuing execution of, 91
at a specified line, 270
fixing, 270
killing, 54, 54
multithreaded
debugging, 171
resuming execution of, 174
running, 87
under dbx, impacts of, 267
with runtime checking enabled, 138
single-stepping through, 90
status, checking, 298
stepping through, 90
stopping execution

if a conditional statement evaluates to true, 104
if the value of a specified variable has changed, 103
stripped, 53
python-docs
command, 134

Q

qualifying symbol names, 72
quit command, 375
quitting a dbx session, 54
quitting dbx, 36

R

-resumeone event specification modifier, 107, 289
reading a stack trace, 116
reading in
debugging information, 84, 85
registers
AMD64 architecture, 254
Intel architecture, 252
printing the value of, 248
SPARC architecture, 251
regs command, 248, 375
removing
exception types from intercept list, 200
stopped-in function from the call stack, 115
replay command, 55, 58, 376
replaying a saved debugging run, 58
reporting where an exception type would be caught, 200
rerun command, 377
resetting application files for replay, 298
restore command, 55, 57, 377
restoring a saved debugging run, 57
resuming
execution of a multithreaded program, 174
returns event, 283, 284
rprint
command, 377
rtc showmap command, 378
rtc skippatch command, 378
skipping instrumentation, 164
rtc_auto_continue dbxenv variable, 63, 161
rtc_auto_continue environment variable, 138
rtc_auto_suppress dbx variable, 150
rtc_auto_suppress dbxenv variable, 63
rtc_biu_at_exit dbxenv variable, 64
rtc_biu_at_exit dbxenv variables, 148
rtc_error_limit dbxenv variable, 64, 150
rtc_error_log_file_name dbxenv variable, 64, 161
rtc_error_log_file_name environment variable, 138
rtc_error_stack dbxenv variable, 64
rtc_inherit dbxenv variable, 64
rtc_mel_at_exit dbxenv variable, 64
rtcaudit.so, preloading, 156
rtld, 263
run command, 87, 379
run_autostart dbxenv variable, 64
run_io dbxenv variable, 64
run_pty dbxenv variable, 64
run_quick dbxenv variable, 64
run_savetty dbxenv variable, 64
run_setpgrp dbxenv variable, 64
runargs command, 380
running a program, 29, 87
in dbx without arguments, 30, 87
with runtime checking enabled, 138
runtime checking
access checking, 140
application programming interface, 159
attached process, 156
child process, 152
disabling, 137
error suppression, 149
errors, 165
fixing memory leaks, 148
limitations, 162
memory access
checking, 140
error report, 141

errors, 142, 165
memory leak
checking, 143, 145
error report, 145
errors, 144, 169
memory use checking, 148
possible leaks, 144
requirements, 136
suppressing errors, 149
default, 151
examples, 151
suppression of last error, 150
troubleshooting tips, 161
types of error suppression, 150
using fix and continue with, 157
using in batch mode, 160
directly from dbx, 161
when to use, 136

S

sample .dbxrc file, 60
save command, 55, 55, 381
saving
debugging run to a file, 55, 57
series of debugging runs as checkpoints, 57
scope, 70
changing the visiting, 71
current, 67, 70
lookup rules, relaxing, 77
visiting, 71
changing, 72
components of, 71
scope resolution operators, 72
scope resolution search path, 77
scope-look-aside dbxenv variable, 77
scope_global_enums dbxenv variable, 64
scope_look_aside dbxenv variable, 64
scopes command, 381
search command, 381
segmentation fault
finding line number of, 213
Fortran, causes, 213
generating, 213
separate debugging information
executables, 81
object files, 81
session, dbx
quitting, 54
starting, 39
session_log_file_name dbxenv variable, 65
setting
a trace, 108
breakpoints
at a member function of a template class or at a template function, 207
at all instances of a function template, 207
in all member functions a class, 100
in dynamically loaded libraries, 109
in member functions of different classes, 100
in native (JNI) code, 231
in objects, 101
on Java methods, 231
when breakpoint at a line, 109
with filters that contain function calls, 107
dbxenv variables with the dbxenv command, 60
filters on breakpoints, 105
multiple breakpoints in nonmember functions, 101
shared libraries
compiling for dbx, 53
setting breakpoints in, 264
shared objects
.init sections, 264
startup sequence, 264
show_static_members dbxenv variable, 65
showblock command, 137, 382
showleaks command
combining leaks, 147
default output of, 148
error limit for, 151
report resulting from, 144
syntax, 382
using to ask for a leaks report, 147
showmemuse command, 148, 383
sig event, 281
valid variables, 294

signals
  cancelling, 191
  catching, 192
  changing default lists, 193
  forwarding, 191
  FPE, trapping, 193
  handling automatically, 196
  ignoring, 193
  listing those currently being ignored, 193
  listing those currently being trapped, 193
  names that `dbx` accepts, 193
  sending in a program, 196
single stepping
  through a program, 90
single-stepping
  at the machine-instruction level, 246
skimming
  errors, 304
  improving by using `pathmap` command, 305
slicing
  arrays, 127
  C and C++ arrays, 124
  Fortran arrays, 125
`source` command, 383
source files
  finding, 46, 86
  specifying the location of
    C, 230
    C++, 230
    Java source files, 230
source listing, printing, 69
SPARC registers, 251
specifying a path for class files that use custom class loaders, 230
stack frame, defined, 113
stack trace, 214
  displaying, 116
  example, 117, 117
  Fortran, 214
  reading, 116
  using on OpenMP code, 186
`stack-find-source` environment variable, 72
`stack_find_source` `dbxenv` variable, 65
`stack_max_size` `dbxenv` variable, 65
`stack_verbose` `dbxenv` variable, 65
starting `dbx`, 28
starting `dbxtool`, 28
startup options, 328
`status` command, 384
`step` command, 90, 198, 384
`step` event, 284
`step to` command, 33, 90, 386
`step up` command, 90, 385
`step_abflow` `dbxenv` variable, 65
`step_events` `dbxenv` variable, 65
`step_events` environment variable, 111
`step_granularity` `dbxenv` variable, 65
`step_granularity` environment variable, 91
`stepi` command, 246, 386
stepping into a function, 91
stepping through a program, 33, 90
stop
  display of a particular variable or expression, 123
  display of all currently monitored variables, 123
`stop access` command, 102, 388
`stop at` command, 98, 388
`stop change` command, 103, 388
`stop` command, 207
  syntax, 387
  using to set breakpoint at all member functions of a C++ template class, 206
  using to set breakpoints at all instances of a function template, 207
  using to stop in all member functions of a C++ template class, 206
`stop cond` command, 104, 388
`stop` event, 287
`stop in` command, 99, 388
`stop inclass` command, 100, 388
`stop infile` command, 388
`stop inmember` command, 100, 388, 388
`stop inobject` command, 101, 388
`stopi` command, 248, 392
stopping
  in all member functions of a template class, 206

process execution, 54
process with Ctrl+C, 94
program execution
if a conditional statement evaluates to true, 104
if the value of a specified variable has changed, 103
striding across slices of arrays, 127
stripped programs, 53
suppress command
syntax, 393
using to limit reporting of runtime checking errors, 138
using to list errors being suppressed in files not compiled for debugging, 151
using to manage runtime checking errors, 152
using to suppress runtime checking errors, 149
suppress_startup_message dbxenv variable, 65
suppression of last error, 150
symbol names, qualifying scope, 72
symbol_info_compression dbxenv variable, 65
symbols
choosing among multiple occurrences of, 69
determining which dbx uses, 76
printing a list of occurrences, 75
sync command, 395
sync event, 288
syncrtld event, 288
syncs command, 396
sysin event, 282
valid variables, 295
sysout event, 283
valid variables, 295
system event specifications, 279

T

-temp event specification modifier, 290
-thread event specification modifier, 291
templates
class, 202
stopping in all member functions of, 206
displaying the definitions of, 203, 205
function, 202
instantiations, 202
printing a list of, 203, 205
looking up declarations of, 79
thr_create event, 175, 288
valid variables, 295
thr_exit event, 175, 288
thread command, 173, 396
thread creation, understanding, 175
threads
current, displaying, 174
information displayed for, 172
list, viewing, 174
other, switching viewing context to, 173
printing list of all known, 174
printing list of normally not printed (zombies), 174
resuming only the first in which a breakpoint was hit, 107
states, 172
switching to by thread ID, 174
threads command, 174, 398
throw event, 284
timer event, 289
trace command, 108, 400
trace output, directing to a file, 108
trace_speed dbxenv variable, 65
trace_speed environment variable, 108
tracei command, 247, 404
traces
controlling speed of, 108
implementing, 296
listing, 110, 110
setting, 108
traces at the machine-instruction level, 247
track_process_cwd dbxenv variables, 65
trip counters, 273
troubleshooting tips, runtime checking, 161
types
declarations, looking up, 78
derived, Fortran, 220
looking up declarations of, 78
looking up definitions of, 79
printing the declaration of, 79

viewing, 78

changing, 71, 72
components of, 71

U

uncheck command, 137, 405
undisplay command, 123, 123, 406
unhide command, 116, 406
unintercept command, 200, 407
unsuppress command, 149, 152, 408
unwatch command, 409
up command, 72, 114, 409
use command, 410

V

variable type, displaying, 79
variables
  assigning values to, 123, 268
  declarations, looking up, 78
  determining which dbx is evaluating, 119
  displaying functions and files in which defined, 119
  event specific, 294, 294
  examining, 35
  looking up declarations of, 78
  looking up definitions of, 78
  monitoring changes, 122
  outside of scope, 120
  printing the value of, 120
  qualifying names, 72
  stop the display of, 123
  viewing, 78
vdl_mode dbxenv variables, 65
verifying which variable dbx is evaluating, 119
viewing
  classes, 78
  context of another thread, 173
  members, 78
  threads list, 174
  types, 78
  variables, 78
visiting scope, 71

W

walking the call stack, 69, 114
watch command, 122, 410
watch event
  valid variables, 295
whatis command, 78, 79, 301
  syntax, 411
  using to display the definitions of templates and instances, 205
  using to obtain the function name assigned by the compiler, 121
when breakpoint at a line, setting, 109
when command, 109, 269, 272, 413
wheni command, 415
where command, 114, 214, 415
whereami command, 417
whereis command, 76, 205, 417
  macro, 302
  verifying variables, 119
which command, 69, 76, 119, 418
  macro, 301
whocatches command, 200, 418

X

x command, 242, 337