CONVEX CXdb Concepts

First Edition

CONVEX COMPUTER CORPORATION

CONVEX
CXdb Concepts

Order No. DSW-471

First Edition
May 1991

CONVEX Press
Richardson, Texas USA

CONVEX
CXdb Concepts

Order No. DSW-471

Printed in the United States of America

Revision Information for

CONVEX
CXdb Concepts

Edition	Document No.	Description
First Edition	710-015330-001	May, 1991. Initial Release.

Contents

Fundamental Concepts

Advanced Concepts

Optimized Code Concepts

Glossary

Index

Figures

Using This Book

Purpose and audience

The *CONVEX CXdb Concepts* book describes CXdb, the CONVEX Visual Debugger. CXdb has many innovative features such as a graphical user interface and the capability to debug optimized code. These features involve some concepts that are new even to experienced debugger users.

This book has two purposes. The first is to give you a broad overview of CXdb's features and capabilities. The second is to explain how traditional and new debugging concepts are used in CXdb. Understanding these concepts makes it much easier to begin using CXdb.

This book does not describe how to use CXdb. For information on how to use CXdb, please refer to the *CONVEX CXdb User's Guide*. For full details on particular CXdb topics, refer to the *CONVEX CXdb Reference* manual.

This book is intended for two types of readers:

- Those who are new to visual debuggers, including CXdb
- Those who have some exposure to visual debuggers, but are new to CXdb

Organization

This book is organized into five chapters. The first two chapters describe the features and interfaces of CXdb. The remaining chapters discuss concepts you will be using while debugging with CXdb.

- **Chapter 1, "Overview of CXdb"**—Introduces CXdb and explains how it compares to traditional debuggers.
- **Chapter 2, "The Debugger Environment"**—Describes the user interfaces and the various windows available in CXdb.
- **Chapter 3, "Fundamental Concepts"**—Discusses fundamental concepts that all users of CXdb are likely to need.

- **Chapter 4, "Advanced Concepts"**—Covers topics that the more experienced users of CXdb will encounter.
- **Chapter 5, "Optimized Code Concepts"**—Addresses issues involved with debugging optimized code.

Notational conventions

This document uses the following notational conventions:

- The CXdb command line prompt looks like the following: (CXdb)
- `Monospaced type` is used to indicate operating system utilities, filenames, and directories.
- All source code examples are in FORTRAN.
- Notes are indicated by bold type and a heading in the margin, as shown below:

Note

A Note contains information that may be of particular interest regarding the software or your files.

Associated documents

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

- *CONVEX CXdb Quick Reference*—A quick reference card to using CXdb showing command syntax and description.
- *CONVEX CXdb Reference*—The complete reference to all CXdb commands, concepts, parameters, and CXdb messages.
- *CONVEX CXdb User's Guide*—A guide to using CXdb, including examples that you can enter as you read.

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 or call the Technical Assistance Center.

Technical assistance

If you have questions that are not answered by the documentation, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384
- 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.

To use `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

Overview of CXdb

1

CXdb offers all the capabilities of traditional debuggers. In addition, CXdb has new capabilities to provide you with greater power, control, and flexibility when debugging. This chapter describes these debugging capabilities, as well as special features that make CXdb easier to use.

Description

The CONVEX Visual Debugger (CXdb) is a symbolic debugger. A symbolic debugger maps your source code to the executable code created by the compiler. With a symbolic debugger you can debug your program by setting breakpoints, stepping through process execution by source statements, and displaying the values of program variables.

In addition to traditional debugging capabilities, CXdb has three main advantages over traditional debuggers:

- CXdb's window interface makes interacting with the debugger easier than with command-line-only debuggers.
- CXdb enables finer debugging precision by separating source code into syntactical units, called source units, in addition to separating by line number.
- With CXdb you can debug code optimized up to the `-O1` level.

With CXdb, you can look at the disassembly code, the source code, and the output of your program, all at the same time. You can get online help while looking at the actual state of your program in other windows. The window interface, described in Chapter 2, is available on graphics terminals running CXwindows and on ASCII terminals running Maryland Windows.

CXdb enables you to work with your source code in source units, such as routines, blocks, loops, and expressions. Source units give you much more control than just the traditional notions of line numbers and statements. Source units are discussed in Chapter 3.

Optimized code introduces an extra level of complexity when debugging. CXdb correctly relates the effects of scalar optimization found in the executable file back to your original source code. Optimized code concepts are discussed in Chapter 5.

Comparison to other debuggers

This section describes CXdb's traditional debugging capabilities. It also describes CXdb's extra capabilities and features that make it easier to use.

Traditional debugging capabilities

Like most other debuggers, CXdb gives you control over the execution of your process. A process is the running version of your program. With CXdb, as with traditional debuggers, you can:

- Create a process from an executable file
- Examine a core file
- Start, stop, and continue process execution
- Examine and modify program variables, registers, and the stack
- Set breakpoints where you want execution to stop
- Step process execution line-by-line
- Debug at the source level or the disassembly level
- Modify the environment in which your process runs

Extra capabilities

CXdb has many extra capabilities that give you more power and control when debugging your code. Some of these are enhancements to traditional capabilities. Others are new capabilities found only in CXdb. With CXdb you can:

- Debug a process that is already running
- Execute debugger commands while your process is running
- Fully debug code that has been optimized up to the `-O1` level
- Control execution by source unit
- Set traps, called eventpoints, to stop execution when the value of a variable changes, a signal is caught, or an expression becomes true
- Specify a complete set of actions to take when a particular eventpoint stops execution, including resuming execution

- Debug at the machine instruction level, with complete access to the machine state including scalar, vector, and, on the C2, the communication registers
- Create aliases and macros for commonly used commands
- Display 2-dimensional arrays as a table
- Debug programs that are written in both FORTRAN and C
- Access static program variables that are not visible from the current point of execution
- Use debugger variables to store information without affecting program variables
- Debug, edit, and recompile your program without leaving the CXdb environment
- Create command files to automate frequently performed tasks
- Configure CXdb to meet your debugging needs

Special features

CXdb also has several features that make it easier to use the powerful capabilities described above. These features include:

- Multiple windows that allow you to view your program in different ways at the same time
- An online help system that gives you the entire *CONVEX CXdb Reference* at your fingertips
- Access to your default editor from within CXdb
- Complete control over the logging of command input, output, and error messages, which can even be used to create command files
- The highlighting of the current point of execution in the display of your source code
- Graphic symbols representing different eventpoints in the displays of your source code and disassembly code
- A menu system for the CXwindows interface that enables you to issue CXdb commands using a mouse
- An online guide for the CXwindows interface that teaches the basics of using CXdb

Debugging concepts

There are many concepts associated with debugging in CXdb. These concepts are grouped according to complexity and are outlined below.

Fundamental concepts

Fundamental debugging concepts are involved in most typical debugging tasks. These concepts, covered in Chapter 3, include:

- **The CXdb working environment**—The environment in which you work when using CXdb. You can tailor it to suit your debugging needs.
- **The program working environment**—The environment in which your program lives in CXdb. You can tailor it to suit the needs of your program.
- **Source units**—The syntactical units of source code. When you are debugging your program at the source level rather than the disassembly level, you can use source units in addition to line numbers.
- **Stepping**—The executing of your process step-by-step. You can step by source unit to get very precise control over how much of your program executes each time.
- **Predefined eventpoints**—Breakpoints, tracepoints, and watchpoints that perform traditional actions when a particular event occurs.

Advanced concepts

Advanced debugging concepts involve the more powerful capabilities of CXdb. These capabilities offer more flexibility in debugging in addition to giving you greater control over your program.

The advanced concepts, covered in Chapter 5, include:

- **Command and initialization files**—The files consisting of CXdb commands to execute. Initialization files are command files that are executed automatically.
- **Eventpoints and handlers**—The general eventpoints that allow a greater range in the types of events you can trap. Handlers are the set of commands that execute when a particular event occurs.
- **Signals**—The interrupts CXdb catches that are sent to your program by the operating system. You can choose what actions CXdb takes when it catches a signal.

- **Debugger variables**—The variables that you can create to store information about your program or CXdb.
- **Aliases and macros**—The shortcuts you create for commands you frequently use.
- **Process settings**—Settings that control various aspects of your process.
- **Logging**—The logging of command input, output, and error messages.
- **Scope paths**—The paths identifying program variables that are not visible from the current point of execution.

Optimized code concepts

The final chapter describes concepts related to debugging optimized code.

Optimized code differs from normal code in several ways. The compiler can move your instructions around during optimization and can modify your code to take advantage of the CONVEX machine architecture. Traditional debugging practices are not as effective with optimized code. CXdb provides unique functions to deal with these differences.

These concepts, covered in Chapter 6, include:

- **Optimized code**—A description of optimized code at the various levels of optimization.
- **Source units and optimization**—The relationship between source units and optimized code.
- **Debugging optimized code**—Techniques used to debug optimized code.
- **Eventpoints and optimization**—The special functions for setting eventpoints in optimized code.
- **Stepping through optimized code**—Stepping through optimized code at the instruction level or expression level.
- **Threads**—The debugging of a process with multiple threads.

With a grasp of these concepts you can effectively debug your code that has been optimized up to the `-O1` level. Code that has been optimized at higher levels may still be debugged with CXdb, but the resulting information about your program may not be as accurate.

Online information

CXdb has documentation available online. Online material consists of an extensive help facility and, for the CXwindows interface, a guide to using CXdb. These two facilities are intended to provide you with help when you need it most—while you are using CXdb.

A brief description of each of the facilities is presented below. The use of these facilities is fully described in the *CONVEX CXdb User's Guide*.

Help facility

The online help facility enables you to get a separate page of information about CXdb commands, concepts, parameters, and messages. You can get request help on any topic from the following categories:

- **Commands**—In addition to description, syntax, and example usage, each page lists the command's related topics, shortest abbreviation, and alias (if it exists).
- **Concepts**—A complete description of the concept, plus a situational example explaining the use of CXdb commands that relate to the concept. Related topics are also listed.
- **Parameters**—A description of command parameters of a more complex nature. Examples demonstrate various uses of the parameter. Related topics are also listed.
- **CXdb messages**—A description of each message that CXdb may generate, including information messages and all error messages. Where possible, suggestions are given for correcting an error. CXdb messages are identified by their number.

If you are using the CXwindows interface, you can select help topics from a list of available topics. You can also give CXdb a string to look for in all available topics.

Online guide

Note

The online training guide is available to you only if you are using the CXwindows interface.

The online guide covers various aspects of CXdb. The guide consists of a window that lets you page through any of the following materials:

- Using the online guide
- Overview of CXdb concepts and features
- Sample session on basic debugging
- Sample session demonstrating eventpoints
- Sample session demonstrating stepping
- Sample session on advanced debugging

Along with the online guide are some example programs. These programs match the programs debugged in the sample sessions. You can use CXdb to enter the commands shown in the sample session to debug the example programs while reading the session.

Requirements

There are several requirements to run CXdb. These are:

- Running ConvexOS V9.0 (or subsequent releases)
- Compiling your program with either the CONVEX FORTRAN V6.1.3 or CONVEX C V4.1 compilers (or subsequent releases)
- Running CXwindows 2.1 if you want to use the CXwindows interface (or subsequent releases)

The Debugger Environment 2

CXdb is a visual debugger that is designed to operate through windows on your terminal screen. This chapter describes the windows available in CXdb and gives examples of how they look.

Interfaces

There are three types of user interfaces available for CXdb:

- CXwindows
- Maryland Windows (supplied with CXdb)
- Batch mode

The interface determines which windows can be displayed and how they look. The type of interface you can use depends on the platform you are using to run CXdb. CXwindows runs on a variety of graphics terminals that support X Window. Maryland Windows runs on ASCII terminals or on X Window terminals. Batch mode works on both types of terminals.

The following sections describe the three types of interfaces.

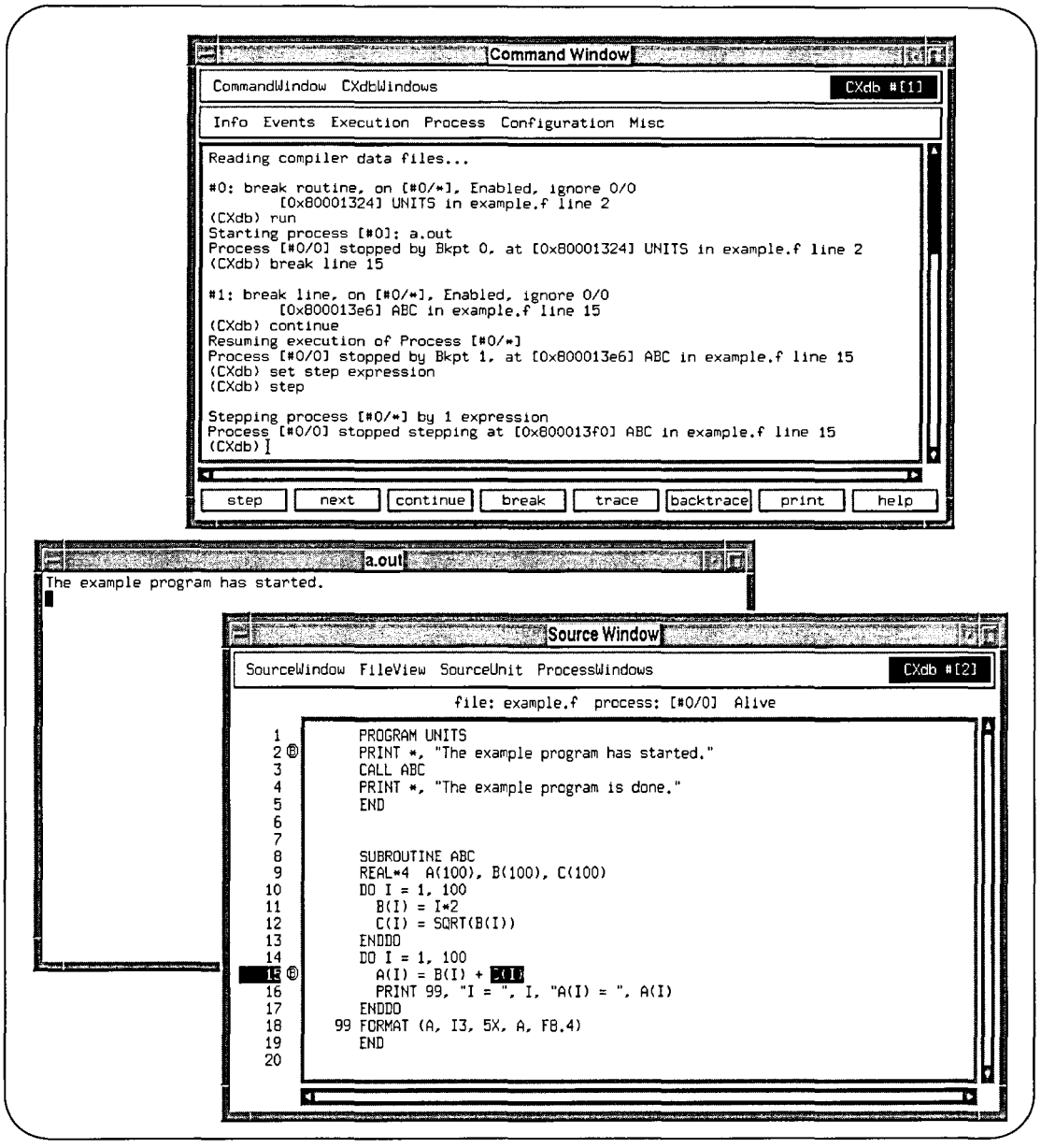
CXwindows

CXwindows is the CONVEX version of the X Window System, which was originally developed by the Massachusetts Institute of Technology. CXwindows is an optional software package that can be purchased separately from CXdb. The CXwindows interface 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
- Online Guide with sample debugging sessions

Figure 1 shows how a CXdb session might look in CXwindows.

Figure 1
CXdb running in CXwindows



CXdb provides a set of default values to define the CXwindows environment. These defaults control settings such as window sizes and locations. You can change these defaults to customize the window environment to suit your particular needs.

Maryland Windows

Maryland Windows creates a window environment for ASCII terminals. It does this by dividing the screen into segments, with each segment representing a different window.

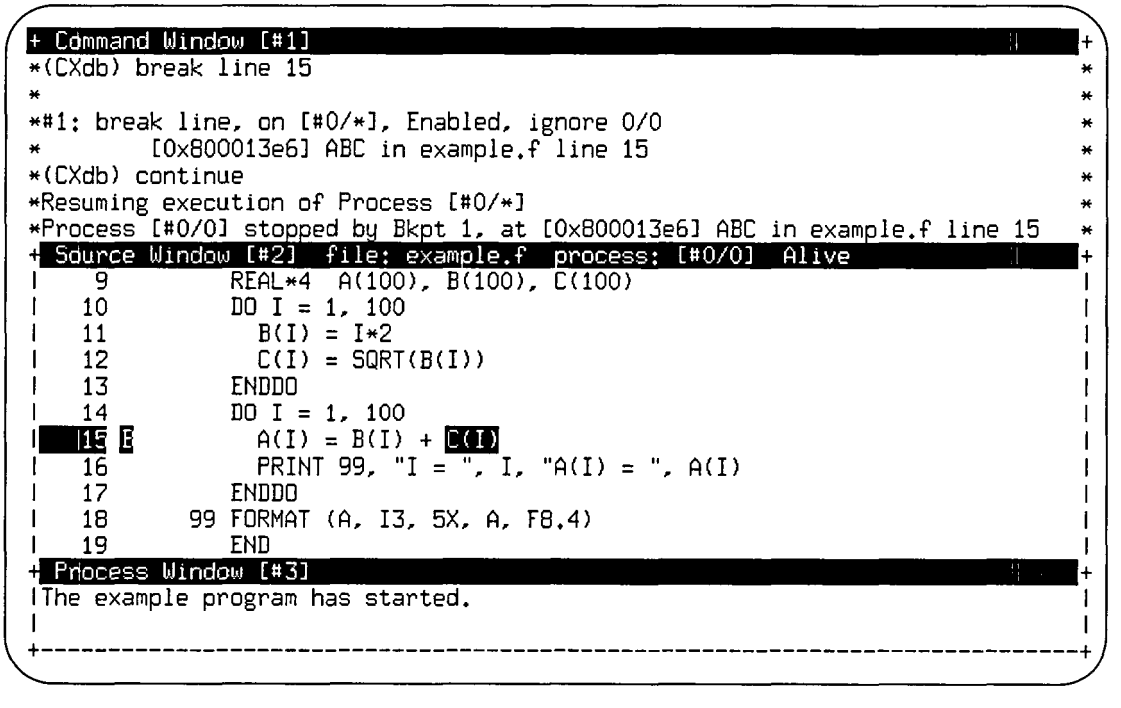
ASCII terminals cannot provide all the mouse capabilities of graphic terminals. However, for the Maryland Windows interface, CXdb provides keyboard functions that allow you to:

- Move, resize, raise, and lower windows
- Scroll the contents of a window
- Move the cursor from one window to another
- Cut and paste text between windows

There are default function keys for performing all of the above functions, but you can also redefine these function keys.

Figure 2 shows how a CXdb session might appear in Maryland Windows. This figure shows the same session displayed in Figure 1 for CXwindows.

Figure 2
CXdb running in Maryland Windows



Batch mode

You can also run CXdb in batch mode. In this mode, there is no interactive user interface as there is with CXwindows or Maryland Windows.

In batch mode, CXdb executes all the commands that you specify on the command line. It also executes all commands in the CXdb initialization file and in specified CXdb command files, if those files exist. (Refer to Chapter 4 for a description of command files and initialization files.) You can direct any output or error messages from the executed commands to specified files. After executing the commands, CXdb exits.

Primary windows

With both the CXwindows and Maryland Windows interfaces, there are three main windows that appear on your screen:

- **Command window**—The primary way to communicate with CXdb.
- **Source window**—The display of your source code.
- **Process interface window**—The means of interacting with your program.

The following sections describe the purpose of each of these windows.

Command window

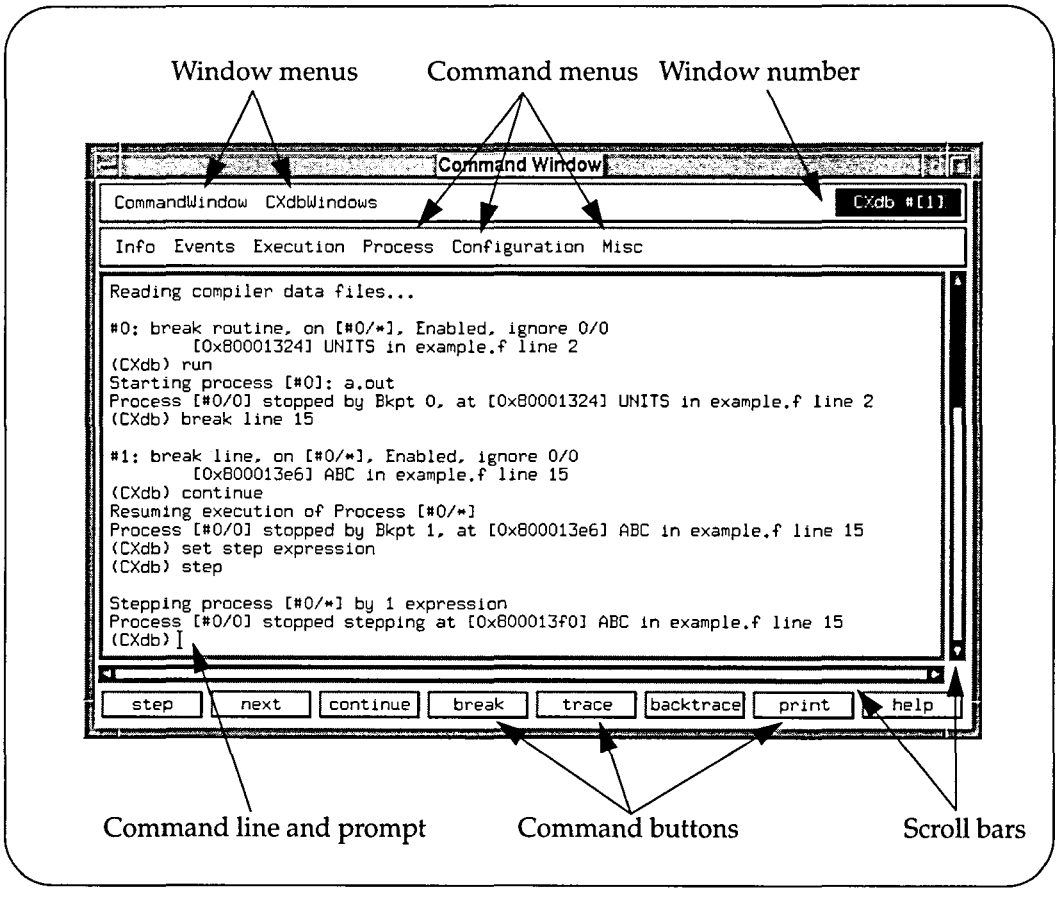
The command window is the primary way for you to communicate with CXdb. It enables you to:

- 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

The command window comes up automatically whenever you invoke CXdb in CXwindows or Maryland Windows. In CXwindows, the command window has some special features such as mouse-activated buttons for frequently used commands and menus for controlling other windows and entering CXdb commands.

Figure 3 illustrates a typical command window in the CXwindows environment.

Figure 3
Command window in CXwindows



Source window

The source window displays the source code for your program. The source window appears automatically whenever you give CXdb the name of an executable file that has been compiled with the `-cxdb` option. Because the entire source listing is usually too long to fit in one window, the source window scrolls automatically to the current point of execution and displays the source code surrounding that point. To view the rest of the source code, you can scroll the source window.

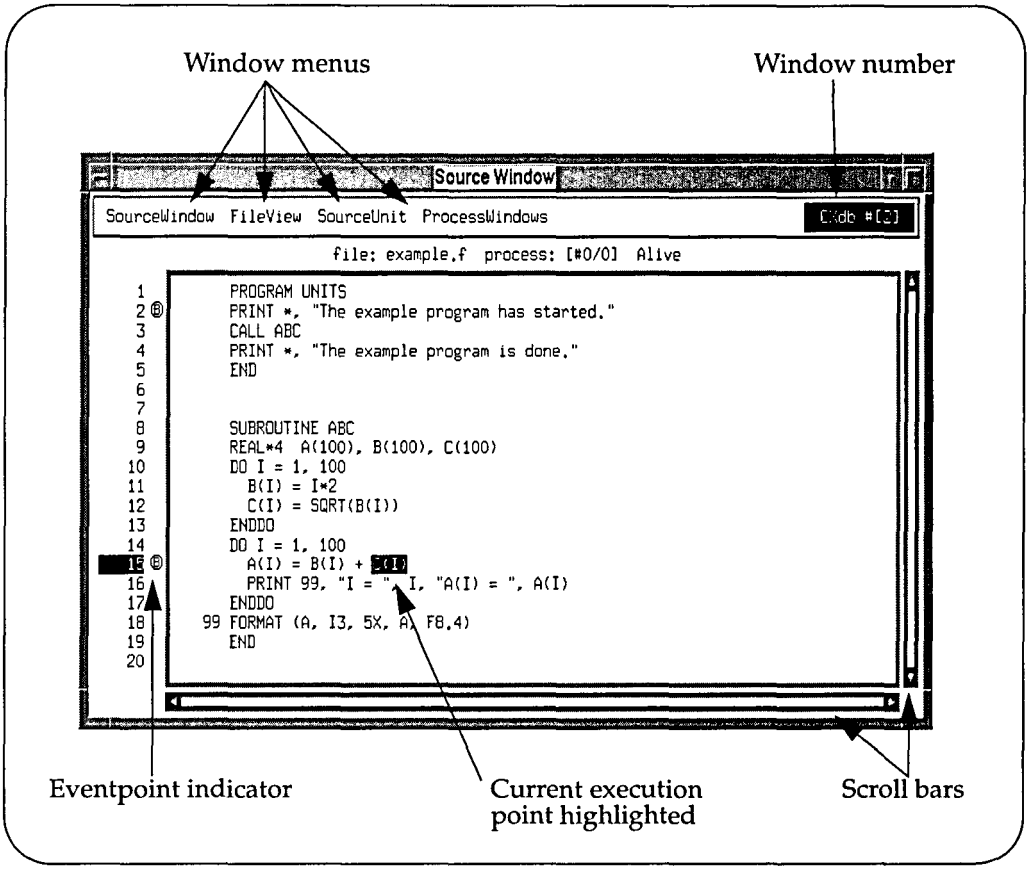
CXdb lets you open multiple source windows at the same time. This enables you to view different parts of the same source file, or several different source files, simultaneously.

When your program execution is halted, CXdb highlights the location in the source code where execution stopped. The source window also displays indicators to show the locations of eventpoints.

In addition to displaying information about the source code, the source window lets you use the mouse (CXwindows only) to select source units and set specific eventpoints. Chapters 3 and 4 of this book explain source units and eventpoints.

Figure 4 shows an example of a source window in the CXwindows environment.

Figure 4
Source window in CXwindows

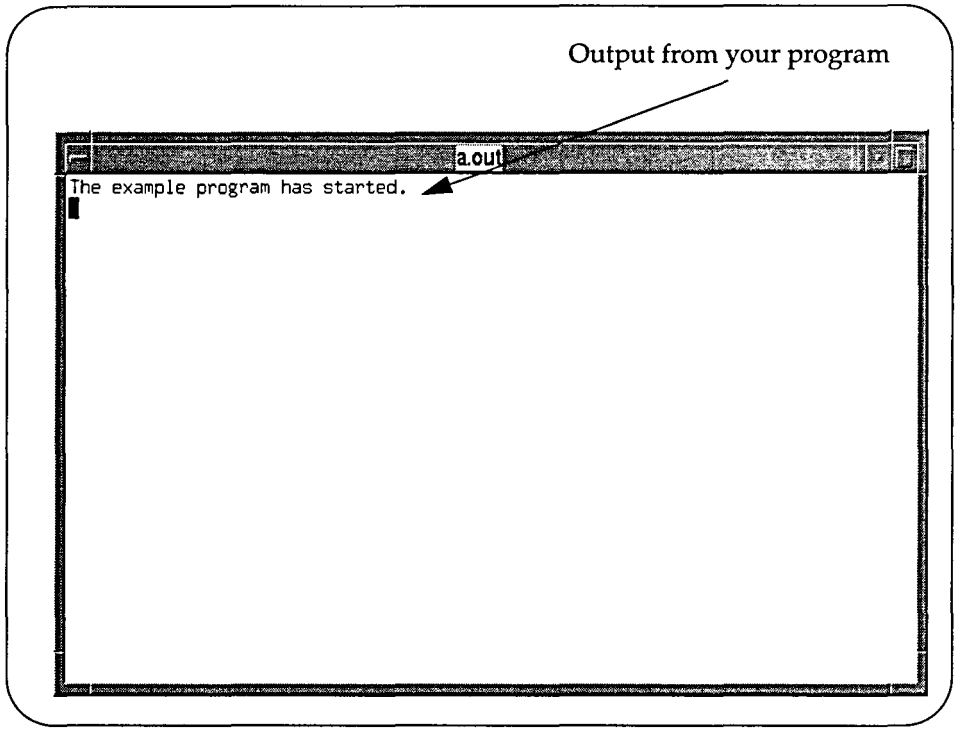


Process interface window

The process interface window enables you to read output from and give keyboard input to your program. The process interface window comes up automatically when your program begins to execute. This feature keeps interaction with your program separate from interaction with CXdb.

Figure 5 illustrates a typical process interface window in CXwindows.

Figure 5
Process interface window in CXwindows



CXdb commands enable you to select the type of shell used as the process interface window. In CXwindows, the process interface window is a separate window. This separate window is an xterm window, which emulates a terminal running under CXwindows. In Maryland Windows, it is another subdivision of your terminal screen.

Optional windows

CXdb provides many other windows for displaying additional information that can make debugging easier for you. These windows are optional in the sense that they do not appear unless you explicitly invoke them.

The optional windows let you display:

- Disassembled code (CXwindows only)
- The process stack (CXwindows only)
- Process memory (CXwindows only)
- The process registers (CXwindows only)
- Online help (CXwindows and Maryland Windows)

The following sections describe these windows.

Disassembly window

The disassembly window displays the disassembled code for the executing process. The disassembled code is the assembly language version of the program.

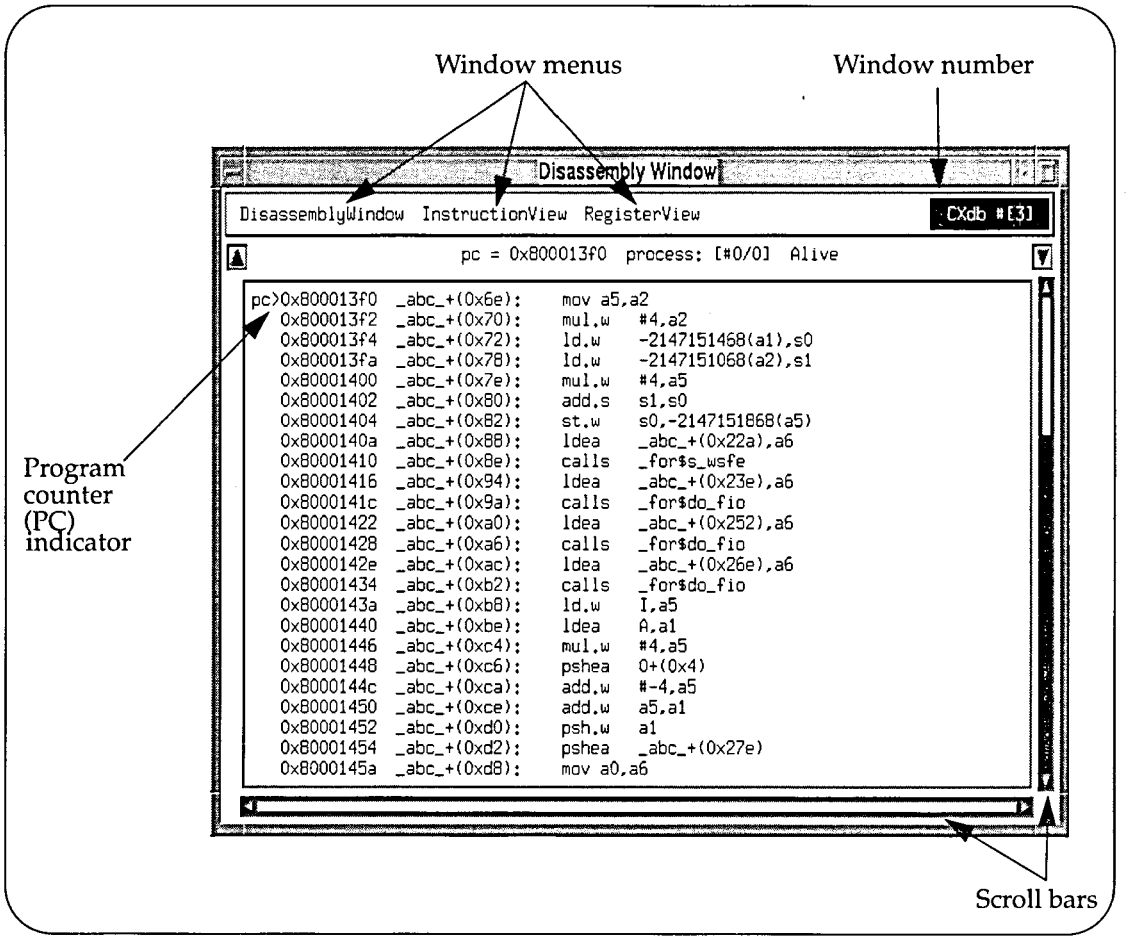
CXdb operates from the executable (binary) file of the program. However, it generates the disassembled code from the executable file so that you can use the disassembled code as an aid to debugging. This is especially important when debugging optimized code because optimization can change the linear relationship between the source code and the resulting machine instructions. The disassembled code always shows the exact order in which instructions are executed by the computer, even after optimization. But the source code shows only the logical order in which you programmed the instructions before optimization.

For both CXwindows and Maryland Windows, you can use commands to display the disassembled code in the command window. However, in CXwindows you can also open a separate window to display the disassembled code. You can open multiple disassembly windows to display different sections of code at the same time.

You can set the disassembly window to be static or to update automatically. In automatic update mode, CXdb tracks the current point of execution and updates the window when execution stops. Automatic update is the default setting.

Figure 6 shows an example of the disassembly window in the CXwindows environment.

Figure 6
Disassembly window in CXwindows



Stack window

For both CXwindows and Maryland Windows, you can use commands to display the process stack and stack frames in the command window. However, in the CXwindows environment, CXdb also provides a special window for viewing the process stack. You can open multiple stack windows at the same time.

The process stack consists of frames that are pushed onto the stack with each subroutine call or function call. Each frame stores the context, or state, of the calling routine. When execution returns to the calling routine, its frame is popped from the stack.

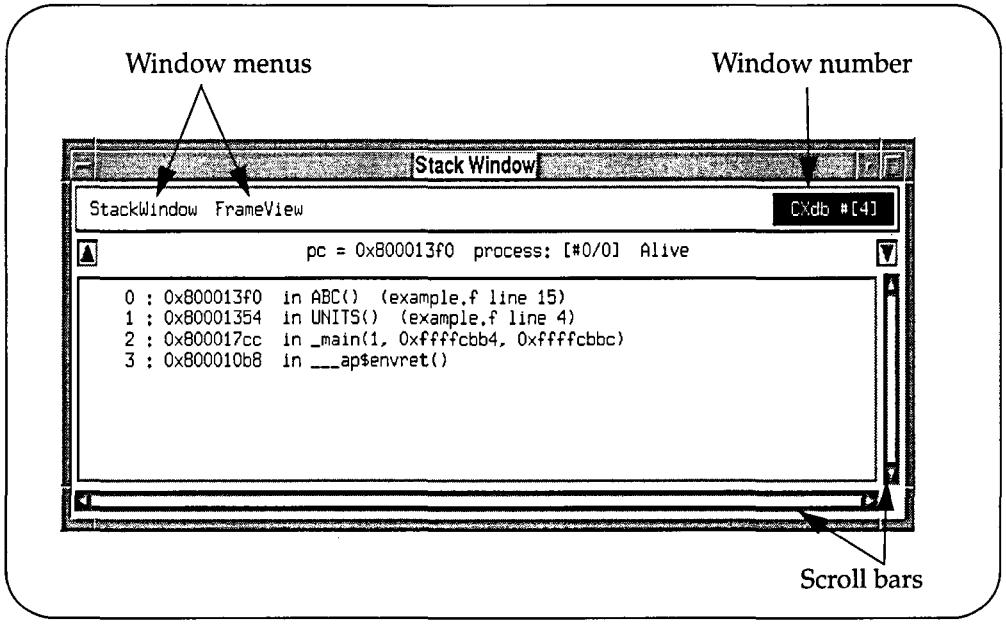
The stack window displays the following information, if applicable:

- A marker to indicate the current frame in the stack
- The frame number of each frame
- The execution address for each frame
- The name of the routine associated with each frame
- The name of the source file that contains the routine for each frame
- The line number from the source code that corresponds to the execution address for each frame

You can set the stack window to be static or to update automatically. In automatic update mode, CXdb tracks the current point of execution and updates the window when execution stops. Automatic update is the default setting.

Figure 7 illustrates the stack window in CXwindows.

Figure 7
Stack window in CXwindows



Examine window

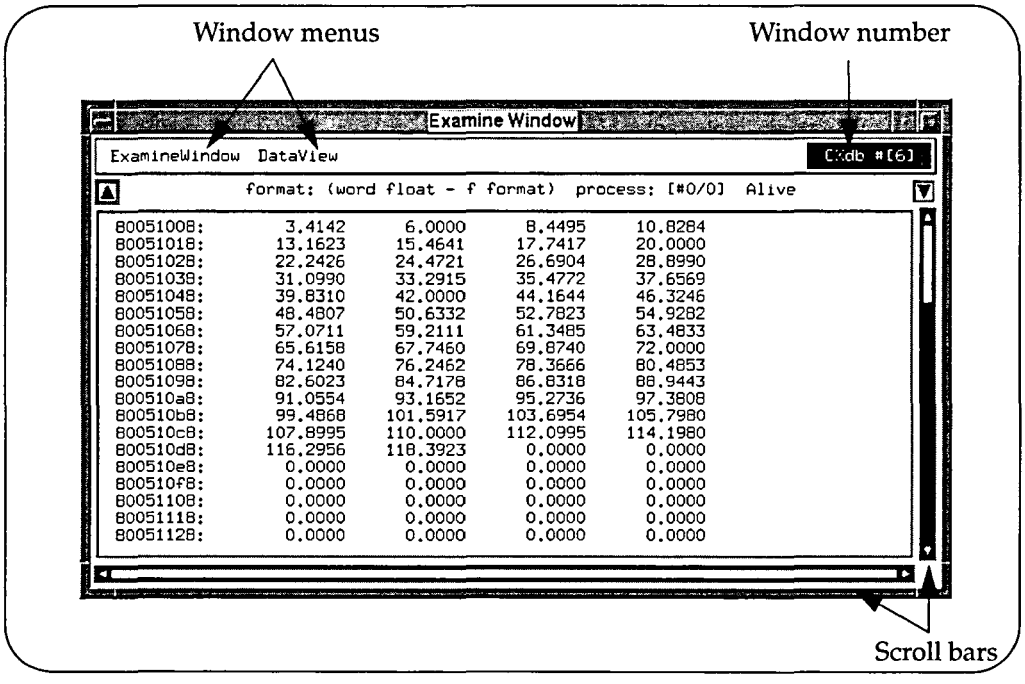
For both CXwindows and Maryland Windows, you can use commands to display the contents of process memory in the command window. However, in the CXwindows environment, you can also display memory in a special window called the examine window. With the examine window you can watch a segment of process memory, such as part of an array, change as you step process execution.

You can open multiple examine windows and look at different sections of memory at the same time. Each line of the examine window shows a memory address and 4 words of memory starting at that address. Using menu options, you can change the display format (for example, binary, decimal, hexadecimal, etc.).

You can set the examine window to be static or to update automatically. In automatic update mode, CXdb tracks the current point of execution and updates the window when execution stops. Automatic update is the default setting.

Figure 8 is an example of the examine window in CXwindows being used to show the values of the array A.

Figure 8
Examine window in CXwindows



Register windows

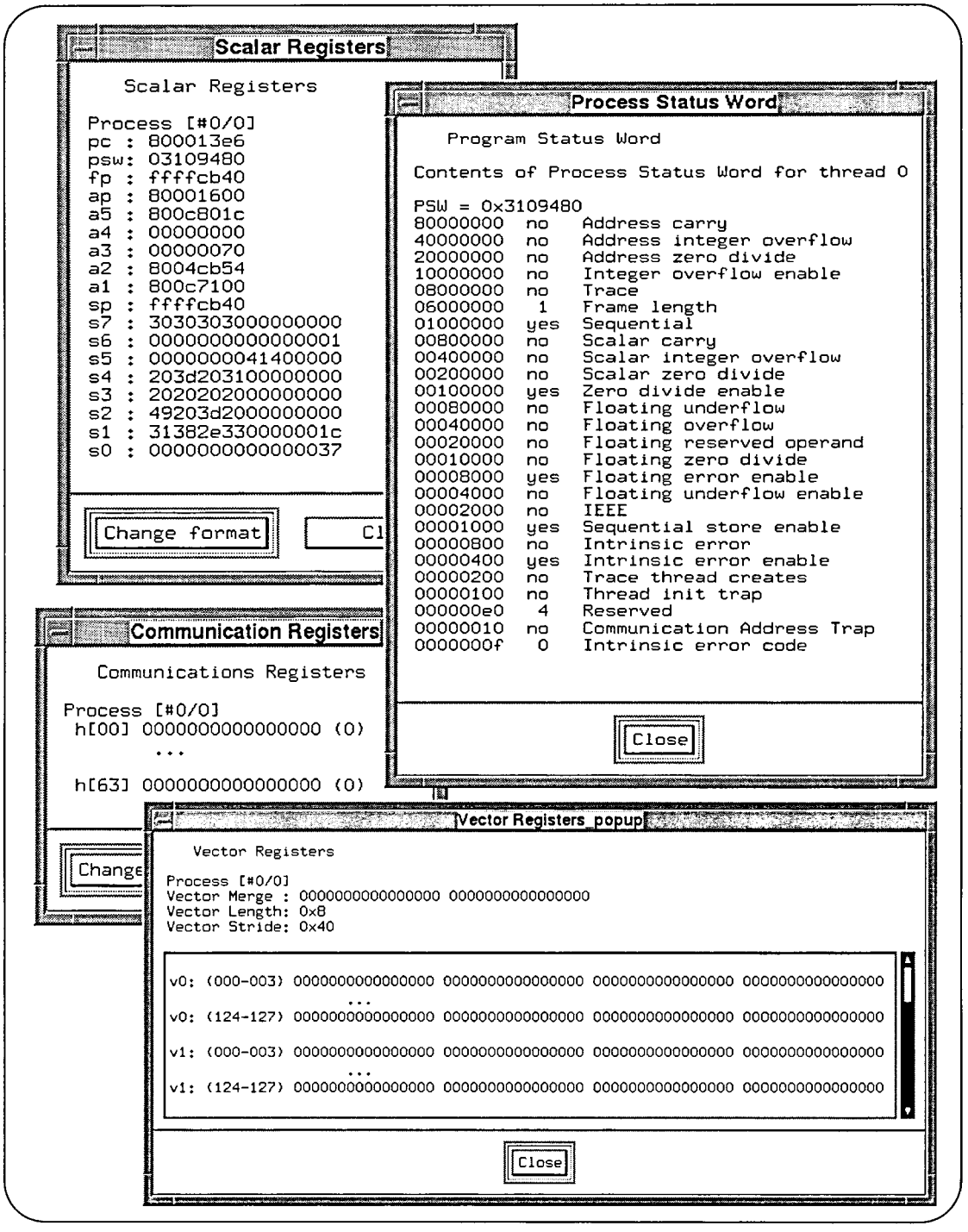
For both CXwindows and Maryland Windows, you can use commands to display the contents of the process registers in the command window. The registers that you can display are:

- Scalar registers (S0 to S7)
- Address registers (A0 to A7)
- Program counter (PC)
- Processor status word (PSW)
- Ring 4 communication registers (H00 to H63), for C200 Series machines only
- Vector registers (V0 to V7)

In the CXwindows environment, you can also display the registers in their own special windows. These register windows appear when you use the mouse to invoke them from the menu in the disassembly window. The register windows update automatically as the contents of the registers change.

Figure 9 is an example of the register windows as they would look for a C200 Series machine. All of the windows are from CXwindows.

Figure 9
Register windows (C200 Series) in CXwindows



Help window

For both CXwindows and Maryland Windows, the help window displays online help text relating to various CXdb topics. The help window appears when you use the `help` command to invoke help from the command window.

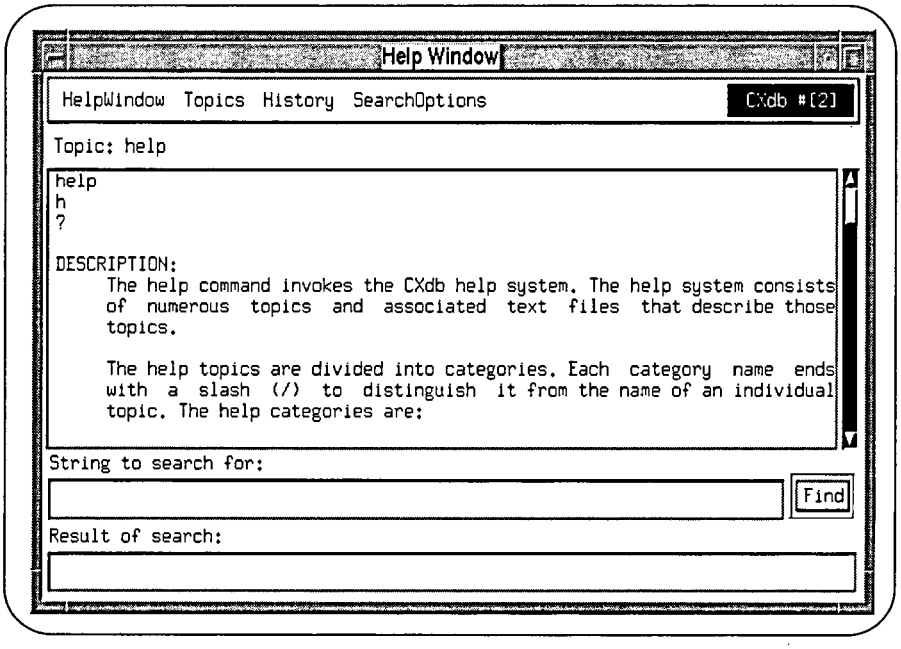
The online help system contains all of the topics in the *CONVEX CXdb Reference* manual. These topics cover four categories of information:

- **Concepts**—Explanations of the major topics involved with using CXdb.
- **Commands**—Descriptions of all the CXdb commands.
- **Parameters**—Descriptions of some of the major command parameters.
- **CXdb messages**—Explanations of informational messages and error messages generated by CXdb.

You can either request help on a listed topic, or you can search the help files to locate a particular word or phrase.

Figure 10 shows an example of the help window in CXwindows.

Figure 10
Help window in CXwindows



Fundamental Concepts 3

This chapter explains fundamental concepts. These concepts are involved in most debugging tasks. Some of these concepts are new to CXdb, others are extensions of traditional debugging concepts. The following concepts are covered:

- CXdb working environment
- Program working environment
- Source units
- Stepping
- Breakpoints, tracepoints, and watchpoints

CXdb working environment

The CXdb working environment is the environment that you work in when using CXdb. You can change the different parts of the environment to meet your debugging needs.

The CXdb working environment consists of:

- Console working directory
- Default process settings
- The settings for command logging

Each of these are further described below.

Console working directory

The console working directory is the current directory from which you are working. You can move around your directory tree just as you would from the shell.

CXdb uses the console working directory as the base path for all relative path names that you give CXdb. You can use relative path names when giving CXdb the names of files or directories.

You can change the console working directory to find your executable files and data files. You can also find these files by adding their directories to a list of directories to search.

Default process settings

CXdb keeps track of several settings that control different aspects of your process, such as the mode it uses when doing floating point calculations. These settings are called process settings. The default process settings are the values given to the process settings each time you debug a program in CXdb. If you frequently debug the same program, or the same types of programs, you can change the default process settings to automatically set the values for each program, rather than having to manually modify the settings each time.

The easiest way of setting the default process settings is through the use of an initialization file. For example, to ensure that every program you debug in CXdb uses IEEE mode for its floating point calculations, you can set the default process setting for floating point mode to IEEE in an initialization file. Then, each new program that you debug with CXdb will use IEEE mode.

The process settings are described later in this chapter in the section "Process settings."

Command logging

While you are debugging with CXdb, you can keep a log of everything you do. There are three separate types of information that you can log: commands entered, command output, and CXdb messages.

By controlling where these three types of information go, you can keep specific logs of your debugging sessions. For more information about logging, refer to Chapter 4, "Logging."

Program working environment

The program working environment is the environment that controls the program you are debugging. You can tailor it to the needs of your program.

The program working environment consists of:

- Executable file and compiler-generated data files
- Process image
- Process working directory
- Search path
- Process settings

The settings of the program working environment are stored in a structure called the process object. It is created when you tell CXdb you want to debug an executable file, a core file, or an existing process.

To debug a different program with CXdb, you give CXdb the name of a different executable file. The old information in the process object is removed, and the new information is stored.

Executable file and data files

The executable file is the binary file generated by the compiler when you compile your program.

To be able to fully debug your program with CXdb, you must compile your program using the CONVEX FORTRAN (V6.1.3 or later) or CONVEX C (V4.1 or later) compiler using the `-cxdb` option.

When you use the `-cxdb` option, the compiler generates data files that describe the mappings between the executable file, program symbols, and source files. CXdb uses these files to map the binary data in the executable file to the source code of your program. There is a list of directories, called a search path, where CXdb looks for the data files of an executable file. If they are not located in a directory on the search path, CXdb does not find them. This limits symbolic debugging to global identifiers.

By storing the debugging information in separate data files, the executable file remains the same size and executes at the same speed had you not specified the `-cxdb` option.

If your program was not compiled with the `-cxdb` option, then the data files for your executable do not exist. Without the data files, CXdb cannot completely match the executable file back to the source file. However, you can still debug the program by using global program symbols and absolute addresses.

If you have compiled the executable file with an optimization level higher than `-O1`, you can still debug your program with CXdb. However, the data files associated with the executable file may not be completely accurate, and thus, the information you get about your program may at times be incorrect.

Process image

When the operating system runs your program, it creates a process. A process image is a snapshot of the current state of the process. CXdb needs a process image to enable you to debug your program. You can give this image to CXdb by using one of the following three methods:

- Run the program from within CXdb
- Attach CXdb to an already running process
- Give CXdb the name of a core file or a checkpoint file

You can run your program within CXdb only if you have given CXdb the name of the executable file. CXdb creates a process from the executable file, and it associates the process image with the debugging information in the executable file and data files.

You can attach CXdb to an already running process. CXdb takes the image from this process and associates it with the executable file, if the executable file has been specified. The process stops, and you can begin debugging with CXdb. You cannot start this process from the beginning, but you can continue its execution. As with all processes under CXdb's control, you can detach from the process when you finish debugging. If you detach from a process, the process continues to run outside the control of CXdb.

You can give CXdb the name of a core file or checkpoint file to debug. A core file contains a static image of a process that abnormally terminated when an exception occurred. A checkpoint file is created by the shell utility `chkpnt`, and it contains a static image of a process in the middle of execution. In either case, CXdb again associates the image with the information in the executable file, if the executable file has been specified. You can use CXdb to examine the image. You cannot execute the image from a core file because the image is from a terminated process. You cannot execute the image from a checkpoint file because this can only be accomplished with the shell `restart` utility.

Process working directory

The process working directory is the directory from which your process is run. The process working directory defaults to the console working directory. Because CXdb runs your program from the process working directory, this directory serves as the base path for all relative path names found in your program.

If your program needs to start from a specific directory, other than CXdb's console working directory, you can change the process working directory so execution will begin in the proper directory.

Search path

The search path is a list of directories that the debugger searches when it needs to find the source files or compiler-generated data files that correspond to the executable file. The search path defaults to the current console working directory and the directory where the executable file was found (if different).

You can modify the search path to include the directories where the source files and data files are located. It is important to add these directories to the search path if they are not yet part of the search path. If you do not add them, CXdb will not be able to find the necessary files.

CXdb searches the directories listed in the search path in the order they are listed, and it uses the first file with the appropriate name.

Process settings

The process settings control different aspects of your process, such as the formats to use when displaying process memory. When the process object is created, the process settings are set to the values of the default process settings found in the CXdb working environment.

You can change the process settings of the program working environment and affect all new processes created from the executable file. This enables you to affect how each new process runs without having to set up each process separately.

If you want to change the settings for only the current program you are debugging, then change the process settings in the program working environment. If you want to change the settings for each new debugging session in CXdb, then change the default process setting in the CXdb working environment by using an initialization file.

The process settings are described below.

- **Process shell**—The shell in which the process is created.
- **Process environment**—The shell environment variables inherited by the process when it is created.
- **Floating point mode**—The floating point mode (fpmode), used by the process when it does floating point calculations.
- **Fixed scheduling**—The setting (enabled or disabled) for fixed scheduling. If fixed scheduling is enabled, the process will not begin execution until all CPUs are available.
- **Step size**—The source unit size, called granularity, used when stepping process execution.
- **Display format**—The format used when displaying process memory.
- **Memory format**—The memory unit used when displaying process memory.
- **seq and sqs bits**—The setting (enabled or disabled) for the seq and sqs bits. These bits control pipelining and memory stores, respectively.

For more information about the above process settings, refer to the *CONVEX CXdb User's Guide.*

Source units

A source unit is a syntactical subdivision of the source code. Source units are classified by granularity, as follows:

- **Expression**—A combination of one or more constants, operators, and operands. Expressions may be nested within other expressions.
- **Statement**—A combination of one or more expressions that constitutes a complete instruction in the 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 of code. Execution of a loop can be controlled by any of the looping constructs available in the source language.
- **Routine**—A subroutine or function.

CXdb uses source units for many of its operations. This is the primary way in which CXdb differs from other debuggers. Most debuggers base their commands on line numbers or statement numbers in the source code, but CXdb bases its commands on source units.

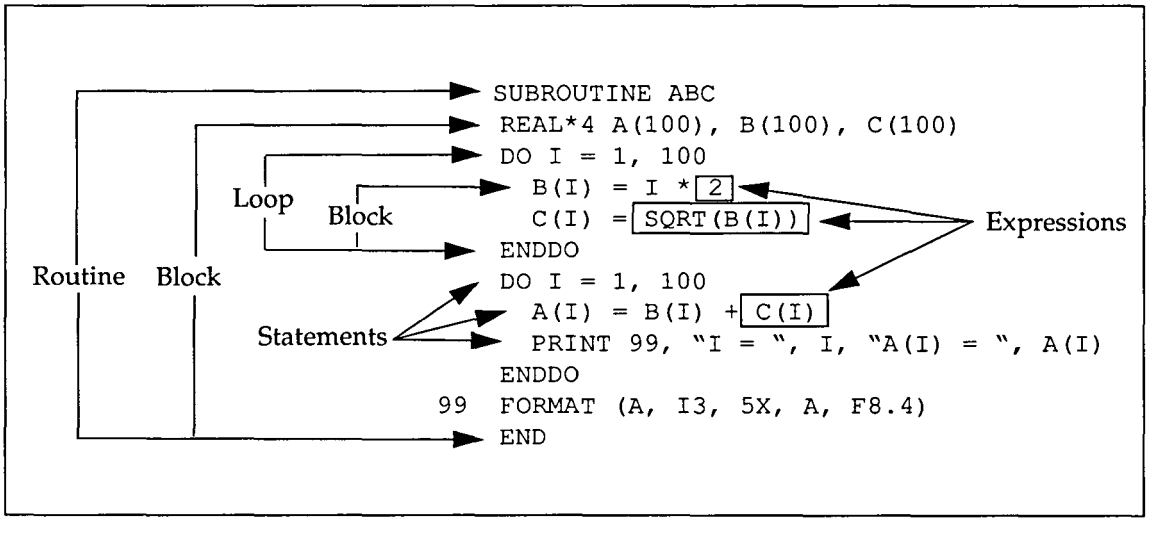
Many CXdb commands let you specify the source unit granularity that you want to use. This gives you much greater resolution and control during the debugging session. For example, when stepping through a program, you can select the step size by specifying a source unit granularity. Thus, with CXdb you can step the program by routine, loop, block, statement, or expression. However, with most other debuggers, you can step the program only by statement.

Controlling the source unit granularity also helps you to visualize the execution of your program. For example, by changing the default source unit granularity, you can adjust the amount of detail shown by the highlighting in the source window.

The CONVEX FORTRAN compiler (V6.1.3 or later) and CONVEX C compiler (V4.1 or later) identify the various source units as they compile the source code. The compilers also assign an identification number to each source unit. These source unit numbers are unique within a given source file. You can then use the identification number in a CXdb command to specify the particular source unit you want the command to affect.

Figure 11 illustrates the different types of source units. Keep in mind that the figure does not show every individual source unit in the listed section of code. Rather, the figure merely shows a few examples of the different source unit granularities.

Figure 11
Source units in FORTRAN



As Figure 11 shows, source units of different sizes are nested inside each other. For example, a loop can contain many statements, expressions, blocks, and even other loops.

In addition, several different source units can start at the same location at the same time. For example, in Figure 11, the line `B(I)=I*2` is the beginning of a block source unit. It is also a statement source unit, and it contains a number of expression source units such as `I`, `2`, and `I*2`.

Stepping

Stepping is the incremental execution of your program. CXdb provides a sophisticated set of stepping commands that allow you to control not only the number of steps to execute but also the size of each step.

CXdb lets you step the program either by machine instructions or by source units. If you choose to step by source units, then you can also select the granularity, or step size. In addition, stepping by source units enables you to do any of the following:

- Step to the beginning of a particular routine, loop, block, statement, or expression
- Step out of (finish) a particular routine, loop, block, statement, or expression
- Step into a particular routine, loop, block, statement, or expression
- Step over a particular routine, loop, block, statement, or expression
- Step to the beginning of a particular source unit within a called routine
- Step to the beginning of the next source unit in the current routine, ignoring any called routines along the way

In each of the above cases, you can specify the number of steps to take as well as the source unit granularity.

Predefined Eventpoints

Breakpoints, tracepoints, and watchpoints are predefined types of eventpoints. Eventpoints are traps that you can place in your executable code to wait for a particular event to occur. When the event occurs, the corresponding eventpoint is said to be reached.

Eventpoints can detect when execution reaches a specified address, when the process receives a signal, when a value in memory changes, or when a relational expression becomes true. To make it easier for you to trap the most common types of events, CXdb provides the following predefined types of eventpoints:

- **Breakpoints**—Stop the process when execution reaches a specific location.
- **Tracepoints**—Notify you when execution reaches a specific location, but do not stop execution.
- **Watchpoints**—Stop execution when the value of a variable or the contents of a memory location changes.

These predefined types of eventpoints are described below.

Breakpoints

Breakpoints are traps you place in your code where you want CXdb to halt the execution of your process. CXdb uses breakpoints similarly to traditional debuggers. When the location of a breakpoint is reached, execution stops and control returns to you. But, with the addition of source units, you have more flexibility in inserting the breakpoint.

There are four ways of inserting a breakpoint. These are:

- **Breaking at a line number**—The breakpoint is placed at the beginning of the line. This is the traditional method of setting breakpoints. Execution stops before the line is executed.
- **Breaking at a source unit**—The breakpoint is placed at the first instruction of a source unit. Execution stops before the source unit is executed. This allows you to stop execution in the middle of an expression, rather than having to stop execution before or after the entire line is executed.
- **Breaking at a routine address**—The breakpoint is placed before the first executable source unit of the routine that contains the address. By setting a breakpoint this way, you can stop execution before any source units in a routine are executed.
- **Breaking at an instruction address**—The breakpoint is placed at the beginning of the instruction at that address. Execution stops before the instruction is executed.

Tracepoints

Tracepoints are very similar to breakpoints. When execution reaches a tracepoint, CXdb displays a message telling you that the tracepoint was triggered. Execution continues past the tracepoint.

Tracepoints are very useful in tracking the execution of your program. You can set tracepoints in the same four ways as breakpoints. With tracepoints, however, execution continues after CXdb informs you that the process has reached the specified location.

Watchpoints

Watchpoints are not placed at specific locations in the code. Instead, a watchpoint is set to watch a region of process memory, such as the region that holds the value of a variable. When the value stored in the region changes, the watchpoint is triggered.

Note

Watchpoints increase execution time significantly because CXdb checks the specified memory region after execution of each statement.

Watchpoints enable you to run your program until something changes. If the value of a variable is changing incorrectly, but you do not know when or where this is occurring, a watchpoint can help you find the problem.

Advanced Concepts 4

This chapter explains concepts related to the advanced capabilities of CXdb. The intent of this chapter is to give an overview of these features. This will help you understand the capabilities and richness of CXdb. For details on how to operate the features, refer to the *CONVEX CXdb Reference* and *CONVEX CXdb User's Guide.*

Command files and initialization files

In addition to entering CXdb commands directly in the command window, you can create input files that contain the commands you want to use. CXdb can then read and execute these files like scripts.

There are two types of input files:

- Command files
- Initialization files

You can create these files either inside or outside of CXdb by using any standard text editor that can create and store files in ASCII format.

Initialization files execute automatically whenever you invoke CXdb. However, other types of command files do not execute unless you explicitly invoke them.

Command files

Command files are particularly helpful in the following cases:

- Initializing CXdb
- Defining macros
- Defining aliases
- Performing the same sequence of debugger operations on several different programs

A command file can contain any valid CXdb command, including aliases and macros. They can also contain commands to execute other command files.

CXdb executes the lines of the command file just as if you had typed them directly in the command window. You can control whether or not CXdb displays the lines of the command file as it executes them.

Initialization files

An initialization file is a special type of command file that executes automatically whenever you invoke CXdb. Like other command files, the initialization file contains a sequence of CXdb commands. You can use initialization files to set up all the parameters that define the CXdb working environment and the program working environment.

There can be any number of initialization files in various directories throughout your system. CXdb can use them in different combinations, depending on which directory you are in when you invoke CXdb.

At the global level, there is a default initialization file in the `/usr/lib/cxdb` directory. This file defines the default CXdb environment for all users on the system. Every time you invoke CXdb, it executes this default initialization file first.

At the local level, you can have your own initialization file that customizes the CXdb environment to suit your individual needs. The name of this file must be `.cxdbinit`, and you can create it in the same way you create any CXdb command file. You can have a separate initialization file in each directory you own, but the name of the file must be `.cxdbinit` in each case.

When you invoke CXdb, it first executes the default initialization file in the `/usr/lib/cxdb` directory. Next CXdb searches your home directory for a `.cxdbinit` file and executes it. Finally, CXdb searches your current directory (the console working directory) for a `.cxdbinit` file and executes it. Thus, the commands in your local initialization files take precedence over the commands in the global default initialization file.

You can prevent the initialization files from being executed when you invoke CXdb.

Eventpoints and handlers

The predefined eventpoint types—breakpoints, tracepoints, and watchpoints—were introduced in Chapter 3. In addition to these predefined eventpoints, there are general eventpoints available in CXdb. The general eventpoints can trap more types of events.

All eventpoints—including the predefined types—can have their own set of commands, called an eventpoint handler. When the eventpoint is triggered, the commands of its eventpoint handler are executed.

General eventpoints

The general eventpoints provide all the functionality found in the predefined eventpoint types; in fact, the predefined types are built from the general eventpoints. However, the general eventpoints offer additional functionality beyond the predefined types.

The general eventpoints can trap the following events:

- Execution reaches a particular instruction, line, routine, or source unit
- The process receives a particular signal
- The content of a memory region or of a variable changes
- A relational expression becomes true
- The process calls the `exec()` function
- A thread is spawned or joined

Refer to the *CONVEX CXdb User's Guide* for a more detailed description of eventpoints and how to use them.

Eventpoint handlers

An eventpoint handler is a set of CXdb commands that execute automatically when an eventpoint is triggered. All eventpoints can have their own handlers.

CXdb provides a default handler for all general eventpoints, breakpoints, tracepoints, and watchpoints. If you do not specify an eventpoint handler for a particular eventpoint, then CXdb executes the default handler for that type of eventpoint.

The default handlers perform basic actions, such as stopping the process and notifying you that the event has occurred. The default handler for tracepoints automatically continues execution of the process.

In many cases, the default actions are all that are needed. However, if you want more specific actions to occur when a particular eventpoint is triggered, you can write your own handler for that individual eventpoint. You can also change the default eventpoint handlers.

Eventpoint handlers can include most CXdb commands, and they can be as complex as necessary. For example, you can write a handler that removes its own eventpoint. The eventpoint is triggered once and then removed.

Signals

Signals are interrupts to a process that are generated by other processes or by the operating system. There are a number of standard signals defined for the UNIX kernel. They indicate events such as illegal instructions, segmentation violations, floating point exceptions, etc. The signals recognized by CXdb are those listed in the man pages for `sigvec(2)`.

If your process is executing under CXdb control, it does not receive a signal directly. Rather, CXdb catches the signal before the process receives it and halts the process. Then one of the following occurs:

- If you have specified an eventpoint to catch the signal, then CXdb executes the handler for that eventpoint
- If you have not specified an eventpoint, then CXdb performs the special actions that are enabled for that signal

For each signal, you can enable or disable the following special actions:

- Resume execution of the process without sending it the signal
- Display a message telling you what signal was received
- Pass the signal to the process when execution resumes

By enabling or disabling the above actions, you can decide what CXdb does when the process receives a signal. Each signal has its own set of the above three actions.

You can also change what signal is actually sent to the process when execution resumes.

Debugger variables

During a debugging session, CXdb creates a number of objects known as windows, eventpoints, and processes. When CXdb creates one of these objects, it assigns an object number to it. You can use this number directly to reference a particular object in subsequent CXdb commands. However, CXdb also enables you to define debugger variables to store the values of the object numbers.

In a long debugging session, you may create many eventpoints, each of which is given a unique number. By assigning a debugger variable to an eventpoint, you can refer to it by name rather than by number.

A debugger variable assumes the data type of the object that is assigned to it. If you assign a new object to an old variable, the variable assumes the data type of the new object.

Debugger variables can refer to other things besides CXdb objects. For example, a debugger variable can also be used to store the value of a language expression. In this case, the debugger variable takes on all of the characteristics of the language expression, including its data type.

Finally, there are some predefined debugger variables for referencing specific process registers and signals. For more information about these special debugger variables, refer to the *CONVEX CXdb Reference*.

Aliases and macros

The standard CXdb command set is extensive and flexible. However, you might want to customize some of the commands to fit your particular application. Or you might want to create an easy way to repeat a specific sequence of commands that you use frequently. Aliases and macros provide these capabilities.

Aliases

An alias is a synonym, or substitute, for characters entered on the CXdb command line. An alias can substitute for part of a command, the entire command, or multiple commands. When it encounters an alias on the command line, CXdb replaces the alias with the character string it represents.

Because it is strictly a character string substitution, an alias cannot accept variable parameters. When you use an alias, it must appear as the first part of a command.

Aliases can represent standard CXdb commands, macros, or even other aliases.

Macros

Macros are a mechanism for building customized commands from the basic CXdb command set. A macro can contain part of a command, a complete command, or multiple commands.

Macros differ from aliases in three important ways:

- Macros can accept variable parameters, but aliases cannot.
- Macros repeat themselves until all parameters passed have been operated on.
- Macros can appear anywhere on the command line, but aliases must appear at the beginning of the line.

Macros can contain standard CXdb commands, aliases, or even other macros.

Logging

Logging is the recording of information generated during the debugging session. Normally, this informations displays in the command window. To log the information, you define a list of files called viewports. Then, as the information is generated, a copy of it is sent to all of the viewports on the list.

There are three types of viewport lists, depending on the kind of information you want to log. The types are:

- **cmdlog**—Viewports for any input entered in the CXdb command window.
- **cmdout**—Viewports for output resulting from CXdb commands.
- **cmderr** —Viewports for error messages generated in response to CXdb commands.

Each viewport list can contain any number of file names in addition to the command window. These lists then serve as the default viewports for the CXdb working environment.

You can override the default viewports by using redirection operators on the command line. Redirection operators enable you to specify individual viewport lists with each command that you execute. These individual viewport lists apply only to the particular command in which they appear. The redirection operators also allow you to specify whether to append the new information to the end of the file or to overwrite the file.

For both the default viewports and the command-specific ones, CXdb creates any files that do not exist. In both cases, you can also control whether or not CXdb writes to viewport files that already exist.

Scope

Scope defines where and when an identifier is visible in a program. The rules that determine the scope of an identifier are different for FORTRAN and C. CXdb uses the same scope rules as the language of the program you are currently debugging.

Scope plays a crucial role when debugging because not all of your program identifiers are visible from the current point of execution. If the identifier is visible within the current scope, you can reference it directly. However, if you want to reference an identifier that is not visible within the current scope, then you can do so by specifying the scope path for that identifier.

The current scope is defined by the current point of execution. If you reference an identifier without a scope path, CXdb looks for the identifier in the current scope.

Scope paths specify exactly what identifier you want to reference. You can use a scope path to access an identifier that is not currently visible, if the variable has memory allocated for it. Scope paths enable you to reference identifiers in other source files, or even another language. For instance, scope paths allow you to access loader symbols.

Most of the time it is enough to be aware of your current scope. By knowing the current scope, you can determine if the identifier you want to reference is visible. If it is, no scope path is needed. If it is not, and the identifier has memory allocated for it, then you can use a scope path to reach the identifier.

In FORTRAN, scope paths are especially useful when you want to reference common block identifiers in routines outside of the current routine.

In C, scope paths are useful when you want to reference static identifiers, or identifiers that are shadowed. An identifier is shadowed when another identifier with the same name exists in an inner block of the current scope.

Optimized Code Concepts 5

This chapter is an introduction to debugging optimized code. CXdb is the first commercial debugger to properly associate optimized code back to the original source code.

Optimized code

Optimization is a process that the compiler goes through to make your programs more efficient. Optimization modifies your code to make maximum use of the architecture of the machine on which the program is running. There are many different techniques for optimization, but all of them result in object code that differs significantly from your original source code.

For example, a common optimization technique known as code motion changes the order of instructions so that they execute in a more efficient sequence. One form of code motion removes instructions from the interior of a loop if those instructions do not need to be repeated every time the loop executes.

While code motion makes the program more efficient, it also makes the code harder to debug. For one thing, the code now executes in a different sequence than the way you programmed it. In addition, the program variables may no longer be where you expect to find them.

The CONVEX FORTRAN and C compilers provide several different levels of optimization. CXdb can debug code that has been optimized to level `-no`, level `-O0`, or level `-O1` on these compilers. In general, these levels involve the following types of optimizations:

- **Level -no**—Optimization of expressions within a single basic block. At this level, each basic block contains one source statement. Thus, in terms of source units, level `-no` optimizes expressions within a single statement. No optimization occurs between statements. Some examples of level `-no` optimization are constant folding and redundant use elimination.

- **Level -O0**—Optimization of expressions within a single basic block. At this level, a basic block can contain a sequence of statements. Thus, in terms of source units, level `-O0` optimizes expressions across multiple statements. Level `-O0` includes all of the level `-no` optimizations plus some others like assignment substitution and common subexpression elimination.
- **Level -O1**—Optimization across multiple basic blocks within a routine. In terms of source units, level `-O1` optimizes across expressions, statements, blocks, or loops within a routine. Level `-O1` includes all of the level `-O0` optimizations plus some others like hoisting and sinking.

Level `-no` is the default optimization level for the CONVEX compilers. Even at this level, the compilers perform some optimizations that can affect the way your program executes.

Source units and optimization

Source units are syntactical elements of source code. As described in Chapter 3, the five different types of source units are expression, statement, block, loop, and routine.

The compiler identifies each source unit as it compiles the source file. At optimization level `-no`, each source unit maps to a contiguous sequence of instructions in the object code. At higher optimization levels, the mapping is less direct because the instructions that correspond to a particular source unit are no longer contiguous. The higher the optimization level, the more dispersed the mapping becomes.

The disassembly window displays the optimized code, and the source window displays the source units of the source code. The source window highlights all source units that correspond to the current instruction indicated in the disassembly window. By observing both windows at the same time, you get a dynamic visual representation of the mapping between optimized executable code and source code.

In some cases, optimization may compress the code, so that one machine instruction relates to many source units. In these cases, the source window highlights all of those source units whenever the corresponding instruction is current.

In other cases, optimization may expand the code, so that many machine instructions relate to the same source unit. In these cases, the source window highlights that source unit whenever any one of the corresponding instructions is current.

In general, then, the highlighting in the source window indicates all of the source units that correspond to the current instruction displayed in the disassembly window.

The mapping between source code and executable code not only helps you to understand the effects of optimization, but it also makes it possible for you to debug the optimized code directly. When you use the debugging commands on optimized code, you can directly observe their effects on both the executable code and the source code at the same time. In addition, if you make any changes to the source code, you can determine the effects of those changes on the optimized code. Source units make all of this possible.

Debugging optimized code

Traditionally, debuggers could not provide much help with the debugging of optimized code. Before, when optimization removed variables or performed code motion, debuggers were unable to properly track these effects on the source code.

Other debuggers either disallow you to debugging optimized code or produce erroneous results (such as printing a value for a variable that has been optimized away).

CXdb will not print erroneous results for code optimized up to the `-O1` level. In addition to this added security, CXdb provides the mapping between source units and disassembled code, as described above.

By stepping through your program an instruction at a time, you can see the highlighting of your source code provide clues as to the kinds of optimization the executable code has gone through. However, debugging optimized code is still inherently difficult, because of the loss of the relationship between the source code and the executable code.

If possible, first debug your program compiled at the -no level. Because the source code closely matches the executable code optimized at the -no level, debugging is more straightforward than at higher optimization levels.

Eventpoints and optimization

There are many different types of events that CXdb can detect. To set an eventpoint for any of those events, you must specify a location for the eventpoint. In general, you can specify the location as either an address, a source unit number, or a source line number.

With optimization, some of the locations where you try to set an eventpoint might no longer exist. For example, you could want to set a tracepoint at a particular expression, but that expression may have been eliminated by optimization because it was dead code that never executed.

Even without optimization, there are some lines of source code that do not contain any source units and, therefore, do not generate executable code. Blank lines and comment lines are examples of these. You cannot set eventpoints at these locations either.

In all cases, you can set an eventpoint at a location only if there is executable code corresponding to the source unit at that location. If you try to set an eventpoint at a location where this is not true, CXdb informs you that this is not a valid location for an eventpoint. Then CXdb asks if you would like to use an alternate location.

In some cases, optimization can move the target of an eventpoint to a different location. For example, you might want to set a breakpoint at a particular line of source code, but that line may have been moved by hoisting. In such cases, CXdb sets the eventpoint at the new (optimized) location of the source unit.

In other cases, optimization can produce multiple instructions from one source unit. If you create an eventpoint for that source unit, CXdb sets the same eventpoint at each one of the corresponding instructions.

Stepping through optimized code

Because of the way optimization modifies the executable code, it affects the way stepping behaves. After optimization, there is no longer a one-to-one relationship between the source units displayed in the source window and the executable instructions displayed in the disassembly window. Therefore, the stepping commands might not always produce the visual results you expect.

For example, if a particular statement of source code has been hoisted out of a loop, one stepping command might skip over that statement at first. Later on, another stepping command could jump backward to that statement even though you are stepping forward. Thus, the highlighting in the source window jumps backward and forward as you step.

This behavior in the source window is due to the fact that execution proceeds in the sequence specified by the optimized executable code, but the source window displays the source statements in the same sequence that you programmed them. As each instruction executes, CXdb maps the instruction back to its corresponding source units and highlights them in the source window. So the highlighting in the source window follows the same sequence as the executable code, which is not necessarily the same sequence as your source code.

In some cases, there will be parts of the source code that you cannot step to with any of the stepping commands. For example, if optimization has eliminated a particular source unit because it contains redundant code, then none of the stepping commands will stop at that source unit. In order for stepping to stop at a source unit, there must be executable code corresponding to that source unit.

In general, when stepping through optimized code, it is best to view the activity in both the source window and the disassembly window at the same time. Then do either of the following:

- **Step by machine instruction**—This shows the mapping from instructions to source units.
- **Step by expression source unit**—This shows the mapping from source units to instructions.

Threads

A thread is a sequence of instructions that is fetched and executed by a single CPU. A process consists of one or more threads, each of which can execute on a different CPU. Memory files, signals, and other process attributes are generally shared among threads in a given process. This enables the threads to joint together in performing a particular task.

Threads are created and terminated in three different ways:

- Automatically by optimization at level `-O3` or higher on a CONVEX compiler
- Explicitly by compiler directives added to the source code
- Explicitly by instructions coded in assembly language programs

The operating system assigns an identification number to each thread that is created. CXdb uses these same thread numbers to identify the threads in the program it is debugging.

If your program has been compiled with an optimization level lower than -O3, and if you have not explicitly created multiple threads in the program, then your program consists of only a single thread. In this case, you do not have to be concerned with thread numbers.

However, if your program has multiple threads, then you can specify thread numbers in many of the CXdb commands to indicate which threads you want to affect. For example, you can set breakpoints on certain threads but not on others, or you can step the execution of some threads but not others. This gives you even greater capability to isolate bugs in optimized code.

Glossary

A

alias
A synonym, or direct substitute, for a CXdb command.

attaching
A method of debugging a running process. CXdb retrieves the image of the process and associates it with the appropriate executable file and data files.

B

batch mode
A method of running CXdb in which there is no interactive user interface. Only command line functionality is available in batch mode.

breakpoint
A predefined eventpoint that you can place before source units, routines, lines of source code, or machine instructions. When the executing process reaches the breakpoint, execution stops and control returns to you.

C

cmderr
The viewports that receive the error messages generated in response to CXdb commands.

cmdlog
The viewports that receive the text of commands entered in the CXdb command window.

cmdout
The viewports that receive the output from CXdb commands.

command file
A text file that consists of CXdb commands. It is one way of inputting commands for CXdb to execute.

command window
The window that displays CXdb command lines, command output, and error messages. It is the primary means of communicating with CXdb.

console working directory
The current directory from which you are working. It is used as the base path for relative path names you use in CXdb commands.

core file
A file that the operating system creates if a fatal error occurs during execution of your process. You can examine the process image retrieved from the core file, but you cannot execute the terminated process.

current scope
The scope based on the current point of execution. If you reference a variable without using a scope path, CXdb looks for the variable in the current scope.

CXdb working environment
The environment in which you work when using CXdb. You can modify the environment to meet your debugging needs.

CXwindows
A user interface designed by CONVEX to provide graphics capability for your workstation. It supports a window environment for networked computer systems and is based on the X Window System developed at the Massachusetts Institute of Technology. CXdb can run in a CXwindows environment.

`.cxdbinit`
The name of a CXdb initialization file. You may have a separate initialization file in many different directories, but each of those files must be named `.cxdbinit`.

D

debugger variable
A symbol that represents either a CXdb object or a program object. You can use debugger variables anywhere CXdb object numbers or source language expressions can be used.

default process settings
The set of process settings in the CXdb working environment. The values of this set are given to the process settings in the program working environment when a process object is created.

disassembled code
The assembly language code that is equivalent to the machine language instructions for your program. CXdb generates the disassembled code from the executable (binary) file for your program.

disassembly window
The window that displays the disassembled code for your program. From the disassembly window, you can open other windows that display the process registers. You can also open more than one disassembly window at the same time.

E

environment variable
A shell variable that contains information about your environment. Environment variables are inherited from the shell, but you can override them with CXdb commands.

eventpoint
Any type of trap that can watch for a particular event to occur in an executing process. Events that can be trapped include signals being caught, variables changing value, locations being reached, and relational expressions becoming true. All eventpoints have predefined handlers, but you can also define your own handlers.

eventpoint handler
The set of CXdb commands to execute when an eventpoint is triggered. If an eventpoint has its own handler, it is used. If it does not, the default handler for eventpoints of its type is used.

examine window
The window that displays the contents of process memory. You can open multiple examine windows simultaneously and control the format of the display.

executable file
A binary file created by the compiler. CXdb uses the executable file to find the compiler-generated data files. The executable file is one of the two items CXdb needs to debug your program. The other item is the process image.

G

granularity
The size of a source unit. The possible sizes are routine, loop, block, statement, and expression.When stepping, you can specify the granularity to control the size of step to take. The default granularity is statement.

H

handler
See *eventpoint handler.*

help
An online facility to display text about CXdb commands, concepts, parameters, and error messages.

I

initialization file
A command file that executes automatically when you invoke CXdb. You may have a separate initialization file in many different directories, but each of those files must be named `.cxdbinit`.

image
See *process image.*

M

macro
A mechanism for customizing the CXdb commands. A macro can substitute for part of a command, a complete command, or multiple commands. It can include parameters with default values, and it can execute iteratively.

Maryland Windows
A user interface developed by the University of Maryland to provide a window environment for line-oriented terminals. CXdb can run in a Maryland Windows environment.

O

optimized code
Executable code that has been enhanced by the compiler during compilation of the source code. The type and extent of optimization depends on the optimization level you specify during compilation. CXdb can fully debug code that has been optimized to level `-no`, `-O0`, or `-O1`.

optimization level
The degree to which source code is optimized by the compiler. The CONVEX FORTRAN and C compilers have five levels of optimization. CXdb can debug code that has been optimized to level `-no`, `-O0`, or `-O1`.

process
The running version of your program. The current state of the process is stored in the process image.

process object
The term given to the structure that holds all of the information about your program in CXdb. It is created when you specify an executable file or process image.

process image
A dynamic snapshot of the current state of a process. CXdb uses the process image when you debug your program. You can create an image by running your program within CXdb, acquire an image from an already running process, or retrieve an image from a core file.

process memory
The portion of system memory that is used by an executing process.

process registers
A set of high-speed storage locations that contain information about your executing process. Process registers include scalar registers, status registers, address registers, vector registers, and communication registers (for C200 Series machines only).

process settings
The collection of settings that control various aspects of your process. You can modify these to match the needs of your process, such as its floating point mode.

process stack
A dynamic, LIFO storage list of frames that the operating system uses to track the flow of execution in your process. Each frame stores the registers and local variables from the previous execution context, the addresses used to manage the current stack frame, and a pointer to the previous stack frame. With CXdb, you can display the stack and move between frames.

process working directory
The directory from which the process is run. The base directory for the relative path names of all files used by your process.

program working environment
The environment that controls various aspects of your program. You can tailor it to meet the needs of your program.

R

registers
See *process registers.*

S

scope
The domain in which a variable is visible. The rules that determine scope are different for FORTRAN and C. If a variable is not visible in the current scope, you can access it by specifying its scope path.

scope path
The complete path to a variable. The scope path enables you to access static variables outside of the current scope.

search path
The list of directories where CXdb searches for the source file and compiler-generated data files associated with an executable file.

signal
An interrupt sent to the process by the operating system. A signal indicates a special event such as a segmentation fault or division by zero. CXdb catches signals before the process receives them. You can determine what actions CXdb takes when it catches a signal.

source code
The uncompiled version of your program, written in C or FORTRAN. Although CXdb actually uses the executable (compiled) version of your program, it maps the executable code back to the source code so that you can debug from the source.

source file
The file that contains your source code.

source unit
A logical subdivision of the source code. The five types (or granularities) of source units are routine, loop, block, statement, and expression.

source window
The window that displays the source code. In this window, CXdb can also indicate all eventpoints and highlight the active source units.

stack
See *process stack.*

stack window
The window that displays the contents of the process stack.

stepping
The incremental execution of your process. You can step the process by machine instructions or by source units.

symbolic debugger
A debugger that can map source code to executable machine instructions. CXdb is a symbolic debugger with the added functionality and capability to debug optimized code symbolically.

T

thread
A linear sequence of executable instructions. A process may have more than one thread executing at the same time. You can select which threads are affected by particular CXdb commands.

tracepoint
A predefined eventpoint that you can place before source units, routines, lines of source code, or machine instructions. When the executing process reaches the tracepoint, CXdb informs you of the event, but execution does not stop.

V

viewport
A location that receives a copy of the CXdb input (cmdlog), output (cmdout), or error messages (cmderr). A viewport may be either a file or the CXdb command window. You can define a separate list of viewports for each of cmdlog, cmdout, and cmderr. The default viewport for each is the command window.

W

watchpoint
A predefined eventpoint that you can set to watch a region of process memory. If the executing process changes the contents of that region, CXdb halts the process and returns control to you.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

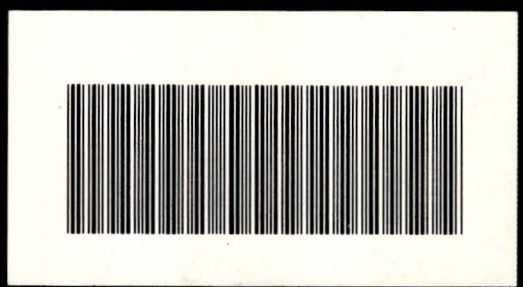
T

U

V

W

X



Order Number
DSW-471

Document Number
710-015330-001