CONVEX CXdb User's Guide

First Edition

CONVEX COMPUTER CORPORATION

CONVEX CXdb User's Guide

Order No. DSW-473

First Edition
October 1991

CONVEX Press
Richardson, Texas
United States of America

CONVEX
CXdb User's Guide

Order No. DSW-473

Copyright © 1991 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OF ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

CONVEX CXdb is a trademark of CONVEX Computer Corporation.

ConvexOS is a trademark of CONVEX Computer Corporation.

COVUE is a trademark of CONVEX Computer Corporation.

CX/Motif is a trademark of CONVEX Computer Corporation.

CXwindows is a trademark of CONVEX Computer Corporation.

Maryland Windows is copyrighted © 1983 University of Maryland Computer Science Department.

UNIX is a registered trademark of AT&T Bell Laboratories.

X-Window and X Window System are trademarks of the Massachusetts Institute of Technology.

Printed in the United States of America

Revision Information for

CONVEX CXdb User's Guide

Edition	Document No.	Description
First	710-015530-001	October 1991. Initial release.

Contents

Using this book xiii

Purpose and audience xiii
Organization xiii
Notational conventions xv
  Command syntax xv
  General conventions xvi
  Notes and cautions xvii
Associated documents xvii
Ordering documentation xvii
Technical assistance xviii
The `contact` utility xviii
Acknowledgments xix

1 Introduction to CXdb 1

General description 1
User interface 2
CXdb concepts 2
  Source units 3
  Eventpoints 4
  Stepping 5
Examples in this book 6
Online guide 6

2 Using CXdb with CXwindows 9

CXwindows 9
Invoking CXdb 10
Loading a program 13
Setting a breakpoint 16
Running a program 18
Printing data 21
Killing the process 22
Passing arguments to the process 22
Running a command in background 25
Stopping a process 28

Getting help online ... 28
Requesting help on a topic ... 31
Getting help on related topics ... 32
Requesting help on a CXdb message ... 34
Using topics lists ... 35
Searching for a topic ... 37
Using search options ... 39
Using the topics history ... 42
Accessing the *CXdb Online Guide* ... 44
Quitting CXdb ... 44

3 Using CXdb with Maryland Windows ... 45

Maryland Windows ... 45
Invoking CXdb ... 46
Loading your program ... 48
Scrolling text in a window ... 50
Moving between windows ... 51
Setting a breakpoint ... 54
Running a program ... 55
Printing data ... 57
Killing a process ... 58
Passing arguments to the process ... 58
Running a command in background ... 60
Stopping a running process ... 63
Getting help online ... 64
Raising and lowering a window ... 67
Moving and resizing a window ... 69
Closing a window ... 73
Quitting CXdb ... 75

4 Breakpoints, tracepoints, and watchpoints ... 77

Preparing for the examples ... 78
Breakpoints ... 78
Tracepoints ... 79
Watchpoints ... 79
Setting a breakpoint ... 80
Setting a breakpoint at a line number ... 80
Setting a breakpoint at a routine ... 82
Getting information about breakpoints ... 83
Removing a breakpoint ... 84
Specifying a source file for a breakpoint ... 85
Removing multiple breakpoints ... 86
Specifying an invalid line number ... 87
Working with tracepoints ... 88
Setting tracepoints ... 88
Getting information about tracepoints ... 89

Working with watchpoints 89
Watching a variable's value 89
Getting information about watchpoints 91
Specifying a watchpoint address range 92
Working with breakpoints, tracepoints, and watchpoints . 96
Quitting the examples 97

5 Stepping 99

Preparing for the examples 99
Stepping concepts 100
Stepping methods 100
Number of steps 100
Step size (granularity) 101
Source units 101
Displaying source units 103
Using source units in CXdb commands 104
Stepping by source units 105
Stepping by statement 106
Stepping by other source units 109
Effect of default granularity on highlighting 114
Finishing a source unit 115
Stepping over the current source unit 118
Stepping to another routine 122
Stepping to the end of the current routine 123
Quitting the examples 124

6 Eventpoints, handlers, and signals 125

Preparing for the examples 126
General eventpoints 126
Working with eventpoints 127
Using reached eventpoints 128
Disabling and enabling eventpoints 128
Setting ignore counts 130
Manipulating multiple eventpoints 132
Eventpoint types 134
Eventpoint handlers 136
Specifying a handler for an eventpoint 136
Specifying a handler for an eventpoint type 141
Changing the default handler for eventpoints 142
Relation eventpoints 143
Debugger variables and eventpoints 145
Signals 146
Signal actions 146
Signal eventpoints 149
Sending a signal to the process 150
Quitting the examples 150

7 Displaying and modifying variables 151
Preparing for the examples 152
Displaying data 152
Displaying information about variables 153
Printing data 155
Formatting printed output 156
Setting print options 157
Modifying data 158
Modifying program data 158
Modifying debugger variables and registers 159
Executing program routines and functions 160
Using other CXdb commands to modify data 161
Examining and modifying memory 161
Examining the contents of memory 161
Searching memory for a byte pattern 163
Modifying the contents of memory 166
Using the examine window in CXwindows 168
Working with array slices 174
Array slices in FORTRAN 175
Array slices in C 177
Using scope paths to access variables 179
Quitting the examples 179

8 Process settings 181
Preparing for the examples 182
Process settings 182
Default process settings 184
Process directory 184
Process environment 185
Floating point mode 189
Display formats 190
Memory unit 192
Fixed scheduling 194
Process shell 194
Search path 194
Stepping granularity 195
SEQ bit 197
SQS bit 197
Signal actions 197
Origin of process setting values 199
Quitting the examples 200

9 Debugging with multiple source files 201

Preparing for the examples 202
Console working directory 202
Search path 203
  Displaying the default search path 205
  Changing the default search path 206
  Displaying the search path 206
  Adding directories to the search path 207
  Removing directories from the search path 208
Initialization files and search paths 208
Displaying a file 210
Displaying a routine 211
Searching source code 212
  Searching forward in the source window 212
  Searching in a different source window 213
  Searching backward in the source window 214
Quitting the examples 215

10 Specifying the program to debug 217

Preparing for the examples 218
Specifying a process to debug 218
Attached processes 219
  Running the program in the shell 219
  Debugging an existing process 220
  Letting an attached process run to completion 222
  Detaching from a process 223
Changing the executable file 224
Issuing a shell command within CXdb 225
Quitting the examples 227

11 Aliases and macros 229

Preparing for the examples 229
Aliases 230
  Displaying aliases 230
  Defining aliases 232
  Redefining and deleting aliases 233
  Using aliases 234
Macros 235
  Defining macros 235
  Displaying macros 236
  Using macros 237
  Redefining and deleting macros 239
Command history 240
Quitting the examples 242

12 Logging, command files, and batch mode........................243

Preparing for the examples 244
Logging CXdb input, output, and messages 244
Logging CXdb output 244
Logging CXdb messages 247
Logging CXdb input 249
Redefining and deleting viewports 251
Using redirection operators 252
Controlling file writes 253
Command files and initialization files 254
Creating a command file 255
Executing a command file 257
Using initialization files 259
Batch mode 261
Quitting the examples 263

13 Scope paths and the stack..................................265

Preparing for the examples 265
Scope 266
Referencing a variable that is not visible 266
Evaluating a variable that is not visible 267
Scope paths 267
Use of scope paths 268
FORTRAN scope paths 269
Getting the current scope 270
Printing a variable from another routine 270
Referencing variables that cannot be evaluated 271
Using scope paths to view common blocks 271
C scope paths 273
Getting the current scope 275
Printing a variable from another routine 275
Referencing variables that cannot be evaluated 276
Block numbering 277
Printing variables from a different language 281
Changing frames to reference variables 283
Stack window 286
Quitting the examples 290

14 Debugging at the instruction level . . . 291

Preparing for the examples . . . 291
Stepping by expression . . . 293
Displaying disassembled code . . . 295
Stepping by instruction . . . 297
The disassembly window in CXwindows . . . 299
Selecting auto update . . . 301
Specifying another address to disassemble . . . 304
Displaying registers . . . 306
Quitting the examples . . . 308

15 Debugging optimized code . . . 309

Source units and optimized code . . . 310
Why source units are important . . . 310
Source unit ranges . . . 311
Optimization of variables . . . 312
Synthesized variables . . . 313
Values that cannot be determined . . . 319
Hints for debugging optimized code . . . 322
Setting eventpoints in optimized code . . . 323
Stepping through optimized code . . . 323
Debugging at various optimization levels . . . 325
Preparing for the examples . . . 325
Level `-no` . . . 326
Level `-O0` . . . 334
Level `-O1` . . . 340
Level `-O2` . . . 346
Level `-O3` . . . 357
Quitting the examples . . . 357

16 Debugging a program with multiple threads . . . 359

Preparing for the examples . . . 360
Threads . . . 360
Optimizations at level `-O3` . . . 361
Fixed scheduling . . . 363
Spawn eventpoints . . . 363
Getting information about threads . . . 365
Making commands thread-specific . . . 366
Sending a signal to a thread . . . 370
Opening thread-specific CXwindows . . . 370
Join eventpoints . . . 371
Quitting the examples . . . 373

A Source code for example program 375

B Quick reference to Maryland Windows 397

C Cross-reference to `csd` commands.............................401

D Cross-reference to `gdb` commands.............................407

Using this book

Purpose and audience

The *CONVEX CXdb User's Guide* describes the CXdb user interface and explains how to use CXdb commands. This book includes numerous examples that you can execute online while reading the text.

This book is intended for both new and experienced users of CXdb. It assumes that you have read the *CONVEX CXdb Concepts* book and that you understand the basic concepts presented there.

Organization

This manual is organized into the following chapters and appendixes:

- **Chapter 1, "Introduction to CXdb"**—Contains an overview of CXdb and a description of its features.
- **Chapter 2, "Using CXdb with CXwindows"**—Explains the CXwindows interface and how to use it to access CXdb.
- **Chapter 3, "Using CXdb with Maryland Windows"**—Explains the Maryland Windows interface and how to use it to access CXdb.
- **Chapter 4, "Breakpoints, tracepoints, and watchpoints"**—Describes how to use breakpoints, tracepoints, and watchpoints to trap various process events.
- **Chapter 5, "Stepping"**—Illustrates the use of stepping commands to perform incremental execution of a program.
- **Chapter 6, "Eventpoints, handlers, and signals"**—Presents the types of eventpoints available in CXdb and explains how to customize them with eventpoint handlers.
- **Chapter 7, "Displaying and modifying variables"**—Shows several methods for displaying and modifying the contents of variables, registers, and memory.

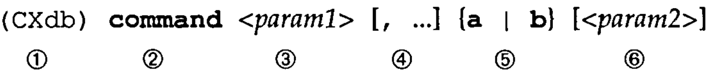
- **Chapter 8, "Process settings"**—Explains the various process settings and how to manipulate these settings to suit your debugging needs.
- **Chapter 9, "Debugging with multiple source files"**—Explains how to debug a program that consists of multiple files located in different directories.
- **Chapter 10, "Specifying the program to debug"**—Describes how to debug attached processes and core files; how to change executables while using CXdb; and how to issue shell commands from within CXdb.
- **Chapter 11, "Aliases and macros"**—Explains how to create and use aliases and macros.
- **Chapter 12, "Logging, command files, and batch mode"**—Describes how to direct CXdb input, output, and messages to files; how to create and use command files and initialization files; and how to run CXdb in batch mode.
- **Chapter 13, "Scope paths and the stack"**—Explains how to use scope paths to reference variables anywhere in the program in both the FORTRAN and C languages. Describes commands that manipulate the stack.
- **Chapter 14, "Debugging at the instruction level"**—Shows how to obtain a more detailed view of your program by using expression source units and disassembled code.
- **Chapter 15, "Debugging optimized code"**—Explains some of the effects produced by optimization of your source code and illustrates how to use CXdb to detect those effects. It includes optimizations that occur at level `-no`.
- **Chapter 16, "Debugging a program with multiple threads"**—Describes CXdb commands designed for the debugging of a multithreaded program. It explains the effect of parallel optimization at level `-O3`.
- **Appendix A, "Source code for example program"**—Lists the source code for the example program used in this book.
- **Appendix B, "Quick reference to Maryland Windows"**—Lists the keyboard functions used in the Maryland Windows interface to CXdb.
- **Appendix C, "Cross-reference to `csd` commands"**—Relates commands of the CONVEX `csd` debugger to the equivalent CXdb commands.
- **Appendix D, "Cross-reference to `gdb` commands"**—Relates commands of the `gdb` debugger to the equivalent CXdb commands.

Notational conventions

This document uses the following notational conventions.

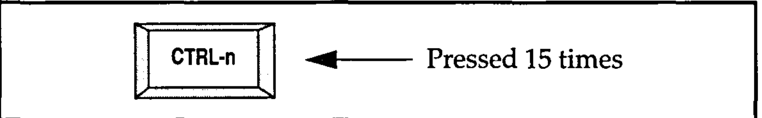
Command syntax

Consider this example:



1. (CXdb) is the CXdb command prompt.
2. **command** must be typed as it appears.
3. <*param1*> indicates a parameter that must be supplied.
4. Horizontal ellipsis in brackets [, ...] indicates that additional parameters can be specified.
5. Either **a** or **b** must be specified.
6. [<*param2*>] indicates an optional user-specified parameter.

Keystrokes that perform particular tasks in CXdb are shown as follows:



The above example indicates that you press CTRL-n 15 times to perform the specified task.

General conventions

- **`Bold constant-width font`** identifies user input in examples.
- *Italics*
  - Designate user-supplied variables in a command-line example when enclosed in angle brackets (<>).
  - Indicate document titles.
  - Introduce important new terms in text.
- `Constant-width font` designates input and output, including:
  - Command names and options
  - System calls
  - Program statements, command output, and error messages returned
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines have been left out of an example.
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- References to the *ConvexOS Man Pages for Users, ConvexOS Man Pages for System Managers,* and *ConvexOS Man Pages for Programmers* appear in the form exec(2), where the name of the man page is followed by its section number enclosed in parentheses.
- The shell prompt is shown as a percent sign (%).
- Unless otherwise indicated, source code examples are in FORTRAN. Where there are differences between how CXdb handles C and FORTRAN, examples in C are also shown.
- FORTRAN examples are shown in uppercase letters. You can, however, use lowercase.

Notes and cautions

Note

A Note highlights supplemental information.

Caution

A Caution highlights procedures or information necessary to avoid damage to software or data.

Associated documents

This manual is not a complete explanation of CXdb. For more information, refer to:

- *CONVEX CXdb Concepts* (DSW-471)—An overview of CXdb and an explanation of how traditional and new debugging concepts are used in CXdb.
- *CONVEX CXdb Reference* (DSW-472)—A complete reference source for CXdb commands, parameters, concepts, and messages.
- *CONVEX CXdb Quick Reference* (DSW-474)—A quick reference for using CXdb, which contains command syntax and brief descriptions.

Ordering documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title.

In some cases, you might not want the latest edition. To order a specific edition of a document, contact your local CONVEX office.

Technical assistance

If you have questions that are not answered by the documentation, contact the CONVEX Technical Assistance Center (TAC). To contact the TAC, use one of the following phone numbers:

- Within the continental U.S., call 1(800)952-0379.
- Outside the continental U.S., contact the local CONVEX office.

The `contact` utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically. The TAC notifies you within 48 hours that your report has been received.

Using `contact` requires:

- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- Full path name of the program or utility in question
- Version number of the program or utility in question

Refer to the contact(1) man page for complete details.

Acknowledgments

The authors wish to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book. In particular:

- The team who developed CXdb: Gary Brooks, Russell Buyse, Mark Chiarelli, Mike Garzione, David Lingle, Steve Simmons, Larry Streepy, and Jeff Woods.
- The team who created tests to help affirm the accuracy of CXdb and this book: Lloyd Tharel and Tim Powell.
- The people who provided vision and management for the product and documentation: Dave Holt, Gary Brooks, Kathy Harris, Mike Turner, and Frank Marshall.
- For her editorial comments that helped make this book more consistent and readable: Mary Clare Bernier.

—Raymond Cetrone and Kenneth S. Harward

Introduction to CXdb

1

The CONVEX Visual Debugger (CXdb) is a symbolic debugger that offers many advanced features not found in conventional debuggers. This chapter gives an overview of CXdb features and the major concepts associated with them.

General description

CXdb provides a controlled environment for executing your programs. It enables you to start and stop a program at any point, display and change the values of program variables, and examine and modify the state of the process stack. It also allows you to control the flow, or order of execution, of your program.

In addition, CXdb provides many advanced features that give you the capability to:

- Debug optimized code
- Debug programs written in FORTRAN, C, or both
- Control execution in increments called source units, which allow you to adjust the size of a step from units as large as a routine to those as small as an individual expression
- Create a break in the execution by monitoring a variety of events or locations such as line numbers, memory addresses, signals, and relational expressions
- Debug at the machine level, with complete access to the machine state, including the process stack and registers
- Attach to and debug a running process
- Debug core files and checkpoint files
- Create aliases, macros, and command files for frequently used functions
- Configure CXdb and customize it to fit your needs

User interface

There are three types of user interfaces for CXdb:

- **CXwindows**—Provides a window environment for X terminals that includes full use of a mouse to perform debugging functions in CXdb.
- **Maryland Windows**—Simulates a window environment on ASCII terminals, but does not include the use of a mouse.
- **Batch mode**—Allows execution of CXdb commands without user interaction.

Both CXwindows and Maryland Windows provide many separate windows that give different views of your program. By using these windows, you can display the maximum amount of information about your program. The windows available in both CXwindows and Maryland Windows are:

- **Command window**—Is the primary window for entering commands to CXdb and receiving output from it.
- **Source window**—Displays the source code for the program you are debugging.
- **Process interface window**—Allows interactive input and output for the program being debugged.
- **Help window**—Displays help text relating to CXdb commands, concepts, messages, and parameters.
- **Display file window**—Displays the contents of any ASCII file.

In addition to the above windows, the CXwindows interface provides the following:

- **Disassembly window**—Displays the assembly language code for the program being debugged. It also displays the processor status word (PSW) as well as the scalar, vector, and communication registers.
- **Examine window**—Displays the contents of process memory.
- **Stack window**—Displays the contents of the process stack.

CXdb concepts

Because CXdb has many features not found in conventional debuggers, some of the terms and concepts used in describing it are new. This section briefly explains these important new terms and concepts. For more detailed descriptions, refer to the *CONVEX CXdb Concepts* book.

Source units

Conventional debuggers generally perform operations on lines of source code. CXdb performs operations on elements called *source units.*

A source unit is a syntactic element of source code. Source units are classified by granularity, as follows:

- **Expression**—Any combination of constants, operators, and operands that is valid in the current source language.
- **Statement**—A combination of expressions that constitutes a complete instruction in the current source language.
- **Block**—The statements that make up the body of a routine, a loop, or a conditional construct.
- **Loop**—A special type of statement that encloses a block and causes it to repeat.
- **Routine**—A complete subroutine or function.

The exact content of a particular type of source unit depends on the syntax of the source language. For example, an expression in C can contain pointers, but in FORTRAN it cannot.

Source units can be nested. For example, a loop can contain many expressions, statements, blocks, and even other loops.

In addition, several source units of different granularities can coexist in the same portion of source code at the same time. For example, `X=1` can be an expression, a statement, and a block, all at the same time.

CXdb uses source units to track source code. When it displays source code in the source window, CXdb highlights the particular source unit that is currently active. This makes it easy for you to locate exactly where the current activity is taking place in your program.

Source units become particularly important when debugging optimized code, because optimization produces machine instructions that are very different than the source code. Source units make it possible to map the optimized code back to the original source code. CXdb does this mapping for you automatically.

Source units also enable you to tell CXdb what size and type of unit you want a command to affect. Conventional debuggers limit you to working only with statements, but CXdb lets you work with routines, blocks, loops, or expressions as well as statements.

A number of CXdb commands let you specify which source unit you want that command to affect. In general, these commands relate to 2 types of activities:

- Setting breakpoints and other types of eventpoints
- Stepping through the execution of your program

The following sections describe eventpoints and stepping.

Eventpoints

A common activity while debugging is to set a breakpoint that halts the execution of your program at a particular location. Conventional debuggers limit you to setting a breakpoint at a line of source code. However, with CXdb you can set breakpoints at source units. This means that you can set a breakpoint at the beginning of an expression, a statement, a block, a loop, or a routine (both before and after the preamble of the routine).

Typically, breakpoints are triggered when execution of the program reaches the specified location in the code. CXdb expands the concept of breakpoints by allowing you to trap other types of events besides locations in the code. These generalized event traps are called *eventpoints*. The types of eventpoints are:

- **Breakpoint**—Stops execution of the program at the specified location. The location can be specified as a machine instruction, a source unit, a line of source code, or the address of a routine.
- **Tracepoint**—Prints a message when execution reaches the specified location in the program, but it does not stop execution. The location can be specified as a machine instruction, a source unit, a line of source code, or the address of a routine.
- **Watchpoint**—Stops execution of the program when the contents of a variable or a memory location is modified.
- **Process-based eventpoint**—Looks for a particular action performed by the program, such as an `exec` system call.
- **Relation-based eventpoint**—Tests the value of a relational expression.
- **Signal-based eventpoint**—Traps a signal sent to the program.
- **Thread-based eventpoint**—Monitors the creation or termination of threads.

In addition to providing many different types of eventpoints, CXdb allows you to define your own handler for each eventpoint. An eventpoint handler is a sequence of CXdb commands that executes automatically whenever the specified event occurs.

Stepping

Stepping is incremental execution of a program. With CXdb, you can step through a program in 2 different ways:

- By machine instructions
- By source units

These methods of stepping give you much greater control over the granularity, or step size, than is available with other debuggers that step only by lines of source code. In addition to specifying the stepping granularity, you can also specify a repeat count with CXdb stepping commands.

In general, CXdb stepping commands allow you to:

- Step by a specified number of machine instructions.
- Step to the beginning of a routine, loop, block, statement, or expression.
- Step out of (finish) a routine, loop, block, statement, or expression.
- Step into a routine, loop, block, statement, or expression.
- Step over a routine, loop, block, statement, or expression.
- Step to the beginning of a source unit within a called routine.
- Step to the beginning of a source unit in the current routine, not counting the source units inside called routines.

Examples in this book

The remaining chapters of this book contain numerous examples of CXdb commands and features. The examples are designed so that you can execute them as you read them. The required files for these examples are in the `/usr/lib/cxdb/examples` directory.

Note

Differences in hardware and in software versions can cause your results to differ from the results shown in the examples. The most noticeable differences will be in memory addresses, process numbers, eventpoint numbers, source unit numbers, and register usage. If the example calls for you to specify one of these items, specify the value that is appropriate for your results rather than the value shown in the example.

Each of the remaining chapters of this book is a separate unit as far as the examples are concerned. The first example in each chapter shows how to invoke CXdb, and the last example shows how to quit CXdb. In between, the examples are sequential.

You can perform the examples in one chapter without performing the examples in other chapters. However, within a chapter, do not try to perform one example unless you have performed all the examples preceding it.

Note

Performing an example out of sequence within a chapter can produce results that are inconsistent with the intent and context of the example.

Online guide

The *CXdb Online Guide* is an online introduction to the CXdb commands. It contains explanatory text as well as many hands-on tutorials. It is available only in the CXwindows interface.

There are 2 ways to access the *CXdb Online Guide*:

- From the shell
- From within CXdb

To access the *CXdb Online Guide* from the shell, enter the commands shown in Figure 1.

Figure 1
Invoking the *CXdb Online Guide*

```
%cd /usr/lib/cxdb/tutorial
%cxdb_guide
```

The `cd` command in Figure 1 changes directories to the one where the *CXdb Online Guide* is located. The `cxdb_guide` command invokes the viewer for the online guide.

To invoke the *CXdb Online Guide* from within CXdb, issue the `help` command. Then select the Tutorial option from the HelpWindow menu. The section "Accessing the *CXdb Online Guide*," in Chapter 2 of this manual, illustrates this method.

Using CXdb with CXwindows

2

This chapter describes the main CXdb windows available through the CXwindows interface. This chapter also gives an overview of basic CXdb commands used to load, run, and quit your program. The commands discussed in this chapter are:

- `break line`
- `continue`
- `cxdb`
- `debug exec`
- `help`
- `info cxdb`
- `info process`
- `kill process`
- `print`
- `quit`
- `rerun`
- `run`
- `stop`

CXwindows

CXwindows is the CONVEX version of the X Window System, which was originally developed at the Massachusetts Institute of Technology. CXwindows is an optional software package that can be purchased separately from CXdb.

The CXwindows interface to CXdb offers the following features:

- Control of window size and location
- Full use of a mouse
- Menus to manage windows and execute CXdb commands

- Scroll bars
- *CXdb Online Guide*, which includes tutorials and sample debugging sessions

The CXwindows interface uses software known as a window manager to control the location, appearance, and operation of the windows. There are a number of different window managers available for CXwindows, but the examples in this book assume the use of CX/Motif as the window manager.

Invoking CXdb

The `cxdb` command, issued from the shell, invokes CXdb. Many command-line options are available with the `cxdb` command. For a complete description of these options, refer to the *CONVEX CXdb Reference* manual or the man page for cxdb(1).

To execute the examples shown in this manual, you must first change to the directory where the example programs are located. Figure 2 shows the steps required to invoke CXdb and to access the example programs.

Figure 2
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb
```

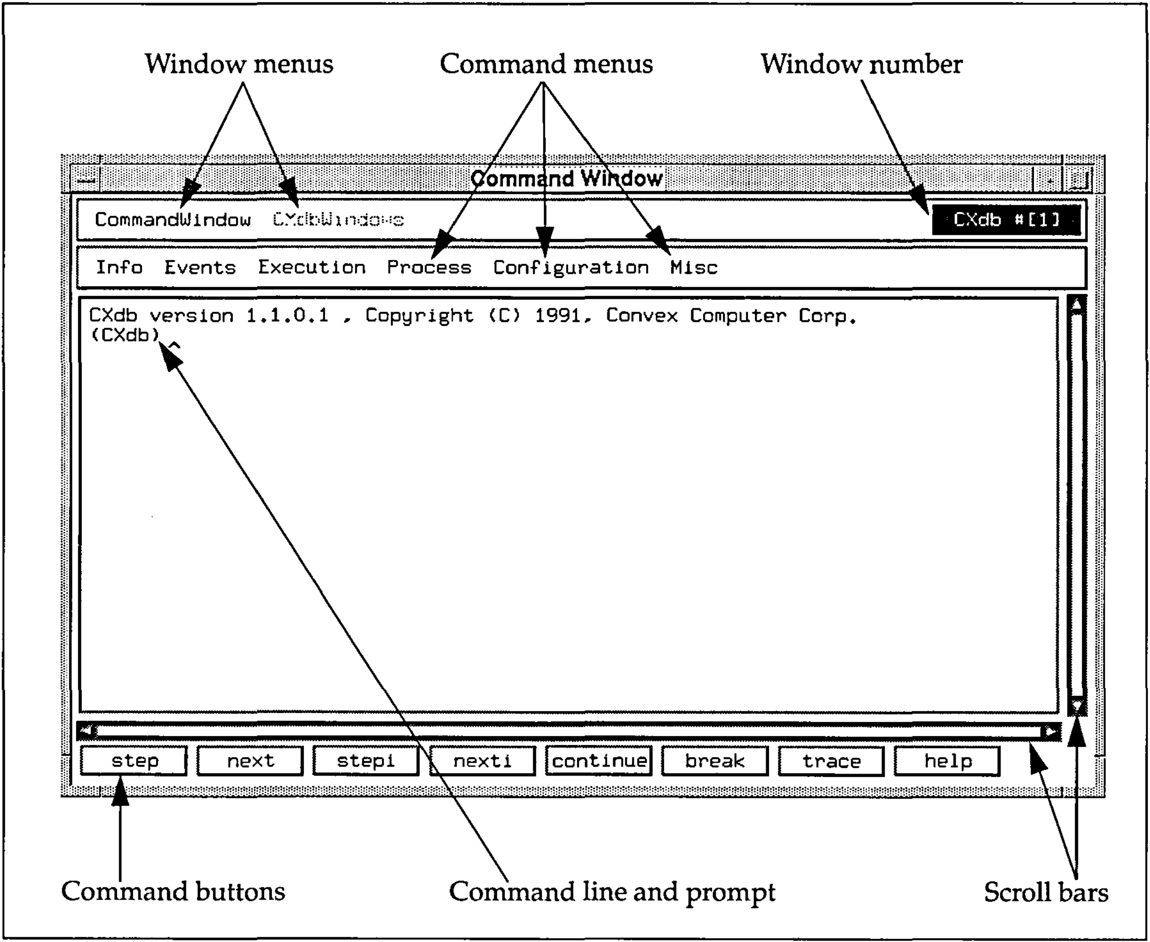
The `cd` command in Figure 2 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb.

When you invoke CXdb, it automatically brings up the command window. The command window is the primary way for you to communicate with CXdb. It enables you to:

- Enter CXdb commands
- Receive output from CXdb commands
- Receive error messages or status information about CXdb commands
- Review previous commands and retrieve them from the command history
- Edit and repeat commands

Figure 3 illustrates the command window in CXwindows. (If you are not using CXwindows, CXdb defaults to using the Maryland Windows interface. Refer to Chapter 3 for a description of Maryland Windows.)

Figure 3
Command window in CXwindows



The command window includes:

- **Window menus**—These pull-down menus allow you to open and close other windows that present different views of your program:
  - CommandWindow quits CXdb and closes the command window.
  - CXdbWindows opens and closes other windows that display additional information about the current process.

- **Window number**—This is a unique number that identifies each window in CXdb. Certain CXdb commands allow you to specify a window number as a parameter.
- **Command menus**—These pull-down menus enable you to execute CXdb commands by using the mouse rather than by entering the commands from the keyboard:
  - Info contains commands that display information about CXdb and the current process.
  - Events contains commands that create and modify various types of eventpoints.
  - Execution contains commands that control the execution of your program.
  - Process contains commands that display more detailed information about the process, such as disassembled code and the contents of process memory.
  - Configuration contains commands for setting and clearing numerous debugging options.
  - Misc contains commands that do not fit into one of the above categories.
- **Command buttons**—These buttons immediately execute the command indicated by the button name. To activate a button, click on it with the mouse.
- **Command line**—This is the area of the window where you enter commands or supply input to CXdb. When the `(CXdb)` prompt is present, you can enter commands by typing them from the keyboard, selecting them from the command menus, or clicking on the command buttons.
- **Scroll bars**—These bars and arrows enable you to scroll the window vertically and horizontally by using the mouse.

Your command window should look similar to the one shown in Figure 3. CXdb has default settings that control the appearance and operation of its windows in the CXwindows interface. You can change these defaults by editing the `.Xdefaults` file in your home directory. For example, the X resource settings control whether or not the command buttons and command menus appear in the command window. Refer to the *CONVEX CXdb Reference* manual for more information about Xdefaults.

Other window decorations, such as the title bar and the resize border, are functions of the window manager used with CXwindows. For information about how to use or modify these decorations, refer to the reference manual for your window manager software.

Loading a program

After invoking CXdb, you can give it the name of the executable file you want to debug. To use the full capabilities of CXdb, you must compile your program by using the `-cxdb` option with the latest release of the CONVEX FORTRAN or CONVEX C compiler. The `-cxdb` option tells the compiler to generate the debugging information CXdb needs.

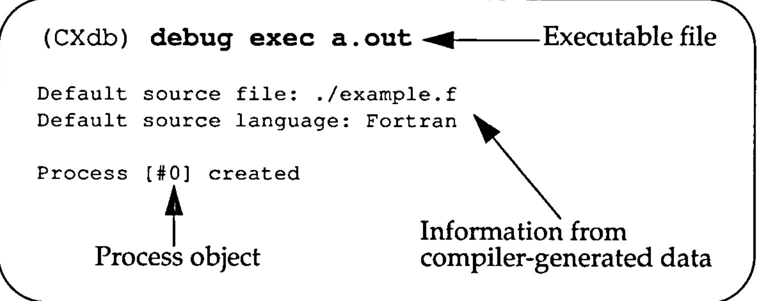
Compiler-generated debugging information is stored in a subdirectory named `.CXdb`. The compiler creates this subdirectory in the same directory as the `.o` (object) file. CXdb must be able to locate this directory and access the data files in order to perform symbolic debugging of your program. You can give CXdb access to the `.CXdb` directory either by invoking CXdb from the directory where the executable file was compiled or by using commands within CXdb to specify the path to the appropriate directory.

Once you have given CXdb access to the compiler-generated data files, you can load an executable file by using the `debug exec` command. This command does the following:

- Gives CXdb the name of the executable file you want to debug
- Creates a process object to store information about the executable file and any processes generated from it
- Maps information from the compiler-generated data files in the `.CXdb` subdirectory to the specified executable file
- Brings up the source window and displays the source file associated with the specified executable file

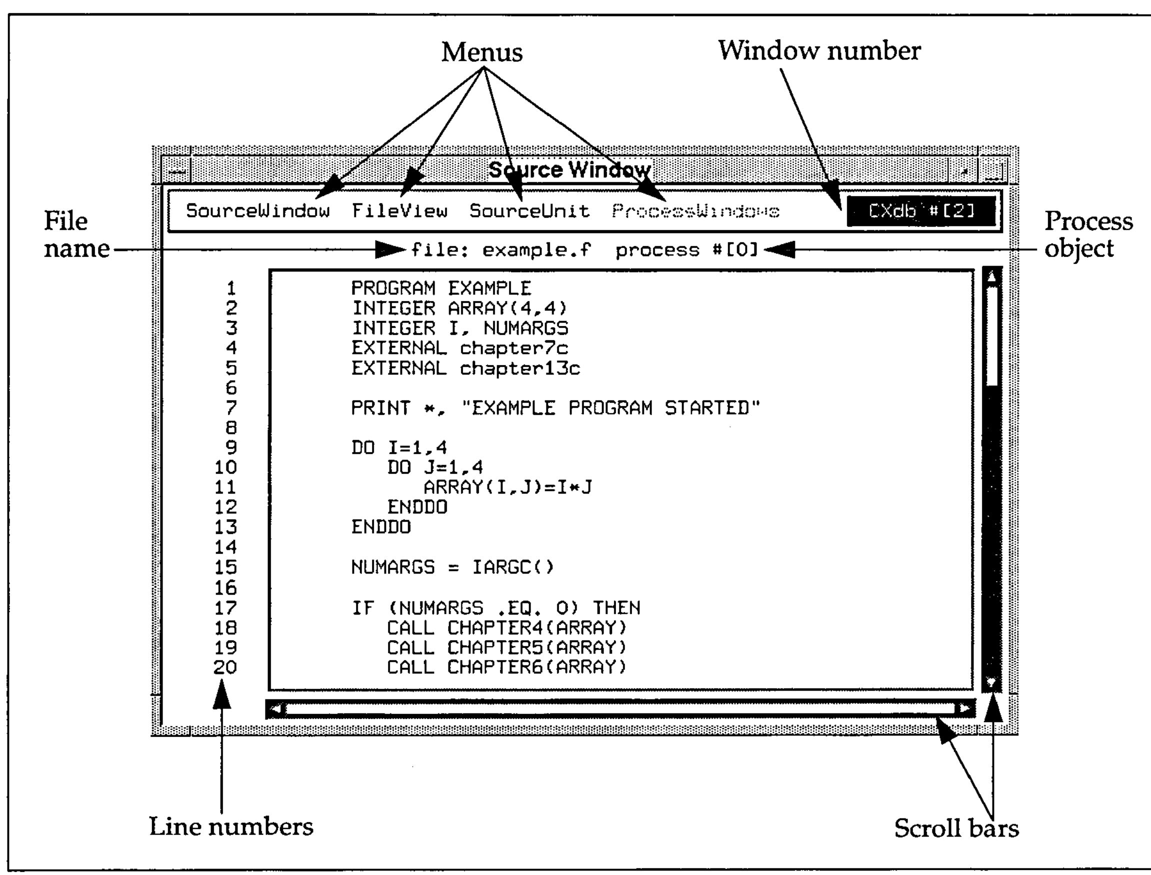
Figure 4 illustrates the use of the `debug exec` command as well as the associated response from CXdb.

Figure 4
Debugging an executable file



CXdb uses the compiler-generated data files to determine the name of the source file corresponding to the main routine of the program. The source window opens to display the main routine, as shown in Figure 5.

Figure 5
Source window in CXwindows



The source window includes:

- **Menus**—These pull-down menus allow you to do the following by using the mouse:
  - SourceWindow closes the source window.
  - FileView lets you specify another file or routine to view in the source window.
  - SourceUnit enables you to select source units from the lines displayed in the source window.
  - ProcessWindows opens and closes other windows that display different information about the current process.

- **Window number**—This is a unique number that identifies each window in CXdb. Certain CXdb commands allow you to specify a window number as a parameter.
- **Scroll bars**—These bars and arrows enable you to scroll the window vertically and horizontally by using the mouse.
- **File name**—This field lists the name of the source file currently displayed in the source window.
- **Process object**—This field lists the identification number of the current process object.
- **Line numbers**—These numbers are added by CXdb to identify each line in the source file.

Setting a breakpoint

Before running a program from within a debugger, it is helpful to set a breakpoint that stops execution of the program before it reaches a problem area of the code. This gives you an opportunity to use other debugger commands to analyze the program while it is stopped. Figure 6 illustrates how to set a simple breakpoint at a line number.

Figure 6
Setting a breakpoint at a line number

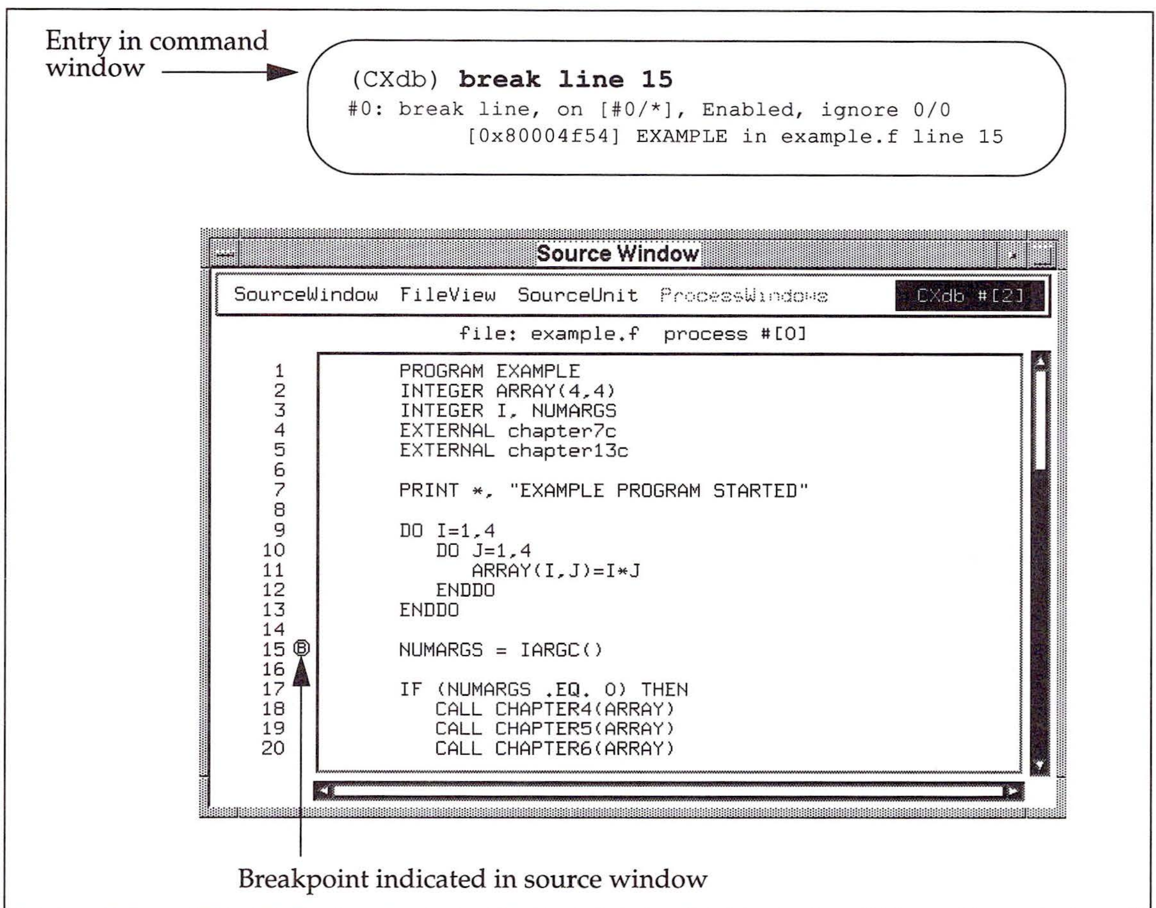
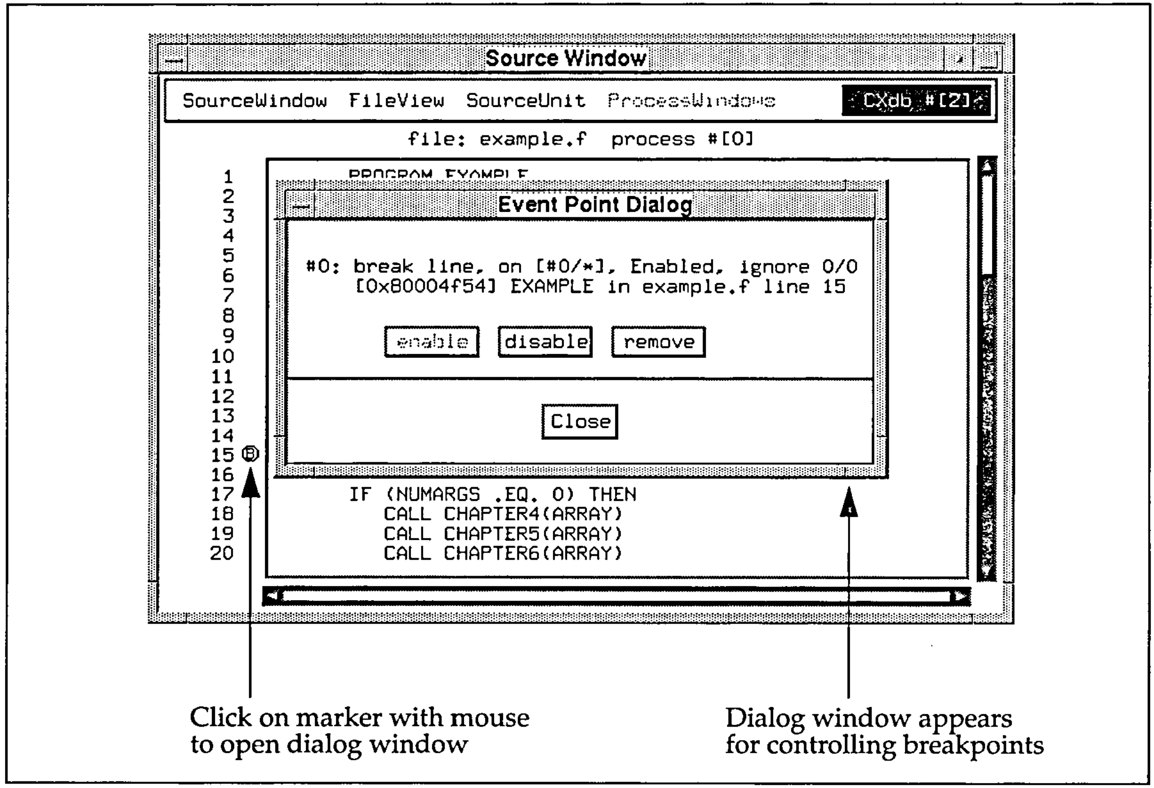


Figure 6 shows that the command `break line 15` is entered in the command window. This command sets a breakpoint at line 15 of the current source file, `example.f`.

Figure 6 also shows that a marker (the letter B enclosed within an octagon) appears in the source window to indicate the location of the breakpoint. You can click on this marker with the left mouse button to open a dialog window for displaying and modifying the breakpoint. Figure 7 illustrates the dialog window for breakpoints.

Figure 7
Dialog window for breakpoints



As Figure 7 shows, the dialog window gives a full description of the breakpoint. It also allows you to enable, disable, or remove the breakpoint. (If you are performing these examples as you read, do not modify the breakpoint at this time. After observing the dialog window, click on its Close button with the mouse.)

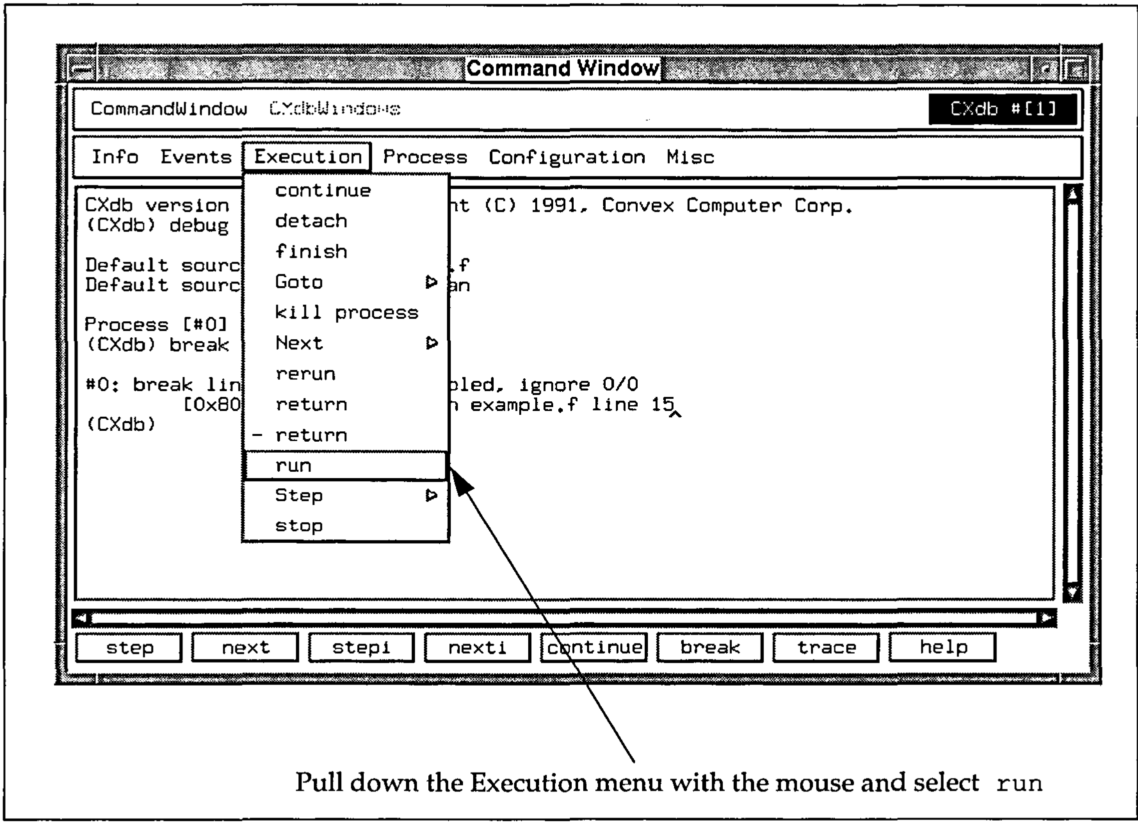
Running a program

After loading an executable file, you can run it from within CXdb by using the `run` command. The `run` command does the following:

- Creates a process image from the executable file
- Associates the process image with the current process object and the related compiler-generated data files
- Starts execution of the process
- Brings up the process interface window for program input and output

Figure 8 illustrates how to execute the `run` command from the pull-down menus in the command window.

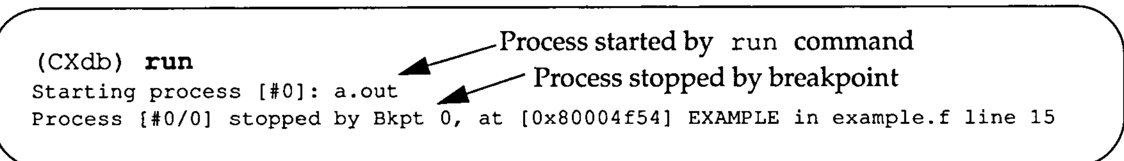
Figure 8
Running the process



Pull down the Execution menu with the mouse and select `run`

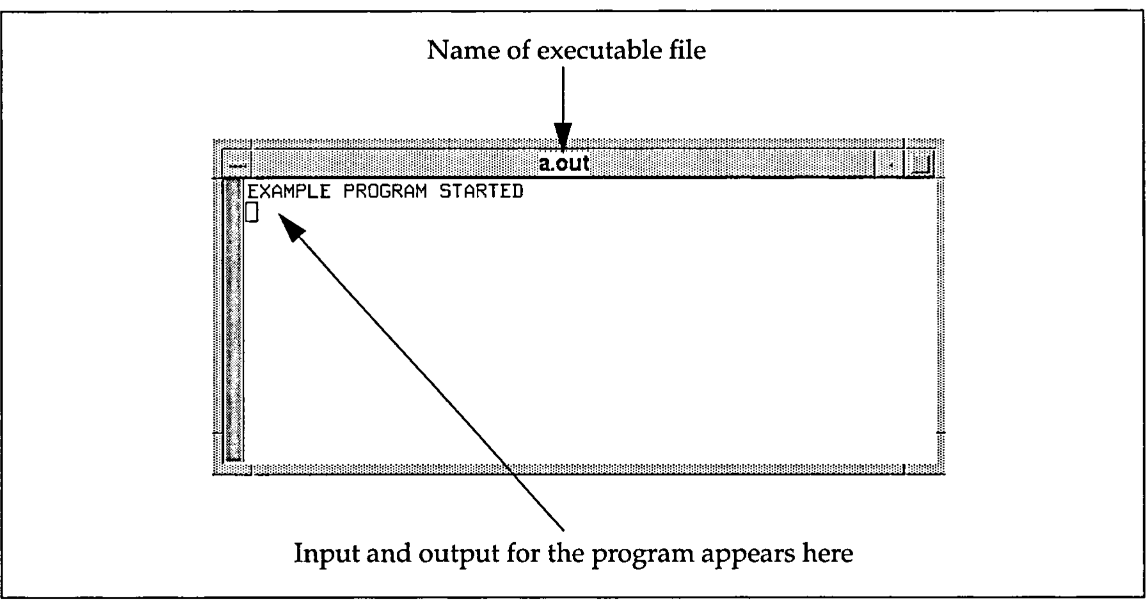
In the command window, CXdb issues a response to the `run` command to indicate that the process has started. However, the breakpoint set at line 15 in Figure 6 stops execution of the program almost immediately, and CXdb issues another message to indicate this. Figure 9 shows both messages.

Figure 9
Results of the `run` command



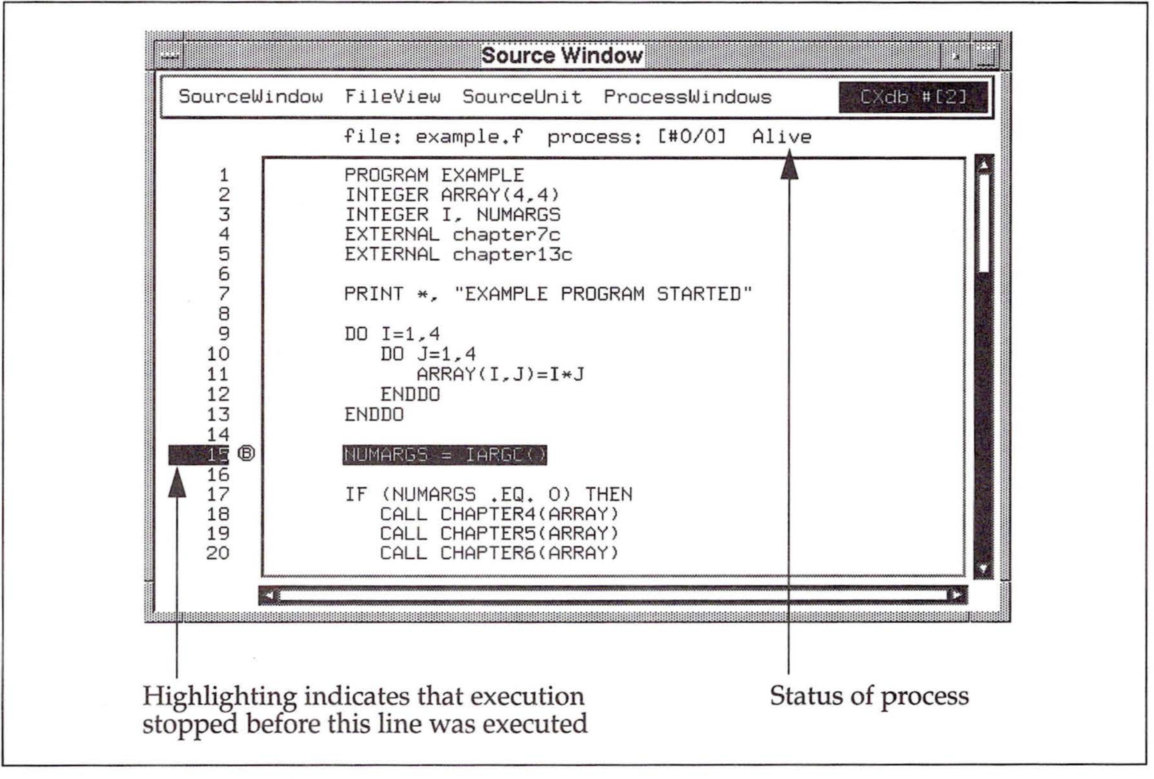
The `run` command automatically brings up the process interface window, which allows interactive input and output with your program. This window keeps program I/O separate from CXdb I/O. Figure 10 illustrates the process interface window.

Figure 10
Process interface window in CXwindows



When execution stops, the source window updates to reflect the current state of the process. Highlighting also appears in the source window to indicate where execution has stopped, as illustrated in Figure 11.

Figure 11
Highlighting in the source window



The highlighting in Figure 11 indicates that execution has stopped at line 15 of the source code. The highlighted line number means that execution has stopped at the beginning of line 15, so no part of this line has been executed yet. When process execution resumes, it will start at line 15.

The highlighted source code represents the portion of code that is currently active. The amount of source code that is highlighted depends on the granularity (size) of the source unit that is active. For more information about source units and their effect on highlighting, refer to the section, "Effect of default granularity on highlighting," in Chapter 5.

Printing data

While the process is stopped, you can display the contents of the program variables. The `print` command is one way to display variables.

Figure 12 shows the use of the `print` command to display the variable `J` from the example program. The response to the command indicates that `J` is a 4-byte integer with a current value of 5.

Figure 12
Printing a single variable

```
(CXdb) print J
(INTEGER*4) 5
```

A feature known as *completion* makes it easy to enter long variable names or command names. With completion, you can type only the first few letters of a command or variable name, then press TAB. CXdb fills in the rest of the name if it recognizes that name as unique. Figure 13 illustrates the use of completion to print the elements of an array named `ARRAY`.

Figure 13
Using completion on a command and a variable name

Command window→

```
(CXdb) pr [TAB] AR [TAB] [RETURN]
INTEGER*4(1:4, 1:4)
(1..4,1) :  1 2 3 4
(1..4,2) :  2 4 6 8
(1..4,3) :  3 6 9 12
(1..4,4) :  4 8 12 16
```

Killing the process

To terminate the program, use the `kill process` command. This command asks for confirmation before killing the process, as shown in Figure 14.

Figure 14
Killing the process

```
(CXdb) kill process
Kill process [#0]? y  <------- Confirmation of kill
Terminated execution of Process [#0]
```

When you kill the process, the process interface window disappears. However, killing the process does not delete the process object. The process object stores all the information about the process, including breakpoint settings. Therefore, the breakpoints remain set even though the process is killed.

Passing arguments to the process

Many programs are written to accept arguments passed to them from the shell. You can execute these programs in CXdb by including the arguments as a quoted string with the run command. Figure 15 illustrates how to do this.

Figure 15
Passing an argument with the `run` command

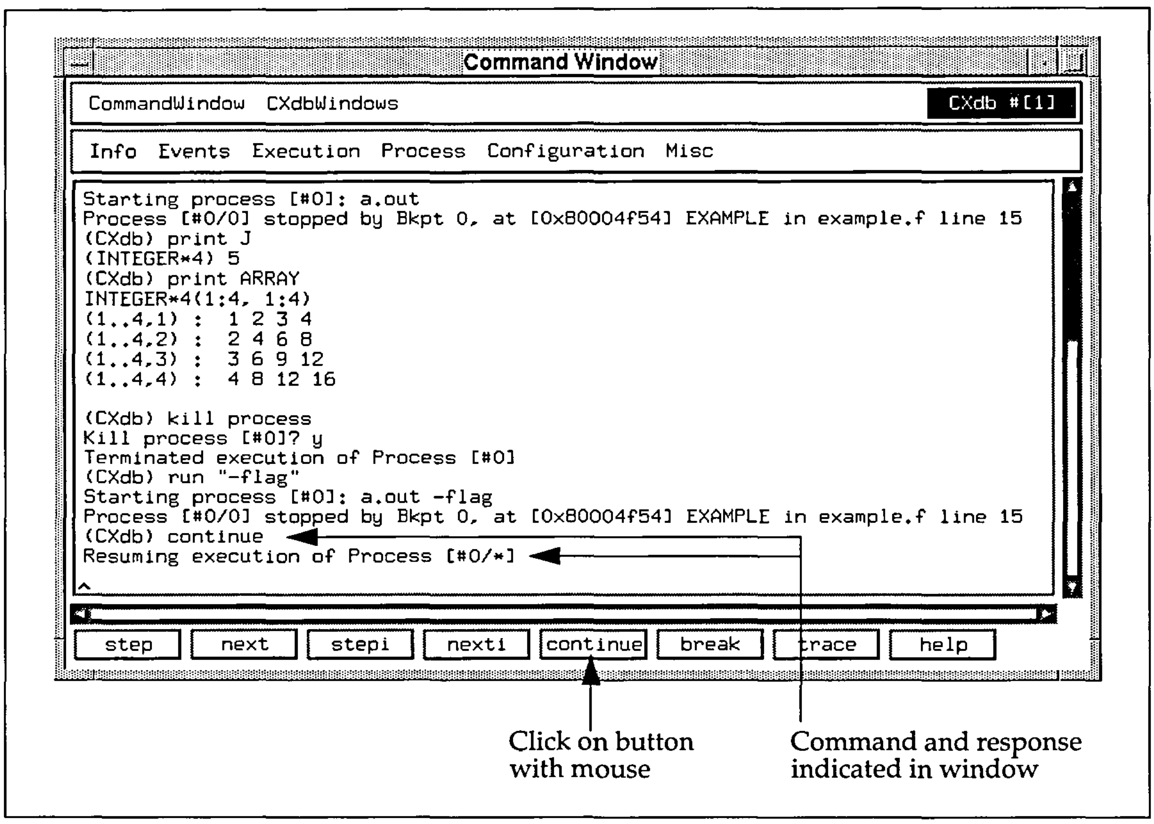
```
(CXdb) run "-flag"  <------- Argument list
 Starting process [#0]: a.out -flag          Process stopped by breakpoint
Process [#0/0] stopped by Bkpt 0, at [0x80004f54] EXAMPLE in example.f line 15
```

The `run` command in Figure 15 passes the argument `-flag` to the process. This command also opens the process interface window again.

Because the breakpoint settings are stored in the process object, the breakpoint set in Figure 6 is still enabled. This breakpoint stops execution of the program, as indicated by the output message in Figure 15.

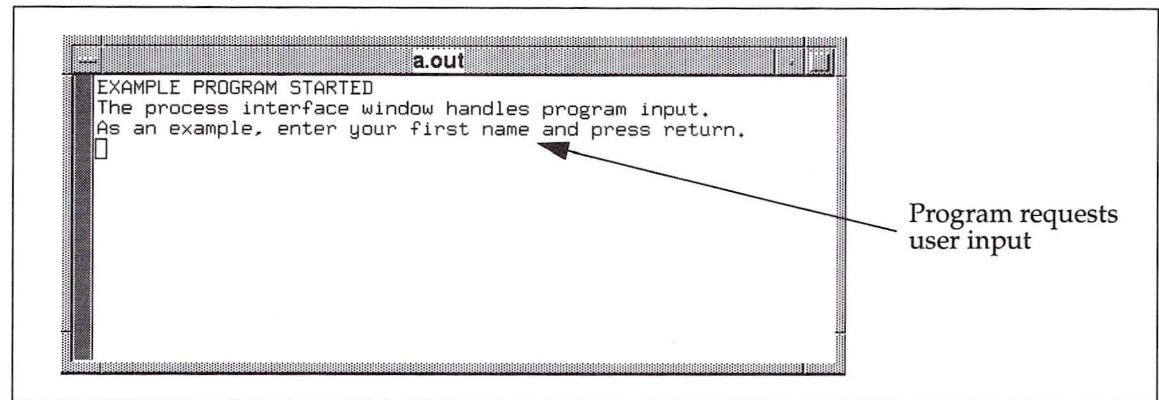
To resume execution of the program, use the `continue` command. You can either type the `continue` command in the command window, use the mouse to select the continue option from the Execution menu in the command window, or use the mouse to click on the continue button at the bottom of the command window. Figure 16 shows how to execute the `continue` command with the command button.

Figure 16
Continuing process execution



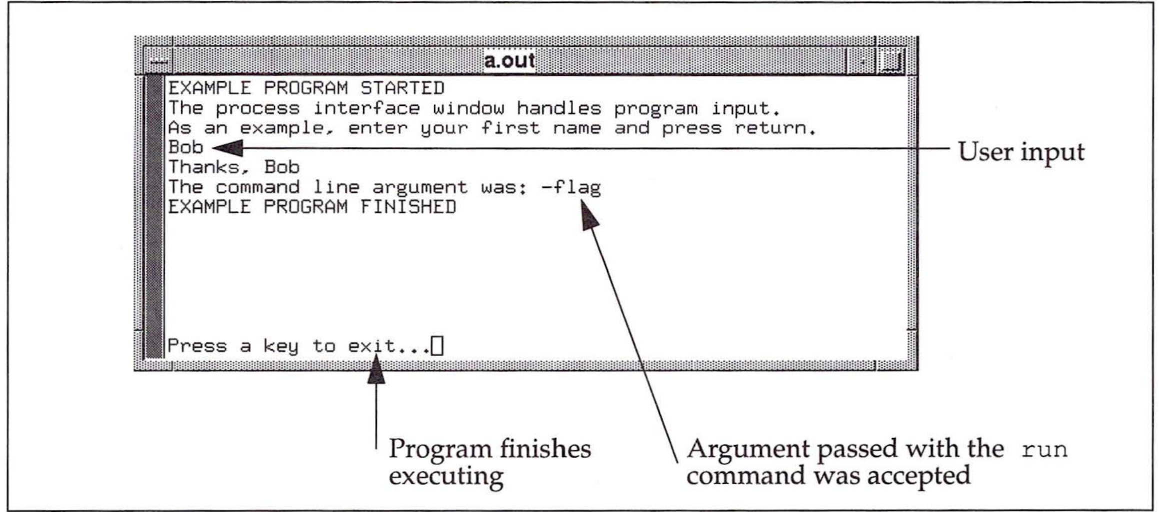
When you continue execution of this example program, the mouse pointer turns into a clock, and the `(CXdb)` prompt does not reappear in the command window. This is because the program is waiting for user input in the process interface window, as shown in Figure 17.

Figure 17
Program is waiting for input in the process interface window



When you enter the input requested by the program in the process interface window, the program executes to completion. The output from the program indicates that the argument `-flag,` passed to it by the `run` command in Figure 15, was accepted. Figure 18 shows the result of providing input to the program.

Figure 18
Providing input through the process interface window



When the program is finished executing, a message appears in the command window to indicate that the process has exited normally. You can then close the process interface window by bringing that window into focus with the mouse and pressing any key on the keyboard.

You can run the program again with the same arguments by using the `rerun` command. This command also brings up the process interface window again. Figure 19 illustrates this command.

Figure 19
Rerunning a process with the same arguments

```
(CXdb) rerun
Starting process [#0]: a.out -flag
Process [#0/0] stopped by Bkpt 0, at [0x80004f54] EXAMPLE in example.f line 15
```

Process is rerun with previous arguments
Process stopped by breakpoint

Running a command in background

Commands that cause process execution can be placed in background relative to CXdb. This means that the `(CXdb)` prompt reappears, and you can enter other CXdb commands while the process is executing. When the backgrounded command completes, CXdb issues a message in the command window.

To execute a command in background, include the ampersand (`&`) at the end of the process execution command. Figure 20 illustrates background execution with the `continue` command.

Figure 20
Running a command in background

```
(CXdb) continue &
Command [#159] backgrounded

Resuming execution of Process [#0/*]
(CXdb)
```

Specifies background execution
Prompt reappears immediately

As Figure 20 indicates, the `(CXdb)` prompt reappears instead of a clock, even though the process is waiting for input through the process interface window.

To check the status of a process, use the `info process` command, as illustrated in Figure 21.

Figure 21
Obtaining information about a process

```
(CXdb) info process
status of process [#0]:

      executable: a.out
       arguments: -flag
fixed scheduling: off
          pshell: csh
    image status: created pid 14796, state = running
     working dir: /usr/lib/cxdb/examples
    default step: statement
default language: Fortran
         threads: 1
  current thread: 0


source file search path:
        .
```

Indicates that process is running

To check the status of the debugging session, use the `info cxdb` command, as illustrated in Figure 22.

Figure 22
Obtaining information about the debugging session

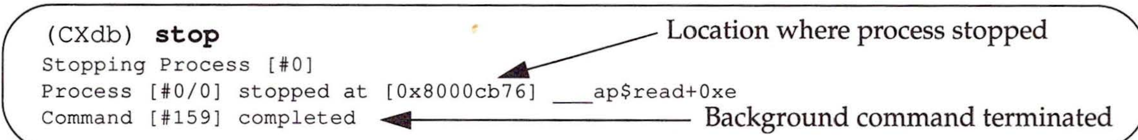
```
(CXdb) info cxdb
Current CXdb state:

     ENVIRONMENT:
              pid: 11013
              cwd: /usr/lib/cxdb/examples
   command modes: echo off, logging off, noclobber off
           cmdout: Window #1
           cmderr: Window #1
           cmdlog:
         evalopts: fpmode = native, iprecision = 4, rprecision = 4
            shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
        step size: statement
    process shell: csh
           fpmode: native
      memory size: (none)
   memory formats: byte=(none), halfword=(none), word=(none)
                    longword=(none), quadword=(none)
      search path:
         .


        PROCESSES:
     process [#0]: created pid 14796, state = running, executable = a.out
                    shell = csh

ACTIVE COMMANDS:

command [#159] - continue &
```

Indicates that the `continue` command is active in background mode

Stopping a process

If you have placed a CXdb command in background, you can use the `stop` command to halt execution of your process. The `stop` command terminates the backgrounded command that started process execution. Therefore, it also stops the process. Figure 23 illustrates the `stop` command.

Figure 23
Stopping a process



As Figure 23 shows, CXdb issues a message to indicate where the process has stopped. CXdb also issues another message to indicate that the command in background (the `continue` command from Figure 20) has completed.

If the process execution command is not running in background, you can abort the command by typing **CTRL-c** in the command window. This also stops process execution.

Getting help online

CXdb has an extensive online help system that contains the same information as the *CONVEX CXdb Reference* manual. The help topics cover 4 categories:

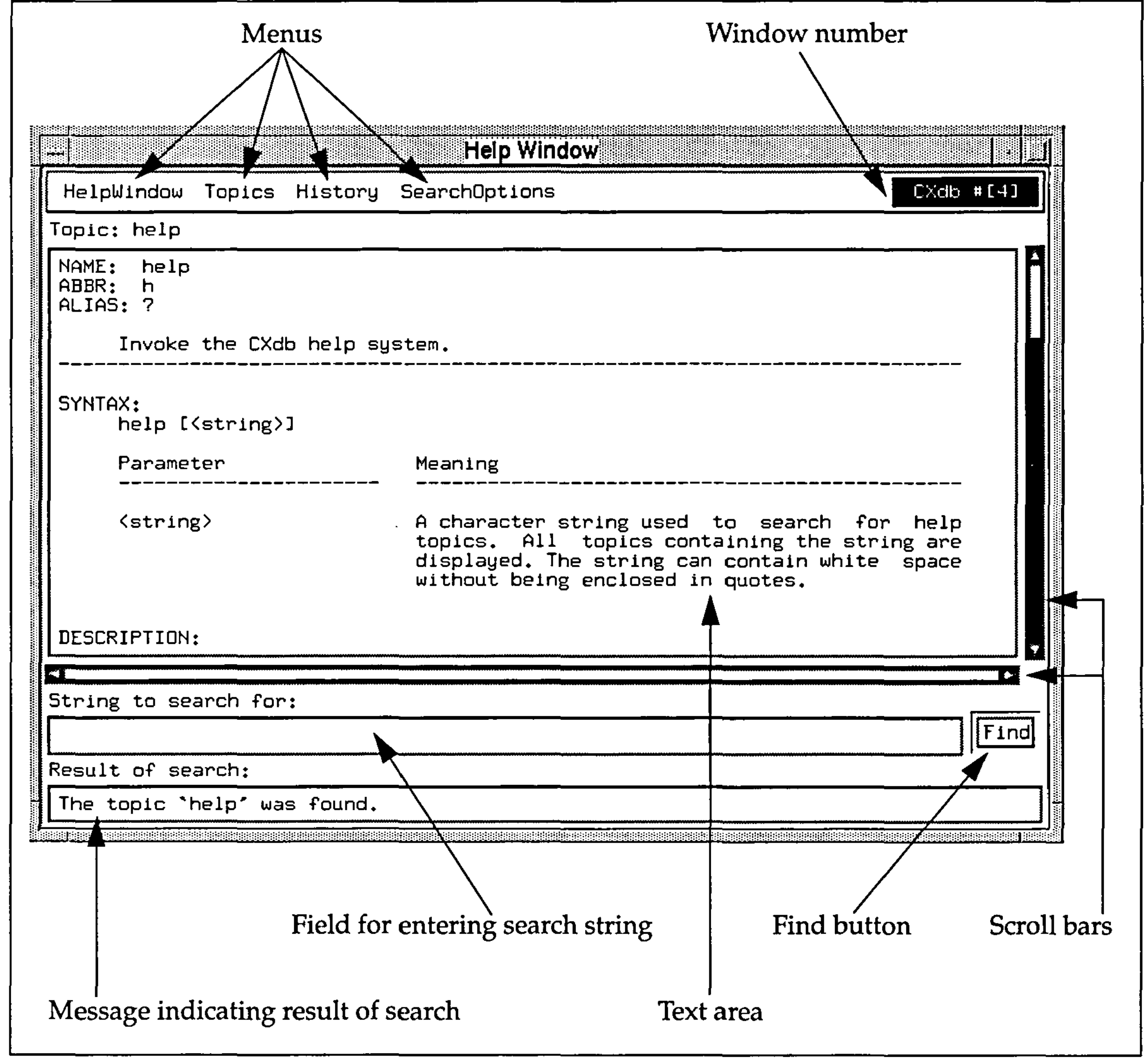
- **Commands**—Complete description, syntax, and examples of each CXdb command.
- **Concepts**—Explanations of the major concepts that tie related commands together.
- **CXdb messages**—Text and explanations for messages displayed by CXdb.
- **Parameters**—Description, syntax, and examples of parameters used with the CXdb commands.

The help system has its own window for accessing and displaying help topics. You can call up the help window from the command window by using one of the following methods:

- Entering the `help` command at the `(CXdb)` prompt
- Selecting the help option from the Misc menu
- Clicking on the help button with the mouse

Figure 24 shows the help window.

Figure 24
Help window in CXwindows



The help window includes:

- **Menus**—These pull-down menus allow you to do the following by using the mouse:
  - HelpWindow accesses the online tutorials or closes the help window.
  - Topics lists the available help topics in the categories of commands, concepts, CXdb messages, and parameters.
  - History lists the topics selected during the current help session.
  - SearchOptions allows you to specify which categories to search for a topic, and whether or not to include the help text in the search.
- **Window number**—This is a unique number that identifies each window in CXdb. Certain CXdb commands allow you to specify a window number as a parameter.
- **Scroll bars**—These bars and arrows enable you to scroll the window vertically and horizontally by using the mouse.
- **String to search for**—This is the field where you enter a character string that represents the name of the topic you want to view. Click on the field with the left mouse button, then type the character string.
- **Find button**—This button starts the search for the character string you entered. Activate this button by clicking on it with the mouse or by pressing RETURN when the help window is active.
- **Result of search**—This field displays a message indicating the result of the string search.
- **Text area**—This part of the window displays either the text of a help page or a list of topics found by the search.

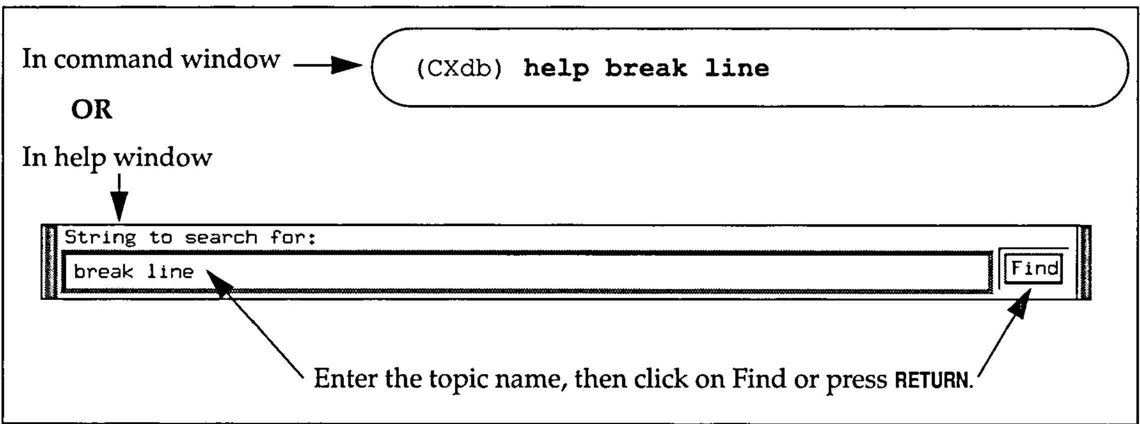
Requesting help on a topic

You can get help on a specific topic by requesting it in one of 2 ways:

- In the command window, specify the name of the topic with the `help` command.
- In the help window, enter the name of the topic in the `String to search for:` field.

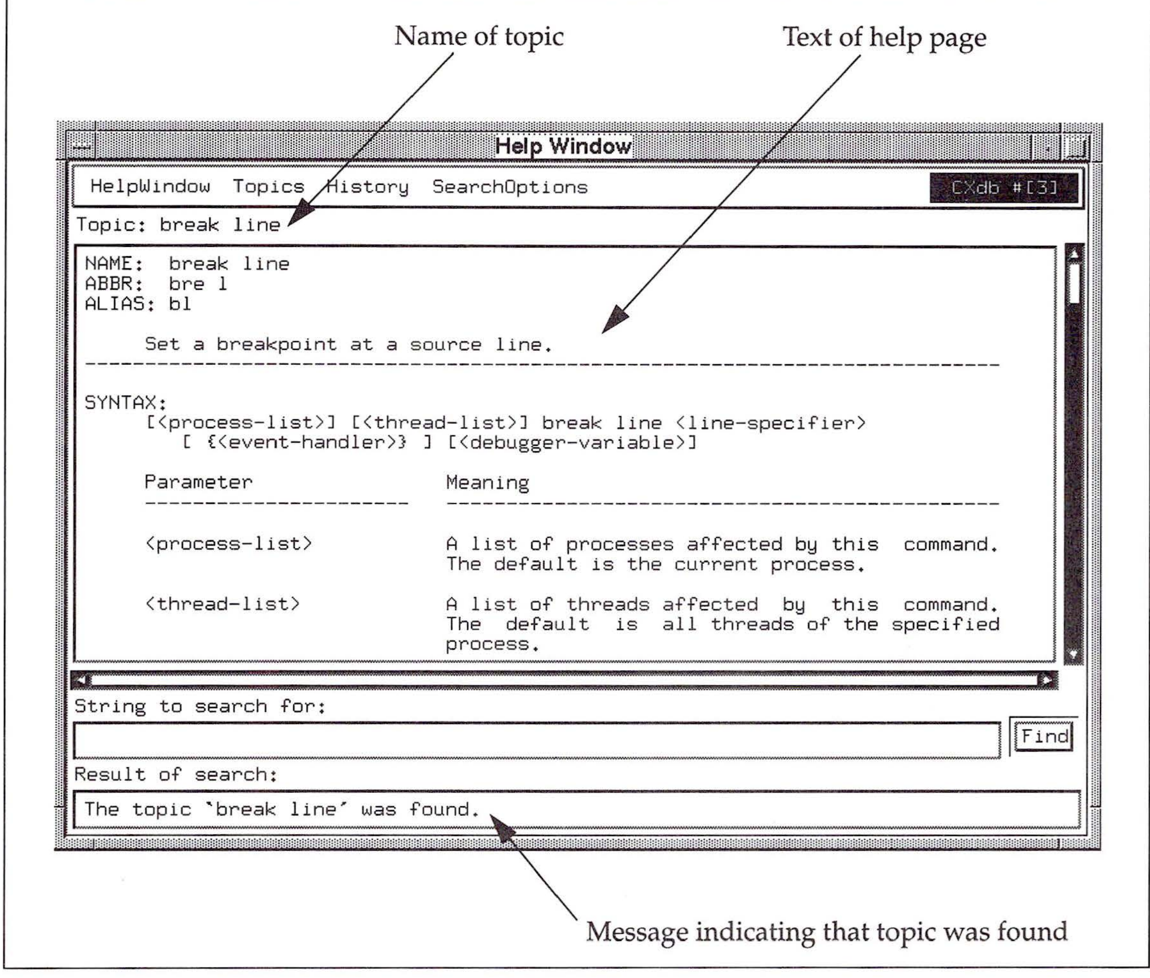
Figure 25 illustrates both methods.

Figure 25
Requesting help on a specific topic



With either method, the help page for the specified topic displays in the text area of the help window, as shown in Figure 26. You can use the scroll bars in the help window to view the rest of the text on a help page.

Figure 26
Sample help page

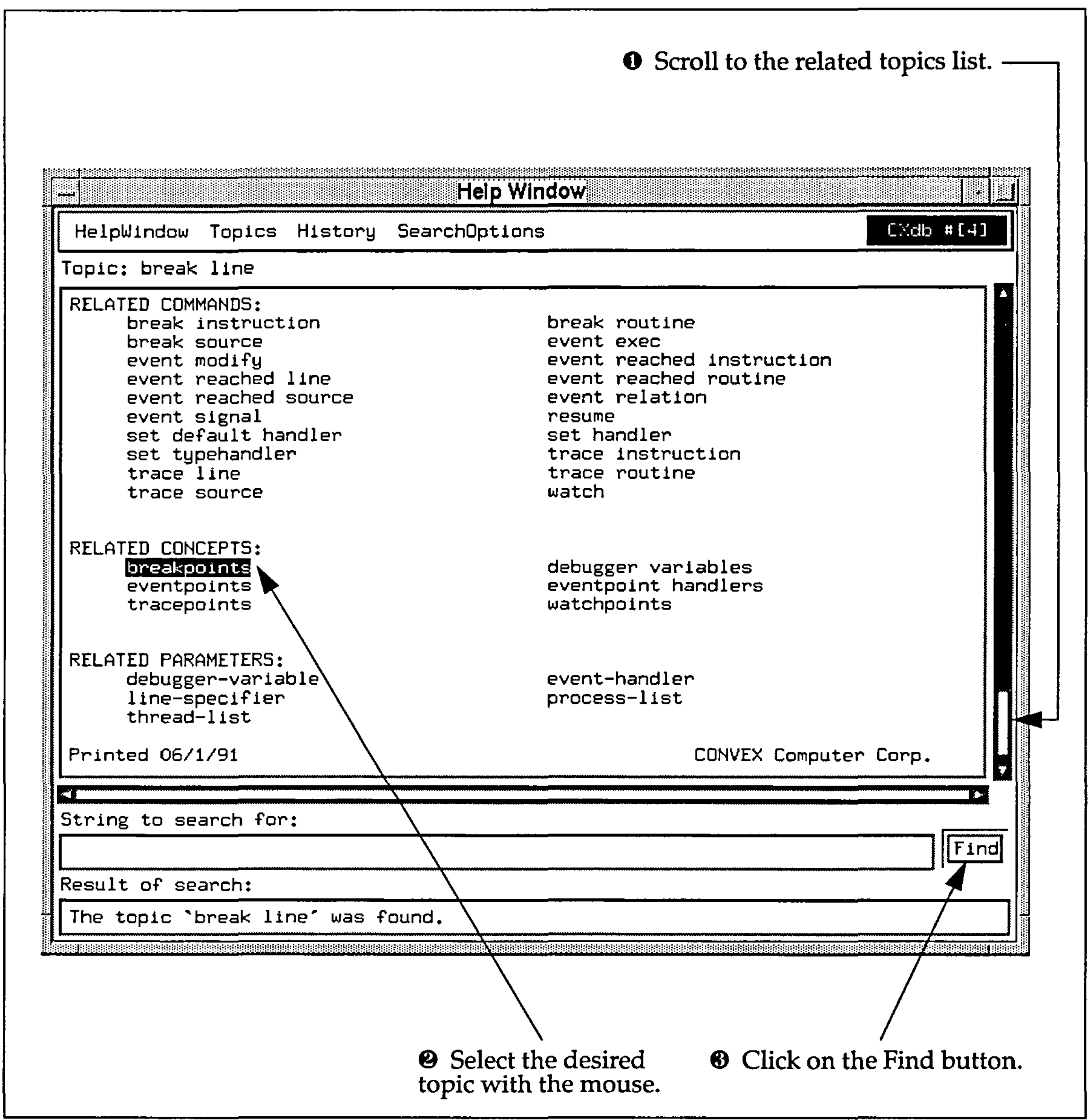


Getting help on related topics

At the bottom of each help page, there are lists of commands, concepts, and parameters that are related to the current help topic. To access these lists, scroll to the bottom of the help page.

You can select one of the related topics with the mouse by clicking on the topic name with the left mouse button and dragging the mouse until the topic name is highlighted. Once the topic name is highlighted, click on the Find button to view the help page for that topic. Figure 27 illustrates this procedure.

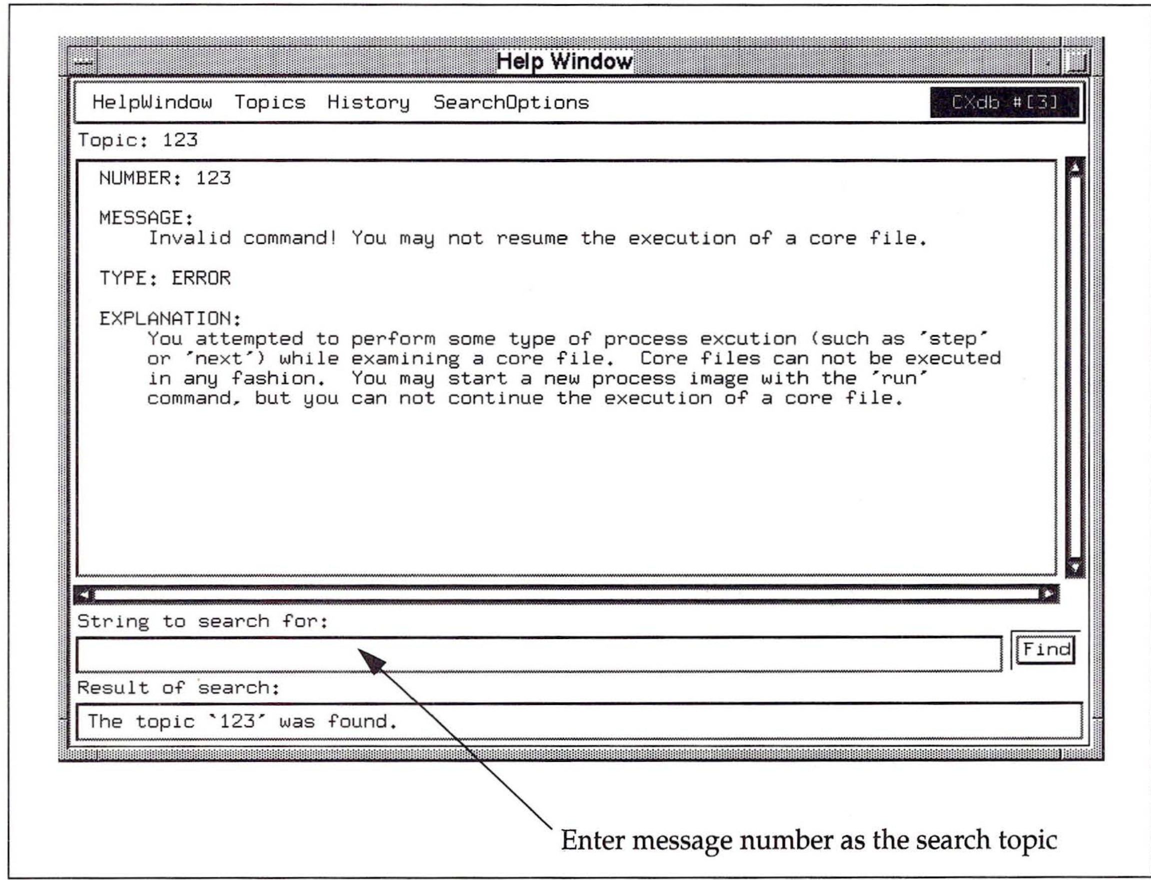
Figure 27
Selecting a related topic



Requesting help on a CXdb message

In response to commands you enter, CXdb can issue informational messages and error messages in the command window. For efficiency, the messages are generally brief, but you can request a more detailed explanation of a message through the online help system. To request help on a CXdb message, simply enter the message number as the topic name, as illustrated in Figure 28.

Figure 28
Requesting help on a CXdb message



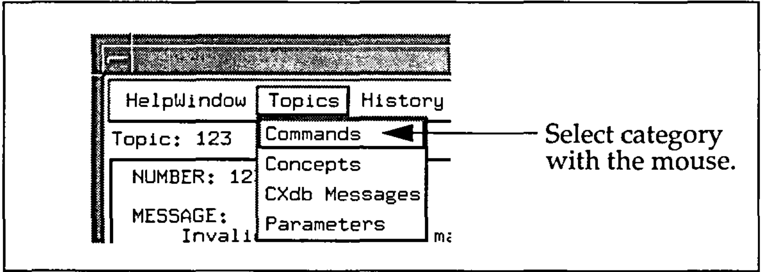
Using topics lists

The Topics menu at the top of the help window lets you display a list of all topics in one of the following categories:

- Commands
- Concepts
- CXdb Messages
- Parameters

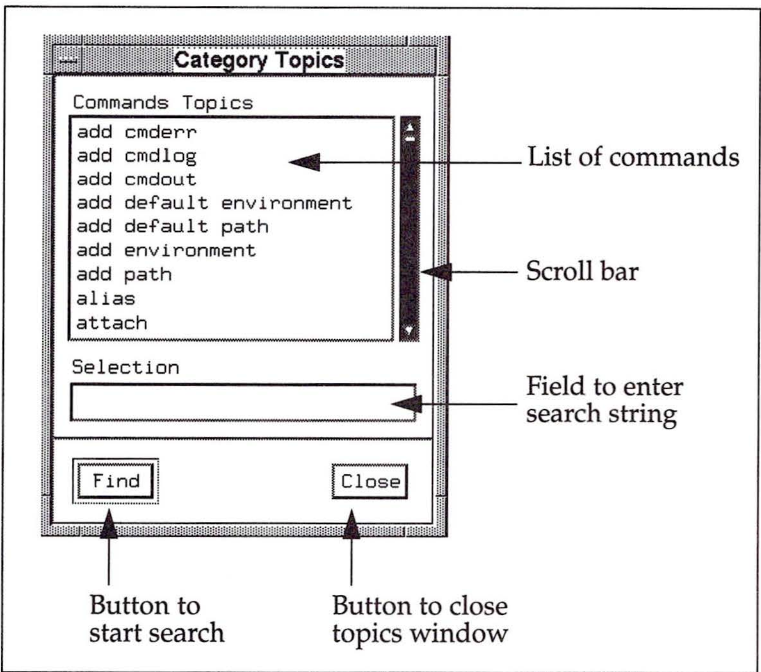
To display one of the lists, pull down the Topics menu with the mouse and select one of the 4 categories, as shown in Figure 29.

Figure 29
Topics menu



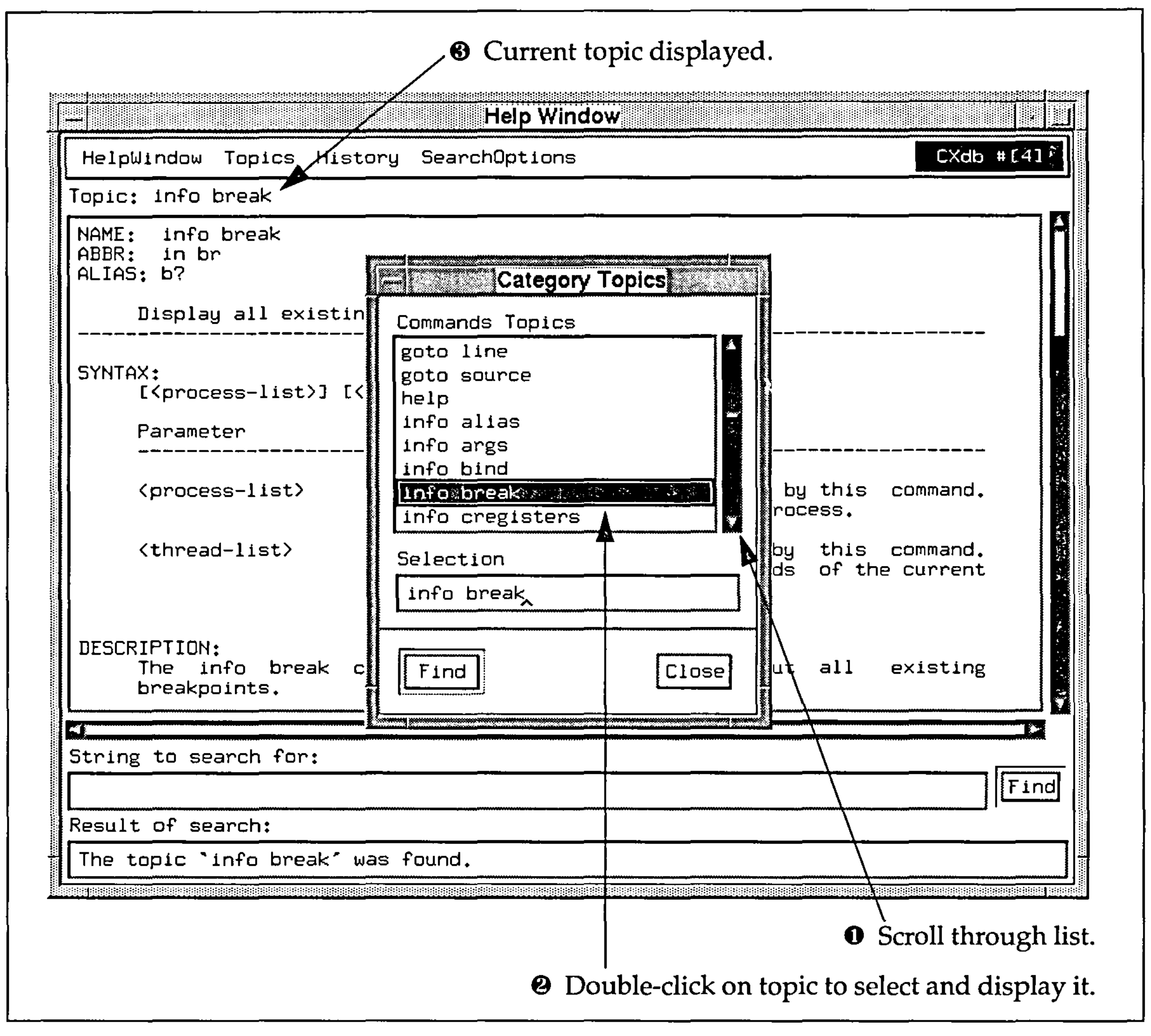
When you select a category from the Topics menu, this brings up a topics window, as shown in Figure 30.

Figure 30
List of topics under Commands



You can select one of the topics from the list by clicking on it with the mouse. If you double-click on the topic, it displays immediately in the help window, as shown in Figure 31.

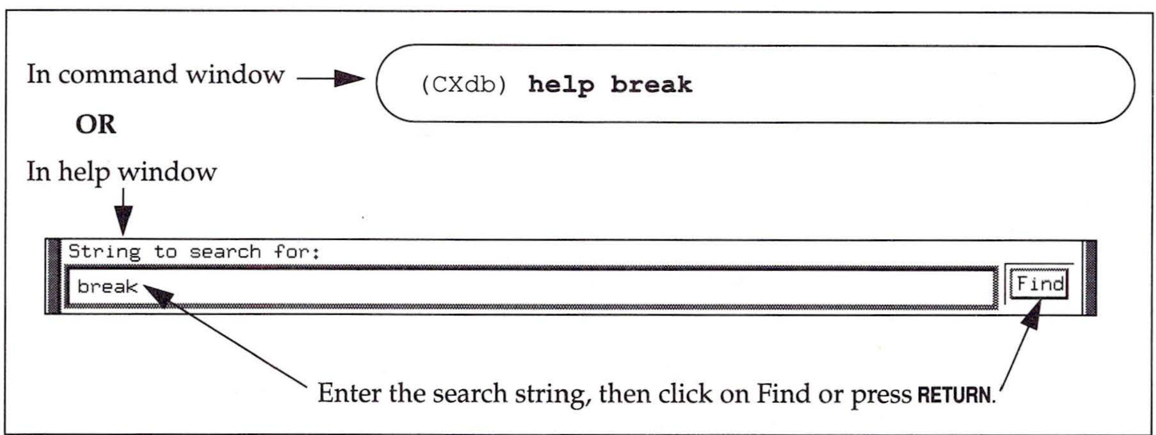
Figure 31
Selecting a topic from the list



Searching for a topic

Even if you do not know the exact name of a topic, you can search for it by entering a character string that is part of the name. You can enter the string either as a parameter of the `help` command in the command window or as an entry in the `String to search for:` field in the help window. Figure 32 illustrates both methods.

Figure 32
Searching for a topic by using a search string



If there is no exact match between the search string and a topic name, the help window displays a list of topics whose names contain the search string. Figure 33 shows this result.

Figure 33
Result of a topic search

Help Window

HelpWindow Topics History SearchOptions CXdb #[2]

The string `break' is a substring of these help topics:

Category: Commands

break instruction
break line
break routine
break source
info break

Category: Concepts

breakpoints

String to search for:

Find

Result of search:

`break' matches 6 topics.

Message indicating result of search

List of topics found

Using search options

The SearchOptions menu at the top of the help window lets you control how searches are conducted in the help system. The options in this menu are:

- **Categories to search**—This option brings up a window that lets you specify which help categories are searched. Figure 34 shows this window. The categories are Commands, Concepts, CXdb Messages, and Parameters.
- **Include text of help pages in search**—This is a toggle button that determines whether or not the text of the help pages will be searched. Figure 35 shows how to toggle this button with the mouse. The text is searched only if the search string is not found in any of the topic names.

Figure 34
Specifying search categories

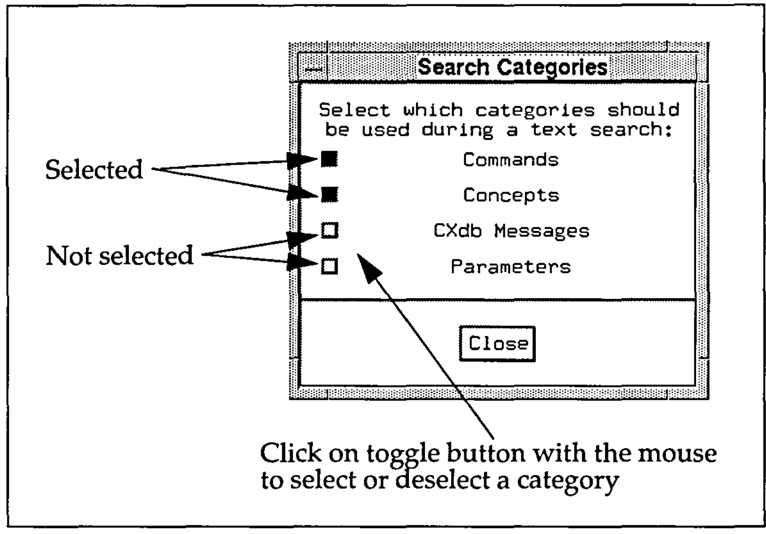
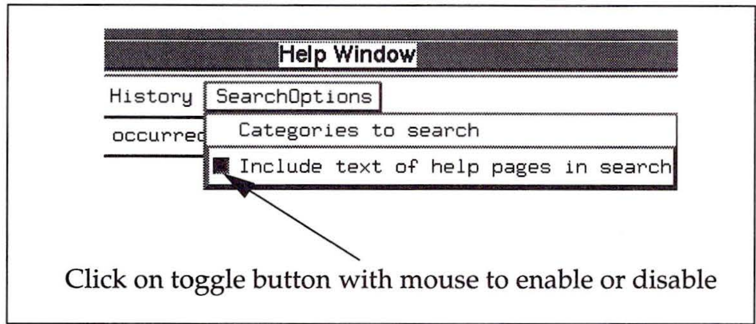
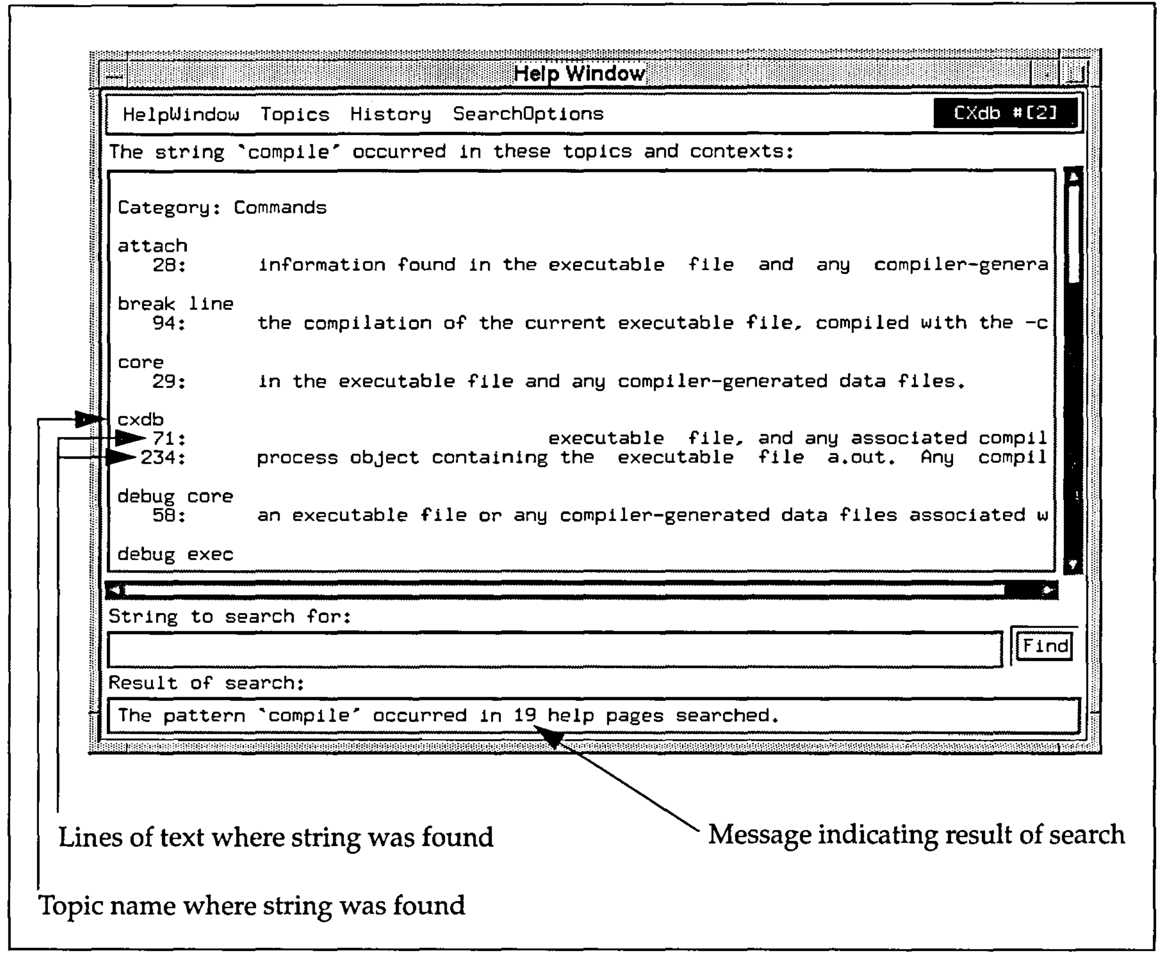


Figure 35
Including the text of the help pages in a search



If you choose the "Include text of help pages in the search" option, then you can search for character strings that are not part of a topic name. For example, the word "compile" is not part of a topic name, but it appears many times in the text of the help pages. Figure 36 shows the results of the text search as they appear in the help window.

Figure 36
Results of text search

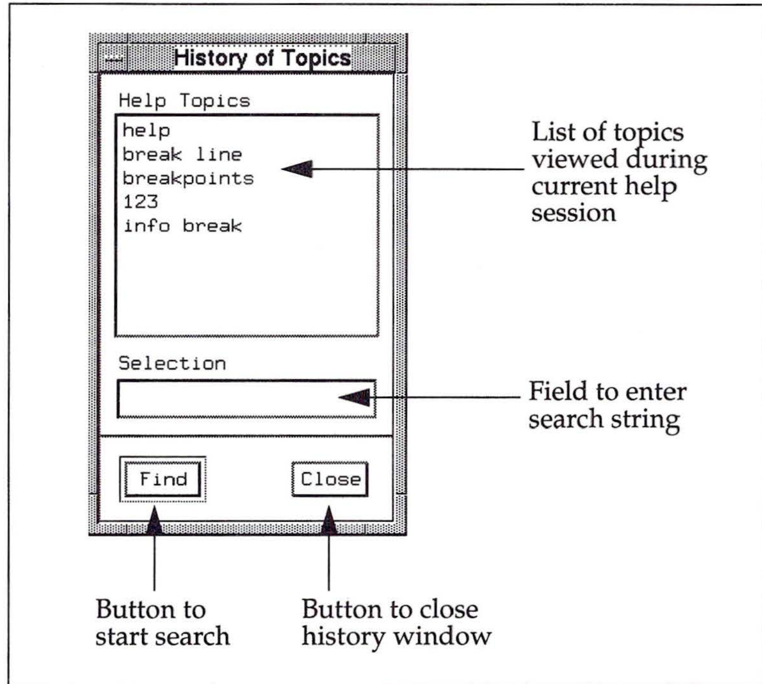


Using the topics history

The topics history lists all of the topics you have viewed during the current help session. Once you close the help window, the history list is deleted.

To view the history list, select the Show topics history option from the History menu at the top of the help window. This brings up the topics history window, as illustrated in Figure 37.

Figure 37
Topics history window

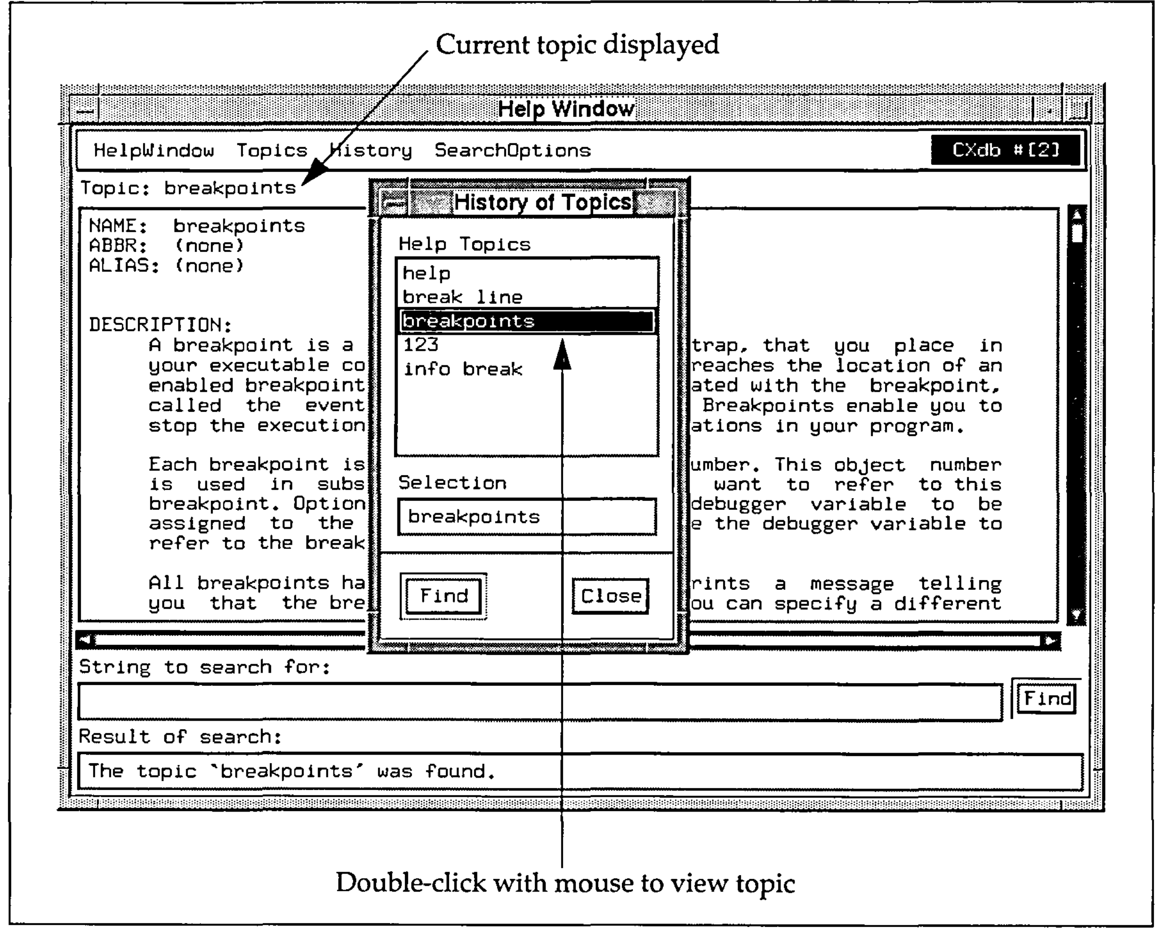


You can select one of the topics from the history list by double-clicking on it with the mouse. You can also search for a topic that is not in the history list by entering a search string in the field labeled Selection.

Figure 38 illustrates selecting a topic from the history list by double-clicking with the mouse.

Closing the history window does not clear the topics history list. However, closing the help window does clear the history list.

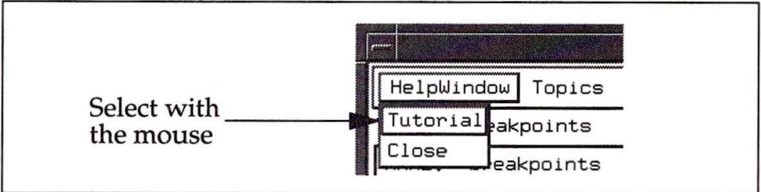
Figure 38
Selecting a topic from the history list



Accessing the *CXdb Online Guide*

The *CXdb Online Guide* is an extensive online tutorial that introduces many of the CXdb commands and concepts. You can access the guide by selecting the Tutorial option from the HelpWindow menu, as shown in Figure 39.

Figure 39
Accessing the *CXdb Online Guide*



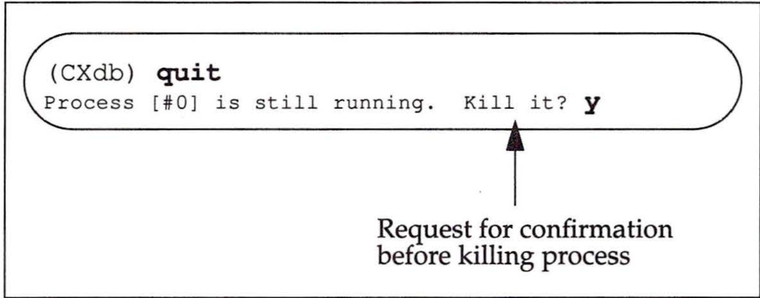
Quitting CXdb

To exit CXdb, use the `quit` command. This command does the following:

- Kills all existing processes and process objects, with your confirmation
- Kills any active CXdb commands
- Closes any files opened by CXdb
- Closes all CXdb windows

Figure 40 illustrates the `quit` command.

Figure 40
Quitting CXdb



Using CXdb with Maryland Windows

3

This chapter describes the main CXdb windows available through the Maryland Windows interface. This chapter also gives an overview of basic CXdb commands used to load, run, and quit your program. The commands discussed in this chapter are:

- `break line`
- `continue`
- `cxdb`
- `debug exec`
- `help`
- `info cxdb`
- `info process`
- `kill process`
- `print`
- `quit`
- `rerun`
- `run`
- `stop`

Maryland Windows

The Maryland Windows interface is a set of library routines that create and manage windows for ASCII terminals. The Maryland Windows interface was developed at the University of Maryland Computer Science Department, and it is included with ConvexOS V9.0 (and later releases).

CXdb uses these library routines to provided multiple windows in CXdb. Each window gives you a unique set of information about your program that you can use while debugging.

Invoking CXdb

The `cxdb` command, issued from the shell, invokes CXdb. Many command line options are available with the `cxdb` command. For a complete description of these options, refer to either the *CONVEX CXdb Reference* manual or the man page for cxdb(1).

To execute the examples shown in this manual, you must first change to the directory where the example programs are located, and then enter the `cxdb` command to invoke CXdb, as shown in Figure 41.

Figure 41
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb
```

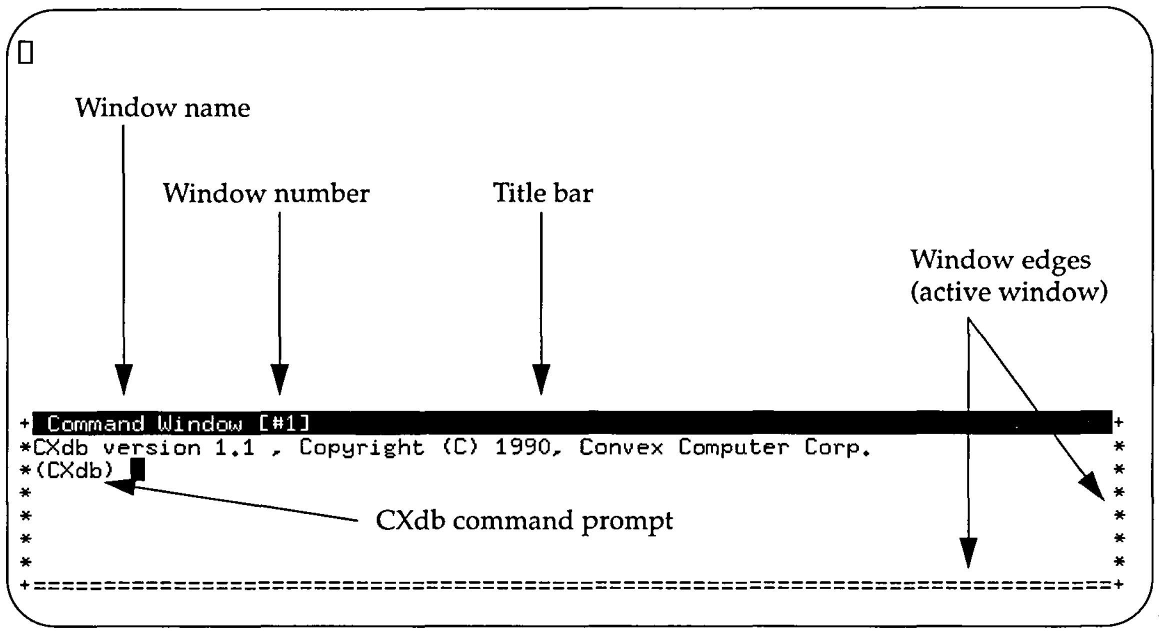
The `cd` command in Figure 41 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb.

When you invoke CXdb, it automatically brings up the command window. The command window is the primary way for you to communicate with CXdb. It enables you to:

- Enter CXdb commands
- Receive output from CXdb commands
- Receive error messages or status information about CXdb commands
- Review previous commands and retrieve them from the command history
- Edit and repeat commands

Figure 42 illustrates the CXdb command window in Maryland Windows. If the environment variable `DISPLAY` is set, CXdb defaults to the CXwindows interface. Refer to Chapter 2 for a description of the CXwindows interface for CXdb.

Figure 42
Command Window in Maryland Windows



The command window includes:

- **Title bar**—The title bar indicates the window name and the window number.
- **Window number**—This is a unique number that identifies each window in CXdb. Certain CXdb commands allow you to specify a window number as a parameter.
- **Window edges**—The edges of the window separate it from other CXdb windows. The window edges change their look depending on whether the cursor is in that window or not. When the cursor is in the window, the window is active, and its edges are made up of asterisks (*) on the sides and equal signs (=) on the bottom. If the window is not active, its edges are made up of pipe signs (|) on the sides and minus signs (–) on the bottom.
- **Command line**—This is the area of the window where you enter commands to CXdb. When the `(CXdb)` prompt is present, you can enter commands from the keyboard.

Your screen should look similar to the screen shown in Figure 42. CXdb has default settings for the size and location of the CXdb windows. The command window is initially placed in the bottom third of your screen.

You can change the size and location of the command window once you have invoked CXdb. For more information about setting the size and location of CXdb windows in Maryland Windows, refer to the reference page for the `cxdb` command in the *CONVEX CXdb Reference*.

The CXdb command window is your primary window for interacting with CXdb. You should always try to keep the command window in view so you can enter CXdb commands and see CXdb messages.

Loading your program

After invoking CXdb, you can give it the name of the executable file you want to debug. To use the full capabilities of CXdb, you must compile your program by using the `-cxdb` option with the latest release of the CONVEX FORTRAN or CONVEX C compilers. This flag tells the compilers to generate the debugging information CXdb needs.

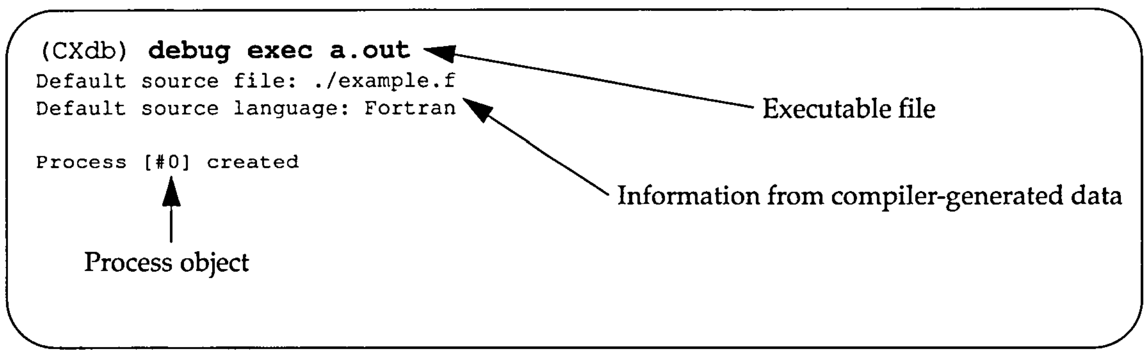
Compiler-generated debugging information is stored in a subdirectory named `.CXdb`. The compiler creates this subdirectory in the same directory as the `.o` (object) file. CXdb must be able to locate this directory and access the data files in order to perform symbolic debugging of your program. You can give CXdb access to the `.CXdb` directory either by invoking CXdb from the directory where the executable file was compiled or by using commands within CXdb to specify the path to the appropriate directory.

To begin debugging an executable file, use the `debug exec` command. This command does the following:

- Tells CXdb the name of the executable file you want to debug
- Creates a process object to store information about the executable file and processes generated from it
- Maps information from the compiler-generated data files in the `.CXdb` directory to the specified executable file
- Brings up the source window and displays the source file associated with the specified executable

Figure 43 illustrates the use of the `debug exec` command, as well as the associated response from CXdb.

Figure 43
Debugging an executable file



CXdb uses the compiler-generated files to determine the name of the source file corresponding to the main routine of the program. The source window opens and displays the main routine, as shown in Figure 44. By default, the source window is located at the top of your screen.

Figure 44
Source window in Maryland Windows

Line numbers Source file name
Window name Window number Process object Window edges

```
+ Source Window [#2]  file: example.f  process #[0]                           +
*   1        PROGRAM EXAMPLE                                                    *
*   2        INTEGER ARRAY(4,4)                                                 *
*   3        INTEGER I, NUMARGS                                                 *
*   4        EXTERNAL chapter7c                                                 *
*   5        EXTERNAL chapter13c                                                *
*   6                                                                           *
+===========================================================================+

+ Command Window [#1]                                                         +
|                                                                             |
|Default source file: ./example.f                                             |
|Default source language: Fortran                                             |
|                                                                             |
|Process [#0] created                                                         |
|(CXdb)                                                                       |
+-----------------------------------------------------------------------------+
```

The source window includes:

- **Window name**—This field identifies this as the source window.
- **Line numbers**—These numbers are added by CXdb to identify each line in the source file.
- **Window number**—This is a unique number that identifies each window in CXdb. Some of the CXdb commands allow you to specify a window number as a parameter.
- **Source file name**—This field lists the name of the source file currently displayed in the source window.
- **Process object**—This field lists the identification number of the current process object.
- **Window edges**—These edges mark the borders of the window. Because the edges are asterisks and equal signs, you can tell that the source window is currently active.

Scrolling text in a window

If the text inside a window cannot all be seen at once, you can scroll the text in the window. For example, you can scroll the source window to see your program's source code.

Table 1 shows the keystrokes involved in scrolling a window.

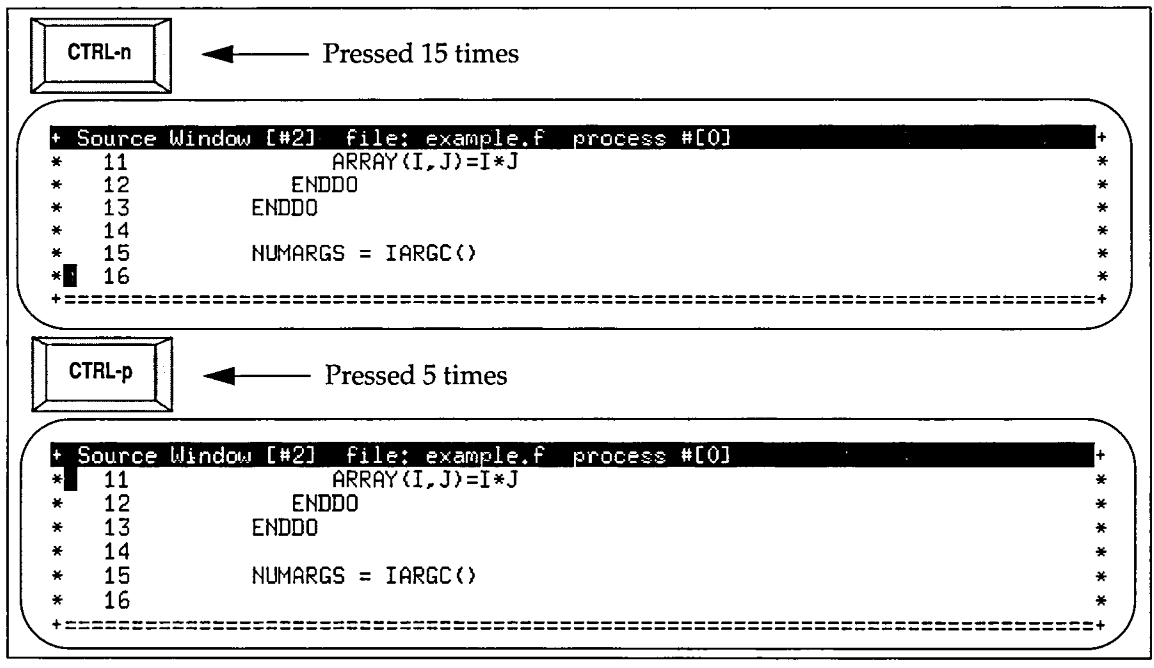
Table 1
Scrolling text in a window

Keystroke	Direction
META-B or CTRL-n	Down one line
META-A or CTRL-p	Up one line
CTRL-v	Back one screen
META-v	Forward one screen
META->	End of text
META-<	Beginning of text

In the command window, use META-A and META-B to scroll the text one line at a time. In the command window, use CTRL-n and CTRL-p to move backward and forward through your command history.

Use the CTRL-n and CTRL-p keystrokes to scroll the source code inside the source window, as shown in Figure 45. In this example, pressing CTRL-n 15 times advances the cursor to line 16 of the source code; subsequently pressing CTRL-p 5 times positions the cursor at line 11.

Figure 45
Scrolling source code in the source window



Moving between windows

When using the Maryland Windows interface, you must be able to move between the different windows of CXdb. When you move the cursor into a window, that window becomes active. A window must be active for you to be able to work with it. Thus, to enter commands, the cursor must be in the command window. If you want to scroll the source code, the cursor must be in the source window.

Each time a new window is created in CXdb, the cursor is positioned automatically in that window. To enter another command, you must return the cursor to the command window.

Table 2 shows keystrokes for moving between windows:

Table 2
Moving between windows

Keystroke	Direction
META-n	Next window
META-p	Previous window

Figure 46 and Figure 47 illustrate the use of these keystrokes to move back and forth between the windows.

Figure 46
Moving between the source window and the command window

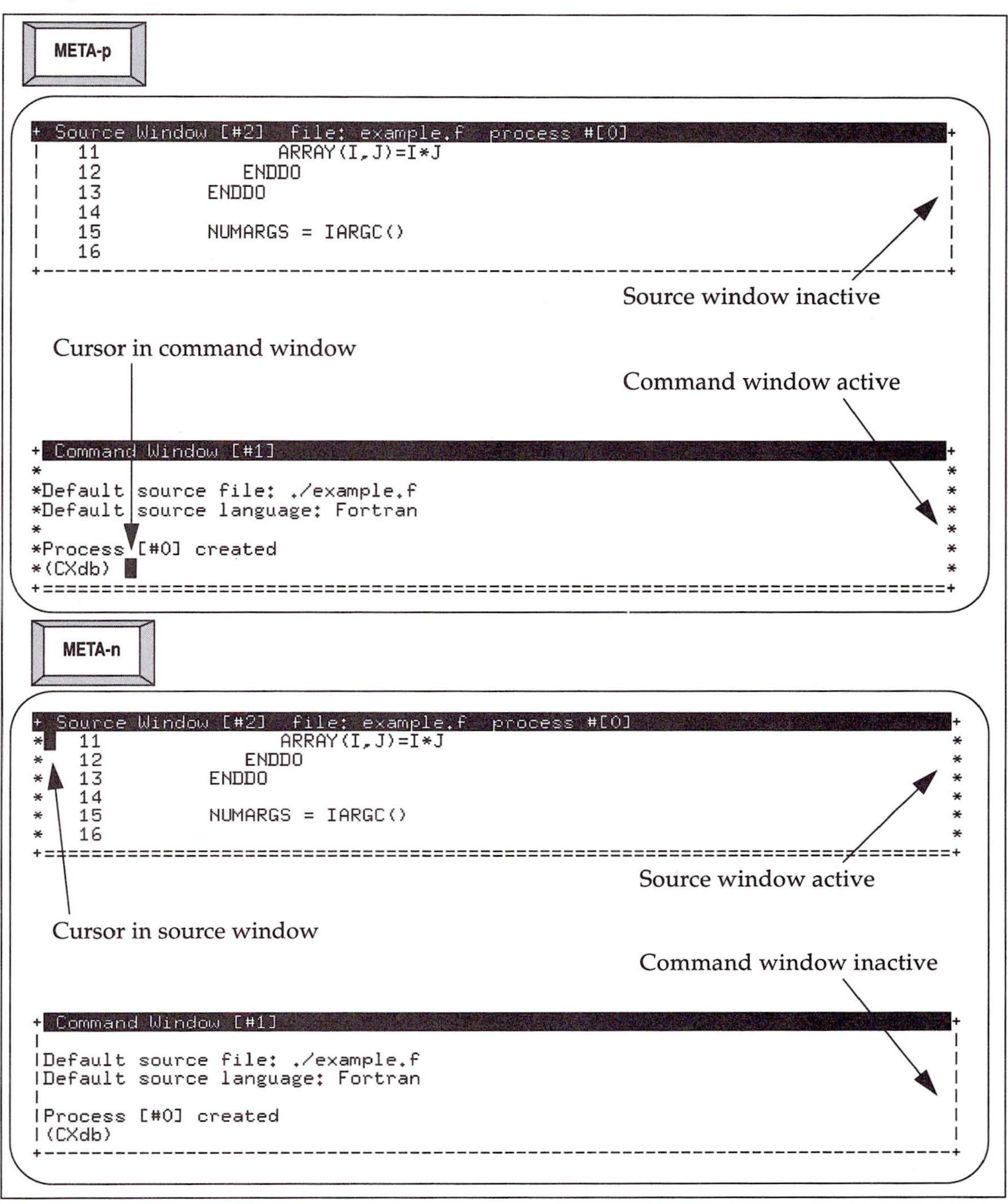
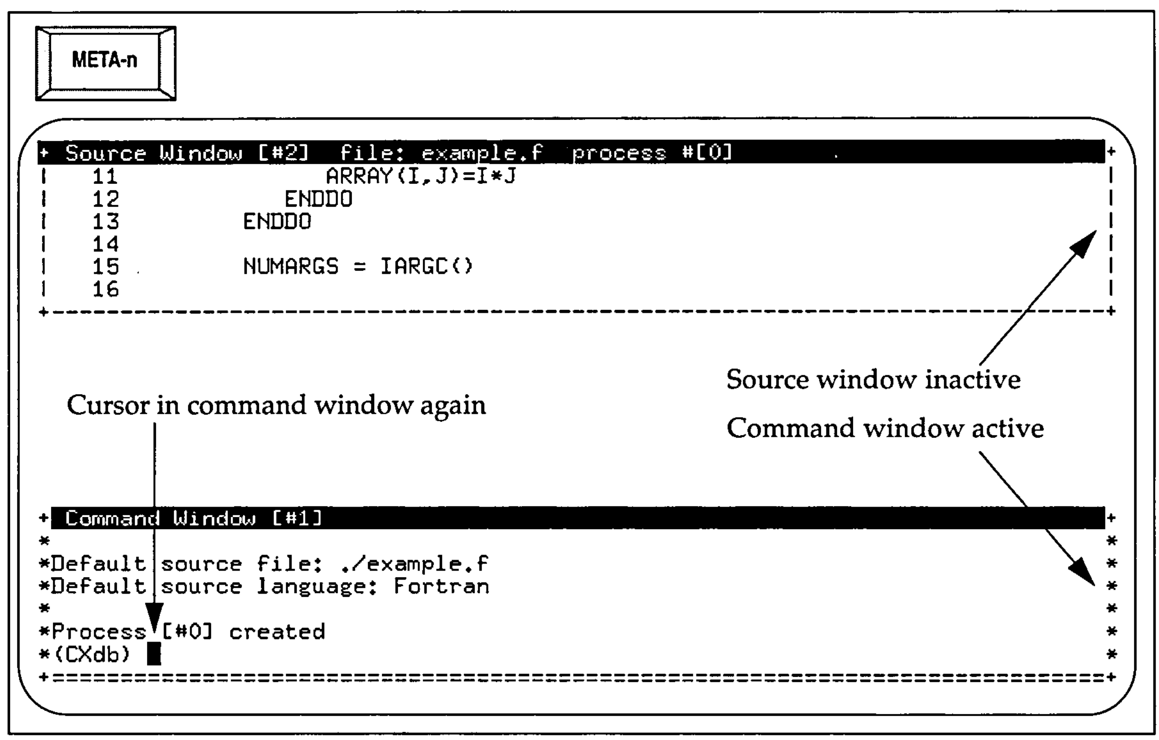


Figure 47
Moving from the source window to the command window



In CXdb, you can move from the highest-numbered window to the lowest-numbered window directly. That is, if you are at the highest-numbered window and use CTRL-n to go to the next window, you will go to window 1, as shown in Figure 46.

Setting a breakpoint

Before running a program from within a debugger, it is helpful to set a breakpoint that stops execution of the program before it reaches a problem area of the code. Figure 48 illustrates how to set a simple breakpoint at a line number.

Figure 48
Setting a breakpoint at a line number

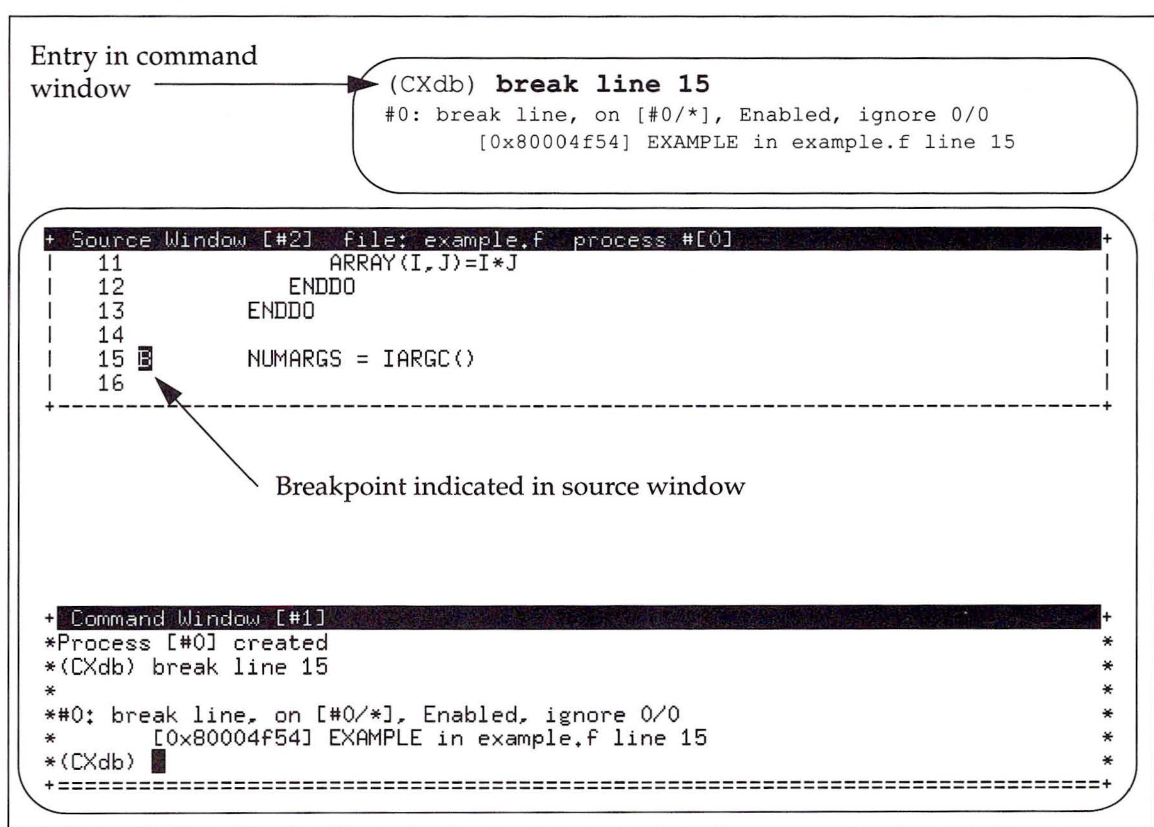


Figure 48 shows that the `break line 15` command is entered in the command window. This command sets a breakpoint at line 15 of the current source file, `example.f`.

Figure 48 also shows that a marker (the letter `B` in reverse video) appears in the source window to indicate the location of the breakpoint.

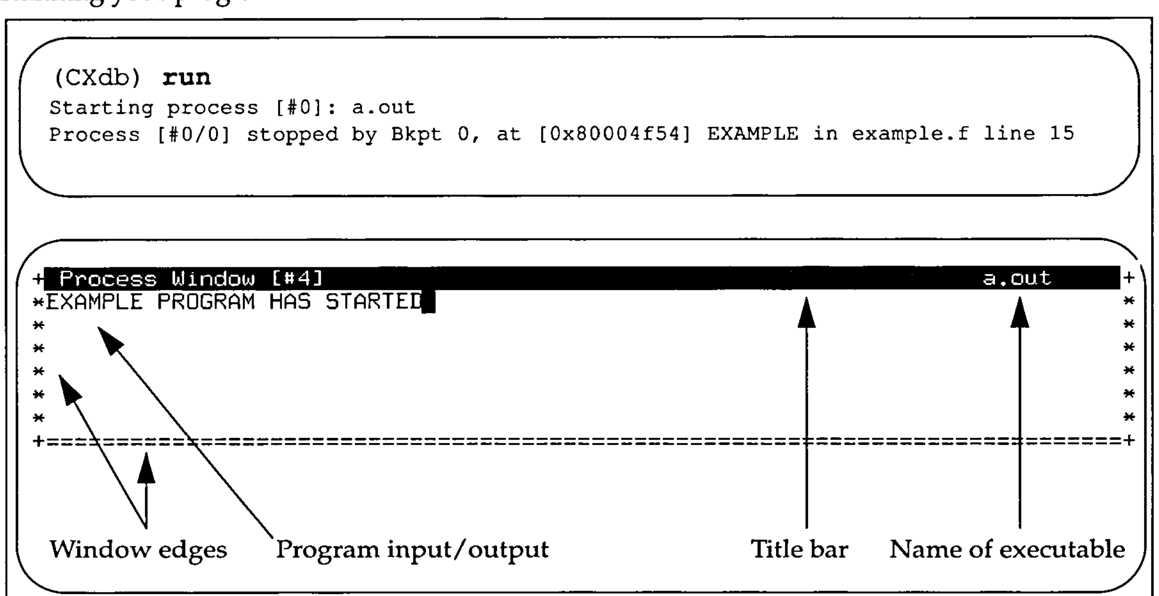
Running a program

After loading an executable file, you can run it from within CXdb by using the `run` command. The `run` command does the following:

- Creates a process from the executable file
- Associates the image of the process with the current process object and the related compiler-generated data files
- Starts execution of the process
- Brings up the process interface window for program input and output

The `run` command automatically brings up the process interface window, which allows interactive input and output with your program. This window keeps program input/output separate from CXdb input/output. Figure 49 illustrates the process interface window being brought up by the `run` command.

Figure 49
Running your program

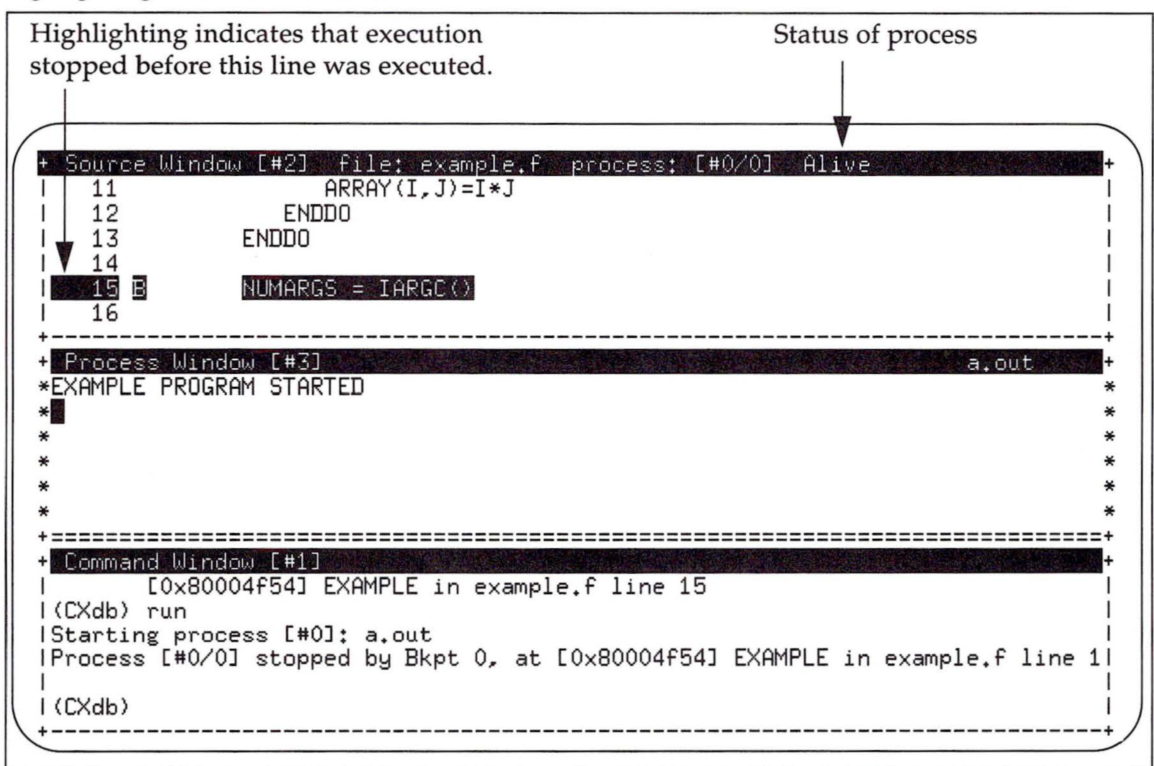


The process interface window is automatically placed in the middle third of the screen, between the source window and the command window. As illustrated in Figure 49, CXdb displays a message in the command window indicating that the breakpoint at line 15 has stopped execution of the process.

The process interface window becomes the active window, because it has just been created. Remember that to enter a command in the command window, you must make the command window active. From this point on in the chapter, the keystrokes needed to make the command window active will not be shown.

When execution stops, the source window updates to reflect the current state of the process. Highlighting also appears in the source window to indicate where execution has stopped, as illustrated in Figure 50.

Figure 50
Highlighting in the source window



The highlighting in Figure 50 indicates that execution has stopped at line 15 of the source code. The highlighted line number means that execution has stopped at the beginning of line 15, so no part of this line has been executed yet. When process execution resumes, it will start at line 15.

The highlighted source code represents the portion of code that is currently active. The amount of source code that is highlighted depends on the size of the source unit that is active. For more information about source units and their effect on highlighting refer to the section, "Effect of default granularity on highlighting," in Chapter 5.

Printing data

When the process is stopped, you can display the contents of program variables. The `print` command is one way to display variables.

Figure 51 shows the use of the `print` command to display the variable `J` from the example program. The response to the command indicates that `J` is a 4-byte integer with a current value of 5.

Figure 51
Printing a single variable

```
(CXdb) print J
(INTEGER*4) 5
```

A feature known as *completion* makes it easy to enter long variable names or command names. With completion, you can type just the first few letters of a command or variable name, then press TAB. CXdb fills in the rest of the name if it recognizes that name as unique. Figure 52 illustrates the use of completion to print the elements of an array named `ARRAY`.

Figure 52
Using completion on a command and a variable name

```
(CXdb) pr [TAB] AR [TAB] [RETURN]
INTEGER*4(1:4, 1:4)
(1..4,1) :  1 2 3 4
(1..4,2) :  2 4 6 8
(1..4,3) :  3 6 9 12
(1..4,4) :  4 8 12 16
```

Killing a process

To terminate the program, use the `kill process` command. This command asks for confirmation before killing the process, as shown in Figure 53.

Figure 53
Killing a process

```
(CXdb) kill process
Kill process [#0]? y
Terminated execution of Process [#0]
```

Killing the process does not delete the process object. The process object stores all the information about the process, including breakpoint settings. Therefore, the breakpoints remain set even though the process is killed.

Passing arguments to the process

Many programs are written to accept arguments passed to them from the shell. You can execute these programs in CXdb by including the arguments as a quoted string with the `run` command. Figure 54 illustrates how to do this.

Figure 54
Passing an argument with the `run` command

```
(CXdb) run "-flag"
Starting process [#0]: a.out -flag
Process [#0/0] stopped by Bkpt 0, at [0x80004f54] EXAMPLE in example.f line 15
```

The `run` command in Figure 54 passes the argument `-flag` to the process. This command also makes the process interface window the active window.

Because the breakpoint settings are stored in the process object, the breakpoint set on line 15 in Figure 48 is still enabled. This breakpoint stops execution of the program, as indicated by the output message in Figure 54.

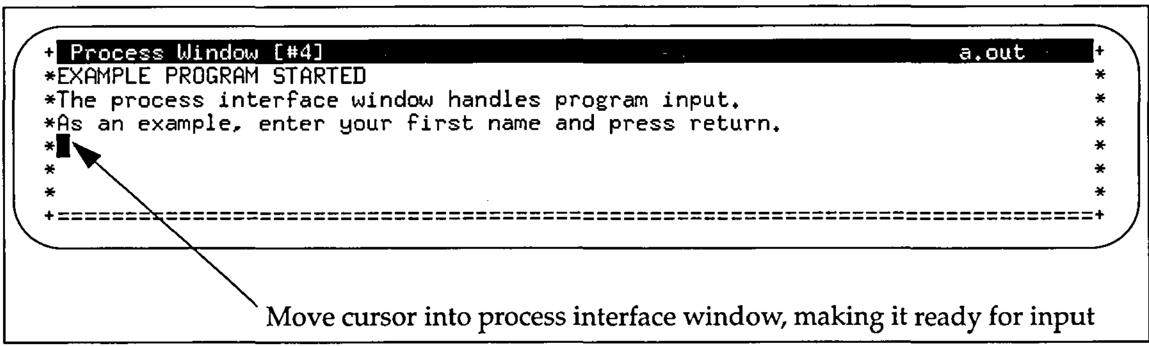
To resume execution of the program, use the `continue` command, as shown in Figure 55.

Figure 55
Continuing process execution

```
(CXdb) continue
Resuming execution of Process [#0/*]
```

The `continue` command in Figure 55 causes process execution to resume, beginning with line 15 of the program. The CXdb prompt does not return in the command window because process execution has not stopped. The program is waiting for input in the process interface window, as shown in Figure 56.

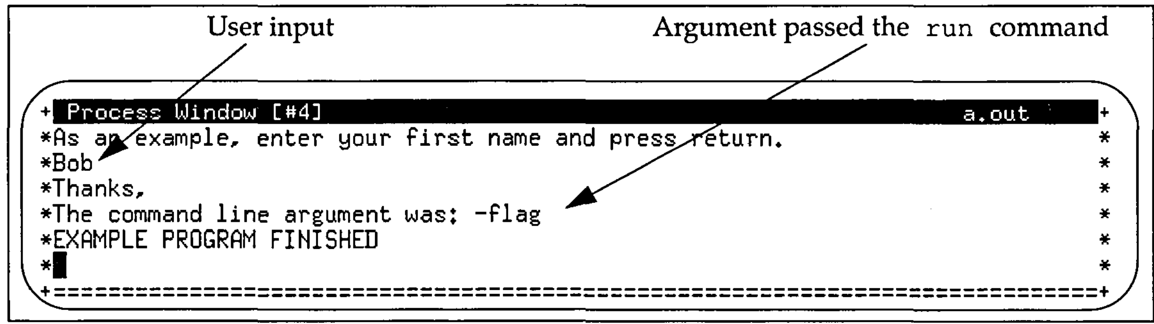
Figure 56
Program waiting for input in the process interface window



To enter input into the process interface window, the window must be active. Activate the process interface window by pressing **META-p**.

Providing the appropriate input in the process interface window enables this example program to execute to completion.

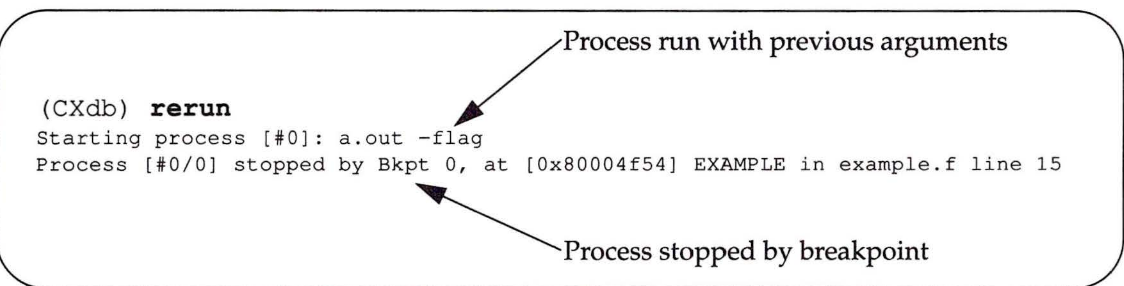
Figure 57
Providing input through the process interface window



In Figure 57, the output from the program indicates that the argument `-flag,` specified in the `run` command in Figure 54, was received. The program has finished execution.

You can run the program again with the same arguments by using the `rerun` command. This command also brings up the process interface window again. Figure 58 illustrates this command.

Figure 58
Rerunning a process with the same arguments



Running a command in background

A command that causes process execution can be placed in the background with relation to CXdb. This means that the `(CXdb)` prompt reappears, and you can continue to enter other CXdb commands while the process is executing. When the command that started process execution finishes, CXdb issues a message in the command window.

To execute a command in background, include the ampersand (`&`) at the end of the process execution command. Figure 59 illustrates background execution with the `continue` command.

Figure 59
Running a command in background

```
(CXdb) continue &
Command [#159] backgrounded

Resuming execution of Process [#0/*]
(CXdb)◄———————— Command prompt returns
```

To check the status of the process, use the `info process` command, as illustrated in Figure 60.

Figure 60
Obtaining information about a process

```
(CXdb) info process
status of process [#0]:

      executable: a.out
       arguments: -flag
fixed scheduling: off
          pshell: csh
    image status: created pid 14796, state = running
     working dir: /usr/lib/cxdb/examples
    default step: statement
default language: Fortran
         threads: 1
  current thread: 0


source file search path:
        .
```

Indicates that process is running

To check the status of the debugging session, use the `info cxdb` command, as illustrated in Figure 61.

Figure 61
Obtaining information about a debugging session

```
(CXdb) info cxdb
Current CXdb state:

     ENVIRONMENT:
              pid: 11013
              cwd: /usr/lib/cxdb/examples
   command modes: echo off, logging off, noclobber off
           cmdout: Window #1
           cmderr: Window #1
           cmdlog:
         evalopts: fpmode = native, iprecision = 4, rprecision = 4
            shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
        step size: statement
    process shell: csh
           fpmode: native
      memory size: (none)
   memory formats: byte=(none), halfword=(none), word=(none)
                    longword=(none), quadword=(none)
      search path:
          .


        PROCESSES:
     process [#0]: created pid 14796, state = running, executable = a.out
                    shell = csh


ACTIVE COMMANDS:

command [#159] - continue &
```

Indicates that the `continue` command is active in background mode

The information displayed by the `info cxdb` command cannot be viewed all at once in the command window. You can scroll the command window using **META-A** and **META-B** to view output from the command.

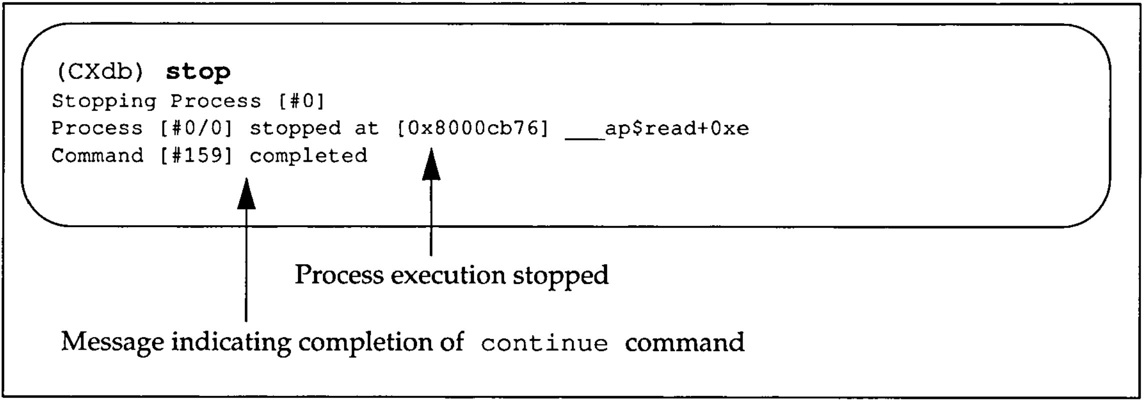
Stopping a running process

If you have placed a process execution command in the background, you can use the `stop` command to stop your process. The `stop` command terminates the currently running process execution command, and therefore stops the process.

If the process is running, and you have not placed the process execution command in the background, you can abort the command by typing **CTRL-c**. This also stops the process.

Figure 62 illustrates stopping the process with the `stop` command.

Figure 62
Stopping process execution



CXdb issues a message telling you where the process stopped. CXdb also issues a message explaining that the `continue` command that was placed in the background in Figure 59 has completed.

Getting help online

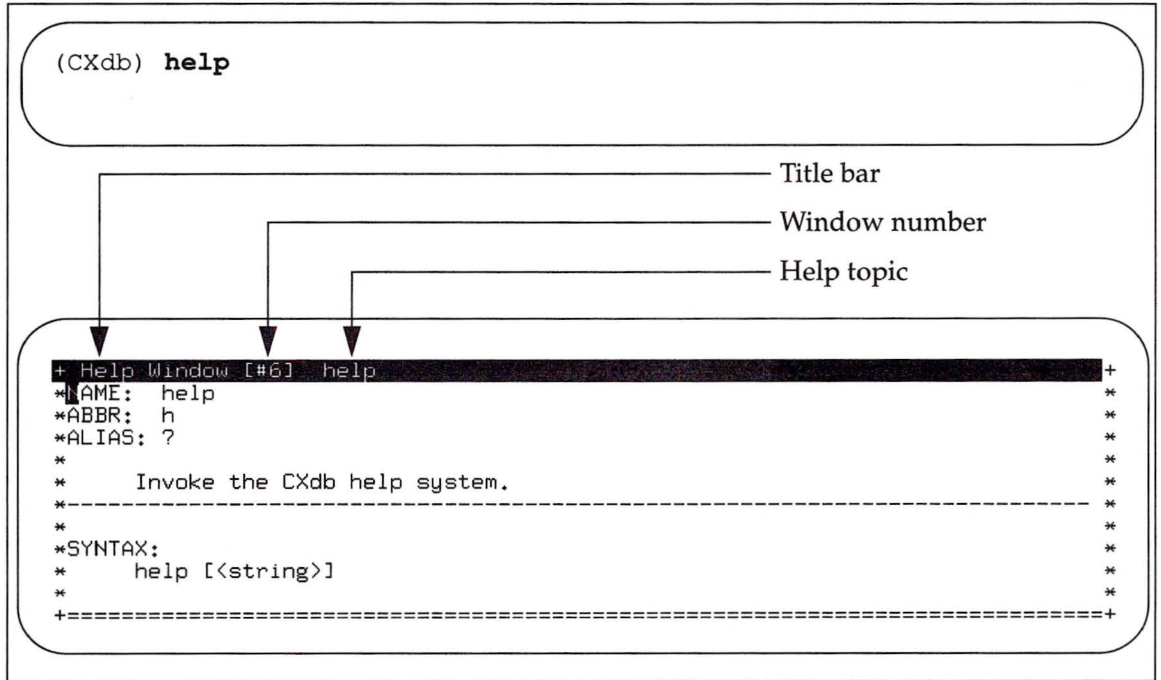
CXdb has an extensive online help system that contains the same information as the *CONVEX CXdb Reference* manual. The help topics cover 4 categories:

- **Commands**—Complete description, syntax, and examples of each CXdb command.
- **Concepts**—Explanations of the major concepts that tie related commands together.
- **CXdb messages**—Text and explanations for messages displayed by CXdb.
- **Parameters**—Description, syntax, and examples of parameters used with the CXdb commands.

The help system has its own window for displaying help topics. You can call up the help window from the command window with the `help` command.

Figure 63 shows the help window being brought up with the `help` command. By default, the help window covers the top half of the screen.

Figure 63
Opening the help window with the `help` command in Maryland Windows



The help window includes:

- **Window number**—This is a unique number that identifies each window in CXdb.
- **Help topic**—The current topic being displayed. In this case, the page for the `help` command is being displayed.
- **Title bar**—The title bar indicates that this is the help window.

When you simply type in the `help` command, the corresponding page for the `help` command is displayed. Because the help window was just created, it becomes the active window. You can scroll through the text to read about the `help` command using **CTRL-n** and **CTRL-p**, or **CTRL-v** and **META-v**.

Figure 64
Using the `help` command to search for a topic

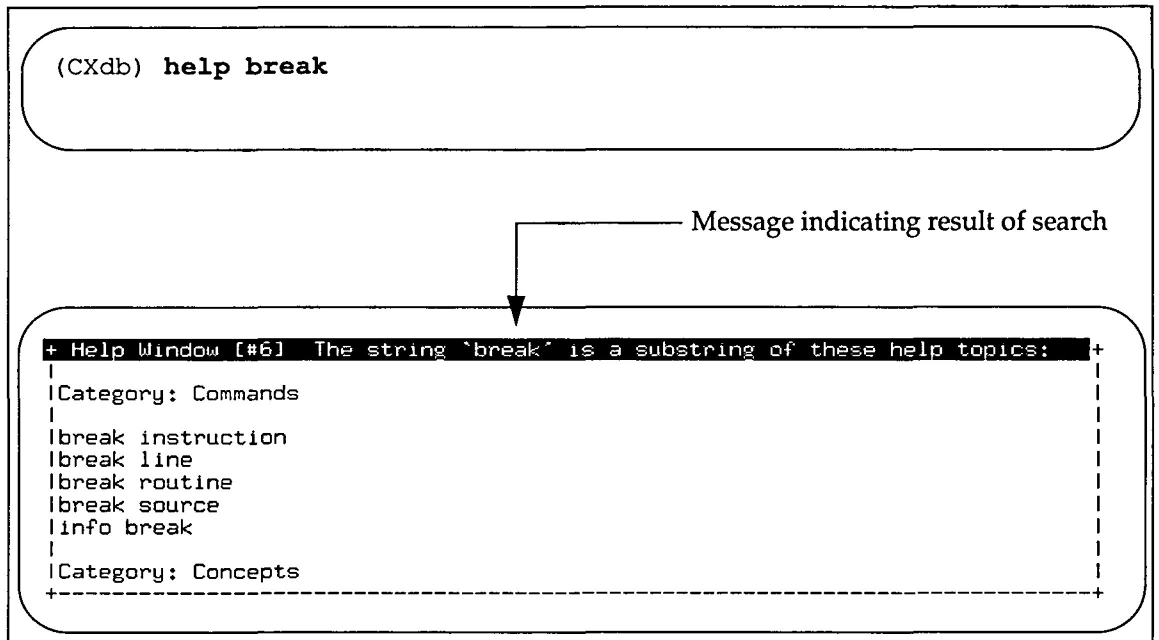
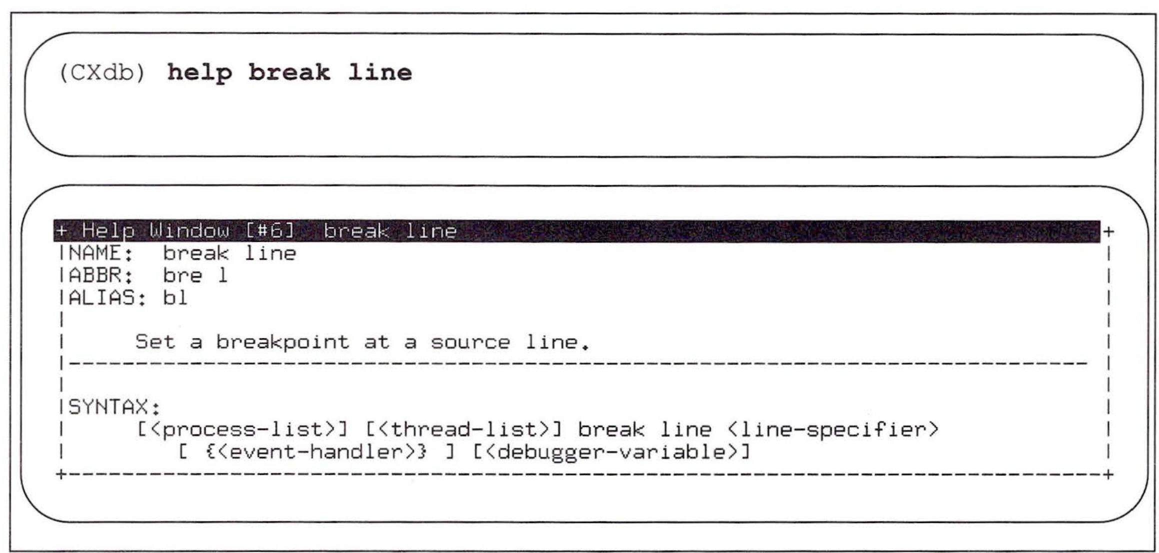


Figure 64 illustrates the use of the `help` command to search for a topic. All commands and concepts that have the string "`break`" in their title are displayed, separated by category.

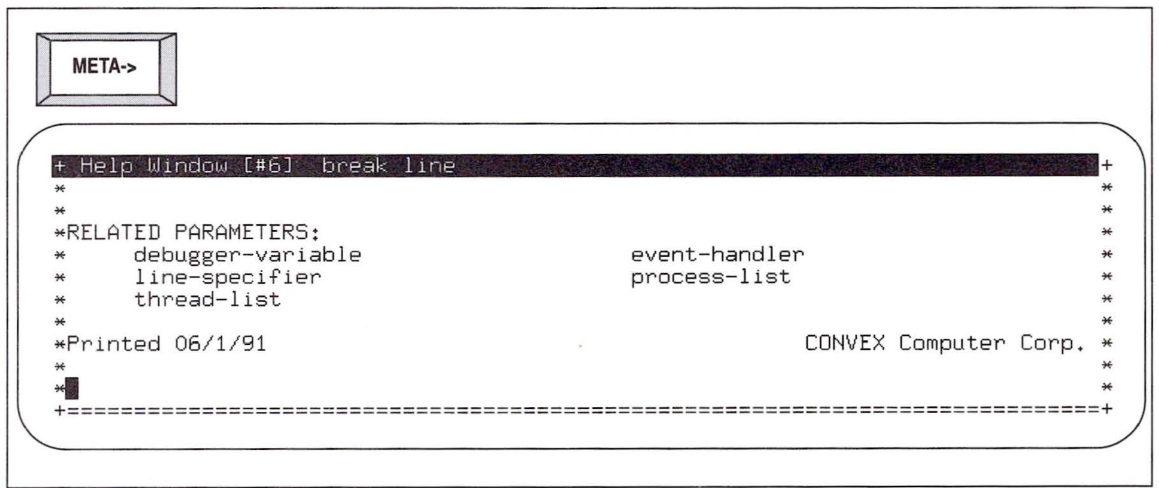
You can get help on a specific command by specifying the name of the command with the `help` command, as shown in Figure 65.

Figure 65
Getting help on a particular topic



The `help` command in Figure 65 brings up the reference page for the `break line` command. At the bottom of each help page is a list of related commands, concepts, and parameters. If you activate the help window, you can scroll to the bottom of the help text using **META->**, as shown in Figure 66.

Figure 66
Scrolling to the bottom of the help window



The related topics section of the help pages can be used to learn about commands, concepts, and parameters that work together in CXdb.

Raising and lowering a window

When using the Maryland Windows interface, you may overlap windows to maximize space on the screen. You can stack any number of windows on top of each other. You raise and lower windows to view the window you want to see.

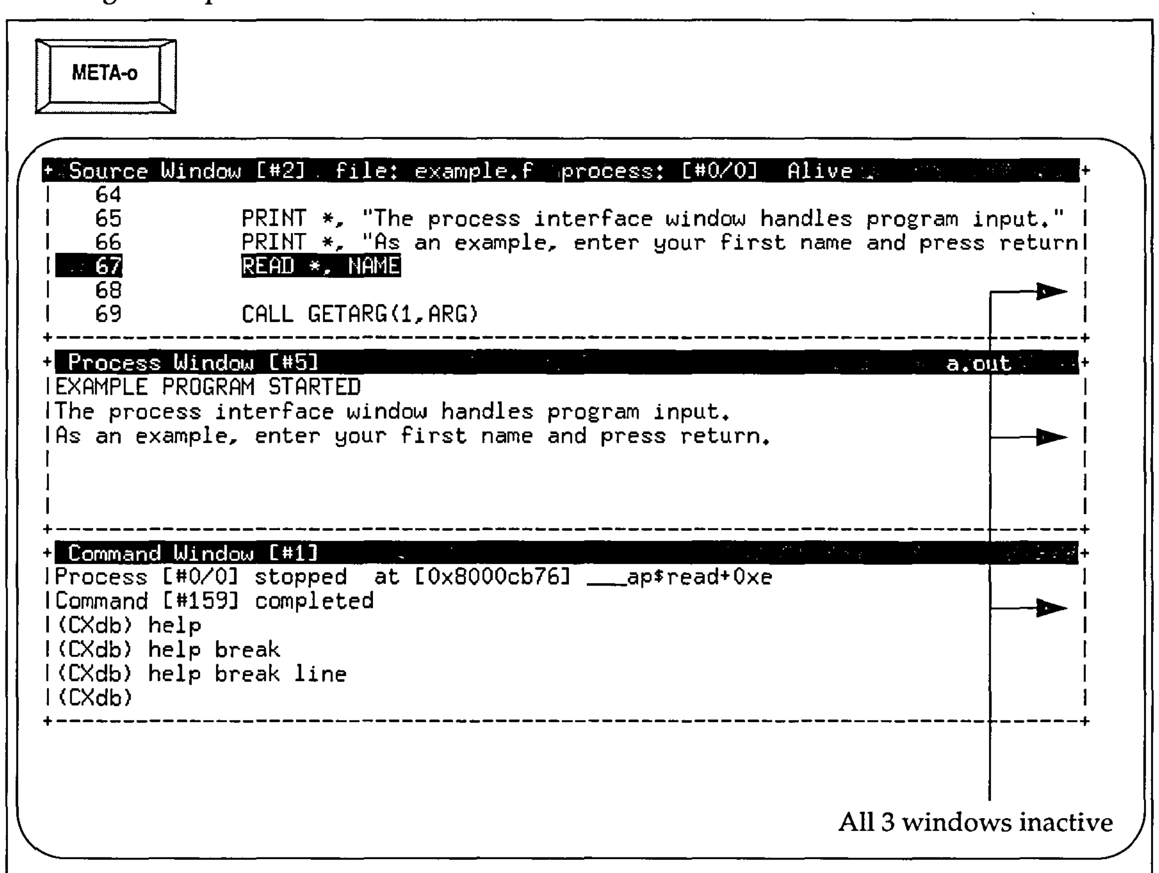
Table 3 lists the keystrokes to raise and lower the active window.

Table 3
Raising and lowering a window

Keystroke	Direction
META-r	Raise window
META-o	Lower window

You can raise or lower only the active window. If you lower a window so that it is completely hidden, it still remains the active window. Figure 67 demonstrates lowering the active window.

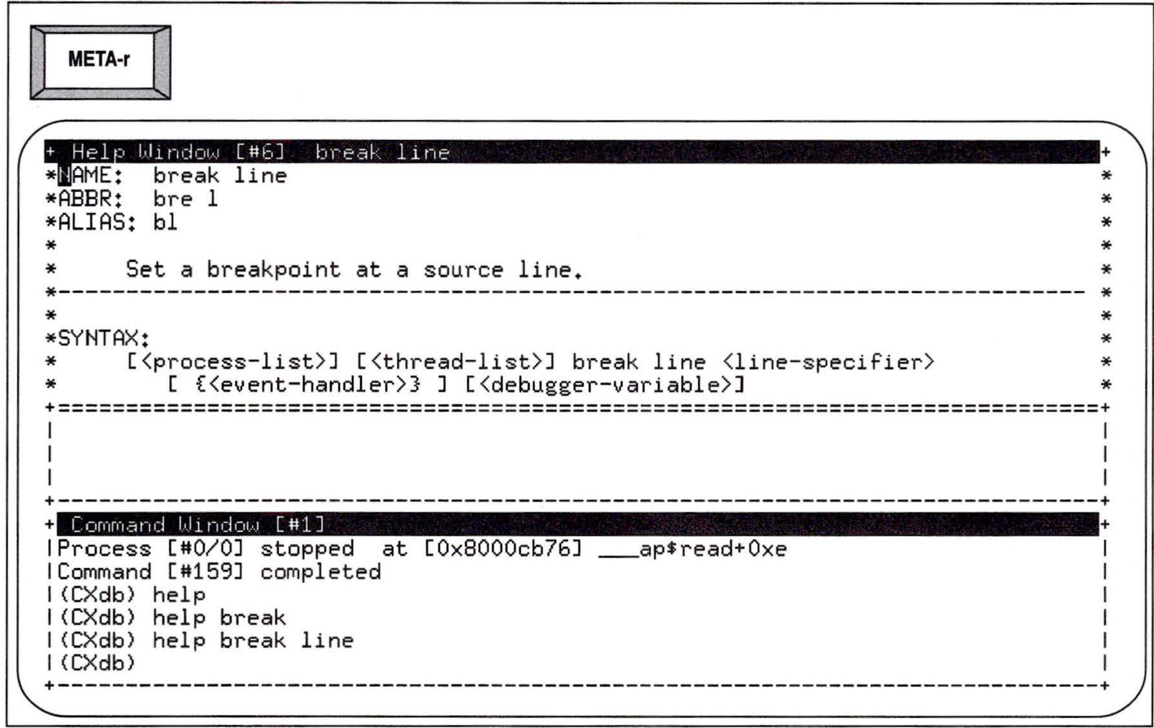
Figure 67
Lowering the help window



The help window is now hidden by the source window and the process interface window. However, the help window is still the active window, even though it cannot be seen. Any keystrokes issued now still affect the help window.

Figure 68 illustrates the use of **META-r** to raise the help window. Again, the window covers over the source window and part of the process interface window.

Figure 68
Raising the help window



Moving and resizing a window

Because of the amount of information available in CXdb, you may not have enough room to display all of it at once. By resizing or moving the windows, you can match the Maryland Window interface to your debugging needs.

As you have seen, all of the windows have a default placement. If you want the window to be located elsewhere, you can move the window after it is created.

In addition to be being able to move a window, you can also resize it. This enables you to make each window as large as you need (within the limits of the CRT).

Table 4 shows the keystrokes used in moving and resizing a window.

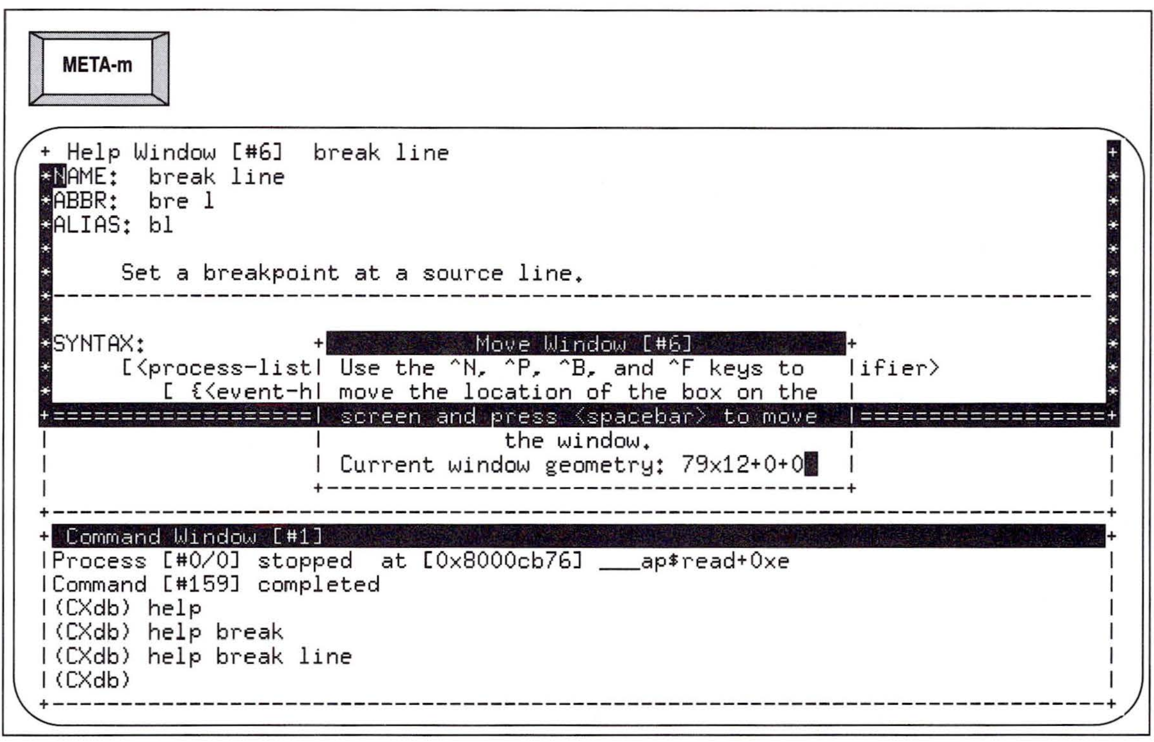
Table 4
Moving and resizing a window

Keystroke	Direction
META-z	Resize window
META-m	Move window

When moving a window, you cannot extend the window beyond the borders of the screen. That is, when you are moving the window down, the bottom of the window cannot be moved beyond the bottom of the screen.

Figure 69 shows the **META-m** keystroke being used to move the help window.

Figure 69
Moving the active window



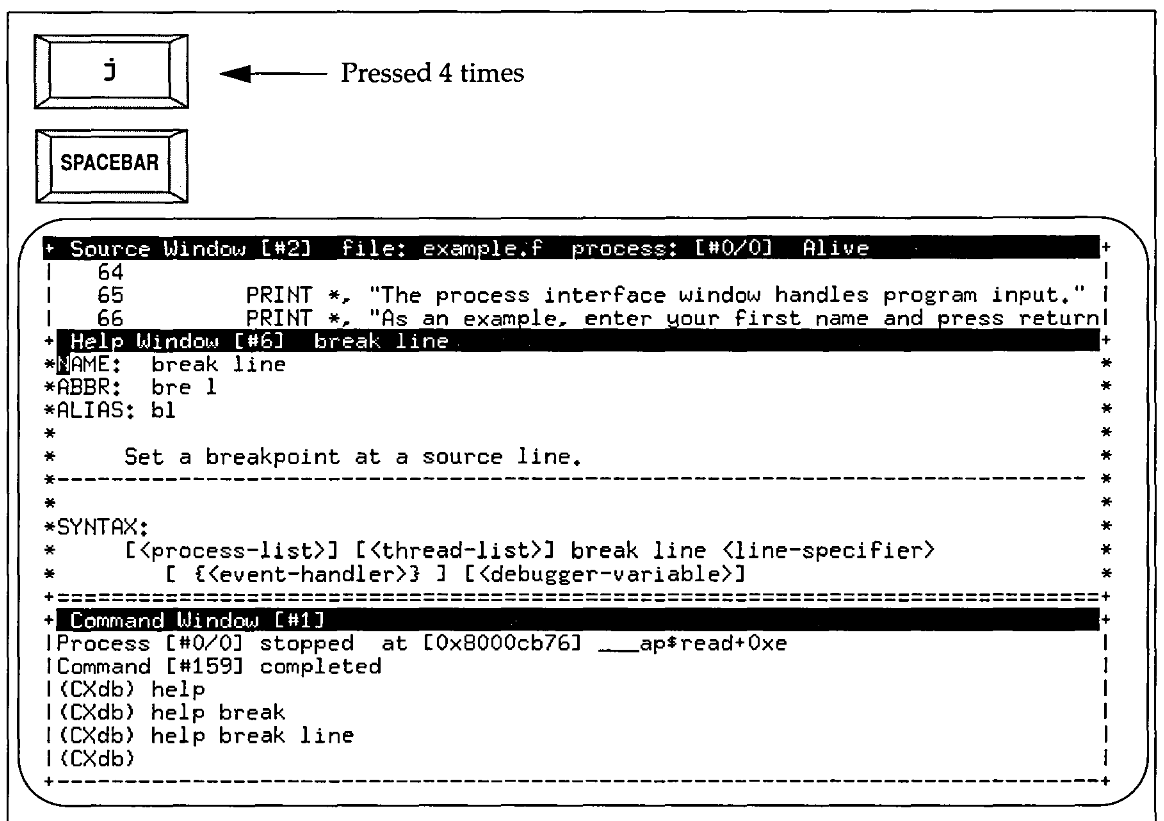
CXdb displays a message in the middle of the screen explaining how to move the window. The keystrokes to move a window are listed below in Table 5.

Table 5
Moving the window placement cursor

Keystroke	Direction
j or **CTRL-n**	Down 1 line
k or **CTRL-p**	Up 1 line
h or **CTRL-b**	Left 1 character
l or **CTRL-f**	Right 1 character
SPACEBAR	Anchor window

Figure 70 illustrates the placing of the help window. The series of keystrokes listed move the window down 4 lines to the center of the screen. Once you have moved the window to the appropriate location, you must anchor the window by pressing **SPACEBAR.**

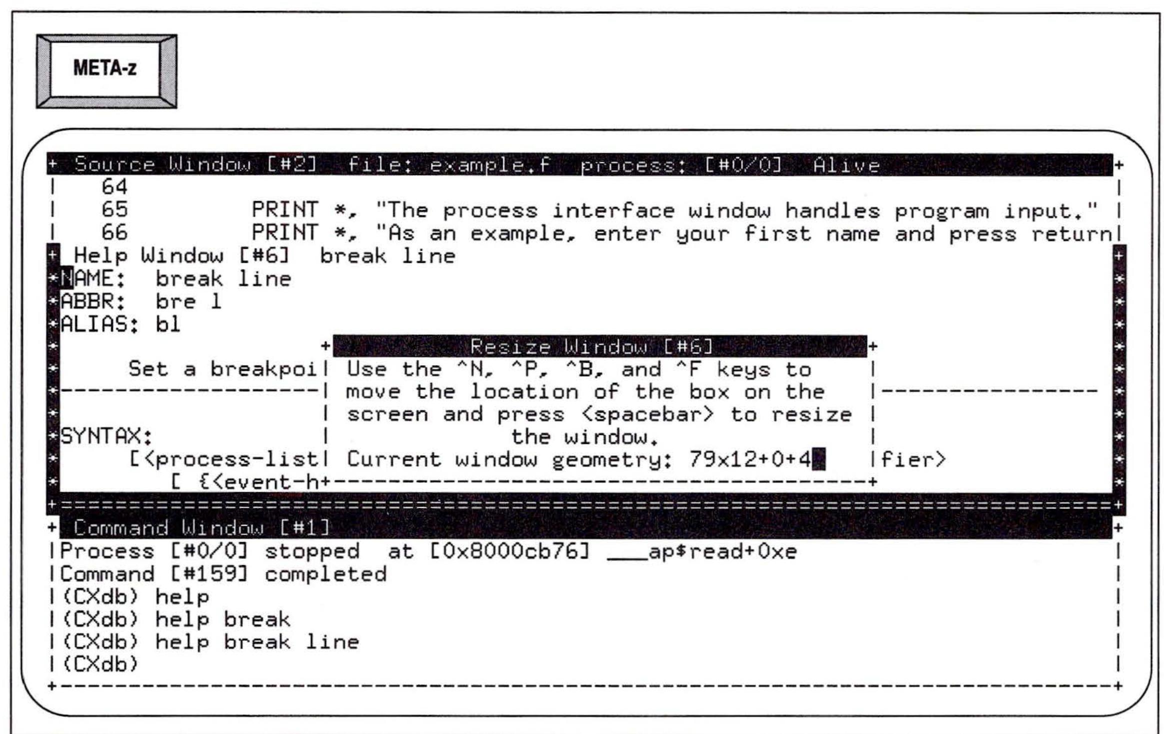
Figure 70
Moving and anchoring the help window



The help window is now located in the middle of the screen. The window that you move remains active after you anchor it in its new location.

You can resize a window. Figure 71 illustrates the use of **META-z** to resize the help window. When you resize a window, you change the position of the bottom-right corner of the window.

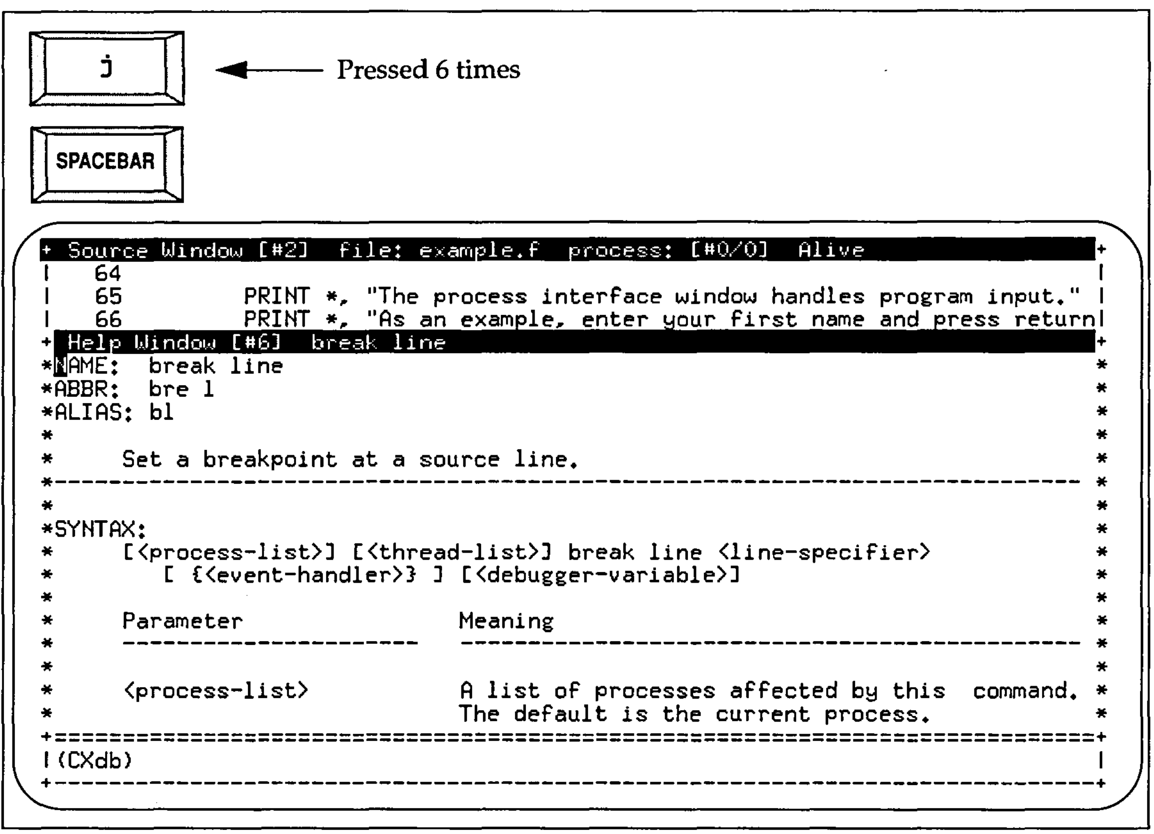
Figure 71
Resizing the active window



When you use **META-z** to resize a window, CXdb again places a message box in the center of your screen explaining how to resize the window. The keystrokes to resize the window are the same as those for moving the window. The only difference is that you are moving the bottom-right corner of the window, rather than the entire window.

Figure 72 shows the resizing of the help window to extend it to the bottom of the screen.

Figure 72
Enlarging the window



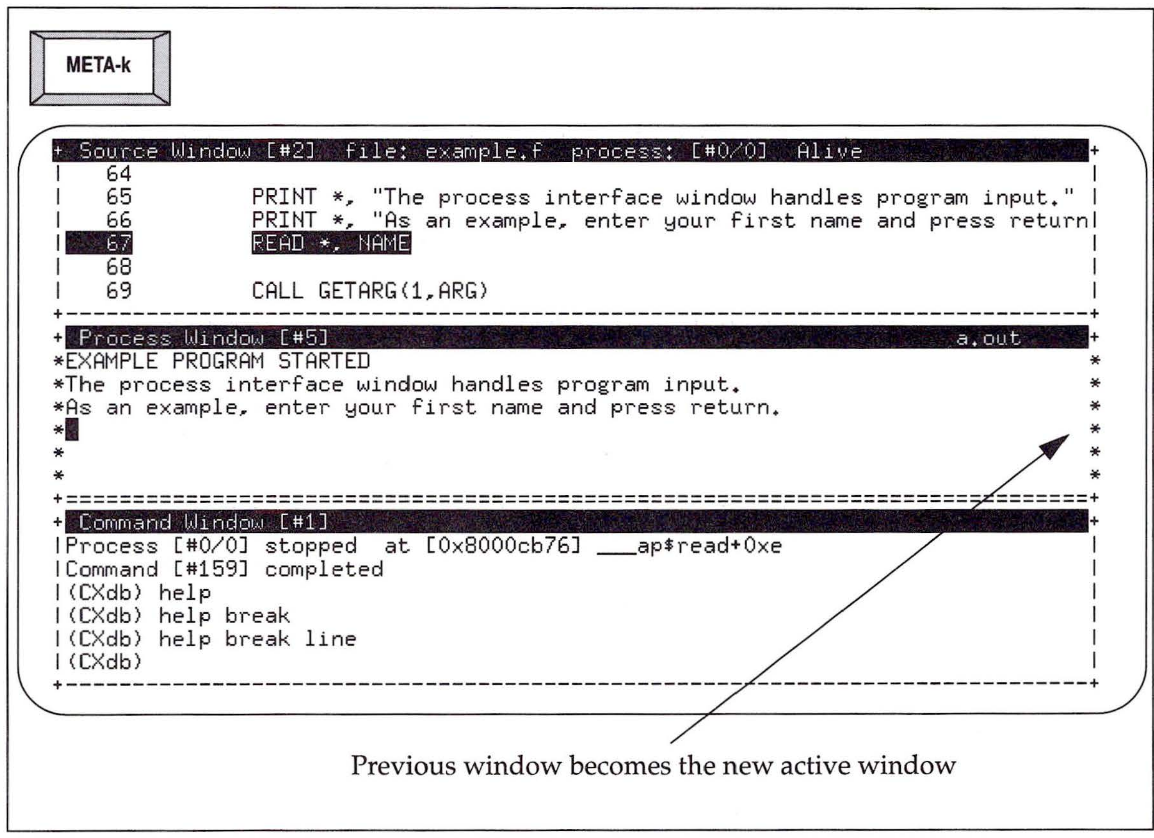
The help window now takes up a majority of the screen. By enlarging the size of windows and using the raising and lowering keystrokes, you can see more information at once and still having multiple windows available.

Closing a window

When you finish with a window, you may want to close it to free up more space on your screen. The **META-k** keystroke closes the active window.

When you close the active window, the previous window becomes the active window. The previous window is the window that was last active before the current window became active. Figure 73 illustrates how to close the help window by using **META-k**.

Figure 73
Closing the help window



Note that the previous window (last window to be active) is now active. In this case, the process interface window (window 5) is now the active window.

Quitting CXdb

To exit CXdb, issue the `quit` command from the command window. This command does the following:

- Kills all existing processes and process objects, with your confirmation
- Kills any active CXdb commands
- Closes any files opened by CXdb
- Closes all CXdb windows

Figure 74 illustrates the `quit` command being entered in the command window.

Figure 74
Quitting CXdb

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

The operating system prompt returns.

Breakpoints, tracepoints, and watchpoints

4

This chapter describes how to use 3 types of eventpoints: breakpoints, tracepoints, and watchpoints. These have special names because they are the most common types of eventpoints used when debugging.

Each type of eventpoint has its own set of actions, called a handler, that executes when the eventpoint triggers.

This chapter covers the following commands in detail:

- `break line`
- `break routine`
- `info break`
- `info event`
- `info trace`
- `info watch`
- `remove event`
- `trace line`
- `trace routine`
- `watch`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 75.

Figure 75
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 75 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 76.

Figure 76
Starting the example program

```
(CXdb) break routine CHAPTER4
#0: break routine, on [#0/*], Enabled, ignore 0/0
       [0x80001356] CHAPTER4 in chapter4.f line 6
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001356] CHAPTER4 in chapter4.f line 6
```

The `break routine` command in Figure 76 sets a breakpoint at the beginning of the routine called `CHAPTER4`, which is a FORTRAN routine used for all examples in this chapter. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

Breakpoints

A breakpoint is a type of eventpoint. Eventpoints are traps that you place in a process to wait for a particular event to occur. When the event occurs, the eventpoint is triggered.

You place breakpoints in your program where you want your program to stop. They wait for program execution to reach a particular location. When program execution reaches the breakpoint, the breakpoint is triggered, and the default handler for breakpoints executes.

The default handler for breakpoints performs two actions:

- Stops the process
- Displays a message in the command window, telling you the breakpoint was triggered

There are 4 ways to set a breakpoint:

- **At a line number**—The breakpoint is placed at the beginning of the line.
- **At a routine**—The breakpoint is placed at the first executable statement of the routine.
- **At a source unit**—The breakpoint is placed at the first instruction of the source unit. Source units are described in more detail in Chapter 5.
- **At an instruction**—The breakpoint is placed at the instruction.

The first two methods are demonstrated in this chapter. Setting a breakpoint at a source unit is discussed in Chapter 5.

Tracepoints

Tracepoints are the same as breakpoints except for one important difference: they do not stop your process. When a tracepoint is triggered, a message is sent to the command window telling you the tracepoint was triggered, but process execution continues.

Tracepoints are ideally suited for tracking the execution of your program. Tracepoints are set in the same 4 ways as breakpoints: at a line number, at a routine, at a source unit, or at an instruction.

Watchpoints

Watchpoints are eventpoints that wait for the contents of a memory region to change. After each program statement, CXdb checks to see if the memory region has changed value. Watchpoints can monitor program variables or address ranges.

Watchpoints that you create exist only for the current process. When the process terminates, or when you create a new process by running the executable again, all watchpoints are removed.

Note

Because of the additional overhead, watchpoints can significantly reduce the speed of execution. Before setting a watchpoint, you should set a breakpoint near the region of code that you wish to monitor.

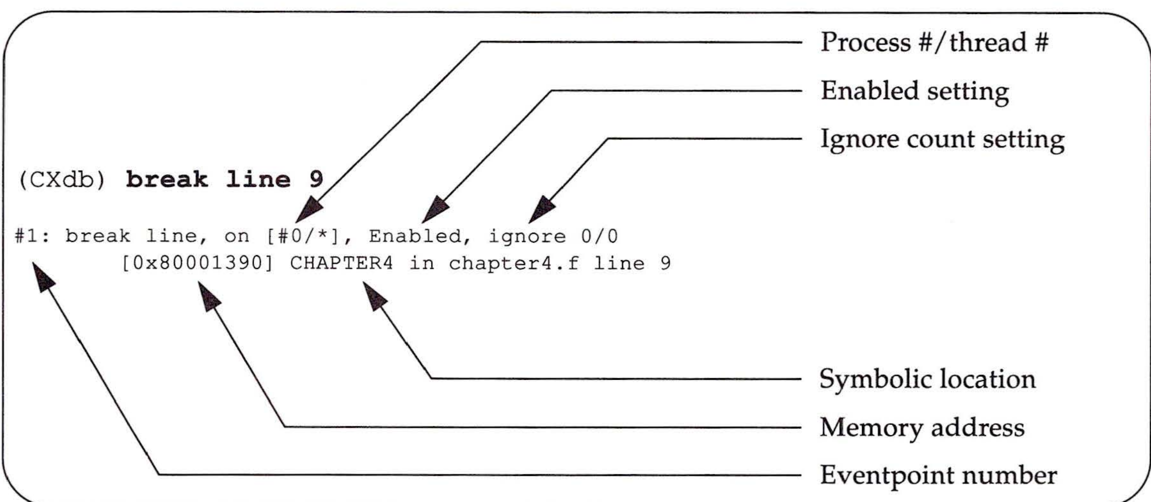
Setting a breakpoint

This section describes how to set a breakpoint at a line number and at a routine.

Setting a breakpoint at a line number

To set a breakpoint at a line number in the current source file, use the `break line` command followed by the line number. The `break line` command in Figure 77 sets a breakpoint at line 9 of the current source file.

Figure 77
Setting a breakpoint at a line number



Note that the current source file, `chapter4.f`, contains the subroutine `CHAPTER4` and is a separate file from the main routine. CXdb numbers the lines of your source file sequentially for each different file that was compiled with the `-cxdb` option.

CXdb displays the following settings for the new breakpoint:

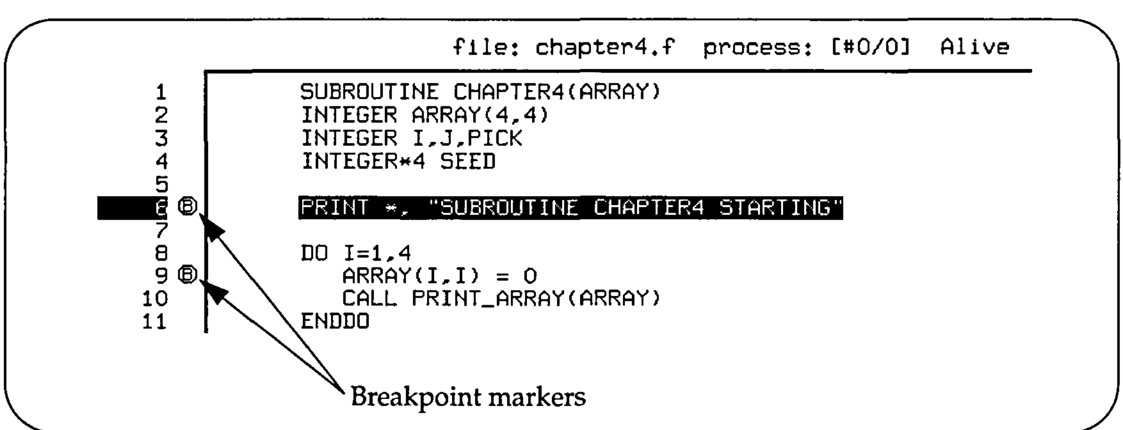
- **Process #/thread #**—The process object and process threads on which the breakpoint is set. In Version 1.1 of CXdb, there is only one process number, 0. If the breakpoint is active on all threads (even if there is only one) of the process, the thread number is shown as an asterisk.
- **Enabled setting**—The breakpoint status, which is either enabled or disabled. All eventpoints are enabled when they are created. An enabled eventpoint can be triggered, while a disabled eventpoint cannot. Enabling and disabling eventpoints is discussed in more detail in Chapter 6, Section, "Enabling and disabling eventpoints."

- **Ignore count**—The number of times the breakpoint has been ignored, and the total number of times you want execution to ignore it. Ignore counts are covered in Chapter 6, the section "Setting ignore counts."
- **Symbolic address**—The source code location of the breakpoint. You can use this information to make sure the breakpoint is at the proper location.
- **Memory address**—The hexadecimal address where the breakpoint is located.
- **Eventpoint number**—The number used in subsequent commands when you want to refer to this breakpoint. All eventpoints are given their own number when they are created. Eventpoints are numbered sequentially starting with 0.

In addition to displaying information about the breakpoint in the command window, CXdb displays a symbol in the source window next to the line number you selected.

Figure 78 shows the breakpoint markers for the breakpoint at line 9 and the breakpoint at line 6.

Figure 78
Breakpoint markers in the source window for CXwindows

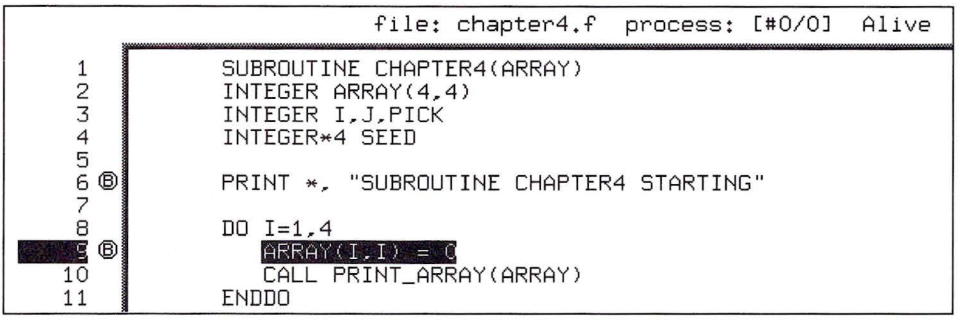


The breakpoint at line 6 was created with the `break routine` command shown in Figure 76.

Figure 79 shows how to continue process execution by using the `continue` command.

Figure 79
Triggering the breakpoint at line 9

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80001390] CHAPTER4 in chapter4.f line 9
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80001390] CHAPTER4 in chapter4.f line 9
```



Each time through the loop, execution is stopped by the breakpoint on line 9. CXdb displays a message in the command window that tells you which breakpoint stopped execution, as well as the memory address and symbolic location where the process stopped.

Setting a breakpoint at a routine

As you have already seen, you can set a breakpoint at a routine in your program by using the `break routine` command. The `break routine` command accepts any language expression as a parameter. CXdb then performs the following three steps:

- Evaluates the language expression to an address
- Finds the routine that contains the address
- Places the breakpoint at the first source unit of the routine

The first source unit of a routine is usually the first executable statement in the routine. The advantage of the above three steps is that you can specify a routine using the name of the routine without having to know the exact starting address or line number.

The breakpoint is placed just after the preamble of the routine. The preamble of a routine sets up the stack for that routine. Unless you specifically want to debug the preamble, the `break routine` command is the simplest method of stopping program execution at the start of a particular routine.

Figure 80 shows the `break routine` command setting a breakpoint at the first source unit in the subroutine `PRINT_ARRAY`.

Figure 80
Setting a breakpoint at a routine

```
(CXdb) break routine PRINT_ARRAY

#2: break routine, on [#0/*], Enabled, ignore 0/0
        [0x8000502c] PRINT_ARRAY in example.f line 39
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 2, at [0x8000502c] PRINT_ARRAY in example.f line 39
```

The output from the `break routine` command is in the same format as that for the `break line` command. Note that this breakpoint, breakpoint 2, is located in the source file `example.f`, rather than the current source file.

Execution continues until it is stopped by the breakpoint in the `PRINT_ARRAY` routine. The source window updates to show the source code for the `PRINT_ARRAY` routine in the file `example.f`.

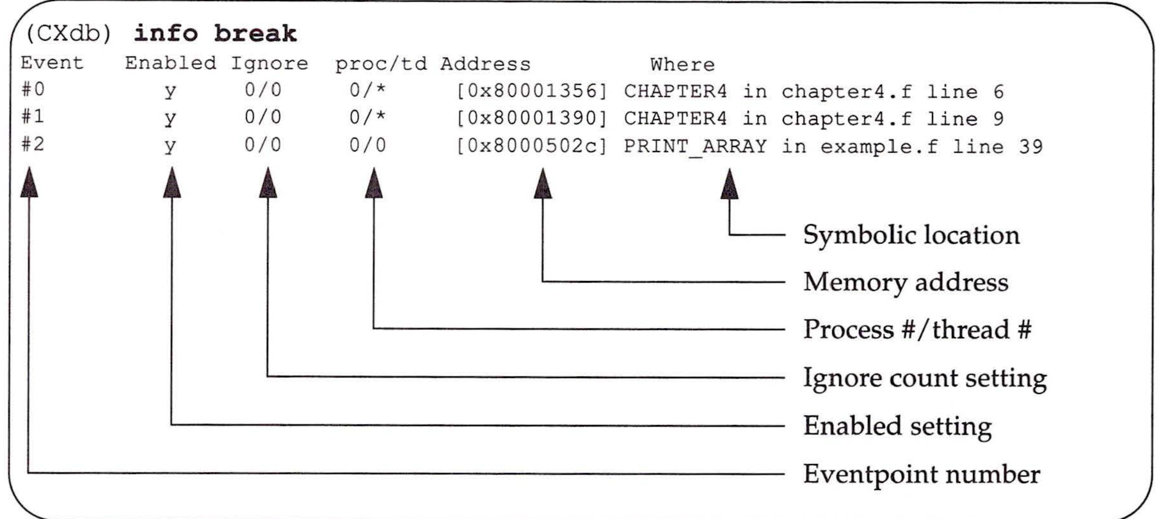
When program execution moves to a different source file, CXdb displays the source code surrounding the current point of execution in the new file. If there is no debugging information for a routine, such as a library routine that has not been compiled with the `-cxdb` option, CXdb displays the last routine for which it has debugging information (generally the calling routine).

Getting information about breakpoints

With the `info break` command, you can get information about all of the breakpoints that you have created.

The `info break` command displays the status of each breakpoint in the current process. The information displayed is in a tabular format that enables you to quickly check the status of all breakpoints, as shown in Figure 81.

Figure 81
Getting information about all breakpoints



The current settings for each breakpoint are displayed: the eventpoint number, enabled setting, ignore count setting, process number and thread number, memory address, and symbolic location.

The `info break` command can be used to find the number of a particular breakpoint. You specify a breakpoint's number in commands that manipulate it.

Removing a breakpoint

When you no longer need a breakpoint, you can remove it from your process. Chapter 6, the section, "Enabling and disabling eventpoints," describes how to disable a breakpoint without having to remove it completely.

Remove all types of eventpoints, including breakpoints, with the `remove event` command. To remove an eventpoint, you must specify its eventpoint number, as demonstrated in Figure 82.

Figure 82
Removing a breakpoint

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80001390] CHAPTER4 in chapter4.f line 9

(CXdb) remove event 2 <-------------------- Eventpoint to be removed
Eventpoint 2 removed
(CXdb) info break
Event    Enabled Ignore  proc/td Address          Where
#0          y      0/0     0/*     [0x80001356] CHAPTER4 in chapter4.f line 6
#1          y      0/0     0/*     [0x80001390] CHAPTER4 in chapter4.f line 9
```

The `continue` command continues process execution until it is stopped by breakpoint 1. The `remove event` command removes breakpoint 2. The number for the breakpoint was determined using the `info break` command, as shown in Figure 81.

Eventpoint numbers are not reused during a debugging session, so the next eventpoint created will be number 3.

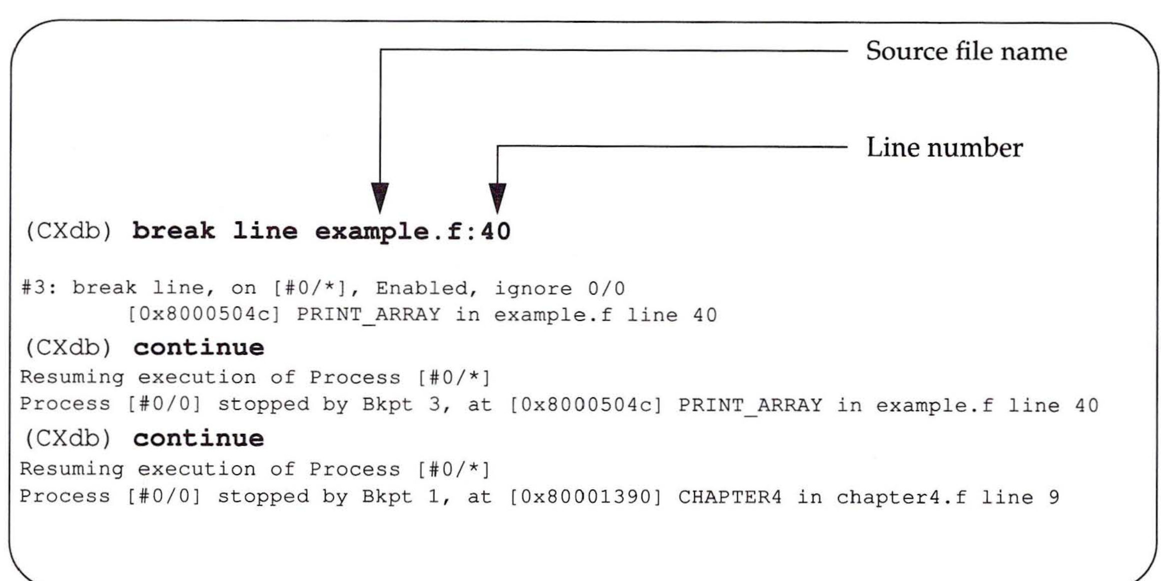
Specifying a source file for a breakpoint

You can set a breakpoint at a source line in a file other than the current source file. To do this, specify the source file name before the line number in the `break line` command. (A colon is used to separate the source file name from the line number.)

The source file must be included in the search path. Generally, this is the case if the source file is in the same directory as the executable. Search paths are discussed in Chapter 9, the section "Search paths."

In Figure 83, the `break line` command places a breakpoint on line 40 of the source file `example.f`. Line 40 is a `print` statement of the routine `PRINT_ARRAY`.

Figure 83
Setting a breakpoint in a different source file



The `break line` command creates breakpoint 3. Note that the breakpoint number is 3, even though breakpoint 2 no longer exists.

The first `continue` command resumes process execution. Breakpoint 3 stops the process at line 40 of the `PRINT_ARRAY` routine in the `example.f` source file. The second `continue` command continues process execution again. Breakpoint 1 stops the process at line 9 in the `chapter4.f` source file.

Removing multiple breakpoints

When you are debugging your program, you can create many breakpoints. You can remove multiple breakpoints at once using the `remove event` command, rather than having to remove one at a time.

To remove multiple breakpoints, separate each eventpoint number on the command line with a comma.

Figure 84
Removing multiple breakpoints

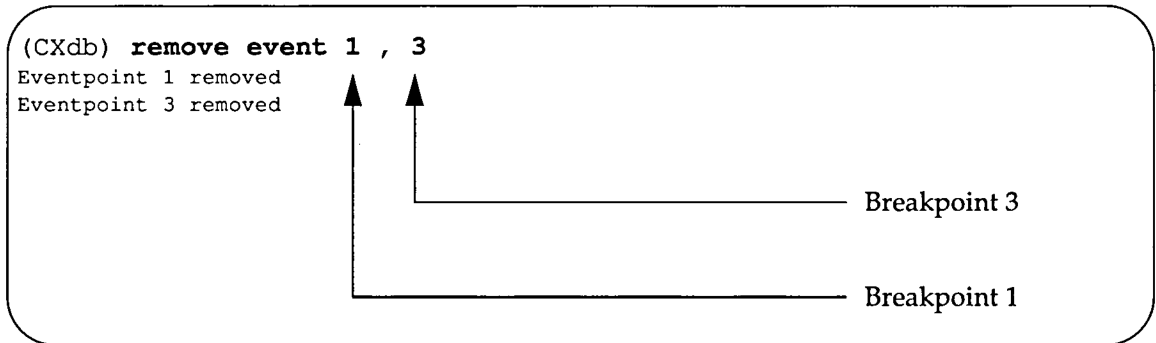


Figure 84 shows the `remove event` command removing the breakpoints on line 9 of the file `chapter4.f` and on line 40 of the file `example.f`.

Specifying an invalid line number

When specifying a line number for a `break line` command, you might specify a line that does not have any source units. This can occur because the line is a comment line, or because the source units for that line have been removed during optimization.

When you specify a line number that does not have any source units, CXdb looks for the next source line in the file that does have source units. CXdb then asks you if you wish to place the breakpoint at that location. If you answer with a `y,` CXdb places the breakpoint there. Otherwise, no action is taken and the CXdb prompt returns.

Figure 85
Specifying an invalid line number

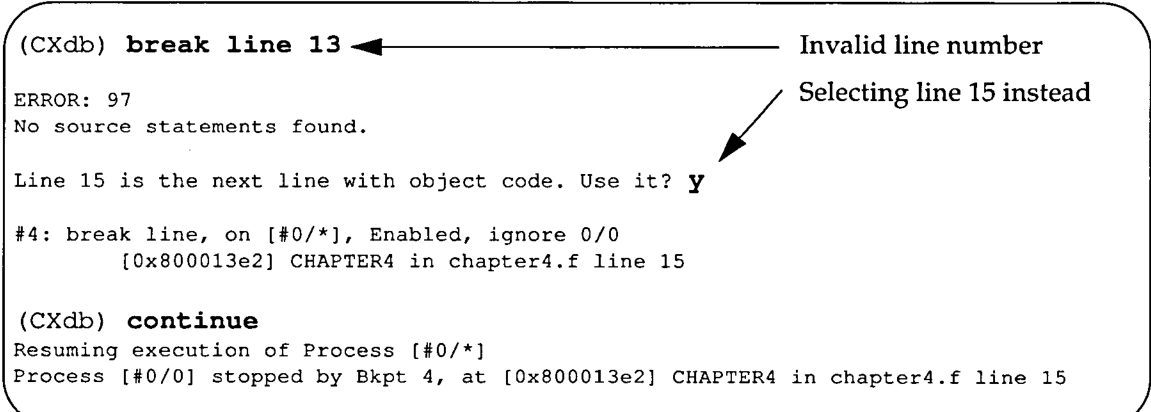


Figure 85 demonstrates CXdb's response when you select a line number that does not have any source units on it. CXdb asks if you want to use the next line with object code, in this case line 15. Responding with a `y` tells CXdb to place the breakpoint at the beginning of line 15.

The `continue` command continues process execution. Execution is stopped by breakpoint 4 at line 15 of the source file `chapter4.f`.

Working with tracepoints

Tracepoints are very similar to breakpoints. The methods of setting them are identical to those used for breakpoints. The important difference between tracepoints and breakpoints is that tracepoints do not stop process execution.

When a tracepoint is triggered, a message is displayed in the command window, just as with breakpoints. This message indicates that the tracepoint has been triggered. However, unlike breakpoints, process execution continues. This enables you to use tracepoints to track the flow of execution in your program.

Setting tracepoints

The general syntax for the `trace line` and `trace routine` command is the same as the corresponding command for breakpoints, as shown in Figure 86.

Figure 86
Setting a tracepoint at a line number

```
(CXdb) run
Process [#0] is already running with pid 28058.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001356] CHAPTER4 in chapter4.f line 6
(CXdb) trace line 9

#5: trace line, on [#0/*], Enabled, ignore 0/0
        [0x80001390] CHAPTER4 in chapter4.f line 9

(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] hit Tracepoint 5 at [0x80001390] CHAPTER4 in chapter4.f line 9
Process [#0/0] hit Tracepoint 5 at [0x80001390] CHAPTER4 in chapter4.f line 9
Process [#0/0] hit Tracepoint 5 at [0x80001390] CHAPTER4 in chapter4.f line 9
Process [#0/0] hit Tracepoint 5 at [0x80001390] CHAPTER4 in chapter4.f line 9
Process [#0/0] stopped by Bkpt 4, at [0x800013e2] CHAPTER4 in chapter4.f line 15
```

The `run` command in Figure 86 restarts the process. Process execution is stopped by breakpoint 0 at the beginning of the routine `CHAPTER4`. A tracepoint is set at line 9 of the current source file using the `trace line` command. When process execution continues, the tracepoint is triggered 4 times. Each time, CXdb displays a message indicating that the tracepoint was triggered, but process execution continues. Only when breakpoint 4 is reached does process execution stop.

Getting information about tracepoints

You can get information about tracepoints using the `info trace` command. The format of the output is the same as that for the `info break` command.

Figure 87 uses the `info trace` command to display the current setting for tracepoint 5.

Figure 87
Getting information about tracepoints

```
(CXdb) info trace
Event   Enabled Ignore  proc/td     Address      Where
#5         y     0/0     0/*        [0x80001390] CHAPTER4 in chapter4.f line 9
```

Working with watchpoints

Watchpoints are a much different type of eventpoint than breakpoints or tracepoints. Watchpoints are not placed at specific locations in your program, but rather wait for a program variable or memory region to change value. CXdb checks to see if the value has changed after the execution of each statement.

A watchpoint is set only for the current process. Thus, when the process terminates, or when you run your process again, all watchpoints are removed.

Watching a variable's value

You can set a watchpoint to watch for a variable to change value. CXdb automatically determines the size of the variable and sets the watchpoint to monitor the entire address range of the variable.

To watch a variable's contents, use the language operator that returns the address of a variable. In FORTRAN, use the `loc()` function. In C, use the `&` operator.

The `watch` command in Figure 88 sets a watchpoint to monitor an element of `ARRAY`.

Figure 88
Setting a watchpoint on a variable

```
(CXdb) print ARRAY(3,3)
(INTEGER*4) 0
(CXdb) watch loc(ARRAY(3,3))

#6: watch 0x8006dd10..0x8006dd13, on [#0/0], Enabled, ignore 0/0
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8006dd10..0x8006dd13 modified
Process [#0/0] stopped by Watchpoint 6, at [0x800013f6] CHAPTER4 in chapter4.f line 17
(CXdb) print ARRAY(3,3)
(INTEGER*4) 1
(CXdb) print I
(INTEGER*4) 3
(CXdb) print J
(INTEGER*4) 4
```

The `print` command displays the current value of the array element `ARRAY(3,3)`. The `watch` command creates the watchpoint. The `continue` command starts execution. Watchpoint 6 stops execution when the value of `ARRAY(3,3)` changes.

The second `print` command displays the new value of the array. The last two `print` commands display the values of the loop counters, `I` and `J`. With a watchpoint, execution stops on the statement *after* the value changes. Thus the value of `J` is now `4`, rather than `3`.

The `watch` command in Figure 89 sets a watchpoint to monitor the entire address range of `ARRAY`.

Figure 89
Setting a watchpoint on an array

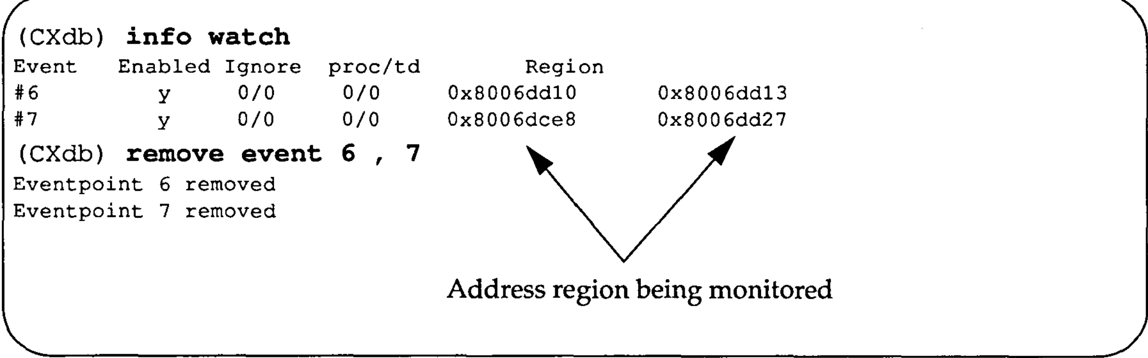
```
(CXdb) watch loc(ARRAY)

#7: watch 0x8006dce8..0x8006dd27, on [#0/0], Enabled, ignore 0/0
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8006dce8..0x8006dd27 modified
Process [#0/0] stopped by Watchpoint 7, at [0x800013ec] CHAPTER4 in chapter4.f line 16
```

In this case, the region is from the hexadecimal address `8006dce8` to `8006dd27`. The `continue` command continues process execution. Execution is stopped by watchpoint 7.

Getting information about watchpoints

With the `info watch` command, you can display the current status of all watchpoints in the process. The `info watch` command displays information about watchpoints in the same way that the `info break` command displays information about breakpoints, as shown in Figure 90.

Figure 90
Getting information about watchpoints



The `info watch` command displays the status of all watchpoints in the current process. The command displays the eventpoint number, enabled setting, ignore count setting, process number and thread numbers, and address region being monitored. The `remove event` command removes the two watchpoints.

Specifying a watchpoint address range

In the previous watchpoint examples, the `loc()` FORTRAN function was used to find the address of the variable to monitor.

There are two other ways to specify an address range in CXdb:

- **Specifying the starting address and ending address**—Two periods are used to separate the two addresses.
- **Specifying the starting address and a count**—A colon is used to separate the count from the starting address. The count is the number of total bytes to monitor, including the starting and ending address.

The source code for the next two examples is shown in Figure 91. Scroll the source code to this section of the program.

Figure 91
Source code modifying variable `PICK`

```
                    file: chapter4.f  process: [#0/0]  Alive
21        SEED = 1
22        PICK = RAN(SEED) * 4 + 1
23        ARRAY(PICK,PICK) = 99
24
25        DO I=1,16
26           PICK = PICK + 4096  <---- Loop modifies 2 low-order bytes of PICK
27        ENDDO
28
29        DO I=1,16
30           PICK = PICK + 65536 <---- Loop modifies 2 high-order bytes of PICK
31        ENDDO
32
33        PICK = PICK + 1  <-------- Statement modifies low-order bytes again
34
35        PRINT *, "SUBROUTINE CHAPTER4 FINISHING"
36        END
```

Figure 91 shows the source code that modifies the variable `PICK`. `PICK` is a 4-byte integer. In a 4-byte integer, the low-order bytes (the last 2 bytes) hold numbers less than 65536. The high-order bytes (the first 2 bytes) hold numbers greater than or equal to 65536. The low-order bytes are modified in the first loop. The high-order bytes are modified in the second loop.

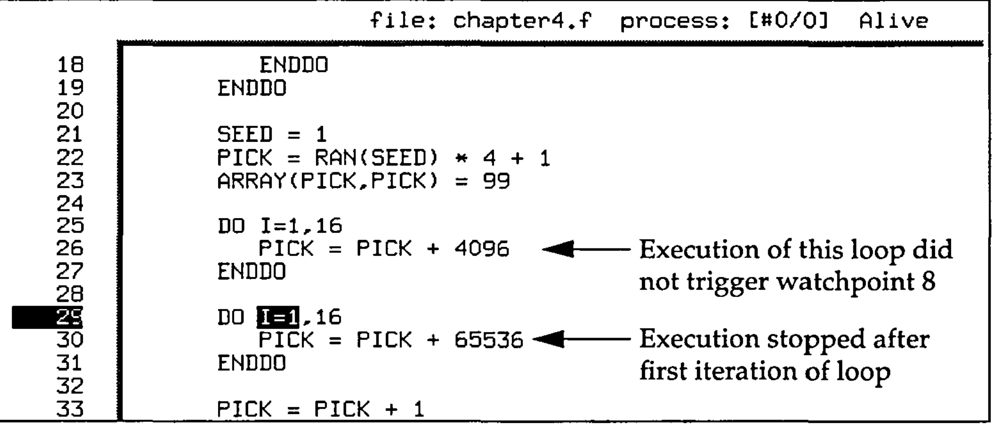
By specifying an address range, watchpoints can monitor a portion of a variable's storage. In the next two examples (Figure 92 and Figure 93), one watchpoint monitors the 2 high-order bytes of `PICK`, while another watchpoint monitors the 2 low-order bytes.

Figure 92
Specifying a starting and ending address for an address range

```
(CXdb) info expression PICK

object type: Fortran identifier
   location: 0x8005a010
       size: 4 bytes
       type: INTEGER*4
      value: 0
       7 liveness ranges:
               Start       End          Location
         1.  0x80001484:0x80001488 - register a5
         2.  0x8000148a:0x8000148c - register a2
         3.  0x80001488:0x80001494 - register a5
         4.  0x800014ac:0x800014b0 - register s0
         5.  0x800014e2:0x800014e8 - register s0
         6.  0x80001510:0x80001514 - register s0
         7.  0x80059000:0x8005a000 - 0x8005a010

(CXdb) watch '8005a010'x .. '8005a011'x

#8: watch 0x8005a010..0x8005a011, on [#0/0], Enabled, ignore 0/0
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8005a010..0x8005a011 modified
Process [#0/0] stopped by Watchpoint 8, at [0x800014d2] CHAPTER4 in chapter4.f line 29
(CXdb) print PICK
(INTEGER*4) 65537
```

Starting address → '8005a010'x

Ending address → '8005a011'x



In Figure 92, a watchpoint is set to monitor the 2 high-order bytes of the variable `PICK`. The `info expression` command displays information about the variable `PICK`, including its value and starting address, `8005a010.` The `info expression` command is covered in detail in Chapter 7.

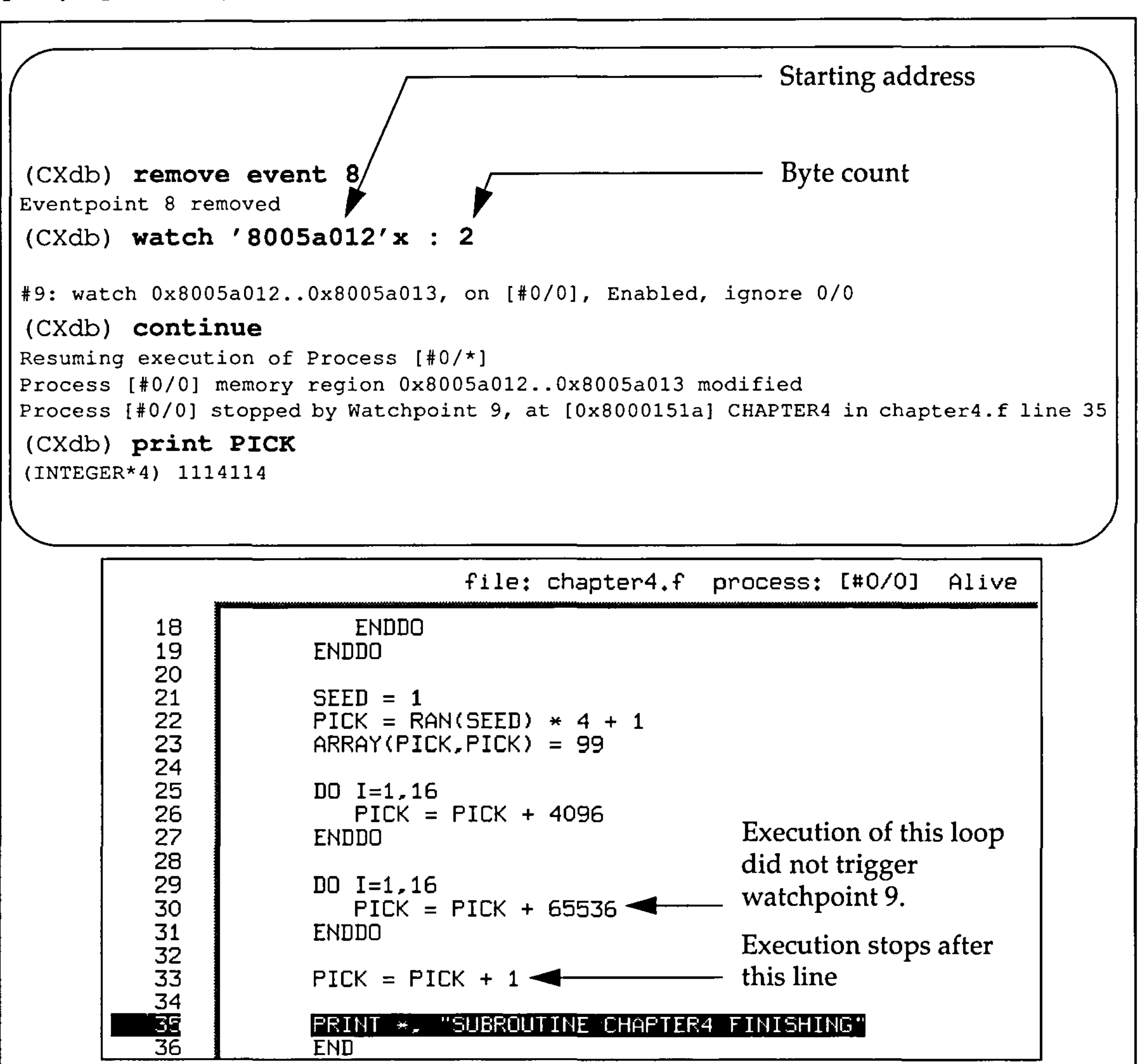
The `watch` command uses FORTRAN syntax to specify the hexadecimal address to monitor. The second number, `8005a011,` is the ending address for the address range.

As long as the decimal valued stored in `PICK` is less than 65536, the watchpoint is not triggered because the 2 high-order bytes being monitored remain unchanged. Thus, even though the value of `PICK` is changing, the watchpoint is not triggered until the value of `PICK` exceeds 65535.

The highlighting in the source window shows that the execution has stopped before the second loop affecting `PICK`. The `print` command in Figure 92 shows that the value of `PICK` has exceeded 65536.

Figure 92 demonstrated how to set a watchpoint to monitor a memory range by specifying a starting address and an ending address. You can also monitor a memory range by specifying a starting address and the number of bytes to monitor, as shown in Figure 93.

Figure 93
Specifying a starting address and a byte count for an address range



In Figure 93, the `remove event` command removes watchpoint 8. A new watchpoint, number 9, is created using the `watch` command. A colon separates the starting address from the byte count. The byte count is the total number of bytes to watch, in this case `2`.

Watchpoint 9 monitors the 2 low-order bytes of the variable PICK. As long as these 2 bytes remain the same, the watchpoint is not triggered. Because each iteration of the loop adds 65536 to PICK, only the two high-order bytes of the variable, 8005a010 and 8005a011, are changed. Watchpoint 9 is not triggered until after line 33.

Working with breakpoints, tracepoints, and watchpoints

During a debugging session, you may create many eventpoints to help control the flow of execution. You have already seen 3 types of info commands that display specific information about the breakpoints, tracepoints, or watchpoints in your program.

The info event command provides a general method for getting information about a particular eventpoint or all eventpoints.

To request information about all eventpoints in the current process, place an asterisk (*) after the info event command, as illustrated in Figure 94.

Figure 94
Getting information about all eventpoints

```
(CXdb) info event *  <---------------- Indicates all eventpoints

#0: break routine, on [#0/*], Enabled, ignore 0/0
       [0x80001356] CHAPTER4 in chapter4.f line 6

#4: break line, on [#0/*], Enabled, ignore 0/0
       [0x800013e2] CHAPTER4 in chapter4.f line 15

#5: trace line, on [#0/*], Enabled, ignore 0/0
       [0x80001390] CHAPTER4 in chapter4.f line 9

#9: watch 0x8005a012..0x8005a013, on [#0/0], Enabled, ignore 0/0
```

You can also get information about specific eventpoints by using the info event command. When specifying multiple eventpoint numbers, you must place a comma between each eventpoint number. This is shown in Figure 95.

Figure 95
Getting information about specific eventpoints

```
(CXdb) info event 4, 5, 9

#4: break line, on [#0/*], Enabled, ignore 0/0
       [0x800013e2] CHAPTER4 in chapter4.f line 15

#5: trace line, on [#0/*], Enabled, ignore 0/0
       [0x80001390] CHAPTER4 in chapter4.f line 9

#9: watch 0x8005a012..0x8005a013, on [#0/0], Enabled, ignore 0/0

(CXdb) remove event *
Eventpoint 0 removed
Eventpoint 4 removed
Eventpoint 5 removed
Eventpoint 9 removed
```

Figure 95 shows the use of the `info event` command to display information about eventpoints 4, 5, and 9. The `remove event` command, which accepts an asterisk to indicate all eventpoints, removes all eventpoints in the current process.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 96.

Figure 96
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Stepping

5

This chapter describes how to step through your program by using the CXdb stepping commands. It also explains the concept of source units and their relation to stepping.

This chapter covers the following commands:

- `break source`
- `finish`
- `info line`
- `info process`
- `info source`
- `next`
- `next over`
- `set step`
- `step`
- `step over`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 97.

Figure 97
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 97 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 98.

Figure 98
Starting the example program

```
(CXdb) break routine CHAPTER5
#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80001658] CHAPTER5 in chapter5.f line 4
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
```

The `break routine` command in Figure 98 sets a breakpoint at the beginning of the routine called `CHAPTER5`, which is a FORTRAN routine used for all examples in this chapter. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

Stepping concepts

Stepping is the incremental execution of code. Rather than executing an entire routine or program at one time, you can execute a smaller portion of code, such as a loop or a statement.

CXdb gives you extensive control over the stepping functions. Three areas of control are:

- Method of stepping
- Number of steps to take at one time
- Size of each step

Stepping methods

There are two methods of stepping in CXdb:

- By assembly language instructions
- By units of source code

This chapter describes how to step by source units. For details about stepping by assembly language instructions, refer to Chapter 14.

Number of steps

With the stepping commands, you can specify a repetition factor that tells CXdb how many times to execute the command. Thus, you can take multiple steps with one command.

Step size (granularity)

When stepping through the source code, you can adjust the step size, or *granularity*. Five levels of granularity are available in CXdb:

- Expression
- Statement
- Block
- Loop
- Routine

By controlling the granularity, you can step a program in small increments, such as an expression, or in larger increments, such as a block or loop. This is in contrast to most other debuggers, which only allow stepping by statements.

Source units

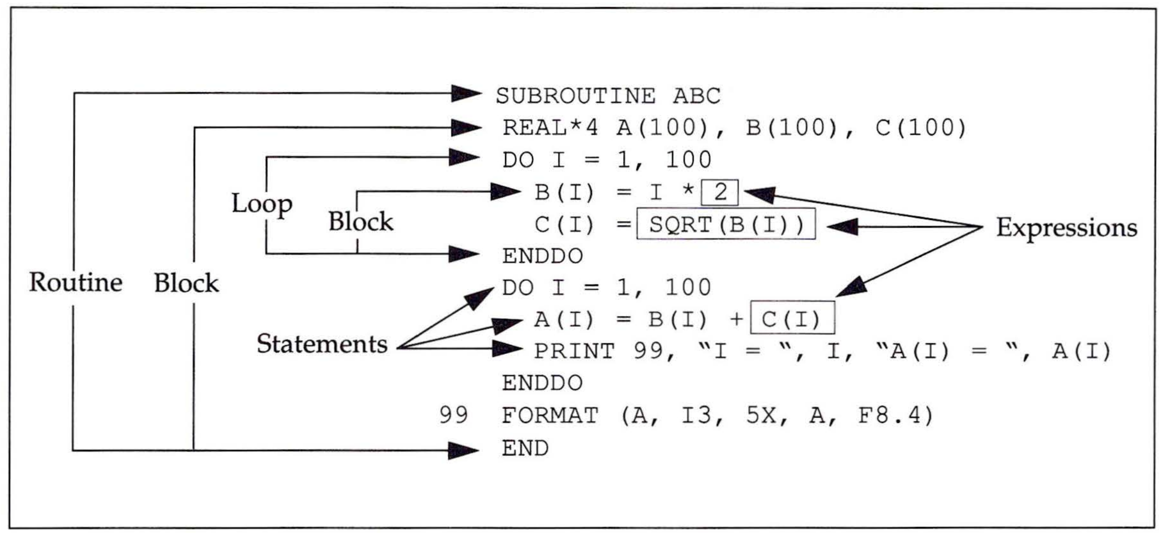
Source units are syntactic elements of source code. There are 5 types of source units that correspond directly to the 5 levels of stepping granularity. They are:

- **Expression**—A combination of one or more constants, operators, and operands. Expressions can be nested within other expressions.
- **Statement**—A combination of one or more expressions.
- **Block**—A set of statements that forms the body of a routine, a loop, or a conditional construct.
- **Loop**—A statement that encompasses a block and the associated looping construct for the block.
- **Routine**—A subroutine or function.

Source units reflect the syntax of the source language. Because of this, the exact form and content of a source unit depend on the source language. For example, the expression `n++` is valid in C language but not in FORTRAN.

Figure 99 shows examples of the different types of source units in a FORTRAN routine.

Figure 99
Example of source units in FORTRAN



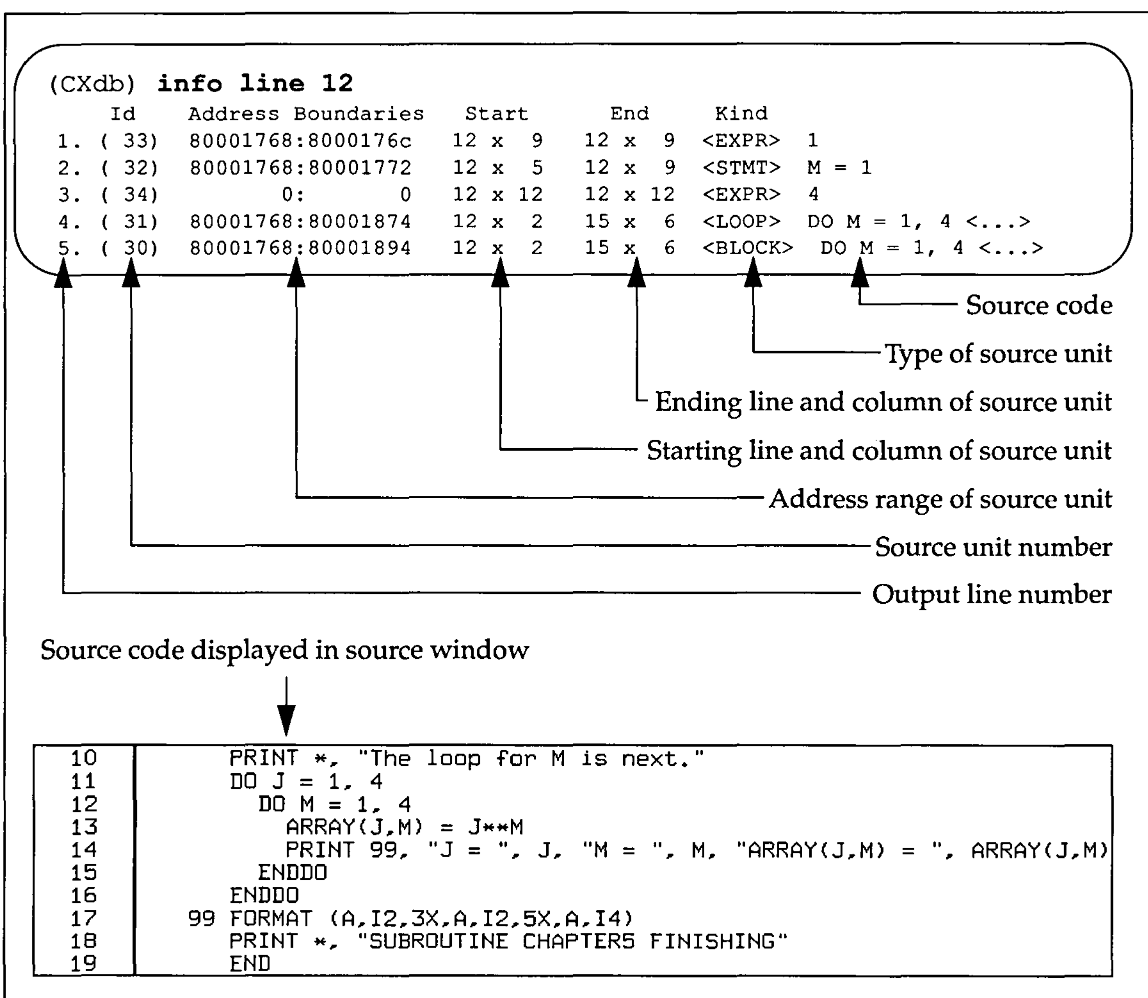
When you compile a program with the `-cxdb` option, the CONVEX FORTRAN or CONVEX C compiler identifies each source unit in the program and assigns it a number. Source unit numbers are sequential and are unique within a given source file.

Several different types of source units can exist at the same location. For example, the FORTRAN code `DO I=1,100` is a statement source unit, but it also is the beginning of a loop source unit, and it contains the expression `I=1`.

Displaying source units

To display the source units in your program, use the `info line` command. Figure 100 illustrates the use of this command with the example program started in Figure 98.

Figure 100
Displaying source units with the `info line` command



The `info line` command in Figure 100 displays the source units for line 12 of the current routine, `CHAPTER5`. Line 12 is the FORTRAN statement `DO M=1,4`. It is the beginning of a loop source unit that ends on line 15. It is also the beginning of a block source unit (the block for `DO J=1,4` on line 11) that ends on line 16. In addition, line 12 contains the expression source units `1` and `4` as well as the statement source unit `M=1`.

If you know the source unit number and want to display the rest of the information about that source unit, use the `info source` command, as shown in Figure 101.

Figure 101
Displaying a source unit with the `info source` command

```
(CXdb) info source 32
 Id    Address Boundaries   Start      End     Kind
( 32)  80001768:80001772   12 x  5   12 x  9  <STMT>  M = 1
```

Using source units in CXdb commands

To use a source unit in a CXdb command, you simply specify the source unit number as a parameter of the command. For example, you can use the `break source` command to set a breakpoint at a particular source unit, as shown in Figure 102.

Figure 102
Setting a breakpoint at a source unit

```
(CXdb) break source 32
#1: break source, on [#0/*], Enabled, ignore 0/0
        [0x80001768] CHAPTER5 in chapter5.f line 12
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80001768] CHAPTER5 in chapter5.f line 12
(CXdb) remove event 1
Eventpoint 1 removed
```

The `break source` command in Figure 102 sets a breakpoint at source unit 32, which is the statement source unit `M=1`. The `continue` command continues execution of the program until it reaches the breakpoint. The `remove event` command removes the breakpoint.

Stepping by source units

There are 5 commands for stepping by source units:

- `step`—Steps to the next source unit in sequence, including source units within called subroutines.
- `next`—Steps to the next source unit in the current routine, stepping past any intervening subroutine calls.
- `finish`—Finishes executing the source unit that contains the current program counter (PC).
- `step over`—Steps from the current source unit to the next source unit in sequence, including source units within called routines.
- `next over`—Steps from the current source unit to the next source unit in the current routine, stepping past any intervening subroutine calls.

Each of these commands allows you to specify the granularity (step size). The default granularity is set as part of the process settings for the current process. If you have not changed the process settings, then the default granularity is statement.

To display the default stepping granularity for the current process, use the `info process` command, as shown in Figure 103.

Figure 103
Displaying the default stepping granularity for the current process

```
(CXdb) info process

status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 6128, state = stopped
     working dir: /usr/lib/cxdb/examples
    default step: statement  <-------------------------  Default stepping granularity
default language: Fortran                                for current process
         threads: 1
  current thread: 0

 thread 0 status: stopped at [0x80001768] CHAPTER5 in chapter5.f line 12

source file search path:
      .
```

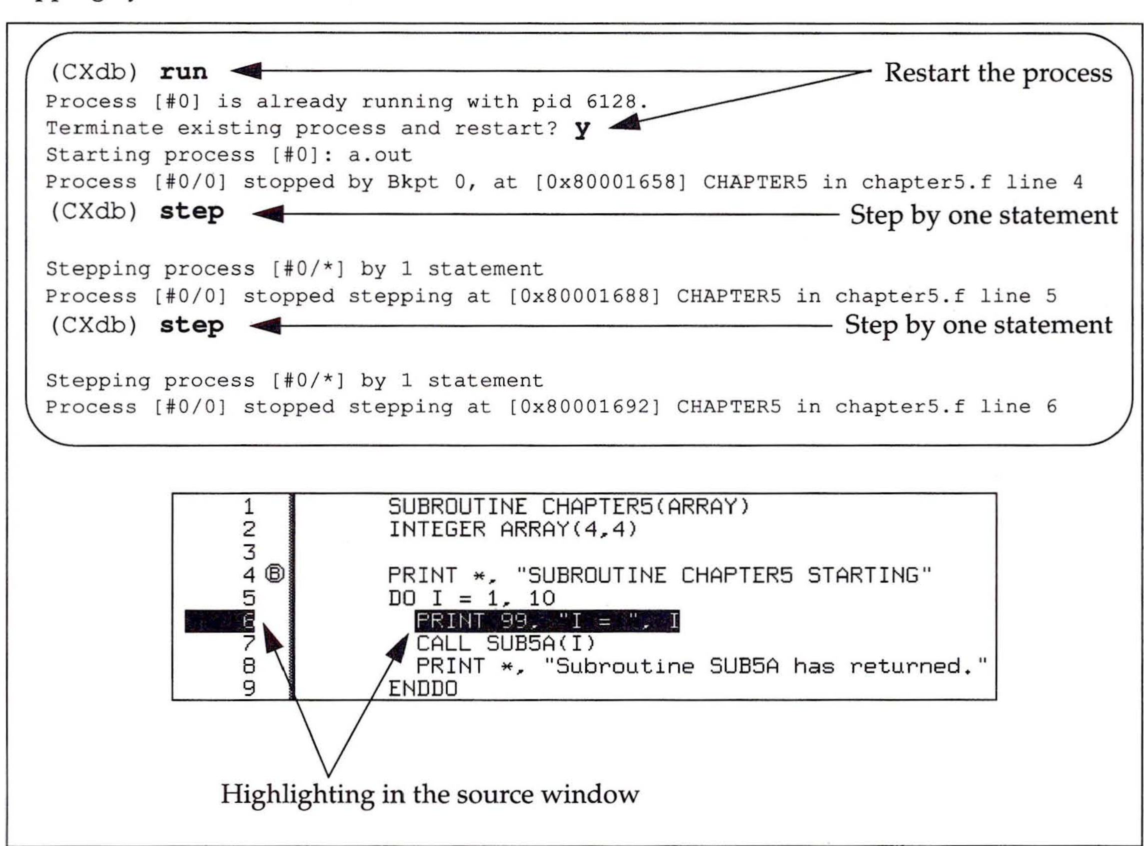
All stepping commands, except `finish,` also allow you to specify a repetition factor for the command. The default repetition factor is 1.

Stepping by statement

The most common form of stepping is to step by statement source units. It is simplest because you are accustomed to viewing your program as a series of source statements, and also because you might already be familiar with stepping by statement from using other debuggers. In addition, because the default stepping granularity is initially set to statement, there is no need to change the default or specify a granularity when stepping by statement.

Figure 104 illustrates stepping by statement.

Figure 104
Stepping by statement

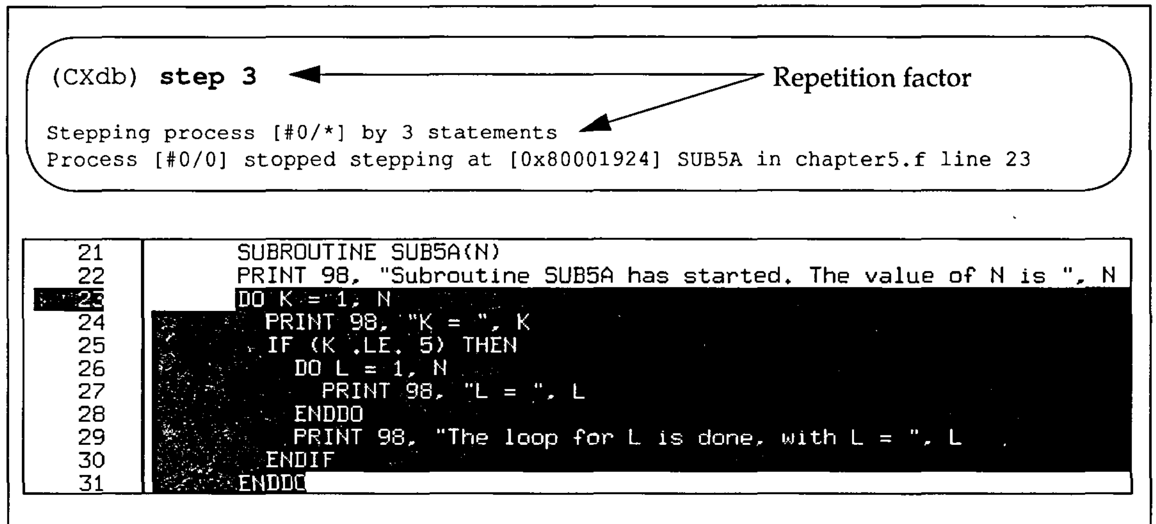


The run command in Figure 104 restarts the process. This is done in the examples so that you can compare the effects of the various stepping commands by always starting them from the same point of execution.

Because the default granularity is statement, the two step commands in Figure 104 each step the process by one statement.

You can specify a repetition factor with stepping commands. This factor tells CXdb how many steps to perform at once. Figure 105 illustrates the use of the repetition factor with the step command.

Figure 105
Stepping by multiple statements with one command



Another command for stepping is the next command. The step and next commands differ in how they handle subroutine calls.

The step command steps to the next statement in execution sequence, regardless of whether that statement is in the current routine or in a called subroutine.

In contrast, the next command steps to the next statement in logical sequence within the same routine. If the next command encounters any intervening subroutine calls, it executes the entire subroutine and steps to the next statement after the call in the parent routine.

Figure 106 illustrates the above differences between the step and next commands.

Figure 106
Difference between the `step` and `next` commands

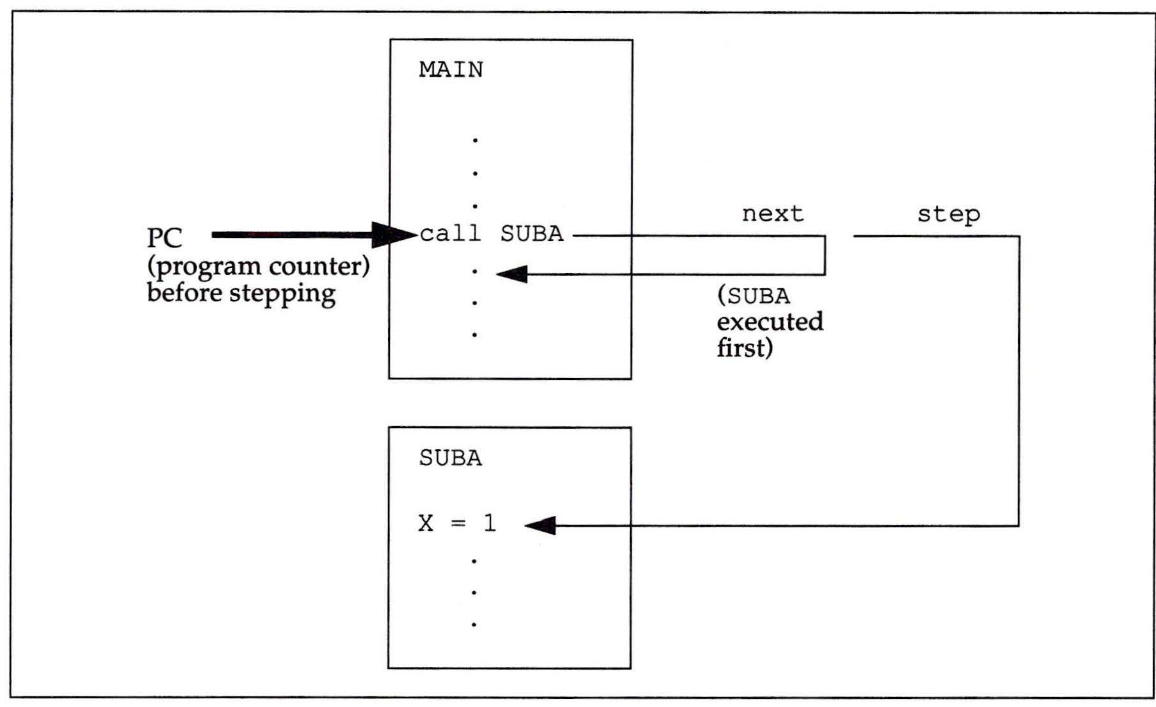


Figure 107 shows how to `next` the process by the default granularity, which in this case is statement.

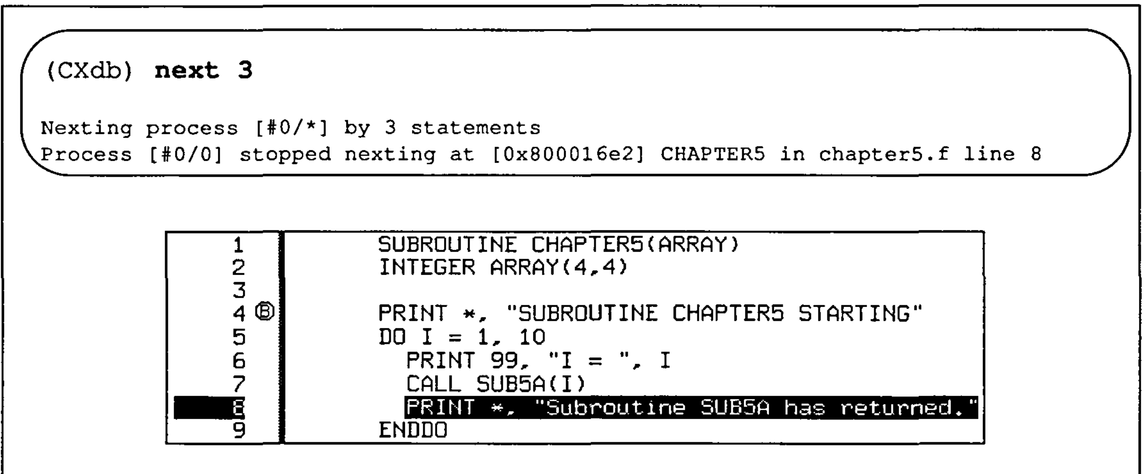
Figure 107
Nexting by statement

```
(CXdb) run
Process [#0] is already running with pid 894.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
(CXdb) next

Nexting process [#0/*] by 1 statement
Process [#0/0] stopped nexting at [0x80001688] CHAPTER5 in chapter5.f line 5
```

```
 1        SUBROUTINE CHAPTER5(ARRAY)
 2        INTEGER ARRAY(4,4)
 3
 4 Ⓑ      PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5        DO I = 1, 10
 6          PRINT 99, "I = ", I
 7          CALL SUB5A(I)
 8          PRINT *, "Subroutine SUB5A has returned."
 9        ENDDO
```

Figure 108 illustrates the use of a repetition factor with the `next` command. At the start of this example, the PC (program counter) points to the beginning of line 5 in the routine `CHAPTER5`. After the `next` command is executed, the PC points to the beginning of line 8 in the same routine. The intervening call to `SUB5A` is executed, but none of the statements in `SUB5A` are counted as one of the 3 specified in the `next 3` command.

Figure 108
Nexting a process by multiple statements with one command



Stepping by other source units

CXdb allows stepping by routine, loop, block, and expression source units in addition to statements. There are 2 ways to specify the stepping granularity:

- Include the granularity as part of the stepping command
- Set the default granularity for the current process

Figure 109 shows an example of stepping by loop source units. The default granularity is still statement in this example, so the granularity of `loop` is specified explicitly with the `step` command.

Figure 109
Stepping by loop

```
(CXdb) run
Process [#0] is already running with pid 917.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
(CXdb) step loop 2

Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x80001924] SUB5A in chapter5.f line 23
```

```
21        SUBROUTINE SUB5A(N)
22        PRINT 98, "Subroutine SUB5A has started. The value of N is ", N
23        DO K = 1, N
24          PRINT 98, "K = ", K
25          IF (K .LE. 5) THEN
26            DO L = 1, N
27              PRINT 98, "L = ", L
28            ENDDO
29            PRINT 98, "The loop for L is done, with L = ", L
30          ENDIF
31        ENDDO
```

The `step` command in Figure 109 steps the process by two loops. When execution stops, the PC points to the beginning of the `DO` loop on line 23 in subroutine `SUB5A,` as indicated by the highlighting in the source window.

You can also specify a granularity with the `next` command, as shown in Figure 110.

Figure 110
Nexting by loop

```
(CXdb) run
Process [#0] is already running with pid 21102.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
(CXdb) next loop 2
Nexting process [#0/*] by 2 loops
Process [#0/0] stopped nexting at [0x8000175e] CHAPTER5 in chapter5.f line 11
```

```
 1          SUBROUTINE CHAPTER5(ARRAY)
 2          INTEGER ARRAY(4,4)
 3
 4 (B)      PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5          DO I = 1, 10
 6            PRINT 99, "I = ", I
 7            CALL SUB5A(I)
 8            PRINT *, "Subroutine SUB5A has returned."
 9          ENDDO
10          PRINT *, "The loop for M is next."
11          DO J = 1, 4
12            DO M = 1, 4
13              ARRAY(J,M) = J**M
14              PRINT 99, "J = ", J, "M = ", M, "ARRAY(J,M) = ", ARRAY(J,M)
15            ENDDO
16          ENDDO
```

The `next` command in Figure 110 nexts the process by two loops. The call to subroutine `SUB5A` in line 7 is executed, but the `next` command does not count any of the loops in this subroutine as one of the two loops specified in the command. When execution stops, the PC points to the beginning of the loop source unit on line 11.

Rather than specifying the granularity with each individual stepping command, you can use the `set step` command to change the default granularity for the current process. Figure 111 illustrates how to do this.

Figure 111
Setting the default stepping granularity for the current process

```
(CXdb) set step block
(CXdb) info process

status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 21999, state = stopped
     working dir: /usr/lib/cxdb/examples
    default step: block
default language: Fortran
         threads: 1
 current thread: 0

 thread 0 status: stopped at [0x8000175e] CHAPTER5 in chapter5.f line 11

source file search path:
            .
```


The `set step` command in Figure 111 changes the default granularity to `block` for the current process. The `info process` command displays the new setting. Any stepping commands that do not explicitly specify a granularity will use this new default. Figure 112 shows an example of stepping with a default granularity of block.

Figure 112
Stepping with a default granularity of block

```
(CXdb) step

Stepping process [#0/*] by 1 block
Process [#0/0] stopped stepping at [0x80001768] CHAPTER5 in chapter5.f line 12
(CXdb) step

Stepping process [#0/*] by 1 block
Process [#0/0] stopped stepping at [0x80001722] CHAPTER5 in chapter5.f line 13
(CXdb) step

Stepping process [#0/*] by 1 block
Process [#0/0] stopped stepping at [0x80001722] CHAPTER5 in chapter5.f line 13
```

```
  1        SUBROUTINE CHAPTER5(ARRAY)
  2        INTEGER ARRAY(4,4)
  3
  4 (B)    PRINT *, "SUBROUTINE CHAPTER5 STARTING"
  5        DO I = 1, 10
  6          PRINT 99, "I = ", I
  7          CALL SUB5A(I)
  8          PRINT *, "Subroutine SUB5A has returned."
  9        ENDDO
 10        PRINT *, "The loop for M is next."
 11        DO J = 1, 4
 12          DO M = 1, 4
 13            ARRAY(J,M) = J**M
 14            PRINT 99, "J = ", J, "M = ", M, "ARRAY(J,M) = ", ARRAY(J,M)
 15          ENDDO
 16        ENDDO
```

Line number where execution stopped

Entire block is highlighted

Each step command in Figure 112 steps the process by one block. In this case, the block is the body of the inner DO loop that starts on line 12. Therefore, stepping by this block is equivalent to stepping through one iteration of the loop.

Figure 112 shows the preferred way to step through a loop by one iteration at a time. Once you are inside the loop, you can step through it by block source units. Each block represents one iteration of the loop.

Effect of default granularity on highlighting

Changing the default granularity affects not only stepping but also the highlighting in the source window. If the default granularity is statement, then the current active statement is highlighted in the source window; if the default granularity is block, then the current active block is highlighted, and so on. Figure 112 and Figure 113 illustrate this point.

Figure 113
Effect on highlighting in the source window when default granularity is block

```
(CXdb) run
Process [#0] is already running with pid 21999.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
```

Line number where execution stopped

Entire block is highlighted

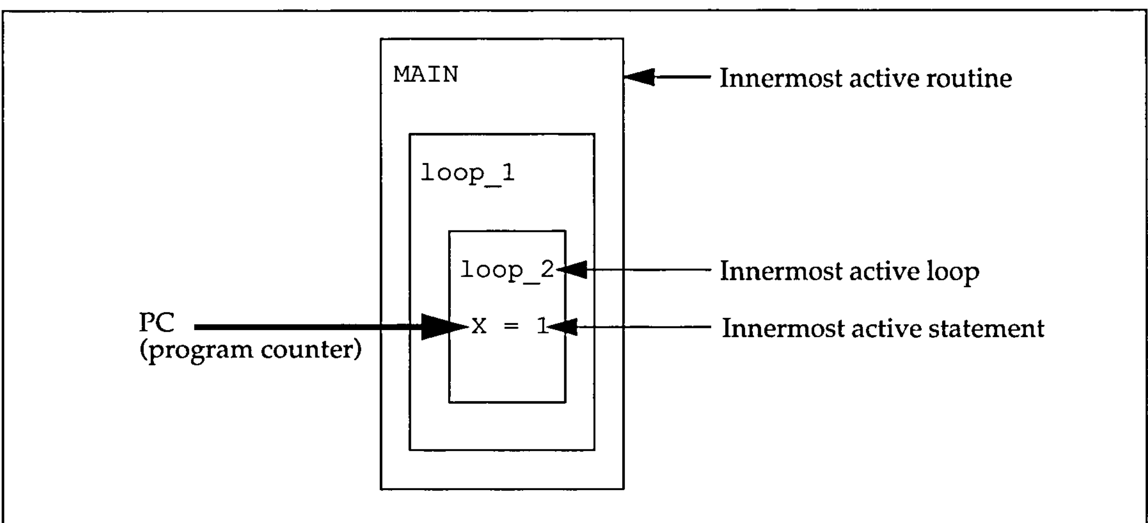
```
 1        SUBROUTINE CHAPTER5(ARRAY)
 2        INTEGER ARRAY(4,4)
 3
 4 (B)    PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5        DO I = 1, 10
 6          PRINT 99, "I = ", I
 7          CALL SUB5A(I)
 8          PRINT *, "Subroutine SUB5A has returned."
 9        ENDDO
10        PRINT *, "The loop for M is next."
11        DO J = 1, 4
12          DO M = 1, 4
13            ARRAY(J,M) = J**M
14            PRINT 99, "J = ", J, "M = ", M, "ARRAY(J,M) = ", ARRAY(J,M)
15          ENDDO
16        ENDDO
17     99 FORMAT (A,I2,3X,A,I2,5X,A,I4)
18        PRINT *, "SUBROUTINE CHAPTER5 FINISHING"
19        END
```

The `run` command in Figure 113 restarts the process. Because the default granularity is set to block, the entire block for subroutine `CHAPTER5` is highlighted in the source window. The breakpoint at line 4 stopped execution of the program. Execution stopped at the beginning of line 4, as indicated by the highlighting of line number 4 at the left side of the source window.

Finishing a source unit

The `finish` command allows you to complete execution of the source unit that contains the current PC (program counter). This source unit is called the *innermost active source unit*. Figure 114 illustrates the concept of innermost active source unit.

Figure 114
Innermost active source unit



As shown in Figure 114, several source units of different granularities can all contain the current PC at the same time. However, the innermost active source unit is the one with the granularity that you specify in the stepping command. If you do not explicitly specify a granularity, then the default stepping granularity is used to determine the innermost active source unit.

For example, if the specified granularity is statement (either explicitly specified or specified by default), then the innermost active source unit in Figure 114 is the statement `X=1`. If the specified granularity is loop, then the innermost active source unit is all of `loop_2`.

The `finish` command steps the process from the innermost active source unit of specified granularity to the next source unit of default granularity. Figure 115 illustrates how to finish execution of a loop.

Figure 115
Finishing a loop

```
(CXdb) step

Stepping process [#0/*] by 1 block
Process [#0/0] stopped stepping at [0x80001692] CHAPTER5 in chapter5.f line 6
(CXdb) set step statement
```

```
 1        SUBROUTINE CHAPTER5(ARRAY)
 2        INTEGER ARRAY(4,4)
 3
 4 (B)    PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5        DO I = 1, 10
 6          PRINT 99, "I = ", I
 7          CALL SUB5A(I)
 8          PRINT *, "Subroutine SUB5A has returned."
 9        ENDDO
10        PRINT *, "The loop for M is next."
11        DO J = 1, 4
```

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped nexting at [0x8000172e] CHAPTER5 in chapter5.f line 10
```

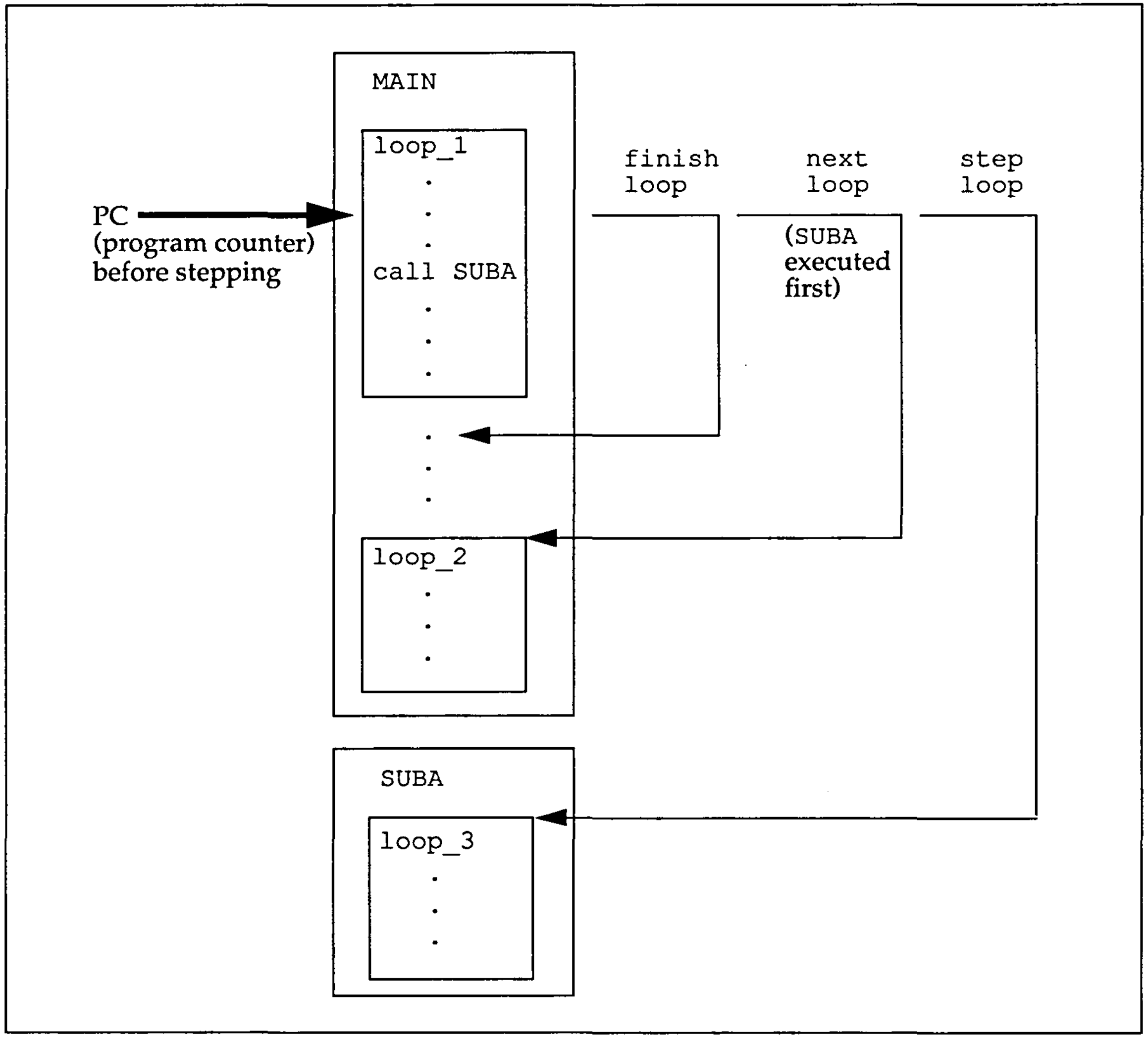
```
 1        SUBROUTINE CHAPTER5(ARRAY)
 2        INTEGER ARRAY(4,4)
 3
 4 (B)    PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5        DO I = 1, 10
 6          PRINT 99, "I = ", I
 7          CALL SUB5A(I)
 8          PRINT *, "Subroutine SUB5A has returned."
 9        ENDDO
10        PRINT *, "The loop for M is next."
11        DO J = 1, 4
```

Because the default granularity was set to `block` in a previous example (see Figure 111), the `step` command in Figure 115 steps the process by one block. The `set step` command then resets the default granularity to `statement.` These steps are done here only to bring the PC to a convenient location to illustrate the `finish` command.

The finish command in Figure 115 steps from the innermost active loop (specified granularity) to the next statement (default granularity) after the loop. At the start of this example, the PC points to line 6, so the innermost active loop is the DO loop that begins on line 5 and ends on line 9. Finishing this loop leaves the PC at the beginning of the next statement after the loop, which is line 10.

Figure 116 illustrates the differences among the step, next, and finish commands.

Figure 116
Comparison of step, next, and finish commands



Stepping over the current source unit

Two other stepping commands are `step over` and `next over`. These commands step from the current source unit of specified granularity to the next source unit of default granularity.

The *current source unit* is the one of specified granularity whose *starting* address is indicated by the value of the current PC (program counter). If there is no current source unit of the specified granularity, then the default granularity is used. For example, the current PC might point to a statement but not to a loop source unit, so you could not step over a loop in that case.

Figure 117 illustrates the concept of current source unit as well as the difference between the `step` and `step over` commands.

Figure 117
Comparison of the `step` and `step over` commands

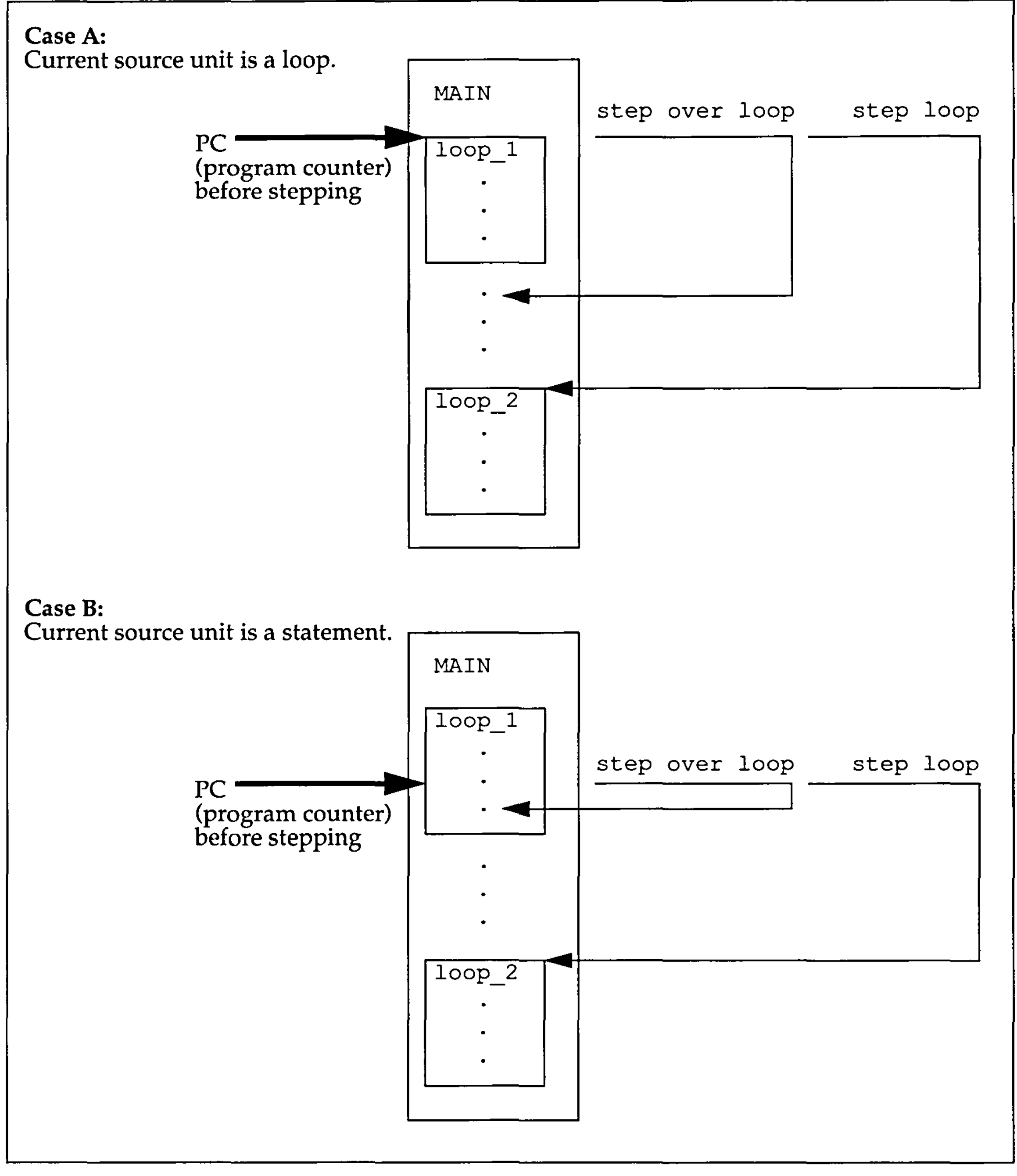


Figure 118 shows an example of stepping over a loop source unit.

Figure 118
Stepping over a loop

```
(CXdb) run
Process [#0] is already running with pid 16483.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
(CXdb) step over loop

Stepping process [#0/*] by 1 statement outside current loop
Process [#0/0] stopped stepping at [0x80001688] CHAPTER5 in chapter5.f line 5
```

```
 1       SUBROUTINE CHAPTER5(ARRAY)
 2       INTEGER ARRAY(4,4)
 3
 4 (B)   PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5       DO I = 1, 10
 6         PRINT 99, "I = ", I
 7         CALL SUB5A(I)
 8         PRINT *, "Subroutine SUB5A has returned."
 9       ENDDO
10       PRINT *, "The loop for M is next."
```

```
(CXdb) step over loop

Stepping process [#0/*] by 1 statement outside current loop
Process [#0/0] stopped stepping at [0x8000172e] CHAPTER5 in chapter5.f line 10
```

```
 1       SUBROUTINE CHAPTER5(ARRAY)
 2       INTEGER ARRAY(4,4)
 3
 4 (B)   PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5       DO I = 1, 10
 6         PRINT 99, "I = ", I
 7         CALL SUB5A(I)
 8         PRINT *, "Subroutine SUB5A has returned."
 9       ENDDO
10       PRINT *, "The loop for M is next."
```

At the beginning of the example in Figure 118, the PC points to line 4. The current source unit at that point is a statement. Therefore, the first `step over loop` command has the same effect as a single `step statement` command because there is no current loop to step over. After stepping, the PC points to the beginning of line 5.

When the PC points to line 5, the current source unit is the `DO` loop that begins on line 5 and ends on line 9. Issuing the second `step over loop` command at this point causes execution of the entire `DO` loop. Execution of the process stops at the next statement (default granularity) after the loop, which is line 10.

The `next over` command differs from the `step over` command in the handling of subroutine calls, similar to the way the `next` command differs from the `step` command. The `step over` command looks for the next source unit in either the current routine or in any called subroutine. However, the `next over` command steps past called subroutines and looks for the next source unit within the current routine only.

Stepping to another routine

All of the stepping commands except `finish` can carry process execution beyond the end of the current routine. Figure 119 illustrates how to step to the next routine.

Figure 119
Stepping to the next routine

```
(CXdb) run
Process [#0] is already running with pid 25609.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001658] CHAPTER5 in chapter5.f line 4
```

```
 1        SUBROUTINE CHAPTER5(ARRAY)
 2        INTEGER ARRAY(4,4)
 3
 4 (B)    PRINT *, "SUBROUTINE CHAPTER5 STARTING"
 5        DO I = 1, 10
 6          PRINT 99, "I = ", I
 7          CALL SUB5A(I)
 8          PRINT *, "Subroutine SUB5A has returned."
 9        ENDDO
```

```
(CXdb) step routine
Stepping process [#0/*] by 1 routine
Process [#0/0] stopped stepping at [0x800018ce] SUB5A in chapter5.f line 21
```

```
21        SUBROUTINE SUB5A(N)
22        PRINT 98, "Subroutine SUB5A has started. The value of N is ", N
23        DO K = 1, N
24          PRINT 98, "K = ", K
25          IF (K .LE. 5) THEN
26            DO L = 1, N
27              PRINT 98, "L = ", L
28            ENDDO
29            PRINT 98, "The loop for L is done, with L = ", L
30          ENDIF
31        ENDDO
32        PRINT 98, "Subroutine SUB5A is done. The value of K is ", K
33        RETURN
34     98 FORMAT (A,I2)
35        END
```

In Figure 119, the `run` command starts the process again. The breakpoint stops execution at the beginning of subroutine `CHAPTER5`. The next routine is `SUB5A`, which is called from line 7 of `CHAPTER5`. Therefore, the `step routine` command continues execution to the beginning of `SUB5A`.

Stepping to the end of the current routine

The `finish routine` command always stops at the end of the current routine. This feature allows you to check the state of the routine before exiting from it. Figure 120 illustrates this feature.

Figure 120
Stopping at the end of a routine

```
(CXdb) finish routine
Finishing innermost routine in Process [#0/*]
Process [#0/0] stopped stepping at [0x80001ab4] SUB5A in chapter5.f line 33
```

```
21        SUBROUTINE SUB5A(N)
22        PRINT 98, "Subroutine SUB5A has started. The value of N is ", N
23        DO K = 1, N
24          PRINT 98, "K = ", K
25          IF (K .LE. 5) THEN
26            DO L = 1, N
27              PRINT 98, "L = ", L
28            ENDDO
29            PRINT 98, "The loop for L is done, with L = ", L
30          ENDIF
31        ENDDO
32        PRINT 98, "Subroutine SUB5A is done. The value of K is ", K
33        RETURN
34     98 FORMAT (A,I2)
35        END
```

```
(CXdb) step

Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x800016e2] CHAPTER5 in chapter5.f line 8
```

```
1         SUBROUTINE CHAPTER5(ARRAY)
2         INTEGER ARRAY(4,4)
3
4 (B)     PRINT *, "SUBROUTINE CHAPTER5 STARTING"
5         DO I = 1, 10
6           PRINT 99, "I = ", I
7           CALL SUB5A(I)
8           PRINT *, "Subroutine SUB5A has returned."
9         ENDDO
```

In Figure 120, the `finish routine` command stops at the `RETURN` statement on line 33 of the current routine, which is `SUB5A`. The `step` command then continues execution to the next statement in the calling routine, which is line 8 in `CHAPTER5`.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 121.

Figure 121
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Eventpoints, handlers, and signals

6

This chapter introduces the general eventpoints. There are 10 types of eventpoints, including breakpoints, tracepoints, and watchpoints. Relational eventpoints are also discussed. In addition, commands that affect settings for eventpoints are covered, as well as eventpoint handlers.

This chapter also explains how to use the signal handling capabilities of CXdb.

The commands discussed in this chapter are:

- `clear default handler`
- `clear handler`
- `clear typehandler`
- `disable event`
- `enable event`
- `event reached line`
- `event relation`
- `event signal`
- `info eventtype`
- `info signal`
- `remove eventtype`
- `set ignore`
- `set default handler`
- `set handler`
- `set signal`
- `set typehandler`
- `signal process`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 122.

Figure 122
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 122 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 123.

Figure 123
Starting the example program

```
(CXdb) break routine CHAPTER6

#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80001fce] CHAPTER6 in chapter6.f line 5
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001fce] CHAPTER6 in chapter6.f line 5
```

The `break routine` command in Figure 123 sets a breakpoint at the beginning of the routine called `CHAPTER6`, which is a FORTRAN routine used for all examples in this chapter. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

General eventpoints

Breakpoints, tracepoints, and watchpoints are all part of the general class eventpoints. These predefined types are commonly used and thus have special commands for them. In addition to these predefined types, you can create 7 other types of eventpoints.

The following is a list of all types of eventpoints (listed by their eventpoint type name):

- **break**—Breakpoints.
- **exec**—Waits for your process to perform an `exec()` system call.
- **join**—Waits for the termination of a thread by means of a `join`, `cfork`, or `wfork` instruction.
- **modify**—Waits for a portion of memory to change value. This is similar to a watchpoint.
- **reached**—Waits for a particular location in your program, such as a line, source unit, routine, or instruction, to be reached. This is similar to a breakpoint or tracepoint.
- **relation**—Waits for a relational expression to evaluate to true.
- **signal**—Waits for the sending of a particular signal to your process.
- **spawn**—Waits for the creation of a thread via the `spawn` or `pfork` instruction.
- **trace**—Tracepoints.
- **watch**—Watchpoints.

The modify eventpoints function just like watchpoints. The reached eventpoints function just like breakpoints. The relation and signal eventpoint types are covered in depth in this chapter. Spawn eventpoints are discussed in Chapter 16, Section "Spawn eventpoints," and join eventpoints are discussed in Chapter 16, Section "Join eventpoints."

Working with eventpoints

Chapter 4 detailed how to set and remove breakpoints, tracepoints and watchpoints. This section introduces a more general form of the breakpoint, the reached eventpoint. It also explains three other actions dealing with eventpoints:

- **Disabling and enabling**—Activating or deactivating an eventpoint without removing it.
- **Setting ignore counts**—Skipping an eventpoint a given number of times.
- **Manipulating multiple eventpoints**—Working with multiple eventpoints at the same location.

Using reached eventpoints

Reached eventpoints are set just like breakpoints. You can set a reached eventpoint at an instruction, a line, a routine, or a source unit. If process execution reaches the location of a reached eventpoint, the eventpoint is triggered and the commands of its handler are executed. The default handler for reached eventpoints stops the process and displays a message, just like breakpoints. Figure 124 shows the `event reached line` command.

Figure 124
Setting a reached eventpoint at a line

```
(CXdb) event reached line 10

#1: reached line, on [#0/*], Enabled, ignore 0/0
        [0x80002042] CHAPTER6 in chapter6.f line 10
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Eventpoint 1, at [0x80002042] CHAPTER6 in chapter6.f
```

The `event reached line` command in Figure 124 sets an eventpoint on line 10. The command output displays the eventpoint number, process number and thread numbers, enable setting, ignore count setting, hexadecimal address, and symbolic location for the new eventpoint.

The `continue` command in Figure 124 continues the process. Execution is stopped by the eventpoint at line 10.

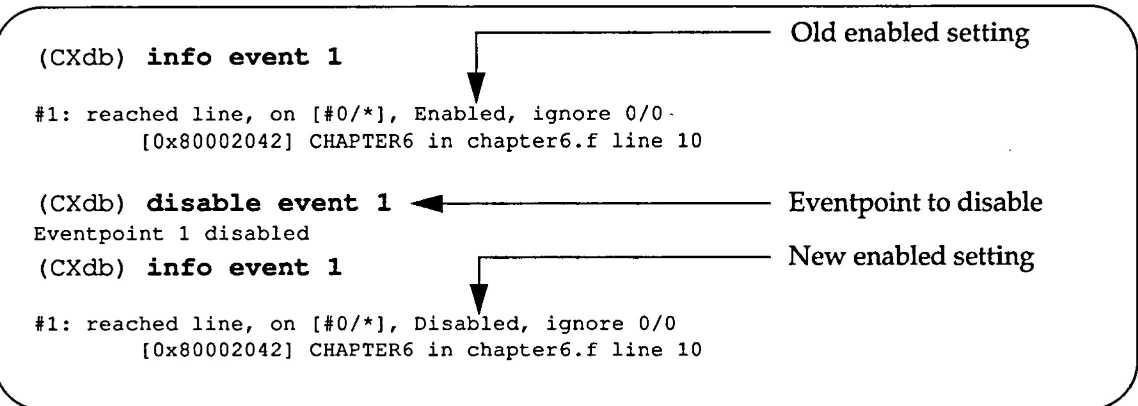
Disabling and enabling eventpoints

When you create an eventpoint, it is initially enabled. Only enabled eventpoints can be triggered.

You can disable an eventpoint so that it cannot be triggered. However, the disabled eventpoint is not removed. Disabling an eventpoint allows you to deactivate it when necessary, then enable it again later. This saves you from having to repeatedly set and remove eventpoints at the same location.

Eventpoint 1 at line 10 in the program is currently enabled. You can disable it with the `disable event` command, as shown in Figure 125.

Figure 125
Disabling an eventpoint



The `disable event` command in Figure 125 disables eventpoint 1. The `info event` command displays the current settings for eventpoint 1. The enabled setting for eventpoint 1 is changed from `Enabled` to `Disabled`.

At this point, eventpoint 1 cannot be triggered. If you step process execution past line 10, the eventpoint is not triggered. This is demonstrated in Figure 126.

Figure 126
Stepping past a disabled eventpoint

```
(CXdb) step loop

Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80002038] CHAPTER6 in chapter6.f line 9
(CXdb) step 3

Stepping process [#0/*] by 3 statements
Process [#0/0] stopped stepping at [0x80002042] CHAPTER6 in chapter6.f line 10
```

The first `step` command in Figure 126 steps the process by 1 loop. The process completes the current loop and then stops on line 9 at the start of the loop again, without eventpoint 1 being triggered. The second `step` command steps the process through 3 more iterations of the loop, without eventpoint 1 being triggered.

You can enable an eventpoint using the `enable event` command, as shown in Figure 127.

Figure 127
Enabling a disabled eventpoint

```
(CXdb) enable event 1
Eventpoint 1 enabled
(CXdb) continue

Resuming execution of Process [#0/*]
Process [#0/0] stopped by Eventpoint 1, at [0x80002042] CHAPTER6 in chapter6.f
```

The `enable event` command enables eventpoint 1. Process execution is continued, and when line 10 is reached, the eventpoint is triggered.

Setting ignore counts

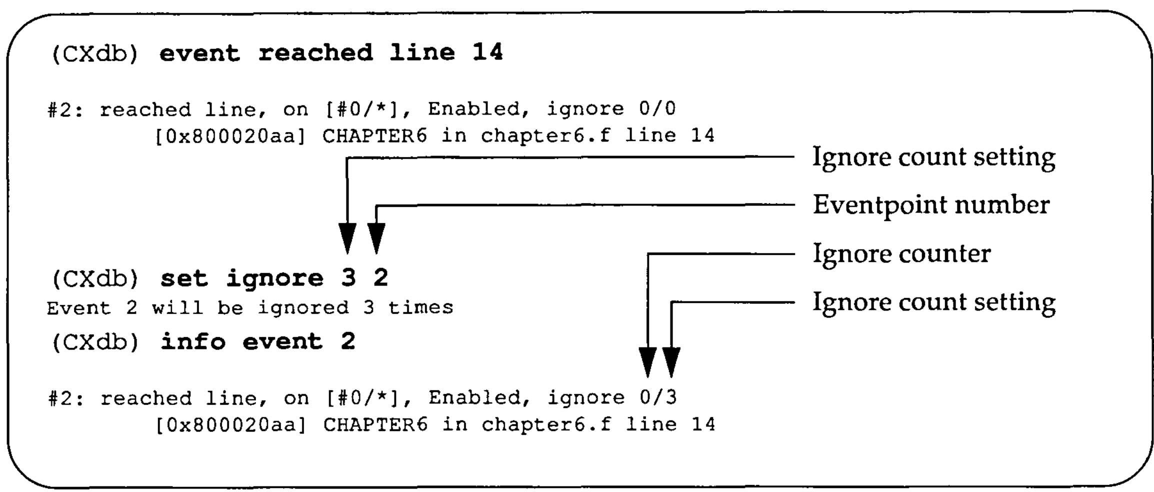
When you want an eventpoint to be disabled for a specific length of time, you can set an ignore count on the eventpoint. An ignore count is the number of times an eventpoint is skipped before it is triggered.

When an eventpoint is reached, CXdb checks to see if the eventpoint has an ignore count. If it does, the ignore counter is incremented by 1 and process execution continues past the eventpoint. If the eventpoint does not have an ignore count, the eventpoint is triggered.

Once the ignore counter of an eventpoint is equal to its ignore count, the ignore counter is reset to 0, and the next time the eventpoint is reached it is triggered.

You can set an ignore count on an eventpoint by using the `set ignore` command. You must create the eventpoint before setting its ignore count. Figure 128 illustrates an ignore count being given to a new eventpoint.

Figure 128
Setting an ignore count



The `event reached line` command creates a reached eventpoint (eventpoint 2). The `set ignore` command sets an ignore count of 3 for the new eventpoint. The `info event` command displays the ignore count setting for eventpoint 2.

The effect of the ignore count is shown in Figure 129.

Figure 129
Triggering an eventpoint that has an ignore count

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Eventpoint 2, at [0x800020aa] CHAPTER6 in chapter6.f
(CXdb) print J
(INTEGER *4) 4                                          Ignore count reset
(CXdb) info event 2

#2: reached line, on [#0/*], Enabled, ignore 0/0
      [0x800020aa] CHAPTER6 in chapter6.f line 14
```

In Figure 129, the `continue` command continues process execution. The first 3 times line 14 is executed, eventpoint 2 is skipped. On the fourth iteration of the loop, eventpoint 2 is triggered. The `print` command verifies that the loop is in its fourth iteration.

The `info event` command shows that the ignore count for eventpoint 2 has been set back to 0. Each time an ignore count is completed, the ignore counter and setting are reset to 0.

You can force an ignore counter to 0 by setting an ignore count of 0 for the eventpoint. Thus, if you set an ignore count for an eventpoint, and then later want to trigger the eventpoint before the ignore count is completed, you can cancel the ignore count.

Manipulating multiple eventpoints

You can place multiple eventpoints at the same location. Each eventpoint can perform a unique action at the location. By disabling eventpoints and using ignore counts, you can control which eventpoint is triggered.

When execution stops at a location with multiple eventpoints, the highest numbered enabled eventpoint is reached.

If the eventpoint does not have an ignore count, it is triggered and the commands of the handler are executed. If it does have an ignore count, its ignore counter is incremented, and CXdb reaches the next enabled eventpoint. The eventpoint is checked for an ignore count.

This procedure repeats until either an eventpoint triggers, or there are no more enabled eventpoints at the location.

Figure 130 and Figure 131 demonstrate how CXdb handles multiple eventpoints at the same location.

Figure 130
Setting multiple eventpoints at the same location

```
(CXdb) event reached line 14
#3: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800020aa] CHAPTER6 in chapter6.f line 14

INFO: 104
Eventpoint 2 also has a breakpoint at address 0x800020aa.
(CXdb) event reached line 14
#4: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800020aa] CHAPTER6 in chapter6.f line 14

INFO: 104
Eventpoint 3 also has a breakpoint at address 0x800020aa.
(CXdb) disable event 4
Eventpoint 4 disabled
(CXdb) set ignore 1 3
Event 3 will be ignored 1 time
```

Two more eventpoints are set using the `event reached line` command. CXdb displays a message indicating that other eventpoints exist at the same location. The `disable event` command disables eventpoint 4, which is the highest numbered eventpoint. The `set ignore` command sets an ignore count of 1 for eventpoint 3. Thus, eventpoint 3 is the highest numbered enabled eventpoint at line 14. However, because it has an ignore count, eventpoint 2 will be triggered. This is demonstrated in Figure 131.

Figure 131
Continuing process execution to trigger different eventpoints

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Eventpoint 2, at [0x800020aa] CHAPTER6 in chapter6.f
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Eventpoint 3, at [0x800020aa] CHAPTER6 in chapter6.f
(CXdb) enable event 4
Eventpoint 4 enabled
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Eventpoint 4, at [0x800020aa] CHAPTER6 in chapter6.f
```

The first `continue` command continues process execution. Eventpoint 2 stops execution. Process execution is again continued with the second `continue` command. This time, execution is stopped by eventpoint 3 because eventpoint 3 no longer has an ignore count and has a higher eventpoint number than eventpoint 2.

The `enable event` command enables eventpoint 4. Process execution continues with the third `continue` command. Because eventpoint 4 is now the highest-numbered enabled eventpoint, it stops the process.

You can also simulate ignore counts by including a count with the `continue` command. The `continue` command by itself has an implied count of 1. When you specify a count for the `continue` command, you are specifying the number of eventpoints that must trigger before execution stops. Thus, when the continue command is used by itself, it stops after 1 eventpoint triggers.

When an eventpoint triggers in the middle of the `continue` command, the eventpoint handler does not execute. Instead, CXdb decrements the specified count and continues execution. This is illustrated in Figure 132.

Figure 132
Specifying a count with the `continue` command

```
(CXdb) continue 2
Resuming execution of Process [#0/*]

Process [#0] exited normally.
```

The `continue` command has a count of 2. Had the `continue` command not been given a count, eventpoint 4 would have stopped the process. But because the count is 2, two eventpoints must trigger before execution stops. Eventpoint 4 is skipped, and no other eventpoints exist in the process, so execution finishes.

Eventpoint types

Eventpoints are separated into types to enable you to perform operations on a set of eventpoints. There are 10 different types of eventpoints. If you want to work with all the eventpoints of a particular type, you can use a set of commands that operate on eventpoint types rather than individual eventpoints.

You can request information on a type of eventpoint by using the `info eventtype` command, as shown in Figure 133.

Figure 133
Getting information on a type of eventpoint

```
(CXdb) info eventtype reached

Status of eventpoints of type Reached:

#4: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800020aa] CHAPTER6 in chapter6.f line 14

#3: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800020aa] CHAPTER6 in chapter6.f line 14

#2: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800020aa] CHAPTER6 in chapter6.f line 14

#1: reached line, on [#0/*], Enabled, ignore 0/0
        [0x80002042] CHAPTER6 in chapter6.f line 10
```

The `info eventtype` command displays information on all eventpoints of type reached.

The `remove eventtype` command removes all eventpoints of the specified type, as illustrated in Figure 134.

Figure 134
Removing all eventpoints of a particular type

```
(CXdb) remove eventtype reached
Eventpoint 4 removed
Eventpoint 3 removed
Eventpoint 2 removed
Eventpoint 1 removed
```

The `remove eventtype` command in removes all reached eventpoints. In this case, eventpoints 1, 2, 3 and 4 are removed.

Eventpoint handlers

An eventpoint handler is a series of CXdb commands that execute when an eventpoint is triggered. There is a default handler for all eventpoints. The default handler performs the appropriate action for the type of eventpoint. That is, it stops process execution for breakpoints, and allows process execution to continue for tracepoints. The default handler also displays a message indicating which eventpoint was triggered.

You can override the default handler in any of three ways:

- Define a new handler for an individual eventpoint.
- Define a new type handler for all eventpoints of a particular type, such as tracepoints.
- Define a new default handler that applies to all eventpoints of every type.

When an eventpoint is triggered, CXdb looks for the eventpoint handler in the order listed above (that is, individual handler first, type handler next, then default handler). It executes the commands of the first handler that it finds.

Specifying a handler for an eventpoint

All CXdb commands that create eventpoints can specify a handler for the eventpoint. The handler must be enclosed in braces. All commands within the handler must end with a semicolon.

Figure 135 uses the `break line` command to create a new breakpoint with a handler.

Figure 135
Setting breakpoints with eventpoint handlers

```
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80001f6ce] CHAPTER6 in chapter6.f line 5
(CXdb) break line 19 {echo/n "Stopped by breakpoint: ";print $self;}
#5: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002126] CHAPTER6 in chapter6.f line 19
    {
        echo/n "Stopped by breakpoint: ";
        print $self;
    }
(CXdb) continue
Resuming execution of Process [#0/*]
Stopped by breakpoint: (INTEGER*4) 5
```

The `run` command in Figure 135 creates a new process. Process execution is stopped at the beginning of the `CHAPTER6` subroutine. The `break line` command creates a breakpoint with a handler. The `echo` command in the handler displays a message, and the `/n` flag suppresses the newline after the message.

Note

The `echo` command must be used in eventpoint handlers to print string constants, rather than the `print` command, because the `print` command alters process memory to store strings.

The `print` command in the handler prints out the value of the debugger variable `$self`, which is a predefined debugger variable that stores the eventpoint number of the last eventpoint triggered. When used in an eventpoint handler, it always contains the eventpoint number of the current eventpoint being triggered.

The `continue` command resumes process execution until it is stopped by breakpoint 5. Note that the handler for breakpoint 5 overrides the default handler for breakpoints.

You can also create an eventpoint handler for an existing eventpoint using the `set handler` command, as shown in Figure 136.

Figure 136
Specifying a handler for an existing eventpoint

```
(CXdb) break line 24

#6: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002172] CHAPTER6 in chapter6.f line 24
(CXdb) evaluate $count=0

(CXdb) set handler 6 {eval $count=$count+1;print $count;resume;}
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
(INTEGER*4) 1
(INTEGER*4) 2
(INTEGER*4) 3
(INTEGER*4) 4
(INTEGER*4) 5
(INTEGER*4) 6
(INTEGER*4) 7
(INTEGER*4) 8
(INTEGER*4) 9
(INTEGER*4) 10
(INTEGER*4) 11
(INTEGER*4) 12
(INTEGER*4) 13
(INTEGER*4) 14
(INTEGER*4) 15
(INTEGER*4) 16
(INTEGER*4) 17
(INTEGER*4) 18
(INTEGER*4) 19
(INTEGER*4) 20
(INTEGER*4) 21
(INTEGER*4) 22
(INTEGER*4) 23
(INTEGER*4) 24
(INTEGER*4) 25
(INTEGER*4) 26
(INTEGER*4) 27
(INTEGER*4) 28
(INTEGER*4) 29
(INTEGER*4) 30
Process [#0/0] stopped nexting at [0x80002208] CHAPTER6 in chapter6.f line 31
```

Eventpoint to receive handler

The `break line` command in Figure 136 sets a breakpoint on line 24. The `evaluate $count` command creates a new debugger variable named `$count` and sets its value to zero. This statement initializes `$count` before it is used in the eventpoint handler.

You can create debugger variables to use in many CXdb commands. Debugger variables are used in CXdb to store the results of language expressions. You have already seen one predefined debugger variable, `$self`. For more information about debugger variables, refer to the "debugger variables" reference page in the "Concepts" section of the *CONVEX CXdb Reference.*

The `set handler` command gives the new breakpoint its own handler that increments the debugger variable `$count` by 1. The `print` command prints the new value of `$count` each time the breakpoint is triggered. The `resume` command resumes process execution.

Note

Within an eventpoint handler, only the `resume` command can be used to continue or restart process execution. The `resume` command always continues process execution in the same manner as before the eventpoint was triggered.

In this case, the `finish loop` command started process execution. The `resume` command continues the `finish loop` command. Thus, process execution stops at the end of the loop rather than when another eventpoint is triggered or when the process terminates.

You can use the debugger variable `$self` to manipulate the eventpoint being triggered. For example, you can use the debugger variable `$self` to remove the eventpoint, as shown in Figure 137.

Figure 137
Setting an eventpoint that removes itself

```
(CXdb) break line 32 {evaluate NEXT=1;remove event $self;}

#7: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002212] CHAPTER6 in chapter6.f line 32
    {
        evaluate NEXT=1;
        remove event $self;
    }
(CXdb) continue
Resuming execution of Process [#0/*]
Eventpoint 7 removed
```

The `break line` command in Figure 137 creates an eventpoint that removes itself. The `evaluate` command sets the program variable `NEXT` equal to `1`. The `remove event` command removes the current eventpoint, in this case, eventpoint 7. By including the above `remove event` command within an eventpoint handler, you can stop the process at a specified location without leaving the eventpoint at that location.

As illustrated in Figure 138, you can use the `if` statement in eventpoint handlers to test a relational expression and control the flow of execution. The `if` statement uses a C-like syntax and can have a then-clause and an else-clause.

Figure 138
Setting an eventpoint handler with an `if` statement

```
(CXdb) break line 36 {if (ARRAY(I,J) .GT. ARRAY(J,I)) echo "GREATER
THAN"; else {echo "LESS, CONTINUING"; resume;}}

#8: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002296] CHAPTER6 in chapter6.f line 36
    {
        if (ARRAY(I,J) .GT. ARRAY(J,I)) echo "GREATER THAN"; else {echo "LESS, ...
    }
(CXdb) continue
Resuming execution of Process [#0/*]
LESS, CONTINUING
LESS, CONTINUING
LESS, CONTINUING
LESS, CONTINUING
GREATER THAN
(CXdb) print I
(INTEGER*4) 2
(CXdb) print J
(INTEGER*4) 1
```

The `break line` command in Figure 138 sets a breakpoint at line 36. The handler tests to see if the current value of `ARRAY(I,J)` is greater than the value of `ARRAY(J,I)`.

The `continue` command resumes process execution. The else clause of the handler is executed 4 times before the conditional expression evaluates to true.

The `print` commands show that execution stopped at the calculation of `ARRAY(2,1)`.

A handler can be removed by using the `clear handler` command followed by the eventpoint number. After the handler is cleared, the eventpoint returns to using the default handler when it is triggered. This is demonstrated in Figure 139.

Figure 139
Removing a handler from an eventpoint

```
(CXdb) clear handler 8
(CXdb) info event 8

#8: break line, on [#0/*], Enabled, ignore 0/0
       [0x80002296] CHAPTER6 in chapter6.f line 36
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 8, at [0x80002296] CHAPTER6 in chapter6.f line 36
```

Figure 139 removes the handler of the breakpoint with the `clear handler` command. The `info event` command shows eventpoint 8 no longer has a handler. The `continue` command continues execution. The breakpoint at line 36 stops the process using the default handler.

Specifying a handler for an eventpoint type

As discussed earlier, there are 10 types of eventpoints. In addition to being able to give a handler to any individual eventpoint, you can create or change the handler for any type of eventpoint.

The `set typehandler` command specifies a handler for a type of eventpoint. Figure 140 uses the `set typehandler` command to change the handler for all breakpoints.

Figure 140
Changing the handler for all breakpoints

```
(CXdb) set typehandler break {echo/n "Stopped by Bp: ";print $self;}
(CXdb) continue
Resuming execution of Process [#0/*]
Stopped by Bp: (INTEGER*4) 8
(CXdb) info eventtype break ◄──────── Eventpoint type to receive handler
Status of eventpoints of type Breakpoint:
Default type handler defined:
    {
        echo/n "Stopped by Bp: ";
        print $self;
    }

#8: break line, on [#0/*], Enabled, ignore 0/0
       [0x80002296] CHAPTER6 in chapter6.f line 36
      .
      .
      .
```

In Figure 140, the `set typehandler` command creates a new handler for all breakpoints. From this point on, any breakpoint without its own handler uses the new handler for breakpoints.

The `continue` command resumes process execution. Execution stops when the breakpoint at line 36 is triggered. The new handler for breakpoints executes, displaying a message.

The `info eventtype` command displays all of the breakpoints in the process. In addition, the `info eventtype` command displays the new handler for breakpoints.

The handler for a given eventtype can be cleared with the `clear typehandler` command, as shown in Figure 141.

Figure 141
Clearing the default handler for breakpoints

```
(CXdb) clear typehandler break
(CXdb) info eventtype break
Status of eventpoints of type Breakpoint:

#8: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002296] CHAPTER6 in chapter6.f line 36
      .
      .
      .
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 8, at [0x80002296] CHAPTER6 in chapter6.f line 36
```

The `clear typehandler` command in Figure 141 clears the handler for breakpoints. The `info eventtype` command displays all breakpoints and shows that a handler does not exist for breakpoints. The `continue` command continues execution. The breakpoint at line 36 stops the process. The message displayed shows the default handler is being used once again.

Changing the default handler for eventpoints

If an eventpoint does not have its own handler, and a handler has not been created for its given type, then the default handler for all eventpoints is executed when the eventpoint is triggered.

Initially, the default handler for eventpoints displays an informative message about the eventpoint that stopped the process and where the process is stopped. You can change the default handler for all eventpoints by using the `set default handler` command. After you have set the default handler, all eventpoints without a handler and all those whose type does not have a handler, use the new default handler.

To reset the default handler use the `clear default handler` command.

Figure 142 uses the `remove eventtype` command to remove all breakpoints from the process. This is necessary to prepare for the next section.

Figure 142
Removing all breakpoints

```
(CXdb) remove eventtype break
Eventpoint 8 removed
Eventpoint 6 removed
Eventpoint 5 removed
Eventpoint 0 removed
```

Relation eventpoints

Relation eventpoints are set to watch for a relational expression to evaluate to true. You can write the relational expression using the syntax of the source language you are debugging. When the eventpoint is created, the expression must evaluate to false.

When process execution continues, CXdb checks the value of the expression after the execution of each statement. If the relational expression evaluates to true, the eventpoint is triggered. The default handler for relation eventpoints stops the process and displays a message.

You can set a relation eventpoint to watch for any valid language expression that evaluates to true or false. In FORTRAN, the expression must evaluate to `.TRUE.` for true or `.FALSE.` for false. In C, the expression must evaluate to nonzero for true or 0 for false.

Note

Because of the additional overhead in evaluating a relational expression, relational eventpoints can significantly reduce the speed of execution. If possible, before setting a relation eventpoint, set a breakpoint at the region of code you wish to monitor.

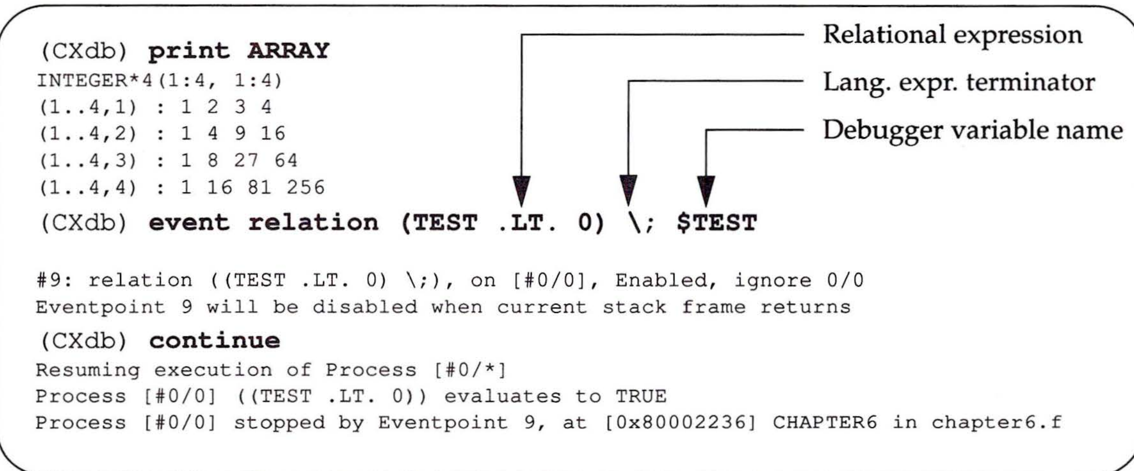
CXdb also displays a message informing you that the eventpoint will be disabled when the current stack frame returns. Relation eventpoints are enabled only for the current routine and all routines that it calls. When the current stack frame returns, the relation eventpoint is disabled.

If you want an relation eventpoint enabled over a larger portion of the program than the current routine, you can move up the stack and place the eventpoint higher in the call chain. Moving up the stack is explained in "Changing frames to reference variables," in Chapter 13.

CXdb disables the relation eventpoint to ensure that a different program variable with the same name but in a different scope does not cause the relational expression to evaluate to true.

You create relation eventpoints with the `event relation` command, as illustrated in Figure 143.

Figure 143
Setting a relation eventpoint



The `print` command in Figure 143 displays the current values of the variable `ARRAY`. The `event relation` command creates an eventpoint that triggers when the value of `TEST` is less than `0`. CXdb responds by displaying the eventpoint number, type of eventpoint, relational expression being watched, and current settings for the newly created eventpoint. CXdb also informs you that eventpoint 9 will be disabled when the current stack frame returns.

The debugger variable `$TEST` has been created to hold the eventpoint number of this eventpoint. The debugger variable must be separated from the language expression by the language expression terminator (`\;`).

The `continue` command resumes process execution. Process execution stops when the relational expression becomes true.

Debugger variables and eventpoints

You can create debugger variables to give your eventpoints names. Then, when you want to refer to a particular eventpoint, you can use its name in a command, rather than having to remember its eventpoint number. This is especially useful in command files, where you might not always be sure what number the eventpoint will be given.

You can manipulate the relation eventpoint by using its debugger variable name, rather than having to remember its eventpoint number. Use debugger variables that store eventpoint numbers in the same way you would use the eventpoint number itself.

Figure 144 demonstrates the debugger variable `$TEST` being used to manipulate the relational eventpoint.

Figure 144
Removing an eventpoint using a debugger variable

```
(CXdb) info event $TEST

#9: relation ((TEST .LT. 0) \;), on [#0/0], Enabled, ignore 0/0

(CXdb) remove event $TEST
Eventpoint 9 removed
```

The `info event` command displays information about the eventpoint number stored in the debugger variable `$TEST`. The `remove event` command removes eventpoint 9 by again referring to its debugger variable.

The debugger variable `$TEST` was assigned to the eventpoint when it was created (refer to Figure 143). It is deleted when the eventpoint is removed.

Signals

A signal is an interrupt sent to a process notifying the process of a particular event's occurrence. There are signals for all types of events, such as stopping a process, continuing a process, or indicating that a floating point error has occurred.

With CXdb, you can catch any of the standard operating signals listed in `sigvec(2)`.

When the operating system sends a signal to your process, CXdb catches the signal before it reaches the process. This gives you an opportunity to find out what signal it is and to choose whether or not to send the signal to the process.

CXdb has several commands that specifically deal with signals. You can uses a signal eventpoint to catch a particular signal. You can send a signal to the process. You can also determine the actions CXdb takes for each different type of signal that it catches.

Note

If you catch a fatal signal, such as a `SIGBUS` signal, it is impossible to continue process execution as if the signal had never occurred. You can gather information about the state of your process, but CXdb will not allow you to continue execution.

Signal actions

Three actions are associated with each signal. These actions control how CXdb behaves when it catches a signal. The actions are listed and described below:

- `stop`—Stops the process when the signal is caught.
- `pass`—Passes the value stored in the predefined debugger variable `$signal` when process execution resumes.
- `print`—Prints a message indicating the signal was caught.

You can enable or disable each action. An action is only performed when enabled.

For each of the 31 signals, you can enable or disable the three signal actions. The `info signal` command displays the current settings for signal actions, as shown in Figure 145.

Figure 145
Displaying the current signal actions

```
(CXdb) info signal

The current signal actions are:
                                                    Signal actions
Signal                          Signal
number  Stop  Pass  Print  name
------  ----  ----  -----  ----------------------
0       Yes   Yes   Yes    Signal 0
1       Yes   Yes   Yes    Hangup
2       Yes   Yes   Yes    Interrupt  <------  SIGINT signal (signal 2)
3       Yes   Yes   Yes    Quit
4       Yes   Yes   Yes    Illegal instruction
5       Yes   No    No     Trace/BPT trap
6       Yes   Yes   Yes    IOT trap
7       Yes   Yes   Yes    EMT trap
8       Yes   Yes   Yes    Floating point exception
9       Yes   Yes   Yes    Killed
10      Yes   Yes   Yes    Bus error
11      Yes   Yes   Yes    Segmentation fault
12      Yes   Yes   Yes    Bad system call
13      Yes   Yes   Yes    Broken pipe
14      No    Yes   No     Alarm clock
15      Yes   Yes   Yes    Terminated
16      No    Yes   No     Urgent I/O condition
17      Yes   Yes   Yes    Stopped (signal)
18      Yes   Yes   Yes    Stopped
19      Yes   Yes   Yes    Continued
20      No    Yes   No     Child exited
21      Yes   Yes   Yes    Stopped (tty input)
22      Yes   Yes   Yes    Stopped (tty output)
23      No    Yes   No     I/O possible
24      Yes   Yes   Yes    Cputime limit exceeded
25      Yes   Yes   Yes    Filesize limit exceeded
26      No    Yes   No     Virtual timer expired
27      No    Yes   No     Profiling timer expired
28      No    Yes   No     Window size changes
29      Yes   Yes   Yes    Resource Lost
30      Yes   Yes   Yes    User defined signal 1
31      Yes   Yes   Yes    User defined signal 2
```

The `info signal` command displays a table of the signal actions for all 31 signals. A `Yes` indicates the signal actions is enabled, and a `No` indicates the action is disabled.

You can change the actions for a signal by using the `set signal` command. When enabling an action, specify the name of the action, either `stop`, `pass`, or `print`. When disabling an action, prefix the action with "no", either `nostop`, `nopass`, or `noprint`. This is demonstrated in Figure 146.

Figure 146
Changing the signal actions for the `SIGINT` signal

```
(CXdb) set signal 2 noprint
(CXdb) info signal SIGINT

Signal                           Signal
number  Stop  Pass  Print  name
------  ----  ----  -----  ----------------------
2        Yes   Yes   No     Interrupt
```

The `set signal` command in Figure 146 sets the signal actions for the `SIGINT` signal. The signal number for the `SIGINT` signal was determined from the `info signal` command in Figure 145. Now, when the `SIGINT` signal is caught, CXdb stops the process, but does not print a message indicating the signal was caught. CXdb will still pass the value of the debugger variable `$signal` to the process when process execution continues. Because the `pass` and `stop` actions were not specified, their settings remain unchanged.

The `info signal` command in Figure 146 verifies that the signal actions are changed for the `SIGINT` signal.

Figure 147
Catching the `SIGINT` signal

```
(CXdb) continue
Resuming execution of Process [#0/*]
(CXdb) print $signal
(INTEGER*4) 2
(CXdb) evaluate $signal = 0
(CXdb) step
Stepping process [#0/*] by 1 statement

INFO: 98
No debugging information available for stepping thread 0.
Continuing until routine exit.
Process [#0/0] stopped stepping at [0x80002336] CHAPTER6 in chapter6.f line 45
```

In Figure 147, the `continue` command resumes execution. Execution is stopped when CXdb catches the `SIGINT` signal. No message is printed, because the `print` action was disabled in Figure 146.

The print command verifies that the SIGINT signal was caught. The debugger variable $signal holds the signal number of the last signal caught. The evaluate command sets the value of the debugger variable $signal to 0. If $signal is set to 0, no signal is passed to the process when execution resumes. This keeps the process from receiving a signal that was caught, even if the pass action is enabled for that signal.

The step command steps execution to the next statement. Because execution had last been stopped inside of the KILL() routine, no debugging information existed for CXdb to step by. CXdb displays the error message informing you of this. When the KILL() routine finishes, execution is then stepped appropriately.

Signal eventpoints

A signal eventpoint watches for a particular signal to be sent to your process. CXdb catches the signal before it reaches the process.

When CXdb intercepts a signal being sent to a process, it checks to see if a signal eventpoint has been created for it. If it has, then the eventpoint is triggered, and the handler for that eventpoint is executed. The default handler for signal eventpoints stops the process, and then displays a message telling you what signal was caught. If a signal eventpoint has not been created, the default actions for that signal are taken.

You can create a signal eventpoint with the event signal command, as shown in Figure 148.

Figure 148
Setting a signal eventpoint

```
(CXdb) event signal int {echo "Caught INT signal";eval $signal=0;}

#10: signal 2 on [#0], Enabled, ignore 0/0
    {
        echo "Caught int signal";
        eval $signal=0;
    }
(CXdb) continue
Resuming execution of Process [#0/*]
Caught INT signal
```

Figure 148 creates a signal eventpoint to watch for a SIGINT signal. The handler for this eventpoint displays a message and then sets the value of $signal to 0.

The `continue` command resumes process execution. Execution stops when CXdb catches the `SIGINT` signal.

Sending a signal to the process

A specific signal can be sent to the process with the `signal process` command. Process execution resumes and the signal is caught by the process.

Figure 149 shows the `signal process` command being used to send the `SIGALRM` signal (signal 14) to the process. The process resumes execution and exits with a message indicating that the signal was received by the process.

Figure 149
Sending a signal to the process

```
(CXdb) signal process SIGALRM
Resuming execution of Process [#0] with signal 14

Process [#0] exited with code 14.
```

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 150.

Figure 150
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Displaying and modifying variables

7

This chapter describes how to display and modify the contents of the variables and memory segments associated with your program. CXdb provides commands that can access the variables either symbolically by identifier or directly by address.

This chapter covers the following commands:

- `copy`
- `evaluate`
- `examine`
- `fill`
- `find memory backward`
- `find memory forward`
- `info expression`
- `info formatting`
- `info locals`
- `print`
- `set printopts maxarray`
- `set printopts padding`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 151.

Figure 151
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 151 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 152.

Figure 152
Starting the example program

```
(CXdb) break line chapter7F.f:10 $BL7F10
#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002788] CHAPTER7 in chapter7F.f line 10
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80002788] CHAPTER7 in chapter7F.f line 10
```

The `break line` command in Figure 152 sets a breakpoint at the beginning of line 10 in the file `chapter7F.f`. The debugger variable `$BL7F10` stores the eventpoint number of this breakpoint. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

Displaying data

CXdb is a symbolic debugger. This means that you can access your program data by variable identifier rather than having to specify the address where the data is stored.

CXdb also allows you to control the format of how the data is displayed. For example, you can display integers as decimal, octal, or hexadecimal numbers, to name just a few formats.

In addition to displaying data, you can display information about the program variables themselves. This information includes the name of the variable, its data type, its storage location, and its current value.

Displaying information about variables

In a large program with many variables, it is often difficult to remember the names of the variables or to know which ones are defined in the current routine. The `info locals` command provides this information, as shown in Figure 153.

Figure 153
Displaying information about local variables

```
(CXdb) info locals
Process [#0/0]
Frame : 0; [0x80002788] CHAPTER7 in chapter7F.f line 10
Number of locals : 3
     1 :    TABLE = INTEGER*4(1:4, 1:4) 0x8005a058
     2 :    I = (INTEGER*4) 5
     3 :    J = (INTEGER*4) 5
```

The response in Figure 153 indicates that there are 3 local variables in the current routine. The first variable is `TABLE`, which is a two-dimensional (4 x 4) array of integers. The starting address of `TABLE` is `8005a058` (hexadecimal). The second variable is the integer `I`, which has a current value of 5. The third variable is the integer `J`, which also has a current value of 5.

You can obtain more detailed information about a variable by using the `info expression` command. Figure 154 illustrates this command.

Figure 154
Displaying information about a single variable

```
(CXdb) info expression J
object type: Fortran identifier
   location: 0x8005a09c
       size: 4 bytes
       type: INTEGER*4
      value: 5
       6 liveness ranges:
              Start         End          Location
         1.  0x80002726:0x8000272a - register a1
         2.  0x8000272c:0x80002730 - register a3
         3.  0x8000272a:0x80002742 - register a1
         4.  0x80002756:0x8000275a - register s0
         5.  0x80002766:0x8000276c - register s0
         6.  0x80059000:0x8005a000 - 0x8005a09c
```

The `info expression` command in Figure 154 displays information about variable `J`. The response gives the object type, storage location, size, data type, and current value of `J`.

The response in Figure 154 also lists 6 liveness ranges and corresponding storage locations for `J`. A *liveness range* is the range of memory where the value of the variable is available for displaying. As long as the program counter (PC) is within the bounds of the liveness range, CXdb can retrieve the value of the variable from the indicated storage location. If the PC is outside the bounds of the liveness range, then the value of the variable is not available.

The `info expression` command can provide information about debugger variables and about any expression that is valid in the current source language. Figure 155 shows how to obtain information about language expressions, and Figure 156 shows how to obtain information about debugger variables.

Figure 155
Displaying information about language expressions

```
(CXdb) info expression 2*6
object type: Fortran expression result
       size: 4 bytes
       type: INTEGER*4
      value: 12
(CXdb) info expression ISQR
object type: Fortran function
entry point: 0x800027d2
return type: INTEGER*4
return size: 4 bytes
  arg count: 1
  prototype: INTEGER*4 ISQR( INTEGER*4 )
```

The `info expression 2*6` command in Figure 155 evaluates the language expression `2*6` and displays information about the result of that evaluation. The result of the evaluation is not stored because this language expression does not modify program memory in any way, such as by an assignment operation (=). Therefore, no storage location or liveness ranges are listed in this case.

The `info expression ISQR` command in Figure 155 displays information about `ISQR`, which is a function defined in the current FORTRAN program. The prototype line in the response shows that `ISQR` takes an integer value as an argument and returns an integer value to the calling routine.

Figure 156
Displaying information about debugger variables

```
(CXdb) info expression $BL7F10
object type: debugger variable
   writable: yes
   var type: reference to eventpoint [#0]
(CXdb) info expression $A0
object type: debugger variable
   writable: yes
   var type: predefined register access
   register: a0 as 32 bits (equivalent to sp)
      value: (INTEGER*4) 0xffffca30
```

The `info expression $BL7F10` command in Figure 156 displays information about the debugger variable `$BL7F10`. This debugger variable contains the eventpoint number for the breakpoint set in Figure 152.

The `info expression $A0` command in Figure 156 displays information about the debugger variable `$A0`. This is one of the debugger variables that have been predefined in CXdb to enable you to access the process registers. In particular, `$A0` references address register A0, which is also the stack pointer (SP). For more information about debugger variables, refer to the *CONVEX CXdb Reference* manual.

Printing data

The `print` command displays program data as well as the contents of debugger variables and the results of language expressions. Figure 157 and Figure 158 illustrate these uses of the `print` command.

Figure 157
Printing program variables and language expressions

```
(CXdb) print J
(INTEGER*4) 5
(CXdb) print I+J
(INTEGER*4) 10
(CXdb) print TABLE
INTEGER*4(1:4, 1:4)
(1..4,1) :  2 4 6 8
(1..4,2) :  5 12 21 32
(1..4,3) :  10 26 54 100
(1..4,4) :  17 48 129 320
```

In Figure 157, the command `print J` prints the current value of the program variable `J`. The command `print I+J` evaluates the language expression `I+J` and prints the result without storing it. The command `print TABLE` prints the elements of the array `TABLE`.

Figure 158
Printing debugger variables and registers

```
(CXdb) print $BL7F10
(INTEGER*4) 0
(CXdb) print $S0
(INTEGER*8) 0x5
(CXdb) print $A0
(INTEGER*4) 0xffffca30
```

In Figure 158, the command `print $BL7F10` prints the contents of the debugger variable `$BL7F10`, which contains a breakpoint number. The command `print $S0` prints the contents of the predefined debugger variable `$S0`, which represents scalar register S0. The command `print $A0` prints the contents of the predefined debugger variable `$A0`, which represents address register A0.

Formatting printed output

The `print` command allows you to specify formatting options for the output, as illustrated in Figure 159.

Figure 159
Formatting printed output

```
(CXdb) print/B J
(INTEGER*4) 0000 0000 0000 0000 0000 0000 0000 0101
(CXdb) print/x TABLE
INTEGER*4(1:4, 1:4)
(1..4,1) :  0x2 0x4 0x6 0x8
(1..4,2) :  0x5 0xc 0x15 0x20
(1..4,3) :  0xa 0x1a 0x36 0x64
(1..4,4) :  0x11 0x30 0x81 0x140
```

The `print/B J` command in Figure 159 prints the value of variable `J` in binary (`/B`) format. The `print/x TABLE` command prints the array `TABLE` in hexadecimal (`/x`) format. For information about other print formatting options, refer to the *CONVEX CXdb Reference*.

Setting print options

The `set printopts` command sets the following options for printing:

- `maxarray`—The maximum number of array elements printed at one time.
- `padding/nopadding`—Leading zeros that force alignment of integers.
- `precision`—The precision for printing real (floating point) numbers.

The `info formatting` command displays the current settings of the print options.

Figure 160 and Figure 161 illustrate the use of these commands.

Figure 160
Setting the `padding` option

```
(CXdb) print TABLE
INTEGER*4(1:4, 1:4)
(1..4,1) :  2 4 6 8          <------ Without padding
(1..4,2) :  5 12 21 32
(1..4,3) :  10 26 54 100
(1..4,4) :  17 48 129 320              With padding
(CXdb) set printopts padding                |
(CXdb) print TABLE                          v
INTEGER*4(1:4, 1:4)
(1..4,1) :   000000002 000000004 000000006 000000008
(1..4,2) :   000000005 000000012 000000021 000000032
(1..4,3) :   000000010 000000026 000000054 000000100
(1..4,4) :   000000017 000000048 000000129 000000320
```

The first `print TABLE` command in Figure 160 prints the elements of array `TABLE`. The values are not aligned properly because padding is disabled. The `set printopts padding` command enables padding with leading zeros. The third command prints the array again, and this time the values are aligned because padding is enabled.

Figure 161
Setting the `maxarray` option

```
(CXdb) set printopts maxarray 25
(CXdb) info formatting
Global format settings:
 padding: On
 max array elements: 25
 floating point format: 10.4

Process [#0] format settings:

Format settings for Thread 0
            Selected memory size: (none)
         Selected Memory Formats:
                 byte=(none), halfword=(none), word=(none)
                 longword=(none), quadword=(none)
```


The `set printopts maxarray 25` command in Figure 161 sets 25 as the maximum number of array elements that print at one time. The `info formatting` command shows the current settings of the print options.

Modifying data

In addition to displaying data values, CXdb allows you to modify them. You can modify data values within any CXdb command that accepts a language expression, but the `evaluate` and `print` commands are specifically designed for modifying data values. Both of these commands evaluate language expressions. The `print` command also prints the result of the evaluation, but the `evaluate` command does not.

Modifying program data

To modify the value of a program variable, you construct a language expression that assigns a new value to that variable. Figure 162 shows how to use such a language expression with the `evaluate` command.

Figure 162
Modifying variables with the `evaluate` command

```
(CXdb) print J
(INTEGER*4) 000000005
(CXdb) evaluate J=J+1
(CXdb) print J
(INTEGER*4) 000000006
```

The first `print J` command in Figure 162 shows that the original value of `J` is 5. The `evaluate J=J+1` command increments `J` by 1. The new value is stored in `J` because the assignment operator (=) is used in this case. The last command prints `J` again, showing its new value is 6.

Another way to do the same thing is by using the language expression in the `print` command, as shown in Figure 163.

Figure 163
Modifying variables with the `print` command

```
(CXdb) print J=J+1
(INTEGER*4) 000000007
```

Modifying debugger variables and registers

The `evaluate` and `print` commands can also modify debugger variables and the contents of process registers. Figure 164 and Figure 165 illustrate this.

Figure 164
Modifying debugger variables

```
(CXdb) evaluate $Count=0
(CXdb) print $BL7F10=2.5
(REAL*4) 2.5000
```

In Figure 164, the `evaluate $Count=0` command creates a new debugger variable called `$Count` and initializes it to 0.

The `print $BL7F10=2.5` command in Figure 164 changes the value of the debugger variable `$BL7F10` to 2.5. Note that `$BL7F10` originally contained an eventpoint number (see Figure 152), but now it contains the real number 2.5.

Note

A debugger variable assumes the data type of the last value assigned to it. To reference an eventpoint, the debugger variable must have a data type of either integer or debugger variable.

Figure 165
Modifying process registers

```
(CXdb) print $S0=33
(INTEGER*8) 0000000000000000033
```

The `print $S0=33` command in Figure 165 assigns the value 33 to the scalar register S0.

Executing program routines and functions

The `print` and `evaluate` commands can also be used to execute routines and functions from your program. Figure 166 illustrates this capability with the `print` command.

Figure 166
Printing the value returned by a program function

```
(CXdb) print ISQR(3)
(INTEGER*4) 000000009
```

The `print ISQR(3)` command in Figure 166 executes the program function `ISQR`. This function accepts an integer value as an argument, squares that value, and returns the result. The `print` command then prints the value returned by the function.

Executing a program routine or function with the `print` or `evaluate` command is separate from execution of the process being debugged. However, it is subject to any eventpoints or signal handling set for the current process. Figure 167 illustrates this point.

Figure 167
Printing the value returned by a program function that contains a breakpoint

```
(CXdb) break routine ISQR

#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800027d4] ISQR in chapter7F.f line 17
(CXdb) print ISQR(3)
Process [#0/0] stopped by Bkpt 1, at [0x800027d4] ISQR in chapter7F.f line 17
(CXdb) continue
Resuming execution of Process [#0/*]
(CXdb) (INTEGER*4) 000000009 <------------- Response to print command
(CXdb) remove event 1
Eventpoint 1 removed
```

The `break routine` command in Figure 167 sets a breakpoint at the beginning of the `ISQR` routine. The `print` command then calls `ISQR`, but execution is halted by the breakpoint. The `continue` command resumes execution of `ISQR`, and the resulting value of 9 is printed. The `remove event` command removes the breakpoint.

Caution

The contents of memory and registers can change when you execute a program routine by calling it from a CXdb command. The results can also differ from those obtained when executing the routine in normal program sequence.

Using other CXdb commands to modify data

Any CXdb command that evaluates language expressions can be used to modify program data and memory. For example, a language expression that contains the assignment operator (=) can change the value of a variable.

Figure 168 shows an `info expression` command that uses a language expression to modify the value and data type of the debugger variable `$Count`.

Figure 168
Modifying data with the `info expression` command

```
(CXdb) info expression $Count=ISQR(3)+$S0
object type: Fortran expression result
       size: 8 bytes
       type: INTEGER*8
      value: 0000000000000000042
```

Examining and modifying memory

In addition to displaying and modifying data by means of language expressions, you can examine and modify the contents of memory directly. You can also search memory for a particular byte pattern.

Examining the contents of memory

The `examine` command displays the contents of memory. This command also provides formatting options that let you specify how the memory is displayed. Figure 169 and Figure 170 illustrate the use of the `examine` command.

Figure 169
Examining memory

```
(CXdb) break line 38 $BL7F38
#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x8000292e] BLD_MATRIX in chapter7F.f line 38
(CXdb) continue 15
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 2, at [0x8000292e] BLD_MATRIX in chapter7F.f line 38
(CXdb) print MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :     -7.5000    -3.0250     0.4600     3.4500     0.0000
(1..5,2,1) :     -2.0250     6.4600    16.4500    28.1571     0.0000
(1..5,3,1) :      4.4600    21.4500    50.1571     0.0000     0.0000
(1..5,4,1) :     12.4500    44.1571   125.6875     0.0000     0.0000
(1..5,5,1) :      0.0000     0.0000     0.0000     0.0000     0.0000
...
(CXdb) examine loc(MATRIX)
Examine Process [#0/0] from 0x8005a0ac to 0x8005a0f8
 8005a0ac:     c1f00000  c141999a  3feb8520  415cccce  00000000  c101999a
 8005a0c4:     41ceb852  4283999a  42e141d4  00000000  418eb852  42ab999a
 8005a0dc:     4348a0ea  00000000  00000000  42473334  4330a0ea  43fb6000
 8005a0f4:     00000000  00000000
```

In Figure 169, the `break line` command sets a breakpoint at line 38 of the example program and stores the breakpoint number in the debugger variable `$BL7F38`. This part of the program constructs a 3-dimensional (5 x 5 x 5) array called `MATRIX`. A loop in the program fills `MATRIX` with real (floating point) numbers. The `continue 15` command cycles through the loop and fills part of the array. The `print MATRIX` command displays the first 25 elements of `MATRIX` (because `maxarray` is set to 25).

The `examine loc(MATRIX)` command in Figure 169 displays the region of memory where `MATRIX` is stored. The FORTRAN function `loc()` returns the starting address of `MATRIX`. However, any language expression that evaluates to a valid program address can be used with the examine command.

The default for the `examine` command displays 20 units of memory at a time. The default memory unit is words, and each element of `MATRIX` is one word long, so only the first 20 elements of `MATRIX` are displayed in the above example. However, you can specify the memory unit size and the number of units as well as the display format for the `examine` command. Figure 170 illustrates these features.

Figure 170
Specifying a memory region and display format with the `examine` command

```
(CXdb) examine loc(MATRIX):25        <-------------- Examining 25 words
Examine Process [#0/0] from 0x8005a0ac to 0x8005a10c
 8005a0ac:    c1f00000  c141999a  3feb8520  415cccce  00000000  c101999a
 8005a0c4:    41ceb852  4283999a  42e141d4  00000000  418eb852  42ab999a
 8005a0dc:    4348a0ea  00000000  00000000  42473334  4330a0ea  43fb6000
 8005a0f4:    00000000  00000000  00000000  00000000  00000000  00000000
 8005a10c:    00000000
(CXdb) examine/b loc(MATRIX):25      <-------------- Examining 25 bytes
Examine Process [#0/0] from 0x8005a0ac to 0x8005a0c4
 8005a0ac:    c1  f0  00  00  c1  41  99  9a  3f  eb  85  20  41  5c  cc  ce
 8005a0bc:    00  00  00  00  c1  01  99  9a  41
(CXdb) examine/bB loc(MATRIX):8      <------ Examining 8 bytes in binary format
Examine Process [#0/0] from 0x8005a0ac to 0x8005a0b3
 8005a0ac:    1100 0001   1111 0000   0000 0000   0000 0000   1100 0001
 8005a0b1:    0100 0001   1001 1001   1001 1010
(CXdb) examine/f loc(MATRIX):25<-------- Examining 25 words in floating
Examine Process [#0/0] from 0x8005a0ac to 0x8005a10c          point format
 8005a0ac:       -7.5000     -3.0250      0.4600      3.4500      0.0000
 8005a0c0:       -2.0250      6.4600     16.4500     28.1571      0.0000
 8005a0d4:        4.4600     21.4500     50.1571      0.0000      0.0000
 8005a0e8:       12.4500     44.1571    125.6875      0.0000      0.0000
 8005a0fc:        0.0000      0.0000      0.0000      0.0000      0.0000
```

In Figure 170, the `examine loc(MATRIX):25` command displays the region of memory occupied by the first 25 words (specified by `:25`) of `MATRIX`. The `examine/b loc(MATRIX):25` command displays the first 25 bytes (`/b`) of `MATRIX`. The `examine/bB loc(MATRIX):8` command displays the first 8 bytes (`/b`) of `MATRIX` in binary (`B`) format. The `examine/f loc(MATRIX):25` command displays the first 25 words of `MATRIX` in floating point (`/f`) format.

Searching memory for a byte pattern

You can search for a particular byte pattern in memory by using the `find memory forward` and `find memory backward` commands, as illustrated in Figure 171.

Figure 171
Searching for a byte pattern in memory

```
(CXdb) find memory forward c141 loc(MATRIX)
Data found at 0x8005a0b0
(CXdb) find memory backward ff loc(MATRIX)
Data found at 0x80058153
```

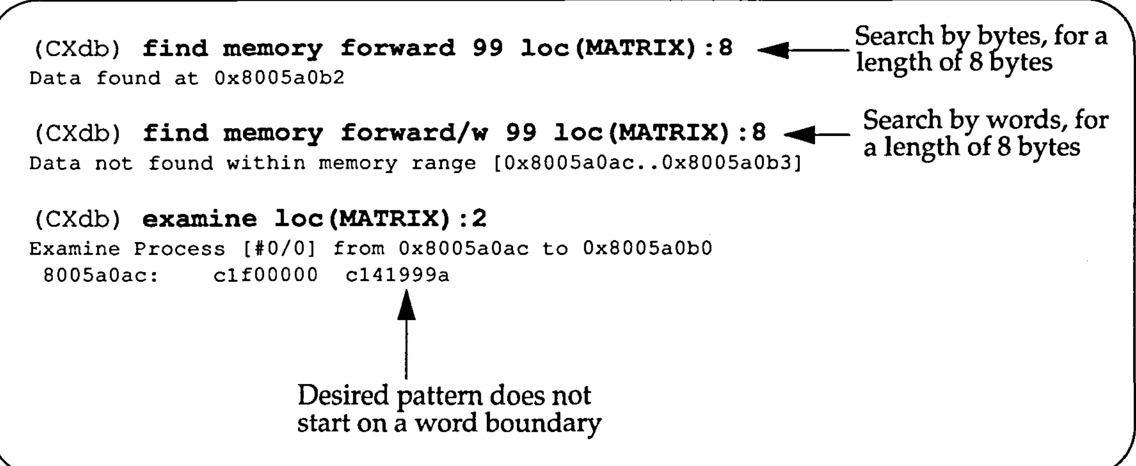
The `find memory forward` command in Figure 171 searches forward through memory for the byte pattern `c141` (hexadecimal). CXdb begins searching memory at the starting address of `MATRIX` and proceeds forward from there.

A byte contains 8 bits, but one hexadecimal digit represents only 4 bits. Therefore, you must specify an even number of hexadecimal digits in the byte pattern for the `find memory` commands.

The `find memory backward` command in Figure 171 searches backward through memory for the byte pattern `ff`. Because the search is backward in this case, CXdb begins searching at the starting address of `MATRIX` and searches backward through memory from that point.

With the `find memory` commands, you can also specify the size of the memory region and the type of memory unit to search, as shown in Figure 172.

Figure 172
Specifying the memory region and search format



The first `find memory forward` command in Figure 172 searches for the byte pattern 99, starting at the beginning address of `MATRIX` and continuing for 8 bytes (specified by `:8`). The second `find memory forward` command also searches the first 8 bytes of `MATRIX` for the byte pattern 99, but it searches only the beginning of each word (specified by `/w`). The `examine` command shows that the byte pattern 99 does not occur as the first byte of a word in the specified memory region, so the second `find memory forward` command does not find this pattern.

Modifying the contents of memory

Two commands modify memory directly: `copy` and `fill`. The `copy` command copies one region of memory into another. The `fill` command fills a region of memory with the resulting value of a language expression. Figure 173, Figure 174, and Figure 175 illustrate the use of these commands.

Caution

Neither the `copy` nor the `fill` command asks for confirmation before writing into process memory. If you have not specified the destination addresses correctly for these commands, they can write over areas of process memory that you did not intend to affect.

Figure 173
Copying one region of memory to another

```
(CXdb) print MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :      -7.5000    -3.0250     0.4600     3.4500     0.0000
(1..5,2,1) :      -2.0250     6.4600    16.4500    28.1571     0.0000
(1..5,3,1) :       4.4600    21.4500    50.1571     0.0000     0.0000
(1..5,4,1) :      12.4500    44.1571   125.6875     0.0000     0.0000
(1..5,5,1) :       0.0000     0.0000     0.0000     0.0000     0.0000
...
(CXdb) copy loc(MATRIX):20 \; loc(MATRIX(1,5,1))
(CXdb) print MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :      -7.5000    -3.0250     0.4600     3.4500     0.0000
(1..5,2,1) :      -2.0250     6.4600    16.4500    28.1571     0.0000
(1..5,3,1) :       4.4600    21.4500    50.1571     0.0000     0.0000
(1..5,4,1) :      12.4500    44.1571   125.6875     0.0000     0.0000
(1..5,5,1) :      -7.5000    -3.0250     0.4600     3.4500     0.0000
...
```

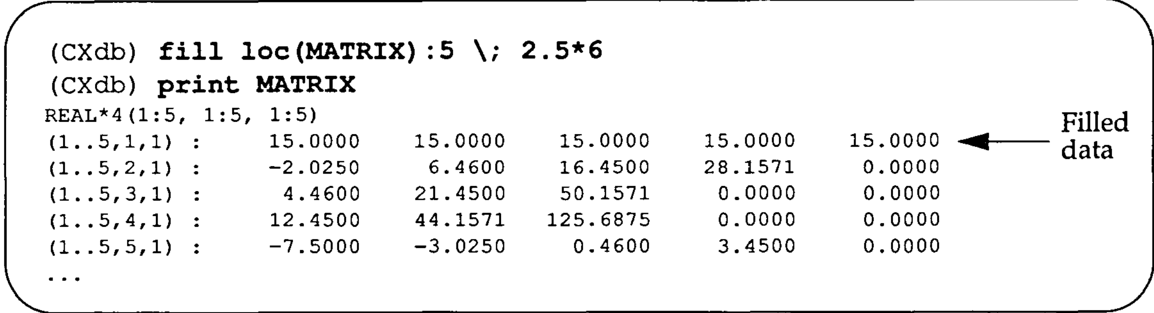

The first `print` command in Figure 173 displays the initial values of the first 25 elements of `MATRIX`. The `copy` command copies the first 20 *bytes* of `MATRIX` into the fifth row `(1,5,1)` of `MATRIX`.

Note

The `copy` command always operates on bytes of memory.

Because each element of `MATRIX` is 4 bytes long, copying the first 20 bytes is equivalent to copying the first row of `MATRIX`. The second `print` command shows the result of the copy.

Figure 174
Filling a region of memory with the value of a language expression



```
(CXdb) fill loc(MATRIX):5 \; 2.5*6
(CXdb) print MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :    15.0000    15.0000    15.0000    15.0000    15.0000
(1..5,2,1) :    -2.0250     6.4600    16.4500    28.1571     0.0000
(1..5,3,1) :     4.4600    21.4500    50.1571     0.0000     0.0000
(1..5,4,1) :    12.4500    44.1571   125.6875     0.0000     0.0000
(1..5,5,1) :    -7.5000    -3.0250     0.4600     3.4500     0.0000
...
```


The `fill` command in Figure 174 evaluates the language expression `2.5*6` and writes the result of the evaluation (15.0) into the first 5 memory units beginning at the starting address of `MATRIX`. The `print` command shows the result of the fill.

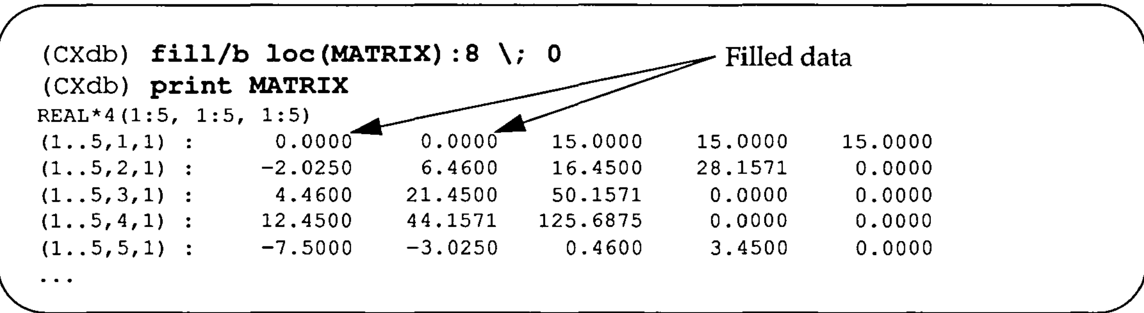
Note

The size of the memory unit used by the `fill` command is determined by the unit size of the item specified as the starting address of the fill.

Because each element of `MATRIX` is one word of memory (4 bytes), the units in this case are words of memory. Thus, the above example fills the first 5 elements (5 words, or 20 bytes) of `MATRIX` with the result of the language expression `2.5*6`.

You can also specify the memory unit size with the `fill` command, as illustrated in Figure 175.

Figure 175
Specifying a memory unit size for the `fill` command



```
(CXdb) fill/b loc(MATRIX):8 \; 0
(CXdb) print MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :     0.0000     0.0000    15.0000    15.0000    15.0000
(1..5,2,1) :    -2.0250     6.4600    16.4500    28.1571     0.0000
(1..5,3,1) :     4.4600    21.4500    50.1571     0.0000     0.0000
(1..5,4,1) :    12.4500    44.1571   125.6875     0.0000     0.0000
(1..5,5,1) :    -7.5000    -3.0250     0.4600     3.4500     0.0000
...
```

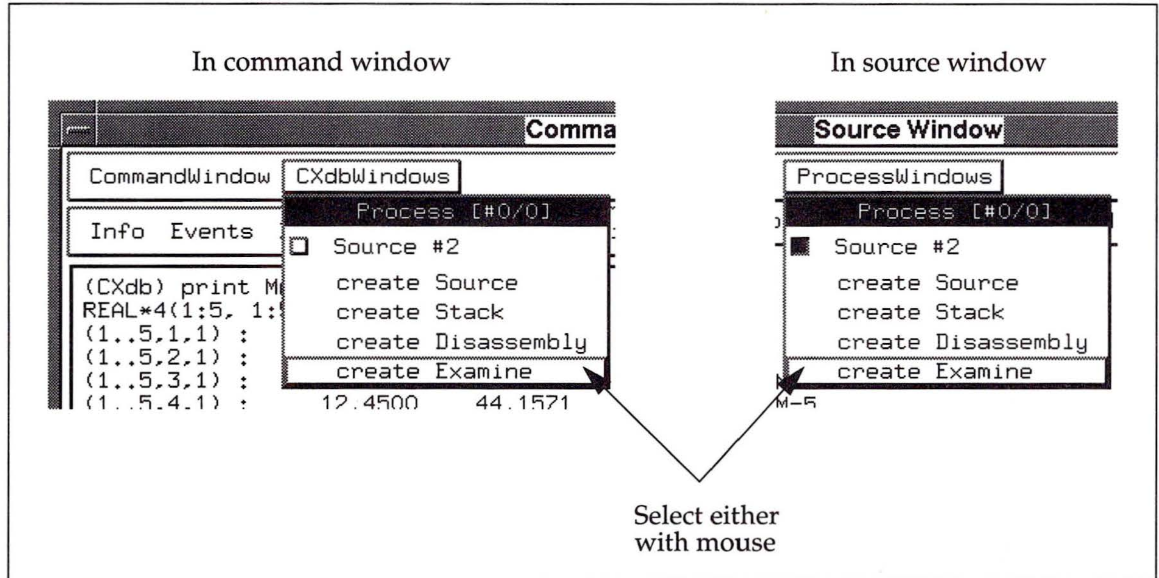

The `fill` command in Figure 175 clears the first 8 bytes (`/b`) of `MATRIX` by filling it with zeros. The `print` command shows the result of the fill.

Using the examine window in CXwindows

In the CXwindows environment, there is a separate examine window for viewing the contents of memory. This section explains how to access and use that window. If you are not using CXwindows, you can skip this section and proceed to the next section, "Working with array slices."

You can open the examine window from either the CXdbWindows menu in the command window or from the ProcessWindows menu in the source window. Both of these methods are shown in Figure 176.

Figure 176
Opening the examine window



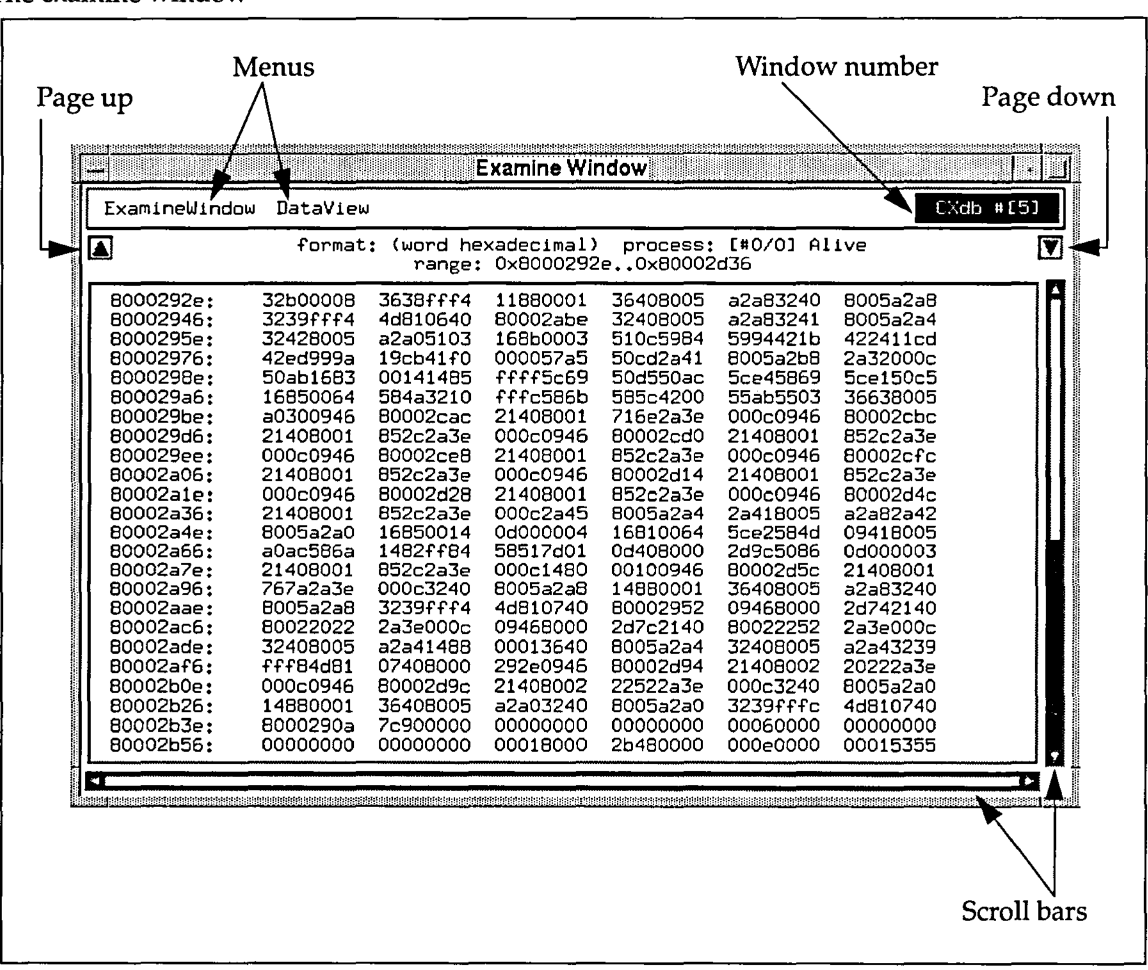
You can open any number of examine windows during a debugging session, and each one can display a different section of memory. When an examine window first opens, it displays the area of memory starting at the current program counter (PC).

Figure 177 shows the examine window. Its parts include:

- **Menus**—These pull-down menus allow you to do the following with the mouse:
  - ExamineWindow closes the examine window or enables and disables the auto update option.
  - DataView lets you select a different area of memory to view, search memory for a byte pattern, or change the format of the display.

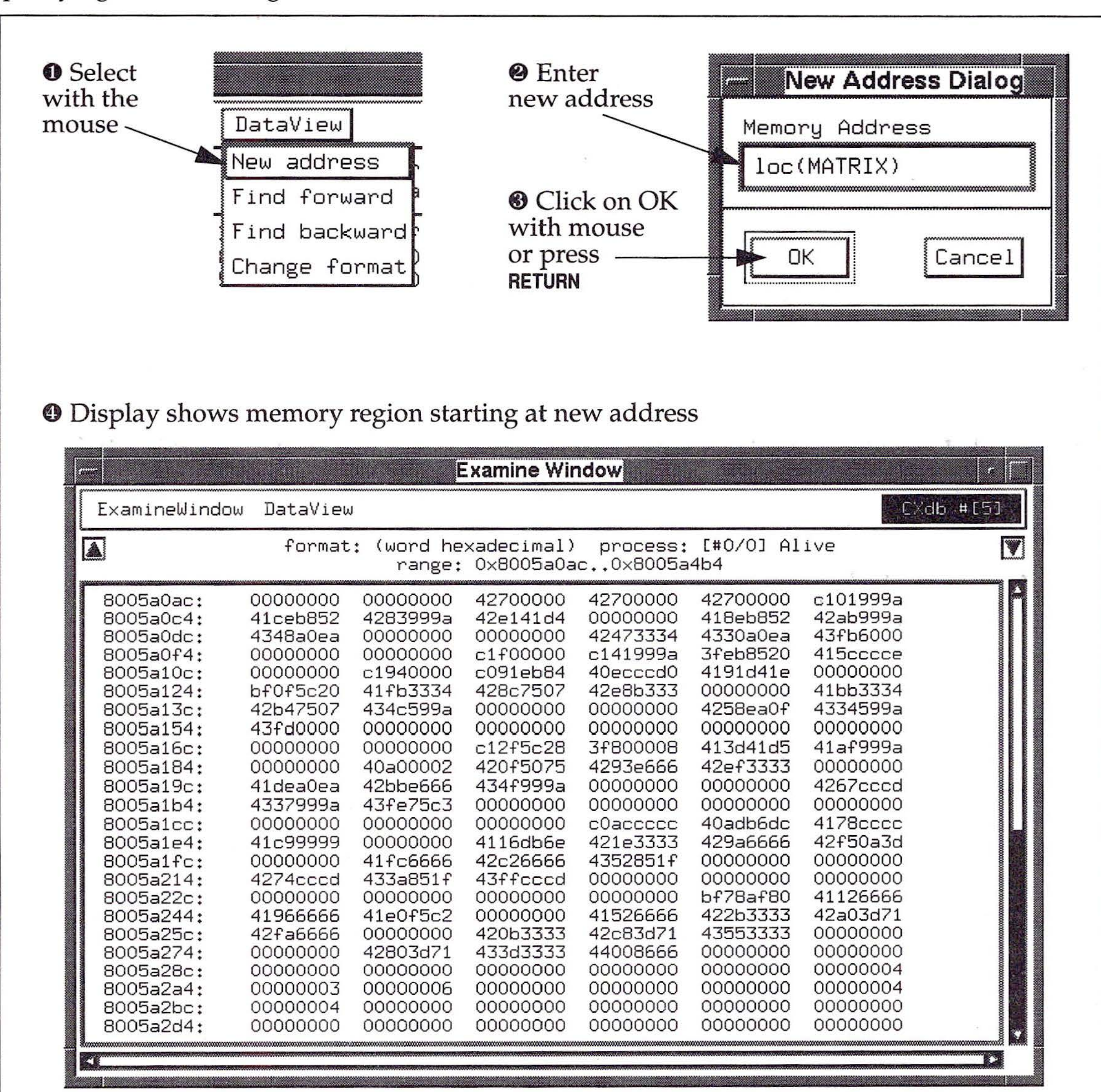
- **Window number**—This is a unique number that identifies each window in CXdb. Certain CXdb commands allow you to specify a window number as a parameter.
- **Scroll bars**—These bars and arrows enable you to scroll the window vertically and horizontally with the mouse.
- **Paging buttons**—Buttons that scroll the window up or down by one page (screen) per click of the button.

Figure 177
The examine window



Generally, the examine window is for viewing the data portion of memory. (To view machine instructions, use the disassembly window, which is described in Chapter 14.) Using the DataView menu, you can specify either the symbolic or absolute starting address of the data you want to display. Figure 178 shows how to use the DataView menu.

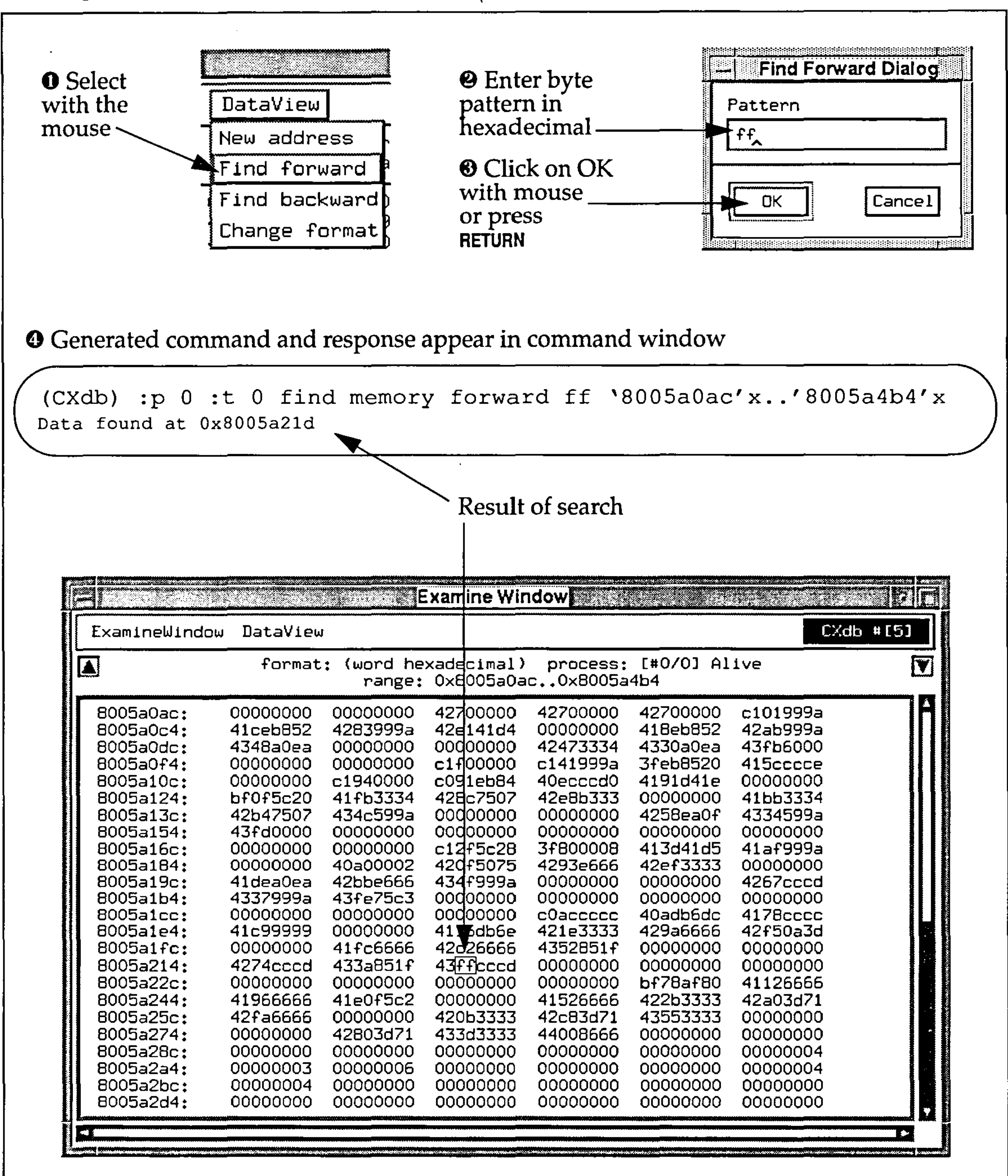
Figure 178
Specifying a new starting address to examine



In Figure 178, the new address to examine is specified as `loc(MATRIX)`, where `loc()` is a FORTRAN function that returns the starting address of the array `MATRIX`. You can specify the address with any language expression that evaluates to a valid program address.

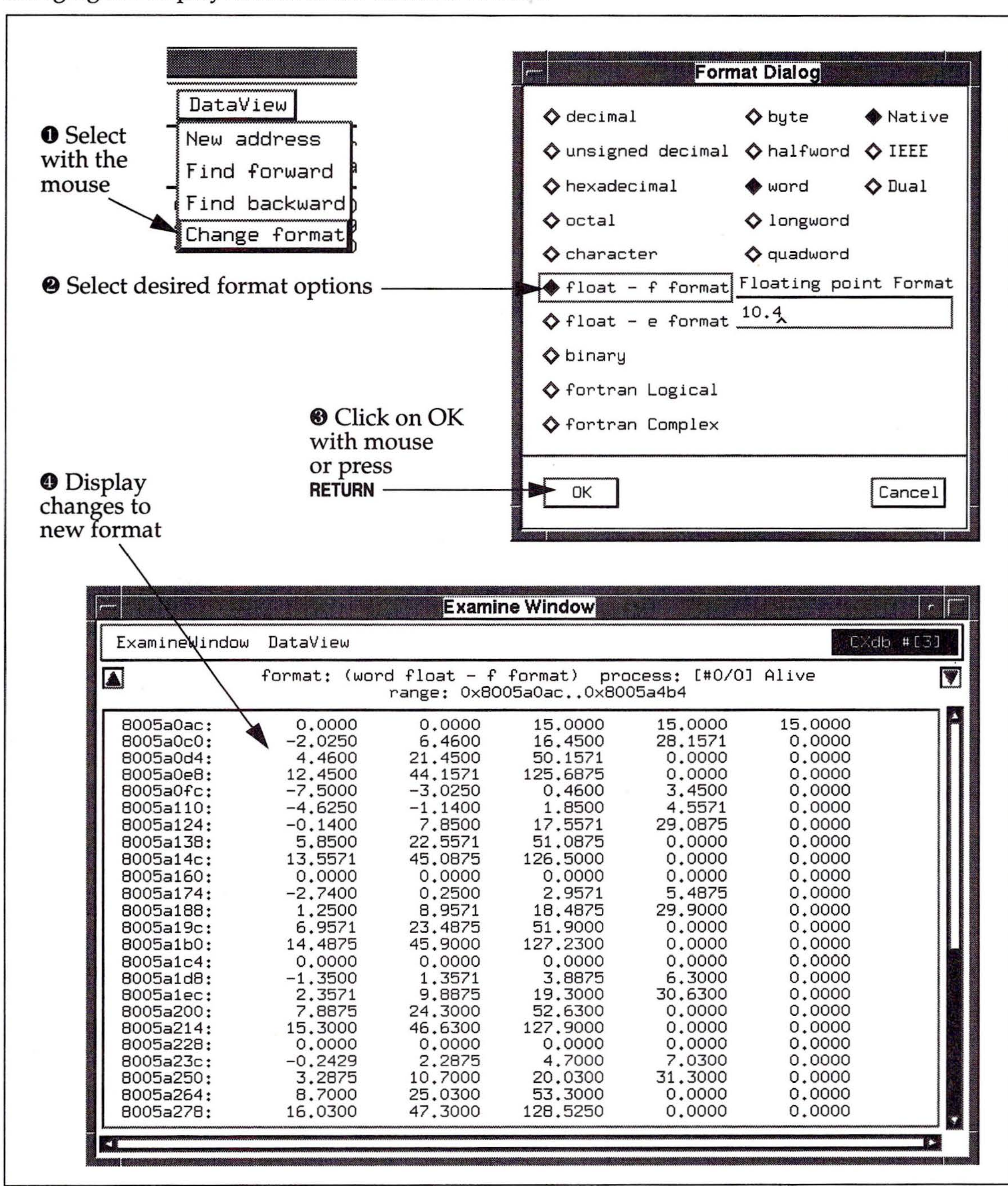
The DataView menu also enables you to search the examine window for a specified byte pattern. The results of the search are displayed in the command window, as shown in Figure 179.

Figure 179
Searching the examine window for a byte pattern



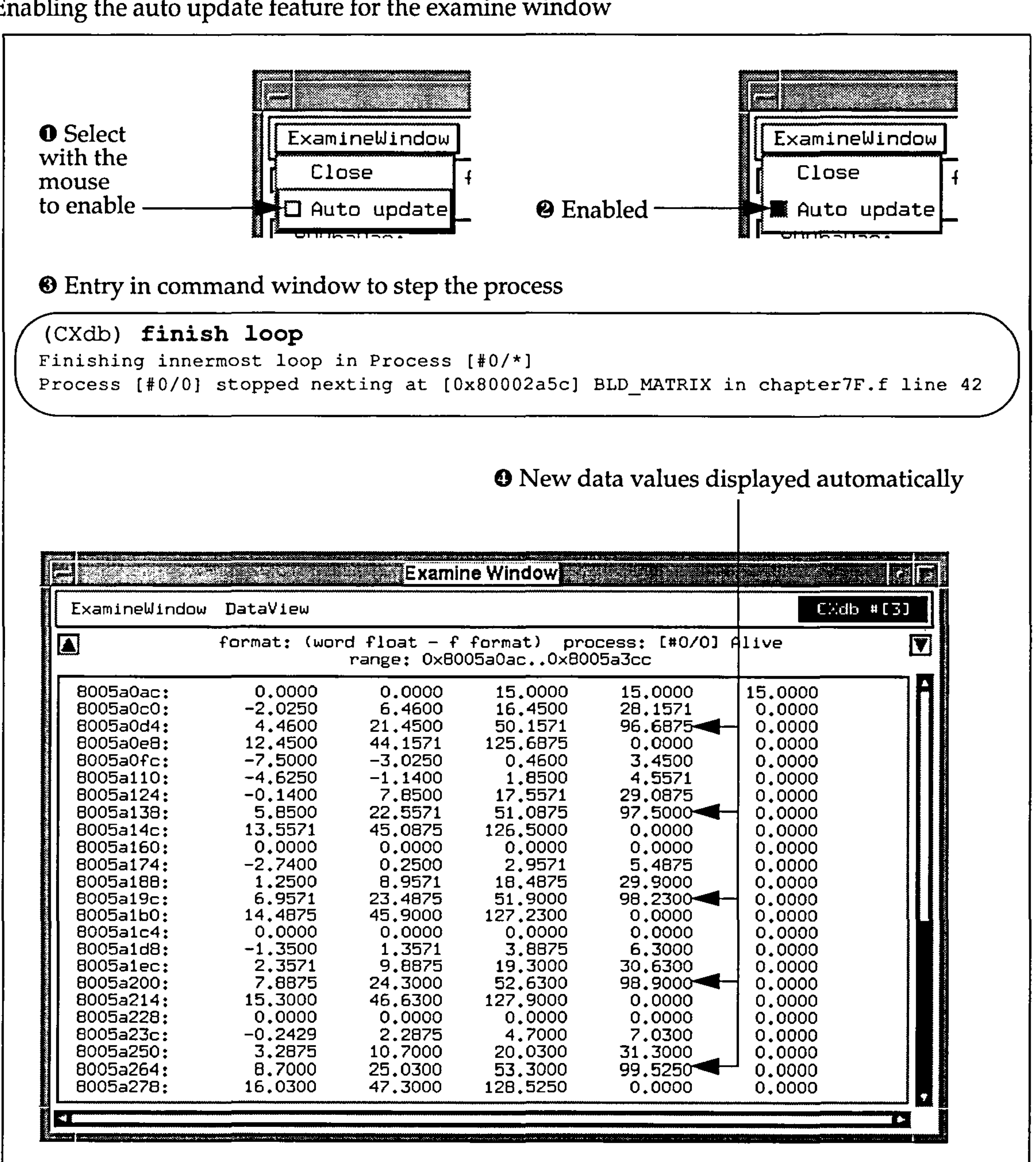
By default, the examine window displays values in hexadecimal format. The DataView menu lets you change the format, as illustrated in Figure 180.

Figure 180
Changing the display format of the examine window



When the examine window first opens, it is not set to update automatically. This means that the display does not update even if the program modifies the area of memory being displayed. However, you can enable the auto update feature by selecting it from the ExamineWindow menu, as shown in Figure 181.

Figure 181
Enabling the auto update feature for the examine window



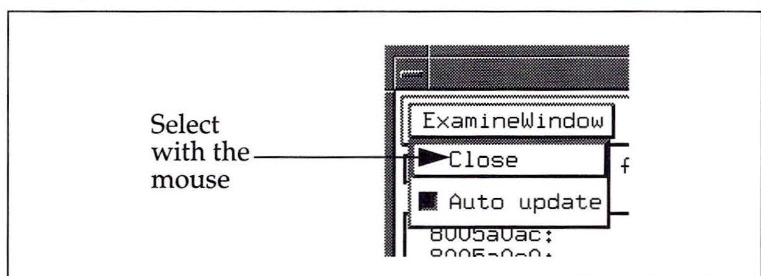
The `finish loop` command in Figure 181 causes the program to modify some of the elements in the array `MATRIX`, which is currently displayed in the examine window. Because the auto update feature is enabled, the examine window automatically displays the updated values of `MATRIX` when process execution stops again.

Note

By setting the appropriate resources in your `.Xdefaults` file, you can enable the auto update feature by default whenever the examine window is opened. For a description of how to do this, refer to the concept "Xdefaults" in the *CONVEX CXdb Reference*.

To close the examine window, select the Close option from the ExamineWindow menu, as shown in Figure 182.

Figure 182
Closing the examine window



Working with array slices

Many CXdb commands enable you to work with a subset of an array, called a slice. The handling of arrays and array slices differs between FORTRAN and C. Therefore, these two languages are treated separately in the next two sections of this chapter. If you work primarily with FORTRAN code, proceed to Section "Array slices in FORTRAN." If you work primarily with C programs, proceed to Section "Array slices in C."

Array slices in FORTRAN

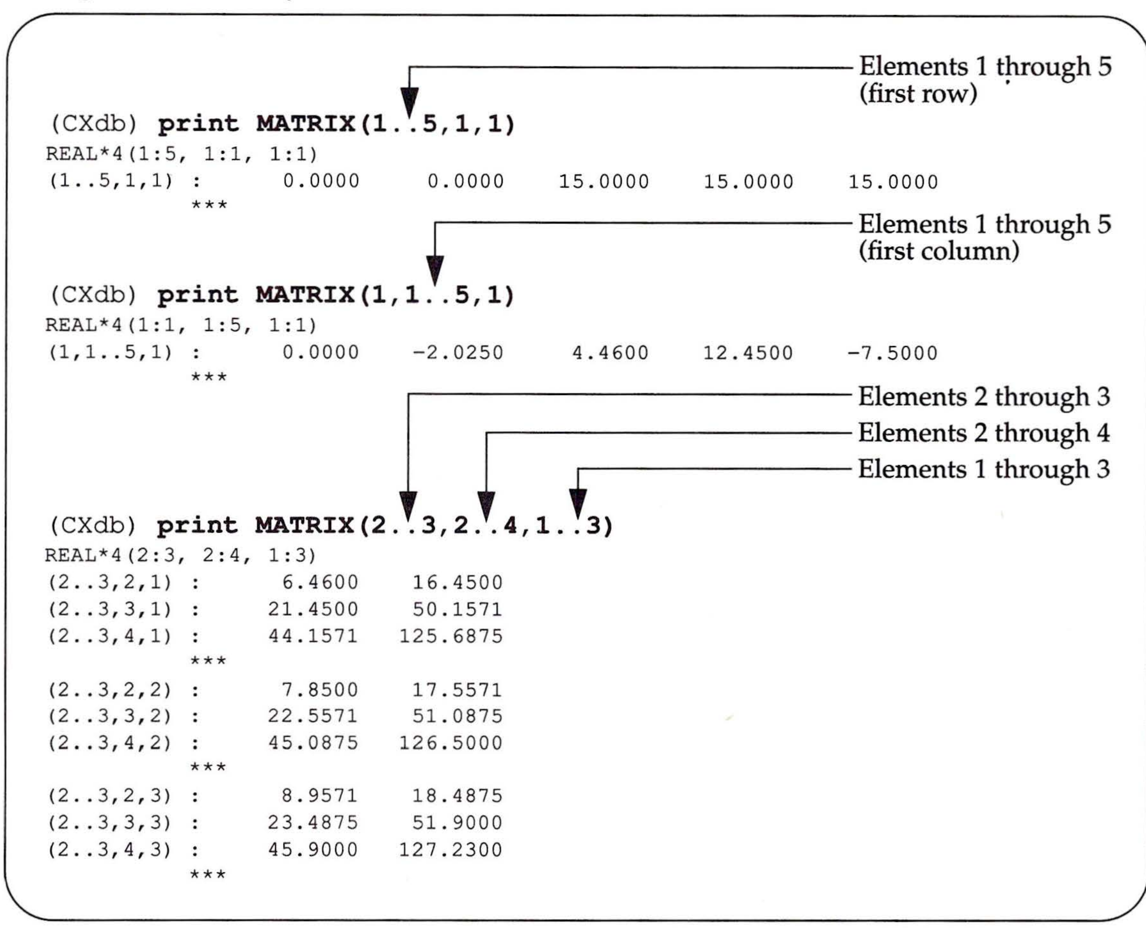
To access an array slice in FORTRAN, you simply specify a subscript range by placing two dots (`..`) between the starting and ending subscripts. For example, Figure 183 shows how to print all elements of array `MATRIX`, and Figure 184 shows how to print specific slices of that array.

Figure 183
Printing all elements of the FORTRAN array `MATRIX`

```
(CXdb) set printopts maxarray 125
(CXdb) print MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :     0.0000     0.0000    15.0000    15.0000    15.0000
(1..5,2,1) :    -2.0250     6.4600    16.4500    28.1571     0.0000
(1..5,3,1) :     4.4600    21.4500    50.1571    96.6875     0.0000
(1..5,4,1) :    12.4500    44.1571   125.6875     0.0000     0.0000
(1..5,5,1) :    -7.5000    -3.0250     0.4600     3.4500     0.0000
          ***
(1..5,1,2) :    -4.6250    -1.1400     1.8500     4.5571     0.0000
(1..5,2,2) :    -0.1400     7.8500    17.5571    29.0875     0.0000
(1..5,3,2) :     5.8500    22.5571    51.0875    97.5000     0.0000
(1..5,4,2) :    13.5571    45.0875   126.5000     0.0000     0.0000
(1..5,5,2) :     0.0000     0.0000     0.0000     0.0000     0.0000
          ***
(1..5,1,3) :    -2.7400     0.2500     2.9571     5.4875     0.0000
(1..5,2,3) :     1.2500     8.9571    18.4875    29.9000     0.0000
(1..5,3,3) :     6.9571    23.4875    51.9000    98.2300     0.0000
(1..5,4,3) :    14.4875    45.9000   127.2300     0.0000     0.0000
(1..5,5,3) :     0.0000     0.0000     0.0000     0.0000     0.0000
          ***
(1..5,1,4) :    -1.3500     1.3571     3.8875     6.3000     0.0000
(1..5,2,4) :     2.3571     9.8875    19.3000    30.6300     0.0000
(1..5,3,4) :     7.8875    24.3000    52.6300    98.9000     0.0000
(1..5,4,4) :    15.3000    46.6300   127.9000     0.0000     0.0000
(1..5,5,4) :     0.0000     0.0000     0.0000     0.0000     0.0000
          ***
(1..5,1,5) :    -0.2429     2.2875     4.7000     7.0300     0.0000
(1..5,2,5) :     3.2875    10.7000    20.0300    31.3000     0.0000
(1..5,3,5) :     8.7000    25.0300    53.3000    99.5250     0.0000
(1..5,4,5) :    16.0300    47.3000   128.5250     0.0000     0.0000
(1..5,5,5) :     0.0000     0.0000     0.0000     0.0000     0.0000
```

The `set printopts` command in Figure 183 sets `maxarray` to 125 so that all 125 elements of the `MATRIX` can print at once. The `print` command then prints `MATRIX`.

Figure 184
Printing FORTRAN array slices



The first `print` command in Figure 184 prints the first row of `MATRIX`. The second `print` command prints the first column of `MATRIX`. The third `print` command prints a slice from the central part of `MATRIX`.

Note

An array slice is a temporary copy of the original array elements. Any operations performed on the slice do not affect the original array. The slice is deleted once the command that invoked it is done executing.

Array slices in C

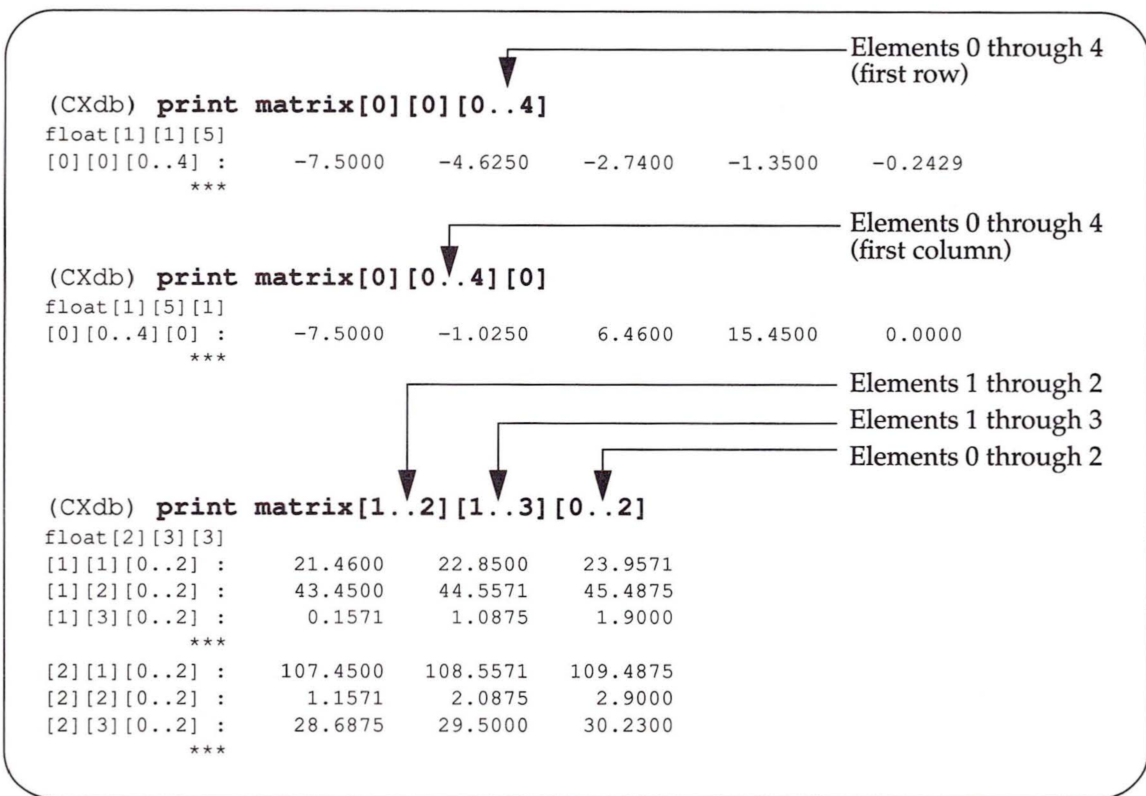
To access an array slice in C language, you simply specify a subscript range by placing two dots (`..`) between the starting and ending subscripts. For example, Figure 185 shows how to print all elements of the C array `matrix`, and Figure 186 shows how to print specific slices of that array.

Figure 185
Printing all elements of the C array `matrix`

```
(CXdb) break line chapter7C.c:50

#3: break line, on [#0/*], Enabled, ignore 0/0
        [0x80004430] chapter7C'bld_matrix in chapter7C.c line 50
(CXdb) disable event $BL7F38
Eventpoint 2 disabled
(CXdb) continue 15
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 3, at [0x80004430] chapter7C'bld_matrix
(CXdb) print matrix
float[5][5][5]
[0][0][0..4] :     -7.5000     -4.6250     -2.7400     -1.3500     -0.2429
[0][1][0..4] :     -1.0250      0.8600      2.2500      3.3571      4.2875
[0][2][0..4] :      6.4600      7.8500      8.9571      9.8875     10.7000
[0][3][0..4] :     15.4500     16.5571     17.4875     18.3000     19.0300
[0][4][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
          ***
[1][0][0..4] :      4.9750      6.8600      8.2500      9.3571     10.2875
[1][1][0..4] :     21.4600     22.8500     23.9571     24.8875     25.7000
[1][2][0..4] :     43.4500     44.5571     45.4875     46.3000     47.0300
[1][3][0..4] :      0.1571      1.0875      1.9000      2.6300      3.3000
[1][4][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
          ***
[2][0][0..4] :     48.4600     49.8500     50.9571     51.8875     52.7000
[2][1][0..4] :    107.4500    108.5571    109.4875    110.3000    111.0300
[2][2][0..4] :      1.1571      2.0875      2.9000      3.6300      4.3000
[2][3][0..4] :     28.6875     29.5000     30.2300     30.9000     31.5250
[2][4][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
          ***
[3][0][0..4] :    315.4500    316.5571    317.4875    318.3000    319.0300
[3][1][0..4] :     -3.8429     -2.9125     -2.1000     -1.3700     -0.7000
[3][2][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
[3][3][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
[3][4][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
          ***
[4][0][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
[4][1][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
[4][2][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
[4][3][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
[4][4][0..4] :      0.0000      0.0000      0.0000      0.0000      0.0000
```

The `break line` command in Figure 185 sets a breakpoint at line 50 of the C function that generates `matrix`. The `disable event` command disables the breakpoint where the process is currently stopped. The `continue` command continues execution of the process, which builds part of `matrix`. The `print` command prints all elements of `matrix`.

Figure 186
Printing C array slices



The first `print` command in Figure 186 prints the first row of `matrix`. The second `print` command prints the first column of `matrix`. The third `print` command prints a slice from the central part of `matrix`.

Note

An array slice is a temporary copy of the original array elements. Any operations performed on the slice do not affect the original array. The slice is deleted once the command that invoked it is done executing.

Using scope paths to access variables

If you want to access a variable that is local to the current routine, you can generally do so without qualifying the variable name. However, if you want to access a variable that is not within the scope of the current routine, you must qualify the variable name with a scope path. Chapter 13 explains scope paths in detail.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 187.

Figure 187
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Process settings

8

Process settings control the environment in which the process runs. CXdb allows you to manipulate many of the process settings, such as the floating point mode.

This chapter details how to change the process settings and the default process settings. The commands covered are:

- `add environment`
- `clear environment`
- `clear fixed sched`
- `clear sqs`
- `clear seq`
- `clear step`
- `info environment`
- `info default environment`
- `info formatting`
- `remove environment`
- `set default environment`
- `set default step`
- `set directory`
- `set environment`
- `set fixed sched`
- `set format`
- `set memory`
- `set pshell`
- `set step`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 188.

Figure 188
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 188 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 189.

Figure 189
Starting the example program

```
(CXdb) break line chapter8.f:22

#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80003062] CHAPTER8 in chapter8.f line 22
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80003062] CHAPTER8 in chapter8.f line 22
```

The `break line` command in Figure 189 sets a breakpoint at line 22 of the routine `CHAPTER8`. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

Process settings

When you use a `debug exec` command (or a `debug proc` or `debug core` command), a process object is created. The process object holds all of the information about the current program being debugged. It holds the process settings as well as the executable file, information from the compiler-debugger interface (CDI) data files, and the process image.

The process settings control various aspects of a process, such as the floating point mode used during process execution or the step size used when stepping.

There are 11 types of process settings. They are briefly described below:

- **Process directory**—The directory from which the process is run. By default, the process is run in the console working directory (the current directory).
- **Process environment**—The set of environment variables passed to the process. By default the environment is a copy of the environment passed to CXdb when it was invoked.
- **Floating point mode**—The floating point mode used by the process (IEEE, native, or dual). By default the floating point mode is dual.
- **Display formats**—The format used when displaying process memory. Initially, there are no default display formats. Instead, CXdb attempts to display process memory in the format most appropriate for the data type being displayed.
- **Memory unit**—The memory unit used when displaying process memory. Initially, there is no default memory unit. Instead, CXdb attempts to display process memory in the memory unit most appropriate for the data being displayed.
- **Fixed scheduling**—A system directive that prevents the process from starting execution until all CPUs are made available. By default, fixed scheduling is disabled.
- **Process shell**—The type of shell in which the process is run. By default the process shell is either the C shell or Bourne shell (depending on the type of shell from which CXdb was invoked). This is the shell that interprets the arguments passed to the process.
- **Search path**—The list of directories to search when CXdb needs the source file, or CDI data file, corresponding to the executing code. By default, the search path consists of the console working directory and the directory of the executable file.
- **Stepping granularity**—The source unit granularity to use when a granularity is not specified with a stepping command. The default granularity is statement.
- **SEQ and SQS bits**—Bits of the PSW that control pipelining and memory stores for the process. By default, both the SEQ and SQS bits are enabled.
- **Signal actions**—The actions for handling signals sent to the process. The actions can be changed for each signal.

You can change these process settings to suit your debugging needs.

The process directory, process environment, and process shell settings are passed to a process only when it is created. If you change one of these 4 process settings while a process exists, the change has no effect on the existing process. However, if you then create a new process using the `run` or `rerun` command, the change is passed to the new process.

Default process settings

Most process settings have a correlating default process setting. The default process settings are passed to the process object when the process object is created.

The following is a list of available default process settings:

- Default process environment
- Default fixed scheduling
- Default format
- Default memory
- Default floating point mode
- Default search path
- Default pshell
- Default step

Note

Changes made to the default process settings have no effect on the process settings of a process object that has already been created, except in the case of the `clear step` command.

In the rest of this chapter, the focus of the examples is on the commands that affect the process settings local to the process object. Unless noted otherwise, inserting `default` after the first word of the command causes the command to affect the default process setting. For example, `set step` versus `set default step`. The action the command takes is the same, except that the default process setting (which is only passed to new process objects) is changed instead.

Process directory

The process directory is the directory from which the process is run. Relative path names used in your program are relative to the process directory. Initially, the process directory is the same as the console working directory (the directory CXdb is currently operating from).

You can display the current process directory using the info process command. If your program must be run from a particular directory, change the process directory using the set directory command, as shown in Figure 190.

Figure 190
Setting the process directory

```
(CXdb) set directory /usr/lib/cxdb
(CXdb) run
Process [#0] is already running with pid 21459.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80003062] CHAPTER8 in chapter8.f line 22
```

In Figure 190, the process directory for the process object is set for the /usr/lib/cxdb directory using the set directory command. The run command creates a new process, which receives the new process directory. In the process interface window, the program outputs the name of the directory in which it is running.

Process environment

The process environment is the collection of environment variables passed to the process when the process is created. Environment variables hold string constants that can be used by programs.

Initially, the process object does not have a process environment. Instead, the default process environment is passed to the process. The default process environment is a copy of the set of environment variables passed to CXdb when it is invoked.

The process environment for the process object is created only after you issue one of the following 4 commands:

- add environment—Adds variables to the process environment.
- clear environment—Clears the environment of all variables.
- remove environment—Removes variables from the process environment.
- set environment—Sets the process environment to consist of only the specified variables.

If a process environment does not exist when one of the above commands is issued, a copy of the default process environment is given to the process object, and then the appropriate changes are made according to the command issued. If the process environment does exist, the changes are made directly to it.

You can get a list of the variables in the current default environment using the `info default environment` command, as shown in Figure 191.

Figure 191
Getting information about the default process environment

```
(CXdb) info default environment
Default environment:
HOME=/usr/jones
SHELL=/bin/csh
EDITOR=vi
MAIL=/usr/spool/mail/jones
LESS=-MQce
MORE=-c
PAGER=less
RNINIT=/usr/jones/.rninit
```

The list of environment variables shown in Figure 191 is an example of some of the variables that might exist in the default environment. This is the environment that was passed to CXdb when it was invoked. Most likely your default environment consists of more variables.

In Figure 192, the `info environment` command displays the process environment, which is the same as the default environment in this case.

Figure 192
Getting information about the process environment

```
(CXdb) info environment
Process [#0] environment: (from default environment)
HOME=/usr/jones
SHELL=/bin/csh
EDITOR=vi
MAIL=/usr/spool/mail/jones
LESS=-MQce
MORE=-c
PAGER=less
RNINIT=/usr/jones/.rninit
```

The process environment is cleared using the `clear environment` command, as demonstrated in Figure 193.

Figure 193
Creating the process environment

```
(CXdb) clear environment
(CXdb) info environment
Process [#0] environment:
```

The `clear environment` command does two things. First, it creates a process environment. Because the environment did not yet exist, it was created so you can change it as needed. Second, the `clear environment` command cleared the environment of all variables, thus leaving it empty, as shown with the `info environment` command.

You can set the environment variables using the `add environment` command, as shown in Figure 194.

Figure 194
Setting the process environment

```
(CXdb) add environment CHAPTER = 8
(CXdb) info environment
Process [#0] environment:
CHAPTER=8
(CXdb) run
Process [#0] is already running with pid 1155.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80003062] CHAPTER8 in chapter8.f line 22
```

The `add environment` command in Figure 194 adds the variable `CHAPTER` to the process environment and sets its value to the string `8`. The `info environment` command shows the new process environment consists of the single environment variable `CHAPTER`. The `run` command creates a new process that is passed the new process environment. The process interface window shows the program has found the correct value for the variable `CHAPTER`, but the variables `DEBUGGER` and `FLAGS` still do not exist.

Figure 195 demonstrates how to add multiple environment variables to the process environment.

Figure 195
Adding multiple variables to the process environment

```
(CXdb) add environment DEBUGGER = cxdb , FLAGS = "-e a.out"
(CXdb) info environment
Process [#0] environment:
CHAPTER=8
DEBUGGER=cxdb
FLAGS=-e a.out
(CXdb) run
Process [#0] is already running with pid 1225.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80003062] CHAPTER8 in chapter8.f line 22
```

The `add environment` command in Figure 195 adds 2 environment variables to the process environment. A comma is used to separate the variables. The string for the variable `FLAGS` is enclosed in quotes because it contains a space character.

The `info environment` command is again used to show the new process environment. The `run` command creates a new process. The process interface window shows the process now has the correct values for all 3 variables.

You can also remove variables from the process environment using the `remove environment` command, as shown in Figure 196.

Figure 196
Removing process environment variables

```
(CXdb) remove environment FLAGS
(CXdb) info environment
Process [#0] environment:
CHAPTER=8
DEBUGGER=cxdb
(CXdb) run
Process [#0] is already running with pid 1313.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80003062] CHAPTER8 in chapter8.f line 22
```

The `remove environment` command removes the environment variable `FLAGS` from the process environment. The `info environment` command shows the variable has been removed. When a new process is created using the `run` command, the process cannot find the missing variable.

The final method of changing the process environment uses the `set environment` command. The `set environment` command removes all variables from the process environment and then adds the specified variables. It is the equivalent of a `clear environment` command followed by an `add environment` command. This is demonstrated in Figure 197.

Figure 197
Setting the process environment

```
(CXdb) set environment DEBUGGER = "cxdb -e a.out", FLAGS = -nw
(CXdb) info environment
Process [#0] environment:
DEBUGGER=cxdb -e a.out
FLAGS=-nw
(CXdb) run
Process [#0] is already running with pid 1338.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80003062] CHAPTER8 in chapter8.f line 22
```

Figure 197 uses the `set environment` command to set the process environment to consist of the variables `DEBUGGER` and `FLAGS`. Note that the environment variable `CHAPTER` no longer exists in the environment, as shown by the `info environment` command.

Floating point mode

The floating point mode, or fpmode, determines the method used by the process when it performs a floating point calculation. The default fpmode (which is passed to the process object's fpmode) can be any of the following:

- **IEEE**—IEEE floating point mode
- **native**—Native floating point mode
- **dual**—either IEEE or native mode, determined by the current frame of execution.

Initially, the default fpmode is dual, which allows the process to decide the mode used. You can override the fpmode used by the process by changing the fpmode process setting with the `set fpmode` command.

The fpmode for the process can be set to IEEE or native mode, but not dual. The current setting of the fpmode and default fpmode can be displayed using the `info process` command and `info cxdb` command, respectively.

Note

As soon as you issue the `set fpmode` command, the process begins using the new floating point mode. The process continues to use that mode until it completes execution or it explicitly changes the floating point mode.

Display formats

A display format determines the format a segment of memory of a particular size, called a memory unit, is displayed in. Each memory unit can have a display format associated with it. For example, by using display formats, memory examined as bytes can be displayed as characters, while memory examined as longwords can be displayed using scientific notation. If a display format has not been set for a particular memory unit, CXdb displays the values as hexadecimal numbers.

Display formats are set using the `set format` command. Display formats affect only memory displayed with the `examine` command.

The available memory units are:

- `byte`—8 bits
- `halfword`—16 bits
- `word`—32 bits
- `longword`—64 bits
- `quadword`—128 bits

The available display formats are:

- `binary`
- `character`
- `complex` (FORTRAN)
- `decimal`
- `eformat` (scientific notation)
- `fformat` (floating point notation)
- `logical` (FORTRAN)
- `hexadecimal`
- `octal`
- `unsigned` (decimal)

Not all display formats are available for the various memory units. Table 6 lists the possible display formats for each memory unit. A black dot (●) represents a valid format for the given memory unit.

Table 6
Possible formats for the different memory units

Display format	Memory unit type				
	byte	halfword	word	longword	quadword
binary	●	●	●	●	●
character	●				
complex				●	●
decimal	●	●	●	●	
eformat (scientific notation)			●	●	●
fformat (floating point)			●	●	●
hexadecimal	●	●	●	●	●
logical	●	●	●	●	●
octal	●	●	●	●	●
unsigned (decimal)	●	●	●	●	

Figure 198 uses the `set format` command to specify a display format for the current process.

Figure 198
Specifying a format for a memory unit

```
(CXdb) examine ARRAY
Examine Process [#0/0] from 0x8006dce8 to 0x8006dd34
 8006dce8:     00000001  00000002  00000003  00000004  00000001  00000004
 8006dd00:     00000009  00000010  00000001  00000008  0000001b  00000040
 8006dd18:     00000001  00000010  00000051  00000100  00000005  00000000
 8006dd30:     00000005  00000005
(CXdb) set format word decimal
(CXdb) examine ARRAY
Examine Process [#0/0] from 0x8006dce8 to 0x8006dd34
 8006dce8:             1          2           3           4
 8006dcf8:             1          4           9          16
 8006dd08:             1          8          27          64
 8006dd18:             1         16          81         256
 8006dd28:             5          0           5           5
```

The `set format` command in Figure 198 sets the display format for words to `decimal`. From this point forward, all memory displayed as words is displayed as a decimal value. The `examine` command displays the variable `ARRAY` in word format. Because the display format for words has been set to `decimal`, the values of `ARRAY` are displayed as decimal values.

The `info formatting` command can be used to display the current formats for the process object, as shown in Figure 199.

Figure 199
Getting information on the display formats for the process object

```
(CXdb) info formatting

Global format settings:
              padding: Off
    max array elements: 20
 floating point format: 10.4

Process [#0] format settings:

Format settings for Thread 0
            Selected memory size: (none)
         Selected Memory Formats:
                 byte=(none), halfword=(none), word=decimal
                 longword=(none), quadword=(none)
```

Display format changed

The `info formatting` command lists the current display formats. The display format for words is set to `decimal`, while the remaining formats have still not been set.

Memory unit

When you examine a region of memory using the `examine` command, CXdb attempts to display the information in the most appropriate memory unit type. You can choose to examine the memory using a different memory unit by explicitly specifying a memory unit with the `examine` command. Or, you can select the memory unit for the process object.

You can set the memory unit used to display all process memory using the `set memory` command. Once you select the memory unit, all process memory is displayed in that memory unit. Thus, if you select the memory unit to be bytes, all memory examined is displayed as bytes. You can still override this default by explicitly specifying a memory unit with the `examine` command.

Once a memory unit for a process object has been specified, you cannot return to an unspecified state (though the memory unit can be changed again). If you want to examine a region of memory in a particular memory size, but do not need to set the process setting, you can use the `examine` command to do so, as described in Chapter 7, Section "Examining and modifying memory."

The available memory units are:

- byte—8 bits
- halfword—16 bits
- word—32 bits
- longword—64 bits
- quadword—128 bits

Figure 200 illustrates the use of the set memory command to specify a memory unit for the current process object.

Figure 200
Specifying a memory unit for the current process

```
(CXdb) set memory byte
(CXdb) examine ARRAY
Examine Process [#0/0] from 0x8006dce8 to 0x8006dcfb
 8006dce8:    00  00  00  01  00  00  00  02  00  00  00  03  00  00  00  04
 8006dcf8:    00  00  00  01
(CXdb) info formatting

Global format settings:
               padding: Off
    max array elements: 20
 floating point format: 10.4

Process [#0] format settings:

Format settings for Thread 0
            Selected memory size: byte <----------- Memory size changed
         Selected Memory Formats:
                 byte=(none), halfword=(none), word=decimal
                 longword=(none), quadword=(none)
```

The set memory command in Figure 200 sets the memory unit for the process object to byte. Now, all process memory displayed with the examine command is displayed as bytes. You can use the set format command to further specify the display format for bytes.

The examine command displays the values of the variable named ARRAY. The elements of ARRAY are printed as bytes because all process memory is now displayed as bytes. Finally, the info formatting command displays the current memory unit for the process.

Fixed scheduling

Fixed scheduling ensures all CPUs are made available for the process when execution begins. If you are debugging a process that can have multiple threads, fixed scheduling should be enabled because it makes thread behavior more deterministic. Initially, fixed scheduling is disabled.

Note

Enabling fixed scheduling can increase system overhead. Enabling fixed scheduling only guarantees that all CPUs are made available when execution begins, not that they will be used.

Fixed scheduling is enabled using the `set fixed sched` command and disabled using the `clear fixed sched` command. The current setting of fixed scheduling and default fixed scheduling can be displayed using the `info process` command and `info cxdb` command, respectively.

For more information about fixed scheduling and threads, refer to Chapter 16.

Process shell

When a process is created in CXdb, it runs inside a shell. The process shell is the shell that interprets arguments passed to the process with the `run` command.

The initial setting of the default process shell depends upon the type of shell from which CXdb is invoked. If CXdb is invoked in a C shell (`csh` or `tcsh`), the default process shell type is the C shell. Otherwise, the default process shell type is the Bourne shell.

The process shell is set using the `set pshell` command. You can set the process shell to either the Bourne shell or the C shell. The current setting of the process shell and default process shell can be displayed using the `info process` command and `info cxdb` command, respectively.

Search path

The search path of a process object is a list of directories CXdb searches for the source files and CDI data files of the program being debugged. When process execution stops in a source file other than the current file being displayed, CXdb looks for the corresponding source file in the directories listed in the search path.

The search path for a process object is initially the default search path plus the directory in which the executable was found (if not yet in the search path). The default search path is initially the console working directory.

Commands that manipulate the search path are listed below:

- `add path`—Add directories to the search path.
- `remove path`—Remove directories from the search path.
- `set path`—Set the search path to the specified directories.

The above commands operate in much the same way as the commands that affect the process environment (covered earlier in this chapter). For a full description of the use of search paths, refer to Chapter 9.

Stepping granularity

You can specify a default granularity for the process object using the `set step` command. When this is done, stepping commands use the new default granularity, unless a granularity is specified at the end of the stepping command. The default stepping granularity is statement.

Figure 201 shows the effect of changing the stepping granularity to `expression` (refer to Chapter 6 for a complete discussion of stepping commands and granularity).

Figure 201
Changing the default stepping granularity

```
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x8000306c] CHAPTER8 in chapter8.f line 23
(CXdb) set step expression
(CXdb) info process

status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 10521, state = stopped
     working dir: /usr/lib/cxdb
    default step: expression  <------------------ Step size changed
(CXdb) step
Stepping process [#0/*] by 1 expression
Process [#0/0] stopped nexting at [0x80003072] CHAPTER8 in chapter8.f line 23
(CXdb) step
Stepping process [#0/*] by 1 expression
Process [#0/0] stopped nexting at [0x80003078] CHAPTER8 in chapter8.f line 23
```

The `set step` command in Figure 201 sets the granularity to `expression`. Subsequent `step` commands operate on expressions, rather than statements, because the default granularity for the process object has changed. The `info process` command displays the default stepping granularity for the process object.

You can clear the default step for the process object by using the `clear step` command. The `clear step` command resets the stepping granularity to the current value of the default stepping granularity (one of the default process settings). If you then change the default stepping granularity using the `set default step` command, the process object's stepping granularity tracks the new value of the default.

Figure 202 shows how the `clear step` command is used to reset the stepping granularity of the process object to the default stepping granularity.

Figure 202
Resetting the stepping granularity

```
(CXdb) clear step
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x8000306c] CHAPTER8 in chapter8.f line 23
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x8000306c] CHAPTER8 in chapter8.f line 23
(CXdb) info process
status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 10521, state = stopped
     working dir: /usr/lib/cxdb
    default step: statement  <-------- Step size reset to default size of statement
```

The `clear step` command in Figure 202 resets the stepping granularity for the process object to `statement`. The `step` commands step the process by statements, rather than by expressions. The `info process` command displays the new stepping granularity.

SEQ bit

The sequential mode (SEQ) bit of the process status word (PSW) controls pipelining within the processor. If this bit is set, the processor executes all instructions sequentially; that is, the execution of the next instruction is initiated only after the previous instruction has been executed. If this bit is clear, the processor operates with the maximum pipelining and overlap.

The default is SEQ set. You can clear the SEQ bit using the `clear seq` command and enable it again using the `set seq` command. The `info psw` command displays the setting of the SEQ bit.

For more information about the PSW and the SEQ bit, refer to the *CONVEX Architecture Reference*, Chapter 3, "Register Sets."

SQS bit

The sequential store enable (SQS) bit of the process status word (PSW) controls sequential stores to memory.

If the SQS bit is set, all stores to memory occur in the same order as instruction execution. If this bit is clear, stores to memory can occur in nonsequential order.

By default, the SQS bit is set. You can clear the SQS bit with the `clear sqs` command and set it again using the `set sqs` command. The `info psw` command displays the setting of the SQS bit.

For more information about the PSW and the SQS bit, refer to the *CONVEX Architecture Reference*, Chapter 3, "Register Sets."

Signal actions

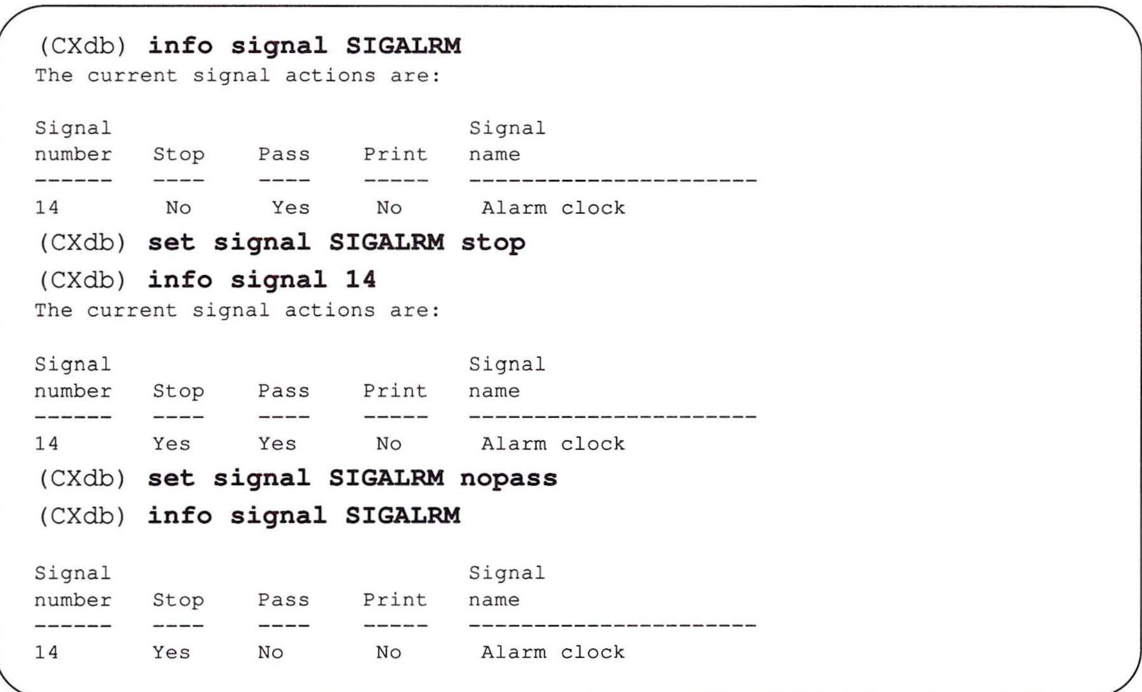
The actions CXdb takes when it catches a particular signal can be set. These signal actions are set using the `set signal` command.

The three signal actions are described below:

- `stop`—If the signal is caught, the process is stopped, and the value for the caught signal is stored in the debugger variable `$signal`.
- `print`—If the signal is caught, the value of `$signal` and the corresponding signal type is displayed in the command window.
- `pass`—If process execution is continued, the value in the `$signal` debugger variable is passed to the process.

Each action can be set and unset for each signal using the `set signal` command, as shown in Figure 203. To unset a signal action, precede the signal action with the string `no` (for example, `nostop`, `nopass`, `noprint`).

Figure 203
Setting the signal actions for the `SIGALRM` signal



```
(CXdb) info signal SIGALRM
The current signal actions are:

Signal                              Signal
number   Stop    Pass    Print    name
------   ----    ----    -----    ----------------------
14        No      Yes     No       Alarm clock
(CXdb) set signal SIGALRM stop
(CXdb) info signal 14
The current signal actions are:

Signal                              Signal
number   Stop    Pass    Print    name
------   ----    ----    -----    ----------------------
14       Yes     Yes      No       Alarm clock
(CXdb) set signal SIGALRM nopass
(CXdb) info signal SIGALRM

Signal                              Signal
number   Stop    Pass    Print    name
------   ----    ----    -----    ----------------------
14       Yes     No       No       Alarm clock
```

In Figure 203, the default actions for the `SIGALRM` signal are displayed using the `info signal` command. The `stop` action is initially unset. The first `set signal` command sets the `stop` action for the `SIGALRM` signal. The second `info signal` command displays the new action settings for the `SIGALRM` signal using its signal number, 2. The second `set signal` command unsets the `pass` action for the `SIGALRM` signal. The last `info signal` command again displays the new settings for the `SIGALRM` signal.

For more information about changing the default actions for signals, refer to the "Signals" section in Chapter 6.

Origin of process setting values

The initial value for process settings is different for each setting. Figure 205 and Figure 204 are flowcharts that demonstrate where the values for the process settings originate. There are two distinct groups of process settings and default process settings: those that affect the current process and those that affect only new processes. Figure 204 displays the first group and Figure 205 displays the second.

In each diagram, a shaded oval represents the default process setting and an unshaded oval represents the process setting for the process object.

Figure 204
Defaults for process settings that affect the current process

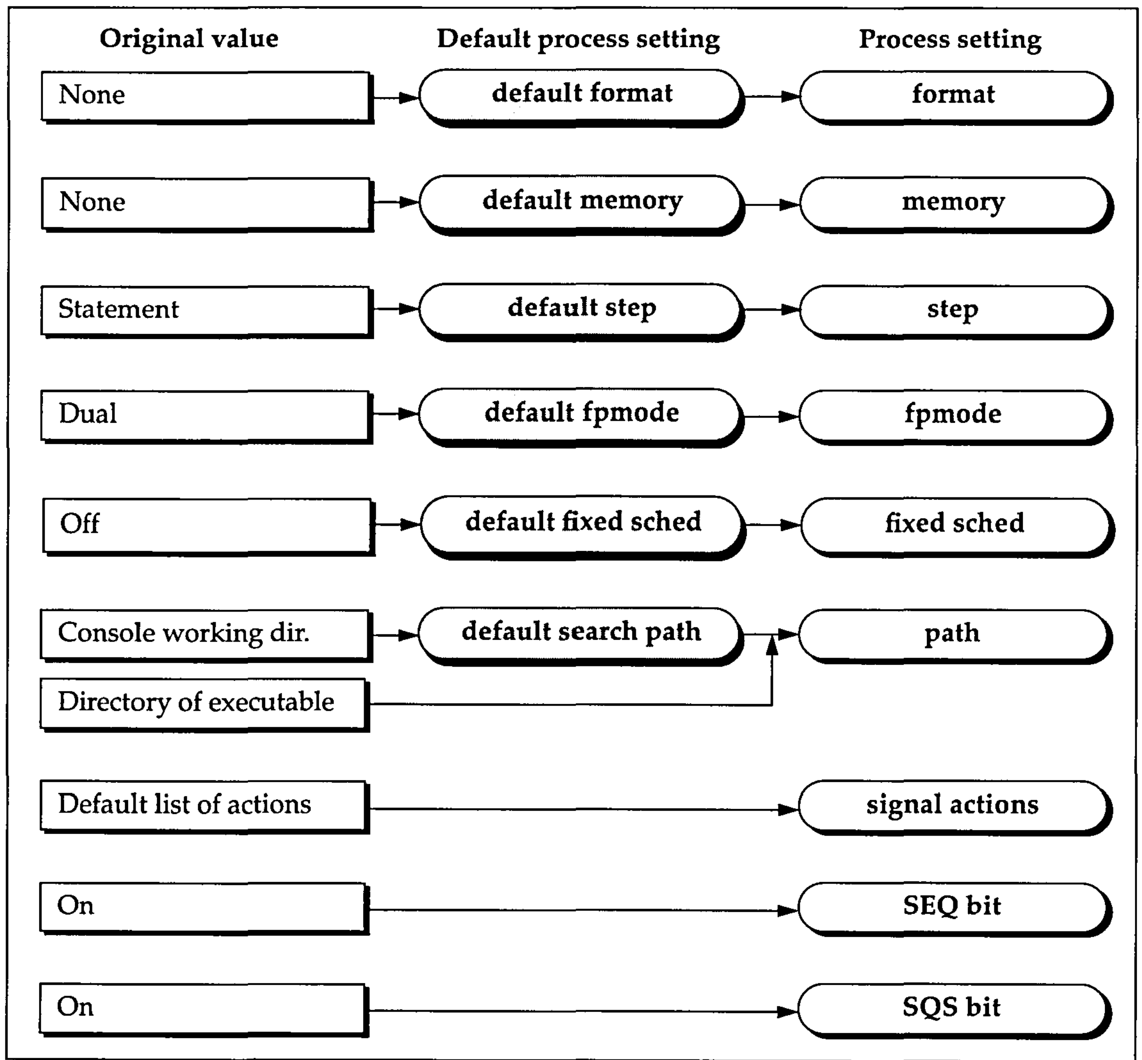
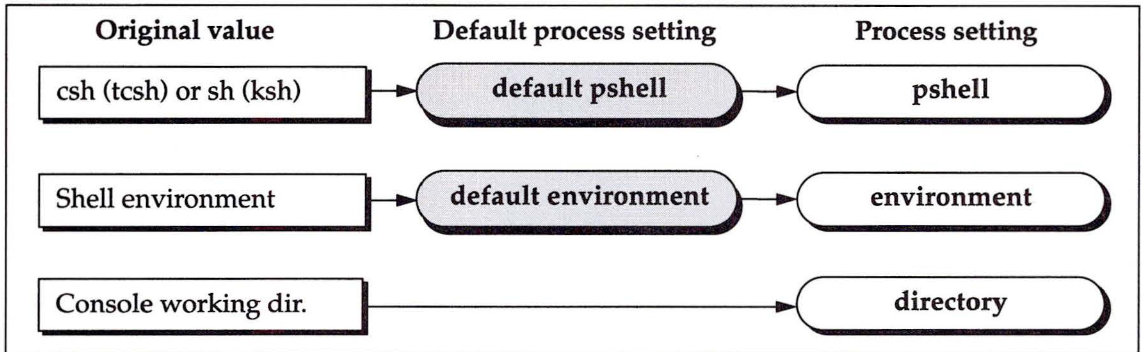


Figure 205
Defaults for process settings that affect only new processes



Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 206.

Figure 206
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Debugging with multiple source files

CXdb has commands that make it easier to debug large programs consisting of multiple source files that may be spread out over several directories. You can use these commands to manipulate the search path, open multiple source windows, and search source windows.

The commands covered in this chapter are:

- `add default path`
- `add path`
- `cd`
- `display routine`
- `display file`
- `find window forward`
- `find window backward`
- `pwd`
- `remove default`
- `remove path`
- `set default path`
- `set path`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 207.

Figure 207
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb
```

The `cd` command in Figure 207 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb.

Console working directory

The search path of the process object defaults to the current console working directory. The console working directory is the current directory CXdb is operating from. Initially, the console working directory is set to the directory from which you invoked CXdb.

Many commands use the console working directory to find files. If a relative path name (a path name that does not begin with a slash or a tilde) is used in any of the following commands, the relative path name is based from the console working directory:

- `add cmderr`
- `add cmdlog`
- `add cmdout`
- `add default path`
- `add path`
- `cd`
- `edit`
- `remove cmderr`
- `remove cmdlog`
- `remove cmdout`
- `remove default path`
- `remove path`
- `set cmderr`
- `set cmdlog`

- `set cmdout`
- `set default path`
- `set directory`
- `set path`

You can change the console working directory by using the `cd` command. You can display the current setting of the console working directory by using the `pwd` command. Both of these commands work much like their operating system counterparts. These two commands are demonstrated in Figure 208.

Figure 208
Changing directories within CXdb

```
(CXdb) pwd
/usr/lib/cxdb/examples
(CXdb) cd example2
(CXdb) pwd
/usr/lib/cxdb/examples/example2
```

In Figure 208, the `pwd` command displays the current console working directory. Initially, this is the directory from which CXdb was invoked. The `cd` command changes the console working directory to the `example2` directory. Note that because the `example2` directory was not given an absolute pathname, it was found using the current console working directory as a base path name.

Search path

Large programs often have multiple source files. These source files can all be in the same directory, as with the example program debugged in preceding chapters, or can be located in different directories.

CXdb uses two search paths to locate files:

- **Default search path**—Directories searched for executable files, core files, and command files.
- **Search path**—Directories searched for source files and compiler-generated data files.

The default search path is initially set to track the console working directory. If the console working directory changes (through the use of the `cd` command), the default search path changes with it. The following commands use the default search path to find files specified with relative path names:

- `core`
- `debug core`
- `debug exec`
- `executable`
- `source`

The search path of the process object is used to locate source files and compiler-generated data files. When the process object is created, the search path defaults to the default search path plus the directory in which the executable file is found (if not in the default search path). The following commands use the search path to locate files:

- `break line`
- `display file`
- `display routine`
- `event reached line`
- `trace line`

You can manipulate the default search path and process object search path by using the following commands:

- `add default path`—Adds directories to the default search path.
- `add path`—Adds directories to the search path.
- `remove default path`—Removes directories from the default search path.
- `remove path`—Removes directories from the search path.
- `set default path`—Sets the default search path to the specified directories.
- `set path`—Sets the search path to the specified directories.

It is important to properly set up the search path if the program's source files are not in the directory of the executable file. If you do not, CXdb will not be able to find the appropriate source files.

Displaying the default search path

You can display the search path by using the `info process` command and the default search path by using the `info cxdb` command. Figure 209 shows how to display the default search path.

Figure 209
Displaying the default search path

```
(CXdb) info cxdb

Current CXdb state:

     ENVIRONMENT:
              pid: 14315
              cwd: /usr/lib/cxdb/examples/example2
   command modes: echo off, logging off, noclobber off
           cmdout: Window #1
           cmderr: Window #1
           cmdlog:
         evalopts: fpmode = dual, iprecision = 4, rprecision = 4
            shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
       step size: statement
   process shell: csh
          fpmode: dual
     memory size: (none)
  memory formats: byte=(none), halfword=(none), word=(none)
                   longword=(none), quadword=(none)
      search path:
          .  <------------------------------- Default search path
```

The `info cxdb` command in Figure 209 displays the current settings for CXdb. The default search path is set to track the current console working directory (represented by a dot). As long as the dot is part of the search path, the search path always includes the current console working directory.

The default search path is passed to the process object when the process object is created. It is usually best to change the default search path so that CXdb can find the source files and CDI data files corresponding to the executable file. If you do not, CXdb generates an error message if it cannot find a needed file. If this type of error message is displayed, you can add the name of the directory where the file is located to the search path, then reissue the command that generated the error message.

Changing the default search path

You can add directories to the default search path with the `add default path` command, as shown in Figure 210.

Figure 210
Setting the default search path

```
(CXdb) add default path sourcetree1, sourcetree2, sourcetree3
Default search path:
        .
        sourcetree1
        sourcetree2
        sourcetree3

(CXdb) debug exec a.out

Default source file: sourcetree1/main.f
Default source language: Fortran

Process [#0] created
```

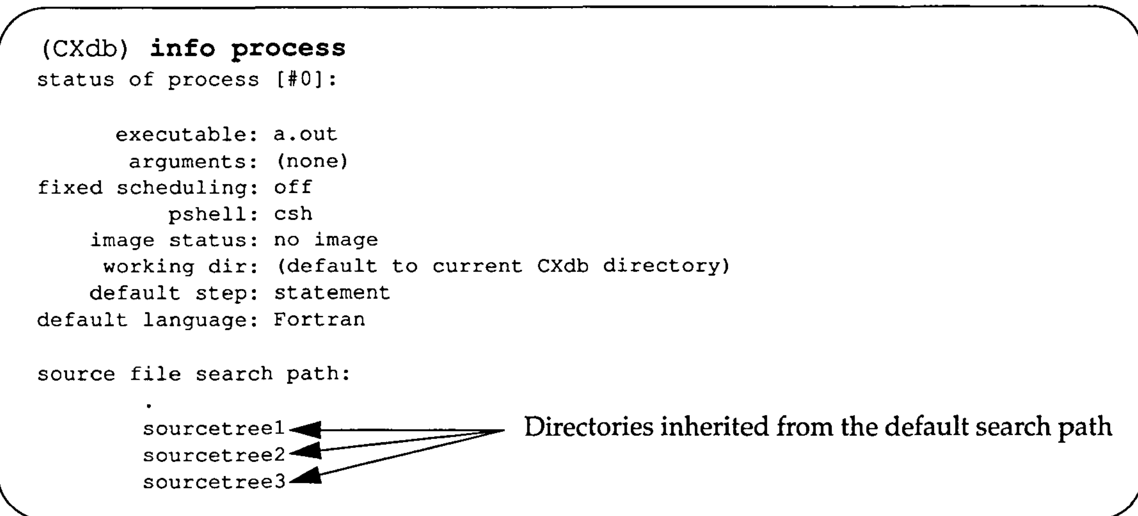
The `add default path` command adds the directories `sourcetree1`, `sourcetree2`, and `sourcetree3` to the default search path. These directories are appended to the current setting of the default path, so the console working directory remains in the search path.

The `debug exec` command creates a new process object. The process object is passed the default process settings, so it receives the new default search path. CXdb finds the default source file `main.f` in the `sourcetree1` directory, and opens the source window.

Displaying the search path

You can display the search path of the process object with the `info process` command, as shown in Figure 211.

Figure 211
Displaying the search path of the process object

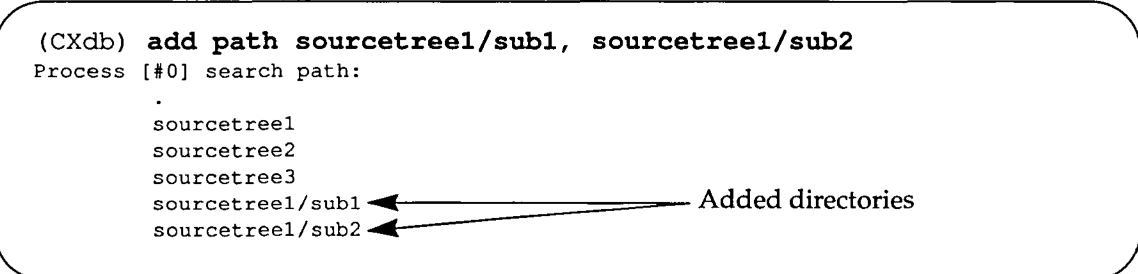


The `info process` command in Figure 211 displays the process settings for the process object. The search path consists of the default search path, plus the directory in which the executable file is located.

Adding directories to the search path

You can add directories to the search path of the process object by using the `add path` command, as illustrated in Figure 212.

Figure 212
Adding directories to the search path



In Figure 212, the `add path` command adds the directories `sourcetree/sub1` and `sourcetree1/sub2` to the search path of the process object. CXdb then displays the new search path for the process object.

Because you usually debug from within the directory containing the executable file, it is a good idea to keep the console working directory in the search path. The `set path` command removes all directories from the search path and then sets the search path to the specified directories. By using the `add path` commands rather than the `set path` commands, you can add the additional directories to the search path without removing the initial setting of the console working directory.

Removing directories from the search path

There may be times when you no longer need a directory in the search path. You can remove one or more directories from the search path by using the `remove path` command (or `remove default path` for the default search path), as shown in Figure 213.

Figure 213
Removing directories from the search path

```
(CXdb) remove path sourcetree1/sub2
Process [#0] search path:
         .
         sourcetree1
         sourcetree2
         sourcetree3
         sourcetree1/sub1
```

The `remove path` command in Figure 213 removes the `sub2` directory from the search path. CXdb no longer will look in the `sub2` directory to find source files or corresponding data files.

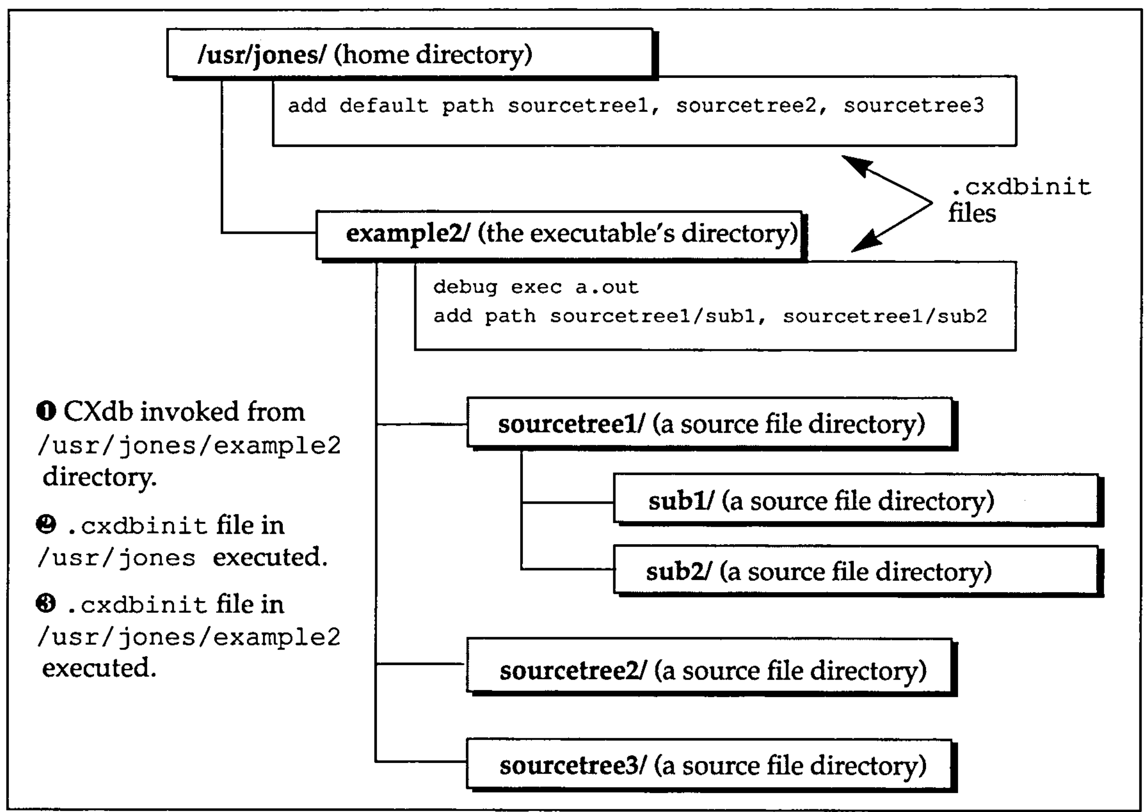
Initialization files and search paths

You can use CXdb initialization files to establish the search paths you need, rather than having to enter the appropriate commands each time you plan to use CXdb. Initialization files allow you to execute CXdb commands automatically, each time CXdb is invoked.

Figure 214 shows an example of the initialization files that might be used to automate the setting up of the search path in the same way as the preceeding examples. In Figure 214, the `.cxdbinit` file in the `/usr/lib/cxdb` directory, which is executed before any other initialization files, is not included because this typically would not be used to establish search paths.

Refer to Chapter 12, Section "Using initialization files," for more information on initialization files.

Figure 214
Using initialization files to set up search paths



When CXdb is invoked, the `.cxdbinit` file in your home directory is executed. Then, the `.cxdbinit` file in the current directory is executed. From Figure 214, the resulting search paths are as follows:

- Default search path
  - `.` (the current console working directory)
  - `sourcetree1`
  - `sourcetree2`
  - `sourcetree3`
- Search path
  - `.` (the current console working directory)
  - `sourcetree1`
  - `sourcetree2`
  - `sourcetree3`
  - `sourcetree1/sub1`
  - `sourcetree1/sub2`

Initialization files can be used to control other process settings. They can also be used to execute other command files, establish aliases, or even begin debugging. In the above example, the `debug exec` command creates a process object. The `add path` command affects the search path of the new process object.

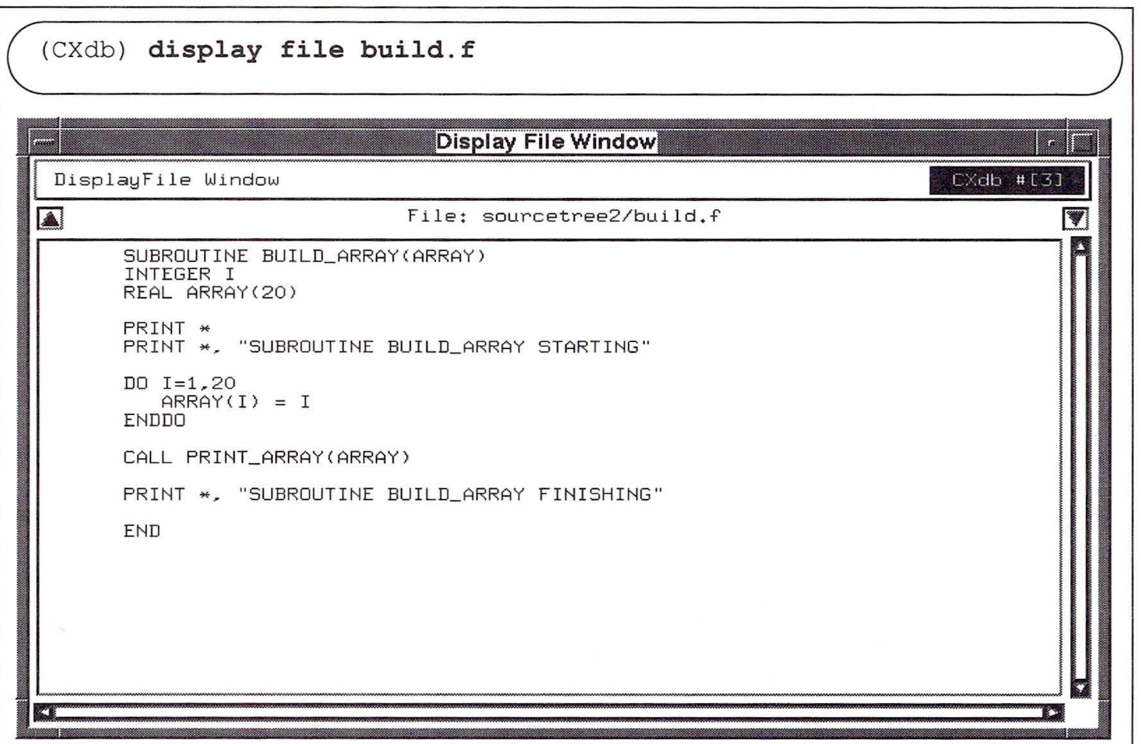
Displaying a file

When you are working with multiple source files, it may be necessary to view the source code of different source files at the same time. With CXdb, you can open multiple display file windows to display the contents of any ASCII text file.

The `display file` command opens a display file window containing the text of the file specified. If the file you want to display is in the search path, you do not need to specify a path name. To display a file outside of the process object's search path, you must specify a path name. You can have multiple display file windows open at once.

Figure 215 shows the `display file` command and the resulting window as seen in CXwindows.

Figure 215
Displaying a file in the display file window (in CXwindows)



In CXwindows, the display file window is a new window. The DisplayFile Window menu allows you to select another file to display in the same window.

In Maryland Windows, the display file window is another subdivision of the screen. This window can be scrolled just as any other window in Maryland Windows.

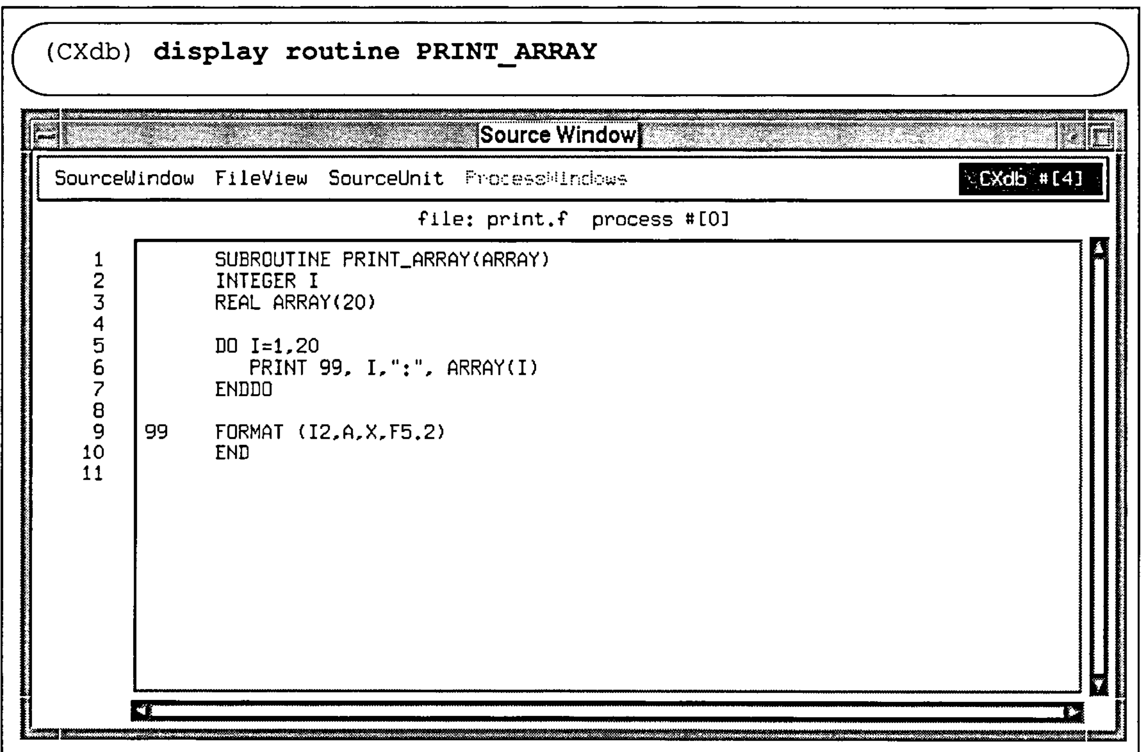
Displaying a routine

Just as you can display a file in CXdb, you can also display a routine. With the `display routine` command, the routine is displayed in a new source window. You can open as many source windows as necessary.

The advantage of displaying routines inside the source window is that the source window displays line numbers for the source file. Also, in the CXwindows interface, you can use the source window to place eventpoints and select source code text.

In Figure 216, the `display routine` command opens a new source window that displays the `PRINT_ARRAY` routine. The line numbers for the routine are displayed.

Figure 216
Displaying a routine in the source window (CXwindows version)



Searching source code

CXdb provides two commands to help you locate a particular piece of source code. The `find window forward` and `find window backward` commands search the text in a particular source window for a specified string. You specify the source window you want to search by giving CXdb the window number for that window.

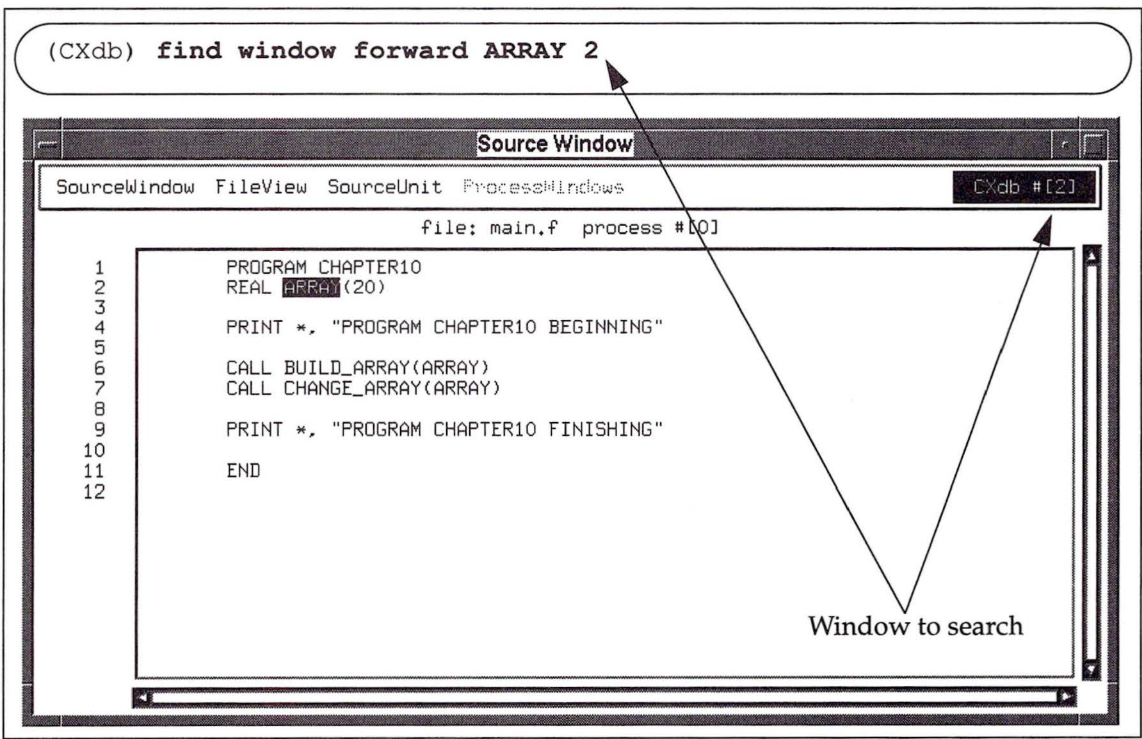
In CXwindows, window numbers are displayed in square brackets in the upper right-hand corner of the window. In Maryland Windows, window numbers are displayed after the window's title in the title bar.

Searching forward in the source window

The `find window forward` command searches forward through the source window. The first search in a window starts from the beginning of the file. Subsequent searches start from where the last search left off. If CXdb does not find the string, a message is displayed. The search does not wrap.

Figure 217 illustrates the `find window forward` command.

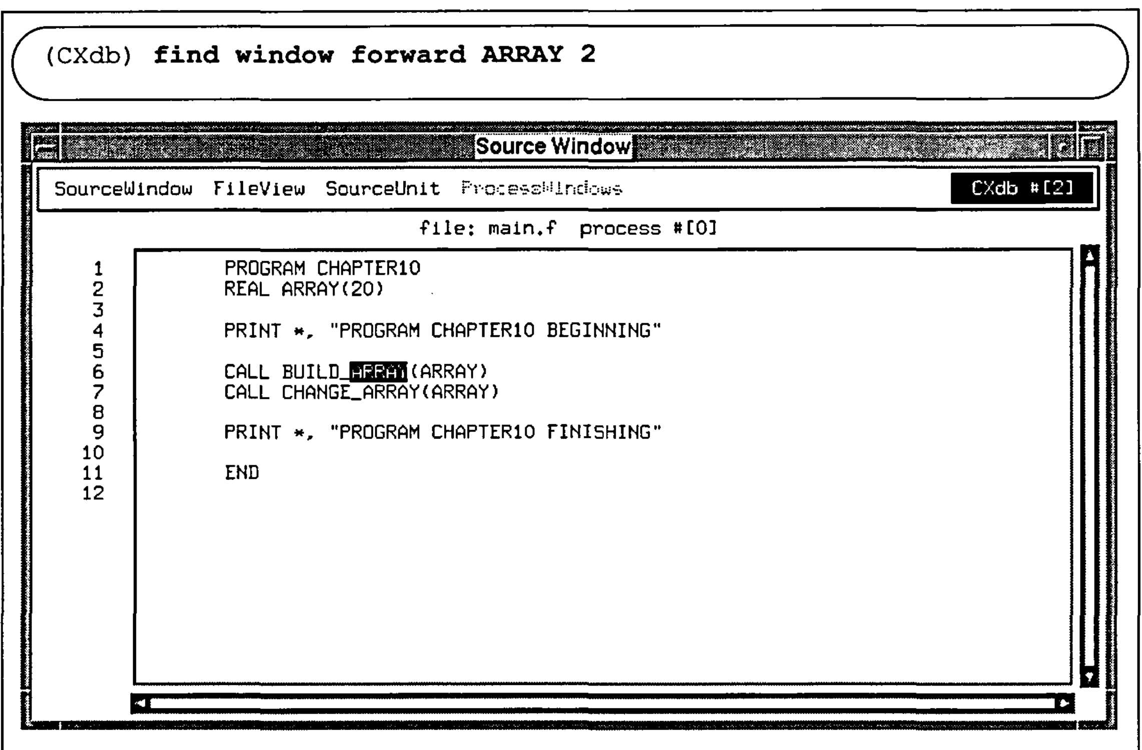
Figure 217
Searching forward through source window 2 for a string



The `find window forward` command in Figure 217 searches for the string ARRAY from the beginning of the file displayed in window 2. CXdb highlights the first occurrence of the string in the window.

Subsequent searches in this window take place from this location, as shown in Figure 218.

Figure 218
Continuing to search forward through the source window



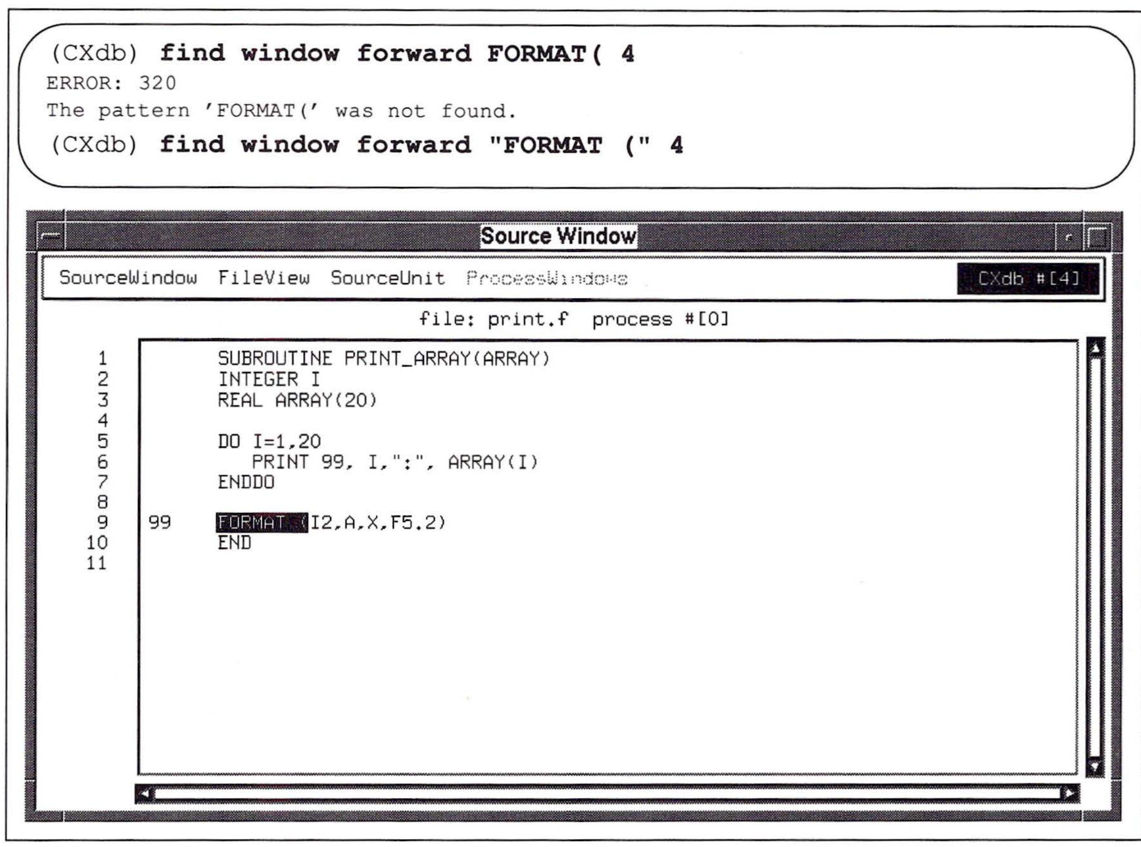
In Figure 218, the `find window forward` command searches forward through window 2 for the next occurrence of the string ARRAY. CXdb highlights the string when it is found.

Searching in a different source window

Any source window can be searched with the `find window forward` and `find window backward` commands. Multiple source windows can be opened with the `display routine` command.

Figure 219 shows how to search source window 4, which was opened by the `display routine` command in Figure 216.

Figure 219
Searching a different source window

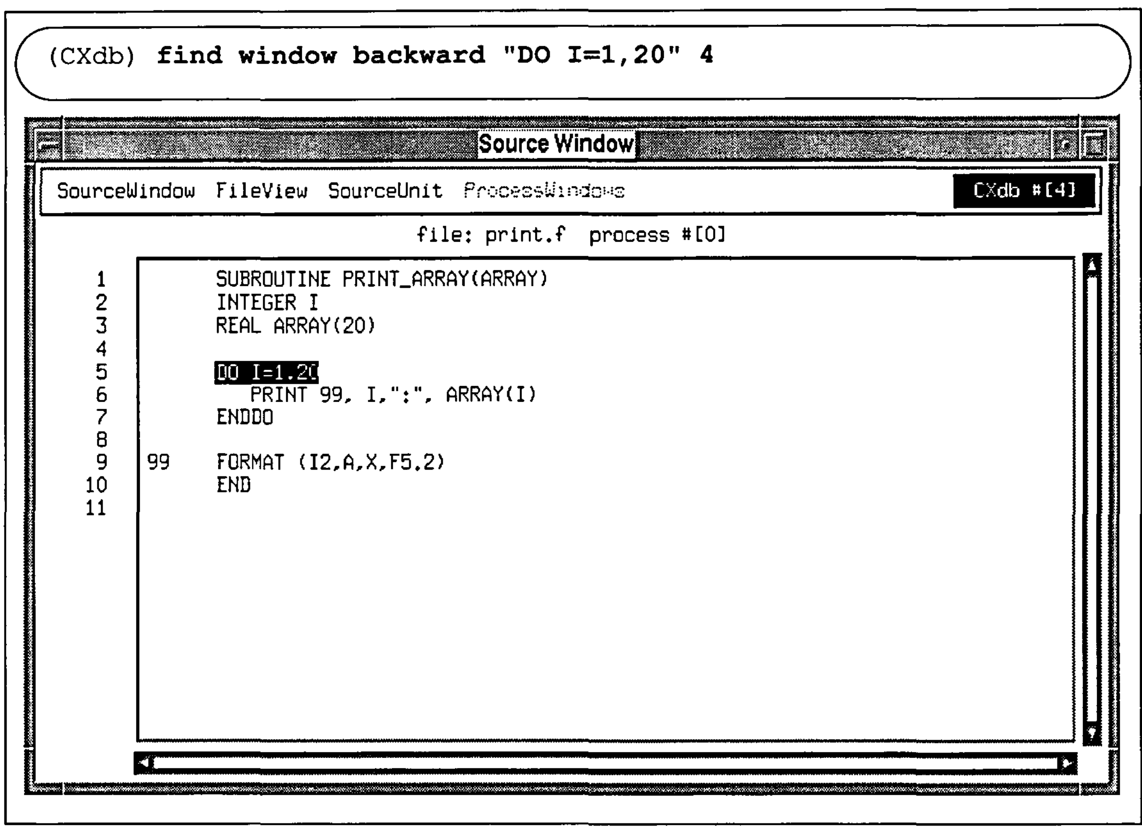


The first `find window forward` command in Figure 219 searches for the string `FORMAT(`, which does not exist in source window 4. CXdb displays a message indicating that it did not find the string. The second `find window forward` command uses double quotes because the string contains a space. CXdb highlights the string when it is found.

Searching backward in the source window

You can search backward through the text in the source window by using the `find window backward` command. The search begins where the last search left off, or from the end of the file if this is the first search. You tell CXdb the window you want to search by specifying the window number of the window, as shown in Figure 220.

Figure 220
Searching backward through the source window for a string



The `find window backward` command in Figure 220 searches window 4 backward from the last search to find the string `"DO I=1,20"`. CXdb highlights the string when it is found.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 221.

Figure 221
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Specifying the program to debug

10

There are several methods for specifying the process to debug. You can debug a new process inside of CXdb, a process that is already running, or a core file. You can also specify a new executable file if you wish to debug a different program.

This chapter covers the methods of specifying a program to debug. It also explains how to execute shell commands from within CXdb.

The following commands are covered:

- `attach`
- `cd`
- `core`
- `debug core`
- `debug exec`
- `debug proc`
- `detach`
- `executable`
- `shell`

Note

Many of the examples in this chapter assume the shell being used can place processes in the background. If you are using the Bourne shell (sh) to execute the examples, you will be unable to complete the examples shown in the Section "Attached processes."

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 222.

Figure 222
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb &
```

The `cd` command in Figure 222 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb. The `&` places its execution in the background.

Specifying a process to debug

To debug your program, you must specify the executable file and a process generated from that executable file. CXdb uses the image of the process to map between the current state of the process and the original source code.

There are 3 ways of specifying a process to debug in CXdb. You can create a new process by running the executable file from within CXdb. This is the method used in the examples in the preceding chapters. You can also attach to a process that is already running outside of CXdb. Or, you can specify a core file.

You can specify the executable file or the process in either order. The first one you specify, however, must be specified using one of the following commands:

- `debug core`—Debug a core file (specifies a core image).
- `debug exec`—Debug a executable file.
- `debug proc`—Debug an already running process.

Once you have specified the first of these two pieces (the executable file or the process), you can specify the other piece using one of the following commands:

- `attach`—Attach to a running process.
- `core`—Retreive the process image from a core file.
- `executable`—Specify the executable file to use.
- `run` or `rerun`—Create a new process.

Attached processes

Typically, when you want to debug a program in CXdb, you specify the executable file and then run the executable file within CXdb. CXdb gains control of the created process, and you can step execution, set breakpoints, and manipulate program variables.

However, sometimes you may not want to debug the process created by the `run` command, but rather a program that has been started elsewhere. In such a case, you can attach CXdb to the process when the process nears the desired area of execution. This is particularly helpful when it takes hours for execution to reach a critical point.

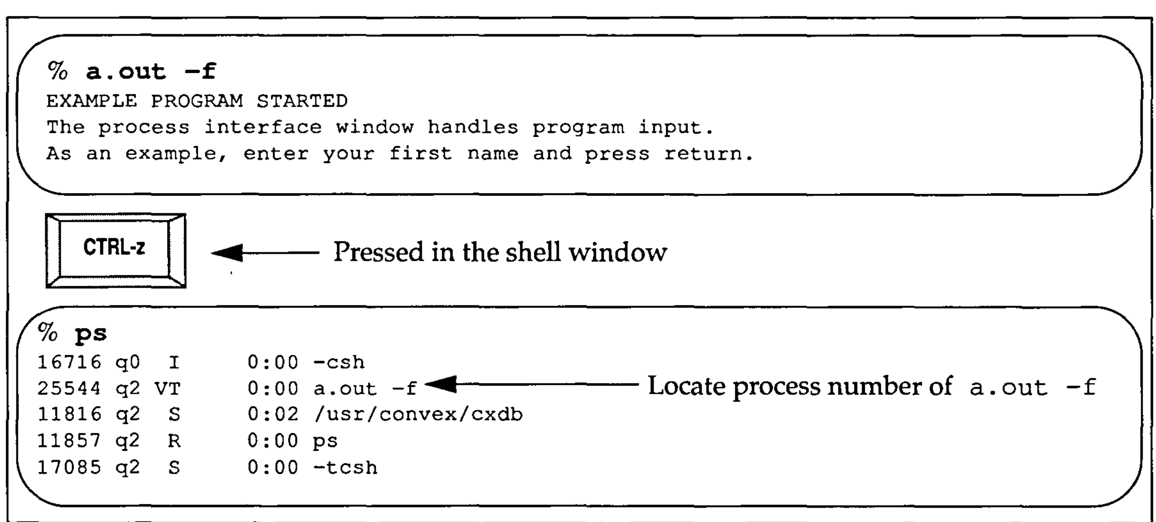
Running the program in the shell

Before debugging a running process, the process must be run in the shell. Figure 223 shows the example program being run in the shell with an argument passed to it.

Note

In the following example, do not provide any input to the process, or else execution will finish before you can attach to the process.

Figure 223
Attaching to a running process



In Figure 223, the `a.out -f` command creates a new process in the shell. The example program prints a message requesting you to enter your first name. Do not provide the requested input; instead press CTRL-z to stop the process and return to the shell prompt.

The ps command prints a list of processes and their process numbers. Locate the a.out -f process and remember its process number. In this case, its process number is 25544.

Debugging an existing process

Once a process is created, you can attach to it. There are two ways to tell CXdb you want to debug an existing process. One is to use the debug proc command to create a process object and bring the specified process into it. You must then specify the executable file using the executable command.

The second method is to create the process object using the debug exec command and bring the specified executable into it. You would then need to specify the process by using the attach command. Figure 224 demonstrates the second method.

Figure 224
Attaching CXdb to a process

```
(CXdb) debug exec a.out

Default source file: ./example.f
Default source language: Fortran

Process [#0] created
(CXdb) attach 25544
Attaching Process [#0] to pid 25544
Process [#0/0] stopped by attach at [0x8000cb86] ___ap$read+0xe
```

The debug exec command creates a process object. The default language is FORTRAN, and the default source file is the example.f file. The executable file has now been specified. However, the image of the process to be debugged still has not been specified.

The attach command specifies the process whose image is brought into the process object. CXdb attaches to the process, and then stops the process (though in this case it was already stopped). The source window highlights the current point of execution.

Now CXdb has all the pieces necessary to allow you to debug the process. The process, however, is still stopped in the shell. The process can be continued with the fg command, as shown in Figure 225.

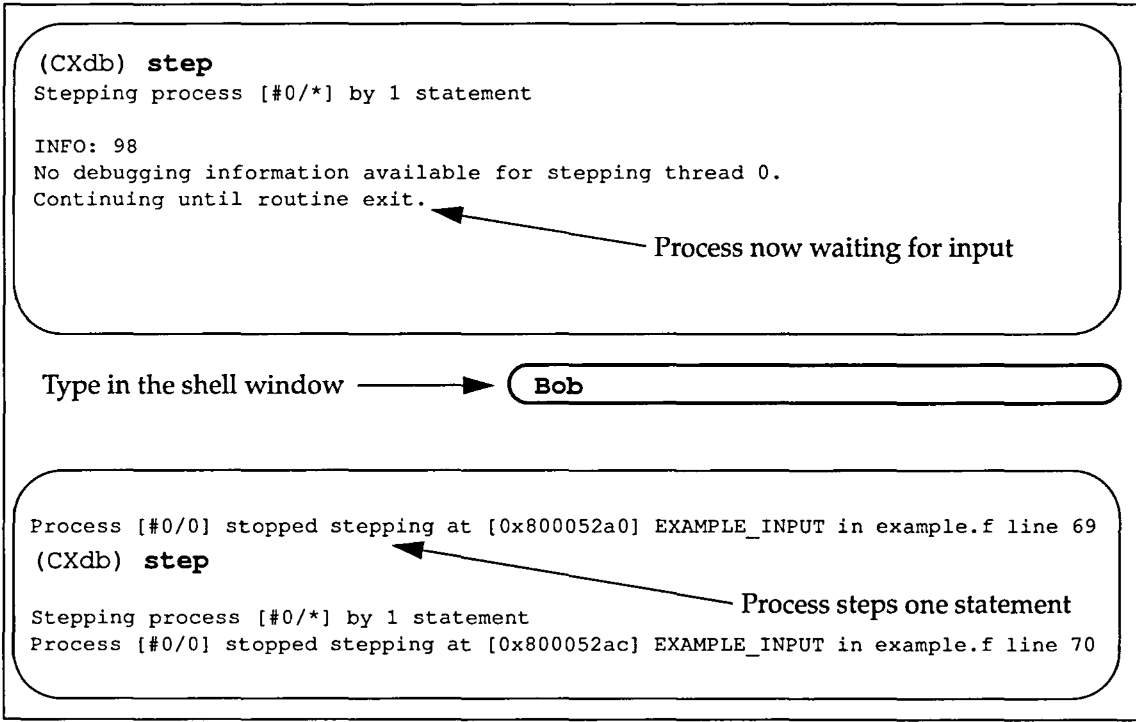
Figure 225
Bringing the process into the foreground

```
% fg
a.out -f
```

In Figure 225, the `fg` command brings the process back into the foreground. In this example, because the process was started outside of CXdb, the shell window in which the process was started acts as the process interface window for the process.

Figure 226 demonstrates stepping an attached process.

Figure 226
Stepping execution of an attached process



The `step` command continues process execution at the `READ` statement on line 67. At this point, you can enter your name in the shell window as input to the process.

When you do so, the `READ` statement completes, and the process stops stepping. At this point, you can continue to debug the program as you would a new process under CXdb's control.

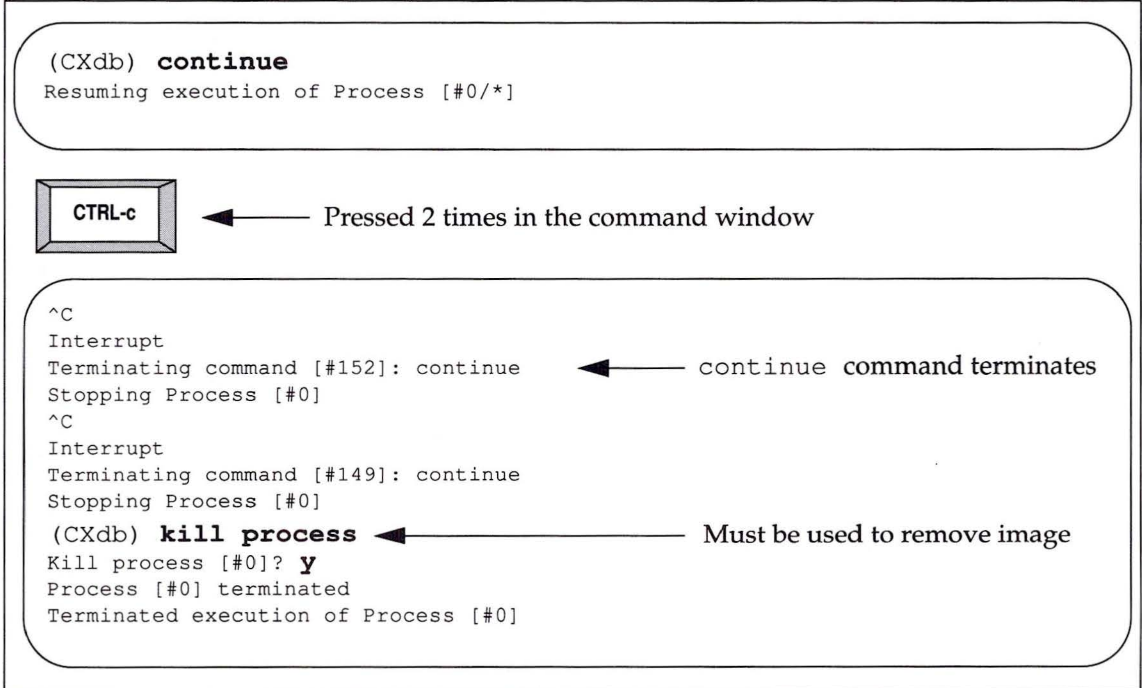
Letting an attached process run to completion

If an attached process runs to completion, the kernel sends a signal to the controlling process, but not to CXdb. Thus, CXdb cannot detect when an attached process completes. This leaves CXdb waiting for the process to stop; this can only be corrected by removing the image from the process object, as shown in Figure 227.

Note

An attached process that runs to completion causes CXdb to lose control over the process. To return CXdb to normal, the `kill process` command must be used.

Figure 227
Returning CXdb to normal after an attached process completes



The `continue` command resumes process execution, and the process runs to completion. CXdb does not receive notification from the kernel of the completion of the process, and therefore is left waiting for the process to stop.

Press **CTRL-c** twice in the command window to terminate the `continue` command. However, as far as CXdb is concerned, the process is still running. The `kill process` command must be used to remove the incorrect image from the process object.

Detaching from a process

After you complete your debugging of a process, you can detach from it by using the `detach` command. You can detach from a process started in CXdb or from an attached process. The process must be stopped before you can detach from it. This is demonstrated in Figure 228.

When you detach from a process, the process exists outside of the control of CXdb. The process is left running.

Note

A process whose order of execution has been altered while being debugged may continue execution unpredictably outside the control of CXdb. It is best not to detach from processes whose order of execution has been changed during debugging.

Figure 228
Detaching from a running process

```
(CXdb) run "-f" &
Starting process [#0]: a.out -f
Command [#167] completed
(CXdb) stop
Stopping Process [#0]
Process [#0/0] stopped at [0x8000cb86] ___ap$read+0xe
(CXdb) detach  <------------------- Detaches from current process
Process [#0] detached
(CXdb) info process
status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: no image    <-------- Image has been removed
     working dir: (default to current CXdb directory)
    default step: statement
default language: Fortran

source file search path:
          .
```

The `run` command creates a new process from the executable file. The `stop` command stops process execution. The `detach` command detaches CXdb from the current process. The process is left running in the shell from which it was invoked.

The `info process` command now shows that there is no process image.

Changing the executable file

When you have completed the debugging of one program in CXdb, you can switch to debugging a new program without leaving CXdb. You do so by specifying a new executable file.

When you specify a different executable file, the new executable replaces the existing executable file in the process object. If a current process exists, it is killed. All eventpoints are removed from the process object.

In the `/usr/lib/cxdb/examples` directory there is a subdirectory containing another program. Figure 229 shows how to debug this program with the `executable` command.

Figure 229
Changing the executable file of the process object

```
(CXdb) cd example3
(CXdb) executable a.out
Replace existing executable? y

Default source file: ./example3.f
Default source language: Fortran
(CXdb) run
Starting process [#0]: a.out

Process [#0] exited normally.
```

The `cd` command in Figure 229 changes the console working directory to the `example3` directory. The `cd` command in CXdb works much like the shell `cd` command. Relative path names now use the `/usr/lib/cxdb/example/example3` directory as a base path because it has become the console working directory.

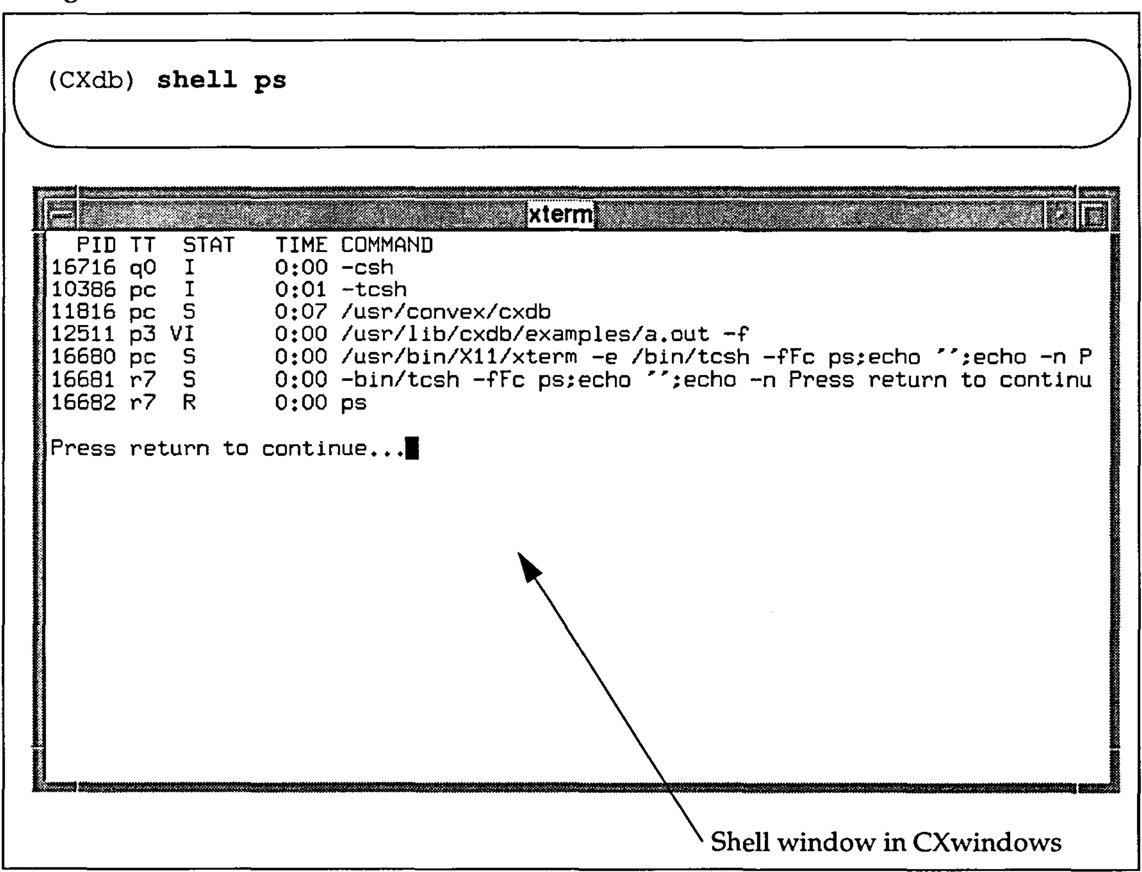
The `executable` command replaces the existing executable file in the process object with the executable file `a.out` located in the `example3` directory. The existing source window is closed and a new source window showing the source code corresponding to the new `a.out` file is opened.

The `run` command runs the new program. The program runs to completion.

Issuing a shell command within CXdb

A shell command of the operating system can be issued from within CXdb using the CXdb `shell` command. The `shell` command is executed and the appropriate output is displayed in an xterm window in CXwindows or on the screen in Maryland Windows. This is demonstrated in Figure 230.

Figure 230
Issuing a shell command from within CXdb



The shell `ps` command, shown in Figure 230, causes CXdb to open a shell window. The shell window displays the output from the `ps` command.

In CXwindows, the shell window is a normal xterm window. When you press RETURN inside of this window, the window closes.

In Maryland Windows, the `shell` command causes the ouput of the `ps` command to appear at the bottom of the terminal screen. Pressing RETURN causes CXdb to continue as normal.

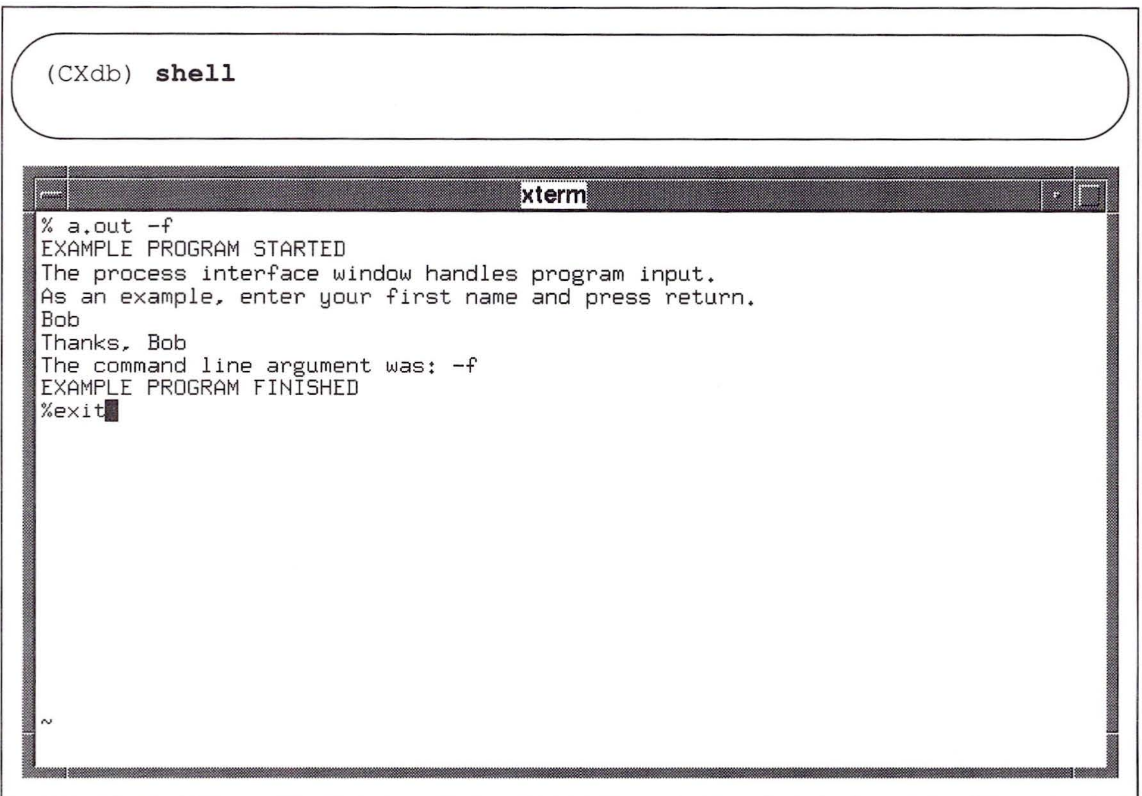
You can also invoke an interactive shell window from within CXdb. To do so, simply enter the `shell` command without any parameters.

In CXwindows, the shell window opens again. This window is a normal xterm window. When you exit the window using the `exit` command, the window closes.

In Maryland Windows, CXdb is stopped and control returns to the shell from which CXdb was invoked. When the `exit` command is used in this shell, CXdb is brought to the foreground again.

Figure 231 demonstrates the use of the `shell` command and shell window to resume execution of the process that was detached (see Figure 228).

Figure 231
Opening an interactive shell window



The `shell` command in Figure 231 opens a shell window in CXwindows. The shell window is an interactive xterm window. The `exit` command exits the shell window.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 232.

Figure 232
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Aliases and macros

11

This chapter describes how to create and use aliases and macros in the CXdb command language. It also explains how to use the command history to recall previous commands.

This chapter covers the following commands:

- `alias`
- `info alias`
- `info history`
- `info macro`
- `macro`
- `recall`
- `remove alias`
- `remove macro`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 233.

Figure 233
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 233 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 234.

Figure 234
Starting the example program

```
(CXdb) break routine chapter7
#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800026c6] CHAPTER7 in chapter7F.f line 4
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x800026c6] CHAPTER7 in chapter7F.f line 4
```

The `break routine` command in Figure 234 sets a breakpoint at the beginning of the routine `chapter7`. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

Aliases

An alias is an alternate name for a command. In the CXdb command language, an alias can represent part of a command, a complete command, or several commands.

The default initialization file (`/usr/lib/cxdb/.cxdbinit`) defines a number of default aliases. To create additional aliases, use the `alias` command.

Displaying aliases

You can use the `info alias` command to list the default aliases as well as any aliases you create. Figure 235 illustrates the `info alias` command.

Figure 235
Listing all aliases

```
(CXdb) info alias

!        "recall"
.        "source"
?        "help"
args     "info args"
b?       "info break"
bi       "break instruction"
bl       "break line"
br       "break routine"
bs       "break source"
bt       "backtrace"
c        "continue"
cfs      "clear fixed sched"
    .
    .
    .
t?       "info trace"
ti       "trace instruction"
tl       "trace line"
tr       "trace routine"
ts       "trace source"
up       "frame +1"
whatis   "info expression"
where    "info scope"
x        "examine"
```

The `info alias` command in Figure 235 lists all current aliases, both the default ones and the ones you have defined. If you want to list only particular aliases, you can limit the list by using a regular expression with the `info alias` command. For example, Figure 236 shows how to list all aliases that start with the letter `b`.

Figure 236
Using a regular expression with the `info alias` command

```
(CXdb) info alias b

b?       "info break"
bi       "break instruction"
bl       "break line"
br       "break routine"
bs       "break source"
bt       "backtrace"
```

Defining aliases

The `alias` command defines an alias. The following rules apply to defining an alias:

- The alias name can contain any printable characters except tab, forward slash (/), and backslash (\).
- Alias names are case sensitive.
- Alias names can consist of multiple words. When defining a multiword alias, precede each blank space with a backslash (\). The backslash is not needed to invoke the alias.
- The alias definition (what the alias name represents) must be a single string.
- If the alias definition contains white space, the entire definition must be enclosed in quotation marks (either single or double quotes).
- The alias definition can contain CXdb commands, macros, and other aliases.
- An alias definition remains in effect until the end of the current CXdb session or until you explicitly delete it, whichever comes first.

Figure 237 illustrates how to define aliases.

Figure 237
Defining aliases

```
(CXdb) alias ssl 'set step loop'
(CXdb) alias sss 'set step statement'
(CXdb) alias loop\ &\ print 'step loop; print ARRAY'
```

The first command in Figure 237 defines `ssl` as an alias for the command `set step loop`. Because the string `set step loop` contains white space, it must be enclosed in quotes. The second command defines `sss` as an alias for `set step statement`. The third command defines the multiword alias `loop & print` to represent two commands: `step loop` and `print ARRAY`. A backslash (\) is needed before each space in the name of this multiword alias.

To verify that the aliases have been defined correctly, use the `info alias` command, as shown in Figure 238.

Figure 238
Displaying the new aliases

```
(CXdb) info alias ss

ssl      "set step loop"
sss      "set step statement"

(CXdb) info alias loop

loop & print      "step loop; print ARRAY"
```

Redefining and deleting aliases

You can also use the `alias` command to assign a new definition to an existing alias name, as illustrated in Figure 239.

Figure 239
Redefining an alias

```
(CXdb) alias loop\ &\ print 'step loop; print TABLE'
```

To delete an alias, use the `remove alias` command, as illustrated in Figure 240. The `remove alias` command takes effect immediately, without asking for confirmation.

Figure 240
Deleting an alias

```
(CXdb) remove alias sss
```

Note

All alias definitions are automatically deleted when you quit the current debugging session. If there are any aliases that you want to use in future debugging sessions, you should define those aliases in an initialization file. (Refer to Chapter 12 for a description of initialization files.)

Using aliases

To invoke an alias, simply enter its name on the command line. The following rules apply to using aliases:

- The alias must be the first entry on the command line.
- CXdb expands the alias to its definition before executing the command line
- You cannot combine multiple aliases to form a single command.
- Aliases cannot accept parameters. (However, macros can.)

Figure 241 shows several ways to invoke aliases.

Figure 241
Invoking aliases

```
(CXdb) ssl
(CXdb) s 2
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x80002700] CHAPTER7 in chapter7F.f line 6
(CXdb) loop & print
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80002700] CHAPTER7 in chapter7F.f line 6
(CXdb) INTEGER*4(1:4, 1:4)
(1..4,1) :  2 0 0 0
(1..4,2) :  5 0 0 0
(1..4,3) :  10 0 0 0
(1..4,4) :  17 0 0 0
```

In Figure 241, the alias `ssl` sets the stepping granularity to loop. The alias `s` is the default alias for the `step` command, so the command `s 2` steps the process by two loops. Because the alias is expanded before the command line is executed, the step count of 2 is a parameter of the `step` command and not a parameter of the alias `s`.

The alias `loop & print` expands into two commands. The first command steps the process by one loop, and the second command prints the array called `TABLE`.

Macros

Macros enable you to customize the CXdb command language to suit your particular needs. Macros and aliases are similar in many respects and can often be used interchangeably. However, macros are more powerful in the following ways:

- Macros can accept parameters, but aliases cannot.
- A macro can invoke itself iteratively, but an alias cannot.
- A macro can appear anywhere within a command, but an alias must be at the beginning of the command.
- Multiple macros can be used in a single command, but only one alias can be used per command.

Defining macros

The `macro` command defines a macro. The following rules apply to defining a macro:

- A macro name can contain alphanumeric characters only. No white space is allowed in a macro name.
- Macro names are case sensitive.
- The macro definition (what the macro name represents) must be a single string.
- If the macro definition contains white space, the entire definition must be enclosed in quotation marks (either single or double quotes).
- The macro definition can contain CXdb commands, aliases, parameters, and other macros.
- A macro definition remains in effect until the end of the current CXdb session or until you explicitly delete it, whichever comes first.

Figure 242 illustrates how to define a macro.

Figure 242
Defining a macro

```
(CXdb) macro sl 'step loop'
```

The `macro` command in Figure 242 defines `sl` as a macro that represents the command `step loop.` Defining `sl` as an alias would have worked just as well in this case. However, if you want to include a parameter in the definition, then a macro is needed. Figure 243 shows how to define a macro that includes a parameter.

Figure 243
Including a parameter in the macro definition

```
(CXdb) macro lp(N:1) 'step loop N; info locals'
```

In Figure 243, the macro `lp` accepts the parameter `N` to specify the number of loops to step. The default value for `N` is 1, specified by `(N:1).` This macro also includes a second command, `info locals.`

Displaying macros

The `info macro` command displays the current definitions of macros. Figure 244 illustrates this command.

Figure 244
Listing all macros

```
(CXdb) info macro

lp( N:1 )       "step loop N; info locals"
sl      "step loop"
```

The `info macro` command in Figure 244 lists the definitions of all current macros. If you want to list only particular macros, you can limit the list by using a regular expression with the `info macro` command. For example, Figure 245 shows how to list all macros that start with the letter `s.`

Figure 245
Using a regular expression with the `info macro` command

```
(CXdb) info macro s

sl      "step loop"
```

Using macros

To invoke a macro, prefix an at-sign (@) to its name. The following rules apply to using macros:

- The macro may appear anywhere on the command line.
- CXdb expands the macro to its definition before parsing the command line.
- Multiple macros can be used to form a single command.
- A macro can invoke itself iteratively. This is not recursion in a strict sense because it does not involve a stack. The macro merely continues to expand until it reaches the end of its parameter list.
- Macros can perform token pasting, which concatenates a macro parameter with another parameter or with a fixed string.

Figure 246 shows how to invoke a macro.

Figure 246
Invoking a macro

```
(CXdb) @lp(2)

Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x80002700] CHAPTER7 in chapter7F.f line 6
(CXdb) Process [#0/0]
Frame : 0; [0x80002700] CHAPTER7 in chapter7F.f line 6
Number of locals : 3
     1 :  TABLE = INTEGER*4(1:4, 1:4) 0x8005b058
     2 :  I = (INTEGER*4) 4
     3 :  J = (INTEGER*4) 5
```

In Figure 246, the macro `lp` is invoked with a value of 2 for the parameter `N`. This steps the process by two loops, then lists the local variables.

A macro can be defined to invoke itself iteratively, as shown in Figure 247.

Figure 247
Defining an iterative macro

```
(CXdb) macro P(X) 'print X; @P'
```

The command in Figure 247 defines the macro `P`. This macro prints the first parameter passed to it as `X`. The macro continues to expand (by `@P`) to print the next parameter passed to it as `X`. The expansion continues until all parameters in the list are printed. Figure 248 illustrates the use of this macro.

Figure 248
Invoking an iterative macro

```
(CXdb) @P(I,J,TABLE)
(INTEGER*4) 4
(CXdb) (INTEGER*4) 5
(CXdb) INTEGER*4(1:4, 1:4)
(1..4,1) : 2 4 6 0
(1..4,2) : 5 12 21 0
(1..4,3) : 10 26 54 0
(1..4,4) : 17 48 129 0
```

In Figure 248, macro `P` is invoked with 3 parameters: `I`, `J`, and `TABLE`. The macro goes through 3 iterations, printing one parameter with each iteration.

Macros can also perform token pasting. Token pasting uses the operator `##` to concatenate a macro parameter with a command line token (lexeme) or with another macro parameter. Figure 249 illustrates how to define a macro that uses token pasting.

Figure 249
Defining a macro with token pasting

```
(CXdb) macro FL(LN:1) 'chapter13F.f:##LN'
```

The command in Figure 249 defines the macro `FL`, which concatenates the parameter `LN` with the fixed string `chapter13F.f:`. The default value for `LN` is 1, specified by `(LN:1)`. This macro can be used to set breakpoints at specific line numbers of the file `chapter13F.f`. Figure 250 illustrates the use of this macro.

Figure 250
Invoking a macro that uses token pasting

```
(CXdb) break line @FL
#1: break line, on [#0/*], Enabled, ignore 0/0
        [0x80003434] CHAPTER13F in chapter13F.f line 1
(CXdb) break line @FL(12)
#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x800034e6] CHAPTER13F in chapter13F.f line 12
```

The first `break line` command in Figure 250 uses the macro `FL` to set a breakpoint at line number 1 in the file `chapter13F.f`. The line number in this case is specified by the default value for the parameter `LN`. The second `break line` command specifies a value of 12 for the parameter `LN`, so the breakpoint is set at line 12 of the file `chapter13F.f`.

Redefining and deleting macros

You can also use the `macro` command to assign a new definition to an existing macro name, as illustrated in Figure 251.

Figure 251
Redefining a macro

```
(CXdb) macro lp(N:1) 'step loop N; print TABLE'
```

To delete a macro definition, use the `remove macro` command, as illustrated in Figure 252. The `remove macro` command takes effect immediately, without asking for confirmation.

Figure 252
Deleting a macro

```
(CXdb) remove macro sl
```

Note

All macro definitions are automatically deleted when you quit the current debugging session. If you want to include specific macros in future debugging sessions, you should define those macros in an initialization file. (Refer to Chapter 12 for a description of initialization files.)

Command history

CXdb stores the last 100 commands you entered in a history buffer. To display the contents of the command history buffer, use the `info history` command, as illustrated in Figure 253.

Figure 253
Displaying the command history buffer

```
(CXdb) info history

debug exec a.out
break routine chapter7
run
info alias
info alias b
alias ssl 'set step loop'
alias sss 'set step statement'
alias loop\ &\ print 'step loop; print ARRAY'
info alias ss
info alias loop
alias loop\ &\ print 'step loop; print TABLE'
remove alias sss
ssl
s 2
loop & print
macro sl 'step loop'
macro lp(N:1) 'step loop N; info locals'
info macro
info macro s
@lp(2)
macro P(X) 'print X; @P'
@P(I,J,TABLE)
macro FL(LN:1) 'chapter7F.f:##LN'
break line @FL
break line @FL(12)
macro lp(N:1) 'step loop N; print TABLE'
remove macro sl
info history
```

As Figure 253 shows, the history buffer also includes the command just entered, which is the `info history` command.

If you do not want to view the entire history buffer, you can specify the number of commands to display, as illustrated in Figure 254. The count starts from the bottom of the history list (last command entered) and proceeds toward the top of the list.

Figure 254
Displaying a specified number of commands

```
(CXdb) info history 8

@P(I,J,TABLE)
macro FL(LN:1) 'chapter7F.f:##LN'
break line @FL
break line @FL(12)
macro lp(N:1) 'step loop N; print TABLE'
remove macro sl
info history
info history 8
```

You can view the history list one line at a time by using **CTRL-p** to go to the previous line and **CTRL-n** to go to the next line.

You can also retrieve a command from the history list and execute it again by using the `recall` command. This command searches for the first entry in the history list that contains the search string you specify. The search starts at the bottom of the history list (last command entered) and proceeds toward the top of the list. Figure 255 illustrates the `recall` command.

Figure 255
Recalling a command

```
(CXdb) recall break
(CXdb) break line @FL(12)   <-------------------- Recalled command

#3: break line, on [#0/*], Enabled, ignore 0/0
       [0x800034e6] CHAPTER13F in chapter13F.f line 12

INFO: 104
Eventpoint 2 also has a breakpoint at address 0x800034e6.
```

The `recall` command in Figure 255 searches for the first entry in the history list that *begins* with the string `break.` It finds the entry `break line @FL(12)` and executes it immediately.

To search for a string anywhere within any of the history entries, prefix the search string with a question mark (`?`), as shown in Figure 256.

Figure 256
Using a search string to recall a command

```
(CXdb) recall ?his
(CXdb) info history 8 <——— Recalled command

break line @FL
break line @FL(12)
macro lp(N:1) 'step loop N; print TABLE'
remove macro sl
info history
info history 8
break line @FL(12)
info history 8
```

The `recall` command in Figure 256 searches for the first entry in the history list that *contains* the string `his`. It finds the entry `info history 8` and executes it immediately.

Note

The `recall` command immediately executes the entry it retrieves from the history list, without asking for confirmation and without waiting for you to press RETURN.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 257.

Figure 257
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Logging, command files, and batch mode 12

This chapter explains how to use files to manipulate CXdb input and output. It describes file manipulation in relation to 3 major functions:

- Logging CXdb input, output, and messages
- Using command files and initialization files
- Running CXdb in batch mode

This chapter covers the following commands:

- `add cmderr`
- `add cmdlog`
- `add cmdout`
- `clear echo`
- `clear logging`
- `cxdb -b`
- `edit`
- `redirection operators`
- `remove cmderr`
- `remove cmdlog`
- `remove cmdout`
- `set cmderr`
- `set cmdlog`
- `set cmdout`
- `set echo`
- `set logging`
- `set noclobber`
- `source`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 258.

Figure 258
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 258 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Caution

The examples in this chapter create and delete files in your home directory. You should carefully consider what effects this might have on your home directory before executing any of the examples.

Logging CXdb input, output, and messages

During a debugging session, you can keep a record (or log) of any commands you enter as well as any output or messages generated by CXdb. To log CXdb input and output, you direct it to locations called *viewports*. A viewport is either a file or the CXdb command window.

Logging CXdb output

The standard output from CXdb is called cmdout, and it normally appears in the command window. To log the output to a file, use the `add cmdout` command to define the file as a viewport for cmdout. Figure 259 shows how to define a file as a viewport for output.

Figure 259
Adding files to the viewport list for cmdout

```
(CXdb) add cmdout ~/output.log, ~/session.cxdb
New cmdout: Window #1, /usr/homedir/output.log, /usr/homedir/session.cxdb
```

The `add cmdout` command in Figure 259 adds two files to the viewport list for cmdout. The file names are `output.log` and `session.cxdb`. The tilde (~) specifies that the files are to go in your home directory. The response to the command indicates that the new viewport list includes these two files plus `Window #1` (the command window), which is the default viewport for cmdout. Any output from CXdb commands now goes to all three of these viewports simultaneously.

To check the viewport list, use the `info cxdb` command, as illustrated in Figure 260.

Figure 260
Checking the viewport list for cmdout

Viewport list for CXdb output

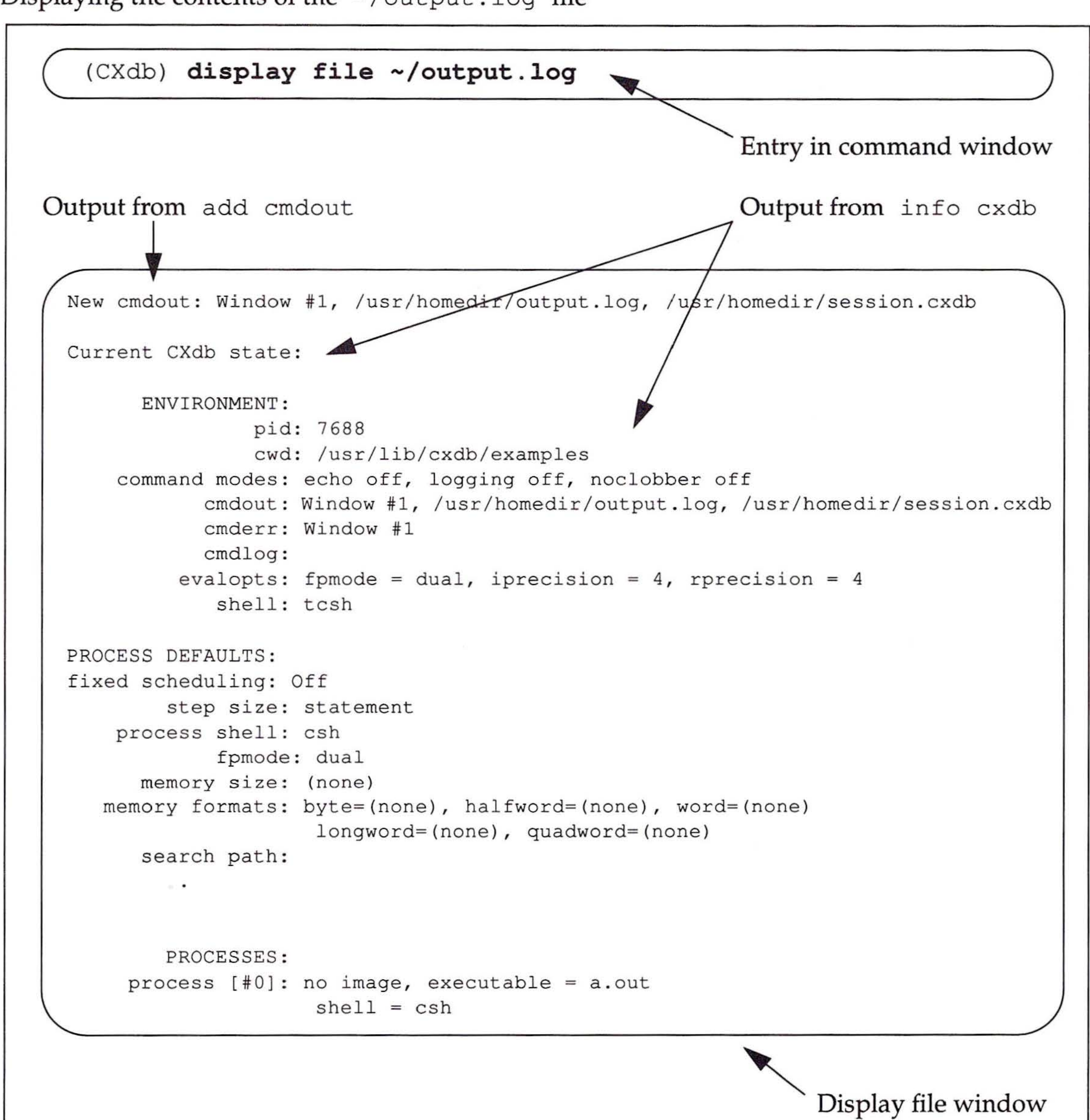
```
(CXdb) info cxdb

Current CXdb state:

      ENVIRONMENT:
              pid: 7688
              cwd: /usr/lib/cxdb/examples
    command modes: echo off, logging off, noclobber off
           cmdout: Window #1, /usr/homedir/output.log, /usr/homedir/session.cxdb
           cmderr: Window #1
           cmdlog:
         evalopts: fpmode = dual, iprecision = 4, rprecision = 4
            shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
        step size: statement
    process shell: csh
           fpmode: dual
      memory size: (none)
   memory formats: byte=(none), halfword=(none), word=(none)
                    longword=(none), quadword=(none)
      search path:
          .


        PROCESSES:
     process [#0]: no image, executable = a.out
                    shell = csh
```

To display the contents of the log files, you can use either of the following methods:

- Use the `display file` command.
- Use the `shell` command to access the shell, then display the file with `cat` or a similar command.

At this point, both `output.log` and `session.cxdb` contain the same information. Both files are a log of the output from the `add cmdout` and `info cxdb` commands in Figure 259 and Figure 260, respectively. Figure 261 uses the `display file` command to show the contents of the `output.log` file.

Figure 261
Displaying the contents of the `~/output.log` file



Logging CXdb messages

In response to your commands, CXdb can issue informational and error messages. These standard messages are collectively called cmderr, and they normally display in the command window. To log the messages to a file, use the `add cmderr` command to define the file as a viewport for cmderr. Figure 262 shows how to define a file as a viewport for CXdb messages.

Figure 262
Adding files to the viewport list for cmderr

```
(CXdb) add cmderr ~/error.log, ~/session.cxdb
New cmderr: Window #1, /usr/homedir/error.log, /usr/homedir/session.cxdb
```

The `add cmderr` command in Figure 262 adds two files to the viewport list for cmderr. The file names are `error.log` and `session.cxdb`. The response to the command indicates that the new viewport list includes these two files plus `Window #1` (the command window), which is the default viewport for cmderr. Any messages from CXdb are now sent to all three of these viewports simultaneously.

To check the viewport list, use the `info cxdb` command, as illustrated in Figure 263.

Figure 263
Checking the viewport list for cmderr

```
(CXdb) info cxdb

Current CXdb state:

      ENVIRONMENT:
               pid: 7688
               cwd: /usr/lib/cxdb/examples
    command modes: echo off, logging off, noclobber off
           cmdout: Window #1, /usr/homedir/output.log, /usr/homedir/session.cxdb
           cmderr: Window #1, /usr/homedir/error.log, /usr/homedir/session.cxdb
           cmdlog:
         evalopts: fpmode = dual, iprecision = 4, rprecision = 4
            shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
       step size: statement
   process shell: csh
          fpmode: dual
     memory size: (none)
  memory formats: byte=(none), halfword=(none), word=(none)
                   longword=(none), quadword=(none)
     search path:
        .


       PROCESSES:
   process [#0]: no image, executable = a.out
                  shell = csh
```

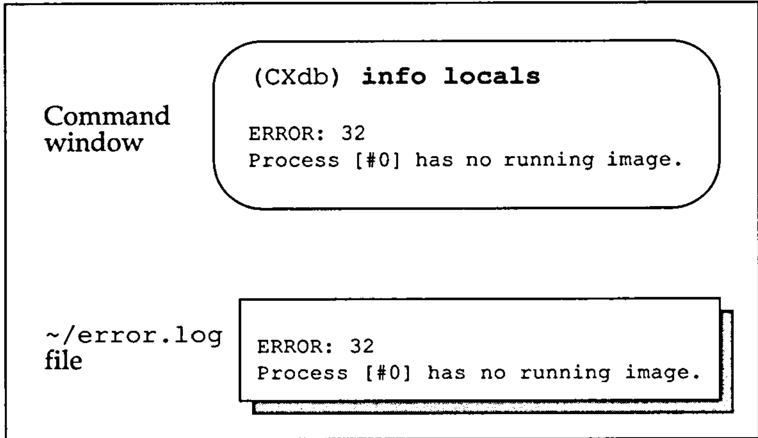
Viewport list for CXdb messages

To display the contents of the log files, you can use either of the following methods:

- Use the `display file` command.
- Use the `shell` command to access the shell, then display the file with `cat` or a similar command.

Figure 264 shows a command that results in an error message. It also shows the contents of the `error.log` file after this command is executed.

Figure 264
Logging an error message



Logging CXdb input

Standard input to CXdb is called cmdlog. Normally this input is echoed in the command window. To log your input to a file, use the `add cmdlog` command to define the file as a viewport for cmdlog. Then use the `set logging` command to enable logging of the input. Figure 265 illustrates how to log CXdb input.

Figure 265
Logging your input

```
(CXdb) add cmdlog ~/input.log, ~/session.cxdb
New cmdlog: /usr/homedir/input.log, /usr/homedir/session.cxdb
(CXdb) set logging
```

The `add cmdlog` command in Figure 265 adds two files to the viewport list for cmdlog. The file names are `input.log` and `session.cxdb`. The `set logging` command then enables logging of input to these files. Your input to CXdb now goes to these two files as well as being echoed in the command window.

To check the viewport list and the status of the logging option, use the `info cxdb` command, as illustrated in Figure 266.

Figure 266
Checking the status of logging for cmdlog

```
(CXdb) info cxdb

Current CXdb state:

      ENVIRONMENT:
               pid: 7688
               cwd: /usr/lib/cxdb/examples
     command modes: echo off, logging on, noclobber off
            cmdout: Window #1, /usr/homedir/output.log, /usr/homedir/session.cxdb
            cmderr: Window #1, /usr/homedir/error.log, /usr/homedir/session.cxdb
            cmdlog: /usr/homedir/input.log, /usr/homedir/session.cxdb
          evalopts: fpmode = dual, iprecision = 4, rprecision = 4
             shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
       step size: statement
   process shell: csh
          fpmode: dual
     memory size: (none)
  memory formats: byte=(none), halfword=(none), word=(none)
                  longword=(none), quadword=(none)
     search path:
          .


       PROCESSES:
    process [#0]: no image, executable = a.out
                  shell = csh
```


To display the contents of the log files, you can use either of the following methods:

- Use the `display file` command.
- Use the `shell` command to access the shell, then display the file with `cat` or a similar command.

Figure 267 shows several commands being entered in the command window. It also shows the contents of the `input.log` file after those commands are entered.

Figure 267
Logging input

Command window

```
(CXdb) executable a.out
Replace existing executable? y

Default source file: example.f
Default source language: Fortran
(CXdb) break line chapter7F.f:40

#0: break line, on [#0/*], Enabled, ignore 0/0
       [0x800029c0] BLD_MATRIX in chapter7F.f line 40
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x800029c0] BLD_MATRIX in chapter7F.f line 40
```

~/input.log file

```
info cxdb
executable a.out
break line chapter7F.f:40
run
```

Redefining and deleting viewports

You can use the `add cmdout`, `add cmderr`, and `add cmdlog` commands to add viewports at any time, as shown in Figure 268.

Figure 268
Adding another viewport

```
(CXdb) add cmderr ~/message.log
New cmderr: Window #1, /usr/homedir/error.log, /usr/homedir/session.cxdb,
/usr/homedir/message.log
```

To delete a viewport from one of the lists, use the `remove cmdout`, `remove cmderr`, or `remove cmdlog` command, as illustrated in Figure 269.

Figure 269
Deleting a viewport

```
(CXdb) remove cmderr ~/error.log
New cmderr: Window #1, /usr/homedir/session.cxdb, /usr/homedir/message.log
```

You can also delete an old viewport list and replace it with a new list by using the `set cmdout`, `set cmderr`, or `set cmdlog` command. Figure 270 illustrates the use of the `set cmdout` command to reset the viewport list so that `Window #1` (the command window) is the only viewport for CXdb output.

Figure 270
Resetting the viewport list

```
(CXdb) set cmdout 1
New cmdout: Window #1
```

To disable logging of input to all viewports except the command window, use the `clear logging` command, as shown in Figure 271.

Figure 271
Disabling input logging

```
(CXdb) clear logging
```

Using redirection operators

In addition to the viewport lists mentioned in the previous sections, you can also use redirection operators to send CXdb output or messages to specified viewports. (There are no redirection operators for CXdb input.) The redirection operators affect only the particular CXdb command with which they appear.

Figure 272 shows the use of a redirection operator (>) to direct the output of the `info process` command to the file `process.log`. Redirection operators override all other viewport lists, so in this case the output of the `info process` command does not even display in the command window.

Figure 272
Redirecting output to a file

```
(CXdb) info process > ~/process.log
(CXdb)
```

For more information about redirection operators, refer to the parameter description *<redirection-operator>* in Chapter 3, "Parameters," of the *CONVEX CXdb Reference* manual.

Controlling file writes

When logging cmderr, cmdlog, or cmdout to files, CXdb creates the files if they do not exist. If a specified file already exists, then CXdb overwrites it unless you use the noclobber option. Figure 273 illustrates the use of this option.

Figure 273
Using noclobber to prevent overwriting of a file

```
(CXdb) set noclobber
(CXdb) add cmdout ~/output.log

ERROR: 162
File /usr/homedir/output.log exists. Can't overwrite with NOCLOBBER set.
```

In Figure 273, the `set noclobber` command enables the noclobber option. The `add cmdout` command tries to add the file `output.log` to the viewport list for cmdout. Because this file already exists in the current directory, the noclobber option prevents the file from being overwritten, and an error results.

To display the current status of the noclobber option, use the `info cxdb` command, as illustrated in Figure 274.

Figure 274
Checking the status of the noclobber option

```
(CXdb) info cxdb

Current CXdb state:

      ENVIRONMENT:
               pid: 7688
               cwd: /usr/lib/cxdb/examples
     command modes: echo off, logging off, noclobber on
            cmdout: Window #1
            cmderr: Window #1, /usr/homedir/session.cxdb, /usr/homedir/message.log
            cmdlog: /usr/homedir/input.log, /usr/homedir/session.cxdb
          evalopts: fpmode = dual, iprecision = 4, rprecision = 4
             shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
       step size: statement
   process shell: csh
          fpmode: dual
     memory size: (none)
  memory formats: byte=(none), halfword=(none), word=(none)
                  longword=(none), quadword=(none)
     search path:
        .


       PROCESSES:
    process [#0]: created pid 21128, state = stopped, executable = a.out
                  shell = csh
```


Command files and initialization files

A command file is any file that contains CXdb commands. A command file is a convenient way to store a sequence of commands that you want to execute more than once. Whenever you want to repeat the sequence, you can execute the command file by invoking the file with the `source` command.

There is a special type of command file known as an initialization file. Initialization files execute automatically whenever you invoke CXdb.

Creating a command file

A command file is an ASCII text file that contains CXdb commands. Therefore, you can create and edit a command file in the same way you create any ASCII text file. One method of creating a command file is to log the commands (cmdlog) to a file the first time you enter them, then edit the log file to generate the desired command file. Figure 275 illustrates this process.

Figure 275
A command file created from a cmdlog file

Contents of `~/input.log` file

```
info cxdb
executable a.out
break line chapter7F.f:40
run
add cmderr ~/message.log
remove cmderr ~/error.log
set cmdout 1
clear logging
```

Command file `chapter12.cmds` created by editing `input.log` file

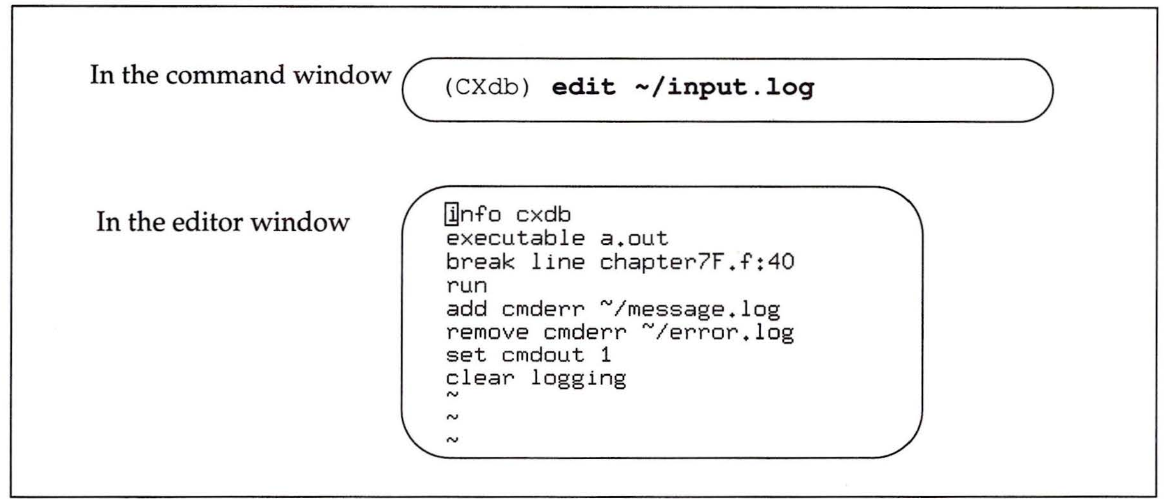
```
executable a.out
break line chapter7F.f:40 {if(MATRIX(I,J,K).GT.100)\
 {echo "Value too high:"; echo/n " I= "; print I;\
 echo/n " J= "; print J; echo/n " K= "; print K;\
 echo/n " MATRIX(I,J,K)= "; print MATRIX(I,J,K);}\
 else resume;} $BL40
run
```

Figure 275 shows the command file `chapter12.cmds,` which is created by editing the cmdlog file `input.log.` The command file sets a breakpoint with a handler that tests whether the value of an array element exceeds 100. Because the handler is lengthy, it would be difficult to type correctly in the command window. But the command file provides an easy way to create the handler and to save it so that it can be modified and executed again.

The command file in Figure 275 also uses the debugger variable `$BL40` to store the eventpoint number of the breakpoint. This makes it possible to refer to the breakpoint by name, rather than by number, in other CXdb commands.

You can edit a command file from the shell, or you can use the `edit` command to invoke your default editor from within CXdb. The `edit` command opens an editor window, as shown in Figure 276.

Figure 276
Using the `edit` command



Executing a command file

After creating the command file, you can execute it with the `source` command, as shown in Figure 277.

Figure 277
Executing the command file

```
(CXdb) source chapter12.cmds
Replace existing executable? y   <---- CXdb waits for confirmation

All eventpoints on process [#0] removed

Default source file: example.f
Default source language: Fortran

#1: break line, on [#0/*], Enabled, ignore 0/0
        [0x800029c0] BLD_MATRIX in chapter7F.f line 40
    {
        if(MATRIX(I,J,K).GT.100) {echo "Value too high:"; echo/n " I= "; print ...
    }
Process [#0] is already running with pid 788.
Terminate existing process and restart? y
Starting process [#0]: a.out
Value too high:
 I= (INTEGER*4) 3
 J= (INTEGER*4) 4
 K= (INTEGER*4) 1
 MATRIX(I,J,K)= (REAL*4) 125.6875
```

As Figure 277 indicates, the commands from the command file are not echoed in the command window as they are executed. To enable echoing of the command file commands, use the `set echo` command, as illustrated in Figure 278.

Figure 278
Executing the command file with echoing enabled

```
(CXdb) set echo
(CXdb) source chapter12.cmds
(CXdb) executable a.out
Replace existing executable? y

All eventpoints on process [#0] removed

Default source file: example.f
Default source language: Fortran
(CXdb) break line chapter7F.f:40 {if(MATRIX(I,J,K).GT.100) {echo "Value too ...

#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x800029c0] BLD_MATRIX in chapter7F.f line 40
    {
        if(MATRIX(I,J,K).GT.100) {echo "Value too high:"; echo/n " I= "; print ...
    }
(CXdb) run
Process [#0] is already running with pid 16279.
Terminate existing process and restart? y
Starting process [#0]: a.out
Value too high:
 I= (INTEGER*4) 3
 J= (INTEGER*4) 4
 K= (INTEGER*4) 1
 MATRIX(I,J,K)= (REAL*4) 125.6875
```

Echoed commands

To check the current status of the echo option, use the `info cxdb` command, as shown in Figure 279.

Figure 279
Checking the status of the echo option

```
(CXdb) info cxdb

Current CXdb state:

      ENVIRONMENT:
               pid: 7688
               cwd: /usr/lib/cxdb/examples
    command modes: echo on, logging off, noclobber on
           cmdout: Window #1
           cmderr: Window #1, /usr/homedir/session.cxdb, /usr/homedir/message.log
           cmdlog: /usr/homedir/input.log, /usr/homedir/session.cxdb
         evalopts: fpmode = dual, iprecision = 4, rprecision = 4
            shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
        step size: statement
    process shell: csh
           fpmode: dual
      memory size: (none)
   memory formats: byte=(none), halfword=(none), word=(none)
                   longword=(none), quadword=(none)
      search path:
         .


        PROCESSES:
     process [#0]: created pid 18128, state = stopped, executable = a.out
                   shell = csh
```


Using initialization files

An initialization file is a command file that executes automatically when you invoke CXdb. Initialization files have the same format, and are created in the same way, as command files.

Use initialization files to:

- Define aliases
- Define macros
- Set defaults for CXdb and the processes being debugged
- Execute a sequence of commands as a standard start-up procedure for CXdb

An initialization file must always be named `.cxdbinit`. You can have a different `.cxdbinit` file in each directory you own. Each time it is invoked, CXdb executes a maximum of 3 `.cxdbinit` files, in the following order:

- The default `.cxdbinit` file in the directory `/usr/lib/cxdb`
- The `.cxdbinit` file in your home directory
- The `.cxdbinit` file in the console working directory (the directory from which you invoked CXdb)

If a `.cxdbinit` file does not exist at one of these three levels, that level is ignored. The commands from a later `.cxdbinit` file override the commands from preceding `.cxdbinit` files. Figure 280 shows of how these three levels of `.cxdbinit` files interact.

Figure 280
Initialization files

❶ Contents of
`/usr/lib/cxdb/.cxdbinit`
(contains default aliases)

```
alias denv+ "add default environment"
alias env+ "add environment"
alias dp+ "add default path"
          .
          .
          .
alias tl "trace line"
alias tr "trace routine"
alias ts "trace source"
```

❷ Contents of
`.cxdbinit` in your home directory

```
alias ss 'set shell'
add default path sourcetree2
```

❸ Contents of
`.cxdbinit` in the console working directory

```
alias ss 'set step'
macro P(X) 'print X; @P'
debug exec a.out
```

As Figure 280 indicates, the first initialization file executed is the default `.cxdbinit` file in the `/usr/lib/cxdb` directory. This file defines default aliases for CXdb commands.

The second initialization file executed is the .cxdbinit file in the user's home directory. In Figure 280, this file contains two commands. The first command defines an alias named ss for the set shell command. The second command adds the directory sourcetree2 to the default search path. These settings override any default settings contained in the initialization file at level 1 (/usr/lib/cxdb/.cxdbinit).

The third initialization file is the .cxdbinit file in the console working directory (which is /usr/lib/cxdb/examples in this case). The file contains three commands that do the following:

- Define the alias ss to mean set step. This overrides the alias ss defined in the .cxdbinit file at level 2 (the home directory).
- Define the macro P
- Create a process object by loading the executable file a.out with the debug exec command

Batch mode

You can execute CXdb commands in batch mode without invoking the interactive user interface. The primary purpose of batch mode is to allow you to conduct all CXdb input and output through files.

To execute CXdb in batch mode, use the -b option for the cxdb command. Figure 281 shows an example of batch mode execution. The example includes the contents of the input and output files as well as the shell command that invokes CXdb in batch mode.

Figure 281
Using CXdb in batch mode

In shell window

```
% cxdb -b -e a.out -f batch.input > ~/batch.output &
```

Contents of `batch.input` file

```
executable a.out
break line chapter7F.f:40 {if(MATRIX(I,J,K).GT.100)\
 {echo "Value too high:"; echo/n " I= "; print I;\
 echo/n " J= "; print J; echo/n " K= "; print K;\
 echo/n " MATRIX(I,J,K)= "; print MATRIX(I,J,K);}\
 else resume;} $BL40
run
quit
```

Contents of `~/batch.output` file after execution of the `cxdb` command

```
CXdb version 1.1, Copyright (C) 1991, Convex Computer Corp.
(CXdb) debug exec a.out

Default source file: ./example.f
Default source language: Fortran

Process [#0] created
(CXdb) source batch.input
(CXdb) executable a.out
Replace existing executable? yes

Default source file: example.f
Default source language: Fortran
(CXdb) break line chapter7F.f:40 {if(MATRIX(I,J,K).GT.100) {echo "Value too high
:"; echo/n " I= "; print I; echo/n " J= "; print J; echo/n " K= "; print K; echo
/n " MATRIX(I,J,K)= "; print MATRIX(I,J,K);} else resume;} $BL40

#0: break line, on [#0/*], Enabled, ignore 0/0
 [0x800029co] BLD_MATRIX in chapter7F.f line 40
 {
; echo/n " J= "; print J; echo/n " K= "; print K; echo/n " MATRIX(I,J,K)= "; pri
nt MATRIX(I,J,K);} else resume;
 }
(CXdb) run
Starting process [#0]: a.out
Value too high:
 I= (INTEGER*4) 3
 J= (INTEGER*4) 4
 K= (INTEGER*4) 1
 MATRIX(I,J,K)= (REAL*4) 125.6875
(CXdb) quit
Process [#0] is still running. Kill it? yes
```

In Figure 281, the -b option used with the cxdb command invokes CXdb in batch mode. The option -e a.out specifies a.out as the executable file, and it is equivalent to the command debug exec a.out. The option -f batch.input executes the command file batch.input, and it is equivalent to the command source batch.input. The option > ~/batch.output redirects the CXdb output to the file batch.output in your home directory. The background directive (&) tells the shell to execute the cxdb command in background.

CXdb executes all the options specified with the cxdb command before reading any input from the included command file batch.input. The last line of the batch.input file is the quit command. The quit command is needed to exit from batch mode.

Note

Batch mode also allows you to enter your input directly in the shell window rather than reading it in from a command file. The command cxdb -b is sufficient to activate CXdb in this mode. This is not completely interactive execution because the CXdb prompts do not appear in the shell window, but the CXdb output does.

Quitting the examples

If you have been working the examples in this chapter, you have created some log files in your home directory. You can delete these files and exit CXdb by issuing the commands shown in Figure 282.

Figure 282
Deleting the log files and quitting CXdb

```
(CXdb) cd ~
(CXdb) shell 'rm output.log error.log input.log message.log'
(CXdb) shell 'rm session.cxdb batch.output process.log'
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

The cd command in Figure 282 changes directories to your home directory. The shell commands invoke the shell and pass it commands to delete (rm) the log files created by the examples in this chapter. The quit command exits CXdb.

Scope paths and the stack 13

Chapter 7 gave examples of printing the values of variables local to the current routine. With scope paths, you can reference variables in any part of a program. Scope paths enable you to specify static variables, common block variables, loader symbols, and even variables defined in other languages.

This chapter covers the use of scope paths and the stack. The following commands are introduced:

- `backtrace`
- `frame`
- `info frame`
- `info frame at`
- `info scope`

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 283.

Figure 283
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 283 changes directories to the directory where the example program files are stored. The `cxdb a.out` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 284.

Figure 284
Starting the example program

```
(CXdb) break line chapter13F.f:15 {remove event $self;}
#0: break line, on [#0/*], Enabled, ignore 0/0
      [0x80003510] CHAPTER13F in chapter13F.f line 15
   {
      remove event $self;
   }
(CXdb) run
Starting process [#0]: a.out
Eventpoint 0 removed
```

The `break routine` command in Figure 284 sets a breakpoint at line 15 of the file `chapter13F.f`. The breakpoint's handler removes the breakpoint after it is triggered. The `run` command runs the example program. The program stops executing when it reaches the breakpoint and then the breakpoint is removed.

Scope

A declaration binds an identifier to a variable. A language's scope rules determine where in a program a declaration is in effect. A variable is said to be visible where its declaration is in effect.

When you use an identifier in a CXdb command, the variable the identifier denotes is determined by the declaration that is in effect at the current scope. The program counter (PC) of the current stack frame is mapped to a source unit. The scope of this source unit is the current scope, and it determines which variables are visible.

When execution stops (at a breakpoint, for example), a stack frame is selected to be the current stack frame. This is normally the last frame pushed onto the stack, frame 0, and is located at the top of the stack. You can change the current stack frame to a different frame on the stack. The program counter from the current stack frame determines the current scope. Thus, changing stack frames also changes the current scope.

Referencing a variable that is not visible

When debugging, you may want to reference a variable that is not visible from the current scope. You can reference the variable by preceding its identifier with a scope path. Scope paths enable you to reference a variable from any part of the program, regardless of whether or not it is currently visible.

Evaluating a variable that is not visible

A variable that is visible has storage allocated for it and can usually be evaluated. Variables that are not visible, but have storage allocated for them, can be evaluated in the following cases:

- **Allocated in global memory**—Variables allocated storage in global memory include external variables, static variables, and FORTRAN local variables (unless they were compiled using the `-re` option). Variables whose storage is in global memory can be evaluated by using a scope path.
- **Allocated on the stack**—Variables allocated on the stack include C local variables, routine arguments, and FORTRAN local variables compiled using the `-re` option. There are two cases when stack-based variables are not visible:
  - **Shadowed in the current stack frame**—A variable that is shadowed in the current stack frame may be evaluated by specifying a scope path.
  - **Allocated in a different frame**—A variable that is allocated in a frame other than the current stack frame is evaluated by changing to the appropriate frame.

Scope paths

A scope path specifies a scope in which a variable is visible. An identifier preceded by a scope path is called a qualified identifier. There are two types of qualified identifiers:

- **Fully qualified**—A fully qualified identifier is one whose scope path includes the language and complete syntactic location of the variable. A fully qualified identifier can be used to reference a variable in any scope.
- **Partially qualified**—A partially qualified identifier is one whose scope path only partially describes the syntactic location of the identifier. The current stack frame determines the initial scope with which to interpret the path.

A generic scope path is shown in Figure 285. The scope path must include the components needed to uniquely specify a variable.

Figure 285
A generic scope path

```
[language-specifier$] [scope-block`] [...]
```

The parts of a generic scope path are described below:

- *language-specifier*—The language in which the identifier is declared. The possible language specifiers are:
  - `f$`—FORTRAN program variables.
  - `c$`—C program variables.
  - `l$`—Loader symbols.
  - `cxdb$`—Debugger variables, including predefined variables such as `$signal` and debugger variables you create.
  - `s$`—Synthesized variables.
- *scope-block*—A scope block is an artificial name that enables you to specify a particular scope, such as a routine, or source file. It may take multiple scope blocks to fully specify the desired scope. For example, in C, a file name and a routine name are necessary to specify variables in other source files.

The `f$` and `c$` language specifiers are not needed if the variable you are trying to reference is defined in the current language and the first specified scope block is visible. Thus, these prefixes are usually only necessary in mixed language programs.

The `cxdb$` prefix can be abbreviated to `$`. Synthesized variables are created by the compiler when a program is optimized. They are covered in detail in Chapter 15, Section "Synthesized variables."

Use of scope paths

Scope paths can precede an identifier in any language expression. Typically, scope paths are used with the `print` and `info expression` commands. You can always reference an identifier with the `info expression` command. However, you can only evaluate a variable whose storage is available.

The structure of scope paths differ for FORTRAN and C because the rules of scope are different between the two languages. Therefore, the two languages are treated separately in the next two sections of this chapter. If you work primarily with FORTRAN code, proceed to the section titled "FORTRAN scope paths." If you work primarily with C programs, proceed to the section titled "C scope paths."

FORTRAN scope paths

In FORTRAN, scope paths enable you to reference a variable in a different routine, particularly the variables of a common block in a different routine. In FORTRAN, only local variables compiled with the `-re` option are allocated on the stack.

A generic FORTRAN scope path is shown in Figure 286.

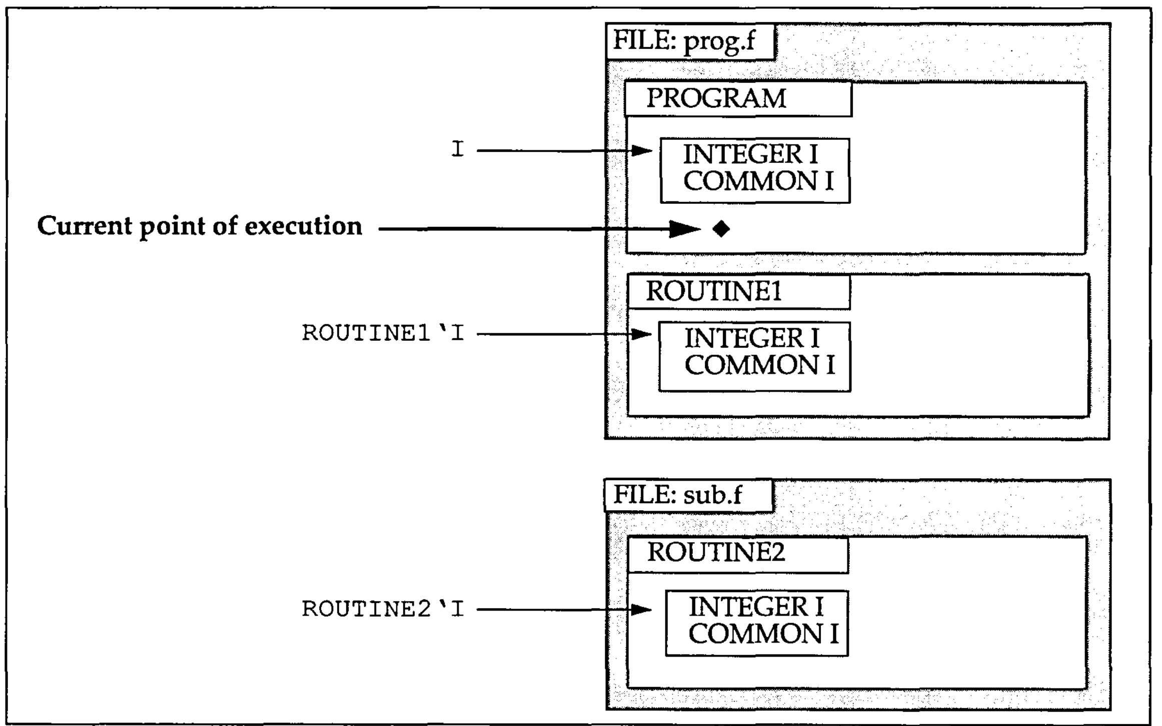
Figure 286
A generic FORTRAN scope path qualifying an identifier

```
f$routine-name`identifier
```

The routine name specifies the routine in which the identifier is declared. Note that file names are not necessary because all routines are global in FORTRAN. If the `f$` is omitted, the identifier is partially qualified and the current scope determines the language of the identifier.

Figure 287 shows generic examples of FORTRAN scope paths.

Figure 287
Generic example of FORTRAN scope paths



Getting the current scope

The info scope command displays the scope path of the current scope, as illustrated in Figure 288.

Figure 288
Displaying the current scope

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$CHAPTER13F
```

The info scope command in Figure 288 shows that the current scope path is f$CHAPTER13F, and frame 0 is the current stack frame that determines this scope path. In the scope path name, the f$ indicates that FORTRAN is the current language. CHAPTER13F is the current routine.

Printing a variable from another routine

To reference a variable that is not visible within the current scope, you must qualify its identifier with a scope path. Unqualified variables are interpreted in the context of the current scope. Figure 289 illustrates the use of scope paths with the print command.

Figure 289
Referencing variables outside the current scope

```
(CXdb) step 2
Stepping process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x8000368c] SUB13C in chapter13F.f line 64
(CXdb) print I
(INTEGER*4) -1
(CXdb) print CHAPTER13F'I
(INTEGER*4) 1
(CXdb) print SUB13B'I
(INTEGER*1) 7
```

In Figure 289, the step 2 command steps execution into the SUB13C routine, changing the current scope to this routine.

The print I command displays the value of the variable I local to the SUB13C routine. The print CHAPTER13F'I command prints the variable I local to the CHAPTER13F routine. The print SUB13B'I command prints the value of the variable I local to the SUB13B routine. All three identifiers, though named the same, denote distinct variables.

Referencing variables that cannot be evaluated

In FORTRAN, the local variables of a routine compiled with the `-re` option are allocated on the stack. If the frame for this routine is not the current stack frame, these variables cannot be evaluated solely with a scope path, because the appropriate stack frame is not selected. The routine `SUB13D` of the example program was compiled with the `-re` option. Its local variable `I` can be referenced but not evaluated, as shown in Figure 290.

Figure 290
Referencing a variable without storage available

```
(CXdb) print SUB13D'I
ERROR: 196
Variable's storage is not available, line: 1 col: 7, I.
(CXdb) info expression SUB13D'I
ERROR: 196
Variable's storage is not available, line: 1 col: 17, I.

object type: FORTRAN identifier
   location: <none>
       size: 4 bytes
       type: INTEGER
      value: <unknown>
       1 liveness ranges:
               Start       End          Location
         1. 0x8000371a:0x80003728 - @($fp-4)
```

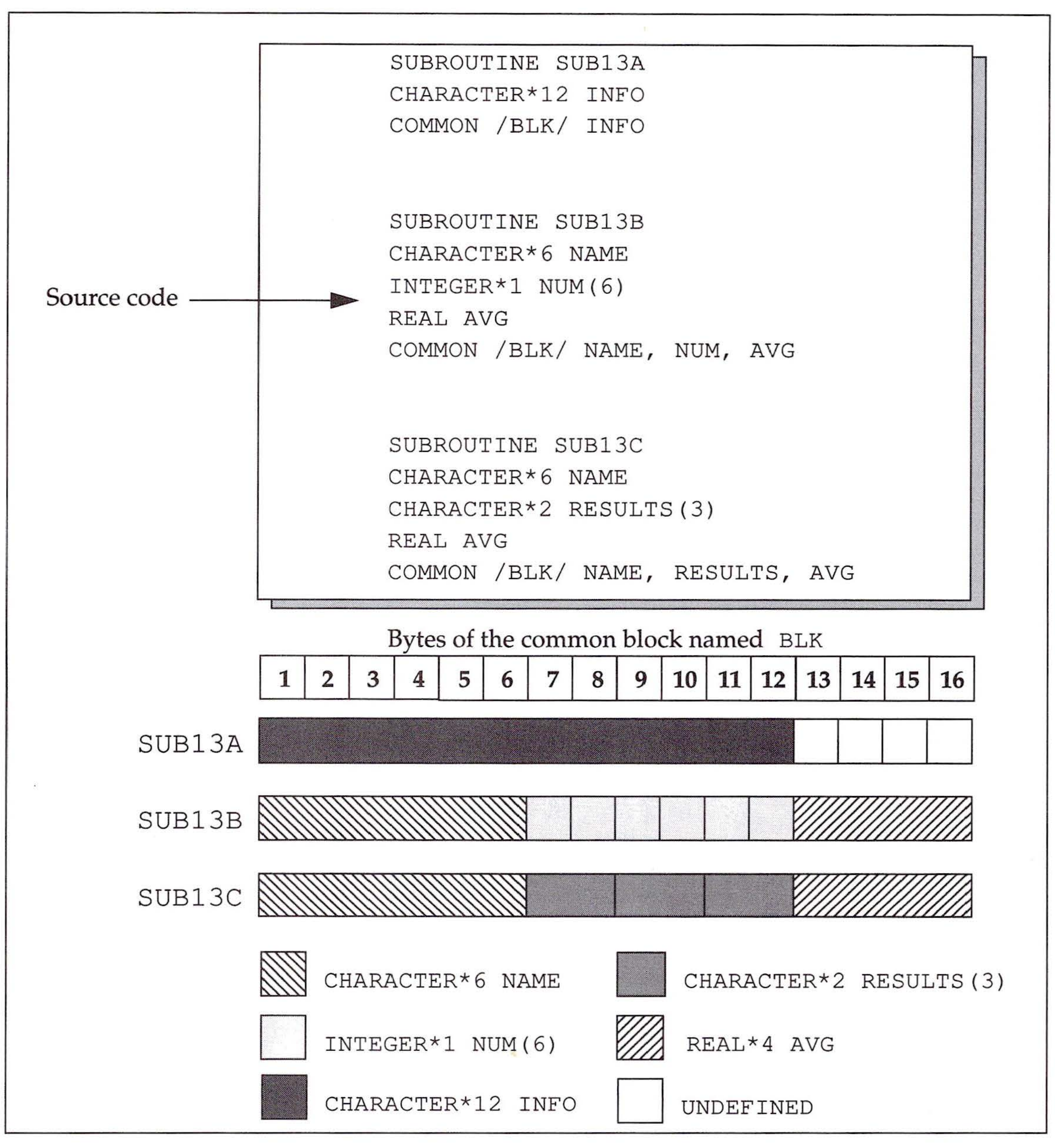
The `print SUB13D'I` command attempts to evaluate the local variable `I` of the `SUB13D` routine. CXdb displays a message indicating that the variable's storage is not available. This is because the variable's storage is stack-based, and the `SUB13D` routine is no longer on the stack.

The `info expression SUB13D'I` command displays information about the variable `I`. The variable can still be referenced using a scope path, but not evaluated.

Using scope paths to view common blocks

Scope paths are especially useful in printing variables that are part of a common block. Common blocks may be declared differently in different routines. In the example program, each of the subroutines in `chapter13F.f` declares the common block `BLK` differently, as shown in Figure 291.

Figure 291
Subroutines' use of the common block BLK



All three subroutines manipulate the data stored in the same common block. However, as shown in Figure 291, each subroutine divides the common block in different ways. You can use scope paths to reference the common block as it is declared in each of the different routines, as shown in Figure 292.

Figure 292
Referencing variables of a common block

```
(CXdb) print SUB13A'INFO
(CHARACTER*12) 'CONVEX999897'
(CXdb) print SUB13B'NUM
INTEGER*1(1:6)
(1..3)= 57
(4)=56
(5)=57
(6)=55

(CXdb) print SUB13C'RESULTS
CHARACTER*2(1:3)
(1)='99'
(2)='98'
(3)='97'
```

In Figure 292, the `print` commands all reference the same memory, through different common block variables.

C scope paths

In C, scope paths allow you to reference variables in different routines and files. Local variables in C are allocated on the stack, unless declared `static`. Scope paths also enable you to reference shadowed identifiers.

A generic C scope path is shown in Figure 293.

Figure 293
A complete C scope path qualifying an identifier

c$*file-name*'*routine-name*'*block-list*'*identifier*

- *file-name*—The file containing the routine in which the variable is declared. The file name can be omitted if the routine is visible.
- *routine-name*—The routine in which the variable is declared. The routine name can be omitted if the block or identifier is visible.
- *block-list*—A list of numbers specifying the block source unit in which the variable is declared. Block numbers are discussed in more detail in Section "Block numbering."

Figure 294 provides some generic examples of ANSI C scope paths.

Figure 294
Generic example of ANSI C scope paths

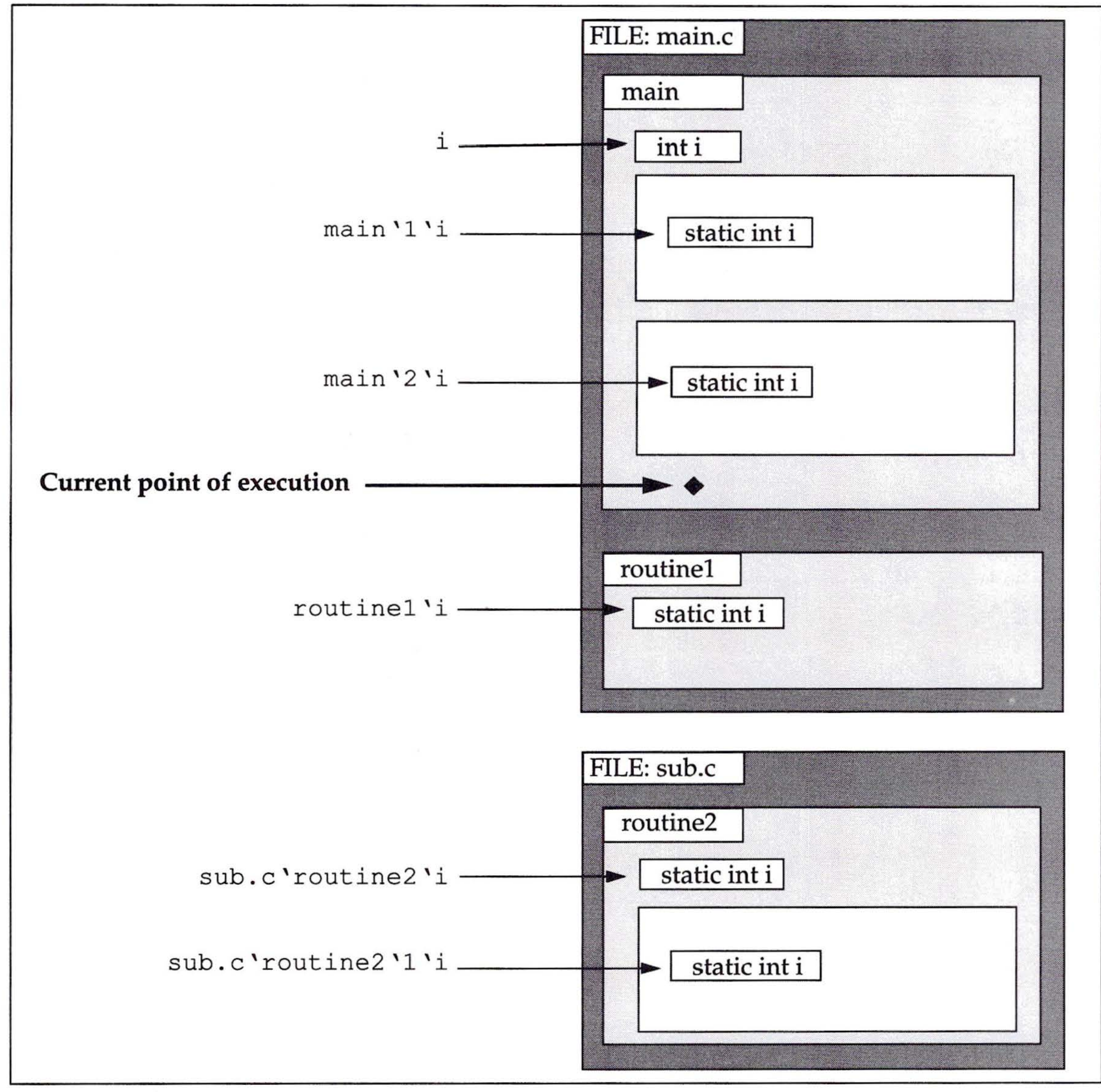


Figure 294 shows the scope paths that would be used to reference the different variables named `i`, if execution was stopped at the end of the main routine.

The example program contains the routine `chapter13c_`, which is written in C. Figure 295 shows how to set a breakpoint in this part of the program.

Figure 295
Starting the C routine

```
(CXdb) break line chapter13C.c:27 {remove event $self;}
#1: break line, on [#0/*], Enabled, ignore 0/0
      [0x80004722] chapter13C`chapter13c_`1 in chapter13C.c line 27
   {
      remove event $self;
   }
(CXdb) continue
Resuming execution of Process [#0/*]
Eventpoint 1 removed
```

The `break line` command in Figure 295 sets a breakpoint at line 27 of the routine `chapter13c_`. The `continue` command resumes process execution. Execution stops at the new breakpoint, which is then removed, as specified in the handler.

Getting the current scope

The `info scope` command displays the scope path of the current scope, as illustrated in Figure 296.

Figure 296
Displaying the current scope

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$chapter13C`chapter13c_`1
```

The `info scope` command in Figure 296 shows that the current scope is `c$chapter13C`chapter13c_`1`, and that frame 0 is the current stack frame. The `c$` indicates that C is the language of the current routine. The current routine is `chapter13c_` in the file `chapter13C.c`.

Printing a variable from another routine

To reference a variable that is not visible within the current scope, you must qualify the variable name by specifying its scope path. In C, only static variables and external variables are active outside the current scope. Figure 297 illustrates the use of scope paths to reference static variables.

Figure 297
Referencing variables outside the current scope

```
(CXdb) step 2
Stepping process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x80004bf6] chapter13C'subxc in chapter13C.c
(CXdb) print i
(int) -1
(CXdb) print chapter13C'chapter13c_'i
(int) 1
(CXdb) print chapter13C'sub13b'i
(int) 3
(CXdb) print chapter7C'chapter7c_'i  <---- Variable
(int) 4
```


In Figure 297, the `step` command steps execution into the `subxc` routine. The `print i` command prints the value of the local variable `i`.

The `print chapter13C'chapter13c_'i` command prints the value of the static variable `i` from the routine `chapter13c_` of the file `chapter13C.c`. The remaining two `print` commands print the value of the variable `i` from the `subxb` routine and `chapter7c_` routine, respectively.

Referencing variables that cannot be evaluated

In C, local variables are allocated on the stack, unless they are declared `static`. If the local variables are not allocated in the current stack frame, these variables cannot be evaluated solely with a scope path because the appropriate stack frame is not selected. The local variable named `data` of the `chapter13c_` routine can be referenced using scope paths, but not evaluated, as shown in Figure 298.

Figure 298
Referencing a variable without global memory

```
(CXdb) print chapter13C'chapter13c_'data
ERROR: 196
Variable's storage is not available, line: 1 col: 7, data.
(CXdb) info expression chapter13C'chapter13c_'data
ERROR: 196
Variable's storage is not available, line: 1 col: 17, data.

object type: C identifier
   location: <none>
       size: 24 bytes
       type: struct info_block
      value: <unknown>
       1 liveness ranges:
              Start        End         Location
         1. 0x80004660:0x800047c6 - @($fp-24)
```

The print chapter13C'chapter13_'data command attempts to evaluate the variable data of the chapter13_ routine. CXdb displays a message indicating that the variable's storage is not available from the current stack frame. However, the identifier can be referenced, as shown by the info expression command, but again, its value cannot be determined.

Block numbering

In C, you can declare variables inside of a block. Such variables are local to the block. If multiple variables with the same name are declared in nested blocks, only the variable of the innermost block is visible. The other variables are said to be shadowed, and are not visible. Because of this, a special mechanism exists to allow you to specify the block of a variable.

Blocks that have at least one variable declared in them are numbered by the C compiler. The blocks are numbered consecutively if they are at the same level, and the first block in the routine is numbered 1. Each new level of nesting starts over at 1.

The numbering of the blocks is different between ANSI C mode and pcc mode. In pcc mode, the block defining the routine is counted; in ANSI C mode it is not.

The example program has two nearly identical routines, `ansi_block` and `pcc_block`. The `ansi_block` routine has been compiled in ANSI C mode, and the `pcc_block` routine has been compiled in `pcc` mode (compiled with the `-pcc` flag). The next two sections provide examples of scope paths using block numbers in the respective modes.

Block numbering in ANSI C

In ANSI C, the block defining the routine is not counted. Therefore, the first block inside of the routine is block 1, as shown in Figure 299.

Figure 299
ANSI C mode block numbering

```
ansi_block(i)
int i;
{
    i = 0 ;                                      No block number

    if (i == 0)
    {
        int i=1 ;                                Block 1

        if (i == 1)
        {
            int i=11 ;                           Block 1'1
        }

        if (i == 1)
        {
            int i=12 ;                           Block 1'2
        }
    }
Current PC --> printf("\nThe routine ansi_block is finishing.\n");
}
```

Figure 300 shows how to use scope paths to reference the different variables (all named `i`) in the `ansi_block` routine.

Figure 300
Using scope paths in ANSI C mode

```
(CXdb) break line ansi_block.c:20

#2: break line, on [#0/*], Enabled, ignore 0/0
    [0x80004da2] ansi_block'ansi_block in ansi_block.c line 20
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 2, at [0x80004da2] ansi_block'ansi_block
(CXdb) print ansi_block'i
(int) 0
(CXdb) print ansi_block'1'i
(int) 1
(CXdb) print ansi_block'1'1'i
(int) 11
(CXdb) print ansi_block'1'2'i
(int) 12
```


The `print` commands in Figure 300 use scope paths with block numbers to reference the different variables named `i` in the routine.

Block numbers in `pcc` mode

In `pcc` mode, the block of the routine is counted as block 1. This is because the local variables of a routine have a different scope than the arguments to the routine. The local variables of the routine are declared in block 1. The arguments to the routine are not given their own block. This is shown in Figure 301.

Figure 301
pcc mode block numbering

```
pcc_block(i)
int i;
{
   int i = 1 ;                              block 1

   if (i == 1)
   {
      int i=11 ;                            block 1`1

      if (i == 11)
      {
         int i=111 ;                        block 1`1`1
      }

      if (i == 11)
      {
         int i=112 ;                        block 1`1`2
      }
   }
Current PC ---> printf("\nThe routine pcc_block is finishing.\n");
}
```

Figure 302 illustrates how to use scope paths to reference the different variables (all named i) in the pcc_block routine.

Figure 302
Using -pcc mode scope paths

```
(CXdb) break line pcc_block.c:20
#3: break line, on [#0/*], Enabled, ignore 0/0
        [0x80004e62] pcc_block`pcc_block`1 in pcc_block.c line 20
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 3, at [0x80004e62] pcc_block`pcc_block`1
(CXdb) print pcc_block`i   <----- Argument to the routine
(int) 0
(CXdb) print pcc_block`1`i   <----- Local variable of the routine
(int) 1
(CXdb) print pcc_block`1`1`i
(int) 11
(CXdb) print pcc_block`1`1`1`i
(int) 111
(CXdb) print pcc_block`1`1`2`i
(int) 112
```

The print commands in Figure 302 print the values of all the different variables named i. Because the routine block is counted, the scope paths to all of the variables in the routine are affected.

Printing variables from a different language

Scope paths allow you to reference variables that are in routines written in a different language. As with variables defined in the current language, variables must be either global, statically allocated, or on the stack to ensure that you can print their values.

When you use a scope path to qualify a variable in another language, the scope path must conform to the rules of the variable's language. Figure 303 demonstrates how to reference C variables from a FORTRAN routine.

Figure 303
Referencing C variables from a FORTRAN routine

```
(CXdb) break routine f$CHAPTER13F

#4: break routine, on [#0/*], Enabled, ignore 0/0
       [0x80003436] CHAPTER13F in chapter13F.f line 6
(CXdb) run
Process [#0] is already running with pid 3796.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 4, at [0x80003436] CHAPTER13F in chapter13F.f
(CXdb) print c$chapter13C'chapter13c_'i
(int) 0
```

C language identifier

The break routine command in Figure 303 sets a breakpoint at the beginning of the CHAPTER13F FORTRAN routine. The run command creates a new process, and execution stops at the beginning of the CHAPTER13F routine.

The print c$chapter13C'chapter13c_'i command prints the value of the C language variable named i from the chapter13c_ routine in the chapter13C.c file. The c$ prefix indicates to CXdb that the variable is in the C language.

Figure 304 demonstrates how to reference FORTRAN variables from a C routine.

Figure 304
Referencing FORTRAN variables from a C routine

```
(CXdb) break routine chapter13c_
#5: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80004664] chapter13C'chapter13c_ in chapter13C.c line 18
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 5, at [0x80004664] chapter13C'chapter13c_
(CXdb) print f$CHAPTER13F'I
(INTEGER*4) 4
```

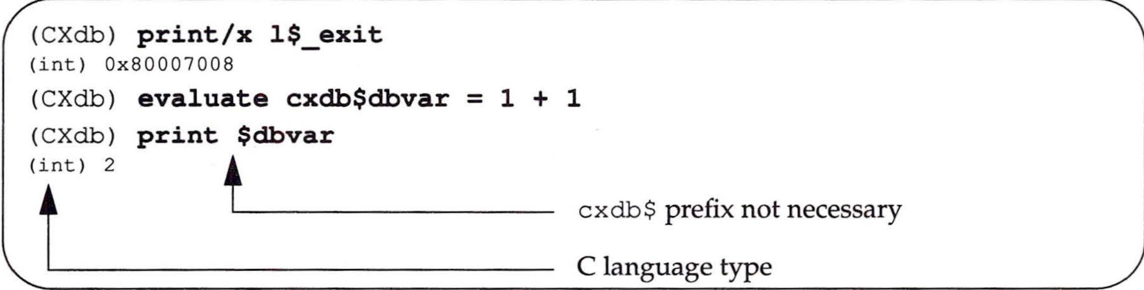
FORTRAN language identifier

In Figure 304, the `break routine` command sets a breakpoint at the start of the `chapter13c_` routine. The `continue` command causes execution to continue until it is stopped by breakpoint 5.

The `print f$CHAPTER13F'I` command prints the value of the variable `I` from the `CHAPTER13F` routine. The `f$` prefix indicates to CXdb that the variable is in the FORTRAN language. Because the scope path is a FORTRAN scope path, a file name is not necessary.

Figure 305 shows how to reference loader symbols and debugger variables.

Figure 305
Referencing loader symbols and debugger variables



The `print/x l$_exit` command in Figure 305 prints the value of the loader symbol `_exit` as a hexadecimal number.

The `evaluate` command creates the `$dbvar` debugger variable. The debugger variable's type is dependent upon the current language (in this case, C). The `print` command prints the value of the new debugger variable.

Changing frames to reference variables

Another way to reference a remote identifier is to change stack frames to the frame where the identifier is visible. Because the current stack frame determines the current scope, changing stack frames also changes the current scope.

Generally, when execution stops, the last frame pushed onto the stack is the current stack frame, and therefore determines the current scope.

Enter the commands shown in Figure 306 to prepare the examples for this section.

Figure 306
Preparing to change stack frames

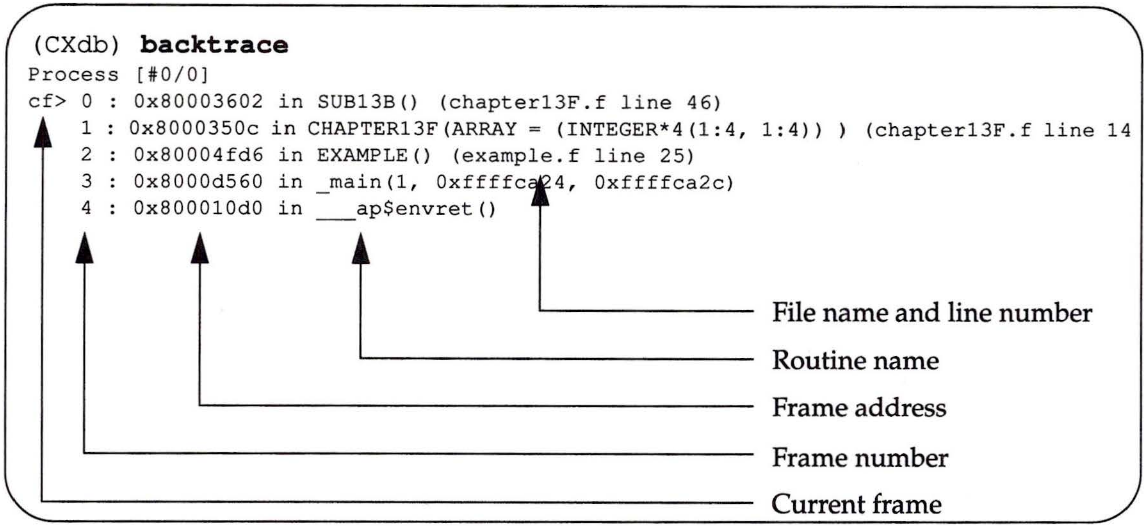
```
(CXdb) run
Process [#0] is already running with pid 3796.
Terminate existing process and restart? y
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 4, at [0x80003436] CHAPTER13F in chapter13F.f
(CXdb) step loop 2
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x80003602] SUB13B in chapter13F.f line 46
(CXdb) info scope
Process [#0/0], frame 0 scope: f$SUB13B
(CXdb) print INFO_BLOCK

ERROR: 181
Identifier not visible from the current lexical scope, line: 1 col: 7 'INFO_BLOCK'
```

The `run` command restarts execution. Execution is stopped by breakpoint 4, which was set in Figure 303. The `step loop 2` command steps execution into the `SUB13B` routine.

The `info scope` command shows that the current scope is the `SUB13B` routine. The `print` command attempts to print the variable `INFO_BLOCK`, which is declared in the `CHAPTER13F` routine. Because the current scope does not include the `CHAPTER13F` routine, the variable is not found.

You can obtain information about the stack and the frames of the stack. The `backtrace` command displays a list of all the frames of the stack, as shown in Figure 307.

Figure 307
Getting information about a stack frame



The `backtrace` command in Figure 307 displays a list of the frames currently on the stack. The display includes the frame number, address, and corresponding routine (and its arguments) for each frame. The file name and line number for each routine (where applicable) are listed.

When looking at a backtrace, it is important to remember that each frame was called by the next highest numbered frame. Thus, frame 0 was called by frame 1, and frame 1 was called by frame 2. Routines that begin with an underscore character (_) are generally assembly-language routines that control the starting and exiting of the process.

You can get more information about a particular frame on the stack by using the `info frame` and `info frame at` commands. The `info frame` command displays the contents of a particular frame on the stack. The `info frame at` command displays information about a frame at a particular address. These commands are demonstrated in Figure 308.

Figure 308
Getting information on a specific frame

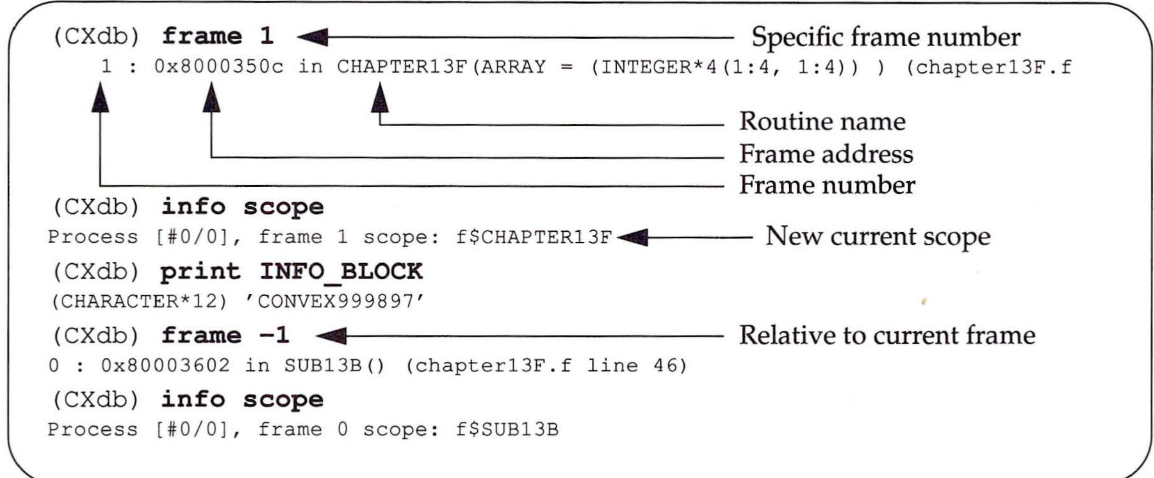
```
(CXdb) info frame 2 ◄──────────── Specific frame number
Process [#0/0]
Frame : 2; [0x80004fd6] EXAMPLE in example.f line 25
Frame address : 0xffffc9b0
Saved Registers : pc=0x80004fd6 psw=0x7909400 fp=0xffffc9c0 ap=0x8000539c
Floating point mode : NATIVE; Language : FORTRAN
Number of arguments : 0
(CXdb) info frame at $fp ◄──────────── Frame address
Process [#0/0]
[0x8000350c] CHAPTER13F in chapter13F.f line 14
Frame address : 0xffffc9a0
Saved registers : pc=0x8000350c psw=0x87109480 fp=0xffffc9b0 ap=0x800038d8
Floating point mode : NATIVE; Language : FORTRAN
Number of arguments : 1
```

The `info frame` command in Figure 308 requests information on frame 2. CXdb responds by displaying the frame's number, address, saved registers, floating point mode (`IEEE` or `NATIVE`), language, and number of arguments.

The `info frame at` command requests information on the frame located at a particular address. It can be used to determine the frame that the frame pointer, `$fp`, points to, or to interpret the contents of a block of memory as a stack frame. The information displayed is in the same format as the information displayed by the `info frame` command.

Because the current frame determines the current scope, you can change the current scope by changing the current frame. This allows you to reference variables in another scope without having to use a scope path. Figure 309 demonstrates the use of the `frame` commands.

Figure 309
Changing stack frames to reference a variable



The `frame 1` command changes the current frame to frame 1. This also causes the current scope to change, as shown by the `info scope` command.

Because the current scope is now `CHAPTER13F`, the variable `INFO_BLOCK` can now be referenced without a scope path. The `frame -1` command decrements the current frame number by 1. Thus, the current frame again becomes frame 0. The second `info scope` command verifies that the current scope is the `SUBXB` routine.

Note

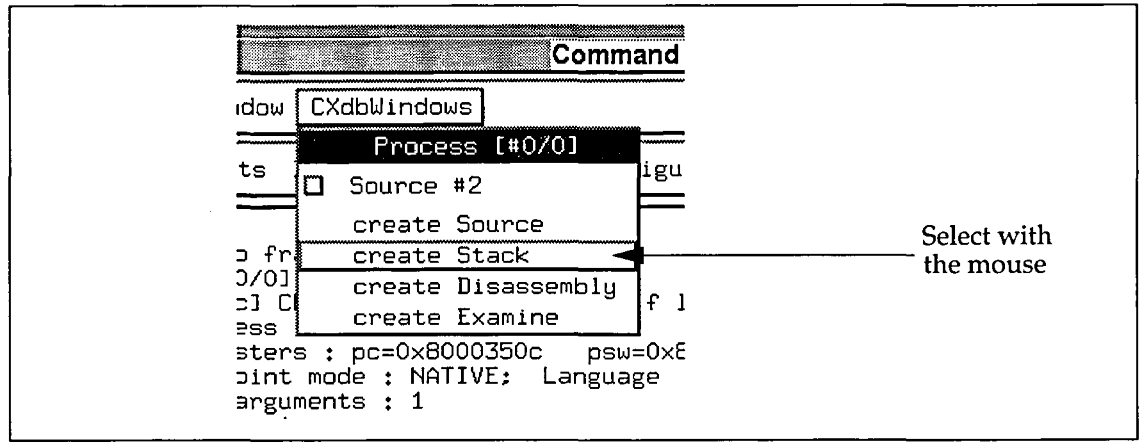
Process execution always proceeds from frame 0, regardless of which frame is selected as the current stack frame.

Stack window

The stack window is only available in CXwindows. It displays the frames of the program stack. It is opened from the CXdbWindows menu of the command window or from the ProcessWindows menu of the source window.

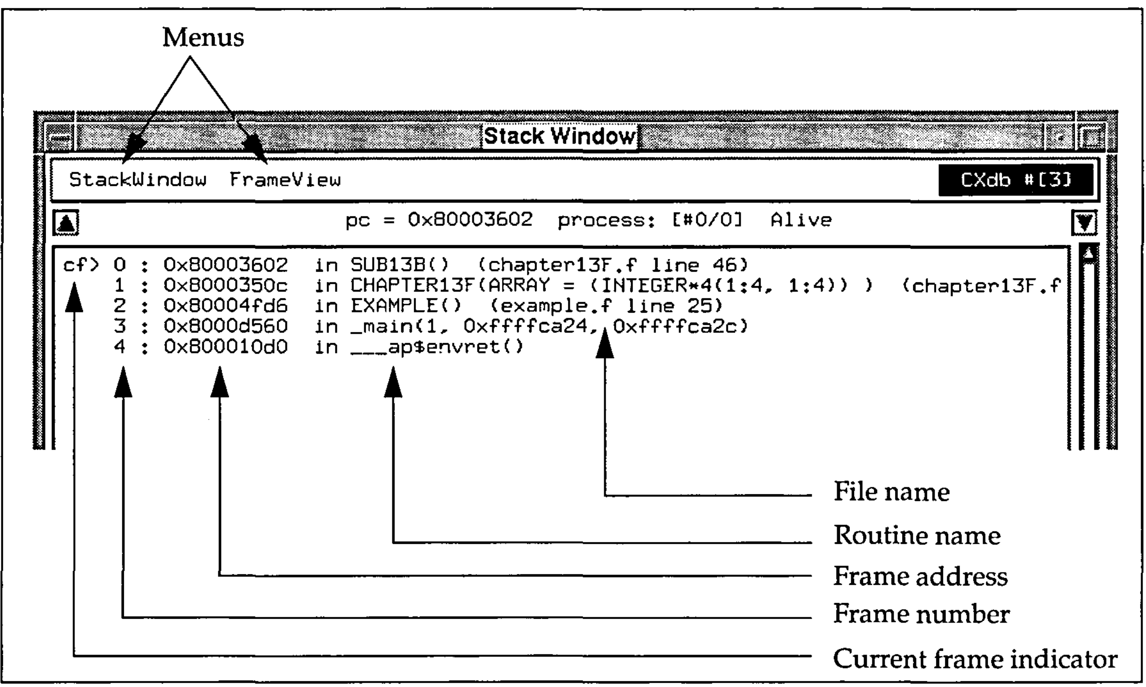
Figure 310 shows how to open the stack window from the CXdbWindows menu.

Figure 310
Opening the stack window



The stack window is shown in Figure 311.

Figure 311
The stack window



The stack window displays the current frame indicator (`cf>`). It also displays the number, address, routine name, and file name for each frame on the stack.

You can use the stack window to track the state of the program stack, and to obtain information about the frames on the stack. To open a stack frame description window, click the left mouse button on a line shown in the stack window. The stack frame description window for the `CHAPTER13F` routine is shown in Figure 312.

Figure 312
Stack frame description window

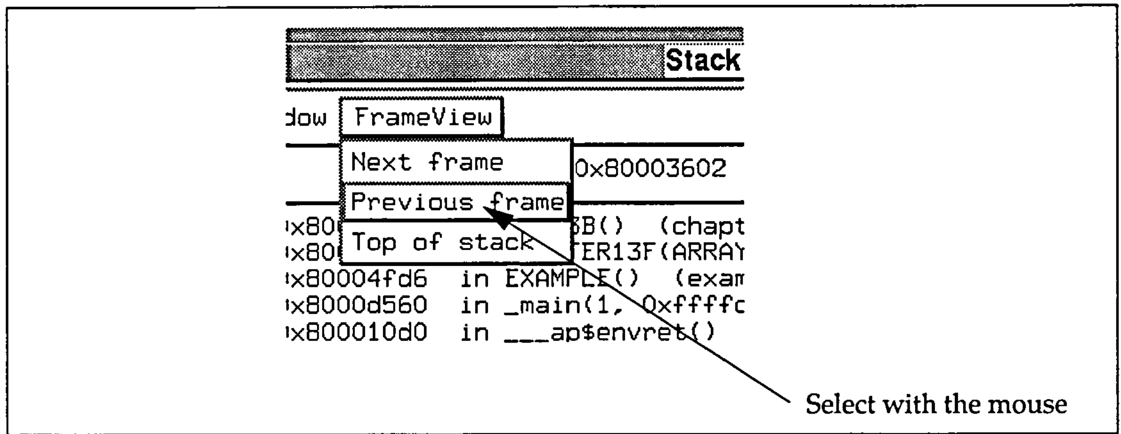
```
Stack Frame Description

Process [#0/0]
Frame : 1; [0x8000350c] CHAPTER13F in chapter13F.f line 14
Frame address : 0xffffc9a0
Saved Registers : pc=0x8000350c   psw=0x87109480   fp=0xffffc9b0   ap=0x800038d8
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 1

Number of arguments : 1

     1 : ARRAY= INTEGER*4(1:4, 1:4) 0x8006dce8

Number of locals : 2

     1 :  I = (INTEGER*4) 1
     2 :  INFO_BLOCK = (CHARACTER*12) 'CONVEX999897'

Close
```

The stack frame description window shown in Figure 312 was created by clicking with the left mouse button anywhere on the line describing the stack frame for `CHAPTER13F` in the stack window. The stack frame description window displays information about the frame itself, as well as information on the routine's local variables and arguments (if any).

You can use the stack window to change the current frame of the process stack. Using FrameView menu options, you can set the current frame to the previous frame or next frame on the stack. The Top of Stack option, in the FrameView menu, resets the current frame to 0.

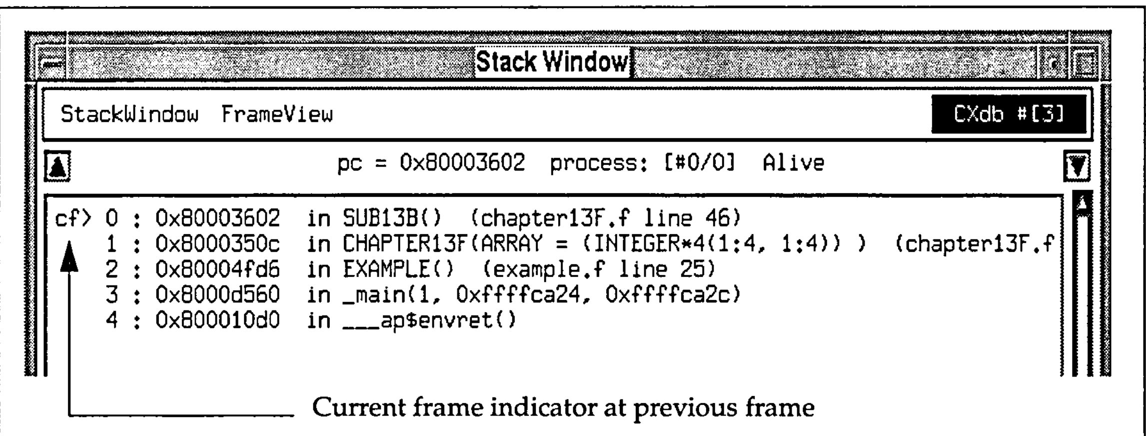
Figure 313 illustrates how to select the previous frame (the frame below the current frame) in the stack window.

Figure 313
Changing stack frames with the stack window



When you select the Previous Frame option, from the FrameView menu, the stack window updates to show the new position of the current frame indicator (`cf>`), as shown in Figure 314.

Figure 314
Result of selecting the Previous frame menu option



The new position of the current frame pointer is shown in Figure 314. Initially, the stack window updates to reflect the current state of the stack each time process execution stops. You can set the stack window to remain static by toggling the auto update option from the StackWindow menu. You can enable auto-updating of the stack window in the same manner.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 315.

Figure 315
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Debugging at the instruction level

14

This chapter explains features of CXdb that help you to debug code at the machine instruction level. These features include highlighting expression source units, displaying disassembled code, and stepping by instruction.

This chapter covers the following commands:

- `disassemble`
- `next expression`
- `next instruction`
- `step expression`
- `step instruction`

This chapter also covers the disassembly window, which is available through the CXwindows interface.

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 316.

Figure 316
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 316 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 317.

Figure 317
Starting the example program

```
(CXdb) break line chapter7F.f:39
#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80002952] BLD_MATRIX in chapter7F.f line 39
(CXdb) run
Starting process [#0]: a.out
Process [#0/0] stopped by Bkpt 0, at [0x80002952] BLD_MATRIX in chapter7F.f line 39
```

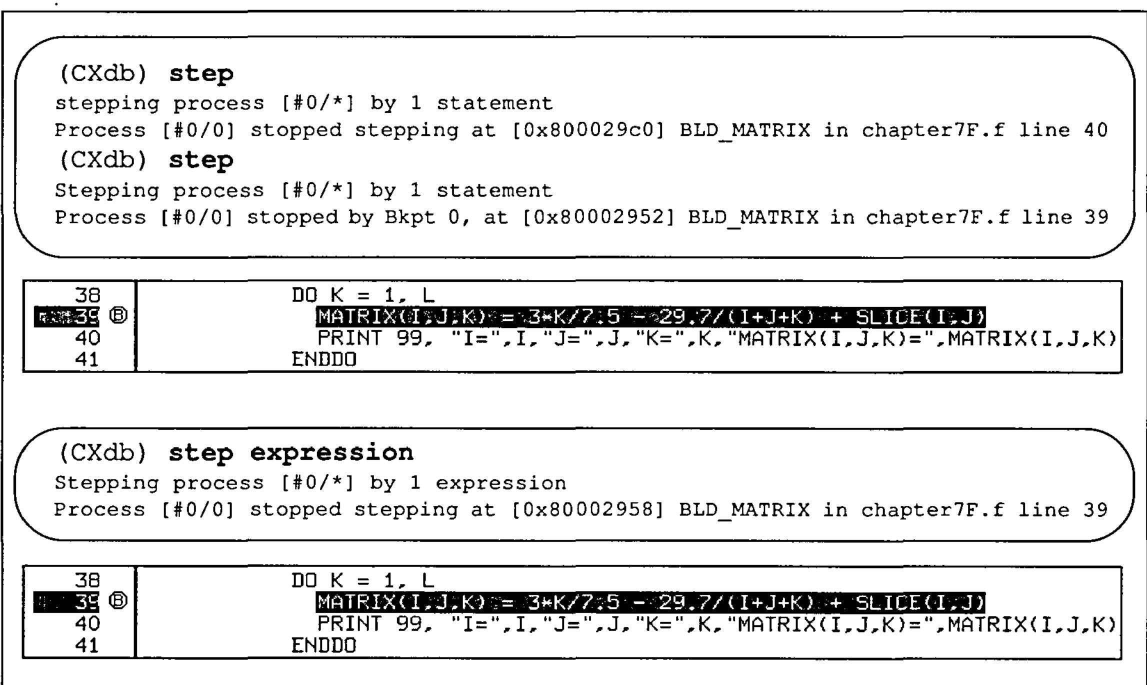
The `break line` command in Figure 317 sets a breakpoint at the beginning of line 39 in the file `chapter7F.f`. The `run` command runs the example program. The program stops executing when it reaches the breakpoint.

Stepping by expression

As discussed in Chapter 5, CXdb stepping commands give you control over the stepping granularity. The smallest granularity is expression, which enables you to step a process by a single source language expression.

Stepping by expression is particularly helpful if you want to see how a complicated source statement is evaluated. Figure 318 and Figure 319 illustrate this point.

Figure 318
Stepping by expression with a default granularity of statement



Because the default stepping granularity is set to statement, the first two `step` commands in Figure 318 step the process by one statement each. In the source window, the entire statement at line 39 is highlighted. Line 39 is a complicated statement, and stepping through it by expression would provide more insight into how the expressions within the statement are evaluated.

The `step expression` command in Figure 318 steps the process by only one expression. However, because the default granularity is still set to statement, the entire statement at line 39 is highlighted in the source window. To make the highlighting more detailed, set the stepping granularity to expression, as shown in Figure 319.

Figure 319
Stepping by expression with a default granularity of expression

```
(CXdb) set step expression
(CXdb) step
Stepping process [#0/*] by 1 expression
Process [#0/0] stopped stepping at [0x8000295e] BLD_MATRIX in chapter7F.f line 39
```

```
38        DO K = 1, L
39 (B)      MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
40          PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
41        ENDDO
```

```
(CXdb) step
Stepping process [#0/*] by 1 expression
Process [#0/0] stopped stepping at [0x80002964] BLD_MATRIX in chapter7F.f line 39
```

```
38        DO K = 1, L
39 (B)      MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
40          PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
41        ENDDO
```

```
(CXdb) step
Stepping process [#0/*] by 1 expression
Process [#0/0] stopped stepping at [0x8000296a] BLD_MATRIX in chapter7F.f line 39
```

```
38        DO K = 1, L
39 (B)      MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
40          PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
41        ENDDO
```

The `set step` command in Figure 319 changes the stepping granularity to expression. This causes the source window to highlight expression source units rather than statement source units. Because the default granularity is now set to expression, each `step` command in Figure 319 steps the process by one expression. The highlighting in the source window shows that the expression for `I` is evaluated, then the expression for `3*K`, and then the expression for `I+J+K`.

The `next expression` command can also be used for stepping by expression. When this command encounters a subroutine call, it executes the subroutine but does not count any of its expressions as part of the step. The `step expression` command, on the other hand, counts all expressions in both the calling and the called routines.

Both the `step expression` and `next expression` commands accept a repeat count. Figure 320 illustrates the use of a repeat count with the `next expression` command.

Figure 320
Using a repeat count with the `next expression` command

```
(CXdb) next expression 3
Nexting process [#0/*] by 3 expressions
Process [#0/0] stopped nexting at [0x80002974] BLD_MATRIX in chapter7F.f line 39
```

Displaying disassembled code

An important aid to instruction level debugging is the `disassemble` command. With this command, you can display the assembly language instructions that are generated from your source code. Figure 321 illustrates the use of the `disassemble` command.

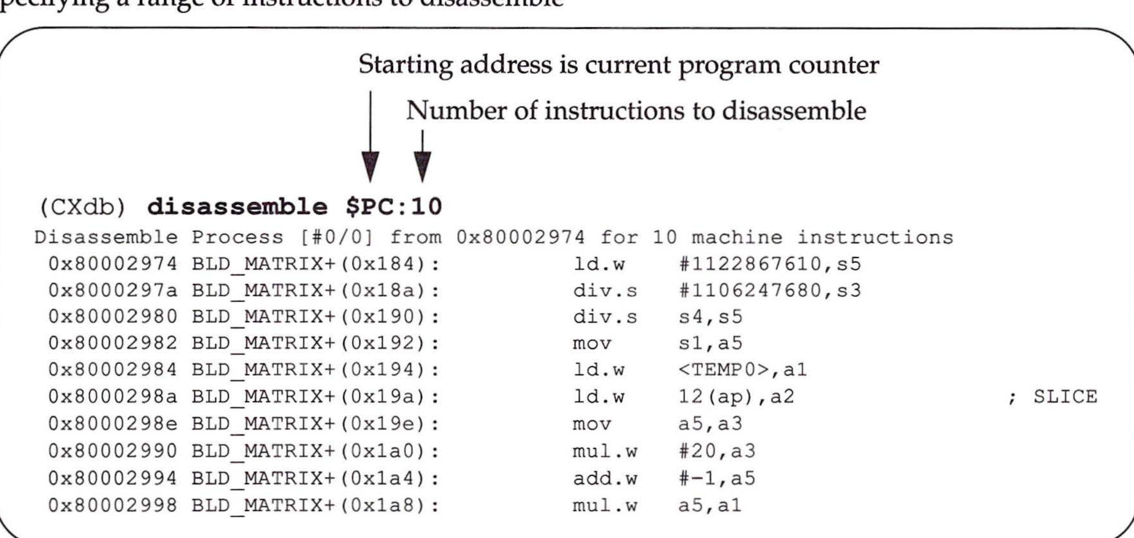
Figure 321
Displaying disassembled code

```
(CXdb) disassemble
Disassemble Process [#0/0] from 0x800027f0 to 0x80002b42
 0x800027f0 BLD_MATRIX:          sub.w   #16,a0
 0x800027f4 BLD_MATRIX+(0x4):            ld.w    @4(ap),s0              ; M
 0x800027f8 BLD_MATRIX+(0x8):            ld.w    @0(ap),s1              ; N
                     .
                     .
                     .
 0x80002b36 BLD_MATRIX+(0x354):          ld.w    -4(fp),s1              ; <TEMP2>
 0x80002b3a BLD_MATRIX+(0x35a):          le.w    s0,s1
 0x80002b3c BLD_MATRIX+(0x35c):          jmps.t  _bld_matrix_+(0x11a)
 0x80002b42 BLD_MATRIX+(0x362):          rtn
```

As shown in Figure 321, the `disassemble` command without any parameters displays all instructions of the current routine. (The vertical ellipsis in the figure indicates that some of the output has been omitted from the example.) The output displays the hexadecimal address of the instruction, the relative address based on the start of the routine, the assembly language code, and the associated program variable if there is one.

Usually you will want to display only certain instructions rather than all instructions of the current routine. To do this, you can specify a starting address and the number of instructions to disassemble, as shown in Figure 322.

Figure 322
Specifying a range of instructions to disassemble




```
(CXdb) disassemble $PC:10
Disassemble Process [#0/0] from 0x80002974 for 10 machine instructions
 0x80002974 BLD_MATRIX+(0x184):          ld.w     #1122867610,s5
 0x8000297a BLD_MATRIX+(0x18a):          div.s    #1106247680,s3
 0x80002980 BLD_MATRIX+(0x190):          div.s    s4,s5
 0x80002982 BLD_MATRIX+(0x192):          mov      s1,a5
 0x80002984 BLD_MATRIX+(0x194):          ld.w     <TEMP0>,a1
 0x8000298a BLD_MATRIX+(0x19a):          ld.w     12(ap),a2                ; SLICE
 0x8000298e BLD_MATRIX+(0x19e):          mov      a5,a3
 0x80002990 BLD_MATRIX+(0x1a0):          mul.w    #20,a3
 0x80002994 BLD_MATRIX+(0x1a4):          add.w    #-1,a5
 0x80002998 BLD_MATRIX+(0x1a8):          mul.w    a5,a1
```

The `disassemble` command in Figure 322 displays 10 instructions (specified by `:10`), starting at the address indicated by the current PC (program counter). The predefined debugger variable `$PC` stores the current value of the program counter.

To disassemble instructions from other locations in the program, you can specify either a symbolic or an absolute starting address. For example, the `disassemble` command in Figure 323 displays 5 instructions starting at 14 bytes past the beginning of the routine called `chapter6`.

Figure 323
Specifying a relative starting address to disassemble

```
(CXdb) disassemble chapter6+14:5
Disassemble Process [#0/0] from 0x80001fda for 5 machine instructions
 0x80001fda CHAPTER6+(0xe):     ld.w     12(fp),a6
 0x80001fde CHAPTER6+(0x12):             ldea     _chapter6_+(0x480),a6
 0x80001fe4 CHAPTER6+(0x18):             calls    _for$do_lio
 0x80001fea CHAPTER6+(0x1e):             ld.w     12(fp),a6
 0x80001fee CHAPTER6+(0x22):             ldea     _chapter6_+(0x494),a6
```

Stepping by instruction

Stepping a process by machine instruction is the third feature for debugging at the instruction level. When combined with highlighting of expression source units in the source window and with displaying of disassembled code, stepping by instruction provides the most detailed view of how a process executes. Figure 324 illustrates the combined use of these three features.

Figure 324
Stepping by instruction, combined with disassembled code and expression highlighting

```
(CXdb) disassemble $PC:5
Disassemble Process [#0/0] from 0x80002974 for 5 machine instructions
 0x80002974 BLD_MATRIX+(0x184):          ld.w     #1122867610,s5
 0x8000297a BLD_MATRIX+(0x18a):          div.s    #1106247680,s3
 0x80002980 BLD_MATRIX+(0x190):          div.s    s4,s5
 0x80002982 BLD_MATRIX+(0x192):          mov      s1,a5
 0x80002984 BLD_MATRIX+(0x194):          ld.w     <TEMP0>,a1
(CXdb) step instruction
Stepping process [#0/*] by 1 instruction
Process [#0/0] stopped stepping at [0x8000297a] BLD_MATRIX in chapter7F.f line 39
```

```
 38          DO K = 1, L
 39 (B)        MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
 40            PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
 41          ENDDO
```

```
(CXdb) step instruction
Stepping process [#0/*] by 1 instruction
Process [#0/0] stopped stepping at [0x80002980] BLD_MATRIX in chapter7F.f line 39
```

```
 38          DO K = 1, L
 39 (B)        MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
 40            PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
 41          ENDDO
```

The `disassemble` command in Figure 324 displays 5 instructions starting at the current PC, or address `80002974`. The `step instruction` command executes the instruction at address `80002974`. When execution stops, the PC contains the address `8000297a`, and the source window shows that the expression `3*K/7.5` is highlighted. Therefore, the expression `3*K/7.5` maps to the current instruction at address `8000297a`. The `disassemble` command shows that the instruction at `8000297a` is a division operation (`div.s`), which agrees with the operation specified in the expression `3*K/7.5`.

The second `step instruction` command in Figure 324 executes the instruction at address `8000297a`. Execution stops at address `80002980`, and the expression `29.7/(I+J+K)` is highlighted in the source window. This expression maps to the division instruction (`div.s`) at address `80002980`.

As with other stepping commands, you can specify a repeat count with the `step instruction` command.

In addition to the `step instruction` command, there is also a `next instruction` command. The `next instruction` command does not count instructions in called subroutines as part of the repeat count. In contrast, the `step instruction` command counts all instructions in both calling and called routines.

Figure 325 illustrates the `next instruction` command and the repeat count.

Figure 325
A repeat count with the `next instruction` command

```
(CXdb) disassemble $PC:5
Disassemble Process [#0/0] from 0x80002980 for 5 machine instructions
 0x80002980 BLD_MATRIX+(0x190):           div.s   s4,s5
 0x80002982 BLD_MATRIX+(0x192):           mov     s1,a5
 0x80002984 BLD_MATRIX+(0x194):           ld.w    <TEMP0>,a1
 0x8000298a BLD_MATRIX+(0x19a):           ld.w    12(ap),a2                   ; SLICE
 0x8000298e BLD_MATRIX+(0x19e):           mov     a5,a3
(CXdb)  next instruction 3
Nexting process [#0/*] by 3 instructions
Process [#0/0] stopped nexting at [0x8000298a] BLD_MATRIX in chapter7F.f line 39
(CXdb) disassemble $PC:5
Disassemble Process [#0/0] from 0x8000298a for 5 machine instructions
 0x8000298a BLD_MATRIX+(0x19a):           ld.w    12(ap),a2                   ; SLICE
 0x8000298e BLD_MATRIX+(0x19e):           mov     a5,a3
 0x80002990 BLD_MATRIX+(0x1a0):           mul.w   #20,a3
 0x80002994 BLD_MATRIX+(0x1a4):           add.w   #-1,a5
 0x80002998 BLD_MATRIX+(0x1a8):           mul.w   a5,a1
```

```
  38       DO K = 1, L
  39 (B)     MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
  40         PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
  41       ENDDO
```

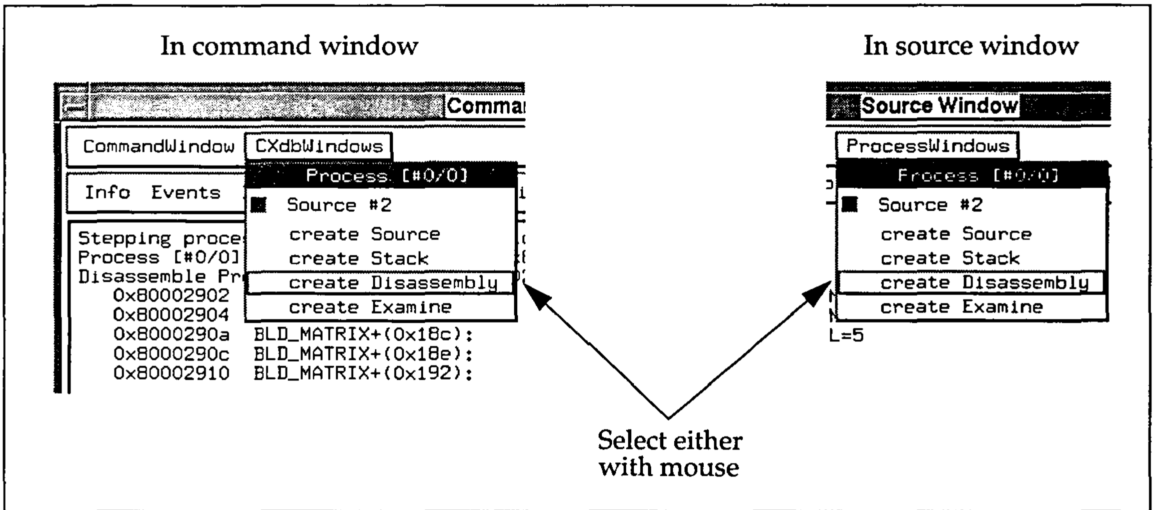
In Figure 325, the `next instruction` command steps the process by 3 instructions. Both before and after stepping, the `disassemble` commands display the 5 instructions starting at the current PC.

The disassembly window in CXwindows

The CXwindows environment has a separate disassembly window for viewing disassembled code. This section explains how to access and use that window. If you are not using CXwindows, you can skip this section and proceed to the section "Quitting the examples," at the end of this chapter.

You can open the disassembly window from either the CXdbWindows menu in the command window or from the ProcessWindows menu in the source window. Both of these methods are shown in Figure 326.

Figure 326
Opening the disassembly window



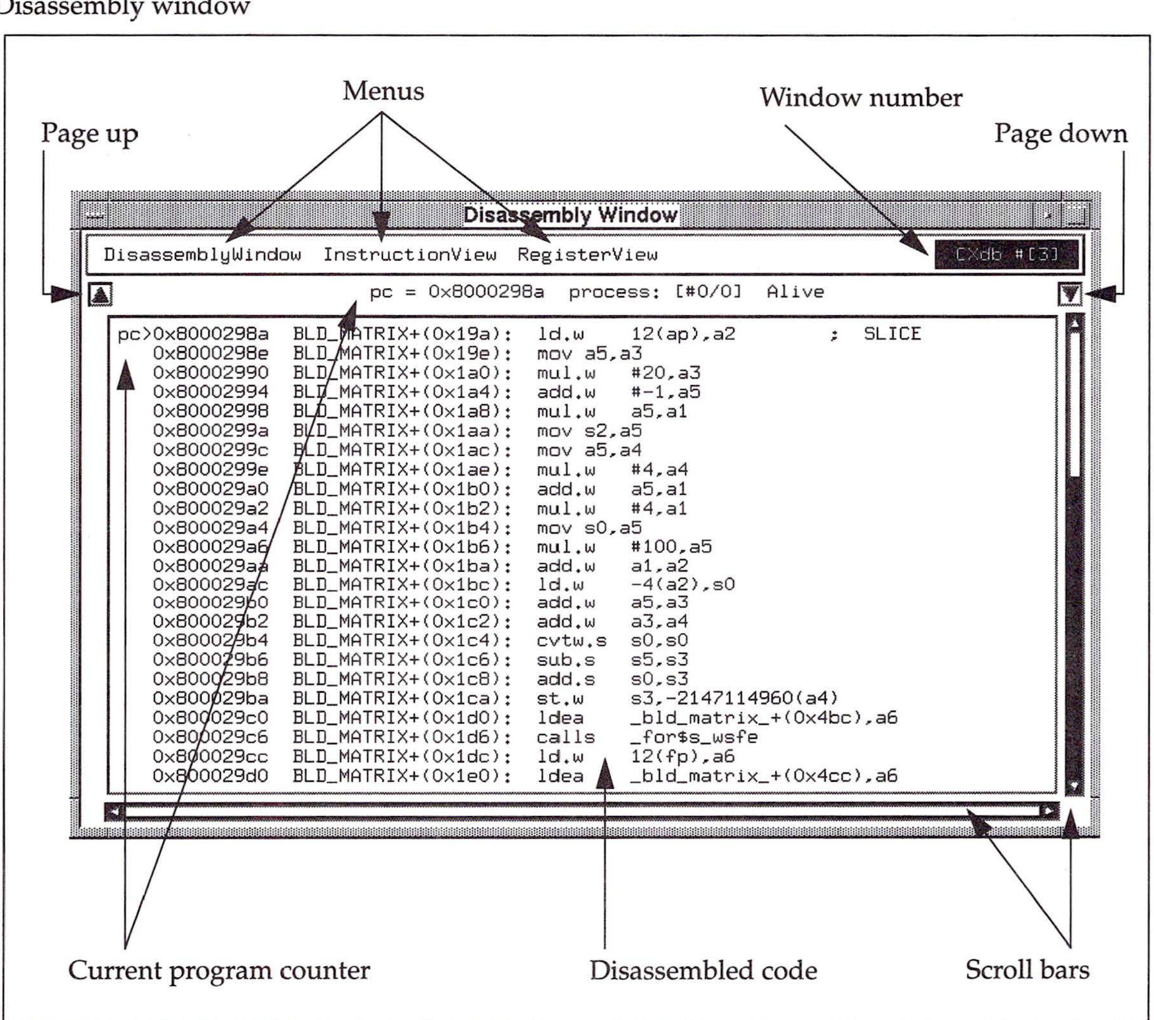
You can open any number of disassembly windows during a debugging session, and each can display a different section of code. When a disassembly window first opens, it displays the machine instructions starting at the current program counter (PC).

Figure 327 shows the disassembly window. Its parts include:

- **Menus**—These pull-down menus allow you to do the following with the mouse:
  - DisassemblyWindow closes the disassembly window or enables and disables the auto update option.
  - InstructionView allows you to select a different section of disassembled code to view.
  - RegisterView opens special windows for viewing the processor status word, scalar, vector, and communication registers.

- **Window number**—This is a unique number that identifies each window in CXdb. Certain CXdb commands allow you to specify a window number as a parameter.
- **pc**—A marker to indicate the instruction that the current program counter is pointing to. This is the next instruction to be executed if the process continues normally.
- **Scroll bars**—These bars and arrows enable you to scroll the window vertically and horizontally with the mouse.
- **Paging buttons**—Buttons that allow you to scroll the window up or down by one page (screen) per click of the button.

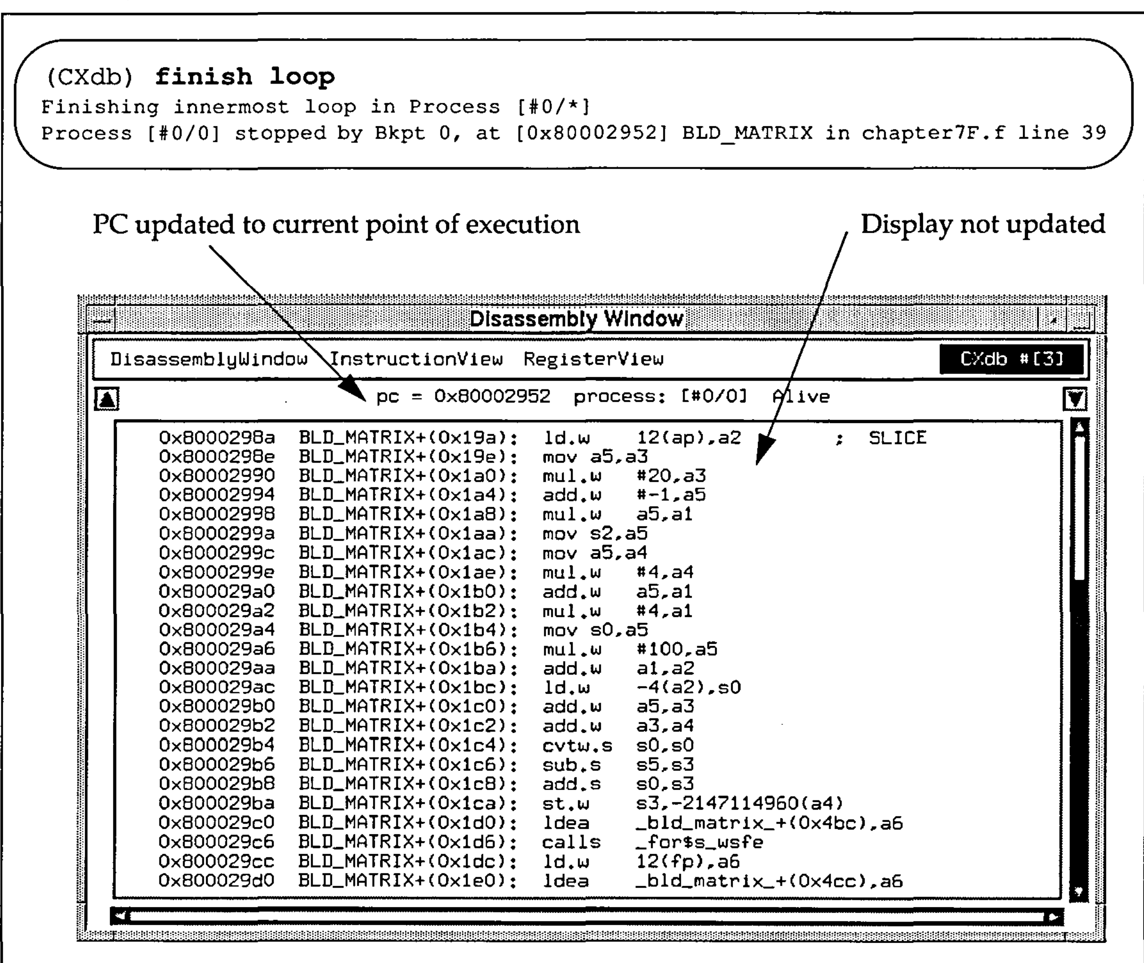
Figure 327
Disassembly window



Selecting auto update

When the disassembly window first opens, auto update is not enabled. This means that the display does not update if execution continues beyond the instructions currently shown in the window. Figure 328 illustrates this point.

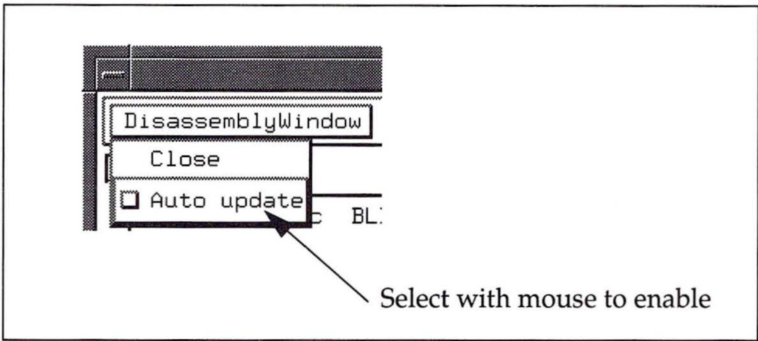
Figure 328
Disassembly window with auto update disabled



The `finish loop` command in Figure 328 continues execution beyond the instructions currently displayed in the disassembly window. Because auto update is disabled, the display does not update to show the instructions at the new point of execution. However, the value indicated for the PC is updated.

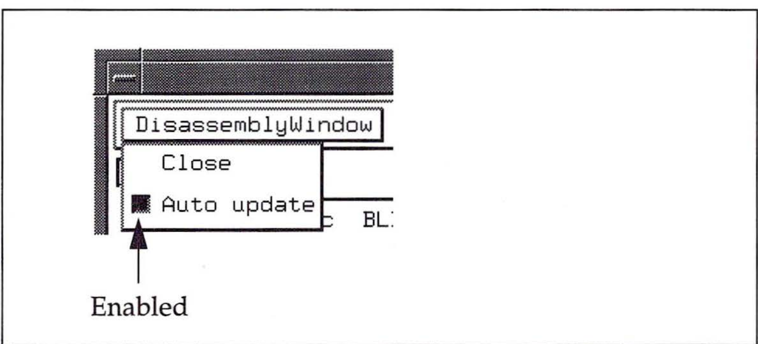
To enable the auto update feature, select it from the DisassemblyWindow menu, as shown in Figure 329.

Figure 329
Enabling the auto update feature



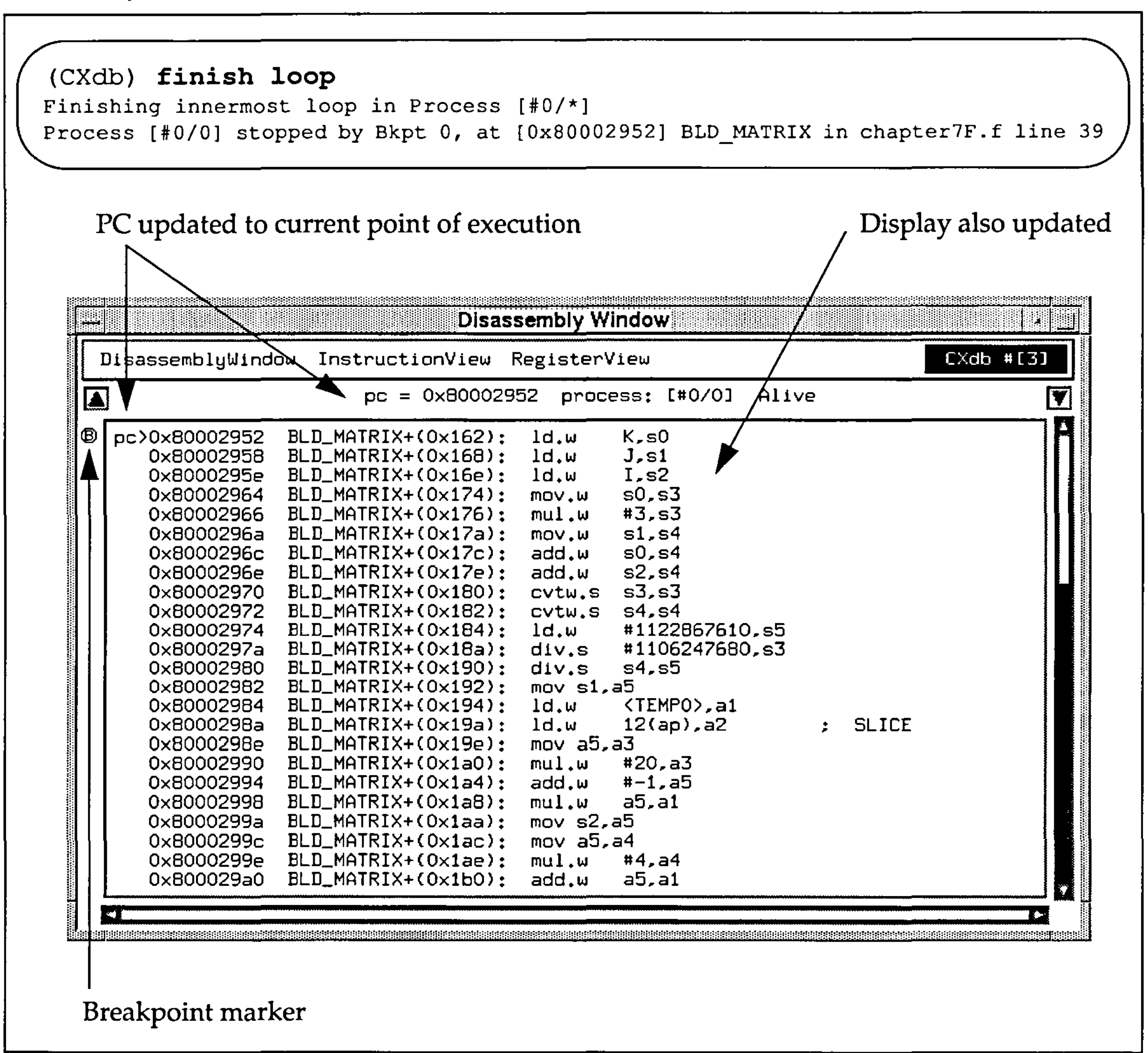
After you enable the auto update option, the DisassemblyWindow menu looks like Figure 330.

Figure 330
Auto update enabled



When auto update is enabled, the disassembly window automatically updates to display the instructions starting at the current point of execution, as shown in Figure 331. The figure also shows that the disassembly window displays eventpoints.

Figure 331
Disassembly window with auto update enabled



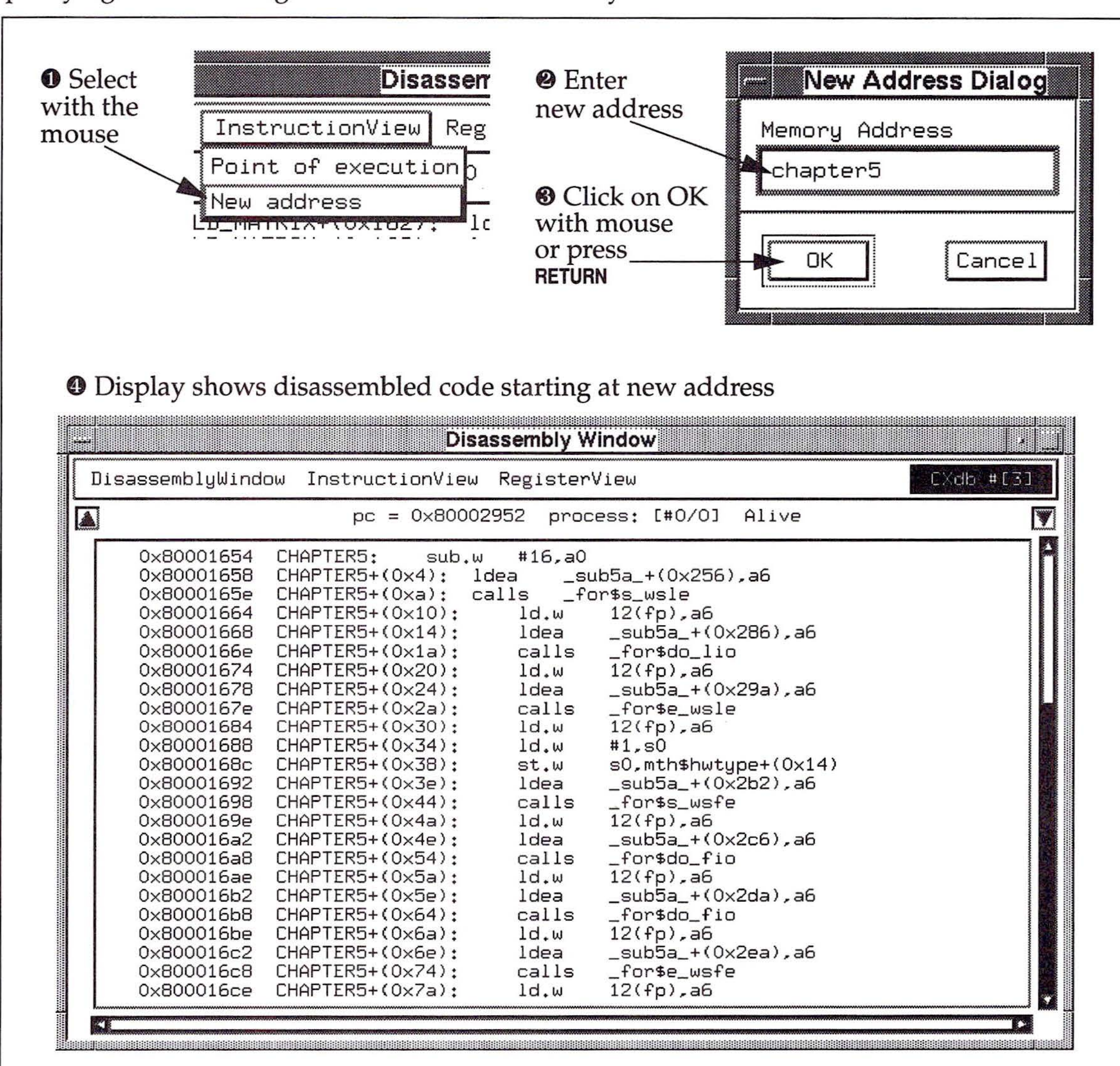
Note

By setting the appropriate resources in your `.Xdefaults` file, you can enable the auto update feature by default whenever the disassembly window is opened. For a description of how to do this, refer to the concept "Xdefaults" in the *CONVEX CXdb Reference*.

Specifying another address to disassemble

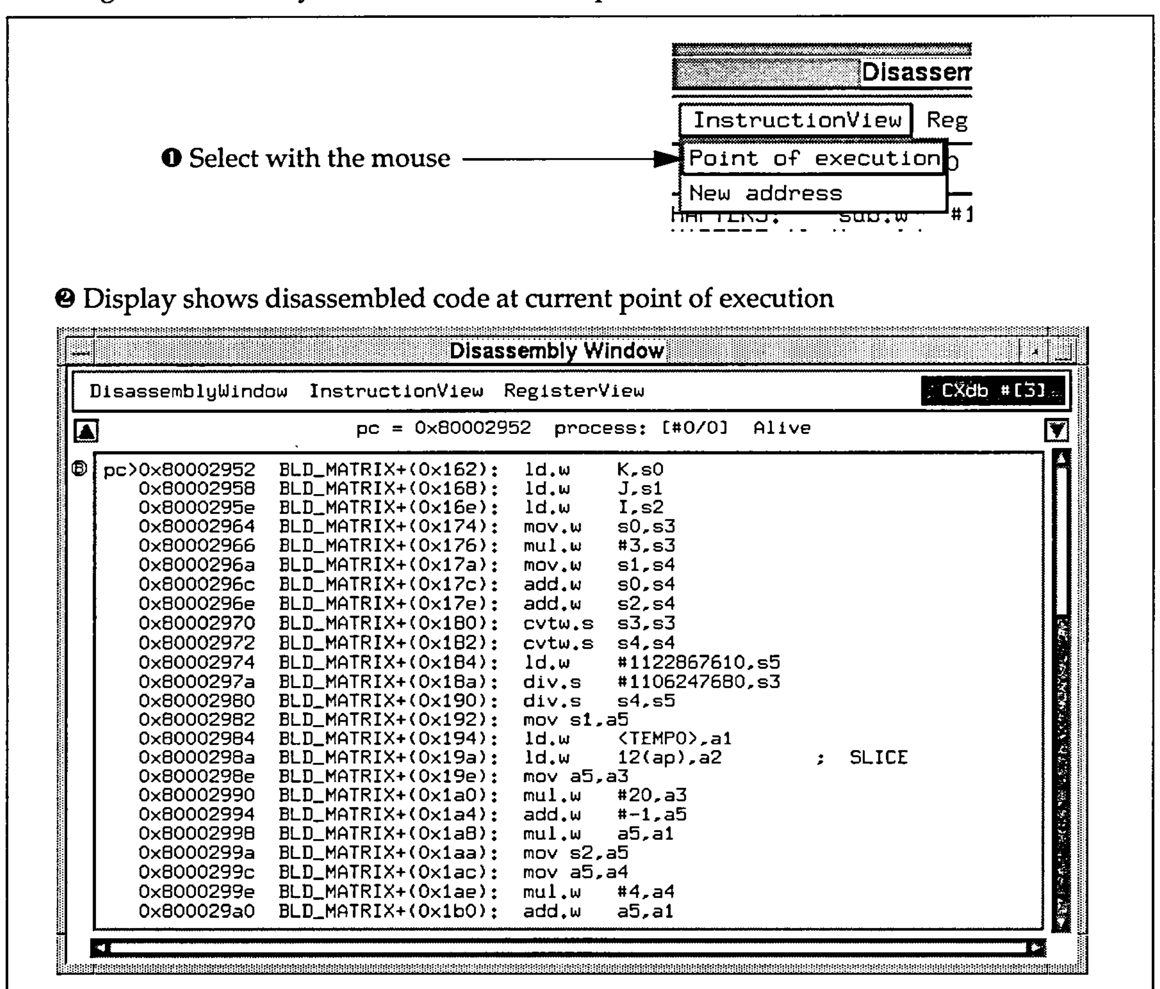
To view a section of disassembled code that is not near the current point of execution, you can specify a new starting address for the disassembly by selecting the New address option from the InstructionView menu. The address can be either symbolic or absolute. Figure 332 illustrates this feature.

Figure 332
Specifying a new starting address for the disassembly



Select the Point of execution option from the InstructionView menu to return the disassembly window to the current point of execution, as shown in Figure 333.

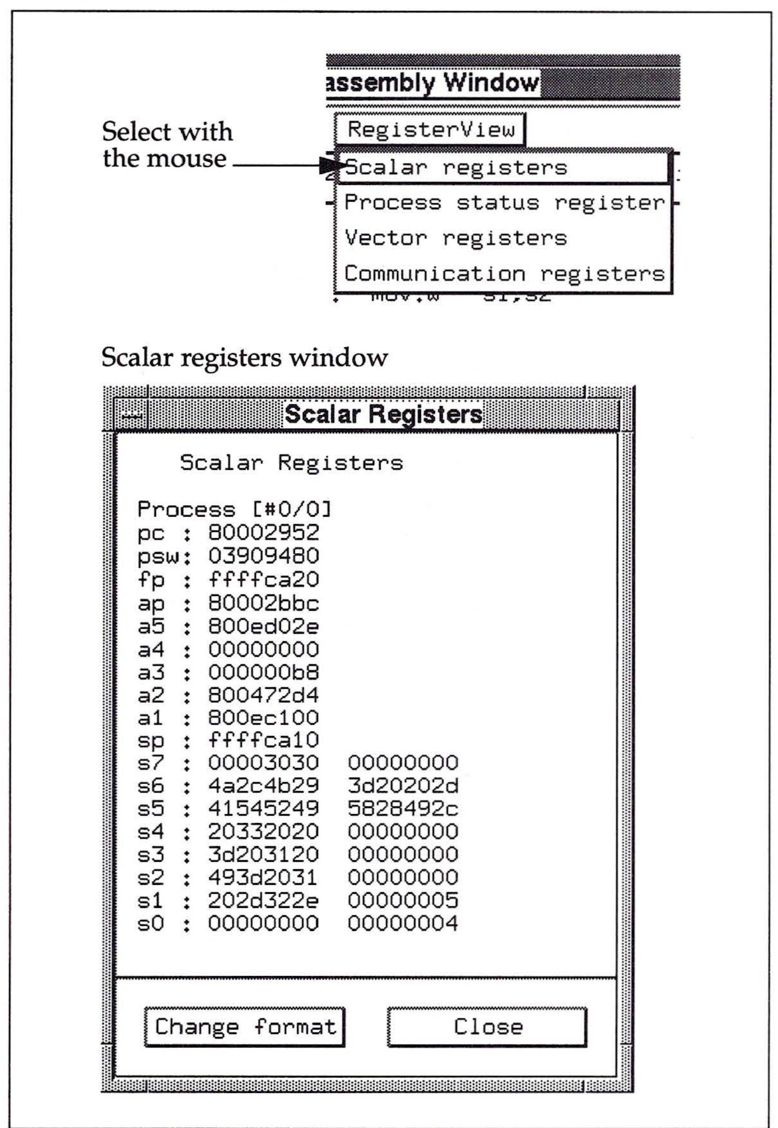
Figure 333
Returning the disassembly window to the current point of execution



Displaying registers

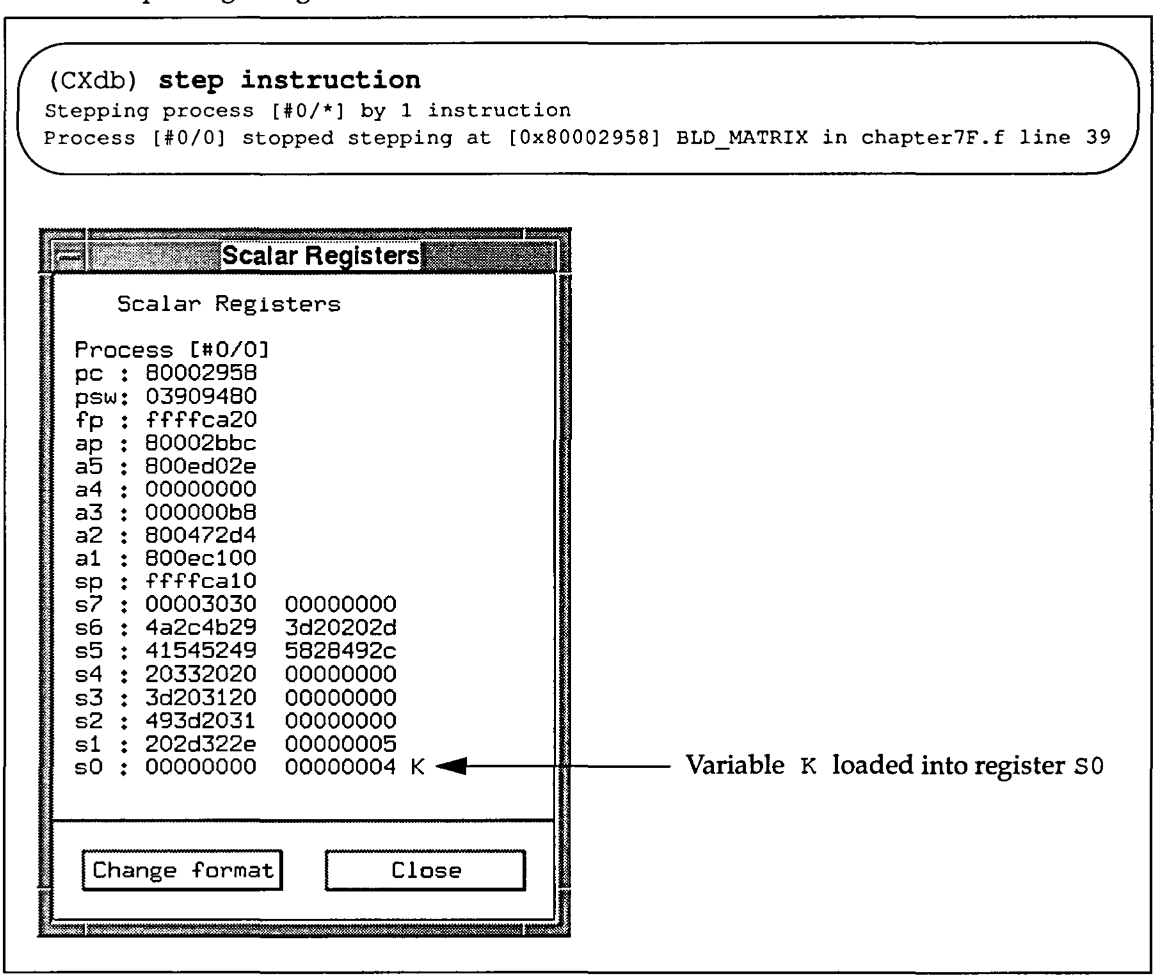
From the RegisterView menu of the disassembly window, you can open special windows that display the scalar registers, vector registers, communication registers, and processor status word register. Figure 334 shows how to display these register windows.

Figure 334
Displaying the scalar registers



The register windows update automatically, as shown in Figure 335.

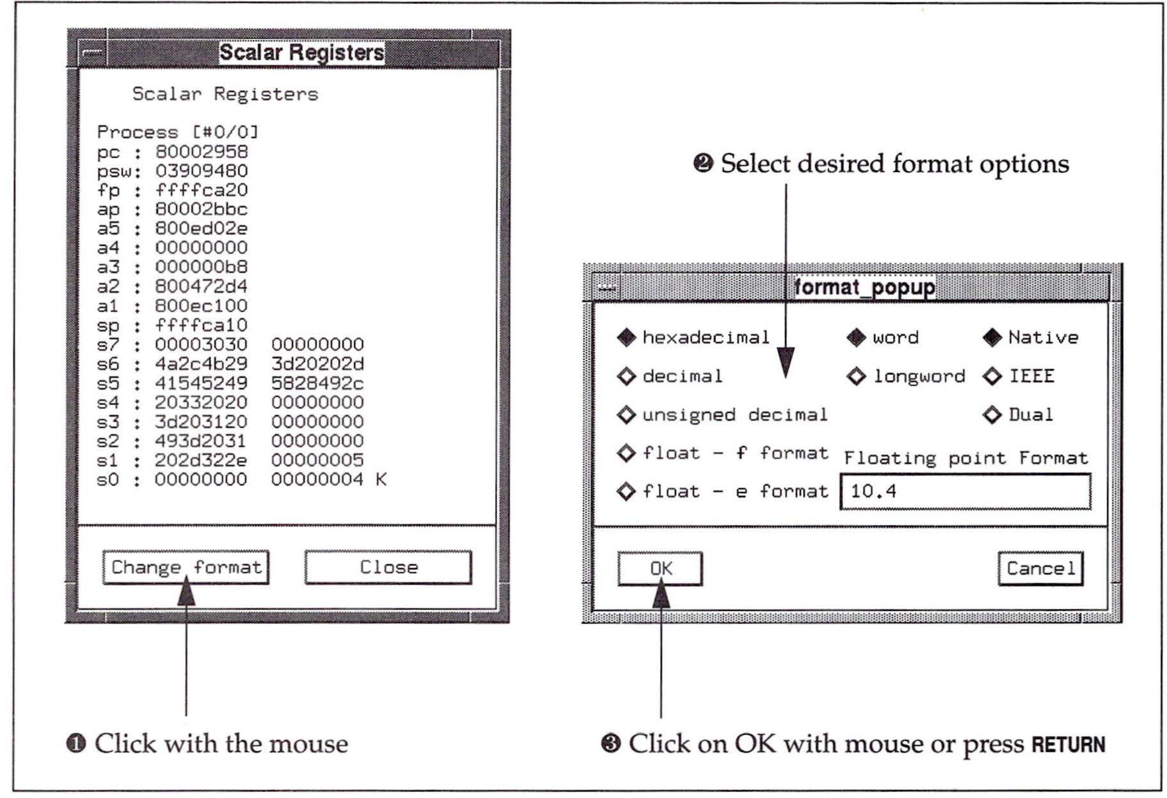
Figure 335
Automatic updating of register windows



The `step instruction` command in Figure 335 executes the next machine instruction. This instruction, shown in Figure 333, loads the variable `K` into scalar register `S0`. By placing the name of the variable beside the register that contains it, the scalar register window indicates that the load has occurred.

By default, the register windows display values in hexadecimal format. To change the format, click on the Change format button at the bottom of the register window, as shown in Figure 336.

Figure 336
Changing the display format of the register window



Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 337.

Figure 337
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Debugging optimized code 15

This chapter describes basic principles involved in debugging optimized code with CXdb. It is assumed that you are already familiar with concepts and terminology relating to optimization. For information about optimization, refer to either the *CONVEX FORTRAN Optimization Guide* or the *CONVEX C Optimization Guide*.

The goal of this chapter is to provide some helpful suggestions for debugging optimized code. It is not a detailed guide to optimization, nor does it explain how to debug every type of optimization. Rather, this chapter tries to explain, in general terms, what CXdb can and cannot do to debug optimized code.

Note

Optimizations occur even if your program is compiled with the `-no` option (the default optimization level). This chapter contains important information about the effects of level `-no` optimizations. You should read this chapter even if you do not optimize your program beyond level `-no`.

Three major debugging functions are most significantly affected by optimization:

- Setting eventpoints
- Stepping program execution
- Displaying the values of program variables

The examples in this chapter illustrate how different levels of optimization affect these three debugging functions.

Source units and optimized code

Source units provide a means of representing source code as syntactical units. They subdivide the source code into a set of routines, blocks, loops, statements, and expressions. These units provide a convenient way to reference and manipulate the source code. Source units also reflect the syntax and structure of the source code.

Optimization, on the other hand, produces object code that is directed toward making more efficient use of the capabilities of the machine. The optimized object code strongly reflects characteristics of the machine architecture. Because of this, optimized code has a much different structure and order of execution than unoptimized code.

Even though optimized code differs markedly from unoptimized code, CXdb is able to map the source units to specific ranges of optimized object code. The mapping is made possible by the data files generated by the CONVEX FORTRAN and CONVEX C compilers. When you compile your program with the `-cxdb` option, the compiler creates the data files that CXdb needs to debug your program symbolically, regardless of optimization level. These data files reside in the `.CXdb` directory, and they contain most of the debugging information about your program.

Why source units are important

Conventional debuggers operate on line numbers of source code. When code has been optimized, source line numbers usually do not map well to object code. Thus, conventional debuggers cannot track the execution of optimized code directly.

CXdb, on the other hand, tracks source units. (Refer to Chapter 5.) The granularity of the source units ranges from something as large as a routine to something as small as a single identifier (expression). These different types of source units make it possible for CXdb to map the source code directly to the optimized object code. CXdb displays this mapping graphically in the source window by highlighting the source unit that is currently active.

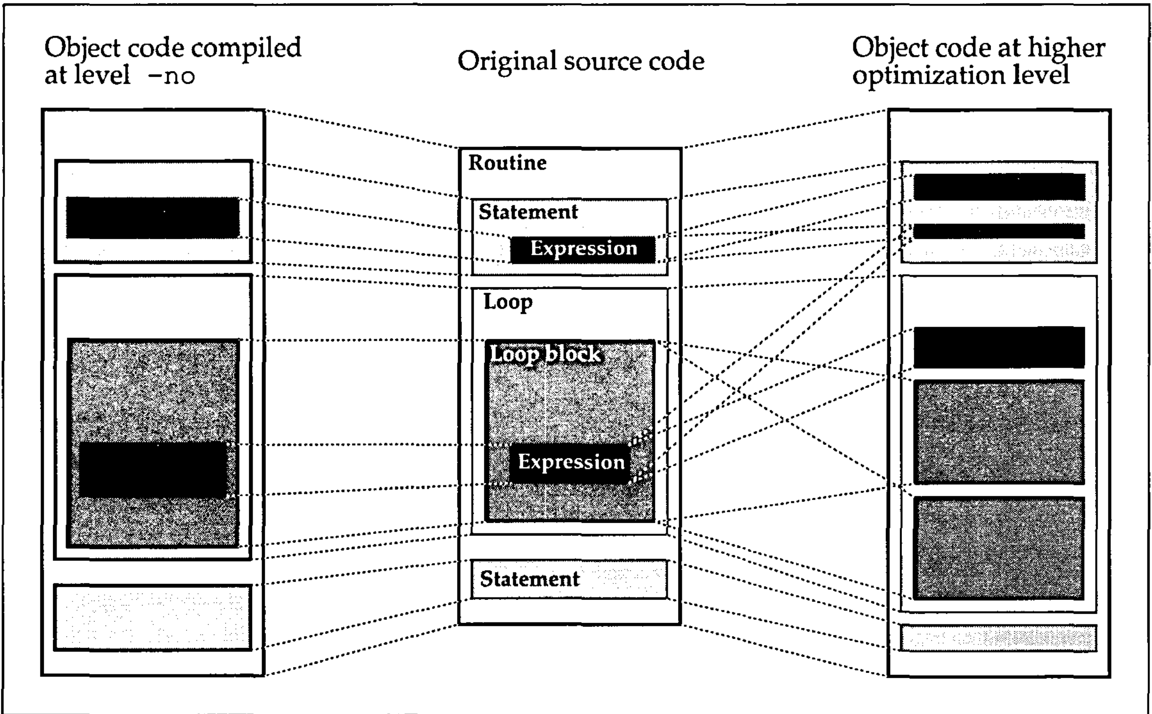
The result is that CXdb enables you to track the execution of the optimized code to whatever degree of accuracy (granularity) you desire. As you step through the execution of your program, you can use the `disassemble` command to display the assembly language instructions from the optimized object code. To analyze the optimizations in more detail, you can then correlate the disassembled object code with the source units that are highlighted in the source window.

Source unit ranges

Each source unit maps to zero or more ranges of object code. A *range* is a sequence of one or more machine instructions in the object code.

The CONVEX FORTRAN and CONVEX C compilers perform various code transformations at each optimization level, even level `-no`. At level `-no`, the transformations are less pronounced than at higher optimization levels. Therefore, the mapping of source units to ranges of optimized code is essentially linear at level `-no`. However, at higher optimization levels, the mapping can quickly become more dispersed. Figure 338 illustrates this concept.

Figure 338
Mapping of source units to ranges of optimized code



As the left half of Figure 338 indicates, at optimization level `-no` each source statement maps to only one range of instructions in the object code. This is because, at level `-no`, the compiler puts each source statement into its own basic block. The optimization transformations at level `-no` occur within a basic block (that is, within a statement) rather than between blocks. Therefore, the object code retains the same logical order as the source code.

Note

A basic block in the object code is generated by the compiler. It is not the same as a source unit block recognized by CXdb in the source code.

At optimization level `-O0` and above, a basic block usually consists of more than one statement. At level `-O1` and above, optimization transformations can occur between the basic blocks as well as within them. For example, transformations such as elimination, hoisting, and loop distribution change the logical order of instructions in the optimized object code. As the optimization level increases, the transformations to the code become more dramatic.

As a result of the code transformations, even at optimization level `-no`, each source unit can map to more than one range of object code. For the same reasons, one range of object code can map to more than one source unit. The higher the optimization level, the more dispersed the mapping will be. The right half of Figure 338 illustrates this type of dispersed mapping.

Optimization of variables

Often during the debugging session, you might want to print the current value of a particular variable in your program. Optimization can affect program variables in a number of ways, such as by folding, propagation, or elimination. Because of these optimizations, it might be difficult or impossible to determine the exact value of a program variable at any given time.

For example, if all references to a variable are eliminated by optimization, then the compiler does not generate any code to store the value of the variable. If the value of the variable is not stored, then the debugger cannot retrieve it to display it for you.

In most cases, optimization does not completely eliminate all references to a variable. However, the storage location of the variable is changed to optimize performance. In general, there are three ways a variable can be stored:

- In memory (either globally or on the process stack)
- In a register
- In the derivation of a synthesized variable (at optimization level `-O1` and higher)

If the variable is stored in memory or in a register, it is relatively easy for the debugger to retrieve and display the value. However, at optimization level `-O1` and higher, some program variables are not stored directly. Instead, the compiler generates a set of synthesized variables whose use is more efficient than the original program variable. Conventional debuggers cannot track synthesized variables, but CXdb can.

Synthesized variables

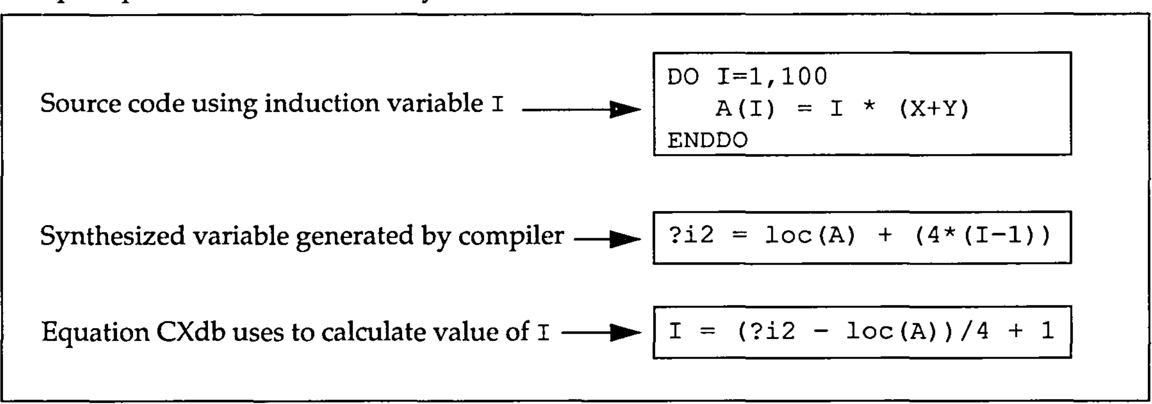
Synthesized variables are generated by the compiler at optimization levels `-O1` and higher. The compiler uses synthesized variables for one of two reasons:

- To replace a program variable with a more efficient construct
- To provide runtime support for the program

The most important use of synthesized variables is to replace loop induction variables in your program. Frequently, induction variables are used as array subscripts. If the induction variable were used as originally defined in the source code, the compiler would have to generate machine code to recalculate the address of the referenced array element each time the induction variable is incremented. For greater efficiency, the compiler creates a synthesized variable that is a pointer into the array. Instead of incrementing the original induction variable, the optimized code increments the synthesized pointer by the length of an array element.

Because the optimized code no longer needs the original induction variable, it does not compute or store the value. However, CXdb can calculate the value of the program variable from the equations that the compiler uses to generate the synthesized variables. Figure 339 shows an example of this.

Figure 339
Example equation used to derive a synthesized variable



In Figure 339, the source code uses the induction variable `I` as an index into the array `A`. Rather than incrementing `I` and computing the address of `A(I)` on each iteration of the loop, the compiler generates the synthesized variable `?i2` to serve as a pointer to the desired array element. On each iteration of the loop, `?i2` is incremented by 4, which is the number of bytes in one array element.

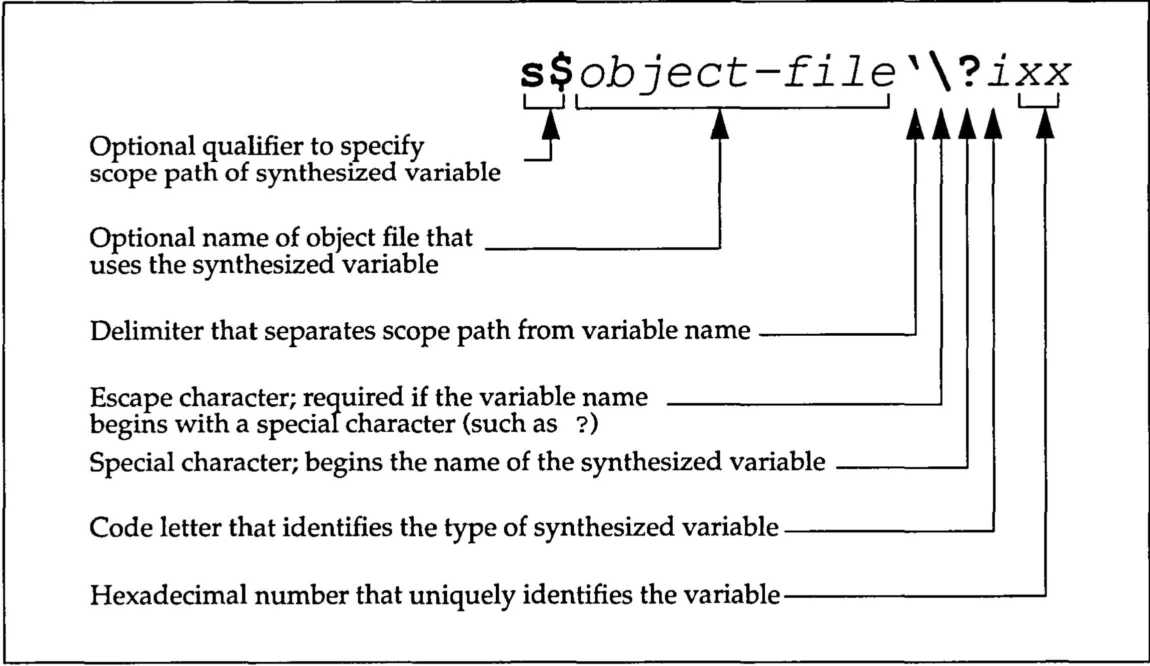
When you ask CXdb to display the current value of `I`, it must calculate the value from the equation that the compiler used to generate `?i2`. CXdb rearranges the equation for `?i2` and solves it for `I` based on the current value of `?i2`.

In many cases, CXdb must solve several equations simultaneously to calculate the value of a single program variable. This can sometimes lead to equations that have no solution or that have multiple, inconsistent solutions. If CXdb cannot calculate any value for the program variable you requested, CXdb issues an informational message stating that the variable is not available. If there are multiple values for the variable, CXdb lists all of the values and indicates which value it has selected from the list.

Figure 339 also illustrates the naming convention that CXdb uses for synthesized variables. The identifier, or name, of the synthesized variable begins with a special character (such as `?` or `#`) that generally is not part of an identifier name in the source language of the program. The name includes a code letter to indicate the variable type, and it ends with a hexadecimal number that uniquely identifies each variable. These synthesized variable names are the same as those found in the assembler listing generated by the compiler.

If the synthesized variable name begins with a special character (such as `?`) that has meaning in a CXdb command, then you must precede the name with a backslash (\). The full scope path for a synthesized variable starts with `S$` and includes the name of the object file that contains the synthesized variable. Figure 340 illustrates the format of synthesized variable names.

Figure 340
Full scope path of a synthesized variable



One program variable can be associated with the generation of many synthesized variables. The `info expression` command displays all of the synthesized variables derived from a single program variable. The `info expression` command lists the name of the synthesized variable, the equation used to generate it, and the reason for its use. The reason for its use is listed as a four-letter acronym. The possible reasons and their corresponding acronyms are listed and described in the following sections.

Call temporaries (CTMP) and temporary pointer (TPTR)

When arguments are passed in subroutine calls, synthesized variables are used as temporary storage for the arguments that are expressions. Call temporaries are used to pass arguments by value, and temporary pointers are used to pass arguments by reference.

Loop strength reduction (INDV)

Loop strength reduction is simplification of the iterative operator for an induction variable.

Subscript expansion (SEXP)

Subscript expansion occurs when the subscripts of a multidimensional array have constant, predetermined bounds. The subscripts are folded into one, and the array is accessed as a 1-dimensional array.

Reduced subexpression (REXP)

A reduced subexpression is one that is hoisted out of a loop. Its value is calculated and stored in a synthesized variable.

Outer strip (OSTR) and inner strip (ISTR)

Strip mining transforms a single loop into two new loops: one inner and one outer. The inner loop usually represents the portion of code that is either vectorized or parallelized, and the outer loop controls the iteration count. The inner strip (ISTR) and outer strip (OSTR) are the synthesized variables for these two new loops.

Simple sink (SINK)

In some cases, a loop that is marked for strip mining is not vectorized because the iteration count is too small. When this happens, a new sink variable is generated to replace the original induction variable.

Simple trip (TRIP)

A trip count is often used to replace a more complex loop iteration quantity.

MLXS (MLXS)

During vectorization, a multidimensional array can sometimes be transposed into a 1-dimensional array. This differs from subscript expansion, which is performed after vectorization.

Unrolled induction value (URIV), unrolled trip (URTP), and unrolled expression (UREX)

For an unrolled or partially unrolled loop, synthesized variables are needed to replicate values from the loop header.

Put back use (PBKU) and put back expression (PBKE)

Loop-invariant expressions and constants are placed back into the middle of a loop as a single reference to a synthesized variable.

Forward link (FLNK)

During vectorization, a constant value may be propagated to the distributends of a loop. A synthesized variable is used to represent this constant value.

Cheat min (CMIN)

Pattern matching is used for some types of optimization. In these cases, an array can be traversed to find the minimum element. A synthesized variable is used as the index to the array.

Alternate entry (ALTE)

For a loop that executes conditionally, this synthesized variable represents an alternate entry point to the loop.

Vector mask (VMSK), bit bucket (BITB), and zero mask (ZMSK)

These synthesized variables are used for storing and assigning the current vector mask.

Strip mine length (STML)

This synthesized variable stores the strip mine length, which is usually 128.

Memory addresses (TBSS, TEXT, TDATA, DATA, BSS)

These variables store the addresses of the designated memory sections. TBSS is the area of memory that is thread-specific, uninitialized, static storage. TEXT is the area of memory where machine instructions are stored. TDATA is process-wide uninitialized storage. DATA is process-wide initialized storage. BSS is process-wide, uninitialized, static storage.

Communication register (CREG)

This synthesized variable represents a communication register.

Scalar boot (BOOT) and vector boot (VBOT)

During register allocation, a specific value can be removed from a register so that the register is free to be reallocated. The synthesized variable that stores the removed variable is called the boot.

Scalar spill (SPLL) and vector spill (VSPL)

Just before a subroutine call, all active registers are stored (spilled) into memory.

Distributed expression (DEXP)

A variable from one loop can be distributed over several optimized loops.

Unsolvable induction variable (DEAD)

In some cases, it is technically impossible to calculate the correct value of a loop induction variable. In such cases, CXdb does not report a synthesized variable.

Values that cannot be determined

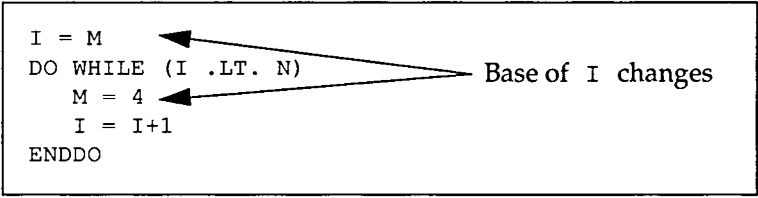
Although CXdb far exceeds the capabilities of many conventional debuggers in its ability to track synthesized variables, there are still a number of cases where CXdb cannot calculate the value of a variable. Those cases are enumerated below.

If you try to print a variable in one of these cases, CXdb issues a message indicating that the value is not available.

Changing the base

If the base value of an induction variable is changed inside the loop, then the value of the variable cannot be determined during a particular iteration of the loop. Figure 341 shows an example of an induction variable `I` that is based on the value of variable `M`. Because the value of `M` is changed inside the loop, the base value of `I` also changes.

Figure 341
An induction variable with a changing base



Parallel equations

For some induction variables, the compiler uses parallel equations to generate the synthesized variables. In these cases, the induction value cannot be determined. Figure 342 shows a case where the induction variables `I` and `J` generate parallel equations.

Figure 342
Induction variables with parallel equations

```
I = M
J = 1
DO WHILE (A(I+J) .LT. N)
   J = J+1      <---- Parallel equations
   I = I+1      <----
ENDDO
```

Inactive variables

If optimization has removed references to a variable, that variable can become inactive during and after the body of the loop. Therefore, the value of the variable is not available within those areas of the program. Figure 343 shows an example from a C program. In this example, the pointer `p` is referenced prior to the start of the loop, but not inside the loop body.

Figure 343
A pointer that becomes inactive inside a loop

```
i = m;
do while (p[i++] < n)
{
    sum += b->data[i]; ◄— Variable p not referenced
}
```

Completely unrolled loops

If a loop is unrolled completely, no variable is synthesized to keep track of the iterations. Therefore, the value of the induction variable is not available.

Variables within intrinsics

The value of an intrinsic function such as `log` or `sine` can be calculated, but the value of a parameter passed to an intrinsic cannot be calculated. For example, in the equation `Y=LOG(X)`, the value of `Y` can be determined, but the value of `X` cannot.

Variables used with special operators

The following operators make it impossible to determine the value of a variable:

- `BITSHIFT`
- `MIN`
- `MAX`
- All relational operators
- All logical operators

For example, in the equation `Z=MAX(X,Y)`, the value of `Z` can be determined if the values of `X` and `Y` are known. However, the value of X cannot be determined even if `Y` and `Z` are known.

Variables derived from user-defined functions

To avoid any possible side effects to the user's address space, CXdb does not invoke a user-defined function in order to determine the value of a variable.

Variables that are not updated

In a type of optimization known as redundant store elimination, the value of a variable is loaded into a register, but the value is not stored back into the memory location of the variable unless the variable is used in a different basic block of the code. If the source code makes no other references to the variable, the memory location of the variable is never updated. Printing the variable yields the old (stale) value in memory and not the updated value in the register.

Figure 344 shows an example from a C program. In this example, the value of the expression `x+1` is loaded into a register, but the value of `x` in memory is not updated because of redundant store elimination. Because the source code does not reference `x` outside the loop, its value is not stored back into memory as the register value is incremented. Therefore, the current value of `x` is the original value in memory, not the current value in the register.

Figure 344
A variable whose memory location is not updated

```
do while (y < n)
{
   x = x+1;
   y = foo(x);
}
```

Value of `x` not stored in memory

Incomplete symbolic equation

In a few cases, CXdb is not able to infer the correct symbolic equation to derive the value of a variable. CXdb does, however, report as much information as it can correctly infer.

Hints for debugging optimized code

Although specific debugging steps will vary quite a bit from one situation to another, those steps can be generalized into a set of guidelines that provide a sound overall approach to debugging.

The list below presents such a set of guidelines for debugging optimized code. The items in the list are numbered for purposes of organization only. Each of the items is equally important, and they should all be used together.

The section "Debugging at various optimization levels" shows examples that apply the guidelines listed here.

1. **Compile your program at optimization level `-no` and debug it at that level first.** Debugging optimized code does not supplant the effectiveness of debugging new code without optimizations. After you are certain that the program is running properly at level `-no`, recompile it at level `-O0` and debug it at that level. Then precede to level `-O1`, and so forth, until you reach the desired level of optimization. This approach makes it easier to isolate any problems introduced into the program by a particular type of optimization.
2. **Use the `set step expression` command to set the stepping granularity to expression.** This gives the most detailed highlighting in the source window. Because the highlighting illustrates the mapping between machine instructions and source units, it provides a graphical way to track the execution of your program.
3. **Use the `info line` command to display the source units on a given line of your program.** This shows the address ranges of the machine instructions that map to each source unit. It can help with setting eventpoints, stepping the program, and tracking the execution.
4. **Use the `info expression` command to display your program variables.** This displays the liveness ranges and storage locations of the variables. At optimization levels `-O1` and above, it also displays how synthesized variables are derived from user-defined variables.
5. **Use the `disassemble` command to display the disassembled code surrounding the current point of execution.** (In CXwindows, use the disassembly window with auto update enabled.) This shows which machine instructions map to the current source unit. It also shows which instruction will execute next, thus revealing the execution order of the optimized code.

6. **Use the `step instruction` command to step through a part of the program that is of critical interest to you.** Stepping by larger increments could cause you to overshoot or miss the critical point.

Setting eventpoints in optimized code

For eventpoints that are based on location (for example, `trace line` or `break source`), CXdb sets the eventpoint at the machine instruction that maps to the location you specify. Because the optimized machine code has a different ordering than the source code, the eventpoint marker shown in the source window might not always appear where you expect to see it.

In some cases, multiple eventpoints will appear at different addresses, even though you specified only one eventpoint at one location. This is because one source unit can map to several different ranges of machine instructions, and one machine instruction can map to several different source units.

Stepping through optimized code

Program execution follows the order dictated by the machine instructions. Because the ordering of the machine instructions is usually quite different than the ordering of your source statements, stepping by source units does not follow the sequence you would expect from looking at only the source code.

At optimization levels `-O0` and above, it generally is not possible to predict where execution will stop if you are stepping a process by source units. The higher the optimization level, the more dispersed is the mapping between source units and machine instructions. Therefore, the more difficult it is to calculate exactly how many source unit steps there are between any two points in the source code. For example, it usually is not possible to step the optimized code by 5 statements and have execution stop exactly 5 lines of source code from where it started.

Even at level `-no`, optimizations such as instruction scheduling can cause a program to execute in a different sequence than what you would expect from the source code. A methodical approach to stepping (such as the one outlined below) can enhance the predictability of the results.

Stepping carries execution forward, but not backward. Therefore, it is important to avoid stepping past the location in the program that is of critical interest to you. The following guidelines for stepping can help achieve this objective:

1. Step by large source units such as blocks, loops, or routines, until the current point of execution is relatively near a critical location in the program.
2. When in doubt about what stepping granularity to use or how many steps to take at once, be conservative. Take fewer steps at one time, and use a smaller stepping granularity.
3. For loop iterations, use the `step block` or `next block` commands rather than stepping by loop.
4. Near a critical point of the program, step by instruction until you reach the precise location that is of interest to you.
5. Frequently during procedures 1 through 4, observe the disassembled code and the source window to determine where execution has stopped and what instruction will execute next.

Debugging at various optimization levels

The following sections present examples that illustrate significant points about debugging at various optimization levels. The examples use the same FORTRAN source code compiled at each of the optimization levels. This helps to emphasize the changes that occur in the object code when a program is compiled at different optimization levels.

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 345.

Figure 345
Invoking CXdb

```
%cd /usr/lib/cxdb/examples
%cxdb a.out
```

The `cd` command in Figure 345 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Once you have invoked CXdb, move the cursor to the command window and enter the commands shown in Figure 346.

Figure 346
Starting the example program

```
(CXdb) set printopts maxarray 50
(CXdb) set step expression
```

The `set printopts` command in Figure 346 allows a maximum of 50 array elements to be displayed at one time, instead of the default of 20 elements.

The `set step` command in Figure 346 sets the stepping granularity to expression. The highlighting in the source window is affected by the granularity. With the granularity set to expression, the source window will highlight the smallest active source unit at the current point of execution. This provides the most detailed graphical view of the source units.

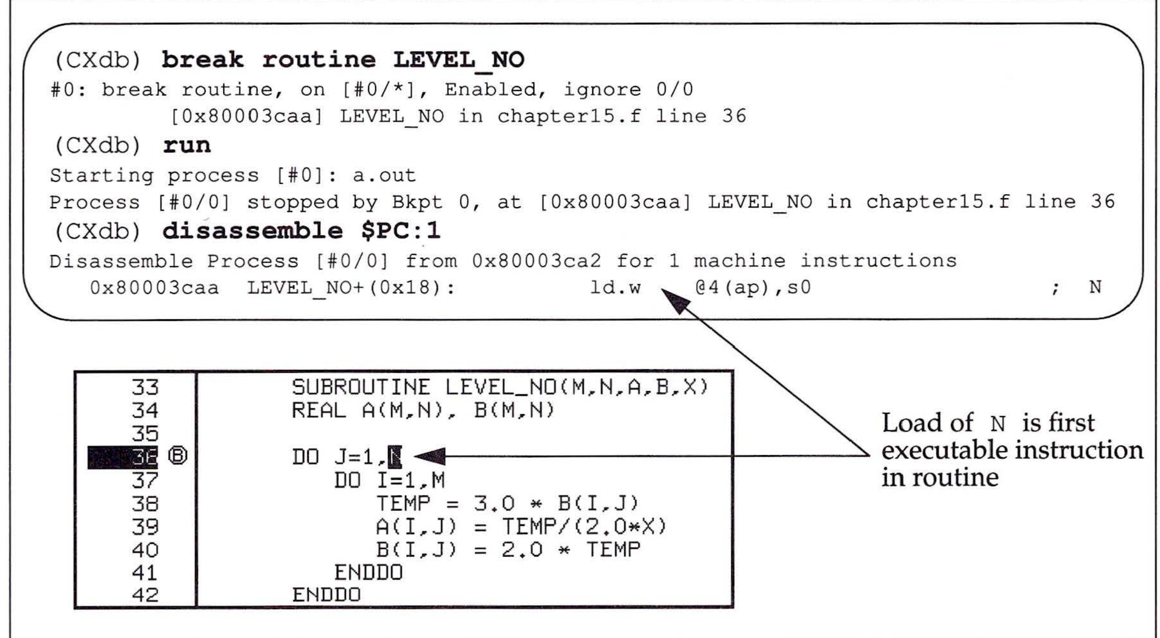
Note

The text and examples in this section are cumulative. It is strongly recommended that you read all of the material in the sequence presented. Do not skip any topics, even if you are using only one of the optimization levels discussed here.

Level `-no`

At any level of optimization, CXdb sets an eventpoint at the first machine instruction that corresponds to the location you specified for the eventpoint. For example, Figure 347 illustrates the effects of the `break routine` command at optimization level `-no`.

Figure 347
Breaking on a routine at level `-no`



The `break routine` command in Figure 347 sets a breakpoint at the machine instruction that maps to the first executable source unit of the routine `LEVEL_NO`. The `run` command starts execution. When execution is stopped by the breakpoint, the source window highlights the expression `N` because `N` is the first executable expression in the routine. The `disassemble` command verifies this fact by displaying the instruction at the current point of execution (current PC). This instruction loads the value of `N` (the highlighted source unit) into scalar register `s0`.

Components of a loop

There are 2 major components of a loop in either FORTRAN or C:

- Loop header
- Loop body

The loop header contains setup code for the loop, such as the start condition, stop condition, and loop step size of the loop. This setup code consists of a number of statement and expression source units. Because of optimization, some of these loop header source units can be merged with source units in other parts of the program.

The loop body contains the iterative portion of the loop. It is a block source unit, but it can contain other expressions, statements, loops, and blocks. Because of optimization, some of the loop body source units can be merged with each other, or they can be moved out of the loop and merged with source units in other parts of the program.

Figure 348 uses stepping by statement to reveal some facts about the header of a `DO` loop.

Figure 348
Stepping through a loop header

```
(CXdb) step statement
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80003cb2] LEVEL_NO in chapter15.f line 36
(CXdb) info line 36
     Id    Address Boundaries     Start      End      Kind
 1. ( 78) 80003cb2:80003cb6    36 x 12    36 x 12   <EXPR>   1
 2. ( 77) 80003cb2:80003cbc    36 x 10    36 x 12   <STMT>   J=1
 3. ( 79) 80003caa:80003cae    36 x 14    36 x 14   <EXPR>   N
 4. ( 76) 80003caa:80003dc2    36 x 7     42 x 11   <LOOP>   DO J=1,N   <...>  ENDDO
 5. ( 75) 80003caa:80003dc4    36 x 7     45 x 9    <BLOCK>  DO J=1,N   <...>  END
```

```
33        SUBROUTINE LEVEL_NO(M,N,A,B,X)
34        REAL A(M,N), B(M,N)
35
36 (B)    DO J=1,N
37           DO I=1,M
38              TEMP = 3.0 * B(I,J)
39              A(I,J) = TEMP/(2.0*X)
40              B(I,J) = 2.0 * TEMP
41           ENDDO
42        ENDDO
```

The `step statement` command in Figure 348 executes one statement source unit. You might anticipate that this would cause the highlighting in the source window to move from line 36 to line 37. Instead, the highlighting moves from variable `N` in line 36 to constant `1` in the same line of source code. This is because the header for the `DO` loop consists of 2 source units: one for the expression `N` and one for the statement that initializes the loop counter (`J=1`).

The `info line` command in Figure 348 displays the source units that make up the header of the `DO` loop. Source unit 76 is the loop itself. This source unit corresponds to the test for the condition `J>N.` Source unit 77 is the initialization of the loop counter, `J=1.`

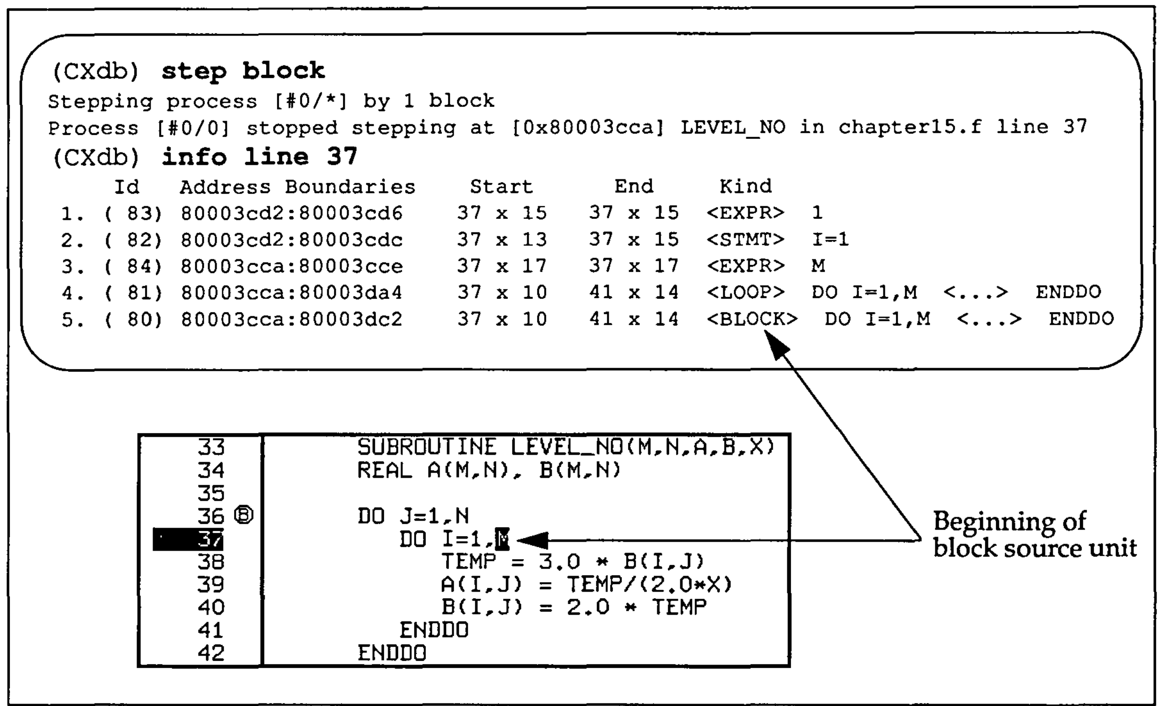
Note that the starting address of the expression source unit for `N` (source unit 79) is `80003caa.` This is the same as the starting address of the loop source unit (source unit 76). This indicates that both of these source units have the same entry point and begin with the same machine instruction. So `N` is the first expression source unit in the loop.

Similarly, the expression source unit for the constant `1` (source unit 78) has the same starting address as the statement source unit for `J=1` (source unit 77).

Also note that source unit 75 is listed as a block source unit, and it has the same starting address as the loop (source unit 76). This block is the body of the routine `LEVEL_NO.` The `DO` loop that starts on line 36 is the entire routine body in this case.

The body of a loop is a block source unit. Figure 349 shows the results of stepping by block.

Figure 349
Stepping by block at level `-no`



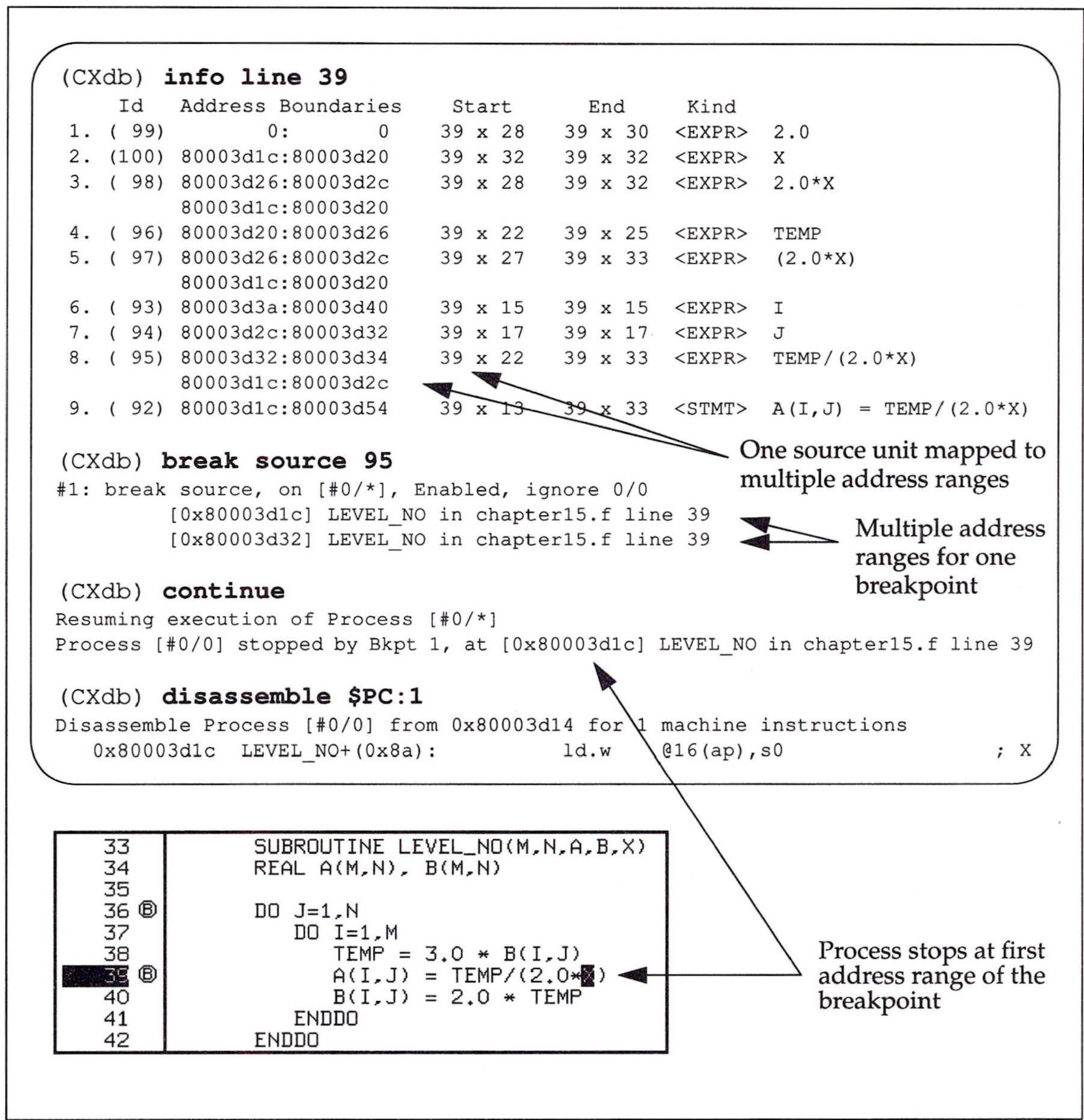
The `step block` command in Figure 349 steps execution to the beginning of the next block, which is the `DO` loop that begins on line 37. This block is also the loop body for the loop that begins on line 36. The `info line` command shows that the block is source unit 80, and its starting address is `80003cca`.

The loop source unit for the loop on line 37 (source unit 81) and the expression source unit for `M` (source unit 84) have the same starting address as the block (source unit 80). This is also the address indicated by the current PC, as shown in the response to the `step block` command

Address ranges of source units

Even at level `-no`, one source unit can map to several different ranges of machine instructions, and vice versa. Figure 350 illustrates this point.

Figure 350
Breaking at a source unit that maps to multiple address ranges



The `info line` command in Figure 350 shows that source unit 95 represents the expression `TEMP/(2.0*X)`. This source unit maps to 2 address ranges. Level `-no` optimizations such as instruction scheduling account for this dispersed mapping between source units and machine instructions.

Note that source unit 99 has an address range of 0. This means that there are no machine instructions that map directly to source unit 99.

The `break source` command in Figure 350 creates a breakpoint event for source unit 95. Because source unit 95 maps to 2 address ranges, the breakpoint also maps to those same 2 address ranges. The `continue` command continues execution, but the process stops at the first address range of the breakpoint for source unit 95. The source window highlights the expression `X`, and the `disassemble` command shows that this highlighted expression corresponds to the instruction for loading the value of `X` into register `s0`.

Continuing the process brings it to the next address range of the breakpoint for source unit 95, as illustrated in Figure 351.

Figure 351
Continuing execution to the next address range of the breakpoint

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80003d32] LEVEL_NO in chapter15.f line 39
(CXdb) disassemble $PC:1
Disassemble Process [#0/0] from 0x80003d32 for 1 machine instructions
   0x80003d32  LEVEL_NO+(0xa0):         div.s   s0,s1
(CXdb) remove event 1
Eventpoint 1 removed
```

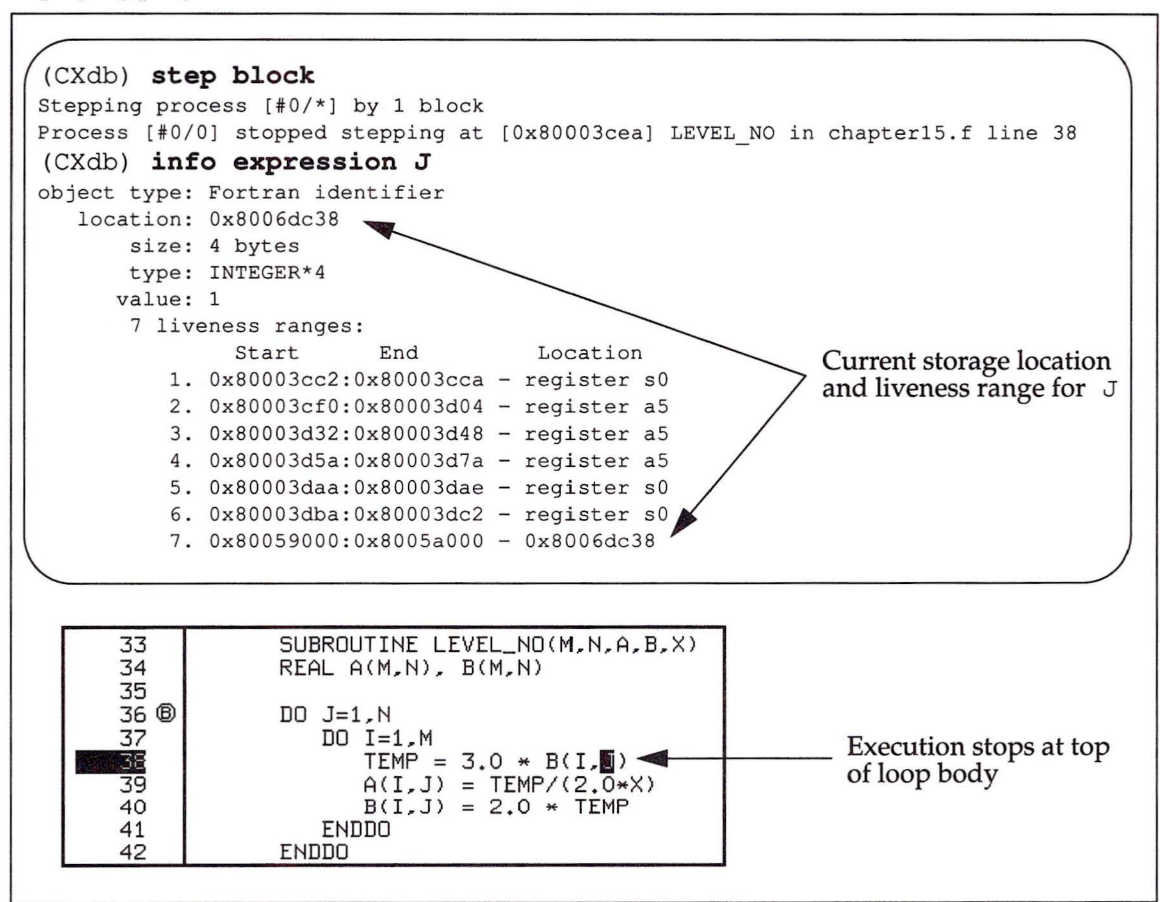
```
33      SUBROUTINE LEVEL_NO(M,N,A,B,X)
34      REAL A(M,N), B(M,N)
35
36 (B)  DO J=1,N
37         DO I=1,M
38            TEMP = 3.0 * B(I,J)
39            A(I,J) = TEMP/(2.0*X)
40            B(I,J) = 2.0 * TEMP
41         ENDDO
42      ENDDO
```

In Figure 351, the source window highlighting and the `disassemble` command show that the next breakpoint occurs at the instruction for dividing `TEMP` by `(2.0*X)`. The `remove event` command removes the breakpoint so that it does not interfere with the remaining examples in this chapter.

Displaying variables

Displaying program variables is an important aid to debugging. You can use the `print` command to display the value of a variable, but the `info expression` command provides much more than just the value, as illustrated in Figure 352.

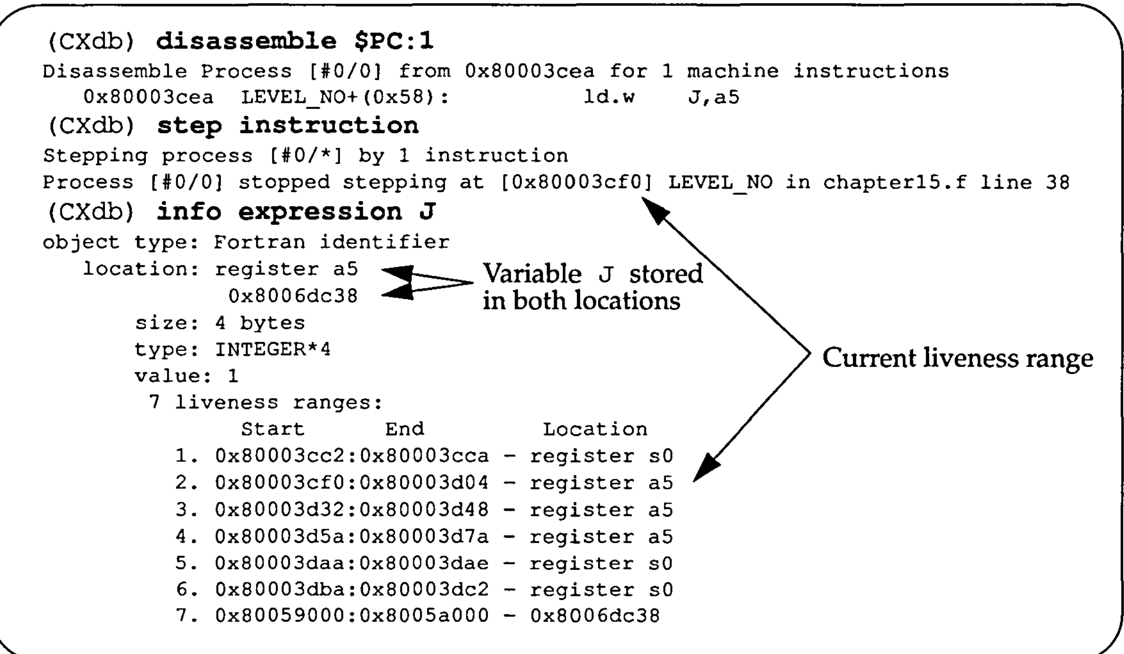
Figure 352
Displaying program variables at level `-no`



The `step block` command in Figure 352 steps the process to the top of the loop body at line 38. The `info expression` command shows the current value of the variable `J`, as well as its storage locations and liveness ranges. Currently, `J` is stored in memory at address `8006dc38`. Note that the stopping point (current PC) of the process is `80003cea`, which does not fall into any of the first 6 liveness ranges for `J`. This information indicates that `J` is not stored in a register at this point in the process.

Figure 353 shows what happens when execution brings the PC into one of the other liveness ranges for `J`.

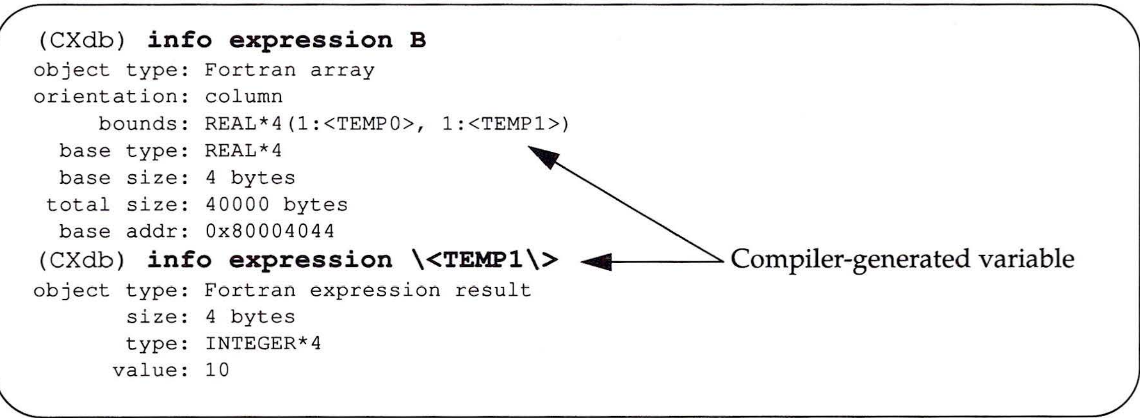
Figure 353
Stepping into another liveness range of a variable



The `disassemble` command in Figure 353 shows that the current instruction loads the value of `J` into register `a5`. The `step instruction` command executes this instruction. The `info expression` command shows that `J` is now stored in register `a5` as well as in memory. Execution stopped at address `80003cf0`, which is in liveness range 2 of variable `J`.

For purposes of optimization, the compiler generates some temporary variables which you can access with CXdb. Figure 354 shows an example of such a compiler-generated variable.

Figure 354
Displaying compiler-generated variables



The `info expression B` command in Figure 354 displays information about array variable `B`. The upper bounds of the array are defined by the variables `<TEMP0>` and `<TEMP1>`, which were generated by the compiler.

The `info expression \<TEMP1\>` command shows that the upper bound of the column subscript for array `B` is 10. The angle brackets (<>) are part of the name of the compiler-generated variable. Because angle brackets are also redirection operators in CXdb commands, a backslash (\) must precede each angle bracket in this case.

Level -O0

At level `-O0`, optimizations occur within a *basic block* but not between basic blocks. A basic block is a linear sequence of statements that has a single entry point and that ends in a conditional or unconditional branch.

Note

A basic block in the object code is generated by the compiler. It is not the same as a block source unit recognized by CXdb in the source code.

Redundant-use elimination

Within a basic block at level `-O0`, different occurrences of a variable in the source code can be merged into one load or store instruction. This type of optimization is known as redundant-use elimination. Figure 355 illustrates how CXdb displays redundant-use elimination.

Figure 355
Redundant-use elimination of a variable at level `-O0`

```
(CXdb) break routine LEVEL_O0
#2: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80003de4] LEVEL_O0 in chapter15.f line 53
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 2, at [0x80003de4] LEVEL_O0 in chapter15.f line 53
(CXdb) step block 2
Stepping process [#0/*] by 2 blocks
Process [#0/0] stopped stepping at [0x80003e14] LEVEL_O0 in chapter15.f line 57
(CXdb) disassemble $PC:1
Disassemble Process [#0/0] from 0x80003e0c for 1 machine instructions
   0x80003e14  LEVEL_O0+(0x48):          ld.w    J,a5
```

```
50        SUBROUTINE LEVEL_O0(M,N,A,B,X)
51        REAL A(M,N), B(M,N)
52
53 (B)    DO J=1,N
54           DO I=1,M
55              TEMP = 3.0 * B(I,J)
56              A(I,J) = TEMP/(2.0*X)
57              B(I,J) = 2.0 * TEMP
58           ENDDO
59        ENDDO
```

All occurrences of `J` highlighted

Line numbers are not meaningful beyond level `-no`

The `step block` command in Figure 355 brings execution to the start of the block (loop body) for the `DO` loop on line 55. The `disassemble` command shows that the first instruction of this block loads the value of `J` into register `a5`. Because `J` is referenced several times within the block, redundant-use elimination causes the value of `J` to remain in register `a5` through all instructions of the block. In the source window, CXdb highlights all occurrences of `J` that are affected by the current instruction.

Note that the source window also highlights line number 57. This does not mean that execution has stopped at line 57 in this case. It simply means that one of the highlighted source units is on this line.

Note

At optimization levels -O0 and above, source code line numbers should not be used for setting breakpoints or for stepping. Use source units, memory addresses, or machine instructions instead.

Code motion

Within a basic block at level `-O0`, local optimizations can change the order of the instructions in the object code. This is known as code motion.

A basic block can represent multiple source statements. Because of code motion between the statements of a single basic block, the execution order of the object code no longer matches the logical order of the source code statements. Figure 356 illustrates this point.

Figure 356
Code motion at level `-O0`

```
(CXdb) step statement 2
Stepping process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x80003e2a] LEVEL_O0 in chapter15.f line 56
(CXdb) print TEMP
REAL*4) 0.0000
(CXdb) print B(I,J)
REAL*4) 2.0000
(CXdb) disassemble $PC:20
Disassemble Process [#0/0] from 0x80003e2a for 20 machine instructions
   0x80003e2a  LEVEL_O0+(0x5e):        ld.w    @16(ap),s0          ; X
   0x80003e2e  LEVEL_O0+(0x62):        mov     a2,a4
   0x80003e30  LEVEL_O0+(0x64):        add.w   #1,a2
   0x80003e32  LEVEL_O0+(0x66):        add.w   #-1,a5
   0x80003e36  LEVEL_O0+(0x6a):        mul.w   a5,a1
               .
               .
               .
   0x80003e5e  LEVEL_O0+(0x92):        st.w    s1,TEMP
   0x80003e64  LEVEL_O0+(0x98):        mov.w   s1,s2
   0x80003e66  LEVEL_O0+(0x9a):        div.s   s0,s2
   0x80003e68  LEVEL_O0+(0x9c):        add.s   s1,s1
```

```
50        SUBROUTINE LEVEL_O0(M,N,A,B,X)
51        REAL A(M,N), B(M,N)
52
53 (B)    DO J=1,N
54           DO I=1,M
55              TEMP = 3.0 * B(I,J)
56              A(I,J) = TEMP/(2.0*X)
57              B(I,J) = 2.0 * TEMP
58           ENDDO
59        ENDDO
```

Variable `TEMP` not updated yet

The `step statement` command in Figure 356 advances execution to the expression for `X` on line 56 of the source code. Based on this stepping command and the highlighting in the source window, you might expect that line 55 of the source code has executed already. However, the `print` command shows that the value of `TEMP` is still `0,` while the value of `B(I,J)` is `2.` If line 55 had already executed, the value of `TEMP` should be `3.0*B(I,J),` or `6.`

The `disassemble` command in Figure 356 reveals that the instruction for storing a new value into `TEMP` has not executed yet, so `TEMP` still contains its initial value of `0.` This reordering of instructions is a result of optimizations such as instruction scheduling and redundant-assignment elimination.

At level `-O0,` the compiler can reorder any of the machine instructions within a basic block to achieve maximum execution efficiency. Because the loop body is a basic block, the compiler can move instructions from one source statement to another within the loop body. The highlighting in the source window reflects this reordering of machine instructions by moving back and forth between the source statements in the loop body.

Note

At optimization levels –O0 and above, source statements do not map to a single basic block of machine instructions. For this reason, stepping by statement can produce results that you might not expect. Step by larger units, such as routines or blocks, or by smaller units, such as instructions.

Assignment substitution

Another form of optimization at level `-O0` is assignment substitution. In this type of optimization, subsequent references to a variable are replaced by the value assigned to the variable. Figure 357 illustrates assignment substitution for the variable `TEMP`.

Figure 357
Assignment substitution at level `-O0`

```
(CXdb) step instruction 13
Stepping process [#0/*] by 13 instructions
Process [#0/0] stopped stepping at [0x80003e52] LEVEL_O0 in chapter15.f line 57
(CXdb) disassemble $PC:1
Disassemble Process [#0/0] from 0x80003e52 for 1 machine instructions
   0x80003e52  LEVEL_O0+(0x86):          mul.s   #1094713344,s1
(CXdb) info expression TEMP
object type: Fortran identifier
   location: 0x8006dc54
       size: 4 bytes
       type: REAL*4
      value: 0.0000
       1 liveness ranges:
              Start      End          Location
         1. 0x80059000:0x8005a000 - 0x8006dc54
```

```
50      SUBROUTINE LEVEL_O0(M,N,A,B,X)
51      REAL A(M,N), B(M,N)
52
53 (B)  DO J=1,N
54         DO I=1,M
55            TEMP = 3.0 * B(I,J)
56            A(I,J) = TEMP/(2.0*X)
57            B(I,J) = 2.0 * TEMP
58         ENDDO
59      ENDDO
```


The `step` command in Figure 357 advances execution to the machine instruction for the expression `3.0*B(I,J)` in line 55. The `disassemble` command shows this instruction. Because assignment substitution replaces the variable `TEMP` with the value of `3.0*B(I,J)`, all subsequent occurrences of `TEMP` are highlighted in the source window.

The `info expression` command in Figure 357 shows that `TEMP` has a liveness range and storage location. This indicates that the variable `TEMP` is still available, even though assignment substitution has replaced `TEMP` with the value `3.0*B(I,J)`. Optimizations at higher levels can result in the elimination of the original variable, as the following sections illustrate.

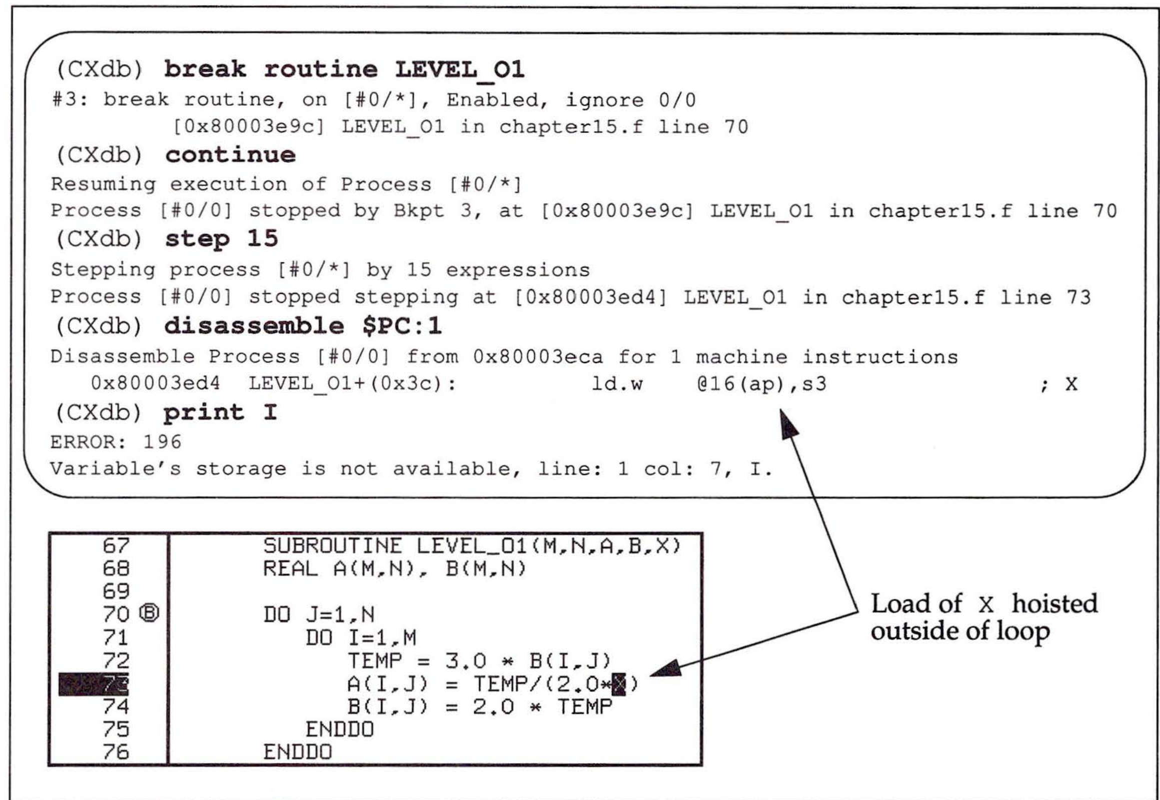
Level -O1

At level -O1, optimizations occur between basic blocks as well as within them. This causes the mapping between source units and machine instructions to be even more dispersed than at level -O0. In addition, synthesized variables are introduced at level -O1.

Code motion

Optimizations at level -O1 move loop-invariant operations out of the loop body and place them in a basic block preceding or following the loop. Figure 358 illustrates some of the effects of code motion at level -O1.

Figure 358
Code motion at level -O1



The break routine, continue, and step commands in Figure 358 advance process execution to the instruction for loading the value of X into register s3. The disassemble command shows this instruction.

From the source code, `X` appears to be inside the loop body. However, `X` is a constant whose value does not change inside the loop. Therefore, the instruction to load `X` into a register is hoisted to a location preceding the loop. The `print` command shows the loop counter `I` has no current value. This is because execution has not yet entered the loop body.

Synthesized variables

As a result of optimizations at level `-O1`, the compiler generates synthesized variables to replace less efficient calculations for values of program variables. For example, your program might use a loop induction variable as an index to array elements. At level `-O1`, the compiler replaces this induction variable with a synthesized pointer to the array elements. This is a form of optimization known as strength reduction. Figure 359 illustrates how synthesized variables replace a loop induction variable.

Figure 359
Strength reduction of a loop induction variable

```
(CXdb) step block 3
Stepping process [#0/*] by 3 blocks
Process [#0/0] stopped stepping at [0x80003ef8] LEVEL_O1 in chapter15.f line 72

(CXdb) info expression J
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 1
       used to create 7 synthesized variable(s):
         1. <INDV>      ?i0 = J+((-1*N)-1)
         2. <INDV>      ?i1 = loc(B)+((4*M)*(J-1))
         3. <INDV>      ?i2 = loc(A)+((4*M)*(J-1))
         4. <INDV>      ?i5 = (-4+?i1)+(4*(I-1))
         5. <INDV>      ?i6 = ?i1+(4*(I-1))
         6. <INDV>      ?i7 = ?i2+(4*(I-1))
         7. <SEXP>      ?c8 = ?i1+(-1*(4*(1+(-1*M))))

(CXdb) print/x \?i2
(INTEGER*4) 0x8005a3a8

(CXdb) print/x \?i7
(INTEGER*4) 0x8005a3b0

(CXdb) examine/f loc(A):10
Examine Process [#0/0] from 0x8005a3a8 to 0x8005a3cc
 8005a3a8:         2.3077      3.4615      0.0000      0.0000      0.0000
 8005a3bc:         0.0000      0.0000      0.0000      0.0000      0.0000
```

Annotations:
- location: Variable `J` not stored because it is replaced by synthesized variables
- value: Value of `J` calculated from equations
- ?i2: Pointer to current row of array `A`
- ?i7: Pointer to current element of `A`
- print/x \?i2 result: Address of current row of array `A`
- print/x \?i7 result: Address of current element of `A` (arrow points to the boxed value 0.0000 at 8005a3b0)

The `step block` command in Figure 359 advances execution through 2 iterations of the inner loop body and stops execution at the beginning of the third iteration. This generates the first 2 elements of arrays `A` and `B`.

The `info expression` command in Figure 359 shows that the program variable `J` gives rise to seven synthesized variables. The name, equation, and reason for each synthesized variable are listed. The current value of `J` is 1. Note that there is no storage location or liveness range listed for `J`. This is because `J` has been completely replaced by the synthesized variables. Even though the value of `J` is not stored directly, CXdb can calculate it from the equations for the synthesized variables. (Refer to the section "Synthesized variables" in this chapter for an explanation of the reason codes.)

The first synthesized variable of interest in the above example is `?i2`. The reason for `?i2` is INDV, which means that it provides strength reduction of a loop induction variable (`J` in this case). The base of the equation for `?i2` is the starting address of array `A`, as designated by the FORTRAN function `loc(A)`. Thus, `?i2` is a *pointer* to the current row of `A`, just as `J` is an *index* to the current row. Each time the induction variable `J` is incremented by 1, the synthesized variable `?i2` is incremented by `4*M`, where `M` is the number of elements in one row and `4` bytes is the size of one element of `A`. Because the current value of `J` is 1, `?i2` currently points to the first row of `A`.

The synthesized variable `?i7` also provides strength reduction of an induction variable (INDV). The equation for `?i7` starts with `?i2` as a base. Each time the program variable `I` is incremented by 1, `?i7` is incremented by `4`, which is the number of bytes in one array element. Thus, `?i7` is a pointer to the current element of array `A`.

The `print` commands in Figure 359 display the values of `?i2` and `?i7` in hexadecimal format. The `examine` command displays the first 10 elements of array `A`, which include the current element referenced by `?i7`.

Inconsistent values for a variable

One program variable usually gives rise to many synthesized variables at level `-O1` and above. Each synthesized variable gives rise to an individual equation for calculating its value, but the equations are expressed in terms of other synthesized variables. Trying to solve these equations simultaneously to determine the current value of a program variable can sometimes lead to inconsistent values. Figure 360 illustrates this point.

Figure 360
Inconsistent values derived from synthesized variables at level `-O1`

```
(CXdb) step 3
Stepping process [#0/*] by 3 expressions
Process [#0/0] stopped stepping at [0x80003f04] LEVEL_O1 in chapter15.f line 72

(CXdb) info expression I
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 3
       used to create 3 synthesized variable(s):
         1. <INDV>    ?i5 = (-4+?i1)+(4*(I-1))
         2. <INDV>    ?i6 = ?i1+(4*(I-1))
         3. <INDV>    ?i7 = ?i2+(4*(I-1))

INFO: 410
Inconsistent induction value for I, using 3, from the set of {3, 3, 4}

(CXdb) disassemble $PC:1
Disassemble Process [#0/0] from 0x80003efa for 1 machine instructions
   0x80003f04  LEVEL_O1+(0x6c):         add.w   #4,a2

(CXdb) info registers
Process [#0/0]
pc : 0x80003f04
psw: 0x03909480
fp : 0xffffca20
ap : 0x800040d4
a5 : 0xfffff064
a4 : 0x80063fe8 ?i1
a3 : 0x8005a3b4 ?i7
a2 : 0x80063ff0 ?i6
a1 : 0x80064f84
sp : 0xffffca18
s7 : 0x00d0000000000fa0 ?c4
s6 : 0x40380000fffffff6 ?i0
s5 : 0x00157f0380064f84 ?c8
s4 : 0x4552313580063fec ?i5
s3 : 0x4e45204341266666 ?c9
s2 : 0x535542524591ac4f
s1 : 0x54415254415d89d9
s0 : 0x0000000042400000
```


The `info expression` command in Figure 360 shows a value of 3 for variable `I`, but an informational messages also displays to indicate that other values for `I` are equally valid at this point in the process execution. CXdb simply picks the lowest value in the set of possible values, and it prints a message to indicate the other possibilities.

The reason for the inconsistent values is also revealed in Figure 360. The `disassemble` command shows that the next instruction modifies the contents of register `a2`. The `info registers` command shows that register `a2` contains the value of the synthesized variable `?i6`. Because `?i6` is one of the synthesized variables used to calculate the value of `I`, modifying the contents of register `a2` (and therefore `?i6`) also affects the value of `I`.

In the above example, execution stopped at a point in the process where some of the synthesized variables used to derive the value of `I` had been modified but the rest had not. Inconsistent values generally occur at such locations in the process.

For each basic block in the object code, CXdb determines the value, availability, and liveness ranges of a variable. At the beginning of a basic block such as a loop body, variable values are consistent, as Figure 361 illustrates.

Figure 361
Variable values remain consistent within a basic block

```
(CXdb) step block
Stepping process [#0/*] by 1 block
Process [#0/0] stopped stepping at [0x80003ef8] LEVEL_01 in chapter15.f line 72
(CXdb) info expression I
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 4
       used to create 3 synthesized variable(s):
         1. <INDV>    ?i5 = (-4+?i1)+(4*(I-1))
         2. <INDV>    ?i6 = ?i1+(4*(I-1))
         3. <INDV>    ?i7 = ?i2+(4*(I-1))
```

```
  67        SUBROUTINE LEVEL_01(M,N,A,B,X)
  68        REAL A(M,N), B(M,N)
  69
  70 Ⓑ      DO J=1,N
  71           DO I=1,M
  72              TEMP = 3.0 * B(I,J)
  73              A(I,J) = TEMP/(2.0*X)
  74              B(I,J) = 2.0 * TEMP
  75           ENDDO
  76        ENDDO
```

The `step block` command in Figure 361 advances execution to the beginning of the basic block that is inner loop body. The `info expression` command shows that the variable `I` has a single, consistent value of 4 at this location. The value of `I` remains 4 throughout one full iteration of the inner loop. The value becomes inconsistent again when execution steps beyond the boundaries of the inner loop body.

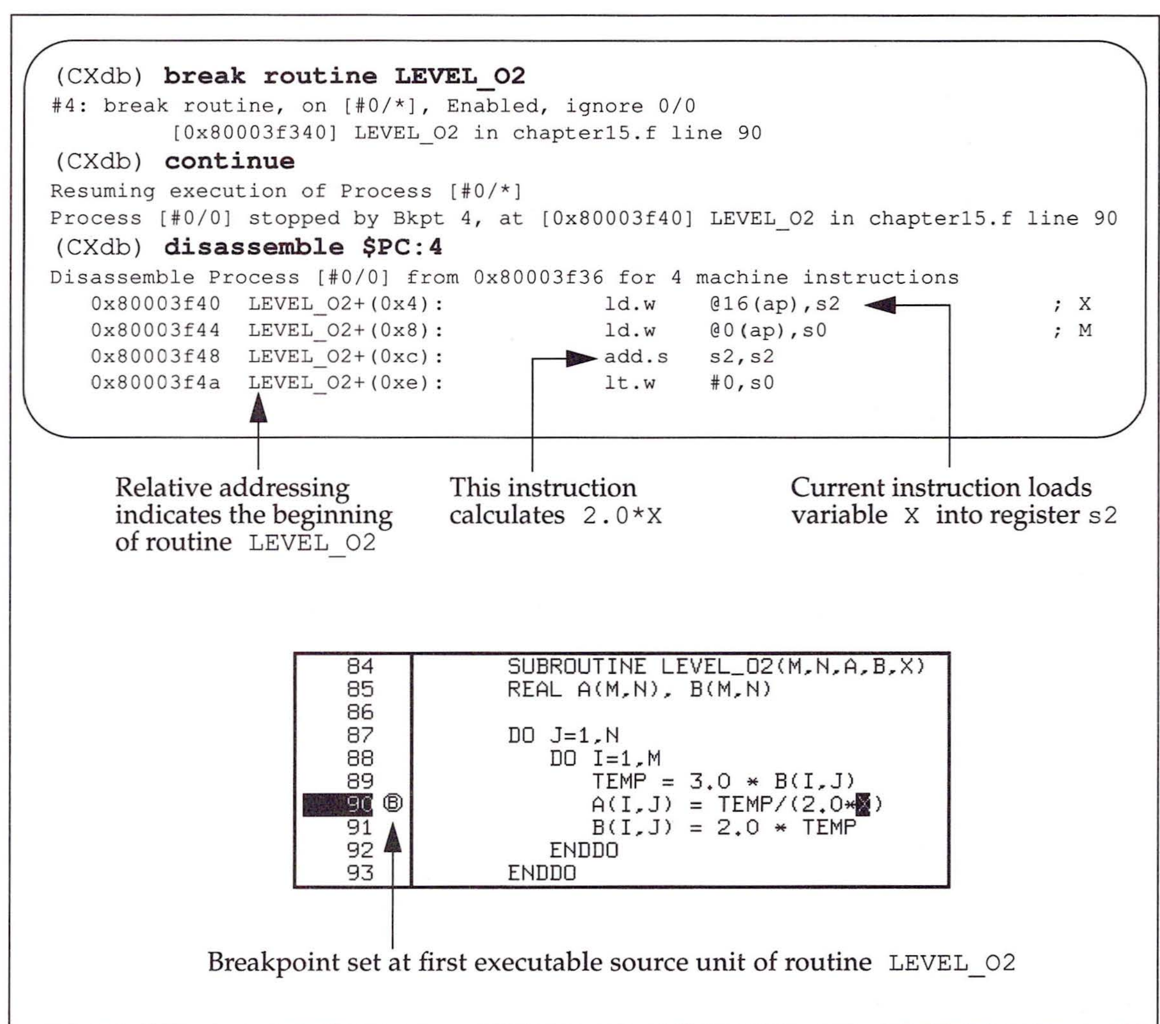
Level -O2

At level -O2, the optimizations focus on the vectorization of loops. Vectorization involves the use of the vector registers that are available on CONVEX machines.

Code motion

Optimizations at level -O2 move loop-invariant operations out of the loop body and place them in a basic block preceding or following the loop. Figure 362 illustrates some effects of code motion at level -O2.

Figure 362
Code motion at level -O2



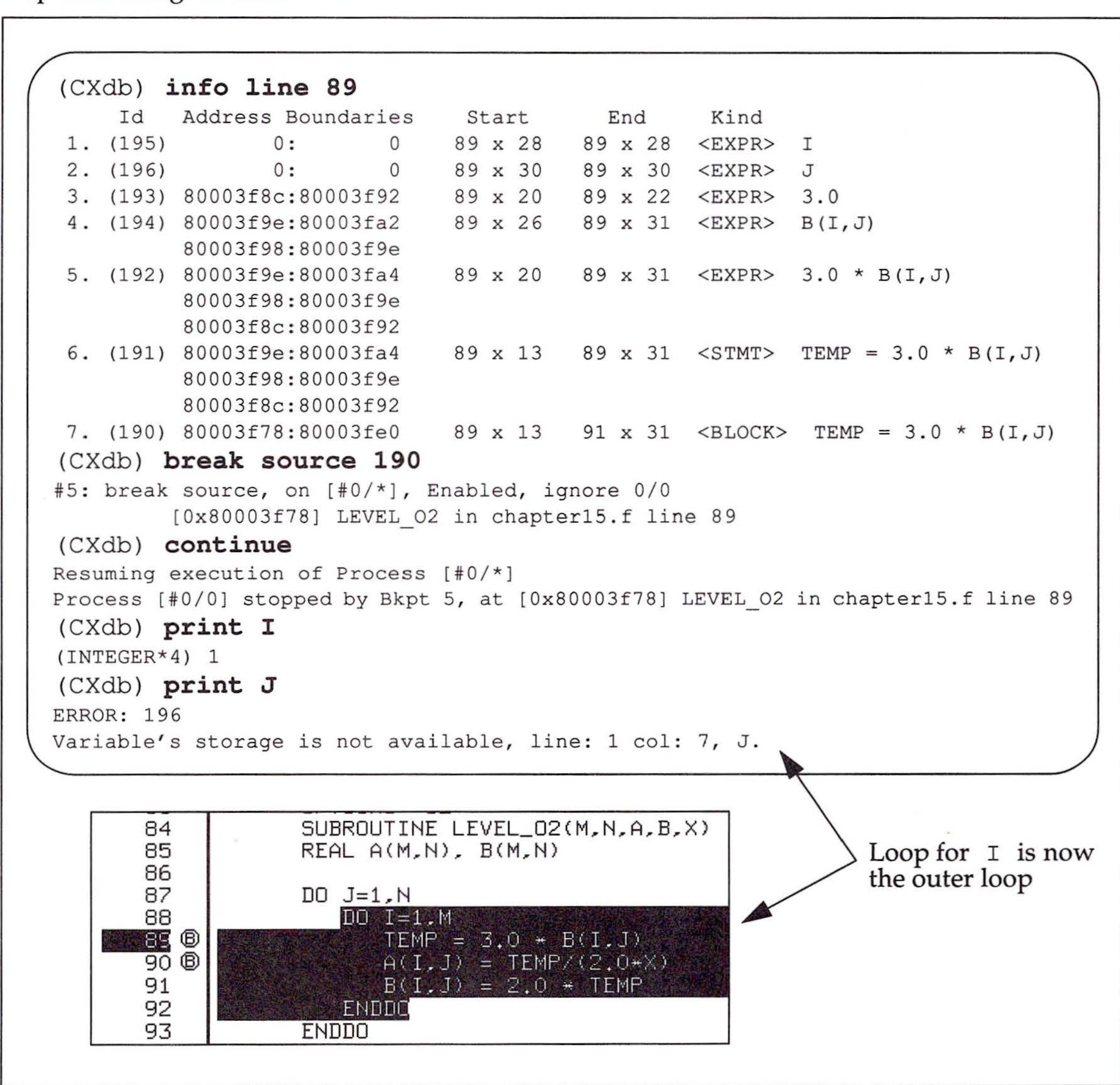
The `break routine` command in Figure 362 sets a breakpoint at the first executable source unit of the routine `LEVEL_O2`. Contrary to what you might expect, the breakpoint marker and the highlighting in the source window appear at line 90 in the source code rather than at line 87. This is because the expression `2.0*X` has been hoisted out of the `DO` loop and moved to the beginning of the routine. The `disassemble` command verifies that the current instruction is at the beginning of the routine, as indicated by the relative address `LEVEL_O2+(0x4)`.

Even though the breakpoint appears to be in the middle of the `DO` loop in Figure 362, it does not stop execution of the loop. This is because the breakpoint is associated with a source unit that has been hoisted out of the loop.

Loop interchange

Another important type of optimization at level `-O2` is loop interchange, which interchanges the inner and outer loops of a nested pair. Loop interchange improves vectorization and memory accesses. Figure 363 shows some of the effects of loop interchange.

Figure 363
Loop interchange at level `-O2`



The `info line` command in Figure 363 shows that source unit 190 is the block for the body of the `I` loop. The `break source` command sets a breakpoint at source unit 190. The `continue` command advances process execution until it is stopped by the breakpoint. The breakpoint marker and the highlighting in the source window confirm that source unit 190 is associated with the `I` loop.

The `print I` command in Figure 363 shows that `I` has a current value of 1. However, the `print J` command shows that the value of `J` is not available. This is because execution has not yet entered the `J` loop. The 2 loops have been interchanged, so the `I` loop is now the outer loop and the `J` loop is the inner loop. Source unit 190 and its associated breakpoint are part of the outer (`I`) loop.

Figure 364 shows how to stop execution at the loop body of the inner (J) loop.

Figure 364
Setting a breakpoint at the inner loop body

```
(CXdb) info line 88
     Id   Address Boundaries       Start       End       Kind
 1. (188)         0:         0     88 x 15    88 x 15    <EXPR>   1
 2. (187)         0:         0     88 x 13    88 x 15    <STMT>   I=1
 3. (189) 80003f54:80003f58        88 x 17    88 x 17    <EXPR>   M
          80003f44:80003f48
 4. (186) 80003f40:80003fe0        88 x 10    92 x 14    <LOOP>   DO I=1,M <...> ENDDO
 5. (185) 80003f9e:80003fbe        88 x 10    92 x 14    <BLOCK>  DO I=1,M <...> ENDDO
(CXdb) break source 185
#6: break source, on [#0/*], Enabled, ignore 0/0
        [0x80003f9e] LEVEL_02 in chapter15.f line 89
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 6, at [0x80003f9e] LEVEL_02 in chapter15.f line 89
(CXdb) print I
(INTEGER*4) 1
(CXdb) print J
(INTEGER*4) 1
(CXdb) print A
REAL*4(1:<TEMP0>, 1:<TEMP1>)
(1..1000,1) :        0.0000      0.0000      0.0000      0.0000      0.0000
```

```
84        SUBROUTINE LEVEL_02(M,N,A,B,X)
85        REAL A(M,N), B(M,N)
86
87        DO J=1,N
88           DO I=1,M
89 ★             TEMP = 3.0 * B(I,J)
90 Ⓑ             A(I,J) = TEMP/(2.0*X)
91               B(I,J) = 2.0 * TEMP
92           ENDDO
93        ENDDO
```

The body of the J loop is really the block source unit that starts on line 88 of the source code. The info line command in Figure 364 shows that this block is source unit 185. The break source command sets a breakpoint at source unit 185, and the continue command advances execution to that breakpoint.

The print commands in Figure 364 show that both I and J have a current value of 1. Therefore, execution has reached the start of the J (inner) loop. The print A command shows that none of the elements of array A have been modified, so no iterations of the J loop have occurred yet.

Vectorization

The purpose of loop interchange is to improve vectorization and memory access. Vectorization converts scalar loop operations into equivalent vector operations. Figure 365 illustrates some of the effects of vectorization at level `-O2`.

Figure 365
Vectorization of a loop at level `-O2`

```
(CXdb) disassemble $PC:4
Disassemble Process [#0/0] from 0x80003f94 for 4 machine instructions
   0x80003f9e  LEVEL_O2+(0x62):         ld.w    0(a2),v0
   0x80003fa2  LEVEL_O2+(0x66):         mul.s   v0,s0,v1
   0x80003fa4  LEVEL_O2+(0x68):         div.s   v1,s3,v0
   0x80003fa6  LEVEL_O2+(0x6a):         st.w    v0,0(a3)
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 6, at [0x80003f9e] LEVEL_O2 in chapter15.f line 89
(CXdb) print A(1..10,1..5)
REAL*4(1:10, 1:5)
(1..10,1) : 2.3077 3.4615 4.6154 5.7692 6.9231 8.0769 9.2308 10.3846 11.5385 12.6923
(1..10,2) : 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000  0.0000  0.0000  0.0000
(1..10,3) : 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000  0.0000  0.0000  0.0000
(1..10,4) : 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000  0.0000  0.0000  0.0000
(1..10,5) : 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000  0.0000  0.0000  0.0000
(CXdb) print A(125..130,1..5)
REAL*4(125:130, 1:5)
(125..130,1) : 145.3846 146.5385 147.6923 148.8462 0.0000 0.0000
(125..130,2) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
(125..130,3) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
(125..130,4) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
(125..130,5) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
(CXdb) print I
(INTEGER*4) 1
(CXdb) print J
(INTEGER*4) 2
(CXdb) print $vs
(INTEGER*4) 4        <---- Vector stride
(CXdb) print $vl
(INTEGER*4) 128      <---- Vector length
(CXdb) print/f $v0
INTEGER*4(0:127)     <---- Vector register v0 contains the values of array A
(0)= 2.3077
(1)= 3.4615
(2)= 4.6154
    .
    .
    .
(125)= 146.5385
(126)= 147.6923
(127)= 148.8462
```

First 128 elements of array `A` are processed

The `disassemble` command in Figure 365 shows that the next sequence of machine instructions uses the vector registers. The `continue` command advances execution through one iteration of the inner (`J`) loop because the breakpoint at the beginning of the `J`-loop body stops execution after each iteration of the inner loop. During the loop iteration, the process performs the vector operations indicated by the `disassemble` command.

The `print A(1..10,1..5)` command prints elements 1 through 10 of rows 1 through 5 in array `A`. The `print A(125..130,1..5)` command prints elements 125 through 130 of rows 1 through 5 in array `A`. These 2 commands show that the first 128 elements of array `A` have been processed, even though execution has gone through only one iteration of the inner loop. This is a result of the vectorization technique known as strip mining.

Strip mining processes up to 128 array elements at one time. It stems from the fact that each vector accumulator register can hold up to 128 elements.

In Figure 365, the `print I` and `print J` commands show that `J` has been incremented to 2, but `I` is still 1. This confirms that the `J` (inner) loop has completed only one iteration so far. For each iteration of the `J` loop, the program processes 128 elements per row of array `A`. Thus, it is the `I` loop that is vectorized and strip mined in this case.

The `print $vs` command in Figure 365 prints the contents of the vector stride register, `vs`. The vector stride is the number of bytes between consecutive array elements. Because each element of array `A` is 4 bytes long, the vector stride is 4 in this case.

The `print $vl` command in Figure 365 prints the contents of the vector length register, `vl`. The vector length is the number of array elements loaded into the vector accumulators during each iteration of the loop. Therefore, it is also the number of elements strip mined during each iteration of the loop. The vector length in this case is 128, which is the maximum possible.

The `print/f $v0` command in Figure 365 prints the contents of vector accumulator register `v0` in floating point format (`/f`). The response to this command shows that `v0` contains the elements that have just been processed for array `A`.

Figure 366 shows the results of strip mining for one more iteration of the J loop.

Figure 366
Effects of strip mining

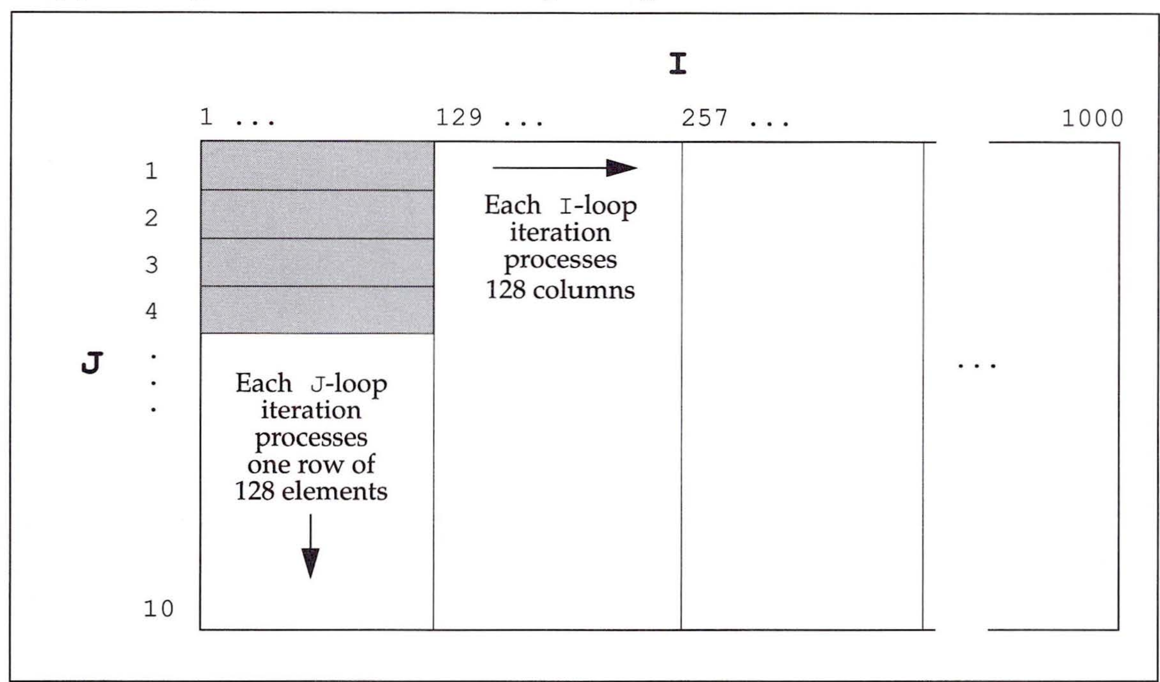
```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 6, at [0x80003f9e] LEVEL_02 in chapter15.f line 89

(CXdb) print I
(INTEGER*4) 1
(CXdb) print J
(INTEGER*4) 3

(CXdb) print A(1..5,1..5)
REAL*4(1:5, 1:5)
(1..5,1) : 2.3077 3.4615 4.6154 5.7692 6.9231
(1..5,2) : 3.4615 4.6154 5.7692 6.9231 8.0769
(1..5,3) : 0.0000 0.0000 0.0000 0.0000 0.0000
(1..5,4) : 0.0000 0.0000 0.0000 0.0000 0.0000
(1..5,5) : 0.0000 0.0000 0.0000 0.0000 0.0000

(CXdb) print A(125..130,1..5)
REAL*4(125:130, 1:5)
(125..130,1) : 145.3846 146.5385 147.6923 148.8462 0.0000 0.0000
(125..130,2) : 146.5385 147.6923 148.8462 150.0000 0.0000 0.0000
(125..130,3) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
(125..130,4) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
(125..130,5) :   0.0000   0.0000   0.0000   0.0000 0.0000 0.0000
```

Each iteration of the J loop processes 128 elements per row of array A

The continue command in Figure 366 executes the second iteration of the J (inner) loop. This iteration processes the first 128 elements in row 2 (J=2) of array A. Then J is incremented to 3 for the beginning of the next iteration.

Figure 367 illustrates how vectorization and strip mining affect array processing in the above example.

Figure 367
Array processing under vectorization and strip mining



As Figure 367 illustrates, each iteration of the `J` (inner) loop processes 128 elements in one row of the array. In the first 10 iterations of `J`, elements 1 through 128 of rows 1 through 10 are processed. After the tenth iteration of the `J` loop, `J` is reset to 1 and `I` is incremented to 129. The eleventh iteration then fills elements 129 through 256 of row 1.

Ten iterations of the J (inner) loop is equivalent to one iteration of the I (outer) loop. Figure 368 shows the results after one complete iteration of the outer loop.

Figure 368
Completing one iteration of the outer loop

```
(CXdb) continue 8
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 5, at [0x80003f78] LEVEL_02 in chapter15.f line 89
(CXdb) print I
(INTEGER*4) 129
(CXdb) print J
ERROR: 196
Variable's storage is not available, line: 1 col: 7, J.
(CXdb) print A(127..130,1..10)
REAL*4(127:130, 1:10)
(127..130,1) :  147.6923 148.8462 0.0000 0.0000
(127..130,2) :  148.8462 150.0000 0.0000 0.0000
(127..130,3) :  150.0000 151.1539 0.0000 0.0000
(127..130,4) :  151.1539 152.3077 0.0000 0.0000
(127..130,5) :  152.3077 153.4615 0.0000 0.0000
(127..130,6) :  153.4615 154.6154 0.0000 0.0000
(127..130,7) :  154.6154 155.7692 0.0000 0.0000
(127..130,8) :  155.7692 156.9231 0.0000 0.0000
(127..130,9) :  156.9231 158.0769 0.0000 0.0000
(127..130,10) : 158.0769 159.2308 0.0000 0.0000
```

128 elements of each row filled

The `continue` command in Figure 368 advances execution through iterations 3 to 10 of the J (inner) loop. Execution stops at the breakpoint for the body of the outer (I) loop. At this point, I is incremented to 129 but J has not been reset yet, as indicated by the `print I` and `print J` commands. The `print A(127..130,1..10)` command shows that the first 128 elements of all 10 rows have been filled.

The next 10 iterations of the J loop fill elements 129 through 256 of each row, as illustrated by Figure 369.

Figure 369
Processing columns 129 through 256

```
(CXdb) continue 2
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 6, at [0x80003f9e] LEVEL_02 in chapter15.f line 89
(CXdb) print I
(INTEGER*4) 129
(CXdb) print J
(INTEGER*4) 2
(CXdb) print A(127..130,1..3)
REAL*4(127:130, 1:3)
(127..130,1) : 147.6923 148.8462 150.0000 151.1539
(127..130,2) : 148.8462 150.0000   0.0000   0.0000
(127..130,3) : 150.0000 151.1539   0.0000   0.0000
(CXdb) print A(255..258,1..3)
REAL*4(255:258, 1:3)
(255..258,1) : 295.3846 296.5385   0.0000   0.0000
(255..258,2) :   0.0000   0.0000   0.0000   0.0000
(255..258,3) :   0.0000   0.0000   0.0000   0.0000
```

Elements 129 through 256 of row 1 are processed

The continue command in Figure 369 advances execution through one iteration of the J (inner) loop. The print A(127..130,1..3) and print A(255..258,1..3) commands show that this iteration has processed elements 129 through 256 in row 1 of array A.

Level -O3

The optimizations at level `-O3` generate parallel threads of object code that can execute simultaneously on separate CPU's within the same machine. CXdb provides special commands for debugging threads. These commands and the level `-O3` optimizations are discussed in Chapter 16.

Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit from CXdb by issuing the `quit` command, as shown in Figure 370.

Figure 370
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Debugging a program with multiple threads

16

Multiple threads within a program introduce another dimension to the debugging of a process. Several CXdb commands exist to help debug a process with more than one thread. Special eventpoints can be used to trap a thread executing a `spawn` or `join` instruction. Also, information can be requested on a particular thread of a process, or a signal can be passed to a particular thread.

In addition to these new commands, many of the CXdb commands that have already been discussed can be directed to a particular thread rather than the all threads of a process. This chapter covers the use of these thread-related commands.

The following commands are covered:

- `clear default fixed sched`
- `clear fixed sched`
- `event exec`
- `event join`
- `event spawn`
- `info threads`
- `set default fixed sched`
- `set fixed sched`
- `signal thread`

Note

To work the examples in this chapter, your CONVEX machine must be capable of supporting multiple threads. The examples in this chapter were created on a CONVEX C200 Series machine with two CPUs.

Preparing for the examples

If you plan to work the examples in this chapter, you can prepare for them by executing the commands shown in Figure 371.

Figure 371
Invoking CXdb

```
%cd /usr/lib/cxdb/examples/example3
%cxdb a.out
```

The `cd` command in Figure 371 changes directories to the directory where the example program files are stored. The `cxdb` command invokes CXdb and gives it the name of the executable file for the examples.

Threads

A thread is an independent execution stream. An execution stream is a sequence of instructions that are executed by a CPU. A process is made up of one or more threads, each of which can be concurrently executing on a different CPU. The example program of the previous chapters was a single-threaded process.

A CONVEX machine with multiple heads can use threads to speed up time to solution. For example, each thread may execute a sequence of instructions which calculate the values of a large array. Each thread, operating on a different CPU, calculates a different portion of the array.

Parallel optimizations performed by the compiler at the `-O3` level generate an executable that can be multithreaded. Each thread executes an identical set of instructions. This type of parallel processing is called symmetric, and is described in the next section.

You can also explicitly spawn threads from a single thread using a `pfork` instruction. This is an asymmetric form of parallel processing. Processes using either form of parallel processing can be debugged with CXdb.

Note

Differences in CPU availability and resource contention (for example, memory bank contention) may cause the behavior of the threads of a multithreaded process to differ each time it executes. This unpredictable nature is called non-determinism. Though the examples in this chapter have been constructed to behave as consistently as possible, you may experience different behavior while executing the examples.

Optimizations at level -O3

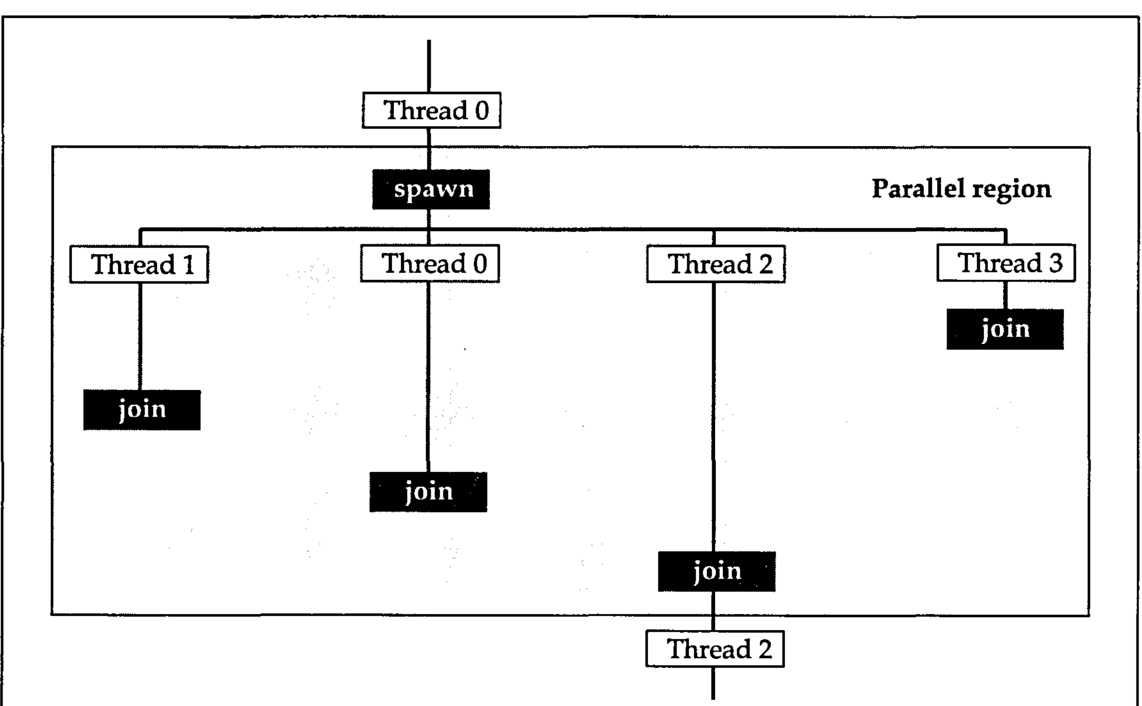
When you compile your program at the `-O3` optimization level, the compiler creates object code that can generate multiple threads. By default, the compiler attempts to only parallelize loops in your program. Typically, a strip-mined loop (created when a loop is vectorized) is parallelized.

Each iteration of a parallelized loop is called a chore. A set of chores eligible to execute in parallel represents a parallel region.

Upon entry to a parallel region, the process executes a `spawn` instruction. The `spawn` instruction requests the allocation of idle CPUs to create multiple threads. The actual number of threads created depends upon the total number of CPUs and CPU availability.

The set of instructions in each thread is a loop which performs chores. When a thread completes a chore, it executes another chore. When there are no more chores to execute, the thread joins (performs a `join`, `cfork`, or `wfork` instruction), and then terminates. However, the last thread remaining does not terminate, but rather continues execution of the process. This is shown in Figure 372.

Figure 372
Execution of a parallel region

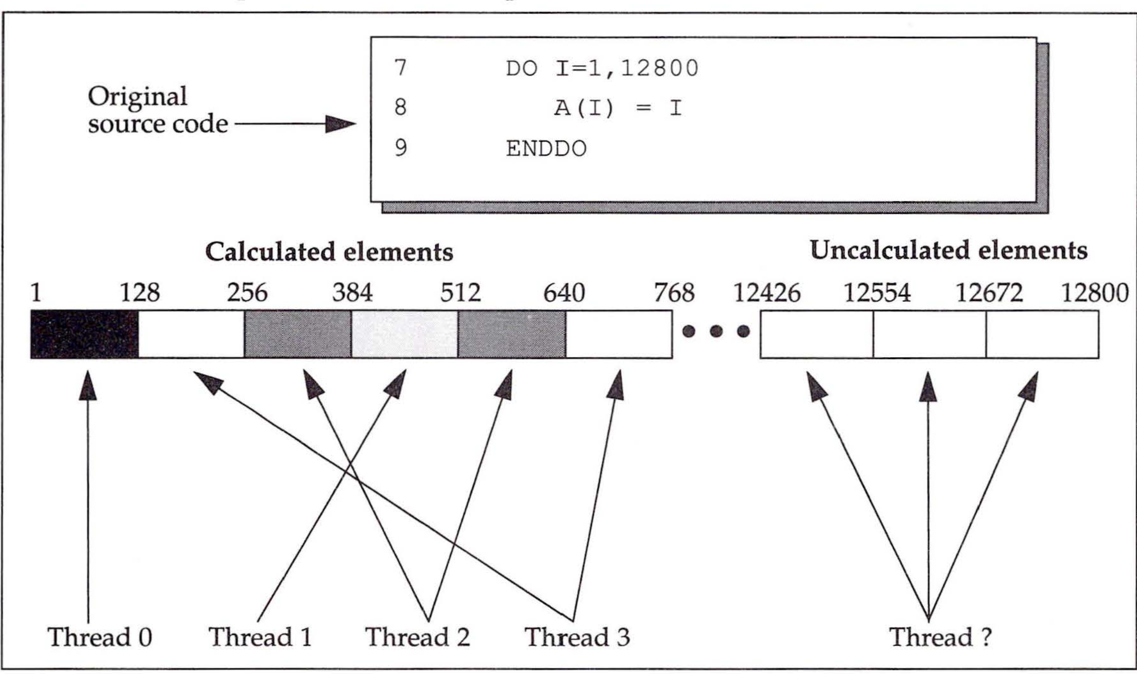


In Figure 372, the process consists of a single thread (thread 0) upon entry to the parallel region. When the next-to-last thread joins (in this case, thread 0), the process is once again single threaded. Execution is continued by thread 2 (the last thread to join).

The example program for this chapter has a simple loop. The program was compiled at level `-O3`. The compiler strip mined and parallelized the loop. The strip mining of a loop causes each iteration (or chore, in the case of parallelization) of the loop to calculate a vector length of the array at a time. In this case, the vector length is 128, so each iteration of the loop calculates 128 elements.

The distribution of chores between threads is completely non-deterministic, due to resource contentions (such as memory bank contention). Figure 373 illustrates the unpredictable distribution of chores between threads. Each shaded box represents a chore executed by a specific thread. The unshaded boxes represent chores that still have not been executed.

Figure 373
Parallel and vector optimizations to the loop



Fixed scheduling

Fixed scheduling ensures that all CPUs of a machine are made available for each timeslice of a process. Fixed scheduling does not ensure that multiple threads are created. However, fixed scheduling should be enabled before entering a parallel region because it minimizes the non-determinism of threads.

Within CXdb, you can enable or disable fixed scheduling by changing the fixed scheduling setting of the process object. This setting must be changed before a process is created to affect the process. The initial value of the setting is the same as the default fixed scheduling setting. The default fixed scheduling setting is initially disabled. For more information on process settings, refer to Chapter 8.

The default fixed scheduling setting is enabled with the `set default fixed sched` command and disabled with the `clear default fixed sched` command.

The fixed scheduling setting for a process object is enabled with the `set fixed sched` command and disabled with the `clear fixed sched` command. To increase the chances of the example program generating multiple threads, fixed scheduling should be enabled, as shown in Figure 374.

Figure 374
Enabling fixed scheduling

```
(CXdb) set fixed sched
```

In Figure 374, the `set fixed sched` command sets fixed scheduling for the process object. Any process now created will be run with fixed scheduling enabled.

Spawn eventpoints

A spawn eventpoint stops process execution when a thread is created. CXdb stops all threads of the process and opens a new source window for the new thread.

You can then debug the process using thread-specific commands. To set a spawn eventpoint, use the `event spawn` command, as shown in Figure 375.

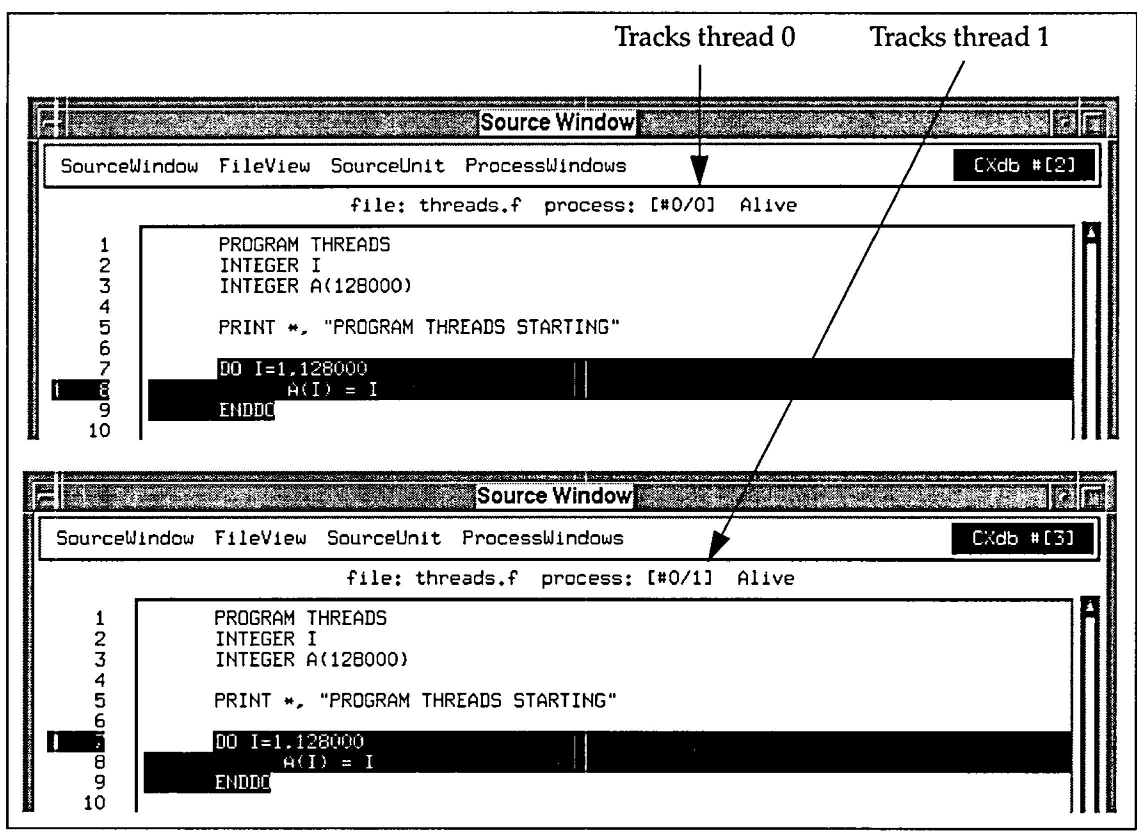
Figure 375
Setting a spawn eventpoint

```
(CXdb) event spawn

#1: spawn, on [#0], Enabled, ignore 0/0
(CXdb) run
Starting process [#0]: a.out
Process [#0/1] thread spawned
Process [#0/0] stopped at [0x8000140a] THREADS in threads.f line 8
Process [#0/1] stopped by Eventpoint 0, at [0x800013a6] THREADS in threads.f line 7
```

In Figure 375, the `event spawn` command creates an eventpoint that watches for the creation of a thread. The `run` command begins process execution. Execution stops when thread 1 is spawned from the process, triggering the eventpoint.

CXdb opens a second source window corresponding to thread 1, as shown in Figure 376. This source window tracks the execution of thread 1. The source window created when CXdb was invoked tracks thread 0.

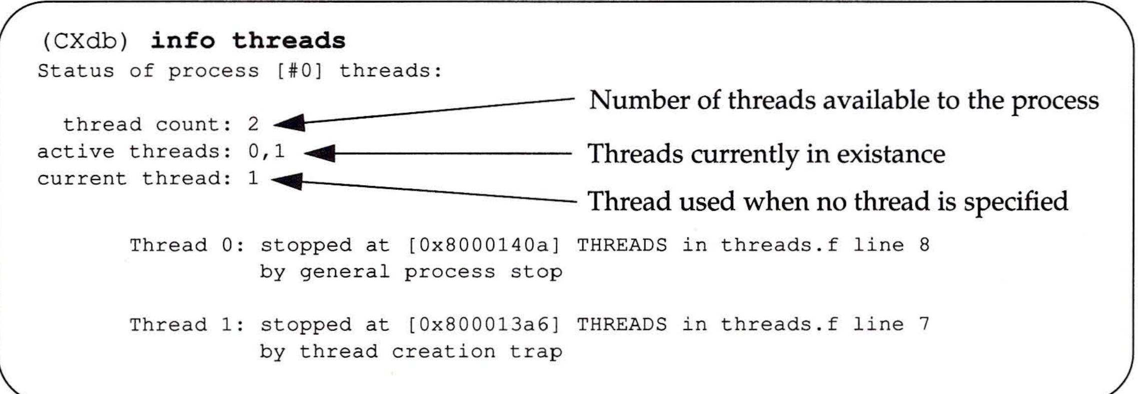
Figure 376
Source windows for threads 0 and 1 in CXwindows



Getting information about threads

You can get information on all threads using the `info threads` command. The status for all active threads is displayed, as well as a summary for all threads of the process. The maximum number of threads that can be created by the process is also displayed. Note that this is not necessarily the actual number of threads that will be created. The `info thread` command is shown in Figure 377.

Figure 377
Getting information about the threads of a process



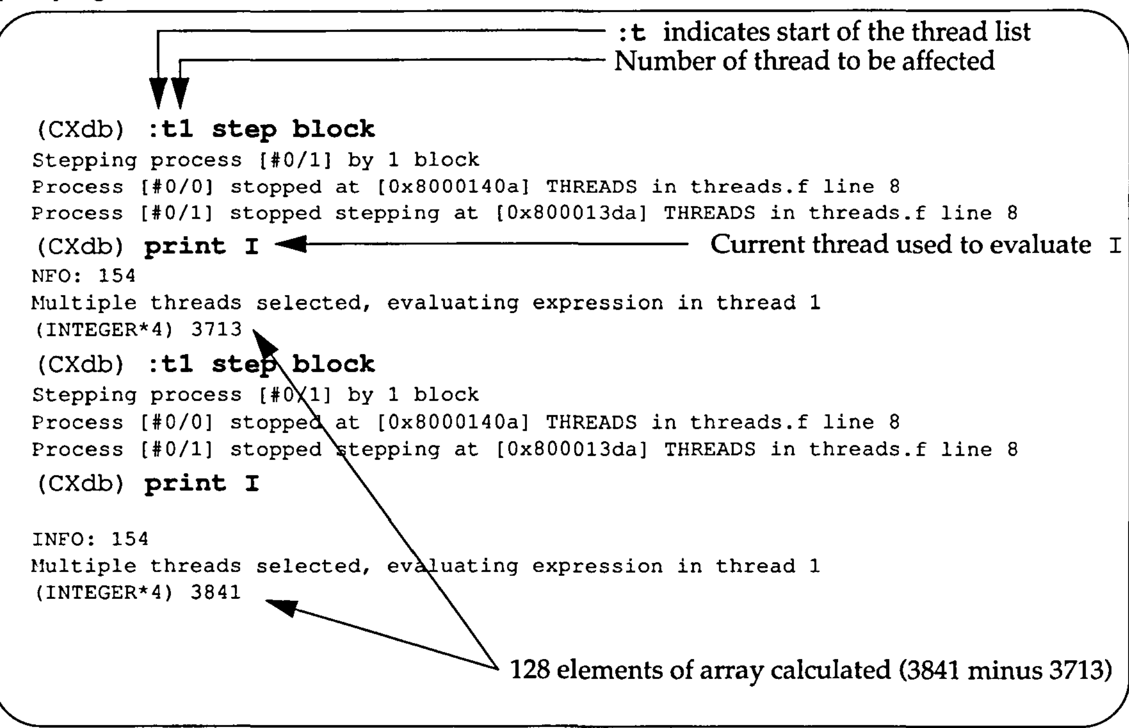
In Figure 377, the `info threads` command displays the status of all threads of the process. The thread count field shows how many threads are available to the process. The active threads field lists the numbers of all threads in existence. In this case, both thread 1 and thread 0 exist. The current thread is the thread used by commands that are thread-specific, such as the `print` command, when a thread is not specified. Thread 1 is the current thread. Finally, the status for all existing threads is shown. Thread 0 is stopped at line 8 of the program and thread 1 is stopped at line 7.

Making commands thread-specific

You can specify a particular thread in many CXdb commands. When you do so, the command affects only the specified thread. Examples of commands that can be made thread-specific are process-execution commands, eventpoint commands, and the `print` command.

A thread list is used to specify the threads a command affects. A thread list appears before the command name itself. Figure 378 shows a thread list being used to step through thread 1 of the process.

Figure 378
Specifying a thread list with the `step` command



The `:t1 step block` command in Figure 378 steps thread 1 by one block. CXdb responds with a message indicating that thread 1 stopped stepping, while thread 0 remains stopped.

The `print` command prints the value of the variable `I` from the current thread. The current thread is generally the last thread that stopped executing. In this case, the value of `I` is 3713.

Note

The values displayed by `print` commands in this section may differ greatly from the values displayed on your screen, due to the non-deterministic nature of the distribution of chores to the threads.

Just as thread 1 calculated an iteration of the strip-mined loop, thread 0 is about to calculate an iteration, as shown in Figure 379.

Figure 379
Building of the array by the two threads

```
(CXdb) :t0 step block
Stepping process [#0/0] by 1 block
Process [#0/0] stopped stepping at [0x800013da] THREADS in threads.f line 8
Process [#0/1] stopped at [0x800013da] THREADS in threads.f line 8
(CXdb) :t0 print I
(INTEGER*4) 19713
(CXdb) :t1 print I
(INTEGER*4) 19585
(CXdb) :t0 step block
Stepping process [#0/0] by 1 block
Process [#0/0] stopped stepping at [0x800013da] THREADS in threads.f line 8
Process [#0/1] stopped at [0x800013da] THREADS in threads.f line 8
(CXdb) :t0 print I
(INTEGER*4) 19841
```

In Figure 379, the `:t0 step block` command steps thread 0 by one block.

The next two `print I` commands print the value of `I` in the two threads. Note that the value of `I` is distinct between the two threads. This is because the synthesized variables, from which `I` is calculated, are stored in thread-specific memory rather than global memory. However, array `A` still remains in global memory. Thus, both threads affect different elements of the same array.

The last `step block` command steps thread 0 by another block. The value of `I` is again printed in the last `print` command.

Eventpoints that can be placed at particular locations in the code can also be placed on particular threads. Normally, a breakpoint is set on all threads of the process. However, you can set a breakpoint for a specific thread by specifying a thread list in front of the `break` command. Figure 380 gives an example of setting a breakpoint on a specific thread.

Figure 380
Setting a breakpoint on a particular thread

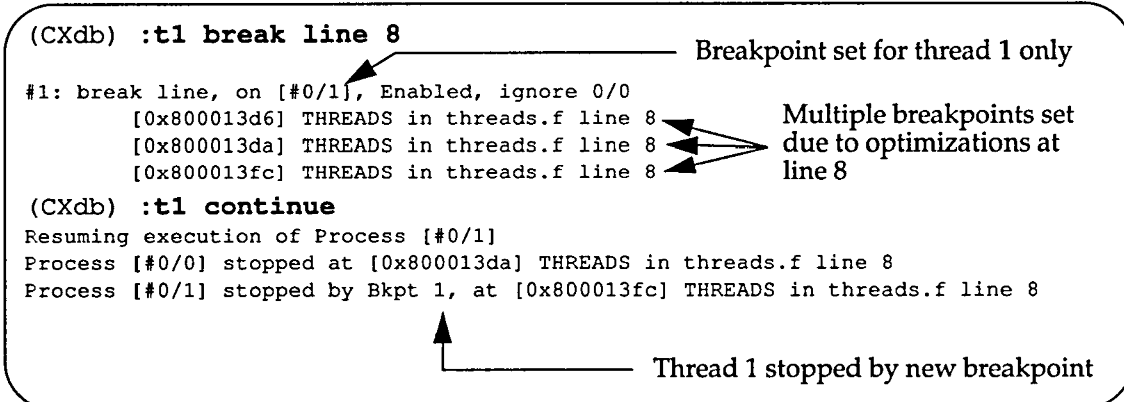
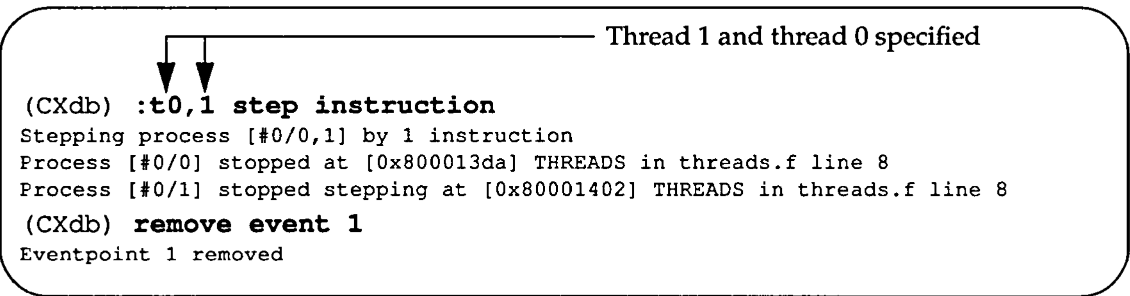


Figure 380 uses the `break line` command to set a breakpoint on thread 1 of the process. The breakpoint is placed at multiple instructions due to optimizations on line 8.

The `continue` command continues execution of thread 1. Thread 1 is stopped by the breakpoint.

You can also specify multiple thread numbers in a thread list. When you do so, all threads listed are affected by the command. Most CXdb commands act on all threads of the process by default. Figure 381 illustrates the `set format` command being used with a multiple thread list.

Figure 381
Using a multiple thread-list



The `:t0,1 step instruction` command in Figure 381 steps both threads by an instruction. Note that only one thread, thread 1, actually stepped. This is because one thread typically completes execution before the other thread begins execution. CXdb stops all threads of a process when it detects one thread has completed execution. The `remove event` command removes the breakpoint.

Sending a signal to a thread

A signal can be sent to a particular thread using the `signal thread` command. The signal is sent to the specified thread, and all threads resume execution. Because the operating system sends the signal to a single thread, only one thread can be specified with the `signal thread` command.

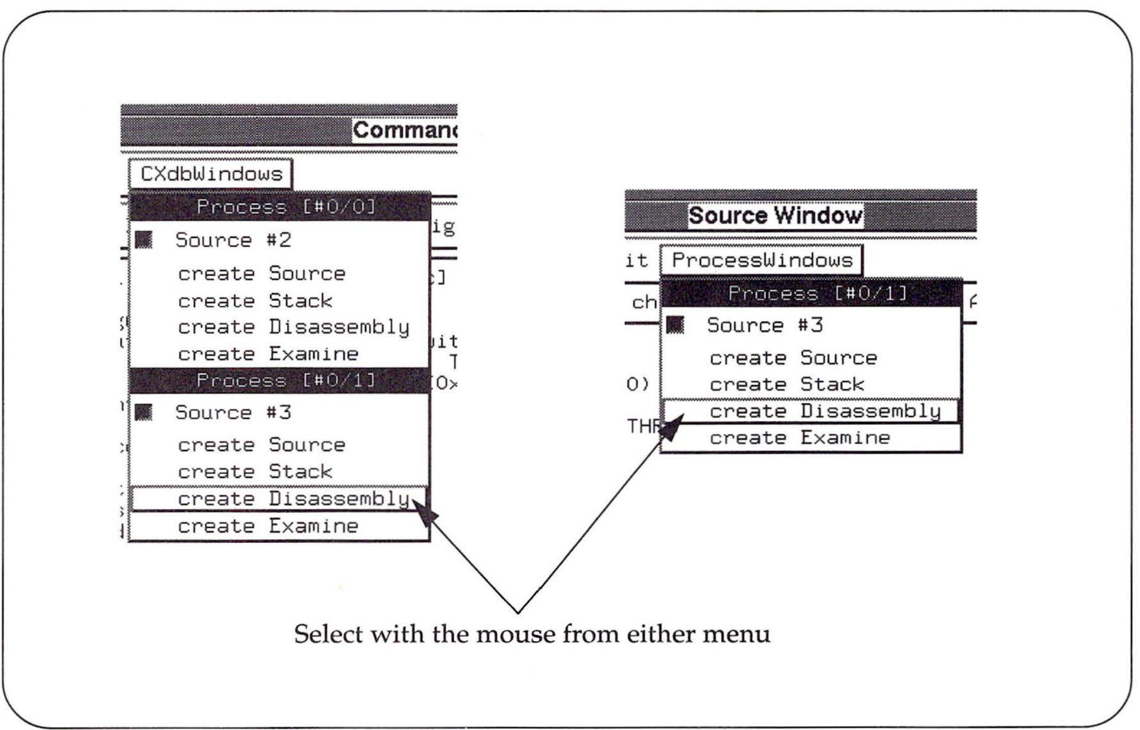
If you use the `signal process` command to send a signal to the process, the operating system determines which thread catches the signal.

Opening thread-specific CXwindows

In the CXwindows interface, each thread of a process can have its own set of process windows. However, only one process interface window exists for a multithreaded process. You can open a disassembly window, stack window, or examine window for each thread of the process.

All of the functionality available in these windows is available for each thread. Figure 382 demonstrates how to open a disassembly window for thread 1.

Figure 382
Opening a disassembly window for thread 1



You can open a disassembly window using either the ProcessWindows menu from thread 1's source window or from the CXdbWindows menu from the command window. The disassembly window opened tracks the execution of thread 1, rather than thread 0.

Join eventpoints

The `event join` command sets an eventpoint that triggers when a thread joins (executes a `join`, `cfork` or `wfork` instruction). A thread joins just before it terminates. A join eventpoint is the opposite of a spawn eventpoint (which detects thread creation).

Just before the thread terminates, the eventpoint is triggered, process execution stops, and CXdb displays a message indicating which thread is joining. The last thread that joins does not terminate, but rather continues process execution.

Note

The source window that tracks the thread executing the remaining portion of the program may not be the original source window. The source window tracking the last thread that joined continues to track the thread when the process is single threaded.

Figure 383 shows the `event join` command being used to stop execution before thread 1 executes a `join` instruction.

Figure 383
Setting a join eventpoint

```
(CXdb) event join

#2: join, on [#0], Enabled, ignore 0/0
(CXdb) :t0 continue
Resuming execution of Process [#0/0]
Process [#0/0] thread joining
Process [#0/0] stopped by Eventpoint 2, at [0x80001424] THREADS in threads.f line 7
Process [#0/1] stopped at [0x80001402] THREADS in threads.f line 8
(CXdb) info threads
Status of process [#0] threads:

  thread count: 2
active threads: 0,1
current thread: 0


       Thread 0: exiting at [0x80001424] THREADS in threads.f line 7

       Thread 1: stopped at [0x80001402] THREADS in threads.f line 8
                 by general process trap
(CXdb) break line 11
#3: break line, on [#0/*], Enabled, ignore 0/0
        [0x8000143e] THREADS in threads.f line 11
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/1] stopped by Bkpt 3, at [0x8000143e] THREADS in threads.f line 11
```

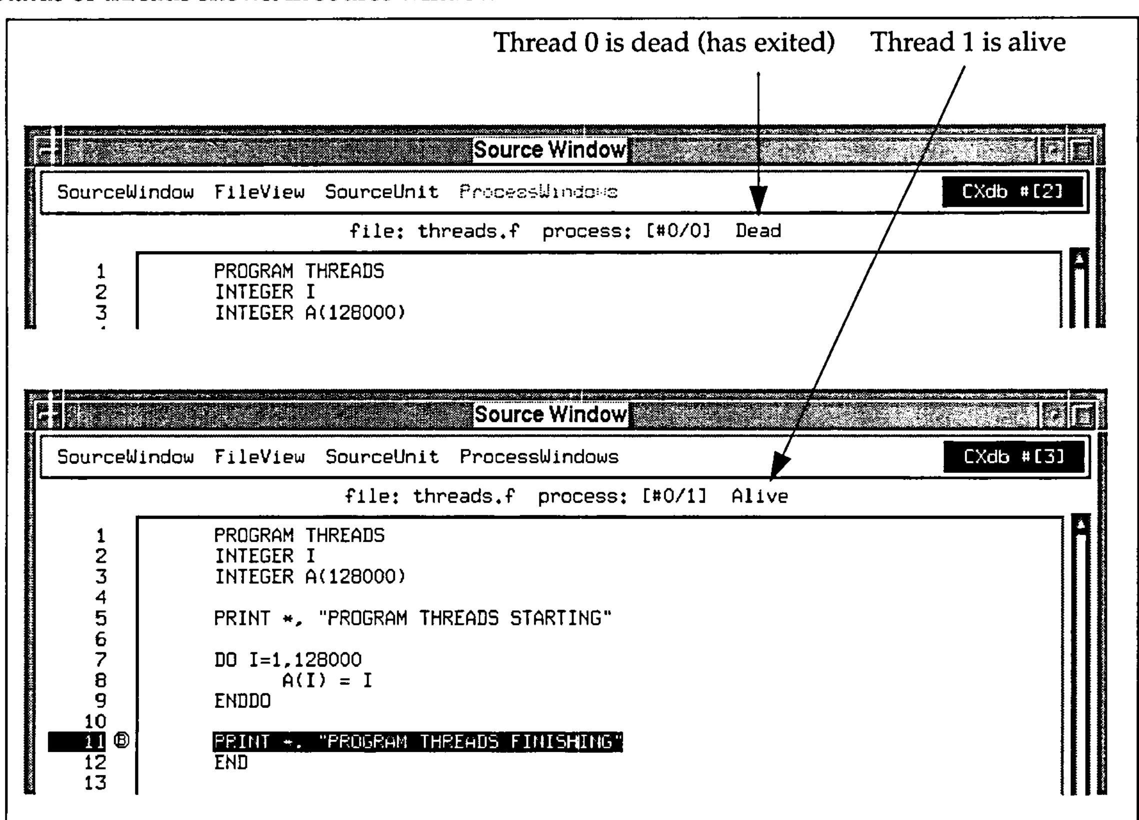
In Figure 383, the `event join` command creates a join eventpoint to watch for a thread to execute a `join` instruction.

The `:t0 continue` command continues execution of thread 0. Thread 0 completes its set of instructions and executes a `join` instruction before exiting. The join eventpoint is triggered, and execution is stopped.

The `info threads` command displays the current status of the threads. From the output of the `info threads` command, it can be seen that thread 0 is exiting. The `break line 11` command sets a breakpoint at line 11. The `continue` command resumes process execution. Execution stops when the breakpoint at line 11 is triggered by thread 1.

The source window for thread 1 remains the active window, as shown in Figure 384.

Figure 384
Status of threads shown in source window



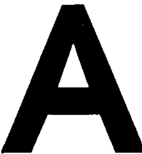
Quitting the examples

If you have been working the examples in this chapter, you can quit them and exit CXdb by issuing the `quit` command, as shown in Figure 385.

Figure 385
Quitting the examples

```
(CXdb) quit
Process [#0] is still running.  Kill it? y
```

Source code for example program



This appendix contains a listing of the source code used in the examples for this book. The code is listed by file name.

The source code contains routines written in FORTRAN and C. These routines are not necessarily intended to be examples of good programming. Rather, they are designed to illustrate specific features of CXdb.

For some of the examples in this book, the same routine is presented in both C and FORTRAN to illustrate the differences in how CXdb handles these two languages. In such cases, the C code is written to be as similar to the FORTRAN code as possible.

Source file: example.f

```
1         PROGRAM EXAMPLE
2         INTEGER ARRAY(4,4)
3         INTEGER I, NUMARGS
4         EXTERNAL chapter7c
5         EXTERNAL chapter13c
6
7         PRINT *, "EXAMPLE PROGRAM STARTED"
8
9         DO I=1,4
10           DO J=1,4
11              ARRAY(I,J)=I*J
12           ENDDO
13        ENDDO
14
15        NUMARGS = IARGC()
16
17        IF (NUMARGS .EQ. 0) THEN
18           CALL CHAPTER4(ARRAY)
19           CALL CHAPTER5(ARRAY)
20           CALL CHAPTER6(ARRAY)
21           CALL CHAPTER7(ARRAY)
22           CALL chapter7c(ARRAY)
23           CALL CHAPTER8(ARRAY)
24           CALL CHAPTER13F(ARRAY)
25           CALL chapter13c(ARRAY)
26           CALL CHAPTER15
27        ELSE
28           CALL EXAMPLE_INPUT
29        ENDIF
30
31        PRINT *, "EXAMPLE PROGRAM FINISHED"
32        END
33
34
```

(Continued on next page.)

```
35        SUBROUTINE PRINT_ARRAY(ARRAY)
36        INTEGER ARRAY(4,4)
37        INTEGER I
38
39        PRINT *
40        PRINT *, "THE TABLE:"
41        DO I=1,4
42             PRINT 99, ARRAY(I,1), ARRAY(I,2), ARRAY(I,3), ARRAY(I,4)
43        ENDDO
44
45        PRINT *
46  99    FORMAT (3X,I2,X,I2,X,I2,X,I2)
47        END
48
49
50        SUBROUTINE CLEAR_ARRAY(ARRAY)
51        INTEGER ARRAY(4,4)
52
53        DO I=1,4
54          DO J=1,4
55            ARRAY(I,J)=0
56          ENDDO
57        ENDDO
58        END
59
60
61
62        SUBROUTINE EXAMPLE_INPUT
63        CHARACTER*30 NAME,ARG
64
65        PRINT *, "The process interface window handles program input."
66        PRINT *, "As an example, enter your first name and press return."
67        READ *, NAME
68
69        CALL GETARG(1,ARG)
70        PRINT 99, "Thanks, ", NAME
71        PRINT 99, "The command line argument was: ", ARG
72
73  99    FORMAT (2A)
74        END
```

Source file: chapter4.f

```
 1         SUBROUTINE CHAPTER4(ARRAY)
 2         INTEGER ARRAY(4,4)
 3         INTEGER I, J, PICK
 4         INTEGER*4 SEED
 5
 6         PRINT *, "SUBROUTINE CHAPTER4 STARTING"
 7
 8         DO I=1,4
 9            ARRAY(I,I) = 0
10            CALL PRINT_ARRAY(ARRAY)
11         ENDDO
12   c
13   c        COMMENT LINE - THIS LINE HAS NO SOURCE UNITS
14   c
15         DO I=1,4
16            DO J=1,4
17               ARRAY(I,J) = 1
18            ENDDO
19         ENDDO
20
21         SEED = 1
22         PICK = RAN(SEED) * 4 + 1
23         ARRAY(PICK,PICK) = 99
24
25         DO I=1,16
26            PICK = PICK + 4096
27         ENDDO
28
29         DO I=1,16
30            PICK = PICK + 65536
31         ENDDO
32
33         PICK = PICK + 1
34
35         PRINT *, "SUBROUTINE CHAPTER4 FINISHING"
36         END
```

Source file: chapter5.f

```
1          SUBROUTINE CHAPTER5(ARRAY)
2          INTEGER ARRAY(4,4)
3
4          PRINT *, "SUBROUTINE CHAPTER5 STARTING"
5          DO I = 1, 10
6            PRINT 99, "I = ", I
7            CALL SUB5A(I)
8            PRINT *, "Subroutine SUB5A has returned."
9          ENDDO
10         PRINT *, "The loop for M is next."
11         DO J = 1, 4
12           DO M = 1, 4
13             ARRAY(J,M) = J**M
14             PRINT 99, "J = ", J, "M = ", M, "ARRAY(J,M) = ", ARRAY(J,M)
15           ENDDO
16         ENDDO
17      99 FORMAT (A,I2,3X,A,I2,5X,A,I4)
18         PRINT *, "SUBROUTINE CHAPTER5 FINISHING"
19         END
20
21         SUBROUTINE SUB5A(N)
22         PRINT 98, "Subroutine SUB5A has started. The value of N is ", N
23         DO K = 1, N
24           PRINT 98, "K = ", K
25           IF (K .LE. 5) THEN
26             DO L = 1, N
27               PRINT 98, "L = ", L
28             ENDDO
29             PRINT 98, "The loop for L is done, with L = ", L
30           ENDIF
31         ENDDO
32         PRINT 98, "Subroutine SUB5A is done. The value of K is ", K
33         RETURN
34      98 FORMAT (A,I2)
35         END
```

Source file: chapter6.f

```
1         SUBROUTINE CHAPTER6(ARRAY)
2         INTEGER ARRAY(4,4),I,J,NEXT,KILL,GETPID,PID,SIG,TEST
3         REAL FRACTION,SQUARE,A,B
4
5         PRINT *, "SUBROUTINE CHAPTER6 STARTING"
6
7         DO I=1,4
8           IF ((I .EQ. 1) .OR. (I .EQ. 2)) THEN
9              DO J=1,4
10                PRINT 99, "LOOP1 ITERATION: ", J
11             ENDDO
12          ELSE
13            DO J=1,4
14                PRINT 99, "LOOP2 ITERATION: ", J
15            ENDDO
16          ENDIF
17        ENDDO
18
19        DO A=4,1000
20           B = 2
21           SQUARE = A/B
22           DO WHILE (SQUARE .GE. B)
23               IF (SQUARE .EQ. B) THEN
24                  PRINT 98, A, " IS A PERFECT SQUARE"
25               ENDIF
26               B=B+1
27               SQUARE = A/B
28           ENDDO
29        ENDDO
30
```

(Continued on next page.)

```
31          NEXT = 0
32          IF (NEXT .EQ. 1) THEN
33              DO I=1,4
34                 DO J=1,4
35                    PRINT 97, "I IS:", I, "J IS:", J
36                    TEST = ARRAY(J,I) - ARRAY(I,J)
37                 ENDDO
38              ENDDO
39
40  c
41  c           NEXT SEVERAL LINES SEND SIGNALS TO THIS PROCESS
42  c
43              PID = GETPID()
44              SIG = KILL(PID,2)
45              SIG = KILL(PID,2)
46          ENDIF
47
48          PRINT *, "SUBROUTINE CHAPTER6 FINISHING"
49  97      FORMAT (A,I2,2X,A,I2)
50  98      FORMAT (F5.1,A)
51  99      FORMAT (A,I2)
52          END
```

Source file: chapter7F.f

```
1          SUBROUTINE CHAPTER7(ARRAY)
2          INTEGER ARRAY(4,4), TABLE(4,4)
3
4          PRINT *, "SUBROUTINE CHAPTER7 STARTING"
5          DO I = 1, 4
6            DO J = 1, 4
7              TABLE(I,J) = ARRAY(I,J) + I*ISQR(J)
8            ENDDO
9          ENDDO
10         CALL BLD_MATRIX(4,4,5,TABLE)
11         PRINT *, "SUBROUTINE CHAPTER7 FINISHING"
12         RETURN
13         END
14
15
16         FUNCTION ISQR(N)
17         ISQR = N*N
18         RETURN
19         END
20
21
```

(Continued on next page.)

```
22        SUBROUTINE BLD_MATRIX(N,M,L,SLICE)
23        INTEGER SLICE(N,M)
24        REAL MATRIX(5,5,5)
25
26        DO I = 1, 5
27          DO J = 1, 5
28            DO K = 1, 5
29              MATRIX(I,J,K) = 0
30            ENDDO
31          ENDDO
32        ENDDO
33        IF (N .GT. 5) THEN N=5
34        IF (M .GT. 5) THEN M=5
35        IF (L .GT. 5) THEN L=5
36        DO I = 1, N
37          DO J = 1, M
38            DO K = 1, L
39              MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
40              PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
41            ENDDO
42            PRINT *
43          ENDDO
44          PRINT *
45        ENDDO
46     99 FORMAT (A, I2, 3X, A, I2, 3X, A, I2, 5X, A, F8.3)
47        RETURN
48        END
```

Source file: chapter7C.c

```
 1  #include <stdio.h>
 2  void chapter7c_(array)
 3  int array[4][4];
 4  {
 5     static int i, j, table[4][4];
 6
 7     printf("The C subroutine chapter7c is starting.\n");
 8     for (i=0; i<4; i++)
 9     {
10        for (j=0; j<4; j++)
11        {
12           table[i][j] = array[i][j] + (i+1)*isqr(j+1);
13        }
14     }
15     bld_matrix(4,4,5,table);
16     printf("Subroutine chapter7c is finished.\n");
17  }
18
19
20  int isqr(n)
21  int n;
22  {
23     return n*n;
24  }
25
26
```

(Continued on next page.)

```
27  void bld_matrix(n,m,l,slice)
28  int n, m, l, slice[5][5];
29  {
30     int i,j,k;
31     float matrix[5][5][5];
32
33     for (i=0; i<5; i++)
34     {
35        for (j=0; j<5; j++)
36        {
37           for (k=0; k<5; k++)
38           {
39              matrix[i][j][k] = 0;
40           }
41        }
42     }
43     if (n > 5) n=5;
44     if (m > 5) m=5;
45     if (l > 5) l=5;
46     for (i=0; i<n; i++)
47     {
48        for (j=0; j<m; j++)
49        {
50           for (k=0; k<l; k++)
51           {
52              matrix[i][j][k] = 3*(k+1)/7.5 - 29.7/(i+j+k+3) + slice[i][j];
53              printf("i= %d    j= %d    k= %d     matrix[i,j,k]= %8.3f\n",
54                     i, j, k, matrix[i][j][k]);
55           }
56           printf("\n");
57        }
58        printf("\n");
59     }
60  }
```

Source file: chapter8.f

```
1           SUBROUTINE CHAPTER8(ARRAY)
2           INTEGER ARRAY(4,4)
3           INTEGER FUNCTION getcwd
4           CHARACTER*40 DIRNAME,EVALUE, ENVVAR(3)
5
6           PRINT *, "SUBROUTINE CHAPTER8 STARTING"
7           CALL getcwd(DIRNAME)
8           PRINT *, "THE CURRENT DIRECTORY IS: ", DIRNAME
9
10          ENVVAR(1) = "CHAPTER"
11          ENVVAR(2) = "DEBUGGER"
12          ENVVAR(3) = "FLAGS"
13          DO I=1,3
14             CALL getenv(ENVVAR(I),EVALUE)
15             IF (EVALUE .EQ. "") THEN
16                PRINT 98, "COULDN'T FIND ENVIRONMENT VARIABLE: ", ENVVAR(I)
17             ELSE
18                PRINT 99, ENVVAR(I),"IS SET TO:", EVALUE
19             ENDIF
20          ENDDO
21
22          DO I = 1, 10
23             VALUE = VALUE + 3 * (I * I + VALUE)
24          ENDDO
25
26          PRINT *, "SUBROUTINE CHAPTER8 FINISHING"
27
28   98     FORMAT(A,A8)
29   99     FORMAT(A8,X,A,X,A)
30          END
```

Source file: chapter13F.f

```
1          SUBROUTINE CHAPTER13F(ARRAY)
2          INTEGER ARRAY(4,4), I
3          CHARACTER*12 INFO_BLOCK
4          COMMON /BLK/ INFO_BLOCK
5
6          PRINT *, "SUBROUTINE CHAPTER13F BEGINNING"
7
8          CALL SUB13D
9          OPEN (7, FILE="data")
10         PRINT *, "TEST NAME     TEST-1   TEST-2   TEST-3   AVERAGE"
11         PRINT *, "---------     ------   ------   ------   -------"
12         DO I=1,3
13            CALL SUB13A
14            CALL SUB13B
15            CALL SUB13C
16         ENDDO
17         CLOSE (7)
18
19         PRINT *
20         PRINT *, "SUBROUTINE CHAPTER13F FINISHING"
21         PRINT *
22
23         END
24
25
26
27
28        SUBROUTINE SUB13A
29        CHARACTER*12 INFO
30        COMMON /BLK/ INFO
31
32        READ (7,98) INFO
33
34  98    FORMAT(A)
35        END
36
37
38
```

(Continued on next page.)

```
39          SUBROUTINE SUB13B
40          CHARACTER*6 NAME
41          INTEGER*1 NUM(6), I
42          REAL AVG
43          COMMON /BLK/ NAME, NUM, AVG
44
45          SUM = 0
46          DO I=1,6,2
47              SUM = SUM + (NUM(I)-48)*10 + (NUM(I+1) - 48)
48          ENDDO
49
50          AVG = SUM / 3
51
52          END
53
54
55
56          SUBROUTINE SUB13C
57          INTEGER I
58          CHARACTER*6 NAME
59          CHARACTER*2 RESULTS(3)
60          REAL AVG
61          COMMON /BLK/ NAME, RESULTS, AVG
62
63          I = -1
64          PRINT 99, NAME,RESULTS(1),RESULTS(2),RESULTS(3),AVG
65
66   99     FORMAT (X,A,8X,A,7X,A,7X,A,6X,F5.2)
67          END
68
69
70
71          OPTIONS -re
72          SUBROUTINE SUB13D
73          INTEGER I
74
75   c      This routine is compiled -re, so its local variable
76   c      is allocated on the stack, rather than being in  global memory
77
78          I = 40
79
80          END
```

Source file: chapter13C.c

```
1   #include <stdio.h>
2   extern void ansi_block() ;
3   extern void pcc_block() ;
4
5   struct info_block {
6      char subject[7] ;
7      int  test[3] ;
8      float avg ;
9   } ;
10
11  void chapter13c_(array)
12  int array[4][4];
13  {
14     struct info_block data ;
15     FILE *fopen(), *fp ;
16     static int i ;
17
18     printf("\nThe C subroutine chapter13c is starting.\n\n");
19     printf("TEST NAME     TEST-1   TEST-2   TEST-3   AVERAGE\n");
20     printf("---------     ------   ------   ------   -------\n");
21
22     fp = fopen("data", "r") ;
23     for(i=1;i<4;i++)
24     {
25        sub13a(&data,fp) ;
26        sub13b(&data) ;
27        sub13c(&data) ;
28     }
29     close(fp) ;
30
31     ansi_block(0) ;
32     pcc_block(0) ;
33
34     printf("\nThe C subroutine chapter13c is finishing.\n\n");
35  }
36
37
38
```

(Continued on next page.)

```
39  sub13a(info,fp)
40  struct info_block *info ;
41  FILE *fp ;
42  {
43      static int i,c ;
44
45      for(i=0;i<6;i++)
46      {
47        c = getc(fp) ;
48        info->subject[i] = (char)c ;
49      }
50
51      for(i=0;i<3;i++)
52      {
53        c = getc(fp) ;
54        info->test[i] = (c - 48) * 10 ;
55        c = getc(fp) ;
56        info->test[i] += (c - 48) ;
57      }
58      c = getc(fp) ;
59  }
60
61
62
63  sub13b(info)
64  struct info_block *info ;
65  {
66     static int i, sum ;
67
68     sum = 0 ;
69     for (i=0; i<3; i++)
70       sum += info->test[i] ;
71
72     info->avg = sum / 3.0 ;
73  }
74
75
76
77  sub13c(info)
78  struct info_block *info ;
79  {
80    int i= -1 ;
81
82    printf(" %s          %d         %d         %d        %5.2f\n",&info->subject,
          info->test[0],info->test[1],info->test[2],info->avg) ;
83  }
```

Source file: chapter15.f

```
1         SUBROUTINE CHAPTER15
2         PARAMETER (M=1000, N=10, X=1.3)
3         REAL A(M,N), B(M,N)
4
5         PRINT *, "SUBROUTINE CHAPTER15 STARTING"
6         CALL LEVEL_NO(M,N,A,B,X)
7         CALL INIT(M,N,A,B)
8         CALL LEVEL_O0(M,N,A,B,X)
9         CALL INIT(M,N,A,B)
10        CALL LEVEL_O1(M,N,A,B,X)
11        CALL INIT(M,N,A,B)
12        CALL LEVEL_O2(M,N,A,B,X)
13        PRINT *, "SUBROUTINE CHAPTER15 FINISHING"
14
15        END
16
17
18        SUBROUTINE INIT(M,N,A,B)
19        REAL A(M,N), B(M,N)
20
21        DO I=1,M
22           DO J=1,N
23              A(I,J) = 0.0
24              B(I,J) = I + J
25           ENDDO
26        ENDDO
27
28        RETURN
29        END
30
31
32
33        SUBROUTINE LEVEL_NO(M,N,A,B,X)
34        REAL A(M,N), B(M,N)
35
36        DO J=1,N
37           DO I=1,M
38              TEMP = 3.0 * B(I,J)
39              A(I,J) = TEMP/(2.0*X)
40              B(I,J) = 2.0 * TEMP
41           ENDDO
42        ENDDO
43
44        RETURN
45        END
```

```
46
47
48
49          OPTIONS -O0
50          SUBROUTINE LEVEL_O0(M,N,A,B,X)
51          REAL A(M,N), B(M,N)
52
53          DO J=1,N
54             DO I=1,M
55                TEMP = 3.0 * B(I,J)
56                A(I,J) = TEMP/(2.0*X)
57                B(I,J) = 2.0 * TEMP
58             ENDDO
59          ENDDO
60
61          RETURN
62          END
63
64
65
66          OPTIONS -O1
67          SUBROUTINE LEVEL_O1(M,N,A,B,X)
68          REAL A(M,N), B(M,N)
69
70          DO J=1,N
71             DO I=1,M
72                TEMP = 3.0 * B(I,J)
73                A(I,J) = TEMP/(2.0*X)
74                B(I,J) = 2.0 * TEMP
75             ENDDO
76          ENDDO
77
78          RETURN
79          END
80
81
82
```

(Continued on next page.)

```
83        OPTIONS -O2
84        SUBROUTINE LEVEL_O2(M,N,A,B,X)
85        REAL A(M,N), B(M,N)
86
87        DO J=1,N
88           DO I=1,M
89              TEMP = 3.0 * B(I,J)
90              A(I,J) = TEMP/(2.0*X)
91              B(I,J) = 2.0 * TEMP
92           ENDDO
93        ENDDO
94
95        RETURN
96        END
```

Source file: ansi_block.c

```
1   ansi_block(i)
2   int i;
3   {
4      i = 0 ;
5
6      if (i == 0)
7      {
8         int i=1 ;
9
10        if (i == 1)
11        {
12            int i=11 ;
13        }
14
15        if (i == 1)
16        {
17            int i=12 ;
18        }
19     }
20     printf("\nThe routine ansi_block is finishing.\n");
21  }
```

Source file: pcc_block.c

```
1   pcc_block(i)
2   int i;
3   {
4      int i = 1 ;
5
6      if (i == 1)
7      {
8         int i=11 ;
9
10        if (i == 11)
11        {
12            int i=111 ;
13        }
14
15        if (i == 11)
16        {
17            int i=112 ;
18        }
19     }
20     printf("\nThe routine pcc_block is finishing.\n");
21  }
```

Quick reference to Maryland Windows

B

This appendix contains a quick reference to the keyboard functions used in Maryland Windows. These keyboard functions manipulate the windows and edit the command line.

The functions are divided into four groups:

- **Text Scrolling Functions**—These functions scroll the text inside a window. You can scroll a line or a screen at a time, or jump to the beginning or end of the buffer.
- **Window Functions**—These functions move between windows, close windows, raise and lower windows, or move and resize windows.
- **Window Positioning Functions**—These functions are used when placing a window or resizing a window.
- **Command Window Functions**—These functions are unique to the command window. They provide a means of scrolling the text in the command window or the command history list. There are also functions to edit the command line.

In the following tables, words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-x** indicates that you must press and hold down the **CTRL** key and then press the **x** key.

For more information on the Maryland Window keyboard functions, including how to change the default key bindings listed here, refer to the Section, "Maryland Windows," in the *CONVEX CXdb Reference.*

Text Scrolling Functions

Function	Key Sequence
left character	**LEFT ARROW**, **CTRL-b**
right character	**RIGHT ARROW**, **CTRL-f**
left word	**META-b**
right word	**META-f**
down line	**DOWN ARROW**, **CTRL-n** (except command window)
up line	**UP ARROW**, **CTRL-p** (except command window)
down screen	**CTRL-v**
up screen	**META-v**
top of buffer	**META-<**
bottom of buffer	**META->**

Window Functions

Function	Key Sequence
next window	**META-n**
previous window	**META-p**
move window	**META-m**
resize window	**META-z**
lower window	**META-o**
raise window	**META-r**
close window	**META-k** (except commmand window)

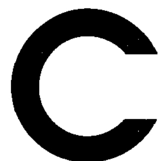
Window Positioning Functions

Function	Key Sequence
up	**k**, **CTRL-p**
down	**j**, **CTRL-n**
right	**l**, **META-f**
left	**h**, **META-b**
anchor window	**SPACEBAR**

Command Window Functions

Function	Key Sequence
down line	META-A
up line	META-B
beginning of line	CTRL-a
end of line	CTRL-e
down history	CTRL-n
up history	CTRL-p
delete character	DELETE, CTRL-d
delete back character	BACKSPACE, CTRL-h
transpose characters	CTRL-t
delete word	META-DELETE, META-d
delete back word	META-BACKSPACE, META-h
set mark	CTRL-@
exchange point and mark	CTRL-x
kill to end ofline	CTRL-k
kill region	CTRL-w
copy region	META-w
yank	CTRL-y
startover	CTRL-g
redrawn screen	CTRL-l, CTRL-r
lowercase	META-l
uppercase	META-u
capitalize	META-c
universal argument	CTRL-u
argument digit	META-0...META-9
newline	RETURN, LINEFEED, CTRL-j, CTRL-m

Cross-reference to csd commands



This appendix contains a cross-reference between the commands for the CONVEX csd debugger and CXdb commands. In cases where there is not a one-to-one correspondence between commands, macros are given if possible.

For the most frequently used csd commands, aliases have been provided in CXdb. The alias name in CXdb is the same as the csd command itself. Those csd commands that have aliases are indicated on the following pages by a plus sign (+) in front of the commands.

The csd aliases are defined in the command file /usr/lib/cxdb/csd_aliases. This command files does *not* execute automatically when you invoke CXdb. If you want to use the aliases, first execute the command file with the source command, as illustrated in Figure 386.

Figure 386
Incorporating the csd aliases

```
(CXdb) source /usr/lib/cxdb/csd_aliases
```

You can include the above source command in your .cxdbinit file so that the aliases are incorporated automatically each time you invoke CXdb.

csd	CXdb
/*<string>*/	---
//	---
?*<string>*?	---
??	---
.,*<count>*?*<mode>*	examine/*<cxdb-mode>* $pc:*<count>*
<address>,*<count>*?*<mode>*	examine/*<cxdb-mode>* *<address>*:*<count>*
+ &*<name>*	print &*<name>*
alias	info alias
alias *<name>*	info alias *<name>*
alias *<name>* *<string>*	alias *<name>* *<string>*
alias *<name>*(*params*) *<string>*	Use macro
+ assign *<var>* = *<exp>*	evaluate *<var>* = *<exp>*
	print *<var>* = *<exp>*
+ call *<routine>*(*<params>*)	eval *<routine>*(*<params>*)
	print *<routine>*(*<params>*)
catch	info signal
catch *<signal>*	set signal *<signal>*; stop; print
cont	:t*<num>* continue
cont *<signal>*	$signal = *<signal>*; :t*<num>* continue
cont all	continue
cont all *<signal>*	$signal = *<signal>*; continue
cregs	info cregisters
cregs *<index>*	print $C*<index>*
+ delete *<index>*	remove event *<index>*
delete all	remove event *
down	frame -1
down *<num>*	frame -*<num>*
+ dump	backtrace
edit	Use edit. File name displayed in source window
edit *<filename>*	edit *<filename>*
edit *<routine>*	---
file	Displayed in source window
+ file *<filename>*	Can be changed with display routine
format	info cxdb
+ format hex	set format byte hexadecimal;
	set format halfword hexadecimal;
	set format word hexadecimal;
	set format longword hexadecimal;
	set format quadword hexadecimal;
+ format decimal	set format byte decimal;
	set format halfword decimal;
	set format word decimal;
	set format longword decimal;
	set format quadword decimal;
fpmode	info cxdb
+ fpmode native	set fpmode native
+ fpmode ieee	set fpmode ieee
+ fpmode auto	set default fpmode dual
func	info scope
func *<routine>*	display routine *<routine>*

csd	CXdb
help	help
help <*command*>	help <*command*>
ignore	info signal
ignore <*signal*>	set signal <*signal*> nostop,nopass,noprint
list	Scroll source window
list <*first,last*>	Scroll source window
+ list <*routine*>	display routine <*routine*>
+ list eventlines <*first,last*>	---
+ mode	info process
+ mode chained	clear seq
+ mode sequential	set seq
next	next
	next expression
	next statement
	next loop
	next block
	next routine
next <*count*>	next <*count*>
	next expression <*count*>
	next statement <*count*>
	next loop <*count*>
	next block <*count*>
	next routine <*count*>
+ next all	next
	next expression
	next statement
	next loop
	next block
	next routine
+ next all <*count*>	next <*count*>
	next expression <*count*>
	next statement <*count*>
	next loop <*count*>
	next block <*count*>
	next routine <*count*>
+ nexti	next instruction
nexti <*count*>	next instruction <*count*>
nexti all	next instruction
nexti all <*count*>	next instruction <*count*>
print <*expr*>	print <*expr*>
print <*expr*>,<*expr*>,...	---
print <*expr*> \ <*type*>	print (<*type*>) <*expr*>
print <*expr*> \ <*var*>	info expr <*var*>; print (<*type*>) <*var*>
quit	quit
+ regs	info registers
regs all	info registers
rerun	rerun
rerun < <*f1*> > <*f2*>	---
rerun <*args*>	run "<*args*>"
rerun <*args*> < <*f1*> > <*f2*>	run "<*args*> < <*f1*> > <*f2*>"

<table>
<tr><th>csd</th><th>CXdb</th></tr>
<tr><td>return</td><td>return</td></tr>
<tr><td></td><td>return <value></td></tr>
<tr><td>return <routine></td><td>---</td></tr>
<tr><td>run</td><td>run</td></tr>
<tr><td>run < <f1> > <f2></td><td>run "< <f1> > <f2>"</td></tr>
<tr><td>run <args></td><td>run "<args>"</td></tr>
<tr><td>run <args> < <f1> > <f2></td><td>run "<args> < <f1> > <f2>"</td></tr>
<tr><td>set</td><td>---</td></tr>
<tr><td>set deref_aaregs = true | false</td><td>---</td></tr>
<tr><td>set dump_lfmt = true | false</td><td>---</td></tr>
<tr><td>set dumpsrc = true | false</td><td>---</td></tr>
<tr><td>set dumpvregs = true | false</td><td>---</td></tr>
<tr><td>+ set num_elements = <num>,<num>,...</td><td>set printopts maxarray <num></td></tr>
<tr><td>+ set precision = <num></td><td>set printopts precision <width>.<decn></td></tr>
<tr><td>sh <cmdline></td><td>shell <cmdline></td></tr>
<tr><td>source <fn></td><td>source <fn></td></tr>
<tr><td>+ status</td><td>info event *</td></tr>
<tr><td>status > <fn></td><td>into event * > <fn></td></tr>
<tr><td>step</td><td>step</td></tr>
<tr><td></td><td>step expression</td></tr>
<tr><td></td><td>step statement</td></tr>
<tr><td></td><td>step loop</td></tr>
<tr><td></td><td>step block</td></tr>
<tr><td></td><td>step routine</td></tr>
<tr><td>step <count></td><td>step <count></td></tr>
<tr><td></td><td>step expression <count></td></tr>
<tr><td></td><td>step statement <count></td></tr>
<tr><td></td><td>step loop <count></td></tr>
<tr><td></td><td>step block <count></td></tr>
<tr><td></td><td>step routine <count></td></tr>
<tr><td>+ step all</td><td>step</td></tr>
<tr><td></td><td>step expression</td></tr>
<tr><td></td><td>step statement</td></tr>
<tr><td></td><td>step loop</td></tr>
<tr><td></td><td>step block</td></tr>
<tr><td></td><td>step routine</td></tr>
<tr><td>+ step all <count></td><td>step <count></td></tr>
<tr><td></td><td>step expression <count></td></tr>
<tr><td></td><td>step statement <count></td></tr>
<tr><td></td><td>step loop <count></td></tr>
<tr><td></td><td>step block <count></td></tr>
<tr><td></td><td>step routine <count></td></tr>
<tr><td>+ stepi</td><td>step instruction</td></tr>
<tr><td>stepi <count></td><td>step instruction <count></td></tr>
<tr><td>stepi all</td><td>step instruction</td></tr>
<tr><td>stepi all <count></td><td>step instruction <count></td></tr>
<tr><td>+ stop at <line></td><td>break line <line></td></tr>
<tr><td>stop at <line> if <cond></td><td>break line <line> {if (!<cond>) resume;}</td></tr>
<tr><td>stop at "<file>":line</td><td>break line <file>:<line></td></tr>
</table>

csd	CXdb
stop at "<*file*>":line if <*cond*>	break line <*file*>:<*line*> {if (!<*cond*>) resume;}
+ stop in <*routine*>	break routine <*routine*>
stop in <*routine*> if <*cond*>	break routine <*routine*> {if (!<*cond*>) resume;}
+ stop if <*cond*>	event relation <*cond*>
stop <*var*>	event modify &<*var*>
stop <*var*> if <*cond*>	event modify &<*var*> {if (!<*cond*>) resume;}
+ stop threads	event spawn; event join
stop threads in <*routine*>	Must be done manually
+ stopi at <*addr*>	break instruction <*addr*>
stopi at <*addr*> if <*cond*>	break instruction <*addr*> { if (!<*cond*>) resume;}
stopi <*var*>	event modify &<*var*>
stopi <*var*> if <*cond*>	event modify &<*var*> { if (!<*cond*>) resume;}
+ thread	info process info threads
thread <*number*>	Use :t command focus on appropriate commands
threads	info process info threads
threads <*number*>	---
+ threads false	remove eventtype join, spawn
+ threads true	event join; event spawn
trace	---
trace <*line*>	trace line <*line*>
trace <*line*> if <*cond*>	trace line <*line*> {if (<*cond*>) echo "*message*"; resume;}
trace "<*file*>":<*line*>	trace line <*file*>:<*line*>
trace "<*file*>":<*line*> if <*cond*>	trace line <*file*>:<*line*> {if (<*cond*>) echo "*message*"; resume;}
trace in <*routine*>	---
trace <*routine*>	trace routine <*routine*>
trace <*routine*> if <*cond*>	trace routine <*routine*> {if (<*cond*>) echo "*message*"; resume;}
trace <*exp*> at <*line*>	trace line <*line*> { print <*exp*>; resume;}
trace <*exp*> at <*line*> if <*cond*>	trace line <*line*> { if (<*cond*>) print <*exp*>; resume;}
trace <*exp*> at "<*file*>":<*line*>	trace line <*file*>:<*line*> {print <*exp*>; resume;}
trace <exp> at "<*file*>":<*line*> if <*cond*>	trace line <*file*>:<*line*> { if (<*cond*>) print <*exp*>; resume;}
trace <*var*> in <*routine*> if <*cond*>	Must be done manually
+ trace threads	event spawn {echo "*message*"; resume;}
trace threads in <*routine*>	Must be done manually
tracei	---
tracei if <*cond*>	---
tracei <*addr*>	trace instruction <*addr*>
tracei <*addr*> if <*cond*>	trace instruction <*addr*> {if <*cond*> echo "*message*"; resume;}

csd	CXdb
tracei *<var>*	event modify &*<var>* {echo "*message*"; resume;}
tracei *<var>* if *<cond>*	event modify &*<var>* {if (*<cond>*) echo "*message*"; resume;}
+ unalias *<name>*	remove alias *<name>*
up	frame +1
up *<num>*	frame +*<num>*
use	info process
+ use *<dir>*...	set path *<dir>*...
+ vregs	info vregisters
vregs *<index>*	print $V*<index>*
vregs all	info vregisters
vregs all *<index>*	info vregisters
+ whatis *<routine>*	info expr *<routine>*
+ whatis *<type>*	info type *<type>*
+ whatis *<variable>*	info expr *<variable>*
+ when at *<line>* {*<cmd>*;...}	event reached line *<line>* {*<cmd>*; ...; resume;}
+ when at *<file>*:*<line>* {*<cmd>*;...}	event reached line *<file>*:*<line>* {*<cmd>*;...; resume;}
+ when in *<routine>* {*<cmd>*;...}	event reached routine *<routine>* {*<cmd>*; ...; resume;}
when *<cond>* {*<cmd>*;...}	event relation *<cond>* {*<cmd>*; ...; resume;}
+ where	backtrace
+ where > *<file>*	backtrace > *<file>*
+ where *<num>*	backtrace *<num>*
+ whereis *<identifier>*	info symbols *<identifier>*
+ which *<identifier>*	info symbols *<identifier>*

Cross-reference to gdb commands

This appendix contains a cross-reference between the commands for the gdb debugger and CXdb commands. In cases where there is not a one-to-one correspondence between commands, macros are given if possible.

For the most frequently used gdb commands, aliases have been provided in CXdb. The alias name in CXdb is the same as the gdb command itself. Those gdb commands that have aliases are indicated on the following pages by a plus sign (+) in front of the commands.

The gdb aliases are defined in the command file /usr/lib/cxdb/gdb_aliases. This command files does *not* execute automatically when you invoke CXdb. If you want to use the aliases, first execute the command file with the source command, as illustrated in Figure 387.

Figure 387
Incorporating the gdb aliases

```
(CXdb) source /usr/lib/cxdb/gdb_aliases
```

You can include the above source command in your .cxdbinit file so that the aliases are incorporated automatically each time you invoke CXdb.

gdb	CXdb
$_	Can be done with a macro
$__	Can be done with a macro
$<*n*>	---
1cont	:t <*thread-#*> cont
1cont <*ignore*>	---
$pc	$pc
$psw	$psw
$sp	$sp
$ap	$ap
$fp	$fp
$a1-5	$a1-5
$s0-7	$s0-7
$v0-7	$v0-7
$vl	$vl
$vs	$vs
$vm	$vmu, $vml
$c0-63	$c0-63
$S0-7	$S0-7
$V0-7	$V0-7
$C0-63	$C0-63
add-file	load object
alias <*new*> <*old*>	alias <*new*> <*old*>
attach <*pid*>	attach <*pid*>
attach <*device*>	---
backtrace	backtrace
backtrace <*count*>	Use backtrace, but you always get all frames
break <*function*>	break routine <*function*>
break +<*offset*>	Must be done manually
break -<*offset*>	Must be done manually
break <*line*>	break line <*line*>
break <*file*>:<*line*>	break line <*file*>:<*line*>
break <*file*>:<*function*>	break routine <*file, no extension*>`<*function*>
break *<*address*>	break instruction <*address*>
break	Must be done manually
break ... <*cond*>	break ... {if (!<*cond*>) resume;}
cd <*dir*>	cd <*dir*>; set directory <*dir*>
clear	Must be done manually using remove event
clear <*function*>	Must be done manually using remove event
clear <*filename*>:<*function*>	Must be done manually using remove event
clear <*linenum*>	Must be done manually using remove event
clear <*filename*>:<*linenum*>	Must be done manually using remove event
commands \n <*cmds*> \n...\n end	set handler <*bnum*> {<*cmds*>; ...}
commands <*bnum*> \n <*cmds*> \n...\n end	set handler <*bnum*> {<*cmds*>; ...}
condition <*bnum*>	Create handler with no condition using set handler <*bnum*> { <*handler*> }
condition <*bnum*> <*cond*>	Create handler with a condition using set handler <*bnum*> { if (<*cond*>) {<*handler*>} else resume;}
cont	continue
cont <*ignore*>	set ignore <*ignore*> <*bnum*>; continue

gdb	CXdb
core-file	---
+ core-file <*file*>	debug core <*file*>
+ define <*name*> \n <*cmd*>... \n end	alias <*name*> '<*cmd*>; ...'
+ delete <*bnum*> ...	remove event <*bnum*>, ...
delete display <*dnums*>	---
+ delete environment <*varname*>	remove environment <*varname*>
detach	detach
directory	Use pwd then set path <*dir*>
+ directory <*dir*>...	add path <*dir*>, ...
+ disable breakpoints <*bnum*> ...	disable event <*bnum*>, ...
disable display	---
disassemble	disassemble
disassemble <*addr*>	disassemble <*addr*>
disassemble <*start*> <*end*>	disassemble <*start*>..<*end*>
display	---
display <*exp*>	---
display/fmt <*exp*>	---
display/fmt <*addr*>	---
document	---
+ down	frame -1
down <*n*>	frame -<*n*>
down -<*n*>	frame +<*n*>
dump-me	Use kill -SEGV <*pid*> to kill CXdb from a shell
echo <*string*>	echo/n <*string*>
echo <*string*>\n	echo <*string*>
+ enable breakpoints <*bnum*> ...	enable event <*bnum*>, ...
enable breakpoints once <*bnum*> ...	Modify handler of each to include disable $self; as the last command
enable breakpoints delete <*bnum*> ...	Modify handler of each to include remove event $self; as the last command
enable display	---
exec-file	---
+ exec-file <*file*>	debug exec <*file*>
finish	finish routine
+ forward-search	find window forward
frame	info frame
frame <*n*>	frame <*n*>
frame <*addr*>	---
+ handle <*sig*> <*action*>...	set signal <*sig*> <*action*>,...
help	help
+ ignore <*bnum*> <*count*>	set ignore <*count*> <*bnum*>
+ info address <*symbol*>	info expr <*symbol*>
info args	info args
info break	info break
info break <*num*>	info event <*num*>
+ info comm-registers	info cregisters
info comm-registers c<*creg*>	info expr $c<*creg*>; pr $cl<*creg*>
info comm-registers <*addr*>	---
info convenience	---
+ info directories	info process

gdb	CXdb
`info display`	---
`info environment`	`info environment`
`info environment` *<var>*	`info environment` *<var>*
+ `info files`	`info process`
`info frame`	`info frame`
+ `info functions`	`info symbols`
`info functions` *<regexp>*	`info symbols` *<regexp>*
`info history`	CXdb version gives command history
`info line`	CXdb version lists source units at a line of code
`info locals`	`info locals`
`info psw`	`pr/B $psw`
`info psw` *<arg>*	`info psw`
`info registers`	`info registers; info cregisters; info vregisters`
`info registers` *<regname>*	`print $`*<regname>*
`info signal`	`info signal`
`info signal` *<signo>*	`info signal` *<signo>*
+ `info sources`	`info objectmap` can be used and the filenames' extensions changed from `.o` to `.c` (etc.)
`info threads`	`info threads`
`info types`	`info type`
`info types` *<regexp>*	`info type` *<regexp>*
+ `info variables`	`info symbols`
+ `jump` *<linenum>*	`goto line`
+ `jump *`*<addr>*	`goto address` *<addr>*
`kill`	`kill process`
`list`	Use `display file` or `display routine`
`next`	`next`
`next` *<n>*	`next` *<n>*
`nexti`	`next instruction`
`nexti` *<n>*	`next instruction` *<n>*
`output`	---
`print`	`print`
`printf` *<format>* *<arg>*`...`	---
+ `printsyms` *<file>*	`info symbols >` *<file>*
+ `ptype` *<typename>*	`info type` *<typename>*
`pwd`	`info cxdb; info process`
`quit`	`quit`
`return`	`return`
`return` *<value>*	`return` *<value>*
+ `reverse-search`	`find window backward`
`run`	`run` (first time) `rerun` (after first time)
`run` *<args>*	`run "`*<args>*`"`
`search`	---
`set`	`evaluate`
`set args`	Part of the `run` and `rerun` commands
+ `set array-max` *<num>*	`set printopts maxarray` *<num>*
`set base`	Use `set format` with `examine` or command line options with `print`

gdb	CXdb
set base <*n*>	Use set format with examine or command line options with print
set compile off	---
set compile on	---
set compiled-breakpoints	---
set debug-flag	---
set editing	Always enabled
set environment <*var*> = <*value*>	set environment <*var*> = <*value*>
set history expansion	---
+ set history file <*file*>	set cmdlog <*file*>
set history size	Fixed at 100
+ set history write on	set logging
+ set history write off	clear logging
+ set parallel fixed	set fixed sched
set parallel off	---
+ set parallel on	clear fixed sched
+ set pipeline off	set seq
+ set pipeline on	clear seq
set prettyprint	---
set prompt	Done in .Xdefaults file
set screensize	Done in .Xdefaults file
set unionprint	---
set verbose	---
shell	shell
shell <*cmds*>	shell "<*cmds*>"
signal <*sig*>	signal process <*sig*>
source <*file*>	source <*file*>
step	step
step <*n*>	step <*n*>
stepi	step instruction
stepi <*n*>	step instruction <*n*>
symbol-file <*file*>	Automatically as needed
+ tbreak ...	break ... {...; disable event $self;}
term-status	---
+ thread	info threads
thread <*n*>	---
tty <*dev*>	CXdb uses separate window
undisplay <*nums*>	---
+ unset environment <*var*>	remove environment <*var*>
+ until	finish loop
until <*location*>	Must be done manually
+ up	frame +1
up <*n*>	frame +<*n*>
up -<*n*>	frame -<*n*>
+ whatis <*exp*>	info expression <*exp*>
+ x/<*fmt*> <*addr*>	examine (some variant)

Index

Symbols

238
$self 137
$signal 149
$v0 352
$v1 352
$vs 352
&
  as a C operator 89
  for background execution 25, 60
(CXdb) 12, 23, 25
.CXdb 13, 310
.cxdbinit. *see* files, initialization
.o 13
.Xdefaults 12
/usr/lib/cxdb 260
/usr/lib/cxdb/examples 6
:t 366
@. *see* macros, invoking
\ 232, 314
\; 144

A

add cmderr. *see* commands, add cmderr
add cmdlog. *see* commands, add cmdlog
add cmdout. *see* commands, add cmdout
add default path. *see* commands, add default path
add environment. *see* commands, add environment
add path. *see* commands, add path
addresses
  memory 81
  ranges 154, 332
  ranges for source units 330
  specifying a range 92
  symbolic 81
alias. *see* commands, alias
aliases
  compared to macros 235
  creating 232
  defaults 230, 260
  deleting 233
  displaying 230
  invoking 234
  modifying 233
  multiword 232
ALTE 317
anchoring windows. *see* Maryland Windows, anchoring
arguments
  block numbers in pcc 279
  passing to a macro. *see* macros, passing parameters to
  passing to a process 22, 25
array slices
  described 174
  in C 177
  in FORTRAN 175
assembly code. *see* disassembled code
assignment substitution 338
attach. *see* commands, attach
attaching to a process 219
auto update 173, 174, 289, 301

B

background
  execution of commands 25, 60
  stopping execution 28, 223
backslash 232, 314
backtrace. *see* commands, backtrace
basic block 311, 334
batch mode 261
BITB 317
BITSHIFT 320
block
  basic. *see* basic block
  source unit 3, 329

block numbers
 described 277
 in ANSI C mode 278
 in pcc mode 279
body of a loop 327, 329
BOOT 317
break line. *see* commands, break line
break routine. *see* commands, break routine
break source. *see* commands, break source
breakpoints
 described 78
 displaying. *see* commands, info break
 number 81
 removing. *see* commands, remove event
 setting
  at a line 16, 80, 87
  at a routine 82
  at a source unit 104
  methods 79
BSS 317
buttons
 command 12
 paging 169, 300

C

c$ 268
cd
 CXdb command. *see* commands, cd
 shell command 10
changing executables 224
clear default fixed sched. *see* commands, clear default fixed sched
clear default handler. *see* commands, clear default handler
clear environment. *see* commands, clear environment
clear fixed sched. *see* commands, clear fixed sched
clear handler. *see* commands, clear handler
clear logging. *see* commands, clear logging
clear seq. *see* commands, clear seq
clear sqs. *see* commands, clear sqs
clear step. *see* commands, clear step
clear typehandler. *see* commands, clear typehandler
closing windows. *see* Maryland Windows, closing
cmderr. *see* logging, messages
cmdlog. *see* logging, input
cmdout. *see* logging, output
CMIN 317
code motion 336, 340, 346
command files. *see* files, command
command history. *see* history
command line 12, 47
command window
 command line 12, 47
 in CXwindows. *see* command window (CXwindows)
 in Maryland Windows. *see* command window (Maryland)
 window number 12, 47
command window (CXwindows)
 buttons 12
 command menus
  Configuration 12
  Events 12
  Execution 12, 18
  Info 12
  Misc 12
  Process 12
 described 10
 scroll bars 12
 window menus
  CommandWindow 11
  CXdbWindows 11
 Xdefaults 12
command window (Maryland) 46
 scrolling 62
 title bar 47
 window edges 47
commands 281
 add cmderr 247
 add cmdlog 249
 add cmdout 244
 add default path
  described 204
  example of 206
 add environment
  described 185
 add path
  described 204
  example of 207
 alias 232
 attach
  described 218
  example of 220
 backtrace 283
 break line
  described 16, 54, 80
  example of 85, 87, 140, 369
  in a source file 85
 break routine 82
 break source 104, 331
 cd
  described 203
  example of 224
 clear default fixed sched 363
 clear default handler 143
 clear environment
  described 185
 clear fixed sched 194

commands (*continued*)
- clear handler 140
- clear logging 252
- clear seq 197
- clear sqs 197
- clear step 184, 196
- clear typehandler 142
- continue
  - described 22
  - example of 25, 58, 133, 372
- copy 166
- core 218
- cxdb
  - -b option 261
  - described 10
  - example of 46
- debug core 218
- debug exec
  - described 13, 218
  - example of 48
- debug proc 218
- detach 223
- disable event 128
- disassemble 295
- display file
  - described 210
  - example of 246
- display routine 211
- echo 137
- edit 256
- enable event 129
- evaluate 138, 158
- event join 371
- event reached 128
- event relation 144
- event signal 149
- event spawn 363
- examine
  - described 161
  - example of 190, 192
- executable 218
- fill 166
- find memory backward 163
- find memory forward 163
- find window backward 212
- find window forward 212
- finish
  - comparison to step 117
  - described 105, 115
  - example of 115, 123
- frame 285
- help
  - described 28
  - example of 31, 64
- if 140
- info alias 230
- info break 83
- info cxdb
  - described 27
  - example of 62, 205, 245, 247, 249, 253, 259
- info default environment 186
- info environment 186
- info event
  - described 96
  - example of 97
- info eventtype 134
- info expression
  - described 153
  - example of 161
  - for synthesized variables 315, 342
- info formatting
  - described 157
  - example of 192, 193
- info frame 284
- info frame at 284
- info history 240
- info line
  - described 103
  - example of 328
- info locals 153
- info macro 236
- info process
  - described 26
  - example of 61, 105, 196, 206
- info registers
  - example of 369
- info scope
  - described 270
  - example of 275
- info signal
  - described 146
  - example of 148, 198
- info source 104
- info threads 365
- info trace 89
- info watch 91
- kill process
  - described 22
  - example of 58, 222
- macro 235
- next
  - comparison to step 107
  - described 105
  - example of 109, 111, 294, 298
- next over 118
  - described 105
- print
  - described 21
  - example of 57, 155, 176, 178, 270, 276, 279, 282, 342
  - formatting output 156
- pwd 203

commands (*continued*)
quit
described 44
example of 75
recall 241
remove alias
described 233
remove cmderr 251
remove cmdlog 251
remove cmdout 251
remove default path 204
remove environment
described 185
remove event
described 84
example of 86, 97
remove eventtype 135
remove macro
described 239
remove path
described 204
example of 208
rerun
described 25, 218
example of 60
resume 139
run
described 18, 218
example of 22, 55, 58
set cmderr 252
set cmdlog 252
set cmdout 252
set default fixed sched 363
set default handler 143
set default path 204
set default step 196
set directory 185
set echo 257
set environment
described 185
set fixed sched
described 194
example of 363
set format 190
set fpmode 189
set handler 137
set ignore 130
set logging 249
set memory 192
set noclobber 253
set path 204
set printopts
described 157
example of 175
set printopts. *see* commands, set printopts 157
set pshell 194
set seq 197
set signal
described 148, 197
example of 197
set sqs 197
set step
described 111
example of 195, 294
set typehandler 141
shell
described 225
example of 226
signal process 150
signal thread 370
source 257
step
comparison to finish 117
comparison to next 107
comparison to step over 118
described 105
example of 107, 110, 122, 293, 297, 367
step over
comparison to step 118
described 105, 118
example of 120
stop
described 28
example of 63, 223
trace line 88
trace routine 88
watch
described 90
example of 93, 95
CommandWindow menu 11
common blocks 271
compiler-generated variables 333
compilers
-cxdb option 13, 310
completion 21, 57
Configuration menu 12
console working directory 202
contact utility xviii
continue. *see* commands, continue
copy. *see* commands, copy
core. *see* commands, core
CREG 317
csd, cross-reference 401
CTMP 315
CTRL-c 28, 222
CTRL-n 241
CTRL-p 241
CTRL-z 219
current frame 283
current scope. *see* scope, current
current source unit 118
current thread 366
CX/Motif 10

CXdb
  concepts 2
  described 1
  features 1
  invoking 10
  Online Guide 6, 44
  overview 1
  prompt 12, 23, 25
  symbolic debugger 1
-cxdb option 13, 310
cxdb$ 268
cxdb(1) 10
cxdb. *see* commands, cxdb
CXdbWindows menu 11, 286, 299
CXwindows
  described 9
  scrollbars 10
  tutorial 10

D

DATA 317
data
  program. *see* program variables
DataView menu 168
DEAD 318
debug core. *see* commands, debug core
debug exec. *see* commands, debug exec
debug proc. *see* commands, debug proc
debugger variables
  $self 137
  $signal 149
  creating 159
  data type 159
  described 139
  displaying 155, 282
  modifying 159
  with eventpoints 145, 152
default process settings. *see* process settings, default
detach *see* commands, detach
detaching from a process 223
DEXP 318
directory
  console working 202, 260
  home 245
  process. *see* process settings, process directory
disable event. *see* commands, disable event
disassemble. *see* commands, disassemble
disassembled code 295
disassembly window
  auto update 301
  changing address 304
  described 299
  menus
    DisassemblyWindow 299
    InstructionView 299
    RegisterView 299
  paging buttons 300
  Xdefaults 303
DisassemblyWindow menu 299
display file window 210, 246
display file. *see* commands, display file
display formats. *see* process settings, display formats
display routine. *see* commands, display routine
DisplayFile Window menu 211
documentation
  ordering xvii
  related xvii
dual mode. *see* process settings, floating point mode

E

echo. *see* commands, echo
edit. *see* commands, edit
enable event. *see* commands, enable event
environment. *see* process settings, process environment
evaluate. *see* commands, evaluate
event join. *see* commands, event join
Event Point Dialog window 17
event reached. *see* commands, event reached
event relation. *see* commands, event relation
event signal. *see* commands, event signal
event spawn. *see* commands, event spawn
eventpoints
  address 81
  breakpoints. *see* breakpoints
  described 4
  disabling 128
  displaying. *see* commands, info event
  enabled setting 80, 129
  enabling 128
  handlers. *see* handlers
  ignore count 81, 130
  in optimized code 323
  marker 17, 54, 81
  multiple, at one location 132
  number 81, 85
  thread-specific 368
  tracepoints. *see* tracepoints
  types
    described 127
    displaying. *see* commands, info eventtype
    join 371
    reached 128
    relation 143
    removing. *see* commands, remove eventtype

signal 149
spawn 363
watchpoints. *see* watchpoints
Events menu 12
examine window
auto update 173, 174
described 168
menus
DataView 168
ExamineWindow 168
paging buttons 169
Xdefaults 174
`examine`. *see* commands, `examine`
ExamineWindow menu 168
examples
directory 6
introduction 6
source code 375
executable file
changing 224
specifying 13
`executable`. *see* commands, `executable`
Execution 18
Execution menu 12
exit 226
expression
language. *see* language expression
source unit 3
expression terminator 144

F

`f$` 268
`fg` 221
files
command
creating 255
described 254
executing 257
compiler-generated 13
displaying 210
executable 224
specifying 13
initialization
`.cxdbinit` 209, 230, 260
creating 255
described 208, 259
order of execution 260
object 13
source 13, 80
FileView menu 14
`fill`. *see* commands, `fill`
`find memory backward`. *see* commands, `find memory backward`
`find memory forward`. *see* commands, `find memory forward`
`find window backward`. *see* commands, `find window backward`
`find window forward`. *see* commands, `find window forward`
`finish`. *see* commands, `finish`
fixed scheduling. *see* process settings, fixed scheduling
FLNK 317
floating point mode. *see* process settings, floating point mode 189
fpmode. *see* process settings, floating point mode
`frame`. *see* commands, `frame`
FrameView menu 288
functions. *see* routine

G

gdb, cross-reference 407
granularity
block 3
described 101
displaying. *see* commands, `info process`
effects on highlighting 114, 293
expression 3
loop 3
routine 3
setting 111, 195
statement 3
guidelines for debugging optimized code 322

H

handlers
default handler 142
described 136
for eventpoint types 141
removing 140
setting 136
header of a loop 327
help 28
see also help, online
help window
in CXwindows. *see* help window (CXwindows)
in Maryland Windows. *see* help window (Maryland)
window number 30, 65
help window (CXwindows)
described
find button 30
menus
HelpWindow 30
History 30, 42
SearchOptions 30, 39
Topics 30, 35
result of search 30
scroll bars 30
string to search for 30

help window (Maryland)
  described
  help topic 65
help, online
  commands 28
  concepts 28
  CXdb messages 28, 34
  parameters 28
  related topics 32
`help`. *see* commands, `help`
HelpWindow 30
highlighting 20
hints for debugging optimized code 322
History 30, 42
history
  described 240
  displaying 240
  retrieving commands from 241
hoisting 347
home directory 245

I

IEEE mode. *see* process settings, floating point mode
`if`. *see* commands, `if`
induction variables
  changing the base of 319
  described 313
INDV 315, 342
`info alias`. *see* commands, `info alias`
`info break`. *see* commands, `info break`
`info cxdb`. *see*commands, `info cxdb`
`info default environment`. *see* commands, `info default environment`
`info environment`. *see* commands, `info environment`
`info event`. *see* commands, `info event`
`info eventtype`. *see* commands, `info eventtype`
`info expression`. *see* commands, `info expression`
`info formatting`. *see* commands, `info formatting`
`info frame at`. *see* commands, `info frame at`
`info frame`. *see* commands, `info frame`
`info history`. *see* commands, `info history`
`info line`. *see* commands, `info line`
`info locals`. *see* commands, `info locals`
`info macro`. *see* commands, `info macro`
Info menu 12
`info process`. *see*commands, `info process`
`info scope`. *see* commands, `info scope`
`info signal`. *see* commands, `info signal`
`info source`. *see* commands, `info source`
`info threads`. *see* commands, `info threads`
`info trace`. *see* commands, `info trace`
`info watch`. *see* commands, `info watch`
initialization files. *see* files, initilization
innermost active source unit 115
instruction scheduling 331
InstructionView menu 299, 304
intrinsic functions 320
ISTR 316

K

`kill process`. *see* commands, `kill process`

L

`l$` 268
language expression
  as an address 82
  displaying 154
  evaluating 159
  relational 143
  terminator 144
line numbers 80, 335
liveness ranges 154, 332
loader symbol 282
`loc()` 89
logging
  input 249
  messages 247
  output 244
  overwrite protection 253
loop interchange 348
loops
  body 327, 329
  components of 327
  header 327
  source unit of 3
  unrolled 316, 320
lowering windows. *see* Maryland Windows, lowering

M

`macro`. *see* commands, `macro`
macros
  compared to aliases 235
  creating 235
  deleting 239
  described 235
  displaying 236
  invoking 237
  iterative 237
  modifying 239
  passing parameters to 237
  token pasting 237, 238

Maryland Windows
  anchoring 71
  closing 73
  described 45
  keystrokes quick reference 397
  lowering 67
  moving 69
  moving between windows 51
  raising 67
  resizing 69
  scrolling 50
`MAX` 320
memory
  display formats 190
  examining 161
  modifying 166
  region. *see* addresses, specifying a range
  searching 163
  unit 192
menus
  CommandWindow 11
  Configuration 12
  CXdbWindows 11, 286, 299
  DataView 168
  DisassemblyWindow 299
  DisplayFile Window 211
  Events 12
  ExamineWindow 168
  Execution 12, 18
  FileView 14
  HelpWindow menu 30
  History 42
  History menu 30
  Info 12
  InstructionView 299, 304
  Misc 12
  Process 12
  ProcessWindows 14, 299
  RegisterView 299, 306
  SearchOptions menu 30, 39
  SourceUnit 14
  SourceWindow 14
  Topics 35
  Topics menu 30
`MIN` 320
Misc menu 12
MLXS 316
moving windows. *see* Maryland Windows, moving 69

N

native mode. *see* process settings, floating point mode
`next over`. *see* commands, `next over`
`next`. *see* commands, `next`
`-no` 326
  see also optimization, levels
noclobber. *see* logging, overwrite protection
notational conventions xv

O

`-O0` 334
`-O1` 340
`-O2` 346
`-O3` 360
object file 13
Online Guide 6, 44
online help. *see* help, online
optimization
  described 310
  hints for debugging 322
  levels
    `-no` 326
    `-O0` 334
    `-O1` 340
    `-O2` 346
    `-O3` 360
  of variables 312
    see also synthesized variables
  types
    assignment substitution 338
    code motion 336, 340, 346
    hoisting 347
    instruction scheduling 331
    loop interchange 348
    parallelization 360
    redundant-use elimination 334
    strength reduction 341
    strip mining 352, 354
    vectorization 346, 351, 354
OSTR 316
output, formatting 156

P

paging buttons
  disassembly window 300
  examine window 169
parallelization 360
parameters. *see* macros, passing parameters to
PBKE 316
PBKU 316
PC. *see* program counter
`print`
  example of 281
`print`. *see* commands, `print`
process image 18
process interface window
  activating in Maryland Windows 59
  closing 24
  input to 23
  invoking 18, 55

Process menu 12
process number. *see* process object
process object 13, 15, 18, 80, 182
process settings
  affecting a new process 199
  affecting the current process 199
  default 184
  described 182
  display formats 190
  fixed scheduling 194, 363
  floating point mode 189
  memory unit 192
  process directory 184, 185, 186, 189, 190, 192, 194, 196, 197, 203, 204, 210, 211, 212
  process environment 185
  process shell 194
  search path 194
  SEQ bit 197
  signal actions 197
  SQS bit 197
  stepping granularity 195
  types 183
processor status word (PSW) 197
ProcessWindows menu 14, 299
program counter (PC) 154, 168, 300
program variables
  array slices. *see* array slices
  displaying 152, 319
  inactive 320
  inconsistent values for 342
  modifying 158
  not updated 321
  optimization of. *see* synthesized variables
`ps` 220
PSW. *see* processor status word
`pwd`. *see* commands, `pwd`

Q

`quit`. *see* commands, `quit`

R

raising windows. *see* Maryland Windows, raising
`recall`. *see* commands, `recall`
recursion. *see* macros, iterative
redirection operators 252
redundant-use elimination 334
registers
  displaying 155, 306
  modifying 159
  vector accumulator 352
  vector length 352
  vector stride 352
  vector. *see* vector registers
RegisterView menu 299, 306
relational expression 143
relocating windows. *see* Maryland Windows, moving
`remove alias`. *see* commands, `remove alias`
`remove cmderr`. *see* commands, `remove cmderr`
`remove cmdlog`. *see* commands, `remove cmdlog`
`remove cmdout`. *see* commands, `remove cmdout`
`remove default path`. *see* commands, `remove dafault path`
`remove environment`. *see* commands, `remove environment`
`remove event`. *see* commands, `remove event`
`remove eventtype`. *see* commands, `remove eventtype`
`remove macro`. *see* commands, `remove macro`
`remove path`. *see* commands, `remove path`
reporting problems xviii
`rerun`. *see* commands, `rerun`
resizing windows. *see* Maryland Windows, resizing
`resume`. *see* commands, `resume`
REXP 316
routine
  displaying 154, 211
  executing 160
  preamble 83
  source unit 3
`run`. *see* commands, `run`

S

`s$` 268
scope
  changing 283
  current 266
  displaying 270, 275
scope paths
  in C 273
  in FORTRAN 269
  language prefix
    `c$` 268
    `cxdb$` 268
    described 268
    `f$` 268
    `l$` 268
    `s$` 268
  mixed languages 281
  to view common blocks 271
scrolling. *see* Maryland Windows, scrolling
search path
  default 203
  for source files 85, 194, 203
searching
  a source file 212
  for a help topic 37
SearchOptions menu 30, 39
SEQ bit. *see* process settings, SEQ bit
`set cmderr`. *see* commands, `set cmderr`

set cmdlog. *see* commands, set cmdlog
set cmdout. *see* commands, set cmdout
set default fixed sched. *see* commands, set default fixed sched
set default handler. *see* commands, set default handler
set default path. *see* commands, set default path
set default step. *see* commands, set default step
set dircetory. *see* commands, set directory
set echo. *see* commands, set echo
set environment. *see* commands, set environment
set fixed sched. *see* commands, set fixed sched
set format. *see* commands, set format
set fpmode. *see* commands, set fpmode
set handler. *see* commands, set handler
set ignore. *see* commands, set ignore
set logging. *see* commands, set logging
set memory. *see* commands, set memory
set noclobber. *see* commands, set noclobber
set path. *see* commands, set path
set printopts. *see* commands, set printopts
set pshell. *see* commands, set pshell
set seq. *see* commands, set seq
set signal. *see* commands, set signal
set sqs. *see* commands, set sqs
set step. *see* commands, set step
set typehandler. *see* commands, set typehandler
SEXP 316
shell
  process. *see* process setting, process shell
shell commands
  cd 10
  exit 226
  fg 221
  issuing from CXdb 225
  ps 220
shell window 225
shell. *see* commands, shell
signal processr. *see* commands, signal process
signal thread. *see* commands, signal thread
signals
  actions 146, 197
  described 146
  displaying. *see* commands, info signal
  sending to a process 150
  sending to a thread 370
  setting actions 147
sigvec() 146
SINK 316
source file 13, 80
source units
  block 3, 329
  current 118
  described 3, 101
  displaying. *see* commands, info line
  examples of 101
  expression 3
  granularity. *see* granularity
  highlighting 20
  in optimized code 310
  innermost active 115
  loop 3
  mapping to object code 310
  ranges 311, 330
  routine 3
  statement 3
source window
  displaying multiple routines 211
  highlighting 20
  in CXwindows. *see* source window (CXwindows)
  in Maryland Windows. *see* source window (Maryland)
  process object 15
  searching 212
  window number 15, 50
source window (CXwindows)
  described 14
  menus
    FileView 14
    ProcessWindows 14
    SourceUnit 14
    SourceWindow 14
  scroll bars 15
source window (Maryland)
  described 49
  window edges 50
source. *see* commands, source
SourceUnit menu 14
SourceWindow menu 14
specifying a process 218
SPLL 317
SQS bit. *see* process settings, SQS bit
stack
  displaying 283
  frames
    current 283
    displaying 284
stack frame description window 288
stack window
  auto update 289
  described 286
  menus
    FrameView 288
    StackWindow 289
StackWindow menu 289
statement, source unit 3
step over. *see* commands, step over
step. *see* commands, step

stepping
  by block 113
  by expression 293
  by instruction 297
  by loop 109
  by loop iterations 113
  by routine 122
  by statement 106, 337
  described 5, 100
  granularity. *see* granularity
  over source units 118
  threads 367
  through optimized code 323
  to the end of a routine 123
STML 317
`stop`. *see* commands, `stop`
strength reduction 341
strip mining 352, 354
symbolic debugger 1
synthesized variables
  at level `-O1` 341
  described 312
  displaying 342
  from parallel equations 319
  incomplete equations 321
  scope path 314
  types 315

T

TAC xviii
TBSS 317
TDATA 317
Technical Assistance Center (TAC) xviii
terminator for language expressions 144
TEXT 317
thread list 366
threads
  current 366
  described 360
  displaying 365
  number 80
  specifying with a list 366
token pasting 237, 238
Topics 30, 35
TPTR 315
`trace line`. *see* comamnds, `trace line`
`trace routine`. *see* commands, `trace routine`
tracepoints
  described 79
  displaying. *see* commands, `info trace` 89
  setting 88
TRIP 316
tutorial. *see* *Online Guide*

U

unrolled loops 316, 320
UREX 316
URIV 316
URTP 316
user interface
  CXwindows. *see* CXwindows
  Maryland Windows. *see* Maryland Windows
  types 2
user-defined functions 321

V

variables
  compiler-generated 333
  debugger. *see* debugger variables
  inactive 320
  induction. *see* induction variables
  not updated 321
  optimization of. *see* synthesized variables
  program. *see* program variables
  see also scope paths
  synthesized. *see* synthesized variables
VBOT 317
vector accumulator registers 352
vector length 352
vector registers 352
vector stride 352
vectorization 346, 351, 354
viewports
  deleting 251
  described 244
  modifying 251
`vl` 352
VMSK 317
`vs` 352
VSPL 317

W

`watch`. *see* commands, `watch`
watchpoints
  described 79
  displaying 91
  setting
    at a variable 89
    at an address range 92
windows
  command. *see* command window
  disassembly. *see* disassembly window
  display file. *see* display file window
  Event Point Dialog 17
  examine. *see* examine window
  help. *see* help window

manager 10, 12
menus. *see* menus
number 12, 15, 30, 47, 50, 65
process interface. *see* process interface window
described 19
shell 225
source. *see* source window
stack frame description 288
stack. *see* stack window 286
thread-specific 370
types 2

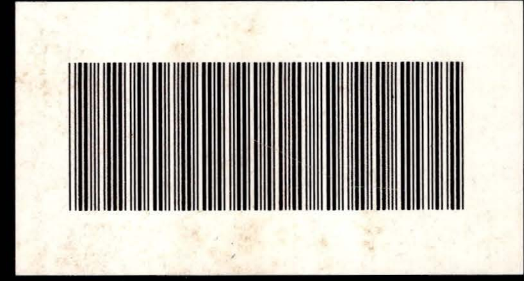
X

X resources. *see* Xdefaults
X Window System 9
Xdefaults
command window (CXwindows) 12
disassembly window 303
examine window 174

Z

ZMSK 317

Order Number
DSW-473



Document Number
710-015530-001