Oracle® Solaris Studio 12.3: Debugging a Program With dbx
Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, brevetter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est concédé sous licence au Gouvernement des États-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l'utilise pour le compte du Gouvernement des États-Unis, la notice suivante s'applique:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007), Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de sécurité, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée de The Open Group.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation.
Contents

Preface ................................................................................................................................................... 23

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 ................................................................. 30
      Setting Breakpoints ................................................................. 31
      Stepping Through Your Program ............................................. 32
      Looking at the Call Stack .......................................................... 33
      Examining Variables ............................................................... 34
      Finding Memory Access Problems and Memory Leaks ............... 35
   Quitting dbx .................................................................................. 35
   Accessing dbx Online Help .......................................................... 36

2 Starting dbx ...................................................................................... 37
   Starting a Debugging Session .......................................................... 37
   Debugging a Core File .................................................................... 38
      Debugging a Core File in the Same Operating Environment ........ 39
      If Your Core File Is Truncated .................................................... 39
      Debugging a Mismatched Core File .......................................... 40
   Using the Process ID ..................................................................... 42
   The dbx Startup Sequence ............................................................ 43
   Setting Startup Properties ............................................................. 43
      Mapping the Compile-time Directory to the Debug-time Directory 43
      Setting dbx Environment Variables .......................................... 44
Contents

Creating Your Own dbx Commands ................................................................. 44
Compiling a Program for Debugging ............................................................... 44
  Compiling with the -g Option ................................................................. 44
  Using a Separate Debug File ................................................................. 45
Debugging Optimized Code ........................................................................ 46
  Parameters and Variables ................................................................. 46
  Inlined Functions ............................................................................... 47
  Code Compiled Without the -g Option ................................................. 47
  Shared Libraries Require the -g Option for Full dbx Support .................. 48
  Completely Stripped Programs .............................................................. 48
Quitting Debugging ................................................................................ 48
  Stopping a Process Execution ............................................................. 48
  Detaching a Process From dbx ............................................................ 48
  Killing a Program Without Terminating the Session ......................... 49
Saving and Restoring a Debugging Run ...................................................... 49
  Using the save Command ................................................................. 49
  Saving a Series of Debugging Runs as Checkpoints ......................... 51
  Restoring a Saved Run .................................................................... 51
  Saving and Restoring Using replay ..................................................... 52

3 Customizing dbx ...................................................................................... 53
  Using the dbx Initialization File ............................................................ 53
  Creating a .dbxrc File ........................................................................ 54
  Initialization File Sample .................................................................... 54
Setting dbx Environment Variables ............................................................ 54
The dbx Environment Variables and the Korn Shell ................................ 60

4 Viewing and Navigating To Code .............................................................. 61
  Navigating To Code ........................................................................... 61
  Navigating To a File ........................................................................... 62
  Navigating To Functions ................................................................... 62
  Printing a Source Listing .................................................................. 63
  Walking the Call Stack to Navigate To Code .................................... 63
  Types of Program Locations .............................................................. 63
  Program Scope ................................................................................. 64
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printing the Value of a Variable, Expression, or Identifier ................................................... 108</td>
</tr>
<tr>
<td>Printing C++ Pointers ................................................................. 108</td>
</tr>
<tr>
<td>Evaluating Unnamed Arguments in C++ Programs ......................................................... 109</td>
</tr>
<tr>
<td>Dereferencing Pointers ................................................................................................. 109</td>
</tr>
<tr>
<td>Monitoring Expressions ................. 110</td>
</tr>
<tr>
<td>Turning Off Display (Undisplaying) ........................................ 110</td>
</tr>
<tr>
<td>Assigning a Value to a Variable ........................................................................ 111</td>
</tr>
<tr>
<td>Evaluating Arrays .................................................................................. 111</td>
</tr>
<tr>
<td>Array Slicing ...................................................................................... 111</td>
</tr>
<tr>
<td>Using Slices ..................................................................................... 114</td>
</tr>
<tr>
<td>Using Strides .................................................................................... 114</td>
</tr>
<tr>
<td>Using Pretty-Printing ........................................................................ 115</td>
</tr>
<tr>
<td>9 Using Runtime Checking ........................................................................ 119</td>
</tr>
<tr>
<td>Capabilities of Runtime Checking ................................................................. 119</td>
</tr>
<tr>
<td>When to Use Runtime Checking ........................................................................ 120</td>
</tr>
<tr>
<td>Runtime Checking Requirements ........................................................................ 120</td>
</tr>
<tr>
<td>Using Runtime Checking ........................................................................ 121</td>
</tr>
<tr>
<td>Turning On Memory Use and Memory Leak Checking ........................................... 121</td>
</tr>
<tr>
<td>Turning On Memory Access Checking ................................................................ 121</td>
</tr>
<tr>
<td>Turning On All Runtime Checking ....................................................................... 121</td>
</tr>
<tr>
<td>Turning Off Runtime Checking ........................................................................ 121</td>
</tr>
<tr>
<td>Running Your Program ............................................................................... 121</td>
</tr>
<tr>
<td>Using Access Checking ........................................................................ 124</td>
</tr>
<tr>
<td>Understanding the Memory Access Error Report .............................................. 125</td>
</tr>
<tr>
<td>Memory Access Errors ............................................................................... 125</td>
</tr>
<tr>
<td>Using Memory Leak Checking ........................................................................ 126</td>
</tr>
<tr>
<td>Detecting Memory Leak Errors ........................................................................ 127</td>
</tr>
<tr>
<td>Possible Leaks ..................................................................................... 127</td>
</tr>
<tr>
<td>Checking for Leaks .................................................................................. 128</td>
</tr>
<tr>
<td>Understanding the Memory Leak Report ......................................................... 128</td>
</tr>
<tr>
<td>Fixing Memory Leaks ................................................................................ 130</td>
</tr>
<tr>
<td>Using Memory Use Checking .......................................................................... 131</td>
</tr>
<tr>
<td>Suppressing Errors .................................................................................... 132</td>
</tr>
<tr>
<td>Types of Suppression ................................................................................. 132</td>
</tr>
</tbody>
</table>
## Contents

- **15 Debugging C++ With dbx** ................................................................. 181
  - Using dbx With C++ .............................................................................. 181
  - Exception Handling in dbx ................................................................. 182
    - Commands for Handling Exceptions ................................................. 182
    - Examples of Exception Handling .................................................... 184
  - Debugging With C++ Templates .......................................................... 185
    - Template Example ............................................................................ 186
    - Commands for C++ Templates .......................................................... 187

- **16 Debugging Fortran Using dbx** ...................................................... 191
  - Debugging Fortran ............................................................................... 191
    - Current Procedure and File ............................................................. 191
    - Uppercase Letters ............................................................................. 192
    - Sample dbx Session .......................................................................... 192
  - Debugging Segmentation Faults ............................................................ 195
    - Using dbx to Locate Problems ......................................................... 195
  - Locating Exceptions ............................................................................ 196
  - Tracing Calls ...................................................................................... 196
  - Working With Arrays .......................................................................... 197
    - Fortran 95 Allocatable Arrays ......................................................... 198
  - Showing Intrinsic Functions ................................................................. 199
  - Showing Complex Expressions ........................................................... 199
  - Showing Interval Expressions ............................................................ 200
  - Showing Logical Operators ............................................................... 201
  - Viewing Fortran 95 Derived Types ..................................................... 201
  - Pointer to Fortran 95 Derived Type .................................................... 202
  - Object Oriented Fortran ....................................................................... 204
    - Allocatable Scalar Type ................................................................. 204

- **17 Debugging a Java Application With dbx** ....................................... 205
  - Using dbx With Java Code ................................................................. 205
  - Capabilities of dbx With Java Code .................................................... 205
  - Limitations of dbx With Java Code ..................................................... 206
  - Environment Variables for Java Debugging ....................................... 206
  - Starting to Debug a Java Application ............................................... 207
Contents

Setting a Breakpoint at an Address .......................................................................................... 227
Using the regs Command ........................................................................................................ 228
Platform-Specific Registers ................................................................................................... 230

19 Using dbx With the Korn Shell .......................................................................................... 237
ksh-88 Features Not Implemented ...................................................................................... 237
Extensions to ksh-88 ............................................................................................................ 238
Renamed Commands ............................................................................................................. 238
Rebinding of Editing Functions ............................................................................................ 238

20 Debugging Shared Libraries .............................................................................................. 241
Dynamic Linker ..................................................................................................................... 241
Link Map .............................................................................................................................. 241
Startup Sequence and .init Sections ................................................................................... 242
Procedure Linkage Tables .................................................................................................... 242
Fix and Continue .................................................................................................................. 242
Setting Breakpoints in Shared Libraries ............................................................................. 242
Setting a Breakpoint in an Explicitly Loaded Library ......................................................... 243

A Modifying a Program State ................................................................................................. 245
Impacts of Running a Program Under dbx ......................................................................... 245
Commands That Alter the State of the Program .................................................................. 246
assign Command .................................................................................................................. 246
pop Command ....................................................................................................................... 246
call Command ....................................................................................................................... 247
print Command ...................................................................................................................... 247
when Command ................................................................................................................... 247
fix Command .......................................................................................................................... 247
cont at Command ................................................................................................................ 248

B Event Management ........................................................................................................... 249
Event Handlers ....................................................................................................................... 249
Event Safety .......................................................................................................................... 250
Creating Event Handlers ....................................................................................................... 251
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hide Command</td>
<td>311</td>
</tr>
<tr>
<td>Syntax</td>
<td>312</td>
</tr>
<tr>
<td>ignore Command</td>
<td>312</td>
</tr>
<tr>
<td>Syntax</td>
<td>312</td>
</tr>
<tr>
<td>import Command</td>
<td>312</td>
</tr>
<tr>
<td>Syntax</td>
<td>312</td>
</tr>
<tr>
<td>intercept Command</td>
<td>313</td>
</tr>
<tr>
<td>Syntax</td>
<td>313</td>
</tr>
<tr>
<td>java Command</td>
<td>314</td>
</tr>
<tr>
<td>Syntax</td>
<td>314</td>
</tr>
<tr>
<td>jclasses Command</td>
<td>314</td>
</tr>
<tr>
<td>Syntax</td>
<td>314</td>
</tr>
<tr>
<td>joff Command</td>
<td>314</td>
</tr>
<tr>
<td>Syntax</td>
<td>314</td>
</tr>
<tr>
<td>jon Command</td>
<td>315</td>
</tr>
<tr>
<td>Syntax</td>
<td>315</td>
</tr>
<tr>
<td>jpkgs Command</td>
<td>315</td>
</tr>
<tr>
<td>Syntax</td>
<td>315</td>
</tr>
<tr>
<td>kill Command</td>
<td>315</td>
</tr>
<tr>
<td>Syntax</td>
<td>315</td>
</tr>
<tr>
<td>language Command</td>
<td>316</td>
</tr>
<tr>
<td>Syntax</td>
<td>316</td>
</tr>
<tr>
<td>line Command</td>
<td>316</td>
</tr>
<tr>
<td>Syntax</td>
<td>316</td>
</tr>
<tr>
<td>Examples</td>
<td>317</td>
</tr>
<tr>
<td>list Command</td>
<td>317</td>
</tr>
<tr>
<td>Syntax</td>
<td>317</td>
</tr>
<tr>
<td>listi Command</td>
<td>319</td>
</tr>
<tr>
<td>loadobject Command</td>
<td>319</td>
</tr>
<tr>
<td>Syntax</td>
<td>319</td>
</tr>
<tr>
<td>loadobject -dumpelf Command</td>
<td>320</td>
</tr>
<tr>
<td>loadobject -exclude Command</td>
<td>320</td>
</tr>
<tr>
<td>loadobject -hide Command</td>
<td>321</td>
</tr>
<tr>
<td>loadobject -list Command</td>
<td>321</td>
</tr>
<tr>
<td>loadobject -load Command</td>
<td>322</td>
</tr>
<tr>
<td>loadobject -unload Command</td>
<td>322</td>
</tr>
<tr>
<td>Command</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
</tr>
<tr>
<td>loadobject -use</td>
<td>322</td>
</tr>
<tr>
<td>lwp</td>
<td>323</td>
</tr>
<tr>
<td>Syntax</td>
<td>323</td>
</tr>
<tr>
<td>lwp</td>
<td>324</td>
</tr>
<tr>
<td>Syntax</td>
<td>324</td>
</tr>
<tr>
<td>macro Command</td>
<td>324</td>
</tr>
<tr>
<td>Syntax</td>
<td>324</td>
</tr>
<tr>
<td>mmapfile Command</td>
<td>324</td>
</tr>
<tr>
<td>Syntax</td>
<td>325</td>
</tr>
<tr>
<td>Example</td>
<td>325</td>
</tr>
<tr>
<td>module Command</td>
<td>325</td>
</tr>
<tr>
<td>Syntax</td>
<td>325</td>
</tr>
<tr>
<td>modules Command</td>
<td>326</td>
</tr>
<tr>
<td>Syntax</td>
<td>326</td>
</tr>
<tr>
<td>native Command</td>
<td>326</td>
</tr>
<tr>
<td>Syntax</td>
<td>326</td>
</tr>
<tr>
<td>next Command</td>
<td>327</td>
</tr>
<tr>
<td>Native Mode Syntax</td>
<td>327</td>
</tr>
<tr>
<td>Java Mode Syntax</td>
<td>328</td>
</tr>
<tr>
<td>nexti Command</td>
<td>328</td>
</tr>
<tr>
<td>Syntax</td>
<td>329</td>
</tr>
<tr>
<td>omp_loop Command</td>
<td>329</td>
</tr>
<tr>
<td>Syntax</td>
<td>329</td>
</tr>
<tr>
<td>omp_pr Command</td>
<td>329</td>
</tr>
<tr>
<td>Syntax</td>
<td>329</td>
</tr>
<tr>
<td>omp_serialize Command</td>
<td>330</td>
</tr>
<tr>
<td>Syntax</td>
<td>330</td>
</tr>
<tr>
<td>omp_team Command</td>
<td>330</td>
</tr>
<tr>
<td>Syntax</td>
<td>330</td>
</tr>
<tr>
<td>omp_tr Command</td>
<td>331</td>
</tr>
<tr>
<td>Syntax</td>
<td>331</td>
</tr>
<tr>
<td>pathmap Command</td>
<td>332</td>
</tr>
<tr>
<td>Syntax</td>
<td>332</td>
</tr>
<tr>
<td>Examples</td>
<td>332</td>
</tr>
<tr>
<td>pop Command</td>
<td>333</td>
</tr>
<tr>
<td>Syntax</td>
<td>333</td>
</tr>
<tr>
<td>Command</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>showleaks</td>
<td>344</td>
</tr>
<tr>
<td>showmemuse</td>
<td>345</td>
</tr>
<tr>
<td>source</td>
<td>345</td>
</tr>
<tr>
<td>status</td>
<td>346</td>
</tr>
<tr>
<td>step</td>
<td>346</td>
</tr>
<tr>
<td>stepi</td>
<td>348</td>
</tr>
<tr>
<td>stop</td>
<td>349</td>
</tr>
<tr>
<td>stopi</td>
<td>353</td>
</tr>
<tr>
<td>suppress</td>
<td>354</td>
</tr>
<tr>
<td>sync</td>
<td>356</td>
</tr>
<tr>
<td>syncs</td>
<td>357</td>
</tr>
<tr>
<td>thread</td>
<td>357</td>
</tr>
<tr>
<td>threads</td>
<td>358</td>
</tr>
<tr>
<td>trace</td>
<td>360</td>
</tr>
<tr>
<td>tracei</td>
<td>363</td>
</tr>
<tr>
<td>uncheck</td>
<td>364</td>
</tr>
</tbody>
</table>
Syntax ..........................................................................................................................................364
undisplay Command .......................................................................................................................365
Native Mode Syntax .......................................................................................................................365
Java Mode Syntax .......................................................................................................................365
unhide Command .........................................................................................................................366
Syntax ..........................................................................................................................................366
unintercept Command ...................................................................................................................366
Syntax ..........................................................................................................................................366
unsuppress Command ...................................................................................................................367
Syntax ..........................................................................................................................................367
unwatch Command .......................................................................................................................368
Syntax ..........................................................................................................................................368
up Command ...................................................................................................................................368
Syntax ..........................................................................................................................................368
use Command .....................................................................................................................................369
watch Command ............................................................................................................................369
Syntax ..........................................................................................................................................369
whatis Command ..........................................................................................................................369
Native Mode Syntax .......................................................................................................................370
Java Mode Syntax .......................................................................................................................371
when Command ..............................................................................................................................371
Syntax ..........................................................................................................................................371
wheni Command ............................................................................................................................373
Syntax ..........................................................................................................................................373
where Command ............................................................................................................................373
Native Mode Syntax .......................................................................................................................373
Java Mode Syntax .......................................................................................................................374
whereami Command .....................................................................................................................374
Syntax ..........................................................................................................................................374
whereis Command .........................................................................................................................375
Syntax ..........................................................................................................................................375
which Command ............................................................................................................................375
Syntax ..........................................................................................................................................375
whocatches Command .................................................................................................................376
Syntax ..........................................................................................................................................376
Preface

The Oracle Solaris Studio 12.3 Debugging a Program With dbx manual gives instructions on how to use the dbx command-line debugger, an interactive source level debugging tool.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- “x86” refers to the larger family of 64-bit and 32-bit x86 compatible products.
- “x64” points out specific 64-bit x86 compatible CPUs.
- “32-bit x86” points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the Oracle Solaris Studio 12.3 Release Notes.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.
- Man pages for command-line tools describe a tool’s command options.
Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories, and onscreen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. machine_name% you have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with onscreen computer output</td>
<td>machine_name% su Password:</td>
</tr>
<tr>
<td>aabbcc123</td>
<td>Placeholder: replace with a real name or value</td>
<td>The command to remove a file is rm filename.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new terms, and terms to be emphasized</td>
<td>Read Chapter 6 in the User’s Guide. A cache is a copy that is stored locally. Do not save the file. Note: Some emphasized items appear bold online.</td>
</tr>
</tbody>
</table>
Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash shell, Korn shell, and Bourne shell</td>
<td>$</td>
</tr>
<tr>
<td>Bash shell, Korn shell, and Bourne shell for superuser</td>
<td>#</td>
</tr>
<tr>
<td>C shell</td>
<td>machine_name%</td>
</tr>
<tr>
<td>C shell for superuser</td>
<td>machine_name#</td>
</tr>
</tbody>
</table>
Getting Started With dbx

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

You can use dbx to debug an application written in C, C++, or Fortran. You can also, with some limitations (see "Limitations of dbx With Java Code" on page 206), debug an application that is a mixture of Java code and C JNI (Java Native Interface) code or C++ JNI code.

dbx tool 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 dbx tool 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 35
- "Accessing dbx Online Help" on page 36

Compiling Your Code for Debugging

You must prepare your program for source-level debugging with dbx by compiling it with the -g option, which is accepted by the C compiler, C++ compiler, Fortran 95 compiler, and Java compiler. For more information, see “Compiling a Program for Debugging” on page 44.
Starting dbx or dbxtool and Loading Your Program

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

$ dbx

To start dbxtool, type the dbxtool command at 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 libnsi.so.1
Reading libdl.so.1
Reading libmp.so.2
Reading libc_psr.so.1
Attached to process 1855
stopped in _libc_poll at 0xefef9437c
0xefef9437c: _libc_poll+0x0004: ta 0x8
Current function is main
48  XtAppMainLoop(app_context);
(dbx)

For more information on the dbx command and start-up options, see “dbx Command” on page 293 and the dbx(1) man page, or type dbx -h.

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

(dbx) debug program_name
To load or switch to a program that includes Java code and C JNI code or C++ JNI code:

(dbx> debug program_name {.class | .jar}
If you are already running dbx, you can also use the debug command to attach dbx to a running process:

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

(dbx) debug program_name {.class | .jar} process_id
For more information on the debug command, see “debug Command” on page 296.

Running Your Program in dbx

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

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

For example:

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

When you run an application that includes Java code, the run arguments are passed to the Java application, not to the JVM software. Do not include the main class name as an argument.
If you repeat the run command without arguments, the program restarts using the arguments or redirection from the previous run command. You can reset the options using the rerun command. For more information on the run command, see “run Command” on page 341. For more information on the rerun command, see “rerun Command” on page 339.

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

### Debugging Your Program With dbx

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

- To determine where and why it is crashing. Strategies for locating the cause of a crash include:
  - Running the program in dbx. dbx reports the location of the crash when it occurs.
  - Examining the core file and looking at a stack trace (see “Examining a Core File” on page 30 and “Looking at the Call Stack” on page 33).
- To determine why your program is giving incorrect results. Strategies include:
  - Setting breakpoints to stop execution so that you can check your program's state and look at the values of variables (see “Setting Breakpoints” on page 31 and “Examining Variables” on page 34).
  - Stepping through your code one source line at a time to monitor how the program state changes (see “Stepping Through Your Program” on page 32).
- To find a memory leak or memory management problem. Runtime checking lets you detect runtime errors such as memory access errors and memory leak errors and lets you monitor memory usage (see “Finding Memory Access Problems and Memory Leaks” on page 35).

### Examining a Core File

To determine where your program is crashing, you may want to examine the core file, the memory image of your program when it crashed. You can use the where command (see “where Command” on page 373) to determine where the program was executing when it dumped core.

#### Note

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

To debug a core file, type:

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

```
% dbx a.out core
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
program terminated by signal SEGV (no mapping at the fault address)
Current function is main
  9 printf("string 's' is %d characters long\n", msg, strlen(msg));
(dbx) where
[1] strlen(0x0, 0x0, 0xff337d24, 0x7efefeff, 0x81010100, 0xff000000), at 0xff2b6dec
=>[2] main(argc = 1, argv = 0xffbef39c), line 9 in "foo.c"
(dbx)
```

For more information on debugging core files, see “Debugging a Core File” on page 38. For more information on using the call stack, see “Looking at the Call Stack” on page 33.

**Note** – If your program is dynamically linked with any shared libraries, it is best to debug the core file in the same operating environment in which it was created. For information on debugging a core file that was created in a different operating environment, see “Debugging a Mismatched Core File” on page 40.

## Setting Breakpoints

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

When your program stops at a breakpoint, you can then examine the state of program and the values of variables. dbx allows you to set many types of breakpoints (see “Using Ctrl+C to Stop a Process” on page 86).

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

```
(dbx) stop in main
(2) stop in main
```
For more information on the stop in command, see "Setting a stop Breakpoint in a Function" on page 89 and "stop Command" on page 349.

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

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

For more information on the stop at command, see "Setting a stop Breakpoint at a Line of Source Code" on page 88 and "stop Command" on page 349.

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

```
(dbx) file t.c
(dbx) list main
10 main(int argc, char *argv[])
11 {  
12    char *msg = "hello world\n";
13    printf(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 83 and "cont Command" on page 292).

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

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

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

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

**Stepping Through Your Program**

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

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 31.

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

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

For more information on stepping through your program, see “Stepping Through a Program” on page 82. For more information on the `step` and `next` commands, see “step Command” on page 346 and “next Command” on page 327.

**Looking at the Call Stack**

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

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

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

```
(dbx) print msg
msg = 0x20a84 "Hello world"
(dbx) print msg[0]
msg[0] = 'H'
(dbx) print *msg
*msg = 'H'
(dbx) print &msg
$msg = 0xeffe8b4
```

You can track when the values of variables and expressions change using data change breakpoints (see “Setting Data Change Breakpoints” on page 91). 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 turn on memory access checking only, type:

(dbx) check -access

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

(dbx) check -memuse

After turning on the types of runtime checking you want, run your program. The program runs normally, but slowly because each memory access is checked for validity just before it occurs. If dbx detects invalid access, it displays the type and location of the error. You can then use dbx commands such as the where command to get 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 on 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 using the process_id option, the process survives and continues when you quit the debugging session. dbx performs an implicit detach before quitting the session.

For more information about quitting dbx, see “Quitting Debugging” on page 48.
Accessing dbx Online Help

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

(dbx) help
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 37
- “Debugging a Core File” on page 38
- “Using the Process ID” on page 42
- “The dbx Startup Sequence” on page 43
- “Setting Startup Properties” on page 43
- “Compiling a Program for Debugging” on page 44
- “Debugging Optimized Code” on page 46
- “ Quitting Debugging” on page 48
- “Saving and Restoring a Debugging Run” on page 49

Starting a Debugging Session

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

You can use dbx entirely from the command line in a terminal window, or run dbxtool, a graphical user interface for dbx. For information on 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.

$ dbx

or

$ dbxtool
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}
```

Oracle Solaris 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.

**Note** – 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 on the dbx command and start-up options, see “dbx Command” on page 293 and the dbx(1) man page.

## Debugging a Core File

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

$ dbx program_name core

or

$ dbxtool program_name core

If you type the following, 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

You can substitute - for the program name and 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 (see "where Command" on page 373) to determine where the program was executing when it dumped core.

When you debug a core file, you can also evaluate variables and expressions to see the values they had at the time the program crashed, but you cannot evaluate expressions that make function calls. You cannot single step. 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 start-up file, re-source the start-up file, and then rerun the program that produced the core file to produce a complete core file.
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, you’ll want to make these original libraries available on the dbx-host.
- dbx uses system libraries in /usr/lib to help understand the implementation details of the run time linker and threads library on the system. It may also be necessary to provide these system libraries from the core-host so that dbx can understand the runtime linker data structures and the threads data structures.

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

dbx 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
```

Eliminating Shared Library Problems

▼ To Eliminate the Library Problems and Debug a “mismatched” Core File

1 Set the dbx environment variable core_lo_pathmap to on.

2 Use the pathmap command to tell dbx where the correct libraries for the core file are located.
3 Use the debug command to load the program and the core file.

For example, assuming that the root partition of the core-host has been exported over NFS and can be accessed 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.)

**Things to Remember**

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

- The pathmap command does not recognize a pathmap for ‘/’ so you cannot use the following command:
  ```
  pathmap / /net/core-host
  ```

- The single-argument mode for the pathmap command does not work with load object pathnames, so use the two argument from-path to-path mode.

- Debugging the core file is likely to work better if the dbx-host has either the same or a more recent version of the Solaris operating environment than the core-host, though this is not always necessary.

- The system libraries that you might need are:
  - For the runtime linker:
    ```
    /usr/lib/ld.so.1
    /usr/lib/64/ld.so.1
    /usr/lib/librtld_db.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 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:

```plaintext
(dbx) where
current thread: t@0
[1] 0x0(), at 0xffffffff
(dbx) lwp
o>t@1 signal SIGSEGV in _sigfillset()
(dbx) lwp t@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.

---

**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.

```plaintext
$ dbx program_name process_id
```

or

```plaintext
dbxtool 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:

```plaintext
$ 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.

```plaintext
$ dbx - process_id
```

or

```plaintext
$ dbxtool - process_id
```

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

For more information, see “Attaching dbx to a Running Process” on page 80.
The dbx Startup Sequence

When you start dbx, if you do not specify the -S option, dbx looks for the installed startup file, dbxrc, in the directory /installation_directory/lib. (The default installation_directory is /opt/solstudio12.2 on Solaris platforms and /opt/oracle/solstudio12.2 on Linux platforms.) If your Oracle Solaris 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 53.

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

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

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

Setting Startup Properties

You can use the pathmap 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.

To establish a new mapping from the directory from to the directory to, type:
(dbx) pathmap [-c] from to

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

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

The mapping of /tmp_mnt to / exists by default.

For more information, see “pathmap Command” on page 331.

### Setting dbx Environment Variables

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

```
$ dbxenv
```

You can also set dbx environment variables. See “Saving and Restoring Using replay” on page 52 for more information about the .dbxrc file and about setting these variables.

For more information, see “Setting dbx Environment Variables” on page 54 and “dbxenv Command” on page 296.

### 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 293.

### 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, type:

```
% CC -g example_source.cc
```
For the C++ compiler:

- The \texttt{-g} option alone (with no optimization level specified) turns on debugging information and turns off inlining of functions.
- The \texttt{-g} option used with the \texttt{-O} option or the \texttt{-xOlevel} option turns on debugging information and does not turn off inlining of functions. This set of options produces limited debugging information and inlined functions.
- The \texttt{-g0} (zero) option turns on debugging information and does not affect inlining of functions. You cannot debug inline functions in code compiled with the \texttt{-g0} option. The \texttt{-g0} option can significantly decrease link time and \texttt{dbx} start-up time (depending on the use of inlined functions by the program).

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

### Using a Separate Debug File

\texttt{dbx} lets you use options in the \texttt{objcopy} command on Linux platforms and the \texttt{gobjcopy} command on 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.

\texttt{dbx} searches for the separate debug file in the following order and reads the debugging information from the first file it finds:

- The directory that contains the executable file.
- A subdirectory named \texttt{debug} in the directory that contains the executable file.
- A subdirectory of the global debug file directory, which you can view or change if the \texttt{dbx} environment variable \texttt{debug\_file\_directory} is set to the path name of the directory. The default value of the environment variable is \texttt{/usr/lib/debug}.

For example, to create a separate debug file for executable \texttt{a.out}, you would do the following.

#### Creating a Separate Debug File

1. Create a separate debug file named \texttt{a.out.debug} containing the debugging information:
   ```sh```
   objcopy --only-keep-debug a.out a.out.debug
   ```
2. Strip the debugging information from \texttt{a.out}:
   ```sh```
   objcopy --strip-debug a.out
   ```
3. Establish the link between the two files:
   ```sh```
   objcopy --add-gnu-debuglink=a.out.debug a.out
   ```
On 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 or not 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.

### Debugging Optimized Code

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

When analyzing optimized code, you can:

- Stop execution at the start of any function (`stop in function` command)
- Evaluate, display, or modify arguments
- Evaluate, display, or modify global, 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 considered an unstable interface and is likely to change from release to release.

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

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

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 \texttt{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 Solaris Studio 12.2 compilers 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**

\texttt{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 \texttt{dbx} operations (for example, \texttt{step}, \texttt{next}, and \texttt{list} commands) on inlined functions as you can perform on non-inlined functions.

The \texttt{where} command shows the call stack with the inlined function and the parameters if location information for the inlined parameters is available.

The \texttt{up} and \texttt{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 sections *Inlining of Functions* and *Parallelization* in Chapter 8 of the *Performance Analyzer* manual contain 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 \texttt{-g}, \texttt{dbx} still provides the following level of support for code compiled without \texttt{-g}:

- Backtrace (\texttt{dbx where} command)
- Calling a function (but without parameter checking)
- Checking global variables

Note, however, that \texttt{dbx} cannot display source code unless the code was compiled with the \texttt{-g} option. This restriction also applies to code that has had \texttt{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: memory use checking works, and access checking works with code stripped with strip -x but not with code stripped with strip.

 Quitting Debugging

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

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

(dbx) quit

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

 Stopping a Process Execution

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

Detaching a Process From dbx

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

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

(dbx) detach
You can detach a process and leave it in a stopped state while you temporarily apply other /proc-based debugging tools that might be blocked when dbx has exclusive access. For more information, see “Detaching dbx From a Process” on page 81.

For more information on the detach command, see “detach Command” on page 299.

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.

To kill a program executing in dbx, type:

```
(dbx) kill
```

For more information, see “kill Command” on page 315.

Saving and Restoring a Debugging Run

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

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

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.

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

You can save part of a debugging run; that is, the whole run minus a specified number of commands from the last one entered.
If you are not sure where you want to end the run you are saving, use the `history` command to see a list of the debugging commands issued since the beginning of the session.

**Note** – By default, the `save` command writes information to a special save-file. If you want to save a debugging run to a file you can restore later, you can specify a file name with the `save` command. See “Saving a Series of Debugging Runs as Checkpoints” on page 51.

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

```
(dbx) save
```

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

```
(dbx) save -number
```
Saving a Series of Debugging Runs as Checkpoints

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

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

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

```
(dbx) save filename
```

Restoring a Saved Run

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

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

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

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

To restore a saved debugging run, type:

```
(dbx) restore
```

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

```
(dbx) restore filename
```
Saving and Restoring Using `replay`

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

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

```
(dbx) replay
```

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

```
(dbx) replay -number
```
This chapter describes the dbx environment variables you can use to customize certain attributes of your debugging environment, and how to use the initialization file, .dbxr c, to preserve your changes and adjustments from session to session.

This chapter is organized into the following sections:

- “Using the dbx Initialization File” on page 53
- “Setting dbx Environment Variables” on page 54
- “The dbx Environment Variables and the Korn Shell” on page 60

### Using the dbx Initialization File

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

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

During startup, the search order is:

1. Installation directory (unless you specify the -S option to the dbx command) 
   `/installation_directory/`Lib/dbxr c (the default `installation_directory` is `/opt/solstudio12.2` on Solaris platforms and `/opt/oracle/solstudio12.2` on Linux platform). If your Oracle Solaris Studio software is not installed in the default `installation_directory`, dbx derives the path to the dbxr c file from the path to the dbx executable.

2. Current directory ./ .dbxr c
3. Home directory $HOME/.dbxrc

Creating a .dbxrc File

To create a .dbxrc file that contains common customizations and aliases, type:

```
(dbx) help .dbxrc
```

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

Initialization File Sample

Here is a sample .dbxrc file:

```
dbxenv input_case_sensitive false
catch FPE
```

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

- `dbxenv` is the command used to set dbx environment variables. (For a complete list of dbx environment variables, see “Setting dbx Environment Variables” on page 54.)
- `input_case_sensitive` is the dbx environment variable that controls case sensitivity.
- `false` is the setting for `input_case_sensitive`.

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

Setting dbx Environment Variables

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

To display the value of a specific variable, type:

```
(dbx) dbxenv variable
```

To show all variables and their values, type:

```
(dbx) dbxenv
```

To set the value of a variable, type:

```
(dbx) dbxenv variable value
```
Table 3–1 shows all of the dbx environment variables that you can set.

**TABLE 3–1  dbx Environment Variables**

<table>
<thead>
<tr>
<th>dbx Environment Variable</th>
<th>What the Variable Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_bounds_check on</td>
<td>off</td>
</tr>
<tr>
<td>c_array_op on</td>
<td>off</td>
</tr>
<tr>
<td>CLASSPATHX</td>
<td></td>
</tr>
<tr>
<td>core_lo_pathmap on</td>
<td>off</td>
</tr>
<tr>
<td>disassembler_version</td>
<td></td>
</tr>
<tr>
<td>event_safety on</td>
<td>off</td>
</tr>
<tr>
<td>fix_verbose on</td>
<td>off</td>
</tr>
<tr>
<td>follow_fork_inherit on</td>
<td>off</td>
</tr>
<tr>
<td>follow_fork_mode parent</td>
<td>child</td>
</tr>
<tr>
<td>follow_fork_mode inner unset</td>
<td>parent</td>
</tr>
</tbody>
</table>

Setting dbx Environment Variables
### TABLE 3-1  dbx Environment Variables (Continued)

<table>
<thead>
<tr>
<th>dbx Environment Variable</th>
<th>What the Variable Does</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input_case_sensitive</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>true/false</strong></td>
<td></td>
</tr>
<tr>
<td><strong>JAVASRCPATH</strong></td>
<td>Specifies the directories in which dbx should look for Java source files.</td>
</tr>
<tr>
<td>**jdbx_mode java</td>
<td>jni</td>
</tr>
<tr>
<td><strong>jvm_invocation</strong></td>
<td>The jvm_invocation environment variable lets you customize the way the JVM software is started. (The terms “Java virtual machine” and “JVM” mean a virtual machine for the Java platform.) For more information, see “Customizing Startup of the JVM Software” on page 211.</td>
</tr>
</tbody>
</table>
| **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 274 for more information on this environment variable. 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: off. |
### TABLE 3–1  dbx Environment Variables  

(Continued)

<table>
<thead>
<tr>
<th>dbx Environment Variable</th>
<th>What the Variable Does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mt_scalable</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>mt_sync_tracing</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_auto_flush</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_base</code> 8</td>
<td>10</td>
</tr>
<tr>
<td><code>output_class_prefix</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_dynamic_type</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_inherited_members</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_list_size num</code></td>
<td>Governs the default number of lines to print in the list command. Default: 10.</td>
</tr>
<tr>
<td><code>output_log_file_name filename</code></td>
<td>Name of the command logfile. Default: <code>/tmp/dbx.log.uniqueID</code>.</td>
</tr>
<tr>
<td><code>output_max_string_length number</code></td>
<td>Sets number of characters printed for char *s. Default: 512.</td>
</tr>
<tr>
<td><code>output_no_literal</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_pretty_print</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_pretty_print_fallback</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>output_short_file_name</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>overload_function</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>overload_operator</code> on</td>
<td>off</td>
</tr>
<tr>
<td><code>pop_auto_destruct</code> on</td>
<td>off</td>
</tr>
</tbody>
</table>
### Setting dbx Environment Variables

<table>
<thead>
<tr>
<th>dbx Environment Variable</th>
<th>What the Variable Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>**proc_exclusive_attach on</td>
<td>off**</td>
</tr>
<tr>
<td>**rtc_auto_continue on</td>
<td>off**</td>
</tr>
<tr>
<td>**rtc_auto_suppress on</td>
<td>off**</td>
</tr>
<tr>
<td>**rtc_biu_at_exit on</td>
<td>off</td>
</tr>
<tr>
<td><strong>rtc_error_limit number</strong></td>
<td>Number of RTC access errors to be reported. Default: 1000.</td>
</tr>
<tr>
<td><strong>rtc_error_log_file_name filename</strong></td>
<td>Name of file to which RTC errors are logged if rtc_auto_continue is set. Default: /tmp/dbx.errlog.uniqueID</td>
</tr>
<tr>
<td>**rtc_error_stack on</td>
<td>off**</td>
</tr>
<tr>
<td>**rtc_inherit on</td>
<td>off**</td>
</tr>
<tr>
<td>**rtc_mel_at_exit on</td>
<td>off</td>
</tr>
<tr>
<td>**run_autostart on</td>
<td>off**</td>
</tr>
<tr>
<td>**run_io stdio</td>
<td>pty**</td>
</tr>
<tr>
<td><strong>run_pty ptyname</strong></td>
<td>Sets the name of the pty to use when run_io is set to pty. Prys are used by graphical user interface wrappers.</td>
</tr>
<tr>
<td>dbx Environment Variable</td>
<td>What the Variable Does</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>run_quick on</td>
<td>off</td>
</tr>
<tr>
<td>run_savetty on</td>
<td>off</td>
</tr>
<tr>
<td>run_setpgrp on</td>
<td>off</td>
</tr>
<tr>
<td>scope_global Enums on</td>
<td>off</td>
</tr>
<tr>
<td>scope look aside on</td>
<td>off</td>
</tr>
<tr>
<td>session_log_file_name filename</td>
<td>Name of the file where dbx logs all commands and their output. Output is appended to the file. Default: &quot;&quot; (no session logging).</td>
</tr>
<tr>
<td>show_static_members</td>
<td>When set to on, -S is the default for printing, watches, and displaying. Default: on.</td>
</tr>
<tr>
<td>stack_find_source on</td>
<td>off</td>
</tr>
<tr>
<td>stack_max_size number</td>
<td>Sets the default size for the where command. Default: 100.</td>
</tr>
<tr>
<td>stackVerbose on</td>
<td>off</td>
</tr>
<tr>
<td>step_abflow stop</td>
<td>ignore</td>
</tr>
<tr>
<td>step_events on</td>
<td>off</td>
</tr>
</tbody>
</table>

**Table 3-1** dbx Environment Variables (Continued)
### The dbx Environment Variables and the Korn Shell

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

<table>
<thead>
<tr>
<th>dbx Environment Variable</th>
<th>What the Variable Does</th>
</tr>
</thead>
</table>
| **step_granularity statement | line** | Controls granularity of source line stepping. When set to statement the following code:
| | a(); b(); | takes the two next commands to execute. When set to line a single next command executes the code. The granularity of line is particularly useful when dealing with multiline macros. Default: `statement`.
| 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. |
| vdl_mode classic | lisp | xml | Value Description Language (VDL) is used to communicate data structures to the graphical user interface (GUI) for dbx. classical mode was used for the Sun WorkShop IDE. Lisp mode is used by the IDE in Sun Studio and Oracle Solaris Studio releases. XML mode is experimental and unsupported. Default: value is set by the GUI. |
Each time the program you are debugging stops, dbx prints the source line associated with the stop location. At each program stop, dbx resets the value of the current function to the function in which the program is stopped. Before the program starts running and when it is stopped, you can move to, or navigate through, functions and files elsewhere in the program.

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

This chapter is organized into the following sections

- “Navigating To Code” on page 61
- “Types of Program Locations” on page 63
- “Program Scope” on page 64
- “Qualifying Symbols With Scope Resolution Operators” on page 66
- “Locating Symbols” on page 68
- “Viewing Variables, Members, Types, and Classes” on page 71
- “Debugging Information in Object Files and Executables” on page 74
- “Finding Source and Object Files” on page 76

**Navigating To Code**

When a program is stopped, you can navigate to code elsewhere in the program. You can navigate to any function or file that is part of the program. Navigating sets the current scope (see “Program Scope” on page 64). It is useful for determining when and at what source line you want to set a stop at breakpoint.
Navigating To a File

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

(dbx) **file** filename

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

(dbx) **file**

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

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

For information on setting a stop at breakpoint at a line of source code, see “Setting a stop Breakpoint at a Line of Source Code” on page 88.

Navigating To Functions

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

(dbx) **func** adjust_speed

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

For more information, see “**func** Command” on page 309

Selecting From a List of C++ Ambiguous Function Names

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

(dbx) **func** block::block

Choosing Among Multiple Occurrences

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

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

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

**Printing a Source Listing**

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

For detailed information on the `list` command, see “list Command” on page 317.

**Walking the Call Stack to Navigate To Code**

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

For more information on walking the call stack, see “Walking the Stack and Returning Home” on page 102.

**Types of Program Locations**

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

- The current address, which is used and updated by the `dis` command (see “dis Command” on page 300) and the `examine` command (see “examine Command” on page 304).
- The current source code line, which is used and updated by the `list` command (see “list Command” on page 317). This line number is reset by some commands that alter the visiting scope (see “Changing the Visiting Scope” on page 65).
- The current visiting scope, which is a compound variable described in the “Visiting Scope” on page 64. The visiting scope is used during expression evaluation. It is updated by the `line` command, the `func` command, the `file` command, the `list func` command and the `list file` command.
Program Scope

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

Variables That Reflect the Current Scope

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

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

Visiting Scope

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

(dbx) print i

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

Components of the Visiting Scope

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

$v scope    Current visiting scope
$v loadobj  Current visiting load object
$v file     Current visiting file
$v lineno   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
- list procedure

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

When you hit a breakpoint, dbx sets the visiting scope to the current location. If the `stack_find_source` environment variable (see “Setting dbx Environment Variables” on page 54) is set to ON, dbx attempts to find and make active a stack frame that has source code.

When you use the up command (see “up Command” on page 368), the down command (“down Command” on page 302), the frame number command (see “frame Command” on page 308), or the pop command (see “pop Command” on page 333) to change the current stack frame, dbx sets the visiting scope according to the program counter from the new stack frame.

The line number location used by the list command (see “list Command” on page 317) changes the visiting scope only if you use the list function or list file command. When the visiting scope is set, the line number location for the list command is set to the first line number of the visiting scope. When you subsequently use the list command, the current line number location for the list command is updated, but as long as you are listing lines in the current file, the visiting scope does not change. For example, if you type the following, dbx lists the start of the source for my_func, and changes the visiting scope to my_func.

```
(dbx) list my_func
```

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

```
(dbx) list 127
```

When you use the file command or the func command to change the current file or the current function, the visiting scope is updated accordingly.
Qualifying Symbols With Scope Resolution Operators

When using the `func` 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 (``). 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 `what is` command). The symbol qualifying rules are the same in all cases; this section covers the rules for all types of symbol name qualifying.

**Backquote Operator**

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

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

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

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

**C++ Double Colon Scope Resolution Operator**

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

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

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

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

For example, if `hand` is the class name and `draw` is the function name, type:
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,$
  #include <stddef.h>
  int main(int argc, char** argv) {
    int i=1;
    {
      int i=2;
      {
        int j=4;
        int i=3;
        printf("hello");
      }
      printf("world\n");
      printf("hi\n");
    }
    printf("hi\n");
  }

whereis i
variable: `a.out.t.c'main'1
variable: `a.out.t.c'main:8'1
variable: `a.out.t.'main:10'1
(dbx) stop at 12 ; run...
(dbx) print i
  i = 3
(dbx) which i
  `a.out.t.c'main:10'1
(dbx) print `main:7' i
  `a.out.t.c'main:10'1
(dbx) print `main:8' i
```

Qualifying Symbols With Scope Resolution Operators
Linker Names

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

```plaintext
(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 (see “which Command” on page 375).

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:

```plaintext
(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.cc\table
```

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

Because information from the dbx symbol table is read in as it is needed, the whereis command registers only occurrences of a symbol that are already loaded. As a debugging session gets longer, the list of occurrences can grow (see “Debugging Information in Object Files and Executables” on page 74).
For more information, see “whereis Command” on page 375.

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:

```c
(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.

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

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

The `which` command gives you a preview of what happens if you make `symbol` (in this example, `block`) an argument of a command that must operate on `symbol` (for example, a `print` command). In the case of ambiguous names, the overload display list indicates that `dbx` does not yet register which occurrence of two or more names it uses. `dbx` lists the possibilities and waits for you to choose one. For more information on the `which` command, see “which Command” on page 375.
Scope Resolution Search Path

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

1. Within the scope of the current function using the current visiting scope (see “Visiting Scope” on page 64). If the program is stopped in a nested block, dbx searches within that block, then in the scope of all enclosing blocks.
2. For C++ only: class members of the current function’s class and its base class.
3. For C++ only: the current name space.
4. The parameters of the current function.
5. The immediately enclosing module, generally, the file containing the current function.
6. Symbols that were made private to this shared library or executable. These symbols can be created using linker scoping.
7. Global symbols for the main program, and then for shared libraries.
8. If none of the above searches are successful, dbx assumes you are referencing a private, or file static, variable or function in another file. dbx optionally searches for a file static symbol in every compilation unit depending on the value of the dbxenv setting scope_look_aside.

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

Relaxing the Scope Lookup Rules

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

dbxenv scope_look_aside on

or use the “double backquote” prefix:

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

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

- Static variables defined in other files if not found in current scope. Files from libraries in /usr/lib are not searched.
- C++ member functions without class qualification.
- Instantiations of C++ inline member functions in other files if a member function is not instantiated in current file.
The `which` command tells you which symbol `dbx` would choose. In the case of ambiguous names, the overload display list indicates that `dbx` has not yet determined which occurrence of two or more names it would use. `dbx` lists the possibilities and waits for you to choose one.

For more information, see "`func Command` on page 309.

**Viewing Variables, Members, Types, and Classes**

The `what is` 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 "`what is Command` on page 369.

**Looking Up Definitions of Variables, Members, and Functions**

To print out the declaration of an identifier, type:

```
(dbx) what is identifier
```

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

For C++ programs, `what is identifier` lists function template instantiations. Template definitions are displayed with `what is -t identifier`. See "Looking Up Definitions of Types and Classes" on page 72.

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

To print out the member function, type:

```
(dbx) what is block::draw
void block::draw(unsigned long pw);
(dbx) what is table::draw
void table::draw(unsigned long pw);
```

To print out the data member, type:
On a variable, the `whatis` command tells you the variable's type:

```c
whatis block::movable
int movable;
```

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

```c
whatis the_table
class table *the_table;
```

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

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

## Looking Up Definitions of Types and Classes

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

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

```c
whatis -t type_or_class_name
```

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

```c
whatis -t -r class_name
```

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

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

Without `-r`, `whatis` reports the members declared in class `table`:
(dbx) **whatis -t class table**
class table : public load_bearing_block {
  public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};

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

(dbx) **whatis -t -r class table**
class table : public load_bearing_block {
  public:
    /* from base class table::load_bearing_block::block */
    block::block();
    block::block(char *name, int w, int h, const class point &pos, class load_bearing_block *blk);
    virtual char *block::type();
    char *block::name();
    int block::is_movable();
    // deleted several members from example protected:
    char *nm;
    int movable;
    int width;
    int height;
    class point position;
    class load_bearing_block *supported_by;
    Panel item panel_item;
    /* from base class table::load_bearing_block */
  public:
    load_bearing_block::load_bearing_block();
    load_bearing_block::load_bearing_block(char *name, int w, int h,
                                           const class point &pos, class load_bearing_block *blk);
    virtual int load_bearing_block::is_load_bearing();
    virtual class list *load_bearing_block::supported_blocks();
    void load_bearing_block::add_supported_block(class block &b);
    void load_bearing_block::remove_supported_block(class block &b);
    virtual void load_bearing_block::print_supported_blocks();
    virtual void load_bearing_block::put_on(class block &object);
    class point load_bearing_block::get_space(class block &object);
    class point load_bearing_block::find_space(class block &object);
    class point load_bearing_block::make_space(class block &object);
  protected:
    class list *support_for;
    /* from class table */
  public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
Debugging Information in Object Files and Executables

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

dbx parses and loads debugging information for each object file (module) on demand, when the information is needed. You can ask dbx to load debug information for any specific module, or for all modules, by using the module command. See also “Finding Source and Object Files” on page 76.

Object File Loading

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

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

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

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

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

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

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

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

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

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

To read in debugging information for all modules, type:

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

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

To print the name of the current module, type:

```
(dbx) module
```

Listing Modules

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

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

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

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

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

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

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

where:
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 $\texttt{pathmap}$ 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 pathname of a source file, but when you type $dbx$ commands, you typically use only the basename of a file. For example:

\begin{verbatim}
stop at test.cc:34
\end{verbatim}

$dbx$ 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 know what 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 $\texttt{pathmap}$ 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 $\texttt{from}$ to the directory $\texttt{to}$:

\begin{verbatim}
(dbx) pathmap [-c] from to
\end{verbatim}

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

The $\texttt{pathmap}$ 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 $/\texttt{tmp\_mnt}$ to $/$ exists by default.
For more information, see “pathmap Command” on page 331.
The commands used for running, stepping, and continuing (run, rerun, next, step, and cont) are called process control commands. Used together with the event management commands described in “cont at Command” on page 248, you can control the run-time behavior of a program as it executes under dbx.

This chapter is organized into the following sections:

- “Running a Program” on page 79
- “Attaching dbx to a Running Process” on page 80
- “Detaching dbx From a Process” on page 81
- “Stepping Through a Program” on page 82
- “Using Ctrl+C to Stop a Process” on page 86

### Running a Program

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

You can set breakpoints in the program before running it.

**Note** – When debugging an application that is a mixture of Java code and C JNI (Java Native Interface) code or C++ JNI code, you may want to set breakpoints in code that has not yet been loaded. For information on setting breakpoints on such code, see “Setting Breakpoints in Native (JNI) Code” on page 211.

Use the run command to start program execution.

To run a program in dbx without arguments, type:
You can optionally add command-line arguments and redirection of input and output.

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

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

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

The `run` command without arguments restarts the program using the previous arguments and redirection. For more information, see "run Command" on page 341. The `rerun` command restarts the program and clears the original arguments and redirection. For more information, see "rerun Command" on page 339.

## Attaching dbx to a Running Process

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

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

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

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

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

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

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

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

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

or

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

For more information, see "debug Command" on page 296 and "attach Command" on page 278.

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, SIGTTOIN signal, or SIGTTOOUT 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 137.

Detaching dbx From a Process

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

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

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

```
(dbx) oproc=$PROC
(dbx) detach -stop
(dbx) /usr/proc/bin/pwdx $oproc
(dbx) attach $PROC
```

For more information, see “detach Command” on page 299.

### 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 let the program execute one source line before stopping again.

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

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

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

For more information on the next and step commands, see “next Command” on page 327 and “step Command” on page 346.

### Single Stepping

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

```
(dbx) next \(n\)
```

or

```
(dbx) step \(n\)
```

The step_granularity environment variable determines the unit by which the step command and next command step through your code (see “Setting dbx Environment Variables” on page 54). The unit can be either statement or line.
The `step_events` environment variable controls whether breakpoints are enabled during a step (see "Setting dbx Environment Variables" on page 54).

The `step_abflow` environment variable controls whether dbx stops when it detects that an abnormal control flow change is about to happen (see “Setting dbx Environment Variables” on page 54). Such a control flow change can be caused by a call to `siglongjmp()` or `longjmp()` or an exception throw.

**Stepping Into a Function**

To step into a function called from the current source code line, use the `step` command. For example, to step into the function `GetDiscount()`, type:

```
step to GetDiscount
```

To step into the last function called, type:

```
step to
```

**Continuing Execution of a Program**

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

```
(dbx) cont
```

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

To continue a program at a specified line, type:

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

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

Using 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 Resume Program Execution at a Specific Line**

1. Use the `assign` command to give the variable a correct value.
Use \texttt{cont at line\_number} to skip the line that contains the function call that would have computed the value incorrectly.

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

\begin{verbatim}
(dbx) assign speed = 180; cont at 124;
\end{verbatim}

For more information, see \textit{“cont Command” on page 292}.

If you use the \texttt{cont} command with a \texttt{when} breakpoint command, the program skips the call to \texttt{how\_fast()} each time the program attempts to execute line 123.

\begin{verbatim}
(dbx) when at 123 { assign speed = 180; cont at 124;}
\end{verbatim}

For more information on the \texttt{when} command, see:

\begin{itemize}
  \item “Setting a \texttt{stop} Breakpoint at a Line of Source Code” on page 88
  \item “Setting Breakpoints in \texttt{member} Functions of Different Classes” on page 90
  \item “Setting Breakpoints in All Member Functions of a Class” on page 90
  \item “Setting Multiple Breakpoints in Nonmember Functions” on page 91
  \item \texttt{when Command} on page 371
\end{itemize}

\section*{Calling a Function}

When a program is stopped, you can call a function using the \texttt{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:

\begin{verbatim}
(dbx) call change\_glyph(1,3)
\end{verbatim}

While the parameters are optional, you must type the parentheses after the \texttt{function\_name}. For example:

\begin{verbatim}
(dbx) call type\_vehicle()
\end{verbatim}

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

A C++ virtual function can be called like any other function using the \texttt{print} command or \texttt{call} command (see \textit{“print Command” on page 333} or \textit{“call Command” on page 279}), or any other command that executes a function call.

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

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

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

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

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

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. Here are some scenarios to watch out for and how you can extricate yourself from
them.

- 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 multi-threaded application, all threads are resumed in order to
  avoid deadlocks, so you may 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 160).

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:

- 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:

- 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.
Setting Breakpoints and Traces

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

A trace displays information about an event in your program, such as a change in the value of a variable. Although a trace’s behavior is different from that of a breakpoint, traces and breakpoints share similar event handlers (see “Event Handlers” on page 249).

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 252.

This chapter is organized into the following sections:

- “Setting Breakpoints” on page 88
- “Setting Filters on Breakpoints” on page 94
- “Tracing Execution” on page 96
- “Setting a when Breakpoint at a Line” on page 97
- “Setting Breakpoints in Dynamically Loaded Libraries” on page 98
- “Listing and Clearing Breakpoints” on page 98
- “Enabling and Disabling Breakpoints” on page 99
- “Efficiency Considerations” on page 99
Setting Breakpoints

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

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

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

- **trace** breakpoints. If a program arrives at a breakpoint created with a **trace** command, the program halts and an event-specific **trace** information line is emitted, then the program continues.

The **stop**, **when**, and **trace** commands all take as an argument an event specification, which describes the event on which the breakpoint is based. Event specifications are discussed in detail in "Setting Event Specifications" on page 252.

To set machine-level breakpoints, use the **stopi**, **wheni**, and **tracei** commands (see Chapter 18, “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 may 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 211.

---

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

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

\[
(dbx) \text{stop at filename:} n
\]

For example

\[
(dbx) \text{stop at main.cc:} 3
\]

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:
Setting Breakpoints

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

```
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 when:

- 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 I C++). In this case, dbx accepts the name if you prefix it with a `#` (see “Linker Names” on page 68).

Consider the following set of declarations:

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

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

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

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

```
stop in x::bar()
```
If you type:

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

If you type:

```
stop in 'bar
```

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

For more information on specifying an in function event, see "in function" on page 252.

**Setting Multiple Breaks in C++ Programs**

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

**Setting Breakpoints in Member Functions of Different Classes**

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

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

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

For more information on specifying an inmember or inmethod event, see "inmember function inmethod function" on page 253.

**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.

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

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

To set a breakpoint in all member functions defined in the class `shape`, and also in functions inherited from the class, type:
stop inclass shape -recurse

For more information on specifying an inclass event, see "inclass classname [-recurse | -norecurse]" on page 254 and "stop Command" on page 349.

Due to the large number of breakpoints that may be inserted by stop inclass and other breakpoint selections, you should be sure to set the dbx environment variable step_events to on to speed up the step and next commands (see “Efficiency Considerations” on page 99).

Setting Multiple Breakpoints in Nonmember Functions

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

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

(dbx) stop infunction sort

For more information on specifying an infunction event, see “infunction function” on page 253.

Setting Breakpoints in Objects

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

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

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

(dbx) stop inobject &foo

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

(dbx) stop inobject &foo -norecurse

For more information on specifying an inobject event, see "inobject object-expression [-recurse | -norecurse]" on page 254 and “stop Command” on page 349.

Setting Data Change Breakpoints

You can use data change breakpoints in dbx to note when the value of a variable or expression has changed.
Stopping Execution When an Address Is Accessed

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

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

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

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

`mode` can also contain either of the following:

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

In both cases the program counter will point at the 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 give a symbolic expression, the size of the region to be watched is automatically deduced; you can override it by specifying `byte-size-expression`. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory.

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

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

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

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

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

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

For more information on specifying an access event, see “`access mode address-expression [, byte-size-expression]`” on page 254 and “`stop Command`” on page 349.
Stopping Execution When Variables Change

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

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

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

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

For more information on specifying a change event, see “change variable” on page 255 and “stop Command” on page 349.

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

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

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

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

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

Stopping Execution on a Condition

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

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

The program stops executing when the `condition` occurs.
Keep these points in mind when using the `stop cond` command:

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

For more information on specifying a condition event, see "cond condition-expression" on page 255 and "stop Command" on page 349.

### 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 at a line or in a function, add an optional `-if condition` modifier statement to the end of a `stop` or `trace` command.

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

With a location-based breakpoint like `in` or `at`, the scope 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 66) to specify the scope precisely.

These two filters are not the same:

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

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

### Using the Return Value of a Function Call as a Filter

You can use 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)
```
Setting Data Change Breakpoints on Local Variables

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.

Using a Filter With a Conditional Event

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

Consider this example:

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

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

If you set a breakpoint with a filter that contains function calls in a multithreaded program, dbx stops execution of all threads when it hits the breakpoint, and then evaluates the condition. If the condition is met and the function is called, dbx resumes all threads for the duration of the call.
For example, you might set the following breakpoint in a multithreaded application where many threads call `lookup()`:

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

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

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

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

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

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

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

For detailed information on event modifiers, see "Event Specification Modifiers" on page 262.

---

**Tracing Execution**

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

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

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

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

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

For the complete syntax of the `trace` command, see “trace Command” on page 360.

The information a trace provides depends on the type of `event` associated with it (see “Setting Event Specifications” on page 252).

**Controlling the Speed of a Trace**

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

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

```
dbxenv trace_speed number
```

**Directing Trace Output to a File**

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

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

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

**Setting a when Breakpoint at a Line**

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

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

The `when` command operates with an implied `cont` command. In the example above, after listing the source code at the current line, the program continues executing. If you included a `stop` command after the `list` command, the program would not continue executing.
For the complete syntax of the `when` command, see “when Command” on page 371. For detailed information on event modifiers, see “Event Specification Modifiers” on page 262.

**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 table into `dbx` and set a breakpoint on the function as follows:

  ```
  (dbx) loadobject -load fullpath_to/mylibrary.so
  (dbx) stop in myfunc
  ```

- Alternatively, 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 fullpath_to/mylibrary.so
  (dbx) stop in myfunc
  ```

**Listing and Clearing Breakpoints**

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

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

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

Deleting Specific Breakpoints Using Handler ID Numbers

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

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

(dbx) delete 3 5

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

(dbx) delete all

For more information, see “delete Command” on page 299.

Enabling and Disabling Breakpoints

Each event management command (stop, trace, when) that you use to set a breakpoint creates an event handler (see “Event Handlers” on page 249). Each of these commands returns a number known as the handler ID (hid). You can use the handler ID as an argument to the handler command (see “handler Command” on page 311) to enable or disable the breakpoint.

Efficiency Considerations

Various events have different degrees of overhead in respect to the execution time of the program being debugged. Some events, like the simplest breakpoints, have practically no overhead. Events based on a single breakpoint have minimal overhead.
Multiple breakpoints such as `in\lass`, that might result in hundreds of breakpoints, have an overhead only during creation time. `dbx` uses permanent breakpoints; the breakpoints are retained in the process at all times and are not taken out on every stoppage and put in on every `cont` command.

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

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

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

Do not use `trace -in main` because the `trace` is effective in the functions called by `main` as well. Do use it in the cases where you suspect that the `lookup()` function is clobbering your variable.
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 357 for information on how to change the current thread.

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.

This chapter is organized into the following sections:

- “Finding Your Place on the Stack” on page 102
- “Walking the Stack and Returning Home” on page 102
- “Moving Up and Down the Stack” on page 102
- “Popping the Call Stack” on page 103
- “Hiding Stack Frames” on page 104
- “Displaying and Reading a Stack Trace” on page 104
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 38).

For more information on the `where` command, see “`where` Command” on page 373.

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 368.
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 302.

Moving to a Specific Frame

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

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

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

You can also move to a specific frame using the pop command (see “Popping the Call Stack” on page 103).

Popping the Call Stack

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

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

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

```
pop 5
```

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

For more information, see “pop Command” on page 333.

## Hiding Stack Frames

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

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

```
hide [ regular_expression ]
```

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

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

```
unhide 0
```

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

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

## Displaying and Reading a Stack Trace

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

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

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

You can stop in a function that was not compiled with the `-g` option. When you stop in such a function `dbx` searches down the stack for the first frame whose function is compiled with the `-g` option and sets the current scope (see "Program Scope" on page 64) to it. This is denoted by the arrow symbol (=>).

In the following example, `main()` was compiled with the `-g` option, so the symbolic names as well as the values of the arguments are displayed. The library functions called by `main()` were not compiled with `-g`, so the symbolic names of the functions are displayed but the hexadecimal contents of the SPARC input registers $i0$ through $i15$ are shown for the arguments.
In the following example, the program has crashed with a segmentation fault. The cause of the crash 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=0x0000: 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"
```

For more examples of stack traces, see "Looking at the Call Stack" on page 33 and "Tracing Calls" on page 196.
In dbx, you can perform two types of data checking:

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

This chapter is organized into the following sections:

- “Evaluating Variables and Expressions” on page 107
- “Assigning a Value to a Variable” on page 111
- “Evaluating Arrays” on page 111
- “Using Pretty-Printing” on page 115

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 375 and “whereis Command” on page 375.
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 66. For example:
  
  (dbx) print 'item

- Visit the function by changing the current function. See "Navigating To Code" on page 61.

Printing the Value of a Variable, Expression, or Identifier

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

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

print expression

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

For more information, see “print Command” on page 333.

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

Printing C++ 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 dbx environment variable output_derived_type.

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

Evaluating Unnamed Arguments in C++ Programs

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

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

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

```
_ARG%n
```

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

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

For more information, see “whatis Command” on page 369.

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

```shell
(dbx) print _ARG1
_ARGL1 = 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`:

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

## Monitoring Expressions

Monitoring the value of an expression each time the program stops is an effective technique for learning how and when a particular expression or variable changes. The `display` command instructs `dbx` to monitor one or more specified expressions or variables. Monitoring continues until you turn it off with the `undisplay` command. The `watch` command evaluates and prints expressions at every stopping point in the scope current at that stop point.

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

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

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

For more information, see “display Command” on page 301.

To watch the value of expression `expression` at every stopping point, type:

```
watch expression, ...
```

For more information, see “watch Command” on page 369.

## Turning Off Display (Undisplaying)

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

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

```
undisplay expression
```

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

```
undisplay 0
```

For more information, see “undisplay Command” on page 365.
Assigning a Value to a Variable

To assign a value to a variable, type:

\[ \text{assign } \text{variable } = \text{expression} \]

Evaluating Arrays

You evaluate arrays the same way you evaluate other types of variables. Here is a sample Fortran array:

\[ \text{integer*4 arr(1:6, 4:7)} \]

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

(dbx) \text{print arr(2,4)}

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

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

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

Array Slicing

Array slicing is supported in the `print`, `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:

\[ \text{print array-expression [first-expression . . last-expression : stride-expression]} \]

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 \( \text{stride-expression} - 1 \)). 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:

```
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
```
To specify rows and columns, type:

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, type:

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

To print column 4, type:

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

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

```cpp
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.

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

```cpp
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. Here are examples showing how to use the shorthand syntax.

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

```plaintext
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.

```plaintext
print arr
```

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

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

### Using Pretty-Printing

Pretty-printing lets your program 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
The `db_pretty_print()` function can be either a static member function or a standalone function.

If the `dbx` environment 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.

Consider also the following:

- Prior to version 7.6 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, the ability of watches, local variables, and balloon evaluation to utilize 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.

- 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.

- Pretty-print functions are invoked for the following:
  - `print -p` or if the `dbx` environment variable `output_pretty_print` is set to on.
  - `display -p` or if the `dbx` environment variable `output_pretty_print` is set to on.
  - `watch -p` or if the `dbx` environment variable `output_pretty_print` is set to on.
  - Balloon evaluation if the `dbx` environment variable `output_pretty_print` is set to on.
  - Local variable if the `dbx` environment variable `output_pretty_print` is set to on.

- Pretty-print functions are not invoked for the following:
  - `[]`. The rationale is that `[]` is intended to be used in scripts and need to be predictable.
• The `dump` command. `dump` uses the same simplified formatting as the `where` command, which may be converted to use pretty-printing in the future. This limitation does not apply to the Local Variables widow in the IDE.

• Nested values will not be pretty-printed because `dbx` does not have the infrastructure to calculate the addresses of nested fields.

• The `db_pretty_print()` 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 may still point to a poorly initialized object.

• The `dbx` environment variable `output_pretty_print_fallback` is set by default to on, meaning that `dbx` will fall back on regular formatting if pretty-printing fails. If the environment variable is set to off, `dbx` will issue an error message if pretty-printing fails.

• 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, the above failures are silent and `dbx` falls back on regular formatting. But if the `output_pretty_print_fallback` environment variable is set to off, `dbx` will issue an error message if pretty-printing fails.

However, if you use the `print -p` command rather than setting the `dbx` environment variable `output_pretty_print` to on, `dbx` stops in the broken function and allows you to diagnose the cause of failure. You can then use the `pop -c` command to clean up the call.

• 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.
Using Runtime Checking

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

The following topics are covered in this chapter:
- “Capabilities of Runtime Checking” on page 119
- “Using Runtime Checking” on page 121
- “Using Access Checking” on page 124
- “Using Memory Leak Checking” on page 126
- “Using Memory Use Checking” on page 131
- “Suppressing Errors” on page 132
- “Using Runtime Checking on a Child Process” on page 135
- “Using Runtime Checking on anAttached Process” on page 137
- “Using Fix and Continue With Runtime Checking” on page 139
- “Runtime Checking Application Programming Interface” on page 140
- “Using Runtime Checking in Batch Mode” on page 141
- “Troubleshooting Tips” on page 143
- “Runtime Checking Limitations” on page 143
- “Runtime Checking Errors” on page 146

Capabilities of Runtime Checking

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

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

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

You can use runtime checking by using the check command.

When to Use Runtime Checking

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

Runtime Checking Requirements

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

- 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 140.
- Programs that are not fully stripped; programs stripped with strip -x are acceptable.

For information on the limitations of runtime checking, see “Runtime Checking Limitations” on page 143.
Using Runtime Checking

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

Turning On Memory Use and Memory Leak Checking

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

(dbx) check -memuse

When memory use checking or memory leak checking is turned on, the showblock command shows the details about the heap block at a given address. The details include the location of the block's allocation and its size. For more information, see “showblock Command” on page 344.

Turning On Memory Access Checking

To turn on memory access checking only, type:

(dbx) check -access

Turning On All Runtime Checking

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

(dbx) check -all

For more information, see “check Command” on page 282.

Turning Off Runtime Checking

To turn off runtime checking entirely, type:

(dbx) uncheck -all

For detailed information, see “uncheck Command” on page 364.

Running Your Program

After turning on the types of runtime checking you want, run the program being tested, with or without breakpoints.
The program runs normally, but slowly because each memory access is checked for validity just before it occurs. If dbx detects invalid access, it displays the type and location of the error. Control returns to you (unless the dbx environment variable rtc_auto_continue is set to on (see “Setting dbx Environment Variables” on page 54.)

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 292.

If the rtc_auto_continue environment variable is set to on, runtime checking continues to find errors, and keeps running automatically. It redirects errors to the file named by the dbx environment variable rtc_error_log_file_name. (See “Setting dbx Environment Variables” on page 54.) The default log file name is /tmp/dbx.errlog.uniqueid.

You can limit the reporting of runtime checking errors using the suppress command. For detailed information, see “suppress Command” on page 354.

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

```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    i = j;
 29    i = j;
 30 }
```
```c
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 0xffff068
which is 96 bytes above the current stack pointer
Variable is 'j'
Current function is access_error
2 9 i=j ;
(dbx) cont
hello world
Checking for memory leaks...
Actual leaks report (actual leaks: 1 total size: 32 bytes)

<table>
<thead>
<tr>
<th>Total</th>
<th>Num of Leaked</th>
<th>Allocation call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Blocks Block Address</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0x21aa8 memory_leak &lt; main</td>
</tr>
</tbody>
</table>

Possible leaks report (possible leaks: 0 total size: 0 bytes)
Checking for memory use...
Blocks in use report (blocks in use: 2 total size: 44 bytes)

| Total | % of Num of Avg Allocation call stack |
|-------|-----------|-----------------------------|
| Size  | All Blocks Size                               |
| ------|----------|-----------------------------|
| 32    | 72%      | 1 32 memory_use < main                  |
| 12    | 27%      | 1 12 memory_use < main                  |

execution completed, exit code is 0
```

The function `access_error()` reads variable `j` before it is initialized. Runtime checking reports this access error as a Read from uninitialized (rui).
The function `memory_leak()` does not free the variable `local` before it returns. When `memory_leak()` returns, this variable goes out of scope and the block allocated at line 20 becomes a leak.

The program uses global variables `hello1` and `hello2`, which are in scope all the time. They both point to dynamically allocated memory, which is reported as Blocks in use (biu).

**Using Access Checking**

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

Programs might incorrectly read or write memory in a variety of ways; these are called memory access errors. For example, the program may reference a block of memory that has been deallocated through a `free()` call for a heap block. Or a function might return a pointer to a local variable, and when that pointer is accessed an error would result. Access errors might result in wild pointers in the program and can cause incorrect program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to track down.

Runtime checking maintains a table that tracks the state of each block of memory being used by the program. Runtime checking checks each memory operation against the state of the block of memory it involves and then determines whether the operation is valid. The possible memory states are:

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

Using runtime checking to find memory access errors is not unlike using a compiler to find syntax errors in your program. In both cases, a list of errors is produced, with each error message giving the cause of the error and the location in the program where the error occurred. In both cases, you should fix the errors in your program starting at the top of the error list and working your way down. One error can cause other errors in a chain reaction. The first error in the chain is, therefore, the “first cause,” and fixing that error might also fix some subsequent errors.

For example, a read from an uninitialized section of memory can create an incorrect pointer, which when dereferenced can cause another invalid read or write, which can in turn lead to yet another error.
# Understanding the Memory Access Error Report

Runtime checking prints the following information for memory access errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Type of error.</td>
</tr>
<tr>
<td>access</td>
<td>Type of access attempted (read or write).</td>
</tr>
<tr>
<td>size</td>
<td>Size of attempted access.</td>
</tr>
<tr>
<td>address</td>
<td>Address of attempted access.</td>
</tr>
<tr>
<td>size</td>
<td>Size of leaked block.</td>
</tr>
<tr>
<td>detail</td>
<td>More detailed information about address.</td>
</tr>
<tr>
<td>stack</td>
<td>Call stack at time of error (with batch mode).</td>
</tr>
<tr>
<td>allocation</td>
<td>If the address is in the heap, then the allocation trace of the nearest heap block is given.</td>
</tr>
<tr>
<td>location</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 148)
- rua (see “Read From Unallocated Memory (rua) Error” on page 148)
- rob (see “Read From Array Out-of-Bounds (rob) Error” on page 148)
- wua (see “Write to Unallocated Memory (wua) Error” on page 149)
- wro (see “Write to Read-Only Memory (wro) Error” on page 148)
- wob (see “Write to Array Out-of-Bounds Memory (wob) Error” on page 148)
Using Memory Leak Checking

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers pointing to the blocks, programs cannot reference them, much less free them. Runtime checking finds and reports such blocks.

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

Typically, memory leaks occur because allocated memory is not freed and you lose a pointer to the allocated block. Here are some examples of memory leaks:

```c
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.

```c
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, */
```
You can avoid memory leaks by always freeing memory when it is no longer needed and paying close attention to library functions that return allocated memory. If you use such functions, remember to free up the memory appropriately.

Sometimes the term memory leak is used to refer to any block that has not been freed. This is a much less useful definition of a memory leak, because it is a common programming practice not to free memory if the program will terminate shortly. Runtime checking does not report a block as a leak, if the program still retains one or more pointers to it.

**Detecting Memory Leak Errors**

Runtime checking detects the following memory leak errors:

- mel (see “Memory Leak (mel) Error” on page 150)
- air (see “Address in Register (air) Error” on page 149)
- aib (see “Address in Block (aib) Error” on page 149)

**Note** – Runtime checking only finds leaks of malloc memory. If your program does not use malloc, runtime checking cannot find memory leaks.

**Possible Leaks**

There are two cases where runtime checking can report a “possible” leak. The first case is when no pointers are found pointing to the beginning of the block, but a pointer is found pointing to the interior of the block. This case is reported as an “Address in Block (aib)” error. If it was a stray pointer that pointed into the block, this would be a real memory leak. However, some programs deliberately move the only pointer to an array back and forth as needed to access its entries. In this case, it would not be a memory leak. Because runtime checking cannot distinguish between these two cases, it reports both of them as possible leaks, letting you determine which are real memory leaks.

The second type of possible leak occurs when no pointers to a block are found in the data space, but a pointer is found in a register. This case is reported as an “Address in Register (air)” error. If the register points to the block accidentally, or if it is an old copy of a memory pointer that has since been lost, then this is a real leak. However, the compiler can optimize references and place the only pointer to a block in a register without ever writing the pointer to memory. Such a case would not be a real leak. Hence, if the program has been optimized and the report was the result of the showleaks command, it is likely not to be a real leak. In all other cases, it is likely to be a real leak. For more information, see “showleaks Command” on page 344.
Note – Runtime leak checking requires the use of the standard \texttt{libc malloc/free/realloc} functions or allocators based on those functions. For other allocators, see “Runtime Checking Application Programming Interface” on page 140.

Checking for Leaks

If memory leak checking is turned on, a scan for memory leaks is automatically performed just before the program being tested exits. Any detected leaks are reported. The program should not be killed with the \texttt{kill} command. Here is a typical memory leak error message:

\texttt{Memory leak (mel):}
\texttt{Found leaked block of size 6 at address \texttt{0x21718}}
\texttt{At time of allocation, the call stack was:}
\begin{itemize}
  \item \texttt{[1] foo()} at line 63 in test.c}
  \item \texttt{[2] main()} at line 47 in test.c
\end{itemize}

A UNIX program has a \texttt{main} procedure (called \texttt{MAIN} in \texttt{f77}) that is the top-level user function for the program. Normally, a program terminates either by calling \texttt{exit(3)} or by returning from \texttt{main}. In the latter case, all variables local to \texttt{main} go out of scope after the return, and any heap blocks they pointed to are reported as leaks (unless global variables point to those same blocks).

It is a common programming practice not to free heap blocks allocated to local variables in \texttt{main}, because the program is about to terminate and return from \texttt{main} without calling \texttt{exit()}. To prevent runtime checking from reporting such blocks as memory leaks, stop the program just before \texttt{main} returns by setting a breakpoint on the last executable source line in \texttt{main}. When the program halts there, use the \texttt{showleaks} command to report all the true leaks, omitting the leaks that would result merely from variables in \texttt{main} going out of scope.

For more information, see “\texttt{showleaks Command}” on page 344.

Understanding the Memory Leak Report

With leak checking turned on, you receive an automatic leak report when the program exits. All possible leaks are reported, provided the program has not been killed using the \texttt{kill} command. The level of detail in the report is controlled by the \texttt{dbx} environment variable \texttt{rtc_mel_at_exit} (see “Setting \texttt{dbx} Environment Variables” on page 54). By default, a nonverbose leak report is generated.

Reports are sorted according to the combined size of the leaks. Actual memory leaks are reported first, followed by possible leaks. The verbose report contains detailed stack trace information, including line numbers and source files whenever they are available.

Both reports include the following information for memory leak errors:
Here is the corresponding nonverbose memory leak report.

### Actual leaks report

- **Total** leaked allocations: 3
- **Total size:** 2427 bytes

<table>
<thead>
<tr>
<th>Size</th>
<th>Blocks</th>
<th>Block</th>
<th>Address</th>
<th>Allocation call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1852</td>
<td>2</td>
<td>-</td>
<td>true_leak</td>
<td>true_leak &lt; true_leak</td>
</tr>
<tr>
<td>575</td>
<td>1</td>
<td>0x22150</td>
<td>true_leak &lt; main</td>
<td></td>
</tr>
</tbody>
</table>

### Possible leaks report

- **Possible leaks:** 1
- **Total size:** 8 bytes

<table>
<thead>
<tr>
<th>Size</th>
<th>Blocks</th>
<th>Block</th>
<th>Address</th>
<th>Allocation call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>0x219b0</td>
<td>in_block</td>
<td>in_block &lt; main</td>
</tr>
</tbody>
</table>

Following is a typical verbose leak report.

### Actual leaks report

- **Actual leaks:** 3
- **Total size:** 2427 bytes

#### Memory Leak (mel): Found 2 leaked blocks with total size 1852 bytes
At time of each allocation, the call stack was:

1. `true_leak()` at line 220 in "leaks.c"
2. `true_leak()` at line 224 in "leaks.c"

#### Memory Leak (mel): Found leaked block of size 575 bytes at address 0x22150
At time of allocation, the call stack was:

1. `true_leak()` at line 220 in "leaks.c"
2. `main()` at line 87 in "leaks.c"

### Possible leaks report

- **Possible leaks:** 1
- **Total size:** 8 bytes

#### Possible memory leak -- address in block (aib):
Found leaked block of size 8 bytes at address 0x219b0
At time of allocation, the call stack was:

1. `in_block()` at line 177 in "leaks.c"
2. `main()` at line 100 in "leaks.c"
Generating a Leak Report

You can ask for a leak report at any time using the `showleaks` command, which reports new memory leaks since the last `showleaks` command. For more information, see "showleaks Command" on page 344.

CombiningLeaks

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:

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3] c() at 0x30000</td>
<td>[3] c() at 0x30000</td>
<td>[3] c() at 0x31000</td>
</tr>
<tr>
<td>[5] a() at 0x50000</td>
<td>[5] a() at 0x50000</td>
<td>[5] a() at 0x50000</td>
</tr>
</tbody>
</table>

If all of these blocks lead to memory leaks, the value of `m` determines whether the leaks are reported as separate leaks or as one repeated leak. If `m` is 2, Blocks 1 and 2 are reported as one repeated leak because the 2 stack frames above `malloc()` are common to both call sequences. Block 3 will be reported as a separate leak because the trace for `c()` does not match the other blocks. For `m` greater than 2, runtime checking reports all leaks as separate leaks. (The `malloc` is not shown on the leak report.)

In general, the smaller the value of `m`, the fewer individual leak reports and the more combined leak reports are generated. The greater the value of `m`, the fewer combined leak reports and the more individual leak reports are generated.

Fixing Memory Leaks

Once you have obtained a memory leak report, follow these guidelines for fixing the memory leaks.
Most importantly, determine where the leak is. The leak report tells you the allocation trace of the leaked block, the place where the leaked block was allocated.

You can then look at the execution flow of your program and see how the block was used. If it is obvious where the pointer was lost, the job is easy; otherwise you can use showleaks to narrow your leak window. By default the showleaks command gives you the new leaks created only since the last showleaks command. You can run showleaks repeatedly while stepping through your program to narrow the window where the block was leaked.

For more information, see “showleaks Command” on page 344.

Using Memory Use Checking

Memory use checking lets you see all the heap memory in use. You can use this information to get a sense of where memory is allocated in your program or which program sections are using the most dynamic memory. This information can also be useful in reducing the dynamic memory consumption of your program and might help in performance tuning

Memory use checking is useful during performance tuning or to control virtual memory use. When the program exits, a memory use report can be generated. Memory usage information can also be obtained at any time during program execution with the showmemuse command, which causes memory usage to be displayed. For information, see “showmemuse Command” on page 345.

Turning on memory use checking also turns on leak checking. In addition to a leak report at the program exit, you also get a blocks in use (biu) report. By default, a nonverbose blocks in use report is generated at program exit. The level of detail in the memory use report is controlled by the dbx environment variable rtc_biu_at_exit (see “Setting dbx Environment Variables” on page 54).

The following is a typical nonverbose memory use report.

```
Blocks in use report (blocks in use: 5 total size: 40 bytes)

<table>
<thead>
<tr>
<th>Total Size</th>
<th>% of All Blocks</th>
<th>Avg Size</th>
<th>Allocation call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>40%</td>
<td>2</td>
<td>nonleak &lt; nonleak</td>
</tr>
<tr>
<td>8</td>
<td>20%</td>
<td>1</td>
<td>nonleak &lt; main</td>
</tr>
<tr>
<td>8</td>
<td>20%</td>
<td>1</td>
<td>cyclic_leaks &lt; main</td>
</tr>
</tbody>
</table>

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 provides a powerful error suppression facility that allows great flexibility in limiting the number and types of errors reported. If an error occurs that you have suppressed, then no report is given, and the program continues as if no error had occurred.

You can suppress errors using the suppress command (see “suppress Command” on page 354).

You can undo error suppression using the unsuppress command (see “unsuppress Command” on page 367).

Suppression is persistent across run commands within the same debug session, but not across debug commands.

Types of Suppression

The following types of suppression are available:

Suppression by Scope and Type

You must specify which type of error to suppress. You can specify which parts of the program to suppress. The options are:
Suppression of Last Error

By default, runtime checking suppresses the most recent error to prevent repeated reports of the same error. This is controlled by the `dbx` environment variable `rtc_auto_suppress`. When `rtc_auto_suppress` is set to on (the default), a particular access error at a particular location is reported only the first time it is encountered and suppressed thereafter. This is useful, for example, for preventing multiple copies of the same error report when an error occurs in a loop that is executed many times.

Limiting the Number of Errors Reported

You can use the `dbx` environment variable `rtc_error_limit` to limit the number of errors that will be reported. The error limit is used separately for access errors and leak errors. For example, if the error limit is set to 5, then a maximum of five access errors and five memory leaks are shown in both the leak report at the end of the run and for each `showleaks` command you issue. The default is 1000.

Suppressing Error Examples

In the following examples, `main.cc` is a filename, `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 mem 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 354.

DefaultSuppressions

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 367.

Using Suppression to Manage Errors

For the initial run on a large program, the large number of errors might be overwhelming. It might be better to take a phased approach. You can do so using the suppress command to reduce the reported errors to a manageable number, fixing just those errors, and repeating the cycle; suppressing fewer and fewer errors with each iteration.

For example, you could focus on a few error types at one time. The most common error types typically encountered are rui, rua, and wua, usually in that order. rui errors are less serious (although they can cause more serious errors to happen later). Often a program might still work correctly with these errors. rua and wua errors are more serious because they are accesses to or from invalid memory addresses and always indicate a coding error.

You can start by suppressing rui and rua errors. After fixing all the wua errors that occur, run the program again, this time suppressing only rui errors. After fixing all the rua errors that occur, run the program again, this time with no errors suppressed. Fix all the rui errors. Lastly, run the program a final time to ensure no errors are left.
If you want to suppress the last reported error, use suppress -last.

**Using Runtime Checking on a Child Process**

To use runtime checking on a child process, you must have the `dbx` environment variable `rtc_inherit` set to on. By default, it is set to off. (See “Setting dbx Environment Variables” on page 54.)

`dbx` supports runtime checking of a child process if runtime checking is enabled for the parent and the `dbx` environment variable `follow_fork_mode` is set to `child` (see “Setting dbx Environment Variables” on page 54).

When a fork happens, `dbx` automatically performs runtime checking on the child. If the program calls `exec()`, the runtime checking settings of the program calling `exec()` are passed on to the program.

At any given time, only one process can be under runtime checking control. The following is an example.

```
% cat -n program1.c
 1 #include <sys/types.h>
 2 #include <unistd.h>
 3 #include <stdio.h>
 4
 5 int
 6 main()
 7 {
 8     pid_t child_pid;
 9     int parent_i, parent_j;
10    
11     parent_i = parent_j;
12    
13     child_pid = fork();
14    
15     if (child_pid == -1) {
16         printf("parent: Fork failed\n");
17         return 1;
18     } else if (child_pid == 0) {
19         int child_i, child_j;
20        
21         printf("child: In child\n");
22         child_i = child_j;
23         if (execl("./program2", NULL) == -1) {
24             printf("child: exec of program2 failed\n");
25             exit(1);
26         }
27     } else {
28         printf("parent: child's pid = \d\n", child_pid);
29     }
30     return 0;
31 }
```
Using Runtime Checking on a Child Process

```sh
% cat -n program2.c
 1 #include <stdio.h>
 2 main()
 3 {
 4     int program2_i, program2_j;
 5     printf ("program2: pid = %d\n", getpid());
 6     program2_i = program2_j;
 7     malloc(8);
 8     return 0;
 9 }
% 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 0xfffff110
   which is 104 bytes above the current stack pointer
Variable is 'parent_j'
Current function is main
   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
   child_pid = fork();
parent: child's pid = 3886
(dbx) cont
child: In child
Read from uninitialized (rui):
   Attempting to read 4 bytes at address 0xfffff108
   which is 96 bytes above the current stack pointer
```

Oracle Solaris Studio 12.3: Debugging a Program With dbx • December 2011
RTC reports an error in the child
Variable is 'child_j'
Current function is main
22 child_i = child_j;

(dbx) cont

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

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)

<table>
<thead>
<tr>
<th>Size</th>
<th>Num of Leaked Allocation call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

On a System Running Solaris

On a system running the 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:
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:

installation_directory/lib/dbx/amd64/runtime/librtc.so on 64-bit AMD64 platforms

installation_directory/lib/dbx/runtime/librtc.so on 32-bit platforms

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 fix and continue to isolate and fix programming errors rapidly. Fix and continue provides a powerful combination that can save you a lot of debugging time. Here is an example:

```c
#include stdio.h
char *s = NULL;

void problem()
{
    *s = 'c';
}

main()
{
    problem();
    return 0;
}
```

```c
#include stdio.h
char *s = NULL;

void problem()
{
    s = (char *)malloc(1);
    *s = 'c';
}
```
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 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
      "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 on using fix and continue, see "Memory Leak (me1) Error" on page 150.

Runtime Checking Application Programming Interface

Both leak detection and access checking require that the standard heap management routines in
the shared library libc.so be used so that runtime checking can keep track of all the allocations
and deallocations in the program. Many applications write their own memory management
routines either on top of the malloc() or free() function or stand-alone. When you use your
own allocators (referred to as private allocators), runtime checking cannot automatically track
them; thus you do not learn of leak and memory access errors resulting from their improper
use.
However, runtime checking provides an API for the use of private allocators. This API allows the private allocators the same treatment as the standard heap allocators. The API itself is provided in the header file `rtc_api.h` and is distributed as a part of Oracle Solaris Studio software. The man page `rtc_api(3x)` details the runtime checking API entry points.

Some minor differences might exist with runtime checking access error reporting when private allocators do not use the program heap. 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:

```
  [-s script] program [args]
```

Use the `-o logfile` option to specify a different name for the logfile. Use the `-s script` option before executing the program to read in the `dbx` commands contained in the file `script`. The `script` file typically contains commands like `suppress` and `dbxenv` to tailor the error output of the `bcheck` utility.

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

To perform only leak checking on hello, type:

bcheck hello

To perform only access checking on mach with the argument 5, type:

bcheck -access mach 5

To perform memory use checking on cc quietly and exit with normal exit status, type:

bcheck -memuse -q cc -c prog.c

The program does not stop when runtime errors are detected in batch mode. All error output is redirected to your error log file logfile. The program stops when breakpoints are encountered or if the program is interrupted.

In batch mode, the complete stack backtrace is generated and redirected to the error log file. The number of stack frames can be controlled using the dbx environment variable stack_max_size.

If the file logfile already exists, bcheck erases the contents of that file before it redirects the batch output to it.

Enabling Batch Mode Directly From dbx

You can also enable a batch-like mode directly from dbx by setting the dbx environment variables rtc_auto_continue and rtc_error_log_file_name (see “Setting dbx Environment Variables” on page 54).

If rtc_auto_continue is set to on, runtime checking continues to find errors and keeps running automatically. It redirects errors to the file named by the dbx environment variable rtc_error_log_file_name. (See “Setting dbx Environment Variables” on page 54.) The default log file name is /tmp/dbx.errlog.uniqueid. To redirect all errors to the terminal, set the rtc_error_log_file_name environment variable to /dev/tty.

By default, rtc_auto_continue is set to off.
Troubleshooting Tips

After error checking has been enabled for a program and the program is run, one of the following errors may be detected:

librtc.so and dbx version mismatch; Error checking disabled

This error can occur if you are using runtime checking on an attached process and have set LD_AUDIT to a version of rtcaudit.so other than the one shipped with your Oracle Solaris 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” next.

Runtime Checking Limitations

Runtime checking has the following limitations.

Works Better 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 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.
Works Better When Sufficient Patch Area is Available Within 8 MB of All Existing Code (SPARC platforms only).

Two problems might arise if 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 up all the address space within 8 megabytes of the particular load or store instruction being replaced, no place exists to put the patch area. In this case, `dbx` invokes a trap handler instead of using a branch. The transfer of control to a trap handler is significantly (up to 10 times) slower, 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 utilizes the higher 32 bits of out registers, the process might run incorrectly when access checking is turned on.

  The compilers use the V8+ architecture by default when creating 32-bit SPARC based binaries, but you can tell them 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 to not work correctly when instrumented with traps:
  - `server/libjvm.so`
  - `client/libjvm.so`
  - `libfisu_isa.so`'__f_cvt_real
  - `libfisu_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 turn on access checking, it is likely that the trap handler limitation affects your program. To work around the limitation, you can do the following:
- Use the rtc skippatch command (see “rtc skippatch Command” on page 341) to skip instrumentation of the code in your program that uses the functions and libraries listed above. Generally it is difficult to track the problem to a specific function, 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. See the rtc_patch_area(1) man page.

  When dbx reaches the 8 megabyte limit, it tells you which load object was too large (the main program, or a shared library) and it prints out the total patch space needed for that load object.

  For the best results, the special patch object files should be evenly spaced throughout the executable or shared library, and the default size (8 megabytes) or smaller should be used. Also, do not add more than 10-20% more patch space than dbx says it requires. For example, if dbx says that it needs 31 megabytes for a.out, then add four object files created with the rtc_patch_area script, each one 8 megabytes in size, and space them approximately evenly throughout the executable.

  When dbx finds explicit patch areas in an executable, it prints the address ranges spanned by the patch areas, which can help you to place them correctly on the link line.

- Try dividing the large load object into smaller load objects.
  Split up the object files in your executable or your large library into smaller groups of object files. Then link them into smaller parts. If the large file is the executable, then split it up into a smaller executable and a series of shared libraries. If the large file is a shared library, then rearrange it into a set of smaller libraries.

  This technique allows dbx to find space for patch code in between the different shared objects.

- 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 up the executable with runtime checking turned on, it asks the runtime linker to place an extra gap between the shared libraries, but when attaching to a process that was not started by dbx with runtime checking enabled, the libraries might be too close together.
If the runtime libraries are too close together, (and if it is not possible to start the program using dbx) then you can try creating a shared library using the rtc_patch_area script and linking it into your program between the other shared libraries. See the rtc_patch_area(1) man page for more details.

**Runtime Checking Errors**

Errors reported by runtime checking generally fall in two categories. Access errors and leaks.

**Access Errors**

When access checking is turned on, runtime checking detects and reports the following types of errors.

**Bad Free (baf) Error**

Problem: Attempt to free memory that has never been allocated.

Possible causes: Passing a non-heap data pointer to free() or realloc().

Example:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
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:

```c
char *cp = malloc (10);
fclass (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:

```c
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, overflowing the bounds of a heap block
Example:

```c
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:

```c
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:

```c
char *cp = malloc (10);
free (cp);
cp[0] = 0;
```

**Memory Leak Errors**

With leak checking turned on, runtime checking reports the following types of errors.

**Address in Block (aib) Error**

Problem: A possible memory leak. There is no reference to the start of an allocated block, but there is at least one reference to an address within the block.

Possible causes: The only pointer to the start of the block is incremented.

Example:

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

**Address in Register (air) Error**

Problem: A possible memory leak. An allocated block has not been freed, and no reference to the block exists anywhere in program memory, but a reference exists in a register.
Possible causes: This can occur legitimately if the compiler keeps a program variable only in a register instead of in memory. The compiler often does this for local variables and function parameters when optimization is turned on. If this error occurs when optimization has not been turned on, it is likely to be an actual memory leak. This can occur if the only pointer to an allocated block goes out of scope before the block is freed.

Example:

```c
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:

```c
char *ptr;

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

Using the `fix` command lets you recompile edited native source code quickly without stopping the debugging process. You cannot use the `fix` command to recompile Java code.

Note – The `fix` command is not available on Linux platforms.

This chapter is organized into the following sections:

- “Using Fix and Continue” on page 151
- “Fixing Your Program” on page 153
- “Changing Variables After Fixing” on page 155
- “Modifying a Header File” on page 156
- “Fixing C++ Template Definitions” on page 156

Using Fix and Continue

The fix and continue feature lets you modify and recompile a native source file and continue executing without rebuilding the entire program. By updating the `.o` files and splicing them into your program, you don’t need to relink.

The advantages of using fix and continue are:

- You do not have to relink the program.
- You do not have to reload the program for debugging.
- You can resume running the program from the fix location.

Note – Do not use the `fix` command if a build is in progress.
How Fix and Continue Operates

Before using the `fix` command you must edit the source. (See “Modifying Source Using Fix and Continue” on page 152 for the ways you can modify your code). After saving changes, type `fix`. For information on the `fix` command, see “fix Command” on page 307.

Once you have invoked the `fix` command, dbx calls the compiler with the appropriate compiler options. The modified files are compiled and shared object (.so) files are created. Semantic tests are done by comparing the old and new files.

The new object file is linked to your running process using the runtime linker. If the function on top of the stack is being fixed, the new stopped in function is the beginning of the same line in the new function. All the breakpoints in the old file are moved to the new file.

You can use fix and continue on files that have been compiled with or without debugging information, but there are some limitations in the functionality of the `fix` command and the `cont` command for files originally compiled without debugging information. See the `-g` option description in “fix Command” on page 307 for more information.

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

The pre-compiled headers feature of the Oracle Solaris Studio C and C++ compilers requires that the compiler options be the same when recompiling. Because the `fix` command changes the compiler options slightly, do not use the `fix` command on object files that were created using precompiled headers.

Modifying Source Using Fix and Continue

You can modify source code in the following ways when using fix and continue:

- Add, delete, or change lines of code in functions
- Add or delete functions
- Add or delete global and static variables

Problems can occur when functions are mapped from the old file to the new file. To minimize such problems when editing a source file:

- Do not change the name of a function.
- Do not add, delete, or change the type of arguments to a function.
- Do not add, delete, or change the type of local variables in functions currently active on the stack.
- Do not make changes to the declaration of a template or to template instances. Only the body of a C++ template function definition can be modified.
If you make any of the above changes, rebuild your entire program rather than using fix and continue.

Fixing Your Program

You can use the `fix` command to relink source files after you make changes, without recompiling the entire program. You can then continue execution of the program.

▼ To Fix Your File

1. Save the changes to your source.
2. Type `fix` at the dbx prompt.

Although you can do an unlimited number of fixes, if you have done several fixes in a row, consider rebuilding your program. The `fix` command changes the program image in memory, but not on the disk. As you do more fixes, the memory image gets out of sync with what is on the disk.

The `fix` command does not make the changes within your executable file, but only changes the `.o` files and the memory image. Once you have finished debugging a program, you must rebuild your program to merge the changes into the executable. When you quit debugging, a message reminds you to rebuild your program.

If you invoke the `fix` command with an option other than `-a` and without a file name argument, only the current modified source file is fixed.

When `fix` is invoked, the current working directory of the file that was current at the time of compilation is searched before executing the compilation line. There might be problems locating the correct directory due to a change in the file system structure from compilation time to debugging time. To avoid this problem, use the command `pathmap`, which creates a mapping from one path name to another. Mapping is applied to source paths and object file paths.

Continuing After Fixing

You can continue executing using the `cont` command (see “cont Command” on page 292).

Before resuming program execution, be aware of the following conditions that determine the effect of your changes.
Changing an Executed Function

If you made changes in a function that has already executed, the changes have no effect until:

- You run the program again
- That function is called the next time

If your modifications involve more than simple changes to variables, use the `fix` command, then the `run` command. Using the `run` command is faster because it does not relink the program.

Changing a Function Not Yet Called

If you have made changes in a function not yet called, the changes will be in effect when that function is called.

Changing a Function Currently Being Executed

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

- If the change is in code that has already been executed, the code is not re-executed. Execute the code by popping the current function off the stack (see “pop Command” on page 333) and continuing from where the changed function is called. You need to know your code well enough to determine whether the function has side effects that can’t be undone (for example, opening a file).
- If the change is in code that is yet to be executed, the new code is run.

Changing a Function Presently on the Stack

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

There are several ways to solve this problem:

- Use the `pop` command to pop the stack until all changed functions are removed from the stack. You need to know your code to be sure that no problems are created.
- Use the `cont at line_number` command to continue from another line.
- Manually repair data structures (use the `assign` command) before continuing.
- Rerun the program using the `run` command.

If there are breakpoints in modified functions on the stack, the breakpoints are moved to the new versions of the functions. If the old versions are executed, the program does not stop in those functions.
Changing Variables After Fixing

Changes made to global variables are not undone by the `pop` command or the `fix` command. To reassign correct values to global variables manually, use the `assign` command. (See “assign Command” on page 277.)

The following example shows how a simple bug can be fixed. The application gets a segmentation violation in line 6 when trying to dereference a NULL pointer.

```
#include <stdio.h>

char *from = "ships";

void copy(char *to)
{
    while ((to++ = *from++) != '\0');
    *to = '\0';
}

main()
{
    char buf[100];
    copy(0);
    printf("%s\n", buf);
    return 0;
}
```

If the program is continued from here, it still gets a segmentation fault because the zero-pointer is still pushed on the stack. Use the `pop` command to pop one frame of the stack:

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

If the program is continued from here, it runs, but does not print the correct value because the global variable `from` has already been incremented by one. The program would print `hips` and not `ships`. Use the `assign` command to restore the global variable and then use the `cont` command. Now the program prints the correct string:

```
(dbx) assign
assigning "testfix.cc" ......
```

```
(dbx) cont
Restarting testfix ...
```

```
(dbx) run
Running: testfix
(process id 4842)
```
Modifying a Header File

Sometimes it may be necessary to modify a header (.h) file as well as a source file. To be sure that the modified header file is accessed by all source files in the program that include it, you must give as an argument to the fix command a list of all the source files that include that header file. If you do not include the list of source files, only the primary (current) source file is recompiled and only it includes the modified version of the header file. Other source files in the program continue to include the original version of that header file.

Fixing C++ Template Definitions

C++ template definitions cannot be fixed directly. Fix the files with the template instances instead. You can use the -f option to override the date-checking if the template definition file has not changed.
dbx can debug multithreaded applications that use either Solaris threads or POSIX threads. With dbx, you can examine stack traces of each thread, resume all threads, step or next a specific thread, and navigate between threads.

dbx recognizes a multithreaded program by detecting whether it utilizes `libthread.so`. The program uses `libthread.so` either by explicitly being compiled with `-lthread` or `-mt`, or implicitly by being compiled with `-lpthread`.

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

This chapter is organized into the following sections:

- “Understanding Multithreaded Debugging” on page 157
- “Understanding Thread Creation Activity” on page 161
- “Understanding LWP Information” on page 162

### Understanding Multithreaded Debugging

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

dbx is synchronous; when any thread or lightweight process (LWP) stops, all other threads and LWPs sympathetically stop. This behavior is sometimes referred to as the “stop the world” model.

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

The following thread information is available in dbx:

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

For native code, each line of information is composed of the following:

- The * (asterisk) indicates that an event requiring user attention has occurred in this thread. Usually this is a breakpoint. An ‘o’ instead of an asterisk indicates that a dbx internal event has occurred.
- The > (arrow) denotes the current thread.
- t@number, the thread id, refers to a particular thread. The number is the thread_t value passed back by thr_create.
- b l@number or b l@number means the thread is bound to or active on the designated LWP, meaning the thread is actually runnable by the operating system.
- The “Start function” of the thread as passed to thr_create. A ?() means that the start function is not known.
- The thread state (see Table 11–1 for descriptions of the thread states).
- The function that the thread is currently executing.

For Java code, each line of information is composed of the following:

- t@number, a dbx-style thread ID
- The thread state (See Table 11–1 for descriptions of the thread states.)
- The thread name in single quotation marks
- A number indicating the thread priority

### TABLE 11–1   Thread and LWP States

<table>
<thead>
<tr>
<th>Thread and LWP States</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>suspended</td>
<td>The thread has been explicitly suspended.</td>
</tr>
<tr>
<td>runnable</td>
<td>The thread is runnable and is waiting for an LWP as a computational resource.</td>
</tr>
<tr>
<td>zombie</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 11-1 Thread and LWP States (Continued)

<table>
<thead>
<tr>
<th>Thread and LWP States</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asleep on syncobj</td>
<td>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.</td>
</tr>
<tr>
<td>active</td>
<td>The thread is active on an LWP, but dbx cannot access the LWP.</td>
</tr>
<tr>
<td>unknown</td>
<td>dbx cannot determine the state.</td>
</tr>
<tr>
<td>lwpstate</td>
<td>A bound or active thread state has the state of the LWP associated with it.</td>
</tr>
<tr>
<td>running</td>
<td>LWP was running but was stopped in synchrony with some other LWP.</td>
</tr>
<tr>
<td>syscall num</td>
<td>LWP stopped on an entry into the given system call.</td>
</tr>
<tr>
<td>syscall return num</td>
<td>LWP stopped on an exit from the given system call.</td>
</tr>
<tr>
<td>job control</td>
<td>LWP stopped due to job control.</td>
</tr>
<tr>
<td>LWP suspended</td>
<td>LWP is blocked in the kernel.</td>
</tr>
<tr>
<td>single stepped</td>
<td>LWP has just completed a single step.</td>
</tr>
<tr>
<td>breakpoint</td>
<td>LWP has just hit a breakpoint.</td>
</tr>
<tr>
<td>fault num</td>
<td>LWP has incurred the given fault.</td>
</tr>
<tr>
<td>signal name</td>
<td>LWP has incurred the given signal.</td>
</tr>
<tr>
<td>process sync</td>
<td>The process to which this LWP belongs has just started executing.</td>
</tr>
<tr>
<td>LWP death</td>
<td>LWP is in the process of exiting.</td>
</tr>
</tbody>
</table>

### Viewing the Context of Another Thread

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

```
```

To display the current thread, type:

```
thread
```

To switch to thread `thread_id`, type:

```
thread thread_id
```

For more information on the `thread` command, see “thread Command” on page 357.
**Viewing the Threads List**

To view the threads list, use the `threads` command. The syntax is:

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

To print the list of all known threads, type:

```
threads
```

To print threads normally not printed (zombies), type:

```
threads -all
```

For an explanation of the threads list, see “Thread Information” on page 158.

For more information on the `threads` command, see “threads Command” on page 358.

**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 (see “call Command” on page 279).

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 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:

- 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 interesting information, you could try the following 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 multi-threading 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);
```

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. There are times, however, when thread level queries cannot be completed. In these cases, use the `lwps` command to show information about LWPs.

```
(dbx) lwps
l@1 running in main()
l@2 running in sigwait()
l@3 running in _lwp_sema_wait()
* l@4 breakpoint in Queue_dequeue()
l@5 running in _thread_start()
```

Each line of the LWP list contains the following:

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

Use the “lwp Command” on page 323 to list or change the current LWP.
Debugging Child Processes

This chapter describes how to debug a child process. `dbx` has several facilities to help you debug processes that create children using the `fork` and `exec` functions.

This chapter is organized into the following sections:

- "Attaching to Child Processes" on page 163
- "Following the exec Function" on page 164
- "Following the fork Function" on page 164
- "Interacting With Events" on page 164

**Attaching to Child Processes**

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

- When starting `dbx`:
  
  ```
  $ dbx program_name process_id
  ```

- From the `dbx` command line:
  
  ```
  (dbx) debug program_name process_id
  ```

You can substitute `program_name` with the name - (minus sign), so that `dbx` automatically finds the executable associated with the given process ID (`process_id`). After using a -, a subsequent run command or rerun command does not work because `dbx` does not know the full path name of the executable.

You can also attach to a running child process in the Oracle Solaris Studio IDE (see "Attaching the Debugger to a Running Process" in the IDE online help) or `dbx` tool (see "Attaching dbxtool to a Running Process" in the `dbx` tool online help).
Following the `exec` Function

If a child process executes a new program using the `exec(2)` function or one of its variations, the process id does not change, but the process image does. `dbx` automatically takes note of a call to the `exec()` function and does an implicit reload of the newly executed program.

The original name of the executable is saved in `$oprog`. To return to it, use `debug $oprog`.

Following the `fork` Function

If a child process calls the `vfork(2)`, `fork1(2)`, or `fork(2)` function, the process id changes, but the process image stays the same. Depending on how the `dbx` environment variable `follow_fork_mode` is set, `dbx` does one of the following.

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

Child   `dbx` automatically switches to the forked child using the new process ID. All connection to and awareness of the original parent is lost.

Both    This mode is available only when using `dbx` through the Oracle Solaris 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 Solaris 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 `dbx` environment variable `follow_fork_inherit` to on, or make the events permanent using the `-perm eventspec` modifier. For more information on using event specification modifiers, see “cont at Command” on page 248.
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 multi-threaded debugging features of dbx. All of the dbx commands that operate on threads and LWPs can be used for OpenMP debugging. dbx does not support asynchronous thread control in OpenMP debugging.

This chapter is organized into the following sections:

- “How Compilers Transform OpenMP Code” on page 165
- “dbx Functionality Available for OpenMP Code” on page 166
- “Execution Sequence of OpenMP Code” on page 173

See the OpenMP API Users Guide for information on the directives, run-time library routines, and environment variables comprising the OpenMP Version 2.0 Application Program Interfaces, as implemented by the Oracle Solaris Studio Fortran 95 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:

```fortran
1  program example
2     integer i, n
3     parameter (n = 1000000)
4     real sum, a(n)
5
6    do i = 1, n
7       a(i) = i*i
8    end do
9
10   sum = 0
11```

The OpenMP directives in the above example are:

1. `!$omp parallel`
2. `!$omp do i = 1, n`
3. `!$omp end do`

These directives are translated by the compiler into the corresponding Fortran statements to create a parallel loop. The above example is transformed into:

```fortran
1  program example
2     integer i, n
3     parameter (n = 1000000)
4     real sum, a(n)
5
6    do i = 1, n
7      a(i) = i*i
8    end do
9
10   sum = 0
11```

The OpenMP directives for parallelism, work-sharing, and master/worker constructs are supported by dbx for OpenMP debugging.
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.

**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 165, and assume that master thread t@1 is at line 10. You single step into line 12, and slave threads t@2, t@3, and t@4 are created to execute the runtime library calls. Thread t@3 reaches the breakpoint first and causes the program execution to stop. So the single step that was initiated by thread t@1 ends on thread t@3. This behavior is different from normal stepping in which you are usually on the same thread after the single step as before.
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 whether or not they are within a parallel construction. For example:

```
(dbx) whatis p_a
# OpenMP first and last private variable
int p_a;
```

The print `-s expression` command prints the value of 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

`dbx` can 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
```

It 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>
```

And it can 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 329.

dbx can 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>

It 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
  source location = test.c:103
  unfinished children = 0
  parent = <no parent>

And it 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>
For more information, see "\texttt{omp_tr Command}" on page 331.

\texttt{dbx} can 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:

\begin{verbatim}
(dbx) \texttt{omp_loop}
ordered loop: no
  lower bound: 0
  upper bound: 3
  step: 1
  chunk: 1
  schedule type: static
  source location: test.c:49
\end{verbatim}

For more information, see "\texttt{omp_loop Command}" on page 329.

\texttt{dbx} can print all the threads on the current team or the team of a specified parallel region. For example:

\begin{verbatim}
(dbx) \texttt{omp_team}
team members:
  0: t@0 state = in implicit barrier, task region = 262145
  1: t@0 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
\end{verbatim}
For more information, see "\texttt{omp\_team Command}" on page 330.

When you are debugging OpenMP code, the \texttt{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 "\texttt{thread Command}" on page 357.

\section*{Serializing the Execution of a Parallel Region}

dbx can 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 "\texttt{omp\_serialize Command}" on page 330.

\section*{Using Stack Traces}

When execution is stopped in parallel region, a \texttt{where} command shows a stack trace that contains the outlined subroutine.

\begin{verbatim}
(dbx) where
  current thread: t@4
  =>[1] $d1E48.main(), line 52 in "test.c"
  [2] $sp1E46.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"
\end{verbatim}

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 \texttt{where} command from a slave thread prints the master thread's stack trace if the relevant frames are still active. A \texttt{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 \texttt{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.

\section*{Using the \texttt{dump} Command}

When execution is stopped in a parallel region, a \texttt{dump} command may print more than one copy of private variables. In the following example, the \texttt{dump} command prints two copies of the variable \(i\):
Two copies of variable \( i \) are printed because the outlined routine is implemented as a nested function of the hosting routine, and private variables are implemented as local variables of the outlined routine. Since a `dump` command prints all the variables in scope, both the \( i \) in hosting routine and the \( i \) in the outlined routine are displayed.

**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 252.

**Synchronization Events**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_barrier</code></td>
<td>Tracks the event of a thread entering a barrier.</td>
</tr>
<tr>
<td><code>omp_taskwait</code></td>
<td>Tracks the event of a thread entering a taskwait.</td>
</tr>
</tbody>
</table>

**`omp_barrier`**

- **type**: `explicit`, meaning track explicit barriers
- **state**: `enter`, meaning report the event when any thread enters a barrier
- `exit`, meaning report the event when any thread exits a barrier
- `all_entered`, meaning report the event when all threads have entered a barrier

If you do not specify `type`, then only explicit barriers are tracked.

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**: `enter`, meaning report the event when a thread enters a taskwait

...
exit, meaning 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* can be:
- begin, meaning report the event when an ordered region begins
- enter, meaning report the event when a thread enters an ordered region
- exit, meaning 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* can be:
- begin, meaning report the event when an atomic region begins
- exit, meaning 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* can be:
- create, meaning report the event when a task has just been created and before its execution begins
- start, meaning report the event when a task starts its execution
- finish, meaning 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 of a parallel region in an OpenMP program, the execution sequence may not be the same as the source code sequence. This difference in sequence occurs because the code in the parallel region is usually transformed and rearranged by the compiler. Single stepping in OpenMP code is similar to single stepping in optimized code where the optimizer has usually moved code around.
This chapter describes how to use dbx to work with signals. dbx supports the catch command, which instructs dbx to stop a program when dbx detects any of the signals appearing on the catch list.

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

This chapter is organized into the following sections.

- “Understanding Signal Events” on page 175
- “Catching Signals” on page 176
- “Sending a Signal to a Program” on page 179
- “Automatically Handling Signals” on page 179

### Understanding Signal Events

When a signal is to be delivered to a process that is being debugged, the signal is redirected to dbx by the kernel. When this happens, you usually receive a prompt. You then have two choices:

- “Cancel” the signal when the program is resumed (the default behavior of the cont command) facilitating easy interruption and resumption with SIGINT (Control-C) as shown in Figure 14–1.
- “Forward” the signal to the process using:

  ```
  cont -sig signal
  ```

  *signal* can be either a signal name or a signal number.
In addition, if a certain signal is received frequently, you can arrange for dbx to forward the signal automatically because you do not want it displayed:

\texttt{ignore signal \# “ignore”}

However, the signal is still forwarded to the process. A default set of signals is automatically forwarded in this manner (see “\texttt{ignore Command}” on page 312).

\section*{Catching Signals}

By default, the catch list contains many of the more than 33 detectable signals. (The numbers depend upon the operating system and version.) You can change the default catch list by adding signals to or removing them from the default catch list.
Note – The list of signal names that dbx accepts includes all of those supported by the versions of the Solaris operating environment that dbx supports. So dbx might accept a signal that is not supported by the version of the Solaris operating environment you are running. For example, dbx might accept a signal that is supported by the Solaris 9 OS even through you are running the Solaris 7 OS. For a list of the signals supported by the Solaris OS you are running, see the signal(3head) man page.

To see the list of signals currently being trapped, type catch with no signal argument.

(dbx) catch

To see a list of the signals currently being ignored by dbx when the program detects them, type ignore with no signal argument.

(dbx) ignore

Changing the Default Signal Lists

You control which signals cause the program to stop by moving the signal names from one list to the other. To move signal names, supply a signal name that currently appears on one list as an argument to the other list.

For example, to move the QUIT and ABRT signals from the catch list to the ignore list:

(dbx) ignore QUIT ABRT

Trapping the FPE Signal (Solaris Platforms Only)

Often programmers working with code that requires floating point calculations want to debug exceptions generated in a program. When a floating point exception like overflow or divide by zero occurs, the system returns a reasonable answer as the result for the operation that caused the exception. Returning a reasonable answer lets the program continue executing quietly. The Solaris OS implements the IEEE Standard for Binary Floating Point Arithmetic definitions of reasonable answers for exceptions.

Because a reasonable answer for floating point exceptions is returned, exceptions do not automatically trigger the signal SIGFPE. Some integer exceptions, such as dividing an integer by zero and integer overflow do, by default, trigger the signal SIGFPE.

To find the cause of an exception, you need to set up a trap handler in the program so that the exception triggers the signal SIGFPE. (See ieee_handler(3m) man page for an example of a trap handler.)
You can enable a trap using:

- `ieee_handler`
- `fpsetmask` (see the `fpsetmask(3c) man page`)
- `-ftrap` compiler flag (for Fortran 95, see the `f95(1) man page`)

When you set up a trap handler using the `ieee_handler` command, the trap enable mask in the hardware floating point status register is set. This trap enable mask causes the exception to raise the SIGFPE signal at run time.

Once you have compiled the program with the trap handler, load the program into dbx. Before you can catch the SIGFPE signal, you must add FPE to the dbx signal catch list.

```
(dbx) catch FPE
```

By default, FPE is on the ignore list.

**Determining Where the Exception Occurred**

After adding FPE to the catch list, run the program in dbx. When the exception you are trapping occurs, the SIGFPE signal is raised and dbx stops the program. Then you can trace the call stack using the `dbx where` command to help find the specific line number of the program where the exception occurs (see "where Command" on page 373).

**Determining the Cause of the Exception**

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

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

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

The dbx cont command supports the -sig signal option, which lets you resume execution of a program with the program behaving as if it had received the system signal signal.

For example, if a program has an interrupt handler for SIGINT (^C), you can type ^C to stop the application and return control to dbx. If you issue a cont command by itself to continue program execution, the interrupt handler never executes. To execute the interrupt handler, send the signal, SIGINT, to the program:

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

The step command, next command, and detach command accept -sig as well.

Automatically Handling Signals

The event management commands can also deal with signals as events. These two commands have the same effect.

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

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

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

In this case, make sure to first move SIGCLD to the ignore list.

```
(dbx) ignore SIGCLD
```
This chapter describes how dbx handles C++ exceptions and debugging C++ templates, including a summary of commands used when completing these tasks and examples with code samples.

This chapter is organized into the following sections:
- “Using dbx With C++” on page 181
- “Exception Handling in dbx” on page 182
- “Debugging With C++ Templates” on page 185

For information on compiling C++ programs, see “Compiling a Program for Debugging” on page 44.

**Using dbx With C++**

Although this chapter concentrates on two specific aspects of debugging C++, dbx allows you full functionality when debugging your C++ programs. You can:
- Find out about class and type definitions (see “Looking Up Definitions of Types and Classes” on page 72)
- Print or display inherited data members (see “Printing C++ Pointers” on page 108)
- Find out dynamic information about an object pointer (see “Printing C++ Pointers” on page 108)
- Debug virtual functions (see “Calling a Function” on page 84)
- Using runtime type information (see “Printing the Value of a Variable, Expression, or Identifier” on page 108)
- Set breakpoints on all member functions of a class (see “Setting Breakpoints in All Member Functions of a Class” on page 90)
- Set breakpoints on all overloaded member functions (see “Setting Breakpoints in Member Functions of Different Classes” on page 90)
- Set breakpoints on all overloaded nonmember functions (see “Setting Multiple Breakpoints in Nonmember Functions” on page 91)
Set breakpoints on all member functions of a particular object (see “Setting Breakpoints in Objects” on page 91)
Deal with overloaded functions or data members (see “Setting a stop Breakpoint in a Function” on page 89)

Exception Handling in dbx

A program stops running if an exception occurs. Exceptions signal programming anomalies, such as division by zero or array overflow. You can set up blocks to catch exceptions raised by expressions elsewhere in the code.

While debugging a program, dbx enables you to:

- Catch unhandled exceptions before stack unwinding
- Catch unexpected exceptions
- Catch specific exceptions whether handled or not before stack unwinding
- Determine where a specific exception would be caught if it occurred at a particular point in the program

If you give a step command after stopping at a point where an exception is thrown, control is returned at the start of the first destructor executed during stack unwinding. If you step out of a destructor executed during stack unwinding, control is returned at the start of the next destructor. When all destructors have been executed, a step command brings you to the catch block handling the throwing of the exception.

Commands for Handling Exceptions

exception [-d | +d] Command

Use the exception command to display an exception’s type at any time during debugging. If you use the exception command without an option, the type shown is determined by the setting of the dbx environment variable output_dynamic_type:

- If it is set to on, the derived type is shown.
- If it is set to off (the default), the static type is shown.

Specifying the -d or +d option overrides the setting of the environment variable:

- If you specify -d, the derived type is shown.
- If you specify +d, the static type is shown.

For more information, see “exception Command” on page 305.
**intercept [-all] [-x] [-set] [typename] Command**

You can intercept, or catch, exceptions of a specific type before the stack has been unwound. Use the `intercept` command with no arguments to list the types that are being intercepted. Use `-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`, you could type:

```
(dbx) intercept -all -x int
```

To intercept exceptions of type `Error`, you would type:

```
(dbx) intercept Error
```

After intercepting too many `CommonError` exceptions, you could exclude these exceptions by typing:

```
(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 313.

**unintercept [-all] [-x] [typename] Command**

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 366.
**whocatches typename Command**

The whocatches command reports where an exception of typename would be caught if thrown at the current point of execution. Use this command to find out what would happen if an exception were thrown from the top frame of the stack.

The line number, function name, and frame number of the catch clause that would catch typename are displayed. The command returns “type is unhandled” if the catch point is in the same function that is doing the throw.

For more information, see “whocatches Command” on page 376.

**Examples of Exception Handling**

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

```c
#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
<dbg> 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
catched exception 99
stopped in main at line 26 in file "foo.cc"
    26 }

---

**Debugging With C++ Templates**

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

- Setting breakpoints at class or function template instantiations (see "`stop in class classname Command` on page 189, “`stop in function name Command` on page 189, and “`stop in function Command` on page 189)
- Printing a list of all class and function template instantiations (see “`whereis name Command` on page 187)
- Displaying the definitions of templates and instances (see “`what is name Command` on page 188)
- Calling member template functions and function template instantiations (see “call 
  function_name(parameters) Command” on page 189)
- Printing values of function template instantiations (“print Expressions” on page 190)
- Displaying the source code for function template instantiations (see “list Expressions” on 
  page 190)

**Template Example**

The following code example shows the class template `Array` and its instantiations and the 
function template `square` and its instantiations.

```cpp
1 template<class C> void square(C num, C *result)
2 {
3     *result = num * num;
4 }
5
6 template<class T> class Array
7 {
8     public:
9         int getlength(void)
10         {
11             return length;
12         }
13
14         T & operator[](int i)
15         {
16             return array[i];
17         }
18
19         Array(int l)
20         {
21             length = l;
22             array = new T[length];
23         }
24
25         ~Array(void)
26         {
27             delete [] array;
28         }
29
30     private:
31         int length;
32         T *array;
33     };
34
35 int main(void)
36 {
37     int i, j = 3;
38     square(j, &i);
39
40     double d, e = 4.1;
41     square(e, &d);
42
43     Array<int> iarray(5);
```
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 name Command

Use the whereis command to print a list of all occurrences of function or class instantiations for a function or class template.

For a class template:

```plaintext
(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:

```plaintext
(dbx) whereis square
function template instance: 'square<int>(__type_0, __type_0*)
function template instance: 'square<double>(__type_0, __type_0*)
```

The __type_0 parameter refers to the 0th template parameter. A __type_1 would refer to the next template parameter.
For more information, see "whereis Command" on page 375.

**whatis name Command**

Use the whatis command to print the definitions of function and class templates and instantiated functions and classes.

For a class template:

(dbx) **whatis -t Array**

```
template<class T> class Array
```

To get the full template declaration, try 'whatis -t Array<int>';

For the class template's constructors:

(dbx) **whatis Array**

More than one identifier 'Array'.
Select one of the following:
0) Cancel
1) Array<int>::Array(int)
2) Array<double>::Array(int)
> 1
Array<int>::Array(int 1);

For a function template:

(dbx) **whatis square**

More than one identifier 'square'.
Select one of the following:
0) Cancel
1) square<int(__type_0,__type_0*)
2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);

For a class template instantiation:

(dbx) **whatis -t Array<double>**

class Array<double>; {
    public:
        int Array<double>::getlength();
        double &Array<double>::operator [][int i];
        Array<double>::Array<double>[int l];
        Array<double>::~Array<double>();
    private:
        int length;
        double *array;
    };

For a function template instantiation:

(dbx) **whatis square(int, int*)**

void square<int num, int *result>;

For more information, see "whatis Command" on page 369.
**stop inclass classname Command**

To stop in all member functions of a template class:

```
(dbx) stop inclass Array
(2) stop inclass Array
```

Use the `stop inclass` command to set breakpoints at all member functions of a particular template class:

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

For more information, see "**stop Command**" on page 349 and "**inclass classname [-recurse | -norecurse]**" on page 254.

**stop infunction name Command**

Use the `stop infunction` command to set breakpoints at all instances of the specified function template:

```
(dbx) stop infunction square
(9) stop infunction square
```

For more information, see "**stop Command**" on page 349 and "**infunction function**" on page 253.

**stop in function Command**

Use the `stop in` command to set a breakpoint at a member function of a template class or at a template function.

For a member of a class template instantiation:

```
(dbx) stop in Array<int>::Array(int l)
(2) stop in Array<int>::Array(int)
```

For a function instantiation:

```
(dbx) stop in square(double, double*)
(6) stop in square(double, double*)
```

For more information, “**stop Command**” on page 349 and “**in function**” on page 252.

**call function_name(parameters) Command**

Use the `call` command to explicitly call a function instantiation or a member function of a class template when you are stopped in scope. If `dbx` is unable to determine the correct instance, it displays a numbered list of instances from which you can choose.
(dbx) **call square**(j,&i)

For more information, see “**call Command**” on page 279.

**print Expressions**

Use the print command to evaluate a function instantiation or a member function of a class template:

(\text{dbx}) \text{print } \text{iarray.getlength()}

\text{iarray.getlength()} = 5

Use print to evaluate the this pointer.

(\text{dbx}) \text{what is this}

\text{class Array<int> *this;}
(\text{dbx}) \text{print } \text{*this}

\text{*this = \{}
  \text{length} = 5
  \text{array} = 0x21608
\text{\}}

For more information, see “**print Command**” on page 333.

**list Expressions**

Use the list command to print the source listing for the specified function instantiation.

(\text{dbx}) \text{list square(int, int*)}

For more information, see “**list Command**” on page 317.
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 191
- “Debugging Segmentation Faults” on page 195
- “Locating Exceptions” on page 196
- “Tracing Calls” on page 196
- “Working With Arrays” on page 197
- “Showing Intrinsic Functions” on page 199
- “Showing Complex Expressions” on page 199
- “Showing Logical Operators” on page 201
- “Viewing Fortran 95 Derived Types” on page 201
- “Pointer to Fortran 95 Derived Type” on page 202

Debugging Fortran

The following tips and general concepts are provided to help you while debugging Fortran programs. For information on debugging Fortran OpenMP code with dbx, see "Interacting With Events" on page 164.

Current Procedure and File

During a debug session, dbx defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, stop at 5 sets different breakpoints, depending on which file is current.
Uppercase Letters

If your program has uppercase letters in any identifiers, dbx recognizes them. You need not provide case-sensitive or case-insensitive commands, as in some earlier versions.

Fortran 95 and dbx must be in the same case-sensitive or case-insensitive mode:

- Compile and debug in case-insensitive mode without the -U option. The default value of the dbx input_case_sensitive environment variable is then false.

  If the source has a variable named LAST, then in dbx, both the print LAST or print last commands work. Fortran 95 and dbx consider LAST and last to be the same, as requested.

- Compile and debug in case-sensitive mode using -U. The default value of the dbx input_case_sensitive environment variable is then true.

  If the source has a variable named LAST and one named last, then in dbx, print last works, but print LAST does not work. Fortran 95 and dbx distinguish between LAST and last, as requested.

Note – File or directory names are always case-sensitive in dbx, even if you have set the dbx input_case_sensitive environment variable to false.

Sample dbx Session

The following examples use a sample program called my_program.

Main program for debugging, a1.f:

```fortran
PARAMETER ( n=2 )
REAL twobytwo(2,2) / 4 *-1 /
CALL mkidentity( twobytwo, n )
PRINT *, determinant( twobytwo )
END
```

Subroutine for debugging, a2.f:

```fortran
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
  END DO
20 CONTINUE
90 CONTINUE
RETURN
END
```
Function for debugging, a3.f:

```fortran
REAL FUNCTION determinant ( a )
REAL a(2,2)
determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
RETURN
END
```

▼ Running the Sample dbx Session

1 Compile and link with the -g option.
   You can do this in one or two steps.
   Compile and link in one step, with -g:
   ```bash
demo% f95 -o my_program -g a1.f a2.f a3.f
```
   Or, compile and link in separate steps:
   ```bash
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.
   ```bash
demo% dbx my_program
```
   Reading symbolic information...

3 Set a simple breakpoint by typing stop in subnam, where subnam names a subroutine, function, or block data subprogram.
   To stop at the first executable statement in a main program.
   ```bash
   (dbx) stop in MAIN
   (2) stop in MAIN
   ```
   Although MAIN must be all uppercase, subnam can be uppercase or lowercase.

4 Type the run command, which runs the program in the executable files named when you started dbx.
   ```bash
   (dbx) run
   Running: my_program
   stopped in MAIN at line 3 in file "a1.f"
   3   call mkidentity( twobytwo, n )
   ```
   When the breakpoint is reached, dbx displays a message showing where it stopped, in this case, at line 3 of the a1.f file.

5 To print a value, type the print command.
   Print value of n:
   ```bash
   (dbx) print n
   n = 2
   ```
Print the matrix twobytwo; the format might vary:

    (dbx) print twobytwo
    twobytwo =
      (1,1) -1.0
      (2,1) -1.0
      (1,2) -1.0
      (2,2) -1.0

Print the matrix array:

    (dbx) print array
    dbx: "array" is not defined in the current scope
    (dbx)

The print fails because array is not defined here, only in \texttt{mkidentity}.

6 To advance execution to the next line, type the next command.

Advance execution to the next line:

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

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

Compare the next command with the step command. The step command executes the next source line or the next step into a subprogram. If the next executable source statement is a subroutine or function call, then:

- The step command sets a breakpoint at the first source statement of the subprogram.
- The next command sets the breakpoint at the first source statement after the call, but still in the calling program.

7 To quit dbx, type the quit command.

    (dbx) quit
    demo%
Debugging Segmentation Faults

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

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.
- The calling routine has a REAL argument, which the called routine has as INTEGER.
- An array index is miscalculated.
- The calling routine has fewer arguments than required.
- A pointer is used before it has been defined.

Using dbx to Locate Problems

Use dbx to find the source code line where a segmentation fault has occurred.

Use a program to generate a segmentation fault:

demo% cat WhereSEGV.f
   INTEGER a(5)
   j = 2000000
   DO 9 i = 1, 5
      a(j) = (i * 10)
   CONTINUE
   PRINT *, a
END

demo%

Use dbx to find the line number of a dbx segmentation fault:

demo% f95 -g -silent WhereSEGV.f
demo% a.out
Segmentation fault
demo% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
   in MAIN at line 4 in file "WhereSEGV.f"
   4       a(j) = (i * 10)
(dbx)
Locating Exceptions

If a program gets an exception, there are many possible causes. One approach to locating the problem is to find the line number in the source program where the exception occurred, and then look for clues there.

Compiling with `-ftrap=common` forces trapping on all common exceptions.

To find where an exception occurred:

demo% cat wh.f
    call joe(r, s)
    print *, r/s
end
subroutine joe(r,s)
    r = 12.
    s = 0.
    return
end

demo% f95 -g -o wh -ftrap=common wh.f
demo% dbx wh
Reading symbolic information for wh
(dbx) catch FPE
(dbx) run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in file “wh.f”
    2  print *, r/s
(dbx)

Tracing Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that led it there. This sequence is called a stack trace.

The `where` command shows where in the program flow execution stopped and how execution reached this point, a stack trace of the called routines.

ShowTrace.f is a program contrived to get a core dump a few levels deep in the call sequence, to show a stack trace.

*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)
quit 174% dbx a.out
Execution stopped, line 23
Reading symbolic information for a.out
Working With Arrays

dbx recognizes arrays and can print them.

demo% dbx a.out
Reading symbolic information...

(dbx) list 1,25
1 DIMENSION IARR(4,4)
2 DO 90 I = 1,4
3 DO 20 J = 1,4
4 IARR(I,J) = (I*10) + J
5 20 CONTINUE
6 90 CONTINUE
7 END

(dbx) stop at 7
(1) stop at "Arraysdbx.f":7

(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"

7 END

(dbx) print IARR
iarr =
(1,1) 11
(2,1) 21
(3,1) 31
(4,1) 41
(1,2) 12
(2,2) 22
(3,2) 32
(4,2) 42
(1,3) 13
(2,3) 23
(3,3) 33
(4,3) 43
(1,4) 14
(2,4) 24
(3,4) 34
(4,4) 44

(dbx) print IARR(2,3)
iarr(2, 3) = 23 - Order of user-specified subscripts ok

(dbx) quit
For information on array slicing in Fortran, see “Array Slicing Syntax for Fortran” on page 112.

**Fortran 95 Allocatable Arrays**

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

```
.demo% f95 -g Alloc.f95
.demo% dbx a.out

dbx) list 1.99
 1 PROGRAM TestAllocate
 2 INTEGER n, status
 3 INTEGER, ALLOCATABLE :: buffer(:)
 4 PRINT *, 'Size?'
 5 READ *, n
 6 ALLOCATE( buffer(n), STAT=status )
 7 IF ( status /= 0 ) STOP 'cannot allocate buffer'
 8 buffer(n) = n
 9 PRINT *, buffer(n)
10 DEALLOCATE( buffer, STAT=status)
11 END

dbx) stop at 6
(2) stop at "alloc.f95":6
(dbx) stop at 9
(3) stop at "alloc.f95":9
(dbx) run
Running: a.out
(process id 10749)
Size?
1000
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, type:

demo% cat ShowIntrinsic.f
INTEGER i
i = -2
END
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: shi
(process id 18019)
stopped in MAIN at line 2 in file "shi.f"
  2       i = -2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print i
i = 0
(dbx) step
stopped in MAIN at line 3 in file "shi.f"
  3       end
(dbx) print i
i = -2
(dbx) print abs(1)
abs(i) = 2

Showing Complex Expressions

dbx also recognizes Fortran complex expressions.

To show a complex expression in dbx, type:

demo% cat ShowComplex.f
COMPLEX z
z = ( 2.0, 3.0 )
END
demo% f95 -g ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
  2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
3 END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%

Showing Interval Expressions

To show an interval expression in dbx, type:

demo% cat ShowInterval.f95
INTERVAL v
 v = [ 37.1, 38.6 ]
END
demo% f95 -g -xia ShowInterval.f95
demo% dbx a.out
$dbx$) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: a.out
(process id 5217)
stopped in MAIN at line 2 in file "ShowInterval.f95"
2 v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16 v
(dbx) print v
v = [0.0,0.0]
(dbx) next
stopped in MAIN at line 3 in file "ShowInterval.f95"
3 END
(dbx) print v
v = [37.1,38.6]
(dbx) print v+[0.99,1.01]
v+[0.99,1.01] = [38.09,39.61]
(dbx) quit
demo%

Note – Interval expressions are supported only for programs compiled to run on SPARC based platforms, with -xarch=sse|sse2 to run on Solaris x86 SSE/SSE2 Pentium 4-compatible platforms, or with -xarch=amd64 to run on x64 platforms.
Showing Logical Operators

xdb can locate Fortran logical operators and print them.

To show logical operators in dbx, type:

demo% cat ShowLogical.f
LOGICAL a, b, y, z
  a = .true.
  b = .false.
  y = .true.
  z = .false.
END
demo% f95 -g ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
  1    LOGICAL a, b, y, z
  2      a = .true.
  3      b = .false.
  4      y = .true.
  5      z = .false.
  6    END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
(dbx) whatis y
logical*4 y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit

demo%

Viewing Fortran 95 Derived Types

You can show structures, Fortran 95 derived types, with dbx.

demo% f95 -g DebStruc.f95
demo% dbx a.out
(dbx) list 1,99
  1    PROGRAM Struct ! Debug a Structure
  2      TYPE product
  3      INTEGER     id
  4      CHARACTER*16 name
  5      CHARACTER*8 model
  6      REAL        cost
  7      REAL       price
END TYPE product

TYPE(product) :: prod1

prod1%id = 82
prod1%name = "Coffee Cup"
prod1%model = "XL"
prod1%cost = 24.0
prod1%price = 104.0

WRITE (*, *) prod1%name

END

Pointer to Fortran 95 Derived Type

You can show structures, Fortran 95 derived types, and pointers with dbx.

demo% f95 -o debstr -g DebStruc.f95
demo% dbx debstr
(dbx) stop in main
(2) stop in main
(dbx) list 1,99
1 PROGRAM DebStruPtr! Debug structures & pointers
Declare a derived type.
2   TYPE product
3     INTEGER   id
4     CHARACTER*16 name
5     CHARACTER*8 model
6     REAL      cost
7     REAL      price
Declare prod1 and prod2 targets.

Declare curr and prior pointers.

Make curr point to prod2.

Make prior point to prod1.

Initialize prior.

Set curr to prior.

Print name from curr and prior.

Above, dbx displays all fields of the derived type, including field names.

You can use structures and inquire about an item of an Fortran 95 derived type.

Ask about the variable

Ask about the type (-t)

To print a pointer, type:
Object Oriented Fortran

The Object Oriented Fortran features supported in dbx are type extension and polymorphic pointers, which is consistent with C++ support.

The dbx environment 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.
Debugging a Java Application With dbx

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

The chapter is organized into the following sections:
- “Using dbx With Java Code” on page 205
- “Environment Variables for Java Debugging” on page 206
- “Starting to Debug a Java Application” on page 207
- “Customizing Startup of the JVM Software” on page 211
- “dbx Modes for Debugging Java Code” on page 214
- “Using dbx Commands in Java Mode” on page 215

Using dbx With Java Code

You can use the Oracle Solaris 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 (see “Starting to Debug a Java Application” on page 207). 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 environment variables are specific to debugging a Java application with dbx. You can set the JAVASRCPATH, CLASSPATHX, and jvm_invocation environment variables at a shell prompt before starting dbx, 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 ("jon Command" on page 315) and the joff command (see "joff Command" on page 314).

<table>
<thead>
<tr>
<th>jdbx_mode</th>
<th>The jdbx_mode environment variable can have the following settings: java, jni, or native. For descriptions of the Java, JNI, and native modes, and how and when the mode changes, see &quot;dbx Modes for Debugging Java Code&quot; on page 214. Default: java.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVASRCPATH</td>
<td>You can use the JAVASRCPATH environment variable to specify the directories in which dbx should look for Java source files. This variable is useful when the Java sources files are not in the same directory as the .class or .jar files. See &quot;Specifying the Location of Your Java Source Files&quot; on page 210 for more information.</td>
</tr>
<tr>
<td>CLASSPATHX</td>
<td>The CLASSPATHX environment variable lets you specify to dbx a path for Java class files that are loaded by custom class loaders. For more information, see &quot;Specifying a Path for Class Files That Use Custom Class Loaders&quot; on page 210.</td>
</tr>
<tr>
<td>jvm_invocation</td>
<td>The jvm_invocation environment variable lets you customize the way the JVM software is started. (The terms &quot;Java virtual machine&quot; and &quot;JVM&quot; mean a virtual machine for the Java platform.) For more information, see &quot;Customizing Startup of the JVM Software&quot; on page 211.</td>
</tr>
</tbody>
</table>
Starting to Debug a Java Application

You can use dbx to debug the following types of Java applications:

- A file with a file name that ends in .class
- A file with a file name that ends in .jar
- A Java application that is started using a wrapper
- A running Java application that was started in debug mode to which you attach dbx
- A C application or C++ application that embeds a Java application using the JNI_CreateJavaVM interface

dbx recognizes that it is debugging a Java application in all of these cases.

Debugging a Class File

You can debug a file that uses the .class file name extension using dbx as in the following example.

```
(dbx) debug myclass.class
```

If the class that defines the application is defined in a package, you need to include the package path just as when running the application under the JVM software, as in the following example.

```
(dbx) debug java.pkg.Toy.class
```

You can also use a full path name for the class file. dbx automatically determines the package portion of the class path by looking in the .class file and adds the remaining portion of the full path name to the class path. For example, given the following path name, dbx determines that pkg/Toy.class is the main class name and adds /home/user/java to the class path.

```
(dbx) debug /home/user/java/pkg/Toy.class
```

Debugging a JAR File

A Java application can be bundled in a JAR (Java Archive) file. You can debug a JAR file using dbx as in the following example.

```
(dbx) debug myjar.jar
```

When you start debugging a file that has a file name ending in .jar, dbx uses the Main_Class attribute specified in the manifest of this JAR file to determine the main class. (The main class is the class within the JAR file that is your application’s entry point. If you use a full path name or relative path name to specify the JAR file, dbx uses the directory name and prefixes it to the class path in the Main-Class attribute.)
If you debug a JAR file that does not have the Main-Class attribute, you can use the JAR URL syntax jar:<url>!/{entry} that is specified in the class JarURLConnection of the Java 2 Platform, Standard Edition to specify the name of the main class, as in the following examples.

(dbx) debug jar:myjar.jar!/myclass.class
(dbx) debug jar:a/b/c/d/e.jar!/x/y/z.class
(dbx) debug jar:file:/a/b/c/d.jar!/myclass.class

For each of these examples dbx would do the following:

- Treat the class path specified after the ! character as the main class (for example, /myclass.class or /x/y/z.class)
- Add the name of the JAR file ./myjar.jar, /a/b/c/d/e.jar, or /a/b/c/d.jar to the class 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 21), the file name of the JAR file is not automatically added to the class path. In this case, you must add the file name of the JAR file to the class path when you start debugging.

## Debugging a Java Application That Has a Wrapper

A Java application usually has a wrapper to set environment variables. If your Java application has a wrapper, you need to tell dbx that a wrapper script is being used by setting the jvm_invocation environment variable (see “Customizing Startup of the JVM Software” on page 211).

## Attaching dbx to a Running Java Application

You can attach dbx to a running Java application if you specified the options shown in the following example when you started the application. After starting the application, you would use the dbx command (see “dbx Command” on page 293) with the process ID of the running Java process to start debugging.

$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent myclass.class
$ dbx - 2345
For the JVM software to locate \texttt{libdbx\_agent.so}, you need to add the appropriate path to \texttt{LD\_LIBRARY\_PATH} before running the Java application:

- If you are using the 32-bit version of the JVM software on a system running the Solaris OS, add \texttt{/installation\_directory/SUNWspro/lib/libdbx\_agent.so}.
- If you are using the 64-bit version of the JVM software on a SPARC based system running the Solaris OS, add \texttt{/installation\_directory/SUNWspro/lib/v9/libdbx\_agent.so} to \texttt{LD\_LIBRARY\_PATH}.
- If you are using the 64-bit version of the JVM software on an x64 based system running the Linux OS, add \texttt{/installation\_directory/sunstudio12/lib/amd64/libdbx\_agent.so} to \texttt{LD\_LIBRARY\_PATH}.

The \texttt{installation\_directory} is the location where the Oracle Solaris Studio software is installed.

When you attach \texttt{dbx} to the running application, \texttt{dbx} starts debugging the application in Java mode.

If your Java application requires 64-bit object libraries, include the \texttt{-d64} option when you start the application. Then when you attach \texttt{dbx} to the application, \texttt{dbx} will use the 64-bit JVM software on which the application is running.

\texttt{\$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx\_agent -d64 myclass.class}

\texttt{\$ dbx - 2345}

### Debugging a C Application or C++ Application That Embeds a Java Application

You can debug a C application or C++ application that embeds a Java application using the \texttt{JNI\_CreateJavaVM} interface. The C application or C++ application must start the Java application by specifying the following options to the JVM software:

\texttt{-Xdebug -Xnoagent -Xrundbx\_agent}

For the JVM software to locate \texttt{libdbx\_agent.so}, you need to add the appropriate path to \texttt{LD\_LIBRARY\_PATH} before running the Java application:

- If you are using the 32-bit version of the JVM software on a system running the Solaris OS, add \texttt{/installation\_directory/SUNWspro/lib/libdbx\_agent.so} to \texttt{LD\_LIBRARY\_PATH}.
- If you are using the 64-bit version of the JVM software on a SPARC based system running the Solaris OS, add \texttt{/installation\_directory/SUNWspro/lib/v9/libdbx\_agent.so} to \texttt{LD\_LIBRARY\_PATH}.
- If you are using the 64-bit version of the JVM software on an x64 based system running the Linux OS, add \texttt{/installation\_directory/sunstudio12/lib/amd64/libdbx\_agent.so} to \texttt{LD\_LIBRARY\_PATH}.
The installation directory is the location where the Oracle Solaris 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 211.

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 $JAVA_SRC_PATH environment variable to specify the directories in which dbx should look for Java source files. For example

```
JAVA_SRC_PATH=.:/mydir/mysrc:/mydir/mylibsrc:/mydir/myutils
```
causes dbx to look in the listed directories for source files that correspond to the class files being debugged.

Specifying the Location of Your C Source Files or C++ Source Files

dbx might not be able to find your C source files or C++ source files in the following circumstances:

- If your source files are not in the same location as they were when you compiled them
- If you compiled your source files on a different system than the one on which you are running dbx and the compile directory does not have the same path name

In such cases, use the pathmap command (see "pathmap Command" on page 331) 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 you cannot simply type:

```
(dbx) stop in myMethod
```

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 in method`:

```
(dbx) stop in method myMethod
```

But doing so might cause stops in multiple methods named `myMethod`.

Setting Breakpoints in Native (JNI) Code

The shared libraries 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 98.

Customizing Startup of the JVM Software

You might need to customize startup of the JVM software from `dbx` to do the following:

- Specify a path name for the JVM software (see “Specifying a Path Name for the JVM Software” on page 212)
- Pass some run arguments to the JVM software (see “Passing Run Arguments to the JVM Software” on page 212)
- Specify a custom wrapper instead of the default Java wrapper for running Java applications (see “Specifying a Custom Wrapper for Your Java Application” on page 212)
- Specify 64-bit JVM software (see “Specifying 64-bit JVM Software” on page 214)

You can customize startup of the JVM software using the `jvm_invocation` environment variable. By default, when the `jvm_invocation` environment variable is not defined, `dbx` starts the JVM software as follows:

```
java -Xdebug -Xnoagent -Xrundbx_agent:syncpid
```
When the `jvm_invocation` environment variable is defined, dbx uses the value of the variable to start the JVM software.

You must include the `-Xdebug` option in the definition of the `jvm_invocation` environment variable. dbx expands `-Xdebug` into the internal options `-Xdebug -Xnoagent -Xrundbxagent::sync`.

If you do not include the `-Xdebug` option in the definition, as in the following example, dbx issues an error message.

```
jvm_invocation="/set/java/javasoft/sparc-S2/jdk1.2/bin/java"
dbx: Value of 'jvm_invocation' must include an option to invoke the VM in debug mode
```

**Specifying a Path Name for the JVM Software**

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 in the following example.

```
jvm_invocation="/myjava/java -Xdebug"
```

This setting causes dbx to start the JVM software as follows:

```
/myjava/java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent::sync
```

**Passing Run Arguments to the JVM Software**

To pass run arguments to the JVM software, set the `jvm_invocation` environment variable to start the JVM software with those arguments, as in the following example.

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

This causes dbx to start the JVM software as follows:

```
java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent::sync= -Xms512 -Xmx1024 -Xcheck:jni
```

**Specifying a Custom Wrapper for Your Java Application**

A Java application can use a custom wrapper for startup. If your application uses a custom wrapper, you can use the `jvm_invocation` environment variable to specify the wrapper to be used, as in the following example.
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"

This causes dbx to start the JVM software as follows:

/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug -J-Xnoagent -J-Xrundbxagent:sync=process_id

Using a Custom Wrapper That Accepts Command-Line Options

The following wrapper script (xyz) sets a few environment variables and accepts command line options:

```
#!/bin/sh
CPATH=/mydir/myclass:/mydir/myjar.jar; export CPATH
JARGS="-verbose:gc -verbose:jni -DXYZ=/mydir/xyz"
ARGS=
while [ $# -gt0 ]; do
case "$1" in
  -userdir) shift; if [ $# -gt 0 ]; then userdir=$1; fi;;
  -J*) jopt=’expr $1 : ’-J<.*>’’
    JARGS="$JARGS \$jopt";;
*) ARGS="$ARGS \$1" ;;
esac
shift
done
java $JARGS -cp $CPATH $ARGS
```

This script accepts some command line options for the JVM software and the user application. For wrapper scripts of this form, you would set the jvm_invocation environment variable and start dbx as follows:

```
% jvm_invocation="xyz -J-Xdebug -Jany other java options"
% dbx myclass.class -Dide=visual
```

Using a Custom Wrapper That Does Not Accept Command-Line Options

The following wrapper script (xyz) sets a few environment variables and starts the JVM software, but does not accept any command line options or a class name:

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
java <options> myclass
```

You could use such a script to debug a wrapper using dbx in one of two ways:

- You could modify the script to start dbx from inside the wrapper script itself by adding the definition of the jvm_invocation variable to the script and starting dbx:

  ````
  #!/bin/sh
  CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
  ABC=/mydir/abc; export ABC
  jvm_invocation="java -Xdebug <options>"; export jvm_invocation
  dbx myclass.class
  ```

  ```
Once you have made this modification, you could start the debugging session by running the script.

- You could modify the script slightly to accept some command line options as follows:

```bash
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
JAVA_OPTIONS="$1 <options>"
java $JAVA_OPTIONS $2
```

Once you made this modification, you would set the `jvm_invocation` environment variable and start `dbx` as follows:

```
% jvm_invocation="xyz -Xdebug"; export jvm_invocation
% dbx myclass.class
```

**Specifying 64-bit JVM Software**

If you want `dbx` to start 64-bit JVM software to debug an application that requires 64-bit object libraries, include the `-d64` option when you set the `jvm_invocation` environment variable:

```
jvm_invocation="/myjava/java -Xdebug -d64" 
```

## dbx Modes for Debugging Java Code

When debugging a Java application, `dbx` is in one of three modes:

- Java mode
- JNI mode
- Native mode

When `dbx` is Java mode or JNI (Java Native Interface) mode, you can inspect the state of your Java application, including JNI code, and control execution of the code. When `dbx` is in native mode, you can inspect the state of your C or C++ JNI code. The current mode (java, jni, native) is stored in the environment variable `jdbx_mode`.

In Java mode, you interact with `dbx` using Java syntax and `dbx` uses Java syntax to present information to you. This mode is used for debugging pure Java code, or the Java code in an application that is a mixture of Java code and C JNI code or C++ JNI code.

In JNI mode, `dbx` commands use native syntax and affect native code, but the output of commands shows Java-related status as well as native status, so JNI mode is a “mixed” mode. This mode is used for debugging the native parts of an application that is a mixture of Java code and C JNI code or C++ JNI code.

In native mode, `dbx` commands affect only a native program, and all Java-related features are disabled. This mode is used for debugging non-Java related programs.
As you execute your Java application, dbx switches automatically between Java mode and JNI mode as appropriate. For example, when it encounters a Java breakpoint, dbx switches into Java mode, and when you step from Java code into JNI code, it switches into JNI mode.

**Switching from Java or JNI Mode to Native Mode**

dbx does not switch automatically into native mode. You can switch explicitly from Java or JNI Mode to native mode with the `joff` command, and from native mode to Java mode with the `jon` command.

**Switching Modes When You Interrupt Execution**

If you interrupt execution of your Java application (for example, with a control-C), dbx tries to set the mode automatically to Java/JNI mode by bringing the application to a safe state and suspending all threads.

If dbx cannot suspend the application and switch to Java/JNI mode, dbx switches to native mode. You can then use the `jon` command to switch to Java mode so that you inspect the state of the program.

**Using dbx Commands in Java Mode**

When you are using dbx to debug a mixture of Java and native code, dbx commands fall into several categories:

- Commands that accept the same arguments and operate the same way in Java mode or JNI mode as in native mode (see "Commands With Identical Syntax and Functionality in Java Mode and Native Mode" on page 217).

- Commands have arguments that are valid only in Java mode or JNI mode, as well as arguments that are valid only in native mode (see "Commands With Different Syntax in Java Mode" on page 218).

- Commands that are valid only in Java mode or JNI mode (see “Commands Valid Only in Java Mode” on page 219).

Any commands not included in one of these categories work only in native mode.
The Java Expression Evaluation in dbx Commands

The Java expression evaluator used in most dbx commands supports the following constructs:

- All literals
- All names and field accesses
- `this` and `super`
- Array accesses
- casts
- Conditional binary operations
- Method calls
- Other unary/binary operations
- Assignment to variables or fields
- `instanceof` operator
- Array length operator

The Java expression evaluator does not support the following constructs:

- Qualified `this`, for example, `<ClassName>.this`
- Class instance creation expressions
- Array creation expressions
- String concatenation operator
- Conditional operator `?:`
- Compound assignment operators, for example, `x += 3`

A particularly useful way of inspecting the state of your Java application is using the watch facility in the IDE or dbx tool.

Depending on precise value semantics in expressions that do more than just inspect data is not recommended.

Static and Dynamic Information Used by dbx Commands

Much of the information about a Java application is normally available only after the JVM software has started, and is unavailable after the Java application has finished executing. However, when you debug a Java application with dbx, dbx gleans some of the information it needs from class files and JAR files that are part of the system class path and user class path before it starts the JVM software. This allows dbx to do better error checking on breakpoints before you run the application.

Some Java classes and their attributes might not be accessible through the class path. dbx can inspect and step through these classes, and the expression parser can access them, once they are loaded. However, the information it gathers is temporary and is no longer available after the JVM software terminates.
Some information that `dbx` needs to debug your Java application is not recorded anywhere so `dbx` skims Java source files to derive this information as it is debugging your code.

### Commands With Identical Syntax and Functionality in Java Mode and Native Mode

The following `dbx` commands have the same syntax and perform the same operations in Java mode as in native mode.

<table>
<thead>
<tr>
<th>Command</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach</td>
<td>Attaches <code>dbx</code> to a running process, stopping execution and putting the program under debugging control</td>
</tr>
<tr>
<td>cont</td>
<td>Causes the process to continue execution</td>
</tr>
<tr>
<td>dbxenv</td>
<td>List or set <code>dbx</code> environment variables</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes breakpoints and other events</td>
</tr>
<tr>
<td>down</td>
<td>Moves down the call stack (away from <code>main</code>)</td>
</tr>
<tr>
<td>dump</td>
<td>Prints all variables local to a procedure or method</td>
</tr>
<tr>
<td>file</td>
<td>Lists or changes the current file</td>
</tr>
<tr>
<td>frame</td>
<td>Lists or changes the current stack frame number</td>
</tr>
<tr>
<td>handler</td>
<td>Modifies event handlers (breakpoints)</td>
</tr>
<tr>
<td>import</td>
<td>Import commands from a <code>dbx</code> command library</td>
</tr>
<tr>
<td>line</td>
<td>Lists or changes the current line number</td>
</tr>
<tr>
<td>list</td>
<td>Lists or changes the current line number</td>
</tr>
<tr>
<td>next</td>
<td>Steps one source line (steps over calls)</td>
</tr>
<tr>
<td>pathmap</td>
<td>Maps one path name to another for finding source files and such</td>
</tr>
<tr>
<td>proc</td>
<td>Displays the status of the current process</td>
</tr>
<tr>
<td>prog</td>
<td>Manages programs being debugged and their attributes</td>
</tr>
<tr>
<td>quit</td>
<td>Exits <code>dbx</code></td>
</tr>
<tr>
<td>rerun</td>
<td>Runs the program with no arguments</td>
</tr>
<tr>
<td>runargs</td>
<td>Changes the arguments of the target process</td>
</tr>
<tr>
<td>status</td>
<td>Lists the event handlers (breakpoints)</td>
</tr>
</tbody>
</table>
## Command Functionality

- **step up**: Steps up and out of the current function or method
- **stepi**: Steps one machine instruction (steps into calls)
- **up**: Moves up the call stack (toward main)
- **whereami**: Displays the current source line

### Commands With Different Syntax in Java Mode

The following dbx commands have different syntax for Java debugging than for native code debugging, and operate differently in Java mode than in native mode.

<table>
<thead>
<tr>
<th>Command</th>
<th>Native Mode Functionality</th>
<th>Java Mode Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>Assigns a new value to a program variable</td>
<td>Assigns a new value to a local variable or parameter</td>
</tr>
<tr>
<td>call</td>
<td>Calls a procedure</td>
<td>Calls a method</td>
</tr>
<tr>
<td>dbx</td>
<td>Starts dbx</td>
<td>Starts dbx</td>
</tr>
<tr>
<td>debug</td>
<td>Loads the specified application and begins debugging the application</td>
<td>Loads the specified Java application, checks for the existence of the class file, and begins debugging the application</td>
</tr>
<tr>
<td>detach</td>
<td>Releases the target process from dbx’s control</td>
<td>Releases the target process from dbx’s control</td>
</tr>
<tr>
<td>display</td>
<td>Evaluates and prints expressions at every stopping point.</td>
<td>Evaluates and prints expressions, local variables, or parameters at every stopping point</td>
</tr>
<tr>
<td>files</td>
<td>Lists file names that match a regular expression</td>
<td>Lists all of the Java source files known to dbx</td>
</tr>
<tr>
<td>func</td>
<td>Lists or changes the current function</td>
<td>Lists or changes the current method</td>
</tr>
<tr>
<td>next</td>
<td>Steps one source line (stepping over calls)</td>
<td>Steps one source line (stepping over calls)</td>
</tr>
<tr>
<td>print</td>
<td>Prints the value of an expression</td>
<td>Prints the value of an expression, local variable, or parameter.</td>
</tr>
<tr>
<td>run</td>
<td>Runs the program with arguments</td>
<td>Runs the program with arguments</td>
</tr>
<tr>
<td>step</td>
<td>Steps one source line or statement (stepping into calls)</td>
<td>Steps one source line or statement (stepping into calls)</td>
</tr>
<tr>
<td>stop</td>
<td>Sets a source-level breakpoint</td>
<td>Sets a source-level breakpoint</td>
</tr>
<tr>
<td>Command</td>
<td>Native Mode Functionality</td>
<td>Java Mode Functionality</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>thread</td>
<td>Lists or changes the current thread</td>
<td>Lists or changes the current thread</td>
</tr>
<tr>
<td>threads</td>
<td>Lists all threads</td>
<td>Lists all threads</td>
</tr>
<tr>
<td>trace</td>
<td>Shows executed source lines, function calls, or variable changes</td>
<td>Shows executed source lines, function calls, or variable changes</td>
</tr>
<tr>
<td>undisplay</td>
<td>Undoes display commands</td>
<td>Undoes display commands</td>
</tr>
<tr>
<td>whatis</td>
<td>Prints the type of expression or declaration of type</td>
<td>Prints the declaration of an identifier</td>
</tr>
<tr>
<td>when</td>
<td>Executes commands when a specified event occurs</td>
<td>Executes commands when a specified event occurs</td>
</tr>
<tr>
<td>where</td>
<td>Prints the call stack</td>
<td>Prints the call stack</td>
</tr>
</tbody>
</table>

## Commands Valid Only in Java Mode

The following `dbx` commands are valid only in Java mode or JNI mode.

<table>
<thead>
<tr>
<th>Command</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>Used when <code>dbx</code> is in JNI mode to indicate that the Java version of a specified command is to be executed</td>
</tr>
<tr>
<td>javaclasses</td>
<td>Prints the names of all Java classes known to <code>dbx</code> when you give the command</td>
</tr>
<tr>
<td>joff</td>
<td>Switches <code>dbx</code> from Java mode or JNI mode to native mode</td>
</tr>
<tr>
<td>jon</td>
<td>Switches <code>dbx</code> from native mode to Java mode</td>
</tr>
<tr>
<td>jpkg</td>
<td>Prints the names of all Java packages known to <code>dbx</code> when you give the command</td>
</tr>
<tr>
<td>native</td>
<td>Used when <code>dbx</code> is in Java mode to indicate that the native version of a specified command is to be executed</td>
</tr>
</tbody>
</table>
This chapter describes how to use event management and process control commands at the machine-instruction level, how to display the contents of memory at specified addresses, and how to display source lines along with their corresponding machine instructions. The next 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.

This chapter is organized into the following sections:
- “Examining the Contents of Memory” on page 221
- “Stepping and Tracing at Machine-Instruction Level” on page 225
- “Setting Breakpoints at the Machine-Instruction Level” on page 227
- “Using the regs Command” on page 228

### Examining the Contents of Memory

Using addresses and the examine or x command, you can examine the content of memory locations as well as print the assembly language instruction at each address. Using a command derived from adb(1), the assembly language debugger, you can query for:

- The address, using the = (equal sign) character
- The contents stored at an address, using the / (slash) character

You can print the assembly commands using the dis command and the listi command. (See “Using the dis Command” on page 224 and “Using the listi Command” on page 224.)

### Using the examine or x Command

Use the examine command, or its alias x, to display memory contents or addresses.
Use the following syntax to display the contents of memory starting at address for count items in format format. The default address is the next one after the last address previously displayed. The default count is 1. The default format is the same as was used in the previous examine command, or X if this is the first command given.

The syntax for the examine command is:

```
examine [address] [/ [count] [format]]
```

To display the contents of memory from address1 through address2 inclusive, in format format, type:

```
examine address1, address2 [/ [format]]
```

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

```
examine address = [format]
```

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

```
examine +/ i
```

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

```
examine address=format
examine address=
```

### Using Addresses

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

For example, the following are valid addresses:

- `0xff99` An absolute address
- `main` Address of a function
- `main+20` Offset from a function address
- `&errno` Address of a variable
- `str` A pointer-value variable pointing to a string

Symbolic addresses used to display memory are specified by preceding a name with an ampersand (`&`). Function names can be used without the ampersand; `&main` is equal to `main`. Registers are denoted by preceding a name with a dollar sign (`$`).
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.

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 count 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.

Here is the syntax for the dis command.

```
dis [address] [address1, address2] [/count]
```

The dis command:
- Without arguments displays 10 instructions starting at +.
- With the address argument only, disassembles 10 instructions starting at address.
- With the 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 +.

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 63.
For SPARC based systems:

```
(i) listi 13, 14

i = atoi(argv[1]);
0x001003c: main+0x0014: ld [%fp + 0x48], %\l
0x0010040: main+0x0010: add %l0, 0x4, %l0
0x0010044: main+0x001c: ld [%l0], %l0
0x0010048: main+0x0020: or %l0, %g0, %o0
0x001004c: main+0x0024: call 0x000209e8 [unresolved PLT 7: atoi]
0x0010050: main+0x0028: nop
0x0010054: main+0x002c: or %o0, %g0, %l0
0x0010058: main+0x0030: st %l0, [%fp - 0x8]
0x001005c: main+0x0034: ld [%fp - 0x8], %l0
0x0010060: main+0x0038: or %l0, %g0, %o0
0x0010064: main+0x003c: call foo
0x0010068: main+0x0040: nop
0x001006c: main+0x0044: or %o0, %g0, %l0
0x0010070: main+0x0048: st %l0, [%fp - 0xc]
```

For x86 based systems:

```
(i) listi 13, 14

i = atoi(argv[1]);
0x080488fd: main+0x001d: movl 12(%ebp),%eax
0x08048900: main+0x0010: movl 4(%eax),%eax
0x08048903: main+0x0013: pushl %eax
0x08048904: main+0x0014: call atoi <0x8048790>
0x08048909: main+0x0019: addl $4,%esp
0x0804890c: main+0x001c: movl %eax,-8(%ebp)
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 \textbf{nexti} command or the \textbf{stepi} command

The \texttt{nexti} command and the \texttt{stepi} command behave the same as their source-code level counterparts: the \texttt{nexti} command steps over functions, the \texttt{stepi} command steps into a function called by the next instruction (stopping at the first instruction in the called function). The command forms are also the same. See “next Command” on page 327 and “step Command” on page 346 for a description.
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 328 and “`stepi Command`” on page 348.

**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 96 and “`tracei Command`” on page 363.

Here is the general syntax for the `tracei` command:

```
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 363.

For SPARC:
Setting Breakpoints at the Machine-Instruction Level

To set a breakpoint at the machine-instruction level, use the `stopi` command. The command accepts any event specification, using the syntax:

```
stopi event-specification [modifier]
```

Commonly used forms of the `stopi` command are:

- `stopi [at address] [-if cond]`
- `stopi in function [-if cond]`

For more information, see “stopi Command” on page 353.

Setting a Breakpoint at an Address

To set a breakpoint at a specific address, type:

```
stopi at address
```

For example:

```
nexti
stopped in hand::ungrasp at 0x12638
stopi &hand::ungrasp
(3) stopi at &hand::ungrasp
```

Chapter 18 • Debugging at the Machine-Instruction Level
Using the \texttt{regs} Command

The \texttt{regs} command lets you print the value of all the registers.

Here is the syntax for the \texttt{regs} command:

\texttt{regs [-f][-F]}

- \texttt{-f} includes floating point registers (single precision).
- \texttt{-F} includes floating point registers (double precision).

For more information, see “\texttt{regs Command}” on page 338.

For SPARC based systems:

```
dbx[13]  \texttt{regs -F}
current thread: t@1
current frame: [1]
g0-g3 0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7 0x00000000 0x00000000 0x00000000 0x00020c38
e0-e3 0x00000003 0x0000014e 0xef7562b4 0xffffffff
epc 0xeffff420 0x00000000 0x00000000
04-07 0xeff752f0b 0x00000000 0x00000000
10-13 0x00000000 0x00000000 0x00000000 0x00000000
l4-17 0xe0ff438 0x00000001 0xe0ff74df54
10-13 0x00000000 0xe0ff440 0x00000000 0x00000000
i4-i7 0x00000000 0xe0ff440 0x00000000 0x00000000
y 0x00000000
psr 0x00010008
pc 0x00000000
npc 0xe0ff438 0x00000001
f0f1 +0.00000000000000e+00
f2f3 +0.00000000000000e+00
f4f5 +0.00000000000000e+00
f6f7 +0.00000000000000e+00
```

For x64 based systems:

```
(dbx) \texttt{regs}
current frame: [1]
r15 0x0000000000000000
r14 0x0000000000000000
r13 0x0000000000000000
r12 0x0000000000000000
r11 0x0000000000000000
r10 0x0000000000000000
r9 0x0000000000000000
r8 0x0000000000416cf0
rdi 0x0000000000000000
rsi 0x0000000000000000
rbb 0x0000000000000000
rbx 0x0000000000000000
rdx 0x0000000000000000
rcx 0x0000000000000000
rax 0x0000000000000000
```

Using \texttt{the \texttt{regs} Command}
Chapter 18 • Debugging at the Machine-Instruction Level 229

Using the regs Command

trapno 0x0000000000000003
err 0x0000000000000000
rip 0x00000000001709:main+0xf9 movl $0x0000000000000000,0xfffffffffffffffc(%rbp)
cs 0x000000000000004b
eflags 0x0000000000000026
rsp 0xffffffff7ffdf7b0
ss 0x0000000000000043
fs 0x000000000000001b
gs 0x0000000000000000
es 0x0000000000000000
ds 0x0000000000000000
fsbase 0xffffffff7ff3a2000
gsbase 0xffffffff80000000

(dbx) regs -F

current frame: [1]
r15 0x0000000000000000
r14 0x0000000000000000
r13 0x0000000000000000
r12 0x0000000000000000
r11 0x0000000000000000
r10 0x0000000000000000
r9 0x0000000000000000
r8 0x0000000000000000
rdi 0x0000000000000000
rsi 0x0000000000000000
rbp 0xfffffd7fffdff820
rbx 0xfffffd7fff3fb190
rdx 0x0000000000000000
rcx 0x0000000000000000
rax 0x0000000000000000

r15 0x0000000000000000
r14 0x0000000000000000
r13 0x0000000000000000
r12 0x0000000000000000
r11 0x0000000000000000
r10 0x0000000000000000
r9 0x0000000000000000
r8 0x0000000000000000
rdi 0x0000000000000000
rsi 0x0000000000000000
rbp 0xfffffd7fffdff820
rbx 0xfffffd7fff3fb190
rdx 0x0000000000000000
rcx 0x0000000000000000
rax 0x0000000000000000

trapno 0x0000000000000003
err 0x0000000000000000
rip 0x00000000001709:main+0xf9 movl $0x0000000000000000,0xfffffffffffffffc(%rbp)
cs 0x000000000000004b
eflags 0x0000000000000026
rsp 0xffffffff7ffdf7b0
ss 0x0000000000000043
fs 0x000000000000001b
gs 0x0000000000000000
es 0x0000000000000000
ds 0x0000000000000000
fsbase 0xffffffff7ff3a2000
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 0xff000000 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
Platform-Specific Registers

The following tables list platform-specific register names for SPARC architecture, x86 architecture, and AMD64 architecture that can be used in expressions.

SPARC Register Information

The following register information is for SPARC architecture.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g0 through $g7</td>
<td>Global registers</td>
</tr>
<tr>
<td>$o0 through $o7</td>
<td>“out” registers</td>
</tr>
<tr>
<td>$l0 through $l7</td>
<td>“local” registers</td>
</tr>
<tr>
<td>$i0 through $i7</td>
<td>“in” registers</td>
</tr>
<tr>
<td>$fp</td>
<td>Frame pointer, equivalent to register $i6</td>
</tr>
<tr>
<td>$sp</td>
<td>Stack pointer, equivalent to register $o6</td>
</tr>
<tr>
<td>$y</td>
<td>Y register</td>
</tr>
<tr>
<td>$psr</td>
<td>Processor state register</td>
</tr>
<tr>
<td>$wim</td>
<td>Window invalid mask register</td>
</tr>
<tr>
<td>$tbr</td>
<td>Trap base register</td>
</tr>
<tr>
<td>$pc</td>
<td>Program counter</td>
</tr>
<tr>
<td>$npc</td>
<td>Next program counter</td>
</tr>
<tr>
<td>$f0 through $f31</td>
<td>FPU “f” registers</td>
</tr>
<tr>
<td>$fsr</td>
<td>FPU status register</td>
</tr>
</tbody>
</table>
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:

$sq0$ $sq4$ through $sq60$

The following pairs of registers, which combine the least significant 32 bits of two registers, are available on SPARC V8+ hardware:

$s0g1$ through $s6g7$
$s0o1$ through $s6o7$

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

$sxg0$ through $sxg7$
$sxo0$ through $sxo7$
$sxfsr$ $ststate$ $gsr$
$sf32f33$ $sf34f35$ through $sf62f63$ ($sd32$ ... $sd62$)


**x86 Register Information**

The following register information is for x86 architecture.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gs$</td>
<td>Alternate data segment register</td>
</tr>
<tr>
<td>$fs$</td>
<td>Alternate data segment register</td>
</tr>
<tr>
<td>$es$</td>
<td>Alternate data segment register</td>
</tr>
<tr>
<td>$ds$</td>
<td>Data segment register</td>
</tr>
<tr>
<td>$edi$</td>
<td>Destination index register</td>
</tr>
<tr>
<td>$esi$</td>
<td>Source index register</td>
</tr>
<tr>
<td>$ebp$</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$esp$</td>
<td>Stack pointer</td>
</tr>
</tbody>
</table>
Commonly used registers are also aliased to their machine independent names.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp</td>
<td>Stack pointer; equivalent of $uesp</td>
</tr>
<tr>
<td>$pc</td>
<td>Program counter; equivalent of $eip</td>
</tr>
<tr>
<td>$fp</td>
<td>Frame pointer; equivalent of $ebp</td>
</tr>
<tr>
<td>$ps</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ax</td>
<td>General register</td>
</tr>
<tr>
<td>$cx</td>
<td>General register</td>
</tr>
<tr>
<td>$dx</td>
<td>General register</td>
</tr>
<tr>
<td>$bx</td>
<td>General register</td>
</tr>
<tr>
<td>$si</td>
<td>Source index register</td>
</tr>
<tr>
<td>$di</td>
<td>Destination index register</td>
</tr>
</tbody>
</table>
Register Description

$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:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$al</td>
<td>Lower (right) half of register $ax</td>
</tr>
<tr>
<td>$ah</td>
<td>Higher (left) half of register $ax</td>
</tr>
<tr>
<td>$cl</td>
<td>Lower (right) half of register $cx</td>
</tr>
<tr>
<td>$ch</td>
<td>Higher (left) half of register $cx</td>
</tr>
<tr>
<td>$dl</td>
<td>Lower (right) half of register $dx</td>
</tr>
<tr>
<td>$dh</td>
<td>Higher (left) half of register $dx</td>
</tr>
<tr>
<td>$bl</td>
<td>Lower (right) half of register $bx</td>
</tr>
<tr>
<td>$bh</td>
<td>Higher (left) half of register $bx</td>
</tr>
</tbody>
</table>

Registers for the 80387 are:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fctrl</td>
<td>Control register</td>
</tr>
<tr>
<td>$fstat</td>
<td>Status register</td>
</tr>
<tr>
<td>$ftag</td>
<td>Tag register</td>
</tr>
<tr>
<td>$fip</td>
<td>Instruction pointer offset</td>
</tr>
<tr>
<td>$fcs</td>
<td>Code segment selector</td>
</tr>
<tr>
<td>$fopoff</td>
<td>Operand pointer offset</td>
</tr>
<tr>
<td>$fopsel</td>
<td>Operand pointer selector</td>
</tr>
<tr>
<td>$st0 through $st7</td>
<td>Data registers</td>
</tr>
</tbody>
</table>

**AMD64 Register Information**

The following register information is for AMD64 architecture:
<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>rbx</td>
<td>General purpose register - callee-saved</td>
</tr>
<tr>
<td>rcx</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>rdx</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>rbp</td>
<td>General purpose register - stack management/frame pointer</td>
</tr>
<tr>
<td>rsi</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>rdi</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>rsp</td>
<td>General purpose register - stack management/stack pointer</td>
</tr>
<tr>
<td>r8</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>r9</td>
<td>General purpose register - argument passing for function calls</td>
</tr>
<tr>
<td>r10</td>
<td>General purpose register - temporary</td>
</tr>
<tr>
<td>r11</td>
<td>General purpose register - temporary</td>
</tr>
<tr>
<td>r12</td>
<td>General purpose register - callee-saved</td>
</tr>
<tr>
<td>r13</td>
<td>General purpose register - callee-saved</td>
</tr>
<tr>
<td>r14</td>
<td>General purpose register - callee-saved</td>
</tr>
<tr>
<td>r15</td>
<td>General purpose register - callee-saved</td>
</tr>
<tr>
<td>rflags</td>
<td>Flags register</td>
</tr>
<tr>
<td>rip</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>mmx0/st0</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx1/st1</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx2/st2</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx3/st3</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx4/st4</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx5/st5</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx6/st6</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>mmx7/st7</td>
<td>64-bit media and floating point register</td>
</tr>
<tr>
<td>xmm0</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm1</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>Register</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>xmm2</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm3</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm4</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm5</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm6</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm7</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm8</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm9</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm10</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm11</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm12</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm13</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm14</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>xmm15</td>
<td>128-bit media register</td>
</tr>
<tr>
<td>cs</td>
<td>Segment register</td>
</tr>
<tr>
<td>os</td>
<td>Segment register</td>
</tr>
<tr>
<td>es</td>
<td>Segment register</td>
</tr>
<tr>
<td>fs</td>
<td>Segment register</td>
</tr>
<tr>
<td>gs</td>
<td>Segment register</td>
</tr>
<tr>
<td>ss</td>
<td>Segment register</td>
</tr>
<tr>
<td>fcw</td>
<td>fxsave and fxstor memory image control word</td>
</tr>
<tr>
<td>fsw</td>
<td>fxsave and fxstor memory image status word</td>
</tr>
<tr>
<td>ftw</td>
<td>fxsave and fxstor memory image tag word</td>
</tr>
<tr>
<td>fop</td>
<td>fxsave and fxstor memory image last x87 op code</td>
</tr>
<tr>
<td>frip</td>
<td>fxsave and fxstor memory image 64-bit offset into the code segment</td>
</tr>
<tr>
<td>frdp</td>
<td>fxsave and fxstor memory image 64-bit offset into the date segment</td>
</tr>
<tr>
<td>mxcsr</td>
<td>fxsave and fxstor memory image 128 media instruction control and status register</td>
</tr>
</tbody>
</table>
The fieldsofanadvancedvector (AVX) register (ymm0 through ymm15) can be treated as having 
C int, float, or double types.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxcsr_mask</td>
<td>set bits in mxcsr mask indicate supported feature bits in mxcsr</td>
</tr>
<tr>
<td>ymm0</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm1</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm2</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm3</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm4</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm5</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm6</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm7</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm8</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm9</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm10</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm11</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm12</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm13</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm14</td>
<td>256-bit advanced vector register</td>
</tr>
<tr>
<td>ymm15</td>
<td>256-bit advanced vector register</td>
</tr>
</tbody>
</table>
Using dbx With the Korn Shell

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

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

This chapter is organized into the following sections:

- “ksh-88 Features Not Implemented” on page 237
- “Extensions to ksh-88” on page 238
- “Renamed Commands” on page 238

ksh-88 Features Not Implemented

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

- `set -A name` for assigning values to array `name`
- `set -o` particular options: allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw
- `typeset -l -u -L -R -H` attributes
- Backquote (`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

$[ p-- > flags ]$ 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 allows rebinding of editing functions. You can use the command to display or modify the key bindings for Emacs-style editors and vi-style editors. The syntax of the *bind* command is:

```
bind key=definition        Display the current editing key bindings
bind key                  Bind key to definition
bind key=                  Display the current definition for key
bind key=                  Remove binding of key
bind -m key=definition     Define key to be a macro with definition
```
bind -m Same as bind

where:

key is the name of a key.

definition is the definition of the macro to be bound to the key.

The following are some of the more important default key bindings for EMacs-style editors:

\^A = beginning-of-line  \^B = backward-char
\^D = eot-or-delete  \^E = end-of-line
\^F = forward-char  \^G = abort
\^K = kill-to-eo  \^L = redraw
\^N = down-history  \^P = up-history
\^R = search-history  ^^^ = quote
\^? = delete-char-backward  \^H = delete-char-backward
\^[b = backward-word  \^[d = delete-word-forward
\^[f = forward-word  \^[^H = delete-word-backward
\^[^[ = complete  \^[?] = list-command

The following are some of the more important default key bindings for vi-style editors:

a = append  A = append at EOL
c = change  d = delete
G = go to line  h = backward character
i = insert  I = insert at BOL
j = next line  k = previous line
l = forward line  n = next match
N = prev match  p = put after
P = put before  r = repeat
R = replace  s = substitute
u = undo  x = delete character
Rebinding of Editing Functions

\[
\begin{align*}
X &= \text{delete previous character} & y &= \text{yank} \\
- &= \text{transpose case} & _- &= \text{last argument} \\
* &= \text{expand} & = &= \text{list expansion} \\
- &= \text{previous line} & += &= \text{next line} \\
sp &= \text{forward char} & \# &= \text{comment out command} \\
? &= \text{search history from beginning} & / &= \text{search history from current}
\end{align*}
\]

In insert mode, the following keystrokes are special:

\[
\begin{align*}
^? &= \text{delete character} & ^H &= \text{delete character} \\
^U &= \text{kill line} & ^W &= \text{delete word}
\end{align*}
\]
dbx provides full debugging support for programs that use dynamically-linked, shared libraries, provided that the libraries are compiled using the -g option.

This chapter is organized into the following sections:

- “Dynamic Linker” on page 241
- “Fix and Continue” on page 242
- “Setting Breakpoints in Shared Libraries” on page 242
- “Setting a Breakpoint in an Explicitly Loaded Library” on page 243

### Dynamic Linker

The dynamic linker, also known as rtld, Runtime ld, or ld.so, arranges to bring shared objects (load objects) into an executing application. There are two primary areas where rtld is active:

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

dbx uses the term loadobject to refer to a shared object (.so) or executable (a.out). You can use the loadobject command (see “loadobject Command” on page 319) 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.

The dynamic linker first maps all the shared objects, putting them on the link map. Then, the dynamic linker traverses the link map and executes the .init section for each shared object. The syncrtld event (see “syncrtld” on page 261) occurs between these two phases.

**Procedure Linkage Tables**

Procedure linkage tables (PLTs) are structures used by the rtld to facilitate calls across shared object boundaries. For instance, calls to printf go through this indirect table. The details of how this is done can be found in the generic and processor specific SVR4 ABI reference manuals.

For dbx to handle step and next commands across PLTs, it has to keep track of the PLT table of each load object. The table information is acquired at the same time as the rtld handshake.

**Fix and Continue**

Using fix and continue with shared objects loaded with dlopen() requires a change in how they are opened for fix and continue to work correctly. Use mode RTLD_NOW|RTLD_GLOBAL or RTLD_LAZY|RTLD_GLOBAL.

**Setting Breakpoints in Shared Libraries**

To set a breakpoint in a shared library, dbx needs to know that a program will use that library when it runs, and dbx needs to load the symbol table for the library. To determine which libraries a newly-loaded program will use when it runs, dbx executes the program just long enough for the runtime linker to load all of the starting libraries. dbx then reads the list of loaded libraries and kills the process. The libraries remain loaded and you can set breakpoints in them before rerunning the program for debugging.

dbx follows the same procedure for loading the libraries whether the program is loaded from the command line with the dbx command, from the dbx prompt with the debug command, or 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. In specific terms, this means you cannot have dbx stop at _init() in a library loaded by dlopen().
This appendix focuses on dbx usage and commands that change your program or change the behavior of your program when you run it under dbx, as compared to running it without dbx. It is important to understand which commands might make modifications to your program.

The chapter is divided into the following sections:

- “Impacts of Running a Program Under dbx” on page 245
- “Commands That Alter the State of the Program” on page 246

**Impacts of Running a Program Under dbx**

You use dbx to observe a process, and the observation should not perturb the process. However, on occasion, you might drastically modify the state of the process. And sometimes plain observation can perturb execution and cause bug symptoms to come and go mysteriously.

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

- You might have forgotten to take out a -C or disable RTC. Having the RTC support library librtc.so loaded into a program can cause the program to behave differently.

- Your dbx initialization scripts might have some environment variables set that you’ve forgotten about. The stack base starts at a different address when running under dbx. This is also different based on your environment and the contents of argv[], forcing local variables to be allocated differently. If they’re not initialized, they will get different random numbers. This problem can be detected using runtime checking.

- The program does not initialize memory allocated with malloc() before use; a situation similar to the previous one. This problem can be detected using runtime checking.

- dbx has to catch LWP creation and dlopen events, which might affect timing-sensitive multithreaded applications.
- `dbx` does context switching on signals, so if your application makes heavy use of signals, things might work differently.
- Your program might be expecting that `mmap()` always returns the same base address for mapped segments. Running under `dbx` perturbs the address space sufficiently to make it unlikely that `mmap()` returns the same address as when the program is run without `dbx`. To determine if this is a problem, look at all uses of `mmap()` and ensure that the address returned is used by the program, rather than a hard-coded address.
- If the program is multithreaded, it might contain data races or be otherwise dependent upon thread scheduling. Running under `dbx` perturbs thread scheduling and may cause the program to execute threads in a different order than normal. To detect such conditions, use `lock_lint`.

Otherwise, determine whether running with `adb` or `truss` causes the same problems.

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

### Commands That Alter the State of the Program

The following commands might make modifications to your program:

#### `assign` Command

The `assign` command assigns the value of the `expression` to `variable`. Using it in `dbx` permanently alters the value of `variable`.

`assign variable = expression`

#### `pop` Command

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

- `pop`  Pop current frame.
- `pop number`  Pop `number` frames.
- `pop -t 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 333.”
**call Command**

When you use the `call` command in dbx, you call a procedure and the procedure performs as specified:

\[ \text{call proc([params])} \]

The procedure could modify something in your program. dbx is making the call as if you had written it into your program source.

For more information, see “**call Command**” on page 279.

**print Command**

To print the value of the expressions, type:

\[ \text{print expression, ...} \]

If an expression has a function call, the same considerations apply as with the “**call Command**” on page 279 command. With C++, you should also be careful of unexpected side effects caused by overloaded operators.

For more information, see “**print Command**” on page 333.

**when Command**

The `when` command has a general syntax as follows:

\[ \text{when event-specification [modifier] \{command; ... \}} \]

When the event occurs, the `commands` are executed.

When you get to a line or to a procedure, a command is performed. Depending upon which command is issued, this could alter your program state.

For more information, see “**when Command**” on page 371.

**fix Command**

You can use the `fix` command to make immediate changes to your program:

\[ \text{fix} \]
Although it is a very useful tool, the fix command recompiles modified source files and dynamically links the modified functions into the application.

Make sure to check the restrictions for fix and continue. See “Memory Leak (mel) Error” on page 150.

For more information, see “fix Command” on page 307.

**cont at Command**

The cont at command alters the order in which the program runs. Execution is continued at line line. id is required if the program is multithreaded.

```
cont at line [ id ]
```

This could change the outcome of the program.
Event management refers to the capability of dbx to perform actions when events take place in the program being debugged. When an event occurs, dbx allows you to stop a process, execute arbitrary commands, or print information. The simplest example of an event is a breakpoint. Examples of other events are faults, signals, system calls, calls to `dlopen()`, and data changes (see "Setting Data Change Breakpoints" on page 91).

This appendix is organized into the following sections:

- “Event Handlers” on page 249
- "Event Safety” on page 250
- “Creating Event Handlers” on page 251
- “Manipulating Event Handlers” on page 251
- “Using Event Counters” on page 252
- “Setting Event Specifications” on page 252
- “Event Specification Modifiers” on page 262
- “Parsing and Ambiguity” on page 264
- “Using Predefined Variables” on page 265
- “Event Handler Examples” on page 268

**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 252.) 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 262.) 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.

**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 85.

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 241)
- The beginning and end of processes
- Following the `fork()` function and the `exec()` function (see "Following the fork Function" on page 164 and "Following the exec Function" on page 164)
- 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:...

```
stopped in munmap at 0xff3d503c 0xff3d503c: munmap+0x0004: ta %icc,0x00000008
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.

Creating Event Handlers

Use the when command, stop command, and trace command to create event handlers. (For detailed information, see “when Command” on page 371, “stop Command” on page 349, and “trace Command” on page 360.)

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 252).

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

Every command returns a number known as a handler id (hid). You can access this number using the predefined variable $newhandlerid.

Manipulating Event Handlers

You can use the following commands to manipulate event handlers. For more information on each command, see the cited section.

- status – lists handlers (see “status Command” on page 346).
- delete – deletes all handlers including temporary handlers (see “delete Command” on page 299).
- clear – deletes handlers based on breakpoint position (see “clear Command” on page 284).
- handler-enable – enables handlers (see “handler Command” on page 311).
- handler-disable – disables handlers.
- cancel – cancels signals and lets the process continue (see “cancel Command” on page 281).
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 (see "-count n-count infinity" on page 263). Otherwise, use the handler command to individually manipulate event handlers.

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

Setting Event Specifications

Event specifications are used by the stop 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 (see "stop Command" on page 349, "trace Command" on page 360, and "when Command" on page 371).

Breakpoint Event Specifications

A breakpoint is a location where an action occurs, at which point the program stops executing. The following are event specifications for breakpoint events.

```
in function
```

The function has been entered, and the first line is about to be executed. The first executable code after the prolog is used as the actual breakpoint location. This may be a line where a local variable is being initialized. In the case of C++ constructors, execution stops after all base class constructors have executed. If the -instr modifier is used (see "-instr" on page 263), it is the first instruction of the function about to be executed. The function specification can take a formal parameter signature to help with overloaded function names or template instance specification. For example:

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

**Note** – Do not confuse in function with the -in function modifier.
at [filename:] line_number
The designated line is about to be executed. If you specify filename, then the designated line in the specified file is about to be executed. The file name can be the name of a source file or an object file. Although quotation marks are not required, they may be necessary if the file name contains special characters. If the designated line is in template code, a breakpoint is placed on all instances of that template.

at address_expression
The instruction at the given address is about to be executed. This event is available only with the stopi command (see "stopi Command" on page 353) or with the -instr event modifier (see "-instr" on page 263).

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 function
Equivalent to in function for all overloaded functions named function or all template instantiations thereof.

inmember function inmethod function
Equivalent to in function or the member function named function for every class.
\textbf{inclass} \textit{classname} \textbf{[-recurse | -norecurse]}

Equivalent to \textit{in function} for all member functions that are members of \textit{classname}, but not any of the bases of \textit{classname}. \textit{-norecurse} is the default. If \textit{-recurse} is specified, the base classes are included.

\textbf{inobject} \textit{object-expression} \textbf{[-recurse | -norecurse]}

A member function called on the specific object at the address denoted by \textit{object-expression} has been called. \textit{stop inobject ox} is roughly equivalent to the following, but unlike \textit{inclass}, bases of the dynamic type of \textit{ox} are included. \textit{-recurse} is the default. If \textit{-norecurse} is specified, the base classes are not included.

\begin{verbatim}
stop inclass dynamic_type(ox) -if this==ox
\end{verbatim}

\section*{Data Change Event Specifications}

The following are event specifications for events that involve access or change to the contents of a memory address.

\textbf{access} \textit{mode address-expression [, byte-size-expression]}

The memory specified by \textit{address-expression} has been accessed.

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

- \textit{r} The memory at the specified address has been read.
- \textit{w} The memory has been written to.
- \textit{x} The memory has been executed.

\textit{mode} can also contain either of the following:

- \textit{a} Stops the process after the access (default).
- \textit{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.

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

\begin{verbatim}
stop access w 0x5678, sizeof(Complex)
\end{verbatim}

The access command has the limitation that no two matched regions may overlap.
Note – The access event specification is a replacement for the modify event specification.

**change variable**
The value of variable has changed. The change event is roughly equivalent to:

```bash
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 (see “access mode address-expression [, byte-size-expression]” on page 254).

The first time variable is checked causes one event, even though no change is detected. This first event provides access to the initial value of variable. Subsequent detected changes in the value of variable trigger additional events.

**cond condition-expression**
The condition denoted by condition-expression evaluates to true. You can specify any expression for condition-expression, but it must evaluate to an integral type. The cond event is roughly equivalent to:

```bash
stop step -if conditional_expression
```

**System Event Specifications**
The following are event specifications for system events.

**dlopen [ lib-path ]**
**dlclose [ lib-path ]**
These events occur after a dlopen() 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 “+” or a “-”, indicating whether the list of libraries is being added or deleted.

*lib-path* is the name of a shared library. If it is specified, the event occurs only if the given library was loaded or unloaded. In that case, $dlobj contains the name of the library. $dllist is still available.
If *lib-path* begins with a `/`, a full string match is performed. Otherwise, only the tails of the paths are compared.

If *lib-path* is not specified, then the events always occur whenever there is any `dl`-activity. In this case, `$dlobj` is empty but `$dllist` is valid.

**fault**

The *fault* event occurs when the specified fault is encountered. The faults are architecture-dependent. The following set of faults known to `dbx` is defined in the `proc(4)` man page.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLTILL</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>FLTPRIV</td>
<td>Privileged instruction</td>
</tr>
<tr>
<td>FLTBP’T</td>
<td>Breakpoint trap</td>
</tr>
<tr>
<td>FLTTRACE’</td>
<td>Trace trap (single step)</td>
</tr>
<tr>
<td>FLTACCESS</td>
<td>Memory access (such as alignment)</td>
</tr>
<tr>
<td>FLTBOUNDS</td>
<td>Memory bounds (invalid address)</td>
</tr>
<tr>
<td>FLTIOVF</td>
<td>Integer overflow</td>
</tr>
<tr>
<td>FLTIZDIV</td>
<td>Integer zero divide</td>
</tr>
<tr>
<td>FLTPE</td>
<td>Floating-point exception</td>
</tr>
<tr>
<td>FLTSTACK</td>
<td>Irrecoverable stack fault</td>
</tr>
<tr>
<td>FLTDPAGE</td>
<td>Recoverable page fault</td>
</tr>
<tr>
<td>FLTMATCH’</td>
<td>Watchpoint trap</td>
</tr>
<tr>
<td>FLTCPCOVF</td>
<td>CPU performance counter overflow</td>
</tr>
</tbody>
</table>

**Note** – *BPT*, *TRACE*, and *BOUNDS* are used by `dbx` to implement breakpoints and single-stepping. Handling them might interfere with how `dbx` works.

**Note** – *FLTBPT* and *FLTTRACE* are ignored because they would interfere with basic `dbx` functionality such as breakpoints and single-stepping (see “Event Safety” on page 250).
These faults are taken from `/sys/fault.h`. `fault` can be any of those listed above, in uppercase or lowercase, with or without the FLT- prefix, or the actual numerical code.

**Note** - The `fault` event is not available on Linux platforms.

**lwp_exit**
The `lwp_exit` event occurs when `lwp` has been exited. `$lwp` contains the id of the exited LWP (lightweight process) for the duration of the event handler.

**Note** - The `lwp_exit` event is not available on Linux platforms.

**sig signal**
The `sig signal` event occurs when the signal is first delivered to the program being debugged. `signal` can be either a decimal number or the signal name in uppercase or lowercase; the prefix is optional. This is completely independent of the `catch` command and `ignore` command, although the `catch` command can be implemented as follows:

```bash
function simple_catch {
    when sig $1 {
        stop;
        echo Stopped due to $sigstr $sig
    }
}
```

**Note** - When the `sig` event is received, the process has not seen it yet. Only if you continue the process with the specified signal is the signal forwarded to it.

**sig signal sub-code**
When the specified signal with the specified `sub-code` is first delivered to the child, the `sig signal` `sub-code` event occurs. As with signals, you can type the `sub-code` as a decimal number, in uppercase or lowercase; the prefix is optional.

**sysin code | name**
The specified system call has just been initiated, and the process has entered kernel mode.

The concept of system call supported by `dbx` is that provided by traps into the kernel as enumerated in `/usr/include/sys/syscall.h`.

This is not the same as the ABI notion of system calls. Some ABI system calls are partially implemented in user mode and use non-ABI kernel traps. However, most of the generic system calls (the main exception being signal handling) are the same between `syscall.h` and the ABI.
Note – The **sysin** event is not available on Linux platforms.

Note – The list of kernel system call traps in `/usr/include/sys/syscall.h` is part of a private interface in the Solaris 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 Solaris OS that `dbx` supports. It is unlikely that the names supported by `dbx` exactly match those of any particular release of the 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 code | name**
The specified system call is finished, and the process is about to return to user mode.

Note – The **sysout** event is not available on Linux platforms.

**sysin | sysout**
Without arguments, all system calls are traced. Certain `dbx` features, for example, the `modify` event and runtime checking, cause the child to execute system calls for its own purposes and show up if traced.

### Execution Progress Event Specifications

The following are event specifications for events pertaining to execution progress.

**exit exitcode**
The `exit` event occurs when the process has exited.

**next**
The `next` event is similar to the `step` event except that functions are not stepped into.

**returns**
The `returns` event is a breakpoint at the return point of the current `visited` function. The visited function is used so that you can use the `returns` event specification after giving a number of `step up` commands. The `returns` event is always `-temp` and can only be created in the presence of a live process.
returns function

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

- SPARC based systems: $o0
- x86 based systems: $eax
- x64 based systems: $rax, $rdx

The event is roughly equivalent to:

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

step

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

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

When enabling a step event, you instruct dbx to single step automatically next time the cont command is used.

Note – The step (and next) events do not occur upon the termination of the step command. The step command is implemented in terms of the step event roughly as follows: alias step=\"when step -temp \{ whereami; stop; \}; cont\"*

Other Event Specifications

The following are event specifications for other types of events.

attach

dbx has successfully attached to a process.

detach

dbx has successfully detached from the program being debugged.
**lastrites**

The process being debugged is about to expire, which can happen for the following reasons:

- The `exit(2)` system call has been called. (This happens either through an explicit call or when `main()` returns.)
- A terminating signal is about to be delivered.
- The process is being killed by the `kill` command.

The final state of the process is usually, but not always, available when this event is triggered, giving you your last opportunity to examine the state of the process. Resuming execution after this event terminates the process.

**Note** – The lastrites event is not available on Linux platforms.

**proc_gone**

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

**prog_new**

The `prog_new` event occurs when a new program has been loaded as a result of `follow exec`.

**Note** – Handlers for this event are always permanent.

**stop**

The process has stopped. The `stop` event occurs whenever the process stops such that the user receives a prompt, particularly in response to a `stop` handler. For example, the following commands are equivalent:

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

**sync**

The process being debugged has just been executed with `exec()`. All memory specified in `a.out` is valid and present, but preloaded shared libraries have not been loaded. For example, `printf`, although available to `dbx`, has not been mapped into memory.

A `stop` on this event is ineffective; however, you can use the `sync` event with the `when` command.
Note – The sync event is not available on Linux platforms.

**syncrtld**

The syncrtld event occurs after a sync (or attach if the process being debugged has not yet processed shared libraries). It executes after the dynamic linker startup code has executed and the symbol tables of all preloaded shared libraries have been loaded, but before any code in the .init section has run.

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

**thr_create [thread_id]**

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**

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

**throw**

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

Note – The throw event is not available on Linux platforms.

**throw type**

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

**throw -unhandled**

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

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

**timer seconds**

The `timer` event occurs when the program being debugged has been running for `seconds`. The timer used with this event is shared with `collect` command. The resolution is in milliseconds, so a floating point value for `seconds`, for example `0.001`, is acceptable.

### Event Specification Modifiers

An event specification modifier sets additional attributes of a handler, the most common kind being event filters. Modifiers must appear after the keyword portion of an event specification. A modifier begins with a dash (`-`). The following are the valid event specification modifiers.

**-if condition**

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

If the `-if` modifier is used with an event that has an associated singular source location, such as `in` or `at`, `condition` is evaluated in the scope corresponding to that location. Otherwise, qualify it with the desired scope.

Macro expansion is performed on the condition according to same conventions as with the `print` command.

**-resumeone**

The `-resumeone` modifier can be used with the `-if` modifier in an event specification for a multithreaded program, and causes only one thread to be resumed if the condition contains function calls. For more information, see "Using a Filter With a Conditional Event" on page 95.

**-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.
The `disable` modifier creates the handler in the disabled state.

**-count n**

**-count infinity**

The `-count n` and `-count infinity` modifiers have the handler count from 0 (see “Using Event Counters” on page 252). 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 296) or the `attach -r` command (see “attach Command” on page 278).

**-temp**

Creates a temporary handler. Once the event has occurred it is automatically deleted. By default, handlers are not temporary. If the handler is a counting handler, it is automatically deleted only when the count reaches 0 (zero).

Use the `delete -temp` command to delete all temporary handlers.

**-instr**

Makes the handler act at an instruction level. This event replaces the traditional ‘i’ suffix of most commands. It usually modifies two aspects of the event handler:

- Any message prints assembly-level rather than source-level information.
- The granularity of the event becomes instruction level. For instance, `step -instr` implies instruction-level stepping.
**-thread thread_id**

The action is executed only if the thread that caused the event matches thread_id. The specific thread you have in mind might be assigned a different thread_id from one execution of the program to the next.

**-lwp lwp_id**

The action is executed only if the thread that caused the event matches lwp_id. The action is executed only if the thread that caused the event matches lwp_id. The specific thread you have in mind might be assigned a different lwp_id from one execution of the program to the next.

**-hidden**

Hides the handler in a regular status command. Use status -h to see hidden handlers.

**-perm**

Normally all handlers are thrown away when a new program is loaded. Using the -perm modifier retains the handler across debugging sessions. A plain delete command does not delete a permanent handler. Use delete -p to delete a permanent handler.

### Parsing and Ambiguity

The syntax for event specifications and modifiers is:

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

Expressions can have spaces embedded in them, causing ambiguous situations. For example, consider the following two commands:

```
when a -temp
when a -temp
```

In the first example, even though the application might have a variable named temp, the dbx parser resolves the event specification in favor of -temp being a modifier. In the second example, a -temp is collectively passed to a language-specific expression parser. There must be variables named a and temp or an error occurs. Use parentheses to force parsing.
Using Predefined Variables

Certain read-only ksh predefined variables are provided. The following variables are always valid:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ins</td>
<td>Disassembly of the current instruction.</td>
</tr>
<tr>
<td>$lineno</td>
<td>Current line number in decimal.</td>
</tr>
<tr>
<td>$vlineno</td>
<td>Current &quot;visiting&quot; line number in decimal.</td>
</tr>
<tr>
<td>$line</td>
<td>Contents of the current line.</td>
</tr>
<tr>
<td>$func</td>
<td>Name of the current function.</td>
</tr>
<tr>
<td>$vfunc</td>
<td>Name of the current &quot;visiting&quot; function.</td>
</tr>
<tr>
<td>$class</td>
<td>Name of the class to which $func belongs.</td>
</tr>
<tr>
<td>$vclass</td>
<td>Name of the class to which $vfunc belongs.</td>
</tr>
<tr>
<td>$file</td>
<td>Name of the current file.</td>
</tr>
<tr>
<td>$vfile</td>
<td>Name of the current file being visited.</td>
</tr>
<tr>
<td>$loadobj</td>
<td>Name of the current loadable object.</td>
</tr>
<tr>
<td>$vloadobj</td>
<td>Name of the current loadable object being visited.</td>
</tr>
<tr>
<td>$scope</td>
<td>Scope of the current PC in back-quote notation.</td>
</tr>
<tr>
<td>$vscope</td>
<td>Scope of the visited PC in back-quote notation.</td>
</tr>
<tr>
<td>$funcaddr</td>
<td>Address of $func in hex.</td>
</tr>
<tr>
<td>$caller</td>
<td>Name of the function calling $func.</td>
</tr>
<tr>
<td>$dllist</td>
<td>After a dlopen or dlclose event, contains the list of load objects just loaded or unloaded. The first word of dllist is a &quot;+&quot; or a &quot;-&quot; depending on whether a dlopen or a dlclose has occurred.</td>
</tr>
<tr>
<td>$newhandlerid</td>
<td>ID of the most recently created handler. This variable has an undefined value after any command that deletes handlers. Use the variable immediately after creating a handler. dbx cannot capture all of the handler IDs for a command that creates multiple handlers.</td>
</tr>
<tr>
<td>$firedhandlers</td>
<td>List of handler ids that caused the most recent stoppage. The handlers on the list are marked with &quot;*&quot; in the output of the status command.</td>
</tr>
<tr>
<td>$proc</td>
<td>Process ID of the current process being debugged.</td>
</tr>
<tr>
<td>$lwp</td>
<td>Lwp ID of the current LWP.</td>
</tr>
</tbody>
</table>
Variable Definition

$thread Thread ID of the current thread.

$newlwp lwp ID of a newly created LWP.

$newthread Thread ID of a newly created thread.

$prog Full pathname of the program being debugged.

$oprog Previous value of $prog, which is used to get back to what you were debugging following an exec(). When $prog reverts to "-", while $prog is expanded to a full path name, $oprog contains the program path as specified on the command line or to the debug command. If exec() is called more than once, there is no way to return to the original program.

$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.

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

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

Variables Valid for when Command

The following variables are valid only within the body of a when command.

$handlerid

During the execution of the body, $handlerid is the id of the when command to which the body belongs. These commands are equivalent:

when X -temp { do_stuff; }
when X { do_stuff; delete $handlerid; }
**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 B-1**  Variables Valid for `sig` Event

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sig</td>
<td>Signal number that caused the event</td>
</tr>
<tr>
<td>$sigstr</td>
<td>Name of $sig</td>
</tr>
<tr>
<td>$sigcode</td>
<td>Subcode of $sig if applicable</td>
</tr>
<tr>
<td>$sigcodestr</td>
<td>Name of $sigcode</td>
</tr>
<tr>
<td>$sigsender</td>
<td>Process ID of sender of the signal, if appropriate</td>
</tr>
</tbody>
</table>

**TABLE B-2**  Variable Valid for `exit` Event

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$exitcode</td>
<td>Value of the argument passed to <code>exit(2)</code> or <code>exit(3)</code> or the return value of <code>main</code></td>
</tr>
</tbody>
</table>

**TABLE B-3**  Variable Valid for `dlopen` and `dlclose` Events

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dlobj</td>
<td>Pathname of the load object dlopened or dlclosed</td>
</tr>
</tbody>
</table>

**TABLE B-4**  Variables Valid for `sysin` and `sysout` Events

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$syscode</td>
<td>System call number</td>
</tr>
<tr>
<td>$sysname</td>
<td>System call name</td>
</tr>
</tbody>
</table>

**TABLE B-5**  Variable Valid for `proc_gone` Events

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$reason</td>
<td>One of signal, exit, kill, or detach</td>
</tr>
</tbody>
</table>

**TABLE B-6**  Variables Valid for `thr_create` Event

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$newthread</td>
<td>Id of the newly created thread (for example, t@5)</td>
</tr>
</tbody>
</table>
### Event Handler Examples

The following are some examples of setting event handlers.

#### Setting a Breakpoint for Store to an Array Member

To set a data change breakpoint on `array[99]`, type:

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
```

Running: watch.x2

```
watchpoint array[99] (0x2ca88[4]) at line 22 in file "watch.c"
  22 array[i] = i;
```

#### Implementing a Simple Trace

To implement a simple trace, type:

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

#### Enabling a Handler While Within a Function (in function)

To enable a handler while within a function, type:

```
<dbx> trace step -in foo
```

This is equivalent to:
Determining the Number of Lines Executed

To see how many lines have been executed in a small program, type:

```
(dbx) stop step -count infinity       # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

The program never stops, the program terminates. The number of lines executed is 133. This process is very slow. It is most useful with breakpoints on functions that are called many times.

Determining the Number of Instructions Executed by a Source Line

To count how many instructions a line of code executes, type:

```
(dbx) ...                        # get to the line in question
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 instructions were executed
```

If the line you are stepping over makes a function call, the lines in the function are counted as well. You can use the next event instead of step to count instructions, excluding called functions.

Enabling a Breakpoint After an Event Occurs

Enable a breakpoint only after another event has occurred. For example, if your program begins to execute incorrectly in function hash, but only after the 1300'th symbol lookup, you would type:

```
# 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"; }
}
```
```
(dbx) when in lookup -count 1300 {
  stop in hash
  hash_bpt=$newhandlerid
  when proc_gone -temp { delete $hash_bpt; }
}
```

Note – $newhandlerid is referring to the just executed stop in command.

## Resetting Application Files for replay

If your application processes files that need to be reset during a replay, you can write a handler to do that each time you run the program:

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database... # during which database gets clobbered
(dbx) save
... # implies a RUN, which implies the SYNC event which
(dbx) restore       # causes regen to run
```

## Checking Program Status

To see quickly where the program is while it is running, type:

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

Then type `^C` to see a stack trace of the program without stopping it.

This is basically what the collector hand sample mode does (and more). Use SIGQUIT (`^\` ) to interrupt the program because `^C` is now used up.

## Catch Floating Point Exceptions

To catch only specific floating point exceptions, for example, IEEE underflow, type:

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

**Macro Expansion**

Macro expansion is applied to both the variable and the expression in an `assign` command (see “assign Command” on page 277).

In the `call` command (see “call Command” on page 279), macro expansion is applied to the name of the function being called as well as to the parameters being passed.

The `macro` command (see “macro Command” on page 324) takes 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 (see “whatis Command” on page 369) a macro, it shows the macro’s definition. For example:

```
(dbx) whatis B
    #define B(x) b(x)
```

If you give the `which` command (see “which Command” on page 375) 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 (see “`whereis Command` on page 375”) 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 `dbx` environment 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 `dbx` environment 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 recreate definitions by skimming the source file and its include files. Accurate recreation depends on access to the original sources and include files. It also depends on the availability of the pathname to the compiler used, and on compiler options like `-D` and `-I`. This information is available in both DWARF and stabs formats from Oracle Solaris Studio compiler, but not from GNU compilers. See “Skimming Errors” on page 274 and “Using the `pathmap` Command To Improve Skimming” on page 275 for information on ensuring successful skimming.

Which one of the two methods `dbx` uses to recognize macro definitions is controlled by the `dbx` environment variable `macro_source` (see Table 3–1 in Chapter 3, “Customizing `dbx`”).

There are several factors to take into account in choosing which method you want `dbx` to use.

**Compiler and Compiler Options**

One factor 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.
## Tradeoffs in Functionality

Another factor to take into account 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 evaluation 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 it might be better to compile with the `-g3` option than to depend 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 (see “Using the `pathmap` Command To Improve Skimming” on page 275.

### TABLE C–1  Macro Definition Methods Available for Various Build Options

<table>
<thead>
<tr>
<th>Compiler</th>
<th><code>-g</code> option</th>
<th>Debug Information Format</th>
<th>Methods That Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle Solaris Studio</td>
<td><code>-g</code></td>
<td>DWARF</td>
<td>Skimming</td>
</tr>
<tr>
<td>Oracle Solaris Studio</td>
<td><code>-g</code></td>
<td>stabs</td>
<td>Skimming</td>
</tr>
<tr>
<td>Oracle Solaris Studio</td>
<td><code>-g3</code></td>
<td>DWARF</td>
<td>Skimming and from compiler</td>
</tr>
<tr>
<td>Oracle Solaris Studio</td>
<td><code>-g3</code></td>
<td>stabs</td>
<td>Skimming ( <code>-g3</code> option with <code>-xdebugformat=stabs</code> option is not supported)</td>
</tr>
<tr>
<td>gnu</td>
<td><code>-g</code></td>
<td>DWARF</td>
<td>Neither</td>
</tr>
<tr>
<td>gnu</td>
<td><code>-g</code></td>
<td>stabs</td>
<td>N/A</td>
</tr>
<tr>
<td>gnu</td>
<td><code>-g3</code></td>
<td>DWARF</td>
<td>From compiler</td>
</tr>
<tr>
<td>gnu</td>
<td><code>-g3</code></td>
<td>stabs</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The `pathmap` command also helps facilitate file access during skimming. It works for your program’s source files and include files. But it might not work for system include files because `/usr/include` is not usually available through NFS, so macro definitions 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**

- Though 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.
- 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 dbx` environment 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 Solaris 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, i.e., 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 \texttt{pathmap} Command To Improve Skimming

If you move your source files after compiling, or 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 76, 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 \texttt{pathmap} command to help the macro skimmer locate include directories. Imagine, for example, that you compile with the option \\
\texttt{-I/export/home/proj1/include} and have the statement \texttt{#include "module1/api.h"} in your code. Then, if you rename \texttt{proj1} to \texttt{proj2}, the following \texttt{pathmap} command will help the macro skimmer locate your files:

\texttt{pathmap /export/home/proj1 /export/home/proj2}

The \texttt{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 \texttt{pathmap} command to make changes in a pathmapping that are immediately effective.

The \texttt{pathmap} command helps \texttt{dbx} find the correct files when you build on one machine and debug on another. However, system include files such as \texttt{/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. And the value of system-specific and system-dependent macros might not be same on the debug machine as on the build machine.

If the \texttt{pathmap} command does not solve your skimming problems, consider compiling your code with the \texttt{-g3} option and setting the \texttt{macro_source dbx} environment variable to \texttt{skim_unless Compiler} or \texttt{compiler}. 

This appendix gives detailed syntax and functional descriptions of all of the dbx commands.

**assign Command**

In native mode, the `assign` command assigns a new value to a program variable. In Java mode, the `assign` command assigns a new value to a local variable or parameter.

**Native Mode Syntax**

```
assign variable = expression
```

where:

- `expression` is the value to be assigned to `variable`.

**Java Mode Syntax**

```
assign identifier = expression
```

where:

- `class_name` is the name of a Java class, using either the package path (using period (.) as a qualifier; for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers; for example, `
#test1-extra/T1$Inner`). Enclose `class_name` in quotation marks if you use the $ qualifier.

- `expression` is a valid Java expression.

- `field_name` is the name of a field in the class.
identifier is a local variable or parameter, including this, the current class instance variable 
(object_name.field_name) or a class (static) variable (class_name.field_name).

object_name is the name of a Java object.

attach Command

The attach command attaches dbx to a running process, stopping execution and putting the 
program under debugging control. It has identical syntax and identical functionality in native 
mode and Java mode.

Syntax

attach process_id

Begin debugging the program with process ID
process_id. dbx finds the program using /proc.

attach -p process_id program_name

Begin debugging program_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.

▼ 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. libdbx_agent.so:

- If you are using the 32-bit version of the JVM software on a system running the Solaris OS,
add /installation_directory/libdbx_agent.so to LD_LIBRARY_PATH.
- If you are using the 64-bit version of the JVM software on a SPARC based system running
the Solaris OS, add /installation_directory/lib/v9/libdbx_agent.so to LD_LIBRARY_PATH.
If you are using the 64-bit version of the JVM software on an x64 based system running the Solaris OS or the Linux OS, add /installation_directory/lib/amd64/libdbx_agent.so to LD_LIBRARY_PATH.

The installation_directory is the location where the Oracle Solaris Studio software is installed.

2 Start your Java application by typing:
java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent myclass.class

3 Then you can attach to the process by starting dbx with the process id:
dbx -process_id

**bsearch Command**

The bsearch command searches backward in the current source file. It is valid only in native mode.

**Syntax**

\[
\text{bsearch} \text{ string} \quad \text{Search backward for string in the current file.}
\]

\[
\text{bsearch} \quad \text{Repeat search, using the last search string.}
\]

where:

\[
\text{string} \quad \text{is a character string.}
\]

**call Command**

In native mode, the call command calls a procedure. In Java mode, the call command calls a method.

**Native Mode Syntax**

\[
\text{call} \text{ procedure ([parameters]) [-lang language] [-resumeone] [-m] [+m]}
\]

where:

\[
\text{language} \quad \text{is the language of the called procedure.}
\]
*procedure* is the name of the procedure.

*parameters* are the procedure's parameters.

You can also use the `call` command to call a function; to see the return value use the `print` command (see “print Command” on page 333).

Occasionally the called function hits a breakpoint. You can choose to continue using the `cont` command (see “cont Command” on page 292), or abort the call by using `pop -c` (see “pop Command” on page 333). The latter is useful also if the called function causes a segmentation fault.

The `-lang` option 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.

The `-resumeone` option resumes only one thread when the procedure is called. For more information, see “Resuming Execution” on page 160.

The `-m` option specifies that macro expansion be applied to the procedure and parameters when the `dbx` environment variable `macro_expand` is set to `off`.

The `+m` option specifies that macro expansion be skipped when the `dbx` environment variable `macro_expand` is set to `on`.

---

### Java Mode Syntax

```java
call [class_name.]object_name.method (parameters)
```

where:

*class_name* is the name of a Java class, using either the package path (using period (.) as a qualifier for example, `test1.extra.T1.Inner`) or the full path name (preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers; for example, `#test1extra/T1$Inner`). Enclose *class_name* in quotation marks if you use the $ qualifier.

*object_name* is the name of a Java object.

*method_name* is the name of a Java method.

*parameters* are the method's parameters.

Occasionally the called method hits a breakpoint. You can choose to continue using the `cont` command (see “cont Command” on page 292, or abort the call by using `pop -c` (see “pop Command” on page 333). The latter is useful also if the called method causes a segmentation fault.
cancel Command

The cancel command cancels the current signal. It is primarily used within the body of a when command (see “when Command” on page 371). It is valid only in native mode.

Signals are normally cancelled when dbx stops because of a signal. If a when command is attached to a signal event, the signal is not automatically cancelled. The cancel command can be used to explicitly cancel the signal.

Syntax

cancel

catch Command

The catch command catches the given 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

check -access

Turn on access checking. RTC reports the following errors:

- baf  Bad free
- duf  Duplicate free
- maf  Misaligned free
- mar  Misaligned read
- maw  Misaligned write
- oom  Out of memory
- 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 dbx environment variable. When set to on access errors are logged to a file (the file name is controlled by the dbx environment variable rtc_error_log_file_name). See “dbxenv Command” on page 296.

By default each unique access error is only reported the first time it happens. Change this behavior using the dbx environment variable rtc_auto_suppress (the default setting of this variable is on). See “dbxenv Command” on page 296.

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

Turn on leak checking. RTC reports the following errors:
Check Command

Possible memory leak - only pointer points in the middle of the block
Possible memory leak - pointer to the block exists only in register
Memory leak - no pointers to the block

With leak checking turned on, you get an automatic leak report when the program exits. All leaks including possible leaks are reported at that time. By default, a non-verbose report is generated (controlled by the dbx environment variable rtc_mel_at_exit). However, you can ask for a leak report at any time (see “showleaks Command” on page 344).

-matchm is used for combining leaks; if the call stack at the time of allocation for two or more leaks matches n frames, then these leaks are reported in a single combined leak report.

The default value of n is 8 or the value of m (whichever is larger). Maximum value of n is 16. The default value of m is 3 for C++, and 2 otherwise.

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

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 (controlled by the dbx environment variable rtc_biu_at_exit) At any time during program execution you can see where the memory in your program has been allocated (see “showmemuse Command” on page 345).

-matchm is used for combining these reports; if the call stack at the time of allocation for two or more leaks matches m frames, then these are reported in a single combined memory leak report.

The default value of n is 8 or the value of m (whichever is larger). Maximum value of n is 16. The default value of m is 3 for C++, and 2 otherwise. See check -leaks as well.

check -all [-frames n] [-match m]

Equivalent to check -access and check -memuse [-frames n] [-match m]

The value of the dbx environment variable rtc_biu_at_exit is not changed with check -all, so by default no memory use report is generated at exit. See “dbx Command” on page 293 for the description of the rtc_biu_at_exit environment variable.

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 `-g`. However, symbolic (`-g`) information is sometimes needed to guarantee the correctness of certain errors (mostly read from uninitialized memory). For this reason certain errors (`rui` for `a.out` and `rui + aib + air` for shared libraries) are suppressed if no symbolic information is available. This behavior can be changed by using `suppress` and `unsuppress`.

**clear Command**

The `clear` command clears breakpoints. It is valid only in native mode.

Event handlers created using the `stop` 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 reestablishes all the breakpoints.

**Syntax**

```plaintext
clear  Clear all breakpoints at the current stopping point.
clear line  Clear all breakpoints at line.
clear filename:line  Clear all breakpoints at line in filename.
```

where:

`line` is the number of a source code line.

`filename` is the name of a source code file.
**collector Command**

The collector command collects performance data for analysis by the Performance Analyzer. It is valid only in native mode.

**Syntax**

- `collector command_list` Specify one or more of the collector commands
- `collector archive options` Specify the mode for archiving an experiment when it terminates (see “collector archive Command” on page 286).
- `collector dbxsample options` Control the collection of samples when dbx stops the target process (see “collector dbxsample Command” on page 286).
- `collector disable` Stop data collection and close the current experiment (see “collector disable Command” on page 287).
- `collector enable` Enable the collector and open a new experiment (see “collector enable Command” on page 287).
- `collector heaptrace options` Enable or disable collection of heap tracing data (see “collector heaptrace Command” on page 287).
- `collector hwprofile options` Specify hardware counter profiling settings (see “collector hw_profile Command” on page 287).
- `collector limit options` Limit the amount of profiling data recorded (see “collector limit Command” on page 288).
- `collector mpitrace options` Enables or disables collection of MPI tracing data (see “collector mpitrace Command” on page 288).
- `collector pause` Stop collecting performance data but leave experiment open (see “collector pause Command” on page 289).
- `collector profile options` Specify settings for collecting callstack profiling data (see “collector profile Command” on page 289).
- `collector resume` Start performance data collection after pause (see “collector resume Command” on page 289).
- `collector sample options` Specify sampling settings (see “collector sample Command” on page 289).
- `collector show options` Show current collector settings (see “collector show Command” on page 290).
collector Command

**collector status**
Inquire status about current experiment (see "collector status Command" on page 290).

**collector store options**
Experiment file control and settings (see "collector store Command" on page 291).

**collector syntrace options**
Specify settings for collecting thread synchronization wait tracing data (see "collector syntrace Command" on page 291).

**collector tha options**
Specify settings for collecting thread analyzer data (see "collector tha Command" on page 292).

**collector version**
Report the version of libcollector.so that would be used to collect data (see "collector version Command" on page 292).

where:

**options** are the settings that can be specified with each command.

To start collecting data, type either collector enable.

To turn off data collection, type collector disable.

**collector archive Command**
The collector archive command specifies the archiving mode to be used when the experiment terminates.

**Syntax**

collector archive on|off|copy
By default, normal archiving is used. For no archiving, specify off. To copy loadobjects into the experiment for portability, specify copy.

**collector dbxsample Command**
The collector dbxsample command specifies whether or not to record a sample when the process is stopped by dbx.

**Syntax**

collector dbxsample on|off
By default, a sample is collected when the process is stopped by dbx. To not collect a sample at this time, specify off.
**collector disable Command**

The collector disable command causes the data collection to stop and the current experiment to be closed.

**Syntax**

```
collector disable
```

**collector enable Command**

The collector enable command enables the collector and opens a new experiment.

**Syntax**

```
collector enable
```

**collector heaptrace Command**

The collector heaptrace command specifies options for collecting heap tracing (memory allocation) data.

**Syntax**

```
collector heaptrace on|off
```

By default, heap tracing data is not collected. To collect this data, specify `on`.

**collector hw_profile Command**

The collector hw_profile command specifies options for collecting hardware-counter overflow profiling data.

**Syntax**

```
collector hwprofile on|off
```

By default, hardware-counter overflow profile data is not collected. To collect this data, specify `on`.

```
collector hwprofile list
```

Print out the list of available counters.
collector hwprofile counter name interval [name2 interval2]
 Specify hardware counter 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 to you depends on
the system you are using. Many systems do not support hardware-counter overflow profiling.
On these machines, the feature is disabled.

**collector limit Command**

The collector limit command specifies the experiment file size limit.

**Syntax**

collector limit value | 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 mpitrace Command**

The collector mpitrace command specifies options for collecting MPI tracing data.

**Syntax**

collector mpitrace on|off

By default, MPI tracing data is not collected. To collect this
data, specify on.
**collector pause Command**

The collector pause command causes the data collection to stop but leaves the current experiment open. 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 resum. Data collection can be resumed with the collector resume command (see "collector resume Command" on page 289).

**Syntax**

```
collector pause
```

**collector profile Command**

The collector profile command specifies options for collecting profile data.

**Syntax**

```
collector profile on|off
```

Specify profile data collection mode.

```
collector profile timer interval
```

Specify profile timer period, fixed or floating point, with an optional trailing m for milliseconds or u for microseconds.

**collector resume Command**

The collector resume command causes the data collection to resume after a pause created by the collector pause command (see "collector pause Command" on page 289).

**Syntax**

```
collector resume
```

**collector sample Command**

The collector sample command specifies the sampling mode and the sampling interval.

**Syntax**

```
collector sample periodic|manual
```

Specify sampling mode.

```
collector sample period seconds
```

Specify sampling interval in seconds.
collector sample record [name] — Record a sample with an optional name.

where:

seconds is the length of the sampling interval.

name is the name of the sample.

collector show Command

The collector show command shows the settings of one or more categories of options.

**Syntax**

```
collector show
collector show all
collector show archive
collector show profile
collector show synctrace
collector show hwprofile
collector show heaptrace
collector show limit
collector show mpirace
collector show sample
collector show store
collector show tha
```

collector status Command

The collector status command inquires about the status of the current experiment.

**Syntax**

```
collector status
```
**collector store Command**

The `collector store` command specifies the directory and file name where an experiment is stored.

**Syntax**

```plaintext
collector store directory pathname  Specify directory where experiment is stored.
collector store filename filename  Specify experiment file name.
collector store group string     Specify experiment group name.
```

where:

- `pathname` is the pathname of the directory where an experiment is to be stored.
- `filename` is the name of the experiment file
- `string` is the name of an experiment group.

**collector synctrace Command**

The `collector synctrace` command specifies options for collecting synchronization wait tracing data.

**Syntax**

```plaintext
collector synctrace on|off  By default, thread synchronization wait tracing data is not collected. To collect this data, specify on.
collector synctrace threshold microseconds Specify threshold in microseconds. The default value is 1000.
collector synctrace threshold calibrate  Threshold value will be calculated automatically.
```

where:

- `microseconds` is the threshold below which synchronization wait events are discarded.
**collector tha Command**

**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
cont ... -sig signal
cont ... id
cont at line [id]
cont ... -follow parent|child|both
```

Continue execution. In an MT process all threads are resumed. Use Control-C to stop executing the program.

Continue execution with signal `signal`.

The `id` specifies which thread or LWP to continue.

Continue execution at line `line`. `id` is required if the application is multi-threaded.

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 Solaris Studio IDE.
The `dalias` command defines a dbx-style (csh-style) alias. It is valid only in native mode.

**Syntax**

- `dalias` (dbx alias) List all currently defined aliases.
- `dalias name` List the definition, if any, of alias `name`.
- `dalias name definition` Define `name` to be an alias for `definition`. `definition` may contain white space. A semicolon or newline terminates the definition.

where:

- `name` is the name of an alias
- `definition` is the definition of an alias.

`dbx` accepts the following `csh` history substitution meta-syntax, which is commonly used in aliases:

- `!:<n>`
- `!-<n>`
- `!^`
- `!$`
- `!*`

The `!` usually needs to be preceded by a backslash. For example:

```
dalias goto "stop at \!:1; cont; clear"
```

For more information, see the `csh(1)` man page.

**dbx Command**

The `dbx` command starts `dbx`. 
Native Mode Syntax

```
dbx options program_name                    Debug program_name.
dbx options program_name core              Debug program_name with corefile core.
dbx options program_name process_id       Debug program_name with process ID process_id.
dbx options - process_id                   Debug process ID process_id; dbx finds the program using /proc.
dbx options - core                         Debug using corefile core; see also “debug Command” on page 296.
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 295.

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.
```

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 295.

## Options

For both native mode and Java mode, options are the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-B</td>
<td>Suppress all messages; return with exit code of program being debugged.</td>
</tr>
<tr>
<td>-c commands</td>
<td>Execute commands before prompting for input.</td>
</tr>
<tr>
<td>-C</td>
<td>Preload the Runtime Checking library (see &quot;check Command&quot; on page 282).</td>
</tr>
<tr>
<td>-d</td>
<td>Used with -s, removes file after reading.</td>
</tr>
<tr>
<td>-e</td>
<td>Echo input commands.</td>
</tr>
<tr>
<td>-f</td>
<td>Force loading of core file, even if it doesn’t match.</td>
</tr>
<tr>
<td>-h</td>
<td>Print the usage help on dbx.</td>
</tr>
<tr>
<td>-I dir</td>
<td>Add dir to pathmap set (see &quot;pathmap Command&quot; on page 331).</td>
</tr>
<tr>
<td>-k</td>
<td>Save and restore keyboard translation state.</td>
</tr>
<tr>
<td>-q</td>
<td>Suppress messages about reading stabs.</td>
</tr>
<tr>
<td>-r</td>
<td>Run program; if program exits normally, exit.</td>
</tr>
<tr>
<td>-R</td>
<td>Print the readme file on dbx.</td>
</tr>
<tr>
<td>-s file</td>
<td>Use file instead of /current_directory/.dbxrc or $HOME/.dbxrc as the startup file</td>
</tr>
<tr>
<td>-S</td>
<td>Suppress reading of initialization file /installation_directory/lib/dbxrc.</td>
</tr>
<tr>
<td>-V</td>
<td>Print the version of dbx.</td>
</tr>
<tr>
<td>-w n</td>
<td>Skip n frames on where command.</td>
</tr>
<tr>
<td>-x exec32</td>
<td>Run the 32-bit dbx binary instead of the 64-bit dbx binary that runs by default on systems running a 64-bit OS.</td>
</tr>
<tr>
<td>--</td>
<td>Marks the end of the option list; use this if the program name starts with a dash.</td>
</tr>
</tbody>
</table>
**dbxenv Command**

The `dbxenv` command is used to list or set dbx environment variables. It has identical syntax and identical functionality in native mode and Java mode.

**Syntax**

```
dbxenv
```

Display the current settings of the dbx environment variables.

```
dbxenv environment_variable setting
```

Set `environment_variable` to `setting`.

where:

- `environment_variable` is a dbx environment variable.
- `setting` is a valid setting for that variable.

**debug Command**

The `debug` command lists or changes the program being debugged. In native mode, it loads the specified application and begins debugging the application. In Java mode, it loads the specified Java application, checks for the existence of the class file, and begins debugging the application.

**Native Mode Syntax**

```
debug
```

Print the name and arguments of the program being debugged.

```
debug program_name
```

Begin debugging `program_name` with no process or core.

```
debug -c core program_name
```

Begin debugging `program_name` with core file `core`.

```
debug -p process_id program_name
```

Begin debugging `program_name` with process ID `process_id`.

```
debug program_name core
```

Begin debugging `program_name` with core file `core`. `program_name` may be `-`, dbx will attempt to extract the name of the executable from the core file. For details, see "Debugging a Core File" on page 38.
debug Command

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug program_name process_id</td>
<td>Begin debugging program_name with process ID process_id. program_name may be -; dbx finds it using /proc.</td>
</tr>
<tr>
<td>debug -f ...</td>
<td>Force loading of a core file, even if it doesn’t match.</td>
</tr>
<tr>
<td>debug -r ...</td>
<td>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.</td>
</tr>
<tr>
<td>debug -clone ...</td>
<td>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 Solaris Studio IDE.</td>
</tr>
<tr>
<td>debug -clone</td>
<td>Starts another dbx process debugging nothing. Valid only if running in the Oracle Solaris Studio IDE.</td>
</tr>
<tr>
<td>debug [options] -- program_name</td>
<td>Start debugging program_name, even if program_name begins with a dash.</td>
</tr>
</tbody>
</table>

where:

- core is the name of a core file.
- options are the options listed in "Options" on page 298.
- pid is the process ID of a running process.
- program_name is the path name of the program.

Leaks checking and access checking are turned off when a program is loaded with the debug command. You can enable them with the check command (see "check Command" on page 282).

Java Mode Syntax

debugin
  Print the name and arguments of the program being debugged.

debug program_name [.class | .jar]
  Begin debugging program_name with no process.

debug -p process_id program_name [.class | .jar]
  Begin debugging program_name with process ID process_id.
debug program_name [.class | .jar] process_id
    Begin debugging program_name with process ID process_id. program_name may be -; dbx finds it using /proc

d 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.

d 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 Solaris Studio IDE.

d debug -clone
    Starts another dbx process debugging nothing. Valid only if running in the Oracle Solaris Studio IDE.

d debug [options] -- program_name { .class | .jar }
    Start debugging program_name, even if program_name begins with a dash.

where:

file_name is the name of a file.

options are the options listed in "Options" on page 298.

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 331.
- 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
     
     /installation_directory/lib/dbxrc.

-V  Print the version of dbx.

-w n  Skip n frames on where command.

--  Marks the end of the option list; use this if the program name starts with
     a dash.

delete Command

The delete command deletes breakpoints and other events. It has identical syntax and identical
functionality in native mode and Java mode.

Syntax

\texttt{delete [-h] \{handler_id\} 
\ldots} \quad \text{Remove trace commands, when commands, or stop
commands of given handler_ids. To remove hidden
handlers, you must give the -h option.}

\texttt{delete [-h] 0 | all | -all} \quad \text{Remove all trace commands, when commands, and stop
commands excluding permanent and hidden handlers.
Specifying -h removes hidden handlers as well.}

\texttt{delete -temp} \quad \text{Remove all temporary handlers.}

\texttt{delete $firedhandlers} \quad \text{Delete all the handlers that caused the latest stoppage.}

where:

\texttt{handler_id} is the identifier of a handler.

detach Command

The detach command releases the target process from dbx’s control.

Native Mode Syntax

\texttt{detach} \quad \text{Detach dbx from the target, and cancel any pending signals.}

\texttt{detach -sig \{signal\}} \quad \text{Detach while forwarding the given \textit{signal}.}
**detach** -stop

Detach dbx from the target and leave the process in a stopped state. This option allows temporary application of other /proc-based debugging tools that might be blocked due to exclusive access. For an example, see "Detaching dbx From a Process" on page 81.

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 + (see "examine Command" on page 304).

where:

*address* is the address at which to start disassembling. The default value of *address* is the address after the last address previously assembled. This value is shared by the examine command (see "examine Command" on page 304).

*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 reevaluates 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 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 the Oracle Solaris Studio 12.3 release, 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|+p] [-L] [-f format] [-F format] [-m|+m|--] expression, ...
```

See “`print Command` on page 333” for the meaning of these flags.

`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 333.”
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 | +f | -F | +F | -f | +f | -F | +F] expression | identifier, ...
  See “print Command” on page 333 for the meaning of these flags.

where:

class_name is the name of a Java class, using either the package path (using period (.) as a qualifier; for example, test1.extra.T1.Inner) or the full path name (preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers; for example, #test1/extra/T1$Inner). Enclose class_name in quotation marks if you use the $ qualifier.

eexpression is a valid Java expression.

field_name is the name of a field in the class.

format is the output format you want used to print the expression. For information on valid formats, see “print Command” on page 333.

identifier is a local variable or parameter, including this, the current class instance variable (object_name.field_name) or a class (static) variable (class_name.field_name).

object_name is the name of a Java object.

down Command

The down command moves down the call stack (away from main). It has identical syntax and identical functionality in native mode and Java mode.

Syntax

down Move down the call stack one level.
down number Move down the call stack number levels.
down -h [number] Move down the call stack, but don’t skip hidden frames.

where:

number is a number of call stack levels.
dump Command

The `dump` command prints all variables local to a procedure. It has identical syntax and identical functionality in native mode and Java mode.

Syntax

```
dump
```

Print all variables local to the current procedure.

```
dump procedure
```

Print all variables local to `procedure`.

where:

`procedure` is the name of a procedure.

edit Command

The `edit` command invokes `$EDITOR` on a source file. It is valid only in native mode.

The `edit` command uses `$EDITOR` if `dbx` is not running in the Oracle Solaris Studio IDE. Otherwise, it sends a message to the IDE to display the appropriate file.

Syntax

```
edit
```

Edit the current file.

```
edit file_name
```

Edit the specified file `file_name`.

```
edit procedure
```

Edit the file containing function or procedure `procedure`.

where:

`file_name` if the name of a file.

`procedure` is the name of a function or procedure.
examine Command

The `examine` command shows memory contents. It is valid only in native mode.

**Syntax**

```
examine [address] [ / [count] [format]]
```

Display the contents of memory starting at `address` for `count` items in format `format`.

```
examine address1, address2 [ / [format]]
```

Display the contents of memory from `address1` through `address2` inclusive, in format `format`.

```
examine address= [format]
```

Display the address (instead of the contents of the address) in the given format.

- The `address` may be +, which indicates the address just after the last one previously displayed (the same as omitting it).
- `x` is a predefined alias for `examine`.

where:

- `address` is the address at which to start displaying memory contents. The default value of `address` is the address after the address whose contents were last displayed. This value is shared by the `dis` command (see "dis Command" on page 300).

- `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 0
lx, lx  hexadecimal (4 bytes, same as X)
Ld, LD  decimal (8 bytes)
Lo, LO  octal (8 bytes)
Lx, LX  hexadecimal (8 bytes)

**exception Command**

The `exception` command prints the value of the current C++ exception. It is valid only in native mode.

**Syntax**

```
exception [-d | +d]  Prints the value of the current C++ exception, if any.
```

See the “print Command” on page 333 for the meaning of the `-d` flag.

**exists Command**

The `exists` command checks for the existence of a symbol name. It is valid only in native mode.

**Syntax**

```
exists name  Returns 0 if `name` is found in the current program, 1 if `name` is not found.
```
where:

name is the name of a symbol.

file Command

The file command lists or changes the current file. It has identical syntax and identical functionality in native mode and in Java mode.

Syntax

file
Print the name of the current file.

file file_name
Change the current file.

where:

file_name 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 (see “Specifying the Location of Your Java Source Files” on page 210).

Native Mode Syntax

files
List the names of all files that contributed debugging information to the current program (those that were compiled with -g).

files regular_expression
List the names of all files compiled with -g that match the given regular expression.

where:

regular_expression is a regular expression.

For example:
(dbx) files ^r
myprog:
  retregs.cc
  reg_sorts.cc
  reg_errmsgs.cc
  rhosts.cc

Java Mode Syntax

files    List the names of all of the Java source files known to dbx.

fix Command

The fix command recompiles modified source files and dynamically links the modified functions into the application. It is valid only in native mode. It is not valid on Linux platforms.

Syntax

fix            Fix the current file.
fix file_name file_name  Fix files in list.
fix -f         Force fixing the file, even if source hasn’t been modified.
fix -a         Fix all modified files.
fix -g         Strip -O flags and add -g flag.
fix -c         Print compilation line (may include some options added internally for use by dbx).
fix -n         Do not execute compile/link commands (use with -v).
fix -v         Verbos e mode (overrides dbx fix_verbose environment variable setting).
fix +v         Non-verbose mode (overrides dbx fix_verbose environment variable setting).
fixed Command

The fixed command lists the names of all fixed files. It is valid only in native mode.

Syntax

fixed

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

fortran_modules -f module_name

fortran_modules -v module_name

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

frame [-h] number

frame [-h] +[number]

frame [-h] -[number]

frame [-h] -h

where:

number is the number of a frame in the call stack.
**func Command**

In native mode, the *func* command lists or changes the current function. In Java mode, the *func* command lists or changes the current method.

**Native Mode Syntax**

```
func
```

Print the name of the current function.

```
func procedure
```

Change the current function to the function or procedure *procedure*.

where:

*procedure* is the name of a function or procedure.

**Java Mode Syntax**

```
func
```

Print the name of the current method.

```
func [class_name.]method_name [ (parameters) ]
```

Change the current function to the method *method_name*.

where:

*class_name* is the name of a Java class, using either the package path (using period (.) as a qualifier; for example, test1.extra.T1.Inner) or the full path name (preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers; for example, #test1/extra/T1$Inner). Enclose *class_name* in quotation marks if you use the $ qualifier.

*method_name* is the name of a Java method.

*parameters* are the method’s parameters.

**funcs Command**

The *funcs* command lists all function names that match a regular expression. It is valid only in native mode.

**Syntax**

```
funcs
```

List all functions in the current program.
funcs [-f file_name] [-g] [regular_expression]

If -f file_name is specified, list all functions in the file. If -g is specified, list all functions with debugging information. If file_name ends in .o, then all functions, including those created automatically by the compiler, are listed. Otherwise only functions appearing the source code are listed.

If regular_expression is specified, list all functions that match the regular expression.

where:

file_name 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"wprintf
"libc.so.1"printf
"libc.so.1"vprintf
"libc.so.1"vsprintf
```

gdb Command

The gdb command supports the gdb command set. It is valid only in native mode.

Syntax

gdb on | off

Use gdb on to enter the gdb command mode under which dbx understands and accepts gdb commands. To exit the gdb command mode and return to the dbx command mode, type gdb off. dbx commands are not accepted while in gdb command mode; 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
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 handler_id ...**
  - Enable given handlers, specify handler_id as all for all handlers.

- **handler -disable handler_id ...**
  - Disable given handlers, specify handler_id as all for all handlers. Use $fired_handlers instead of handler_id to disable the handlers that caused the most recent stoppage.

- **handler -count handler_id**
  - Print value of trip counter for given handler.

- **handler -count handler_id new_limit**
  - 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
hide regular_expression

List the stack frame filters currently in effect.
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
ignore number ...
ignore signal ...

Print a list of the ignored signals.
Ignore signal numbered number.
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 identical syntax and identical 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 doesn’t match the exception specification of the function it is thrown from is called an “unexpected” throw.

Unhandled and unexpected throws are intercepted by default.

**Syntax**

```
intercept -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 when you give the command. It is valid only in Java mode.

Classes in your program that have not yet been loaded are not printed.

Syntax

```
jclasses
jclasses -a
```

- `jclasses` Print the names of all Java classes known to dbx.
- `jclasses -a` Print system classes as well as other known Java classes.

joff Command

The joff command switches dbx from Java mode or JNI mode to native mode.

Syntax

```
joff
```
jon Command

The jon command switches dbx from native mode to Java mode.

Syntax
jon

jpkgs Command

The jpkgs command prints the names of all Java packages known to dbx when you give the command. It is valid only in Java mode.

Packages in your program that have not yet been loaded are not printed.

Syntax
jpkgs

kill Command

The kill command sends a signal to a process. It is valid only in native mode.

Syntax
kill -l List all known signal numbers, names, and descriptions.
killed Kill the controlled process.
killed job ... Send the SIGTERM signal to the listed jobs.
killed -signal job ... Send the given signal to the listed jobs.

where:

job may be a process ID or may be specified in any of the following ways:
%+ Kill the current job.
%-% Kill the previous job.
%number  Kill job number number.
%string  Kill the job that begins with string.
%?string  Kill the job that contains string.

signal is the name of a signal.

**Language Command**

The `language` command lists or changes the current source language. It is valid only in native mode.

**Syntax**

```
language  List or set the current source language. It is valid only in native mode.
```

where:

- `language` is `c`, `c++`, `fortran`, or `fortran90`.

**Note** – `c` is an alias for `ansic`.

**line Command**

The `line` command lists or change the current line number. It has identical syntax and identical functionality in native mode and in Java mode.

**Syntax**

```
line  Display the current line number.
line number  Set the current line number to number.
line "file_name"  Set current line number to line 1 in file_name.
line "file_name":number  Set current line number to line number in file_name.
```
where:

file_name is the name of the file in which to change the line number. The "" around the filename is optional.

number is the number of a line in the file.

Examples

line 100
line "/root/test/test.cc":100

list Command

The list command displays lines of a source file. It has identical syntax and identical functionality in native mode and in Java mode.

The default number of lines listed, N, is controlled by the dbx output_list_size environment variable.

Syntax

list         List N lines.
list number  List line number number.
list +        List next N lines.
list +n       List next n lines.
list -        List previous N lines.
list          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 64 for more information.
list file_name   List the start of the file file_name.
list file_name:n List file filename from line n.

where:

file_name 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

-1 or -instr      Intermix source lines and assembly code.
-w or -wn         List N (or n) lines (window) around line or function. This option is not allowed in combination with the "+" or "-" syntax or when two line numbers are specified.
-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 1000,5          // list from line 1000 to last line
list 2737 +24       // list line 2737 and next 24 lines
list 1000 -20       // list line 900 to 1000
list test.cc:33     // list source line 33 in file test.cc
list -w             // list N lines around current line
list -w8 "test.cc"func1 // list 8 lines around function func1
list -i 500 +10     // list source and assembly code for line 500 to line 510
**listi Command**

The listi command displays source and disassembled instructions. It is valid only in native mode.

See “list Command” on page 317 for details.

**loadobject Command**

The loadobject command lists and manages symbolic information from load objects. It is valid only in native mode.

**Syntax**

```
loadobject -list [regexp] [-a]  # Show currently loaded load objects (see “loadobject -list Command” on page 321)
loadobject -load loadobject     # Load symbols for specified load object (see “loadobject -load Command” on page 322).
loadobject -unload [regexp]     # Unload specified load objects (see “loadobject -unload Command” on page 322).
loadobject -hide [regexp]       # Remove load object from dbx's search algorithm (see “loadobject -hide Command” on page 321).
loadobject -use [regexp]        # Add load object to dbx's search algorithm (see “loadobject -use Command” on page 322).
loadobject -dumpelf [regexp]    # Show various ELF details of the load object (see “loadobject -dumpelf Command” on page 320).
loadobject -exclude ex_regexp   # Don't automatically load load objects matching ex_regexp (see “loadobject -exclude Command” on page 320).
loadobject exclude -clear       # Clear the exclude list of patterns (see “loadobject -exclude Command” on page 320).
```

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 Solaris OS commands `dump` or `elfdump`.

**loadobject -exclude Command**

The `loadobject -exclude` command tells `dbx` not to automatically load load objects 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.

**loadobject -list Command**

The `loadobject -list` command shows currently loaded load objects. 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 `what is` 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 322.

**loadobject -load Command**

The `loadobject -load` command loads symbols for specified load objects. It is valid only in native mode.

**Syntax**

```
loadobject -load loadobject
```

where:

`loadobject` can be a full path name or a library in `/usr/lib`, `/usr/lib/sparcv9`, or `/usr/lib/amd64`. If there is a program being debugged, then only the proper ABI library directory will be searched.

**loadobject -unload Command**

The `loadobject -unload` command unloads specified 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` may 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 Solaris platforms.

Syntax

lwp

lwp lwp_id

lwp -info

lwp \{lwp_id\} -setfp address_expression

lwp \{lwp_id\} -resetfp

where:

lwp_id is the identifier of a lightweight process.

If the command is used with both an lwp_id and an option, 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 Solaris platforms.

Syntax

lwps List all LWPs in the current process.

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.

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

```
mmapped_file mnnapped_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:

```c

caddr_t vaddr = NULL;
off_t offset = 0;
size_t = 10 * 1024;
int fd;
fd = open("../DATABASE", ...)
vaddr = mmap(vaddr, size, PROT_READ, MAP_SHARED, fd, offset);
index = (DBIndex *) vaddr;

To be able to access the database through the debugger as memory you would type:

```
mmappedfile ../DATABASE $[vaddr] $[offset] $[size]
```

Then you could look at your database contents in a structured way by typing:

```
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
Read in debugging information for the module called name.
module [-f] [-v] [-q] -a
Read in debugging information for all modules.

where:

name is the name of a module for which to read debugging information.

- a specifies all modules.
- f forces reading of debugging information, even if the file is newer than the executable (use with caution!).
- v specifies verbose mode, which prints language, file names, and such.
- q specifies quiet mode.

modules Command

The modules command lists module names. It is valid only in native mode.

Syntax

modules [-v] List all modules.
modules [-v] -debug List all modules containing debugging information.
modules [-v] -read List names of modules containing debugging information that have been read in already.

where:

- v specifies verbose mode, which prints language, file names, 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 dbx Environment Variables” on page 54) controls whether breakpoints are enabled during a step.

**Native Mode Syntax**

next

Step one line (step over calls). With multithreaded programs when a function call is stepped over, all LWPs (lightweight processes) are implicitly resumed for the duration of that function call in order to avoid deadlock. Non-active threads cannot be stepped.

next n

Step *n* lines (step over calls).

next ... -sig signal

Deliver the given signal while stepping.

next ... thread_id

Step the given thread.

next ... lwp_id

Step the given LWP. Will not implicitly resume all LWPs when stepping over a function.

where:

*n* is the number of lines to step.

*signal* is the name of a signal.

*thread_id* is a thread ID.

*lwp_id* is an LWP ID.
When an explicit `thread_id` or `lwp_id` is given, the deadlock avoidance measure of the generic `next` command is defeated.

See also "nexti Command" on page 328 for machine-level stepping over calls.

---

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

---

### Java Mode Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>next n</code></td>
<td>Step <code>n</code> lines (step over calls).</td>
</tr>
<tr>
<td><code>next ... thread_id</code></td>
<td>Step the given thread.</td>
</tr>
<tr>
<td><code>next ... lwp_id</code></td>
<td>Step the given LWP. Will not implicitly resume all LWPs when stepping over a function.</td>
</tr>
</tbody>
</table>

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 given, the deadlock avoidance measure of the generic `next` command is defeated.

---

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

---

### 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.

**Syntax**

```
omp_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
```
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`  
  Serialize the execution of the next encountered parallel region for the current thread.

- `omp_serialize -team`  
  Serialize the execution of the next encountered parallel region 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`  
  Print all the threads in the current team.
omp_team parallel_region_id

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  
omp_tr task_region_id  
omp_tr -ancestors  
omp_tr task_region_id -ancestors  
omp_tr -tree  
```

- `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 (see “Using the pathmap Command To Improve Skimming” on page 275). The `pathmap` command has identical syntax and identical functionality in native mode and in Java mode.

The `pathmap` command is useful for dealing with automounted and explicit NFS mounted filesystems with different paths on differing hosts. Specify `-c` when you are trying to correct problems arising due to the automounter since CWDs are inaccurate on automounted filesystems as well. The `pathmap` command is also useful if source or build trees are moved.

```
pathmap /tmp_mnt / exists by default.
```
The `pathmap` command is used to find load objects for core files when the `dbx` environment variable `core_to_pathmap` is set to on. Other than this case, the `pathmap` command has no effect on finding load objects (shared libraries). See “Debugging a Mismatched Core File” on page 40.

**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 filepath prefixes. `from` refers to the filepath compiled into the executable or object file and `to` refers to the filepath at debug time.

- `from1` is filepath of the first mapping to be deleted.

- `from2` is filepath of the last mapping to be deleted.

- `index` specifies the index with which the mapping is to be inserted in the list. If you do not specify an index, the mapping is added to the end of the list.

- `index1` is the index of the first mapping to be deleted.

- `index2` is the index of the last mapping to be deleted.

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

If you specify `-s`, the existing mappings are listed in an output format that `dbx` can read.

If you specify `-d`, the specified mappings are deleted.

**Examples**

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2
  # maps /export/home/work1/abc/test.c to /net/mmm/export/home/work2/abc/test.c
(dbx) pathmap /export/home/newproject/abc/test.c
  # maps /export/home/work1/abc/test.c to /export/home/newproject/abc/test.c
(dbx) pathmap
(1) -c /tmp_mnt /
```
**pop Command**

The `pop` command removes one or more frames from the call stack. It is valid only in native mode.

You can pop only to a frame for a function that was compiled with `-g`. The program counter is reset to the beginning of the source line at the callsite. You cannot pop past a function call made by the debugger; use `pop -c`.

Normally a `pop` command calls all the C++ destructors associated with the popped frames; you can override this behavior by setting the `dbx pop_auto_destruct` environment variable to off (see "Setting dbx Environment Variables" on page 54).

**Syntax**

```
pop
pop number
pop -f number
pop -c
```

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 -r expression
```

Print the value of the expression `expression, ...`.

Print the value of the expression `expression` including its inherited members.
print +r expression
Don’t print inherited members when the dbx output_inherited_members environment variable is 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 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 dbx environment 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 -t format expression
Use format as the format for integers, strings, or floating-point expressions.

print -F format expression
Use the given format but do not print the left hand side (the variable name or expression).

print -o expression
Print the value of expression, which must be an enumeration as an ordinal value. You may also use a format string here (-tformat). This option is ignored for non-serialization expressions.

print -m expression
Apply macro expansion to expression when the dbx environment variable macro_expand is set to off.

print +m expression
Skip macro expansion of expression when the dbx environment variable macro_expand is set to on.
print Command

print expression  "--" signals the end of flag arguments. This is useful if for the expression may start with a plus or minus (see "Program Scope" on page 64 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, ...
### proc Command

The `proc` command displays the status of the current process. It has identical syntax and identical functionality in native mode and in Java mode.

#### Syntax

- `proc -map` Show the list of loadobjects with addresses.
- `proc -pid` Show current process ID (pid).

---

**print**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`print -r expression</td>
<td>identifier`</td>
</tr>
<tr>
<td>`print +r expression</td>
<td>identifier`</td>
</tr>
<tr>
<td>`print -d [-r] expression</td>
<td>identifier`</td>
</tr>
<tr>
<td>`print +d [-r] expression</td>
<td>identifier`</td>
</tr>
<tr>
<td>`print -- expression</td>
<td>identifier`</td>
</tr>
</tbody>
</table>

**where:**

- `class_name` is the name of a Java class, using either the package path (using period (.) as a qualifier; for example, `test1.extra.T1.Inner`) or the full pathname (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.
**prog Command**

The `prog` command manages programs being debugged and their attributes. It has identical syntax and identical functionality in native mode and Java mode.

**Syntax**

- `prog -readsyms` Read symbolic information which was postponed by having set the `dbx run_quick` environment variable to on.
- `prog -executable` Prints the full path of the executable, - if the program was attached to using `.`.
- `prog -argv` Prints the whole `argv`, including `argv[0]`.
- `prog -args` Prints the `argv`, excluding `argv[0]`.
- `prog -stdin` Prints `<filename>` or empty if stdin is used.
- `prog -stdout` Prints `>` `filename` or `>> filename` or empty if stdout is used. The outputs of `-args`, `-stdin`, `-stdout` are designed so that the strings can be combined and reused with the `run` command (see "run Command" on page 341).

**quit Command**

The `quit` command exits `dbx`. It has identical syntax and identical functionality in native mode and Java mode.

If `dbx` is attached to a process, the process is detached from before exiting. If there are pending signals, they are cancelled. Use the `detach` command (see "detach Command" on page 299) 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 0x00000000 0xe000109b8
  l0-l3  0x00000014 0x0000000a 0x0000000a 0x00010a88
  l4-l7  0xe0000438 0x00000001 0x00000007 0xe0ff4d54
  i0-i3  0x00000001 0xe0ff44a4 0xe0ff4ac 0x00020c00
  i4-i7  0x00000001 0x00000000 0xe0ff4440 0x000108c4
  y     0x00000000
  psr   0x40400086
  pc    0x000109c0:main+0x4
  mov   0x5, %l0
  npc   0x0000109c4:main+0x8
  st    %l0, [%fp - 0x8]
  f0f1  +0.000000000000000e+00
  f2f3  +0.000000000000000e+00
  f4f5  +0.000000000000000e+00
  f6f7  +0.000000000000000e+00
```

replay Command

The `replay` command replays debugging commands since the last run, rerun, or debug command. It is valid only in native mode.
Syntax

**replay [-number]**  
Replay all or all minus *number* commands since last run command,  
**rerun** command, or **debug** command.

where:

*number* is the number of commands not to replay.

**rerun Command**

The **rerun** command runs the program with no arguments. It has identical syntax and identical functionality in native mode and Java mode.

**Syntax**

**rerun**  
Begin executing the program with no arguments.

**rerun arguments**  
Begin executing the program with new arguments by the **save** command  
(see "**save Command**" on page 343).

**restore Command**

The **restore** command restores **dbx** to a previously saved state. It is valid only in native mode.

**Syntax**

**restore [file_name ]**

where:

*file_name* 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 333 for the meanings of the flags).

where:

- `expression` is the expression whose value you want to print.
- `format` is the output format you want used to print the expression. For information on valid formats, see “print Command” on page 333.

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.

Syntax

```
rtc showmap
```

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 143 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.

Since 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 such an error was not suppressed automatically.

Syntax

rtc skippatch load_object ...
   Exclude the specified load objects from being instrumented.

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 (see “debug Command” on page 296) 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 341).
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 ] [ file_name ]

Save all or all minus number commands since last run command, rerun command, or debug command to the default file or file_name.

where:

number is the number of commands not to save.

filename is the name of the file to save the dbx commands executed since the last run, rerun, or debug command.

scopes Command

The scopes command prints a list of active scopes. It is valid only in native mode.

Syntax

scopes

search Command

The search command searches forward in the current source file. It is valid only in native mode.

Syntax

search string

Search forward for string in the current file.
search

Repeat search, using last search string.

where:

string is the character string for which you wish to search.

**showblock Command**

The showblock command shows where the particular heap block was allocated from runtime checking. It is valid only in native mode.

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. See “check Command” on page 282.

**Syntax**

showblock -a address

where:

address is the address of a heap block.

**showleaks Command**

**Note** – The showleaks command is available only on Solaris platforms.

In the default non-verbose case, a one line report per leak record is printed. Actual leaks are reported followed by the possible leaks. Reports are sorted according to the combined size of the leaks.

**Syntax**

showleaks [-a] [-m m] [-n number] [-v]

where:

-a shows all the leaks generated so far (not just the leaks since the last showleaks command).
-m m combines leaks; if the call stack at the time of allocation for two or more leaks matches m frames, then these leaks are reported in a single combined leak report. If the -m option is given, it overrides the global value of m specified with the check command (see "check Command" on page 282).

-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 (see “showleaks Command” on page 344) are also included in the report.

**Syntax**

showmemuse [-a] [-m m] [-n number] [-v]

where:

- a shows all the blocks in use (not just the blocks since the last showmemuse command).

- m m combines the blocks in use reports. The default value of m is 2 or the global value last given with the check command (see "check Command" on page 282). 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 file_name    Execute commands from file file_name. $PATH is not searched.
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
```

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. Such 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 given 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 given thread. Does not apply to `step up`.

`step ...lwp_id` Step the given LWP. Does not implicitly resume all LWPs when stepping over a function.

`step to [ function ]` Attempts to step into `function` called from the current source code line. If `function` is not given, 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.

`lwp_id` is an LWP ID.

`function` is a function name.

Only when an explicit `lwp_id` is given, the deadlock avoidance measure of the generic `step` command is defeated.

When executing the `step to` command, while an attempt is made to step into the last assembly call instruction or step into a function (if specified) in the current source code line, the call may not be taken due to a conditional branch. In a case where the call is not taken or there is no
function call in the current source code line, the step to command steps over the current source code line. Take special consideration on user-defined operators when using the step to command.

See also “stept Command” on page 348 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 given thread. Does not apply to step up.
step ...lwp_id Step the given LWP. Does not implicitly resume all LWPs when stepping over a method.

stept Command

The stept command steps one machine instruction (stepping into calls). It is valid only in native mode.

Syntax

stept Single step one machine instruction (step into calls).
stept n Single step n machine instructions (step into calls).
stept -sig signal Step and deliver the given signal.
stept ...lwp_id Step the given LWP.
stept ...thread_id Step the LWP on which the given thread is active.

where:

n is the number of instructions to step.
signal is the name of a signal.
stop Command

The stop command sets a source-level breakpoint.

Syntax

The stop command has the following general syntax:

```
stop event_specification [ modifier]
```

When the specified event occurs, the process is stopped.

Native Mode Syntax

The following specific syntaxes are some of the more important syntaxes that are valid in native mode. For additional events, see “Setting Event Specifications” on page 252.

```
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 Solaris 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 92.

stop at line-number
   Stop execution at line-number. See also “Setting a stop Breakpoint at a Line of Source Code” on page 88.

stop change variable
   Stop execution when the value of variable has changed.

stop cond condition_expression
   Stop execution when the condition denoted by condition_expression evaluates to true.

stop in function
   Stop execution when function is called. See also “Setting a stop Breakpoint in a Function” on page 89.
```
stop Command

```
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 90.

stop infile file_name
Stop execution when any function in file_name 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 90.

stop inobject object_expression [ -recurse | -norecurse ]
C++ only: set breakpoint on entry into any non-static method of the class and all its base
classes when called from the object object_expression. -recurse is the default. If -norecurse
is specified, the base classes are not included. See also "Setting Breakpoints in Objects" on
page 91.

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.
```
stop Command

- **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, use delete hid.

- **hidden** Hide the event from the status command. Some import modules may choose to use this. Use status -h to see them.

- **lwp 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.

**stop access mode class_name.field_name**

Stop execution when the memory specified by class_name.field_name has been accessed.

**stop at line_number**

Stop execution at line_number.

**stop at file_name:line_number**

Stop execution at line_number in file_name.

**stop change class_name.field_name**

Stop execution when the value of field_name in class_name has changed.

**stop classload**

Stop execution when any class is loaded.
stop classload class_name
Stop execution when class_name is loaded.

stop classunload
Stop execution when any class is unloaded.

stop classunload class_name
Stop execution when class_name is unloaded.

stop cond condition_expression
Stop execution when the condition denoted by condition_expression evaluates to true.

stop in class_name.method_name
Stop execution when class_name.method_name has been entered, and the first line is about
to be executed. If no parameters are specified and the method is overloaded, a list of methods
is displayed.

stop in class_name.method_name([parameters])
Stop execution when class_name.method_name has been entered, and the first line is about
to be executed.

stop inmethod class_name.method_name
Set breakpoints on all non-member methods class_name.method_name.

stop inmethod class_name.method_name ([parameters])
Set breakpoints on all non-member methods class_name.method_name.

stop throw
Stop execution when a Java exception has been thrown.

stop throw type
Stop execution when a Java exception of type has been thrown.

where:

class_name is the name of a Java class, using either the package path (using period (.) as a
qualifier; for example, test1.extra.T1.Inner) or the full path name (preceded by a pound
sign (#) and using slash (/) and dollar sign ($) as qualifiers; for example,
#test1/extra/T1$Inner). Enclose class_name in quotation marks if you use the $ qualifier.

condition_expression can be any expression, but it must evaluate to an integral type.

field_name is the name of a field in the class.

file_name is the name of a file.

line_number is the number of a source code line.

method_name is the name of a Java method.

mode specifies how the memory was accessed. It can be composed of one or all of the letters:
The memory at the specified address has been read.
The memory has been written to.

*mode* can also contain the following:

- **b** Stops the process before the access.

The program counter will point at the offending instruction.

*parameters* are the method’s parameters.

*type* is a type of Java exception. *-unhandled* or *-unexpected* can be used for type.

The following modifiers are valid in Java mode:

- **-if** *condition_expression* The specified event occurs only when *condition_expression* evaluates to true.

- **-count** *number* Starting at 0, each time the event occurs, the counter is incremented. When *number* is reached, execution stops and the counter is reset to 0.

- **-count infinity** Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.

- **-temp** Create a temporary breakpoint that is deleted when the event occurs.

- **-disable** Create the breakpoint in a disabled state.

See also "*stopi Command*" on page 353 for setting a machine-level breakpoint.

For a list and the syntax of all events see "Setting Event Specifications" on page 252.

### 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:

- `stop i at address`  Stop execution at location `address`.
- `stop i in function`  Stop execution when `function` is called.

where:

- `address` is any expression resulting in or usable as an address.
- `function` is the name of a function.

For a list and the syntax of all events see "Setting Event Specifications" on page 252.

**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 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).
supress -r id...
   Remove the unsuppress events as given by the ids, which can be obtained with unsuppress command (see "unsuppress Command" on page 367).

supress -r 0 | all | -all
   Remove all the unsuppress events as given by the unsuppress command (see "unsuppress Command" on page 367).

supress errors
   Suppress errors everywhere.

supress errors in [ functions ] [ files ] [ load_objects ]
   Suppress errors in list of functions, list of files, and list of load_objects.

supress errors at line
   Suppress errors at line.

supress errors at "file":line
   Suppress errors at line in file.

supress errors addr address
   Suppress errors at location address.

where:

address is a memory address.

tools are blank separated and can be any combination of:

all    All errors
aib    Possible memory leak - address in block
air    Possible memory leak - address in register
baf    Bad free
duf    Duplicate free
mel    Memory leak
maf    Misaligned free
mar    Misaligned read
maw    Misaligned write
oom    Out of memory
rob    Read from array out-of-bounds memory
rue    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 132 for more information on suppressing errors.
See the “unsuppress Command” on page 367 for information on unsuppressing errors.

**sync Command**

The sync command shows information about a given synchronization object. It is valid only in native mode.

---

**Note** – The sync command is available only on Solaris platforms.

**Syntax**

`sync -info address` Show information about the synchronization object at address.

where:

`address` is the address of the synchronization object.
**syncs Command**

The **syncs** command lists all synchronization objects (locks). It is valid only in native mode.

**Note** – The **syncs** command is available only on Solaris platforms.

**Syntax**

```
syncs
```

**thread Command**

The **thread** command lists or changes the current thread.

**Native Mode Syntax**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>Display current thread.</td>
</tr>
<tr>
<td>thread thread_id</td>
<td>Switch to thread <strong>thread_id</strong>.</td>
</tr>
</tbody>
</table>

In the following variations, a missing **thread_id** implies the current thread.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread -info [thread_id]</td>
<td>Print everything known about the given thread. For OpenMP threads, the information includes the OpenMP thread id, parallel region id, task region id, and thread state.</td>
</tr>
<tr>
<td>thread -hide [thread_id]</td>
<td>Hide the given (or current) thread. It will not show up in the generic threads listing.</td>
</tr>
<tr>
<td>thread -unhide [thread_id]</td>
<td>Unhide the given (or current) thread.</td>
</tr>
<tr>
<td>thread -unhide all</td>
<td>Unhide all threads.</td>
</tr>
<tr>
<td>thread -suspend thread_id</td>
<td>Keep the given thread from ever running. A suspended thread shows up with an “S” in the threads list.</td>
</tr>
<tr>
<td>thread -resume thread_id</td>
<td>Undo the effect of -suspend.</td>
</tr>
<tr>
<td>thread -blocks [thread_id]</td>
<td>List all locks held by the given thread blocking other threads.</td>
</tr>
<tr>
<td>thread -blockedby [thread_id]</td>
<td>Show which synchronization object the given thread is blocked by, if any.</td>
</tr>
</tbody>
</table>

Appendix D • Command Reference 357
where:

thread_id is a thread ID.

**Java Mode Syntax**

```plaintext
thread            Display current thread.
thread thread_id  Switch to thread thread_id.
```

In the following variations, a missing thread_id implies the current thread.

```plaintext
thread -info [thread_id]          Print everything known about the given thread.
thread -hide [thread_id]          Hide the given (or current) thread. It will not show up in
                                    the generic threads listing.
thread -unhide [thread_id]        Unhide the given (or current) thread.
thread -unhide all                Unhide all threads.
thread -suspend thread_id         Keep the given thread from ever running. A suspended
                                   thread shows up with an “S” in the threads list.
thread -resume thread_id          Undo the effect of -suspend.
thread -blocks [thread_id]        Lists the Java monitor owned by thread_id.
thread -blockedby [thread_id]     Lists the Java monitor on which thread_id is blocked.
```

where:

thread_id is a dbx-style thread ID of the form `t@number` or the Java thread name specified for
the thread.

**threads Command**

The threads command lists all threads.

**Native Mode Syntax**

```plaintext
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 \@num meaning the thread is bound (currently assigned to the designated LWP), or a l@num meaning the thread is active (currently scheduled to run).
- The “Start function” of the thread as passed to thr_create. A ?() means that the start function is not known.
- The thread state, which is one of the following:
  - monitor
  - running
  - sleeping
  - wait
  - unknown
  - zombie

  The function that the thread is currently executing.

Java Mode Syntax

threads  Print the list of all known threads.
threads -all  Print threads normally not printed (zombies).
threads -mode all|filter  Controls whether all threads are printed or threads are filtered. The default is to filter threads.
threads -mode auto|manual  Under the 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
  - wait
  - unknown
  - zombie
- The thread name in single quotation marks
- A number indicating the thread priority

trace Command

The `trace` command shows executed source lines, function calls, or variable changes.

The speed of a trace is set using the `dbx trace_speed` environment variable.

If `dbx` is in Java mode and you want to set a trace breakpoint in native code, switch to Native mode using the `joff` command (see “joff Command” on page 314) or prefix the `trace` command with `native` (see “native Command” on page 326).

If `dbx` is in JNI mode and you want to set a trace breakpoint in Java code, prefix the `trace` command with `java` (see “java Command” on page 314).

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 Command**

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

**trace step**  
Trace each source line, function call, and return.

**trace next -in function**  
Trace each source line while in the given `function`.

**trace at line_number**  
Trace given source line.

**trace in function**  
Trace calls to and returns from the given `function`.

**trace infile file_name**  
Trace calls to and returns from any function in `file_name`.

**trace inmember function**  
Trace calls to any member function named `function`.

**trace infunction function**  
Trace when any function named `function` is called.

**trace inclass class**  
Trace calls to any member function of `class`.

**trace change variable**  
Trace changes to the `variable`.

where:

- `file_name` is the name of the file to which you want trace output sent.
- `function` is the name of a function.
- `line_number` is the number of a source code line.
- `class` is the name of a class.
- `variable` is the name of a variable.

The following modifiers are valid in native mode.

- **-if condition_expression**  
The specified event occurs only when `condition_expression` evaluates to true.

- **-in function**  
Execution stops only if the specified event occurs in `function`.

- **-count number**  
Starting at 0, each time the event occurs, the counter is incremented. When `number` is reached, execution stops and the counter is reset to 0.

- **-count infinity**  
Starting at 0, each time the event occurs, the counter is incremented. Execution is not stopped.

- **-temp**  
Create a temporary breakpoint that is deleted when the event occurs.
-disable  Create the breakpoint in a disabled state.
-instr  Do instruction level variation. For example, step becomes instruction level stepping, and at takes a text address for an argument instead of a line number.
-perm  Make this event permanent across debug. Certain events (like breakpoints) are not appropriate to be made permanent. delete all will not delete permanent handlers, use delete hid.
-hidden  Hide the event from the status command. Some import modules may choose to use this. Use status -h to see them.
-lwp lwpid  Execution stops only if the specified event occurs in the given LWP.
-thread thread_id  Execution stops only if the specified event occurs in the given thread.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

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

trace at line_number
   Trace line_number.

trace at file_name.line_number
   Trace give source file_name.line_number.

trace in class_name.method_name
   Trace calls to and returns from class_name.method_name.

trace in class_name.method_name([parameters]).
   Trace calls to and returns from class_name.method_name([parameters]).

trace inmethod class_name.method_name
   Trace 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, using either the package path (using period (.) as a qualifier, for example, test1.extra.T1.Inner) or the full path name (preceded by a pound
sign (#) and using slash (/) and dollar sign ($) as qualifiers, for example, 
#test1/extra/T1$Inner). Enclose class_name in quotation marks if you use the $ qualifier.

file_name is the name of a file.

line_number is the number of a source code line.

method_name is the name of a Java method.

parameters are the method’s parameters.

The following modifiers are valid in Java mode.

- **-if condition_expression**  The specified event occurs and the trace is printed only when condition_expression evaluates to true.

- **-count number**  Starting at 0, each time the event occurs, the counter is incremented. When number is reached, the trace is printed and the counter is reset to 0.

- **-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 252.

**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**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tracei step</td>
<td>Trace each machine instruction.</td>
</tr>
<tr>
<td>tracei next -in function</td>
<td>Trace each instruction while in the given function.</td>
</tr>
<tr>
<td>tracei at address_expression</td>
<td>Trace the instruction at address.</td>
</tr>
</tbody>
</table>
**tracei in function**
Trace calls to and returns from the given function.

**tracei in member function**
Trace calls to any member function named function.

**tracei in function function**
Trace when any function named function is called.

**tracei in class class**
Trace calls to any member function of class.

**tracei change variable**
Trace changes to the variable.

where:

*filename* is the name of the file to which you want trace output sent.

*function* is the name of a function.

*line* is the number of a source code line.

*class* is the name of a class.

*variable* is the name of a variable.

See “trace Command” on page 360 for more information.

**uncheck Command**

The uncheck command disables checking of memory access, leaks, or usage. It is valid only in native mode.

**Syntax**

```plaintext
uncheck
uncheck -access
uncheck -leaks
uncheck -memuse
uncheck -all
uncheck [functions] [files] [loadobjects]
```

Print current status of checking.

Turn off access checking.

Turn off leak checking.

Turn off memory use checking (leak checking is turned off as well).

Equivalent to uncheck -access; uncheck -memuse.

Equivalent to suppress all in functions files loadobjects.

where:
functions is one or more function names.
files is one or more file names.
loadobjects is one or more loadobject names
See “check Command” on page 282 for information to turn on checking.
See “suppress Command” on page 354 for information on suppressing of errors.
See “Capabilities of Runtime Checking” on page 119 for an introduction to runtime checking.

undisplay Command

The undisplay command undoes display commands.

Native Mode Syntax

undisplay expression, ... Undo a display expression command.
undisplay n, ... Undo all the display commands numbered n, ...
undisplay 0 Undo all display commands.

where:

expression is a valid expression.

Java Mode Syntax

undisplay expression, ... | identifier, ...
   Undo a display expression, ... or display identifier, ... command.
undisplay n, ... Undo the display commands numbered n, ...
undisplay 0 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 0
Delete all stack frame filters.

unhide regular_expression
Delete stack frame filter regular_expression.

unhide number
Delete stack frame filter number number.

where:

regular_expression is a regular expression.

number is the number of a stack frame filter.

The hide command (see "Catch Floating Point Exceptions" on page 270) 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 loadobject. Any other errors can be suppressed only by using the suppress command
   (see "suppress Command" on page 354) with the -d option.

unsuppress -d errors
   Modify the default suppressions for all loadobjects by further unsuppressing errors.

unsuppress -d errors in loadobjects
   Modify the default suppressions in the loadobjects by further unsuppressing errors.

unsuppress -last
   At error location unsuppress present error.

unsuppress -reset
   Set the default suppression mask to the original value (startup time).

unsuppress errors
   Unsuppress errors everywhere.

unsuppress errors in [functions] [files] [loadobjects]
   Suppress errors in list of functions, list of files, and list of loadobjects.

unsuppress errors at line
   Unsuppress errors at line.

unsuppress errors at "file"line
   Unsuppress errors at line in file.
unwatch Command

The `unwatch` command undoes a `watch` command. It is valid only in native mode.

**Syntax**

- `unwatch expression` Undo a `watch expression` command.
- `unwatch n` Undo the watch commands numbered `n`.
- `unwatch 0` Undo all watch commands.

where:

- `expression` is a valid expression.

up Command

The `up` command moves up the call stack (toward main). It has identical syntax and identical functionality in native mode and in Java mode.

**Syntax**

- `up` Move up the call stack one level.
- `up number` Move up the call stack `number` levels.
- `up -h [number]` Move up the call stack, but don’t skip hidden frames.

where:

- `number` is a number of call stack levels.
use Command

The use command lists or changes the directory search path. It is valid only in native mode.

This command is an anachronism and usage of this command is mapped to the following pathmap commands:

use is equivalent to pathmap -s

use directory is equivalent to pathmap directory.

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 expression
    Watch the value of expression expression at every stopping point.

watch [-r|+r|-d|+d|-S|+S|-p|+p|-L| -fformat] [-fformat] [-m|+m| - ] expression
    See "print Command" on page 333 for the meaning of these flags.

where:

expression is a valid expression.

format is the output format you want used to print the expression. For information on valid formats, see "print Command" on page 333.

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 [-r] type`  
Print the declaration of the type type.

`whatis -e [-r] [-d] expression`  
Print the type of the expression expression.

where:

`name` is the name of a non-type or macro.

`type` is the name of a type.

`expression` is a valid expression.

`macro` is the name of a macro.

- `-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.
- `-m` forces macro expansion even if the `dbx` environment 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 (undefined member functions are not listed), the static data members, the class friends, and the data members that are defined explicitly within that class.

Specifying the `-r` (recursive) option adds information from the inherited classes.

The `-d` flag, when used with the `-e` flag, uses the dynamic type of the expression.

For C++, template-related identifiers are displayed as follows:

- All template definitions are listed with `whatis -t`.
- Function template instantiations are listed with `whatis`.
- Class template instantiations are listed with `whatis -t`.
Java Mode Syntax

`what is` `identifier`       Print the declaration of `identifier`.

where:

`identifier` is a class, a method in the current class, a local variable in the current frame, or a field in the current class.

when Command

The `when` command executes commands when a specified event occurs.

If dbx is in Java mode and you want to set a `when` breakpoint in native code, switch to Native mode using the `joff` command (see “`joff` Command” on page 314) or prefix the `when` command with `native` (see “`native` Command” on page 326).

If dbx is in JNI mode and you want to set a `when` breakpoint in Java code, prefix the `when` command with `java` (see “`java` Command” on page 314).

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 \textit{line\_number} \{ \textit{command}; \} \quad \text{Execute } \textit{command} \text{ when } \textit{line\_number} \text{ is reached.}

when in \textit{procedure} \{ \textit{command}; \} \quad \text{Execute } \textit{command} \text{ when } \textit{procedure} \text{ is called.}

where:

\textit{line\_number} is the number of a source code line.

\textit{command} is the name of a command.

\textit{procedure} is the name of a procedure.

Java Mode Syntax

The following specific syntaxes are valid in Java mode.

when at \textit{line\_number} \quad \text{Execute command when source } \textit{line\_number} \text{ is reached.}

when at \textit{file\_name.line\_number} \quad \text{Execute command when } \textit{file\_name.line\_number} \text{ is reached.}

when in \textit{class\_name.method\_name} \quad \text{Execute command when } \textit{class\_name.method\_name} \text{ is called.}

when in \textit{class\_name.method\_name}([parameters]) \quad \text{Execute command when } \textit{class\_name.method\_name}([parameters]) \text{ is called.}

\textit{class\_name} is the name of a Java class, using either the package path (using period (.) as a qualifier; for example, \texttt{test1\_extra\_T1\_Inner}) or the full path name (preceded by a pound sign (#) and using slash (/) and dollar sign ($) as qualifiers; for example, \texttt{#test1/extra/T1$Inner}). Enclose \textit{class\_name} in quotation marks if you use the $ qualifier.

\textit{file\_name} is the name of a file.

\textit{line\_number} is the number of a source code line.

\textit{method\_name} is the name of a Java method.

\textit{parameters} are the method’s parameters.

For a list and the syntax of all events, see “Setting Event Specifications” on page 252.

See “when Command” on page 373 for executing commands on given low-level event.
**wheni Command**

The `wheni` command executes commands when a given low-level event occurs. It is valid only in native mode.

The `wheni` command has the following general syntax:

**Syntax**

```
wheni event_specification [modifier] {command . . . ; }
```

When the specified event occurs, the commands are executed.

The following specific syntax is valid:

```
wheni at address_expression { command; }
```

Execute `command` when `address_expression` 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 252.

**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
where number
where -f number
where -fp address_expression
where -h
where -l
```

- `Print a procedure traceback.`
- `Print the number top frames in the traceback.`
- `Start traceback from frame number.`
- `Print traceback as if fp register had address_expression value.`
- `Include hidden frames.`
- `Include library name with function name.`
where Command

where -q Quick traceback (only function names).
where -v Verbosetraceback (include function args and line info).

where:

number is a number of call stack frames.

Any of the above forms may 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 the correct value has been identified, you can set it with an assign command or \wp command.

Java Mode Syntax

where [thread_id] Print a method traceback.
where [thread_id] number Print the number top frames in the traceback.
where -f [thread_id] number Start traceback from frame number.
where -q [thread_id] Quick trace back (only method names).
where -v [thread_id] Verbosetraceback (include method arguments and line information).

where:

number is a number of call stack frames.

thread_id is a dbx-style thread ID or the Java thread name specified for the thread.

whereami Command

The whereami command displays the current source line. It is valid only in native mode.

Syntax

whereami Display the source line corresponding to the current location (top of the stack), and the source line corresponding to the current frame, if different.
whereami -instr  Same as above, except that the current disassembled instruction is printed instead of the source line.

whereis Command

The whereis command prints all uses of a given name, or symbolic name of an address. It is valid only in native mode.

Syntax

whereis name  Print all declarations of name.
whereis -a address_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 given 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 something that is in scope; for example, a variable, function, class template, or function template.

type is the name of a type.

- n displays the full qualification of a non-type. It is not necessary to specify - n; this is the default if you type the which command with no options.

- t displays the full qualification of a type.
whocatches Command

- m forces macro lookup even if the dbx environment 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.
Numbers and Symbols
:: (double-colon) C++ operator, 66–67

A
access checking, 124
access event, 254–255
address
  current, 63
  display format, 223
  examining contents at, 221–225
adjusting default dbx settings, 53
alias command, 44
AMD64 registers, 233
array_bounds_check environment variable, 55
arrays
  bounds, exceeding, 195
  evaluating, 111–115
Fortran, 197
Fortran 95 allocatable, 198
slicing, 111, 114
  syntax for C and C++, 111–112
  syntax for Fortran, 112–113
striding, 111, 114
  syntax for slicing, striding, 111–113
assembly language debugging, 221
assign command
  syntax, 277–278
  using to assign a value to a variable, 111, 246
  using to reassign correct values to global variables, 155
assign command (Continued)
  using to restore a global variable, 155
  assigning a value to a variable, 111, 246
  at event, 253
attach command, 65, 80, 278–279
attach event, 259
attached process, using runtime checking on, 137–139
attaching
  dbx to a new process while debugging an existing process, 81
  dbx to a running child process, 163
  dbx to a running process, 42, 80–81
  when dbx is not already running, 81

B
backquote operator, 66
bcheck command, 141
  examples, 142
  syntax, 141
bind command, 238
block local operator, 67
breakpoints
  clearing, 98–99
  defined, 31, 87
  deleting, using handler ID, 99
  disabling, 99
  enabling, 99
  enabling after event occurs, 269–270
  event efficiency, 99–100
  event specifications, 252–254
breakpoints (Continued)
In Function, 89
In Object, 91
listing, 98–99, 99
multiple, setting in nonmember functions, 91
On Value Change, 93
overview, 88–94
setting
at a line, 32, 88–89
at a member function of a template class or at a
template function, 189
at all instances of a function template, 189
at an address, 227
at class template instantiations, 185, 189
at function template instantiations, 185, 189
in a function, 31, 89–90
in all member functions of a class, 90–91
in an explicitly loaded library, 243
in dynamically loaded libraries, 98
in member functions of different classes, 90
in native (JNI) code, 211
in objects, 91
in shared libraries, 242
machine level, 227
multiple breaks in C++ code, 90
on Java methods, 211
with filters that contain function calls, 95
setting filters on, 94
stop type, 88
determining when to set, 61
trace type, 88
when type, 88
setting at a line, 97–98
bsearch command, 279

C++
ambiguous or overloaded functions, 62
backquote operator, 66
class
declarations, looking up, 71–73
definition, looking up, 72–73
C++, class (Continued)
displaying all the data members directly defined
by, 108
displaying all the data members inherited
from, 108
printing the declaration of, 72
seeing inherited members, 73
viewing, 71–73
compiling with the -g option, 44
compiling with the -g0 option, 44
double-colon scope resolution operator, 66
exception handling, 182–185
function template instantiations, listing, 71
inherited members, 73
mangled names, 68
object pointer types, 108–109
printing, 108–109
setting multiple breakpoints, 90–91
template debugging, 185
template definitions
displaying, 71
fixing, 156
tracing member functions, 97
unnamed arguments, 109
using dbx with, 181–182
C++ application that embeds a Java application,
debugging, 209–210
C++ source files, specifying the location of, 210
C application that embeds a Java application,
debugging, 209–210
c_array_op environment variable, 55
C source files, specifying the location of, 210
call command
syntax, 279–280
using to call a function, 84
using to call a function explicitly, 84
using to call a procedure, 84, 247
using to explicitly call a function instantiation or
member function of a class template, 189
call stack, 101
defined, 101
deleting
all frame filters, 104
frames, 104
call stack (Continued)
   finding your place on, 102
   frame, defined, 101
   hiding frames, 104
   looking at, 33
   moving
      down, 103
      to a specific frame in, 103
      up, 102
   popping, 103–104, 154, 246
   one frame of, 155
   removing the stopped in function from, 103
   walking, 63, 102

calling
   a function, 84–85
   a function instantiation or a member function of a
      class template, 189
   a procedure, 247
   member template functions, 186

cancel command, 281

casesensitivity, Fortran, 192

catch blocks, 182

catch command, 177, 178, 281

catch signal list, 177

catching exceptions of a specific type, 183

clear command, 284

clearing breakpoints, 98–99

code compiled without -g option, 47

collector archive command, 286

collector command, 285–292

collector dbxsample command, 286

collector disable command, 287

collector enable command, 287

collector heaptrace command, 287

collector hw_profile command, 287–288

collector limit command, 288

collector mpitrace command, 288

collector pause command, 289

collector profile command, 289

collector resume command, 289

collector sample command, 289–290

collector show command, 290

collector status command, 290

collector store command, 291

collector synctrace command, 291

collector tha command, 292

collector version command, 292

collections
   displaying all the data members directly defined
      by, 108
   displaying all the data members inherited from, 108
   looking up declarations of, 71–73
   looking up definitions of, 72–73
   printing the declarations of, 72
   seeing inherited members, 73
   viewing, 71–73

CLASSPATHX environment variable, 55, 206

clear command, 284

clearing breakpoints, 98–99

code compiled without -g option, 47

collector archive command, 286

collector command, 285–292

collector dbxsample command, 286

collector disable command, 287

collector enable command, 287

collector heaptrace command, 287

collector hw_profile command, 287–288

collector limit command, 288

collector mpitrace command, 288

collector pause command, 289

collector profile command, 289

collector resume command, 289

collector sample command, 289–290

collector show command, 290

collector status command, 290

collector store command, 291

collector synctrace command, 291

collector tha command, 292

collector version command, 292

collections
   displaying all the data members directly defined
      by, 108
   displaying all the data members inherited from, 108
   looking up declarations of, 71–73
   looking up definitions of, 72–73
   printing the declarations of, 72
   seeing inherited members, 73
   viewing, 71–73

CLASSPATHX environment variable, 55, 206

clear command, 284

clearing breakpoints, 98–99

code compiled without -g option, 47

collector archive command, 286

collector command, 285–292

collector dbxsample command, 286

collector disable command, 287

collector enable command, 287

collector heaptrace command, 287

collector hw_profile command, 287–288

collector limit command, 288

collector mpitrace command, 288

collector pause command, 289

collector profile command, 289

collector resume command, 289

collector sample command, 289–290

collector show command, 290

collector status command, 290

collector store command, 291

collector synctrace command, 291

collector tha command, 292

collector version command, 292

commands
   alias, 44
   assign
      syntax, 277–278
      using to assign a value to a variable, 111, 246
      using to reassign correct values to global
         variables, 155
      using to restore a global variable, 155
   attach, 80, 278–279
   bcheck, 141
   bind, 238
   bsearch, 279
commands (Continued)
call
  syntax, 279–280
  using to call a function, 84
  using to call a procedure, 84, 247
  using to explicitly call a function instantiation or
  member function of a class template, 189
cancel, 281
catch, 177, 178, 281
clear, 284
collector, 285–292
  collector archive, 286
  collector dbxsample, 286
  collector disable, 287
  collector enable, 287
  collector heaptrace, 287
  collector hw_profile, 287–288
  collector limit, 288
  collector mpi TRACE, 288
  collector pause, 289
  collector profile, 289
  collector resume, 289
  collector sample, 289–290
  collector show, 290
  collector status, 290
  collector store, 291
  collector sync TRACE, 291
  collector tha, 292
  collector version, 292
cont, 153, 160
  continuing execution of your program with, 83, 122
  limitations for files compiled without debugging
  information, 152
  syntax, 292
  using to continue execution after restoring a
  global variable, 155
  using to continue execution of your program
  from a different line, 83, 154, 248
dalias, 293
dbx, 37, 42, 293–295
dbxenv, 44, 54, 296
commands (Continued)
debug
  syntax, 296–299
  using to attach dbx to a running process, 80
  using to attach to a child process, 163
  using to debug a core file, 39
delete, 299
detach, 48–49, 81, 299–300
dis, 63, 224, 300–301
display, 110, 301–302
down, 103, 302
dump, 303
dump
  using on OpenMP code, 170
edit, 303
examine, 63, 221–224, 304–305
exception, 182, 305
exists, 305–306
file, 62, 63, 306
files, 306–307
fix, 152, 153, 247, 307
  effects of, 153
  limitations for files compiled without debugging
  information, 152
fixed, 308
fortran_modules, 308
frame, 103, 308
func, 62–63, 63, 309
funcs, 309–310
gdb, 310–311
handler, 252, 311
hide, 104, 311–312
ignore, 176, 177, 312
import, 312–313
intercept, 183, 313
java, 314
jclasses, 314
joff, 314
jon, 315
jpkgs, 315
kill, 49, 128, 315–316
language, 316
line, 63, 316–317
list, 63
commands, list (Continued)  
syntax, 317–318  
using to print a source listing for a file or function, 63  
using to print the source listing for a function instantiation, 190  
listi, 224–225, 319  
loadobject, 319–323  
loadobject -dumpelf, 320  
loadobject -exclude, 320  
loadobject -hide, 321  
loadobject -list, 321–322  
loadobject -load, 322  
loadobject -unload, 322  
loadobject -use, 322–323  
lwp, 323–324  
lwps, 162, 324  
macro, 271, 324  
mmapfile, 324–325  
module, 75, 325–326  
modules, 75, 326  
native, 326–327  
next, 82, 327–328  
nexti, 225, 328–329  
omp_loop, 329  
omp_pr, 329–330  
omp_serialized, 330  
omp_team, 330–331  
omp_tr, 331  
pathmap, 76, 153, 275, 331–333  
using to map the compile-time directory to the debug-time directory, 43–44  
pop, 65, 103, 155, 246  
syntax, 333  
print  
syntax, 333–336  
syntax to slice a C or C++ array, 111  
syntax to slice a Fortran array, 112  
using to dereference a pointer, 110  
using to evaluate a function instantiation or a member function of a class template, 190  
using to evaluate a variable or expression, 108  
using to print the value of an expression, 247  
proc, 336  
commands (Continued)  
process control, 79  
prog, 337  
quit, 337  
regs, 228–236, 338  
replay, 49, 52, 338–339  
rerun, 339  
restore, 49, 51, 339  
rprint, 340  
rtc showmap, 340  
rtc skipatch, 341  
run, 79, 341  
runargs, 342–343  
save, 49, 343  
scopes, 343  
search, 343–344  
showblock, 121, 344  
showleaks  
default output of, 131  
error limit for, 133  
report resulting from, 127  
syntax, 344–345  
using to ask for a leaks report, 130  
showmemuse, 131, 345  
source, 345  
status, 346  
step, 82, 182, 346–348  
step to, 33, 82, 347  
step up, 82, 347  
stepi, 225, 348–349  
stop, 189  
syntax, 349–353  
using to set breakpoint at all member functions of a C++ template class, 189  
using to set breakpoints at all instances of a function template, 189  
using to stop in all member functions of a C++ template class, 189  
stop change, 93  
stop in class, 90  
stop in member, 90  
stopi, 227, 353–354  
suppress  
syntax, 354–356
commands, suppress (Continued)
   using to limit reporting of runtime checking errors, 122
   using to list errors being suppressed in files not compiled for debugging, 134
   using to manage runtime checking errors, 134
   using to suppress runtime checking errors, 132
sync, 356
syncs, 357
that alter the state of your program, 246–248
thread, 159, 357–358
uncheck, 121, 364–365
undisplay, 110, 365–366
unhide, 104, 366
unintercept, 183, 366–367
unsuppress, 132, 134, 367–368
unwatch, 368
up, 102, 368
use, 369
watch, 110, 369
whatis, 71, 72, 271
   syntax, 369–371
   using to display the definitions of templates and instances, 188
   using to obtain the function name assigned by the compiler, 109
when, 97, 247, 250, 371–372
wheni, 373
where, 102, 196, 373–374
whereami, 374–375
whereis, 68, 107, 187–188, 272, 375
which, 63, 69, 107, 271, 375–376
whocatches, 184, 376
x, 221–224
compiling
   optimized code, 45
   with the -g option, 44
   with the -g0 option, 44
   your code for debugging, 27
cond event, 255
cont command
   continuing execution of your program with, 83, 122
   limitations for files compiled without debugging information, 152
   syntax, 292
   using to continue execution after restoring a global variable, 155
   using to continue execution of your program after fixing, 153
   using to continue execution of your program from a different line, 83, 154, 248
   using to resume execution of a multithreaded program, 160
   continuing execution of a program, 83–84
   after fixing, 153
   at a specified line, 83, 248
controlling the speed of a trace, 97
core file
   debugging, 30, 38–42
   debugging mismatched, 40–42
   examining, 30–31
core_lo_pathmap environment variable, 55
   -count event specification modifier, 263
creating
   a .dbxrc file, 54
   event handlers, 251
current address, 63
current procedure and file, 191
customizing dbx, 53

D
dalias command, 293
data change event specifications, 254–255
data member, printing, 71
dbx, starting, 37
   startup options, 295
   with core file name, 38–42
   with process ID only, 42
dbx command, 37, 42, 293–295
dbx commands
   Java expression evaluation in, 216
   static and dynamic information used by when debugging Java code, 216–217
dbx commands (Continued)
using in Java mode, 215–219
valid only in Java mode, 219
with different syntax in Java mode, 218
with identical syntax and functionality in Java mode
and native mode, 217
dbx environment variable, mt_resume_one, 56
dbx environment variables, 55
and the Korn shell, 60
array_bounds_check, 55
C_array_op, 55
CLASSPATH, 55, 206
core_lo_pathmap, 55
disassembler_version, 55
event_safety, 55
fix_verbose, 55
follow_fork_inherit, 55, 164
follow_fork_mode, 55, 135, 164
follow_fork_mode_inner, 55
for Java debugging, 206
input_case_sensitive, 56, 192
JAVASRCPATH, 56, 206
jdbx_mode, 56, 206
jvm_invocation, 56, 206
language_mode, 56
macro_expand, 56, 272
macro_source, 56, 272
mt_scalable, 57
mt_sync_tracing, 57
output_auto_flush, 57
output_base, 57
output_class_prefix, 57
output_derived_type, 109
output_dynamic_type, 57, 182
output_inherited_members, 57
output_list_size, 57
output_log_file_name, 57
output_max_string_length, 57
output_no_literal, 57
output_pretty_print, 57
output_pretty_printFallback, 57
output_short_file_name, 57
overload_function, 57
dbx environment variables (Continued)
overload_operator, 57
pop_auto_destruct, 57
proc_exclusive_attach, 58
rtc_auto_continue, 58, 122, 142
rtc_auto_suppress, 58, 133
rtc_biu_at_exit, 58, 131
rtc_error_limit, 58, 133
rtc_error_log_file_name, 58, 122, 142
rtc_error_stack, 58
rtc_inherit, 58
rtc_mel_at_exit, 58
run_autostart, 58
run_io, 58
run_pty, 58
run_quick, 59
run_savetty, 59
run_setpgrp, 59
scope_global_enums, 59
scope_look_aside, 59, 70
session_log_file_name, 59
setting with the dbxenv command, 54–60
show_static_members, 59
stack_find_source, 59, 65
stack_max_size, 59
stack_verbose, 59
step_abflow, 59
step_events, 100
step_events, 59
step_granularity, 60, 82
suppress_startup_message, 60
symbol_info_compression, 60
trace_speed, 60, 97
vd1_mode, 60
dbx modes for debugging Java code, 214
dbxenv command, 44, 54, 296
.dbxrc file, 53
creating, 54
sample, 54
use at dbx startup, 43, 53
dbxrc file, use at dbx startup, 43, 53
dbxtool, 27, 37
dbgr command, 65
syntax, 296–299
debug command (Continued)
using to attach dbx to a running process, 80
using to attach to a child process, 163
using to debug a core file, 39
debug_file_directory environment variable, 55
debugging
assembly language, 221
child processes, 163
code compiled without -g option, 47
core file, 30, 38–42
machine-instruction level, 221, 225–227
mismatched core file, 40–42
multithreaded programs, 157
optimized code, 46
debugging information
for a module, reading in, 75
for all modules, reading in, 75
debugging run saved
replaying, 52
restoring, 51
saving, 49–52
declarations, looking up (displaying), 71–73
delete command, 299
deleting
all call stack frame filters, 104
call stack frames, 104
specific breakpoints using handler IDs, 99
dereferencing a pointer, 109–110
detach command, 48–49, 81, 299–300
detach event, 259
detaching
a process from dbx, 48–49, 81–82
a process from dbx and leaving it in a stopped state, 82
determining
cause of floating point exception (FPE), 178
location of floating point exception (FPE), 178
number of instructions executed, 269
number of lines executed, 269
the granularity of source line stepping, 82
where your program is crashing, 30
which symbol dbx uses, 69
differences between Korn shell and dbx commands, 237
dis command, 63, 224, 300–301
-disable event specification modifier, 263
disassembler_version environment variable, 55
display command, 110, 301–302
displaying
a stack trace, 104
all the data members directly defined by a class, 108
all the data members inherited from a base class, 108
an unnamed function argument, 109
declarations, 71–73
definitions of templates and instances, 185
inherited members, 72
source code for function template instantiations, 186
symbols, occurrences of, 68–69
template definitions, 71
the definitions of templates and instances, 188
type of an exception, 182
variable type, 72
variables and expressions, 110
dlopen event, 255–256
documentation, accessing, 23
documentation index, 23
down command, 65, 103, 302
dump command, 303
dump command, using on OpenMP code, 170
dynamic linker, 241–242

E
edit command, 303
enabling a breakpoint after an event occurs, 269–270
error suppression, 132–135
default, 134
equations, 133
types, 132
establishing a new mapping from directory to directory, 43, 76
evaluating
a function instantiation or a member function of a class template, 190
evaluating (Continued)
   an unnamed function argument, 109
   arrays, 111–115
event counters, 252
event handler
   hiding, 264
   retaining across debugging sessions, 264
event handlers
   creating, 251
   manipulating, 251
   setting, examples, 268–270
event_safety environment variable, 55
event-specific variables, 266
event specification modifiers
   -count, 263
   -disable, 263
   -hidden, 264
   -if, 262
   -in, 262
   -instr, 263
   -lwp, 264
   -perm, 264
   -resumeone, 96, 262
   -temp, 263
   -thread, 264
event specifications, 227, 249, 251, 252–262
   access, 254–255
   at, 253
   attach, 259
   change, 255
   cond, 255
   detach, 259
dlopen, 255–256
exit, 258
fault, 256
for breakpoint events, 252–254
for data change events, 254–255
for execution progress events, 258–259
for other types of events, 259
for system events, 255
in, 252
inclass, 254
infile, 253
infunction, 253

event specifications (Continued)
   inmember, 253
   inmethod, 253
   inobject, 254
   keywords, defined, 252
   lastrites, 260
   lwp_exit, 257
   modifiers, 262–264
   next, 258
   omp_barrier, 171
   omp_critical, 172
   omp_flush, 172
   omp_master, 172
   omp_ordered, 172
   omp_single, 172
   omp_task, 172
   omp_taskwait, 171
   prog_gone, 260
   prog_new, 260
   returns, 258, 259
   setting, 252–262
   sig, 257
   step, 259
   stop, 260
   sync, 260–261
   syncrtld, 261
   sysin, 257–258
   sysout, 258
   thr_create, 161, 261
   thr_exit, 161, 261
   throw, 261, 262
timer, 262
   using predefined variables, 265

events
   ambiguity, 264
   child process interaction with, 164
   parsing, 264
examine command, 63, 221–224, 304–305
examining the contents of memory, 221–225
exception command, 182, 305
exception handling, 182–185
   examples, 184
exceptions
   floating point, determining cause of, 178
exceptions (Continued)
   floating point, determining location of, 178
   in Fortran programs, locating, 196
   of a specific type, catching, 183
   removing types from intercept list, 183
   reporting where type would be caught, 184
   type of, displaying, 182
exec function, following, 164
execution progress event specifications, 258–259
exists command, 305–306
exit event, 258
experiments, limiting the size of, 288
expressions
   complex, Fortran, 199
   displaying, 110
   interval, Fortran, 200
   monitoring changes, 110
   monitoring the value of, 110
   printing the value of, 108, 247
   turning off the display of, 110

F
fault event, 256
fflush(stdout), after dbx calls, 85
field type
   displaying, 72
   printing, 72
file command, 62, 63, 65, 306
files
   finding, 43–44, 76
   location of, 76
   navigating to, 62
   qualifying name, 66–68
files command, 306–307
finding
   object files, 43–44
   source files, 43–44, 76
   your place on the call stack, 102
fix and continue (Continued)
   using with shared objects, 242
fix command, 152, 153, 247, 307
   effects of, 153
   limitations for files compiled without debugging information, 152
fix_verbose environment variable, 55
fixed command, 308
fixing
   C++ template definitions, 156
   shared objects, 152
   your program, 153, 247
floating point exception (FPE)
   catching, 270
   determining cause of, 178
   determining location of, 178
follow_fork_inherit environment variable, 55, 164
follow_fork_mode environment variable, 55, 135, 164
follow_fork_mode_inner environment variable, 55
following
   the exec function, 164
   the fork function, 164
fork function, following, 164
Fortran
   allocatable arrays, 198
   array slicing syntax for, 112–113
   case sensitivity, 192
   complex expressions, 199
   derived types, 201
   interval expressions, 200
   intrinsic functions, 199
   logical operators, 201
   structures, 201
fortran_modules command, 308
FPE signal, trapping, 177–178
frame, defined, 101
frame command, 65, 103, 308
func command, 62–63, 63, 65, 309
funcs command, 309–310
function argument, unnamed
   displaying, 109
   evaluating, 109
function template instantiations
   displaying the source code for, 186
function template instantiations (Continued)
   printing a list of, 185, 187
   printing the values of, 186
functions
   ambiguous or overloaded, 62
   calling, 84–85
   currently being executed, changing, 154
   executed, changing, 154
   inlined, in optimized code, 47
   instantiation
      calling, 189
      evaluating, 190
      printing source listing for, 190
   intrinsic, Fortran, 199
   looking up definitions of, 71–72
   member of a class template, calling, 189
   member of class template, evaluating, 190
   navigating to, 62–63
   not yet called, changing, 154
   obtaining names assigned by the compiler, 109
   presently on the stack, changing, 154
   qualifying name, 66–68
   setting breakpoints in, 89–90
   setting breakpoints in C++ code, 90–91

I
   -if event specification modifier, 262
   ignore command, 176, 177, 312
   ignore signal list, 177
   import command, 312–313
   in event, 252
   -in event specification modifier, 262
   In Function breakpoint, 89
   In Object breakpoint, 91
   inclass event, 254
   infile event, 253
   infunction event, 253
   inherited members
      displaying, 72
      seeing, 73
   inmember event, 253
   inmethod event, 253
   inobject event, 254
   input_case_sensitive environment variable, 56, 192
   instances, displaying the definitions of, 185, 188
   -instr event specification modifier, 263
   Intel registers, 231
   intercept command, 183, 313
   invocation options, 295

J
   JAR file, debugging, 207–208
   Java applications
      attaching dbx to, 208
      specifying custom wrappers for, 212
      starting to debug, 207
      that require 64-bit libraries, 209
      types you can debug with dbx, 207
      with wrappers, debugging, 208
   Java class file, debugging, 207
   Java code
      capabilities of dbx with, 205
      dbx modes for debugging, 214
      limitations of dbx with, 206
      using dbx with, 205–206
   java command, 314
   Java debugging, environment variables for, 206
   Java source files, specifying the location of, 210
Index

JAVASRCPATH environment variable, 56, 206
jclasses command, 314
jdbx_mode environment variable, 56, 206
joff command, 314
jon command, 315
jpkg command, 315
jvm_invocation environment variable, 56, 206
JVM software
    customizing startup of, 211
    passing run arguments to, 210, 212
    specifying 64-bit, 214
    specifying a path name for, 212

K
key bindings for editors, displaying or modifying, 238
kill command, 49, 128, 315–316
killing
    program, 49
    program only, 49
Korn shell
    differences from dbx, 237
    extensions, 238
    features not implemented, 237
    renamed commands, 238

L
language command, 316
language_mode environment variable, 56
lastrites event, 260
LD_AUDIT, 138
LD_PRELOAD, 139
libraries
    dynamically loaded, setting breakpoints in, 98
    shared, compiling for dbx, 48
librt.so, preloading, 138
librtld.db.so, 241
libthread.db.so, 157
libthread.so, 157
limiting the experiment size, 288
line command, 63, 316–317
link map, 241
linker names, 68
list command, 63, 65
    syntax, 317–318
    using to print a source listing for a file or function, 63
    using to print the source listing for a function instantiation, 190
listl command, 224–225, 319
listing
    all program modules that contain debugging information, 75
    breakpoints, 98–99, 99
    C++ function template instantiations, 71
    debugging information for modules, 75
    names of all program modules, 75
    names of modules containing debugging information that have already been read into dbx, 75
    signals currently being ignored, 177
    signals currently being trapped, 177
    traces, 99
loading your program, 28–29
loadobject, defined, 241
loadobject -dumpelf command, 320
loadobject -exclude command, 320
loadobject -hide command, 321
loadobject -list command, 321–322
loadobject -load command, 322
loadobject -unload command, 322
loadobject -use command, 322–323
loadobject command, 319–323
looking up
    definitions of classes, 72–73
    definitions of functions, 71–72
    definitions of members, 71–72
    definitions of types, 72–73
    definitions of variables, 71–72
    the this pointer, 72
lwp command, 323–324
    -lwp event specification modifier, 264
    lwp_exit event, 257
LWPs (lightweight processes), 157
    information displayed for, 162
    showing information about, 162
\wps command, 162, 324

M
machine-instruction level
  address, setting breakpoint at, 227
  AMD64 registers, 233
debugging, 221
  Intel registers, 231
  printing the value of all the registers, 228
  setting breakpoint at address, 227
  single stepping, 225–226
  SPARC registers, 230
  tracing, 226–227
macro
  definitions, 272–274
  expansion, 271–272
  skimming, 274
macro command, 271, 324
macro\_expand environment variable, 56, 272
macro\_source environment variable, 56, 272
manipulating event handlers, 251
member functions
  printing, 71
  setting multiple breakpoints in, 90–91
  tracing, 97
member template functions, 186
members
  declarations, looking up, 71–73
  looking up declarations of, 71–73
  looking up definitions of, 71–72
  viewing, 71–73
memory
  address display formats, 223
  display modes, 221–225
  examining contents at address, 221–225
  states, 124
memory access
  checking, 124
    turning on, 35, 121
  error report, 125
  errors, 125, 146–149
memory leak
  checking (Continued)
    turning on, 35, 121
    errors, 127, 149–150
    fixing, 130–131
    report, 128–130
memory use checking, 131–132
  turning on, 35, 121
\mmap file command, 324–325
modifying a header file, 156
module command, 75, 325–326
modules
  all program, listing, 75
  containing debugging information that have already been read into dbx, listing, 75
  current, printing the name of, 75
  listing debugging information for, 75
  that contain debugging information, listing, 75
modules command, 75, 326
monitoring the value of an expression, 110
moving
  down the call stack, 103
to a specific frame in the call stack, 103
up the call stack, 102
mt\_resume\_one environment variable, 56
mt\_scalable environment variable, 57
mt\_sync\_tracing environment variable, 57
multithreaded programs, debugging, 157

N
native command, 326–327
navigating
  through functions by walking the call stack, 63
to a file, 62
to functions, 62–63
next command, 82, 327–328
next event, 258
next\_i command, 225, 328–329

O
object files, finding, 43–44
object pointer types, 108–109
Index

obtaining the function name assigned by the compiler, 109
omp_loop command, 329
omp_pr command, 329–330
omp_serialize command, 330
omp_team command, 330–331
omp_tr command, 331
OpenMP application programming interface, 165
OpenMP code
dbx functionality available for, 166
events for, other, 172
events for, synchronization, 171–172
execution sequence of, 173
printing a description of the current loop, 169
printing a description of the current parallel region, 167
printing a description of the current task region, 168
printing all the threads on the current tea, 169
printing shared, private, and thread private variables in, 167
serializing the execution of the next encountered parallel region, 170
single stepping in, 166
transformation by compilers, 165
using stack traces with, 170
using the dump command on, 170
operators
backquote, 66
block local, 67
C++ double colon scope resolution, 66–67
optimized code
compiling, 45
debugging, 46
output_auto_flush environment variable, 57
output_base environment variable, 57
output_class_prefix environment variable, 57
output Derived_type environment variable, 109
output dynamic_type environment variable, 57, 182
output_inherited_members environment variable, 57
output_list_size environment variable, 57
output_log_file_name environment variable, 57
output max_string_length environment variable, 57
output no literal environment variable, 57
output pretty_print environment variable, 57
output short_file_name environment variable, 57
overload_function environment variable, 57
overload_operator environment variable, 57
P
pathmap command, 76, 153, 275, 331–333
  using to map the compile-time directory to the debug-time directory, 43–44
.perm event specification modifier, 264
pointers
dereferencing, 109–110
printing, 203
pop auto destruct environment variable, 57
pop command
syntax, 333
using to change the current stack frame, 65
using to pop frames from the call stack, 246
using to pop one frame from the call stack, 155
using to remove frames from the call stack, 103
popping one frame of the call stack, 155
the call stack, 103–104, 154, 246
predefined variables for event specification, 265
preloading librtc.so, 138
preloading rtcaudit.so, 138
pretty-printing, using, 115
print command
syntax, 333–336
syntax to slice a C or C++ array, 111
syntax to slice a Fortran array, 112
using to dereference a pointer, 110
using to evaluate a function instantiation or a member function of a class template, 190
using to evaluate a variable or expression, 108
using to print the value of an expression, 247
printing
a description of the current loop, 169
a description of the current parallel region, 167
a description of the current task region, 168
a list of all class and function template instantiations, 185, 187
a list of occurrences of a symbol, 68–69
printing (Continued)
a pointer, 203
a source listing, 63
all the threads on the current team, 169
arrays, 111–115
data member, 71
field type, 72
list of all known threads, 160
list of threads normally not printed (zombies), 160
member functions, 71
shared, private, and thread private variables in
OpenMP code, 167
the declaration of a type or C++ class, 72
the name of the current module, 75
the source listing for the specified function
instantiation, 190
the value of a variable or expression, 108
the value of all the machine-level registers, 228
the value of an expression, 247
values of function template instantiations, 186
variable type, 72
proc command, 336
proc_exclusive_attach environment variable, 58
proc_gone event, 260
procedure, calling, 247
procedure linkage tables, 242
process
attached, using runtime checking on, 137–139
child
attaching dbx to, 163
using runtime checking on, 135–137
detaching from dbx, 48–49, 81–82
detaching from dbx and leaving in a stopped
state, 82
running, attaching dbx to, 80–81, 81
stopping execution, 48
stopping with Ctrl+C, 86
process control commands, definition, 79
prog command, 337
prog_new event, 260
program
continuing execution of, 83–84
after fixing, 153
at a specified line, 248
program (Continued)
fixing, 153, 247
killing, 49
multithreaded
debbuging, 157
resuming execution of, 160
resuming execution of at a specific line, 83
running, 79–80
under dbx, impacts of, 245–246
with runtime checking turned on, 121
single stepping through, 82
status, checking, 270
stepping through, 82–86
stopping execution
if a conditional statement evaluates to true, 93
if the value of a specified variable has changed, 93
stripped, 48
Q
qualifying symbol names, 66–68
quit command, 337
quitting a dbx session, 48–49
quitting dbx, 35
R
reading a stack trace, 104
reading in
debbuging information for a module, 75
debbuging information for all modules, 75
registers
AMD64 architecture, 233
Intel architecture, 231
printing the value of, 228
SPARC architecture, 230
regs command, 228–236, 338
removing
exception types from intercept list, 183
the stopped in function from the call stack, 103
replay command, 49, 52, 338–339
replaying a saved debugging run, 52
Index

event specification modifier, 96, 262

execution of a multithreaded program, 160

memory access
    checking, 124
    error report, 125
    errors, 125–126, 146–149

memory leak
    checking, 126–131
    error report, 128–130
    errors, 127, 149–150

memory use checking, 131–132

possible leaks, 127–128

requirements, 120

suppressing errors, 132–135
    default, 134
    examples, 133

suppression of last error, 133

troubleshooting tips, 143

turning off, 121

types of error suppression, 132

using fix and continue with, 139–140

using in batch mode, 141–142

directly from dbx, 142

when to use, 120

S

sample .dbxrc file, 54

save command, 49, 343

saving
    debugging run to a file, 49–52
    series of debugging runs as checkpoints, 51

scope
    changing the visiting, 65
    current, 61, 64
    defined, 64
    lookup rules, relaxing, 70–71
    visiting, 64
        changing, 65
        components of, 64

scope global enums environment variable, 59

scope look aside environment variable, 59, 70

scope resolution operators, 66–68

scope resolution search path, 70

reporting where an exception type would be caught, 184
rerun command, 339
resetting application files for replay, 270
restore command, 49, 51, 339
restoring a saved debugging run, 51
-resumeone event specification modifier, 96, 262
resuming
    execution of a multithreaded program, 160
    program execution at a specific line, 83
returns event, 258, 259
rprint, command, 340
rtc_auto_continue environment variable, 58, 122, 142
rtc_auto_suppress environment variable, 58, 133
rtc_biu_at_exit environment variable, 58
rtc_error_limit environment variable, 58, 133
rtc_error_log_file_name environment variable, 58, 122, 142
rtc_error_stack environment variable, 58
rtc_inherit environment variable, 58
rtc_mel_at_exit environment variable, 58
rtc_showmap command, 340
rtc_skippatch command, 341
rtcaudit.so, preloading, 138
rtld, 241–242
run_autostart environment variable, 58
run command, 79, 341
run_io environment variable, 58
run_pty environment variable, 58
run_quick environment variable, 59
run_savetty environment variable, 59
run_setpgid environment variable, 59
runargs command, 342–343
running a program, 29–30, 79–80
    in dbx without arguments, 30, 79
    with runtime checking turned on, 121
runtime checking
    a child process, 135–137
    access checking, 124–126
    an attached process, 137–139
    application programming interface, 140–141
    error suppression, 132–135
    errors, 146–150

runtime checking (Continued)
    fixing memory leaks, 130–131

memory access
    checking, 124
    error report, 125
    errors, 125–126, 146–149

memory leak
    checking, 126–131
    error report, 128–130
    errors, 127, 149–150

memory use checking, 131–132

possible leaks, 127–128

requirements, 120

suppressing errors, 132–135
    default, 134
    examples, 133

suppression of last error, 133

troubleshooting tips, 143

turning off, 121

types of error suppression, 132

using fix and continue with, 139–140

using in batch mode, 141–142

directly from dbx, 142

when to use, 120
scopes command, 343
search command, 343–344
segmentation fault
  finding line number of, 195
  Fortran, causes, 195
generating, 195
selecting from a list of C++ ambiguous function names, 62
session, dbx
  quitting, 48–49
  starting, 37–38
show_log_file_name environment variable, 59
setting
  a trace, 97
  breakpoints
    at a member function of a template class or at a template function, 189
    at all instances of a function template, 189
    in all member functions a class, 90–91
    in dynamically loaded libraries, 98
    in member functions of different classes, 90
    in native (JNI) code, 211
    in objects, 91
    on Java methods, 211
    with filters that contain function calls, 95
dbx environment variables with the dbxenv command, 54–60
  filters on breakpoints, 94
  multiple breakpoints in nonmember functions, 91
shared libraries
  compiling for dbx, 48
  setting breakpoints in, 242
shared objects
  fixing, 152
  using fix and continue with, 242
show_static_members environment variable, 59
showblock command, 121, 344
show_leaks command
  default output of, 131
  error limit for, 133
  report resulting from, 127
  syntax, 344–345
  using to ask for a leaks report, 130
show_memuse command, 131, 345
sig event, 257
signals
  cancelling, 175
  catching, 176–178
  changing default lists, 177
  forwarding, 175
  FPE, trapping, 177–178
  handling automatically, 179
  ignoring, 177
  listing those currently being ignored, 177
  listing those currently being trapped, 177
  names that dbx accepts, 177
  sending in a program, 179
single stepping
  at the machine-instruction level, 225–226
  through a program, 82
slicing
  arrays, 114
  C and C++ arrays, 111–112
  Fortran arrays, 112–113
source command, 345
source files, finding, 43–44, 76
source listing, printing, 63
SPARC registers, 230
specifying a path for class files that use custom class loaders, 210
stack_find_source environment variable, 59, 65
stack frame, defined, 101
stack_max_size environment variable, 59
stack trace, 196
  displaying, 104
  example, 104, 105
  reading, 104
  using on OpenMP code, 170
stack_verbose environment variable, 59
starting dbx tool, 28
startup options, 295
status command, 346
step_abflow environment variable, 59
step command, 82, 182, 346–348
step event, 259
step_events environment variable, 59, 100
step_granularity environment variable, 60, 82
Index

393
Index

step to command, 33, 82, 347
step up command, 82, 347
stepi command, 225, 348–349
stepping through a program, 32, 82–86
stop at command, 88, 89
stop change command, 93
stop command, 189
  syntax, 349–353
  using to set breakpoint at all member functions of a C++ template class, 189
  using to set breakpoints at all instances of a function template, 189
  using to stop in all member functions of a C++ template class, 189
stop event, 260
stop in class command, 90
stop in member command, 90
stopi command, 227, 353–354
stopping
  a process with Ctrl+C, 86
  in all member functions of a template class, 189
  process execution, 48
  program execution
    if a conditional statement evaluates to true, 93
    if the value of a specified variable has changed, 93
striding across slices of arrays, 114
stripped programs, 48
suppress command
  syntax, 354–356
  using to limit reporting of runtime checking errors, 122
  using to list errors being suppressed in files not compiled for debugging, 134
  using to manage runtime checking errors, 134
  using to suppress runtime checking errors, 132
suppress_startup_message environment variable, 60
suppression of last error, 133
symbol_info_compression environment variable, 60
symbol names, qualifying scope, 66–68
symbols
  choosing among multiple occurrences of, 62–63
  determining which dbx uses, 69
  printing a list of occurrences, 68–69
sync command, 356
sync event, 260–261
syncrtd event, 261
syncs command, 357
sysin event, 257–258
sysout event, 258
system event specifications, 255

T
  -temp event specification modifier, 263
templates
    class, 185
    stopping in all member functions of, 189
    displaying the definitions of, 185, 188
    function, 185
    instantiations, 185
    printing a list of, 185, 187
    looking up declarations of, 72–73
thr_create event, 161, 261
thr_exit event, 161, 261
thread, resuming only the first in which a breakpoint was hit, 96
thread command, 159, 357–358
thread creation, understanding, 161–162
  -thread event specification modifier, 264
threads
  current, displaying, 159
  information displayed for, 158–159
  list, viewing, 160
  other, switching viewing context to, 159
  printing list of all known, 160
  printing list of normally not printed (zombies), 160
  switching to by thread id, 159
threads command, 160, 358–360
throw event, 261, 262
timer event, 262
trace command, 97, 360–363
trace output, directing to a file, 97
trace_speed environment variable, 60, 97
tracei command, 226, 363–364
traces
  controlling speed of, 97
  implementing, 268
  listing, 99
traces (Continued)
setting, 97
tracing at the machine-instruction level, 226–227
trip counters, 252
troubleshooting tips, runtime checking, 143
turning off
  runtime checking, 121
  the display of a particular variable or expression, 110
  the display of all currently monitored variables, 110
turning on
  memory access checking, 35, 121
  memory leak checking, 35, 121
  memory use checking, 35, 121
types
  declarations, looking up, 71–73
  derived, Fortran, 201
  looking up declarations of, 71–73
  looking up definitions of, 72–73
  printing the declaration of, 72
  viewing, 71–73

U
uncheck command, 121, 364–365
undisplay command, 110, 365–366
unhide command, 104, 366
unintercept command, 183, 366–367
unsuppress command, 132, 134, 367–368
unwatch command, 368
up command, 65, 102, 368
use command, 369

V
variable type, displaying, 72
variables
  assigning values to, 111, 246
  changing after fixing, 155–156
  declarations, looking up, 71–73
  determining which dbx is evaluating, 107
  displaying functions and files in which defined, 107
  event specific, 266, 267
variables (Continued)
  examining, 34
  looking up declarations of, 71–73
  looking up definitions of, 71–72
  monitoring changes, 110
  outside of scope, 108
  printing the value of, 108
  qualifying names, 66–68
  turning off the display of, 110
  viewing, 71–73
vdl_mode environment variables, 60
 verifying which variable dbx is evaluating, 107
viewing
  classes, 71–73
  members, 71–73
  the context of another thread, 159
  the threads list, 160
  types, 71–73
  variables, 71–73
visiting scope, 64
  changing, 65
  components of, 64

W
walking the call stack, 63, 102
watch command, 110, 369
what is command, 71, 72, 271
  syntax, 369–371
  using to display the definitions of templates and instances, 188
  using to obtain the function name assigned by the compiler, 109
when breakpoint at a line, setting, 97–98
when command, 97, 247, 250, 371–372
wheni command, 373
where command, 102, 196, 373–374
whereami command, 374–375
whereis command, 68, 107, 187–188, 272, 375
which command, 63, 69, 107, 271, 375–376
whocatches command, 184, 376
Index

X

X command, 221–224