CONVEX Consultant User's Guide

Document No. 740-002530-203

Eighth Edition
October 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Consultant User's Guide
Order No. DSW-025
Eighth Edition

© 1988, 1987, 1986, 1985 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, stored electronically, 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 (CONVEX) 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 OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
VECLIB is a trademark of CONVEX Computer Corporation.
UNIX is a trademark of AT&T Bell Laboratories.
GVAS is a trademark of Verdix Corporation.

Printed in the United States of America

Revision/Update Information for CONVEX Consultant User's Guide

Edition	Document No.	Description
Eighth	740-002530-203	Released with CONVEX Consultant 8.0, October 1988. Includes the following changes and additions: Enhanced and reorganized Chapters 1 and 2 Added Chapter 3, "Debugging Optimized Code" Added Chapter 4, "Multithreaded Debugging" Moved previous Chapters 3 and 4 to current Chapters 5 and 6 Removed previous Appendices B and C. Moved previous Appendix D to current Appendix B
7.0	740-02530-201	Released with Consultant 6.1.2, May, 1988
6.0	740-02530-200	Released with Consultant 6.1, August, 1987
5.0	740-00530-000	Released with Consultant 6.0, February, 1987
4.0	740-00530-000	Released with Consultant 4.0, September, 1986
3.0	740-00530-000	Released with Consultant 3.0, January 1986
2.0	740-00530-000	Released with Consultant 2.0, September 1985
1.0	740-00530-000	Released with Consultant 1.0, February 1985

Table of Contents

1 Introduction to *csd*

1.1 What Is a Symbolic Debugger? 1-2
1.2 Capabilities Overview 1-2
1.3 Debugging Concepts 1-3
1.3.1 Eventpoints 1-3
1.3.2 Call Stack 1-3
1.3.3 Environment and Scope 1-4
1.3.4 Current Context 1-5
1.3.5 Distinguishing Between Nonunique Symbols 1-5
1.4 Using the Debugger as a Learning Tool 1-6
1.5 Stopping the Program 1-7
1.6 Notes and Limitations 1-7

2 Performing Debugging Tasks

2.1 Invoking the Debugger 2-2
2.2 Using Multiple Source Files and Directories 2-4
2.3 Terminating the Debugging Session: *quit* 2-4
2.4 Getting Help: *help* 2-4
2.5 Displaying Program Information 2-5
2.5.1 Displaying Source Code: *list* 2-6
2.5.2 Displaying Values: *print* 2-8
2.5.2.1 Index and Subranges 2-9
2.5.2.2 Type Transfer 2-10
2.5.3 Displaying Symbol Information: *which, whereis, whatis* 2-12
2.5.4 Displaying the Current Routine Name: *func* 2-14
2.5.5 Displaying the Current Source File Name: *file* 2-17
2.5.6 Displaying Register Contents: *regs, vregs* 2-17
2.5.7 Displaying the Call Stack: *where* 2-18
2.5.8 Displaying a Core Dump: *dump, set* 2-19
2.5.9 Assigning Values: *assign* 2-20
2.5.10 Displaying Raw Memory 2-22
2.5.11 Searching for a Pattern: */, ?* 2-24
2.6 Controlling Program Execution 2-25
2.6.1 Executing a Program: *run, rerun* 2-25
2.6.2 Executing a Single Source Line: *step, next* 2-27
2.6.3 Executing a Single Instruction: *stepi, nexti* 2-29
2.6.4 Resuming Program Execution: *cont* 2-30
2.6.5 Executing Commands From a File: *source* 2-31
2.6.6 Executing a Compiled Routine: *call* 2-32
2.7 Handling Signals 2-32
2.7.1 Specifying Signals to Catch: *catch* 2-33
2.7.2 Specifying Signals to Ignore: *ignore* 2-33
2.7.3 Passing Different Signals: *cont* 2-34
2.8 Manipulating Eventpoints 2-35
2.8.1 Displaying Eventpoints: *status* 2-36
2.8.2 Setting a Breakpoint: *stop, stopi* 2-37
2.8.3 Setting a Tracepoint: *trace, tracei* 2-40
2.8.4 Executing Commands Conditionally: *when* 2-43
2.8.5 Deleting Eventpoints: *delete* 2-44
2.9 Manipulating the Call Stack 2-44
2.9.1 Displaying the Call Stack: *where* 2-45
2.9.2 Moving Through the Call Stack: *down, up* 2-45
2.9.3 Return From Routines on the Call Stack: *return* 2-47
2.10 Customizing *csd* Behavior 2-47
2.10.1 Using Command Aliases: *alias, unalias* 2-47
2.10.2 Changing the Current Execution Mode: *mode* 2-49

2.10.3 Changing the Current Floating-Point Mode: *fpmode* 2-49
2.10.4 Changing the Current Integer Format: *format* 2-50
2.10.5 Changing the Floating-Point Precision 2-50
2.10.6 Changing the Directory Search Path: *use* 2-50
2.11 Executing a Shell Command: *sh* 2-51
2.12 Editing a Source File: *edit* 2-51

3 Debugging Optimized Code
3.1 Effects of Optimization 3-2
3.2 Debugging an Optimized Routine 3-3
3.2.1 About the Routine 3-4
3.2.2 Problems with Debugging Optimized Code 3-6
3.2.3 Tracking a Bug in Optimized Code 3-9

4 Debugging Multithreaded Programs
4.1 Multithreaded Programs 4-2
4.1.1 Terminology 4-2
4.1.2 Multiple Threads 4-3
4.1.3 Registers 4-7
4.2 Current Thread 4-8
4.3 A Multithread Debugging Philosophy 4-9
4.4 Commands for Debugging Multithreaded Code 4-10
4.4.1 New Commands 4-10
4.4.2 Enhancements to Old Commands 4-12
4.5 Sample Multithreaded Routine 4-13
4.6 Monitoring a Multithreaded Program 4-15
4.7 Debugging a Multithreaded Program 4-20

5 Using Profilers
5.1 Using *prof* 5-2
5.1.1 Capabilities Overview 5-2
5.1.2 Basic Operations 5-2
5.2 Using *gprof* 5-7
5.2.1 Capabilities Overview 5-7
5.2.2 Basic Operations 5-7
5.2.3 Analyzing *gprof* Output 5-18
5.3 Using *bprof* 5-19
5.3.1 Capabilities Overview 5-19
5.3.2 Basic Operations 5-20

6 Post-Mortem Dump Utility (*pmd*)
6.1 Capabilities Overview 6-2
6.2 Using *pmd* 6-2
6.2.1 Invoking *pmd* 6-2
6.3 Restrictions on *pmd* 6-3
6.4 Optimization Level Restrictions 6-4
6.5 Sample Programs Run With *pmd* 6-4

Appendices

A *csd* Summary A-1
A.1 Source-Level Commands A-1
A.2 Machine Instruction-Level Commands A-3
A.3 Standard Command Aliases A-4
A.4 Register Names A-5
B Reporting Problems B-1
B.1 Introduction B-1

B.2 Information Required to Report a Problem B-1

List of Tables

2-1 Dump Options 2-20
2-2 *csd* Output Modes 2-24
2-3 Sample Breakpoint Specifications 2-39
2-4 Sample Tracepoint Specifications 2-42
2-5 *csd* Standard Command Aliases 2-48

List of Figures

1-1 Environments for Variable "i" 1-5
2-1 Examples of the *print* Command 2-12
2-2 Sample C Program Using Multiple Environments 2-15
2-3 Output Illustrating Environment in *csd* 2-16
2-4 Example of the *assign* Command in FORTRAN 2-21
2-5 Examples of the *assign* Command in C 2-22
2-6 Sample Signal Handler Program in FORTRAN 2-34
2-7 C Program Using Nested Calls 2-46
3-1 Source of Optimized Subroutine, *tred1* 3-5
4-1 Processes in a Single-Processor System 4-4
4-2 Processes in a Multiprocessor System 4-6
4-3 A Multithreaded Process and Its Registers 4-8
4-4 Source of Parallelized Subroutine 4-14
5-1 Sample FORTRAN Program With *prof* Output 5-4
5-2 Sample C Program With *prof* Output 5-6
5-3 Sample FORTRAN Program With *gprof* Output 5-9
5-4 Sample C Program With *gprof* Output 5-15
5-5 Sample FORTRAN Program With *bprof* Output 5-21
5-6 Sample C Program With *bprof* Output 5-23
5-7 *bprof* Output for FORTRAN Program, *-m* Option 5-24
5-8 *bprof* Output for C Program, *-m* Option 5-24
5-9 *bprof* Output From FORTRAN Program, *-l* Option 5-25
5-10 *bprof* Output From C Program, *-l* Option 5-26
5-11 Sample Source Listing Using *indent* 5-27
6-1 Sample C Program With *pmd* Output 6-5
6-2 FORTRAN Program With *pmd* Output 6-7
B-1 Sample *contact* Session B-3

Preface

Introduction

This guide describes the *csd, prof, bprof, gprof,* and *pmd* utilities used with the CONVEX UNIX operating system. *csd* (formerly *dbx*) is a source-code debugger that requires special support from the compiler and loader to be used. *prof, bprof,* and *gprof* are profilers that help you produce more efficient programs. *pmd* is a post-mortem dump analyzer that displays formatted listings on aborted programs.

Objectives and Intended Audience

This manual addresses the experienced C or FORTRAN programmer and describes the functions and operations of the optional CONVEX Consultant software programs. The guide describes the features, command formats, and symbols used by *csd, prof, bprof, gprof,* and *pmd.* Programmers interested exclusively in debugging assembly-language programs should refer to the *CONVEX adb Debugger User's Guide.*

Organization

The guide is divided into four chapters and four appendices, as follows:

- Chapter 1 provides an overview of the *csd* debugger and describes basic concepts needed to use *csd* effectively.
- Chapter 2 explains how to debug programs using *csd*
- Chapter 3 explains how to debug optimized code.
- Chapter 4 explains how to debug multithreaded code.
- Chapter 5 describes how to use the *prof, gprof,* and *bprof* profilers.
- Chapter 6 describes how to use the post-mortem dump utility, *pmd.*
- Appendix A is a *csd* command summary listed in alphabetical order.
- Appendix B describes how to use the *contact* utility to report problems.

Notational Conventions

The following conventions are used in this document:

- *Italics* within text indicate commands, filenames, or programs.
- Within command sequences and text, **bold** type indicates characters you should type exactly as they appear.

- In command examples and descriptions, command names appear in **bold** type. Words appearing in *italics* should be substituted with actual data.
- Brackets ([ ]) designate optional entries.
- Ellipses (. . .) show repetition of the preceding item(s).
- Words enclosed in rounded rectangles designate ASCII nonprintable characters. For example, RETURN stands for the "return key."
- Words separated by a hyphen and enclosed in rounded rectangles represent two keys you are to press simultaneously. For example, CTRL-D means you are to press and hold the CTRL key and press the D key.
- Information enclosed in a rounded rectangle shows text as it appears on the screen. For example,

```
These characters are seen on the
screen when issuing commands.
```

  — Within a screen, a box represents the position of the cursor.

  — Within a screen, text shown in **bold** indicates characters you should type exactly as they appear.

  — Within a screen, the RETURN symbol follows commands you are to type and indicates that you are to press the "return key" after typing the preceding characters.

- References to the *CONVEX UNIX Programmer's Manual* appear in the form *csh*(1), where the name of the man page is followed by its section number enclosed in parentheses.

Associated Documents

CONVEX provides the following related documents:

- *CONVEX UNIX Programmer's Manual, Parts I* and *II* contains complete reference material on the CONVEX UNIX operating system.
- *CONVEX adb Debugger User's Guide* describes the *adb* object-code debugger and is both a reference and user's guide.
- *CONVEX Loader User's Guide* describes the CONVEX loader and is both a reference and user's guide.
- *CONVEX UNIX System Manager's Guide* addresses the specifics of managing the system.
- *CONVEX Architecture Reference* describes the architecture of the CONVEX family of computers.
- *CONVEX Assembly Language User's Guide* describes the CONVEX assembler.

Chapter 1
Introduction to *csd*

This chapter presents an overview of the CONVEX symbolic debugger, *csd*. It explains some general terms you need to understand to use the debugger and it describes some of general features and uses of the *csd* debugger.

Some of the topics covered in this chapter include

- a description of symbolic debuggers
- capabilities overview
- breakpoints
- tracepoints
- call stack
- environment and scope
- distinguishing between nonunique symbols
- using the debugger as a learning tool
- notes and limitations

The CONVEX *csd* debugger is a symbolic debugger that lets you debug C and FORTRAN programs at the source level. You can also use it as an aid to understanding code written by others.

1.1 What Is a Symbolic Debugger?

A symbolic debugger is a tool that maps source code to executable machine instructions. A debugger can execute programs under its control, always knowing the current state of the executing program. When running an executable program under the control of a symbolic debugger, you can stop programs any time during execution, examine the state of the program, and look at program variables. Symbolic debuggers also let you catch a program when a fatal error occurs rather than letting the operating system create a core dump. Using a symbolic debugger, you can trace program execution at the source level (symbolically) and at the assembly-instruction level.

1.2 Capabilities Overview

csd is a source-level debugging tool designed specifically for debugging FORTRAN and C programs on the CONVEX UNIX operating system. *csd* also provides a limited machine instruction-level debugging function. The major features of the *csd* debugger include:

- **Source-level execution control**—the ability to debug the program at the source code level and at the assembly-language level. This means that breakpoints are defined by source line numbers and stepping is done on a source-line basis.

- **Enhanced capabilities for examining core files and program files in a variety of formats**—you can examine core dumps with *csd* to find the exact line on which the program failed. You can also obtain a symbolic runtime stack trace at the time the program core dumped. This feature is commonly used and can help solve most program failures.

- **Ability to debug multiple source module programs**—*csd* automatically updates the debugging environment as execution branches from module to module.

- **Symbolic access to program variables**—*csd* can access program variables by name and by absolute address. This frees you from having to know the details of storage implementation and variable representation. All you need to know is the variable's name and scope, and *csd* automatically formats it in its correct type.

- **Ability to debug optimized code**—*csd* can debug *fc*(1) and *vc*(1) optimized and vectorized code. You cannot, however, depend on a one-to-one correspondence between the object and source code; you must correlate the assembly code with the source code. The code optimizations performed by the compiler remove, rearrange, and transform the source code. If you try to set a breakpoint or tracepoint on code that has been removed by the optimizer, *csd* displays an error message stating that the source line is not a valid place at which to set a breakpoint or tracepoint. When source lines are listed, lines containing an asterisk after the line number indicate executable lines at which you can set breakpoints or tracepoints. Please see Chapter 3 for detailed information on debugging optimized code.

The *csd* debugger interprets specially constructed symbol tables to access source files used to create the program being debugged. The CONVEX scalar C compiler, *cc*(1), the CONVEX Vector C compiler, *vc*(1), and the CONVEX FORTRAN compiler, *fc*(1), build these symbol tables when you select the *-g* option (for *cc*), and the *-db* option (for *vc* and *fc*). The CONVEX linker, *ld*(1), also uses the *-g* and *-db* options to load a special object library. With programs so compiled and linked, you can then use *csd* to stop or trace program execution to analyze the program's

behavior.

1.3 Debugging Concepts

Using a symbolic debugger effectively depends on understanding several language and implementation concepts, including eventpoints, the debugger call stack, environment and scope, the current context, and distinguishing between nonunique symbols. This section explains these concepts.

1.3.1 Eventpoints

An eventpoint is a point in the program where a specified event can occur. When an eventpoint is reached, you can suspend program execution, display program information, or execute a series of *csd* commands. *csd* recognizes three types of eventpoints:

- breakpoints
- tracepoints
- whenpoints

A breakpoint is a point in a program where the debugger suspends the program's execution. When the debugger stops program execution, it waits for you to enter a command. You can look at the state of the program, display any of the program variables, assign new values to variables, and perform several other actions to help you see what the program is doing.

Note that while the program is executing, the debugger does not accept commands or input, but you can type them. When the program is ready to accept input or the debugger is ready to accept commands, *csd* reads your input.

A tracepoint is a point in a program where the debugger displays information about the progress of the program. Unlike breakpoints, *csd* doesn't return control to you when it encounters a tracepoint, but displays a message stating that the program reached the tracepoint. Execution then continues. Tracepoints are usually used to monitor a program's progress.

A whenpoint is a event that causes a series of *csd* commands to be executed. This event is expressed as a conditional statement (an equality test, for example). When the condition becomes true, a series of specified commands is executed.

1.3.2 Call Stack

The call stack represents the state of all currently active routines in the program being debugged. These are routines that have been called, but have not yet returned to their callers. When the program is executing, the routine that is currently executing is at the top of the call stack. A routine call "pushes" the called routine on the top of the stack. When a routine returns to its caller, the returning routine is "popped" from the top of the stack.

When the debugger suspends program execution at a breakpoint or after executing a single instruction, the routine containing the point where the program execution stopped is at the top of the call stack. The debugger provides a command, *down* that lets you move down the call stack, that is, from the current routine to the routine that called the current one. There are also commands for moving up (*up*) and displaying the call stack (*where*). As explained in section 2.8 "Manipulating the Call Stack," changing the current level on the call stack changes the variables that are directly visible.

1.3.3 Environment and Scope

The environment, or scope, of a symbol refers to the routines and source files in which the symbol is declared. In C, the scope of a variable may be further defined according to nesting levels, or blocks, inside the function. *csd* automatically assigns a name to nested block. The assigned name begins with a "$b", ends with a nesting level number, and uniquely identifies each variable. Nesting levels are sequentially numbered, beginning with 0, when the program is loaded; they are unique across the loaded program. The nesting level number for global variables is zero. Nested variables are not used in FORTRAN programs, as all FORTRAN declarations are unique within a routine.

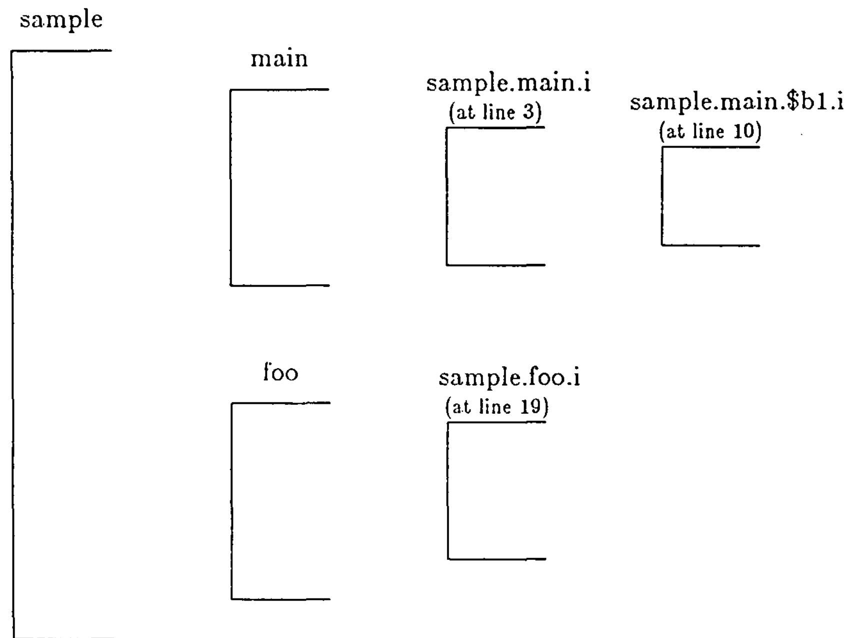
In the following example, the file *sample.c* contains a C program with three different instances of the variable *i*, declared on lines 3, 10, and 19.

```
% cat -n sample.c
1       main()
2       {
3               int i;   /* the first instance of i */
4               int j;
5
6               i = 5;
7               printf ("sample.main.i=%d\n",i);
8               for (j=1;j<i;j++)
9               {
10                      int i;  /* the second instance of i */
11                      i=i+j;
12                      printf ("sample.main.$b1.i=%d\n",i);
13              }
14
15              i=foo(i);
16      }
17
18      foo(i)
19      int i;   /* the third instance of i */
20      {
21              i=i*100;
22              printf ("sample.foo.i=%d\n",i);
23              return(i);
24      }
%
```

Each instance of the variable has its own scope (is defined within a specific environment) and is unknown outside of its scope. The *i* variable declared on line 10 is only defined inside the loop construct; the one declared on line 3 is defined inside the *main* routine; and the instance of *i* declared on line 19 is defined only inside the *foo* routine.

Figure 1-1 graphically illustrates the environments for the variable *i* in this program.

Figure 1–1: Environments for Variable "i"



1.3.4 Current Context

The current context defines the source position where program execution is suspended. The current context is defined by the triplet

(*file, routine, line*)

where

file	is the name of the file containing the executable code
routine	is the name of routine in which the program is executing
line	is the specific source line at which execution is suspended

At any given time, the current source position is defined by the current file, the current routine, and the current line.

1.3.5 Distinguishing Between Nonunique Symbols

Since C programs may use the same name for different symbols (variables or routines), *csd* needs a way to distinguish between them. In the program *sample.c* described earlier, the variable *i* represents a global variable, a routine-specific variable, and an internal loop variable. By default, *csd* assumes you are referring to the variable in the current context. If you suspend execution in the *foo* routine, *csd* assumes you are considering the routine-specific instance of *i* when you refer to the variable.

To look at a different instance of a symbol, you must qualify it with an environmental pathname. The format of this pathname is

[*module_name.*][*routine.*][*block_name.*]*symbol*

where

module_name	is the tail of the source file's pathname without the suffix. For example, *sample* is the *module_name* for */a/b/c/sample.c.*
routine	is the name of the routine in which the symbol is defined.
block_name	is the name of the nested block in which the symbol is defined.
symbol	is the name of the variable or routine.

In Figure 1-1, the environmental pathnames for the variable *i* are

- sample.main.i (line 3 in the source code)
- sample.foo.i (line 19 in the source code)
- sample.main.$b1.i (line 10 in the source code)

A partial pathname can be used if it is unique. The current environmental pathname is automatically maintained by *csd*. **References to non-qualified symbols default to using the current environmental pathname.** The current environmental pathname is changed when the current *routine* changes by the *func* command, when the *module_name* is changed by the *file* command, or when *csd* changes the pathname as the program executes.

1.4 Using the Debugger as a Learning Tool

The CONVEX *csd* symbolic debugger is primarily used as a tool for tracking down runtime errors in programs. Since it relates the source code to the flow of execution, it also helps when modifying or debugging code that someone else has written. The following paragraphs describe an approach to using the debugger to view the operation of another programmer's code.

If a routine's source file is in the directory search path, you can use the debugger's *file* command to view the code without knowing where the file is located. Furthermore, you can use the *step* and *next* commands to walk through the code as it executes. These commands let you read through the source code in the order of execution and help you limit the amount of code you must read to the code of special interest. If you accidentally step into a routine, you can get out of it by using the *return* command.

As you begin to see how the code works, you can examine selected portions more carefully. The *stop* and *delete* commands allow you to select areas that can be reached quickly using the *cont* and *run* commands.

When a breakpoint is reached, you can look at the current state of the program. The *print* command displays the values of variables or selected components of variables. The *where* (call stack) command displays the current call stack and the values of the arguments passed to each routine. The *down* and *up* commands let you move the entire debugging context to another routine on the call stack so you can examine the source code and variables in those routines.

1.5 Stopping the Program

csd stops executing the program and asks for user input when the program being debugged exits, when breakpoints are encountered, or when a fatal program error occurs. Even fatal errors return you to the *csd* command interpreter rather than to the UNIX shell. *csd* lists the number of the line containing the error, enabling you to locate the position in the source file where the error occurred.

csd relies on the hardware for notification of program errors. Because of the pipelined nature of the CONVEX hardware, some errors (such as a bus error after assigning a value through a nil pointer in C) are not detected until the program counter has been advanced by several instructions. *csd* reports the line and function based on the program counter value. Thus, for some error conditions, *csd* may indicate a source line number beyond the point where error actually occurred.

If a program error occurs near the return point of a function or routine, *csd* may report a line number in the function or routine listed next in the source code. When the error occurs, the hardware increments the program counter past the point of the error and then detects the error before exiting the current routine. The return instruction is not executed, so the program counter remains where the hardware set it instead of in the calling routine.

csd also stops when it encounters a signal. (In UNIX, you can select interrupt characters; see *stty* (1) for details.) For more information concerning signals, refer to section 2.6 "Handling Signals."

1.6 Notes and Limitations

Not all modules in the program need to be compiled with the *-g* or the *-db* option. If execution halts in a module not containing *csd* symbols, you can use a set of *csd* machine instruction-level commands to examine memory and give runtime stack traces.

If you compile a source module with *-c* and *-g* or *-db* and don't automatically link the program, you must link with *-g* or *-db*. If you don't, the program links with no errors, but a fatal error occurs when you start *csd*.

If you use *csd* on an application containing a mixture of C and FORTRAN routines, you must append an underscore to FORTRAN function names in all command references.

FORTRAN programmers note the following while issuing commands in *csd*:

- The name of the main program is "MAIN_". If you use a PROGRAM statement, the name of the main program is still "MAIN_", not the name in the PROGRAM statement.
- The array notation uses either parentheses "( )" or brackets "[ ]" for subscripted items.
- Type the names of variables, functions, and routines in lower case.
- Avoid using the *csd* reserved words **if**, **true**, **false**, **hex**, **decimal**, **chained**, **sequential**, **native**, **ieee**, and **auto** as symbolic names.
- The logical constants **.TRUE.** and **.FALSE.** are predefined; use **.true.** and **.false.**, respectively.
- Do not use the logical operators **.not.**, **.and.**, **.or.**; instead use **not**, **and**, **or**, respectively. The logical operators **.eqv.**, **.neqv.**, and **.xor.** are not defined.
- Do not use the relational operators **.eq.**, **.ne.**, **.gt.**, **.ge.**, **.lt.**, **.le.**; instead use ==, !=, >, >=, <, <=, respectively.

- A statement function cannot be called, because the compiler doesn't generate a closed routine but inlines it.
- Assignments of labels to integer variables (*assign 10 to i*, for example) are invalid as *csd* commands.
- Since common statements like

  ```
  common /abc/abc(10), def
  ```

  are permitted, you must prepend an underscore to refer to the name of the common block. Thus "abc" is the array and "_abc" is the common block.
- Complex values can be assigned and displayed. The form for a complex number is two *real* constants (each constant must include a decimal point) enclosed in parentheses and separated by a comma.

C programmers note the following:

- The name of the main program is "main".
- The array notation uses brackets "[ ]" for subscripted items.
- Names of variables and functions are case sensitive; type them as they appear in the source.
- Avoid using the *csd* reserved words **true**, **false**, **hex**, **decimal**, **chained**, **sequential**, **native**, **ieee**, and **auto** as names.
- The enumeration constants **true** and **false** are predefined.
- Use the relational operators ==, !=, >, >=, <, <=
- The *vc* compiler implements an enumerated type as an integer; the member values are treated as though they are defined by a set of *#define* statements. The fact that a variable is an enumerated type is not passed to *csd*. Thus, the response from *csd* for questions about enumerated types depends on whether the program was compiled with *vc* or *cc*.

Each time *csd* opens a source file, it compares the last-modified time for the file agains the time for the executable program. *csd* warns you if the source file is newer than the executable program.

By default, *csd* displays all output on the terminal. Normal output is written to *stdout*; however, error and warning messages are written to *stderr*. When you redirect the output to separate output and error files, the contents of the files will not match the results that would normally have been displayed on the terminal.

Chapter 2
Performing Debugging Tasks

This chapter describes commonly used debugging commands. It explains how the various options and parameters affect the function of a command. Examples are provided for the commands as they are introduced.

Some of the topics covered in this chapter include

- invoking *csd*
- getting help
- displaying program information
- controlling program execution
- handling signals
- manipulating eventpoints
- manipulating the call stack
- customizing *csd* behavior
- executing a shell command
- editing a source file

2.1 Invoking the Debugger

Invoke *csd* with the command

csd [**-r**] [**-I** *dir*] [...] [*-f*] [*objfile* [*corefile*]]

where

objfile — is an executable file produced by a compiler that has been directed with the appropriate option to produce symbol-table information. If you do not specify *objfile*, *csd* prompts you for one. The default name for *objfile* is *a.out*.

corefile — is the pathname of a file containing a core dump generated as the result of an abnormal program termination. Unlike *adb*(1), *csd* does not read a corefile by default; you must explicitly enter its name.

-r — instructs *csd* to execute *objfile* immediately (without waiting for *csd* commands). If the program terminates successfully, *csd* exits. Otherwise, *csd* reports the reason for termination, and you can either enter *csd* commands or let the program take the fault. *csd* reads from */dev/tty* when you specify *-r* and the standard input is not a terminal. If you do not specify this option, *csd* displays a prompt and waits for a command.

-I *dir* — directs *csd* to add the specified directory to the list of directories searched when *csd* looks for a source file. Normally, *csd* looks for source files in the current directory first and then in the directory where *objfile* is located (if different from the the current directory). The directory search path can also be set with the *use* command. Note that you can specify several directories by using this option the appropriate number of times. The space between the option and the directory name is optional.

-f — changes the scheduling mode from dynamic to fixed. When fixed scheduling is used, all CPUs in the system are served for the program when it runs, whether all of them are used or not. For more information about this option, read section 4.3, "A Multithread Debugging Philosophy."

When you invoke *csd*, you must specify an *objfile*. If you do not, *csd* requests one, as shown in the next screen. After reading the symbolic information, *csd* displays its command prompt—(csd).

```
% csd
Convex Symbolic Debugger.
Type 'help' for help.
enter object file name (default is 'a.out'): a.out
reading symbolic information ...

(csd)
```

The *corefile* contains an image of the state of the program at its termination. After using *csd* to access the *corefile*, you can use it to determine the active routines, their arguments, and the current value of all active program variables. After *csd* loads the core image, you can determine

the program's final state by examining runtime stack traces and variable contents. To read the *corefile*, type

csd a.out core

where *a.out* is the default name of the object file created by the compiler and *core* is the name of the core file created by the system. If you rename the object file, substitute that filename for *a.out*.

When *csd* is invoked, it first looks for the file *.csdinit* in the current directory, then in the user's home directory. If *.csdinit* exists, *csd* executes the commands in it immediately after it reads the executable file.

Assume a *.csdinit* file contains the following lines:

```
stop in main
stop in Get_response
run
list
```

When you invoke *csd*, these commands are executed as shown in the following screen.

```
% csd a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...

[1] stopped in main at line 353
  353*  {
  354*      stop_program = FALSE;
  355*      Get_lesson ();
  356*      while (stop_program == FALSE)
  357       {
  358*          Create_frame_index ();
  359*          next_frame = 0;
  360*          in_file = fopen (lesson_name, "r");
  361*          stop_lesson = FALSE;
  362*          while (stop_lesson == FALSE)
  363           {
(csd)
```

When invoked, *csd* reads the *objfile* and extracts the symbolic information from it. After completing the initialization, *csd* prompts you for a command. When you issue a command, *csd* executes it, prints a response, and prompts you for another command. A command line may contain a simple command or a list of commands separated by semicolons (;). Any commands that may interact with the shell (as in *edit, file, rerun, run, source, use*) must appear as the last command on the line. Any commands appearing after one of these commands are ignored.

When you use multiple commands on a command line, *csd* processes each command before displaying the prompt. *csd* treats command lines that consist only of a RETURN or an end-of-file (normally CTRL-D) as null commands, and simply prompts for another *csd* command.

2.2 Using Multiple Source Files and Directories

When specifying the name of a file or directory not in the directory search path the full pathname must be given. The characters "." and "~" (used in relative pathnames) may not occur at the beginning of a pathname. This applies to the commands *run, rerun, status, where, dump, edit, file, source,* and *use.*

csd can handle programs whose source files reside in multiple directories. There are several ways to tell *csd* which directories to search when trying to find a source file. However, if two or more source files in the list of directories have the same name, *csd* always finds the first instance of the filename in the directory search path. When *csd* finds a file other than the one you want, you can identify the correct file by

- executing the *use* debugger command.
- using the *-I* option of the *csd* command.
- specifying a full pathname for the *file* debugger command.

CONVEX recommends that you not have multiple source files with the same name in a directory search path.

2.3 Terminating the Debugging Session: *quit*

To exit *csd,* type

quit

If you prefer to use "exit" or "bye," use the *csd alias* command to create these aliases. If you place the command in the *.csdinit* file, *csd* defines the alias each time you invoke *csd.*

Program control passes to the program that invoked *csd* (typically a UNIX shell).

2.4 Getting Help: *help*

You can display a summary of all *csd* commands with the *help* command. To use the *help* command, type

help

at the *csd* command level, as illustrated in the next example.

```
(csd) help
                    Command Summary
Execution and Tracing
  catch     delete    mode      rerun     run       step      thread    trace
  cont      ignore    next      return    status    stop      threads   when
Displaying and Naming Data
  assign    down      format    print     up        where     which
  call      dump      fpmode    set       whatis    whereis
Accessing Source Files
  edit      file      func      list      use       /         ?
Miscellaneous
  alias     help      quit      sh        source    unalias
Machine Level
  nexti     regs      stepi     stopi     tracei    vregs     .         &
Standard Command Aliases
  a (assign)      f (func)      q (quit)      w (where)
  b (stop)        h (help)      r (run)       st (status)
  c (cont)        l (list)      s (step)      wi (whereis)
  d (delete)      n (next)      t (trace)     W (which)
  e (edit)        p (print)

(csd)
```

To display a synopsis of a particular *csd* command, type

help *command*

The synopsis includes the syntax and function of the *command.* The *help* command describes all alternatives of the *command.* If you require more detailed information, consult this manual or the online manual page for *csd* (type **man csd**).

The following screen shows how to get help on the *delete* command.

```
(csd) help delete
delete <index> ...      - Remove trace, stop or when of given index(es)
delete all              - Remove all traces, stops and whens
(csd)
```

2.5 Displaying Program Information

One of the most common tasks you will perform while using *csd* is looking at the program you are debugging. You might want to look at part of the source code, the value of some variable in the program, the declaration of a routine or variable, or several other parts of the executable program. You may also want to look at the core dump of a failed program.

csd provides several commands for displaying information about the program you are debugging. These commands include

list	display lines from the current source file
print	display the value of a variable or expression
whereis	display all environmental pathnames for a symbol
which	display the environmental pathname for a symbol
whatis	display the declaration for a symbol
regs	display the contents of the scalar registers
vregs	display the contents of the vector registers
where	display the call stack
dump	display the image of the core dump
set	display or change the *pmd* options for a core dump
func	display or change the current routine
file	display or change the current source file
assign	assign a new value to a variable
/	search forward in the source file for a pattern
?	search backward in the source file for a pattern

These commands are described in the following sections as they pertain to specific tasks.

2.5.1 Displaying Source Code: *list*

The *list* command lists lines from the current source file. Entering the command in the following format displays lines in the current source file from the first specified line number to the second specified line number, inclusive (if you do not specify line numbers, the next 10 lines are listed):

list [**eventlines**] [*starting_program_line-number* [[,] *ending_program_line-number*]]

To list 10 lines from the current position, type **list**, as shown in the next screen.

```
(csd) list
  364*          Display_next_frame ();
  365*          testch = getc (in_file);
  366*          switch (testch)
  367           {
  368           case 'b':
  369*              Determine_branch ();
  370*              break;
  371           case 'c':
  372*              Continue_with_frame ();
  373*              break;
(csd)
```

You can also display a range of lines by specifying the starting and ending line numbers. The following example shows how to display lines 242-257.

```
(csd) list 242,257
 242*      while (valid_answer == FALSE)
 243       {
 244*          valid_answer = TRUE;
 245*          printf (teststring);
 246*          gets (answer);
 247*          if ((answer[0] == 'a') || (answer[0] == 'A'))
 248           {
 249*              choice = 1;
 250           }
 251*          else
 252*          if ((answer[0] == 'b') || (answer[0] == 'B'))
 253           {
 254*              choice = 2;
 255           }
 256*          else
 257*          if ((answer[0] == 'c') || (answer[0] == 'C'))
(csd)
```

To display the source lines surrounding a particular *routine*, specify the routine name as a parameter to the *list* command, as shown in the next screen. The lines surrounding the declaration of *Create_frame_index* are displayed.

```
(csd) list Create_frame_index
 201            later.
 202   */
 203
 204
 205   Create_frame_index ()
 206*  {
 207*      total_frames = 0;
 208*      in_file = fopen (lesson_name, "r");
 209*      while (feof (in_file) == 0)
 210       {
 211*          testch = getc (in_file);
(csd)
```

To display the source line in the current context (the last instruction executed), use the command **list .** as shown in the next screen.

```
(csd) list .
 211*          testch = getc (in_file);
(csd)
```

The *list* command precedes each line with its line number and follows the line number with an asterisk for lines at which you can set an eventpoint. In the following listing, only lines 352 and 357 are invalid positions to set eventpoints. You can set eventpoints at any of the remaining lines.

```
(csd) list
  352    main ()
  353*   {
  354*       stop_program = FALSE;
  355*       Get_lesson ();
  356*       while (stop_program == FALSE)
  357        {
  358*          Create_frame_index ();
  359*           next_frame = 0;
  360*           in_file = fopen (lesson_name, "r");
  361*           stop_lesson = FALSE;
(csd)
```

By using the keyword **eventlines**, you can display only those lines at which you can set an eventpoint, as shown in the next example.

```
(csd) list eventlines
  353*   {
  354*       stop_program = FALSE;
  355*       Get_lesson ();
  356*       while (stop_program == FALSE)
  358*          Create_frame_index ();
  359*           next_frame = 0;
  360*           in_file = fopen (lesson_name, "r");
  361*           stop_lesson = FALSE;
(csd)
```

Notice that the command does not list the next ten eventlines; it lists only the number of eventlines contained in the specified number of source lines (10, in this case).

2.5.2 Displaying Values: *print*

Use the *print* command to display the values of variables, symbol addresses, and expressions. The format for *print* is

> **print** *expression* [, *expression* ...]

where *expression* is formed from operands (constants, variables, and function calls), operators, and parentheses. You can use basic arithmetic operations on the constants and variables within the expression. Permissible operators are the address operator (&), the dereference operator (*), the arithmetic operations (unary +, unary -, +, -, *, /, **div**, **mod**), the logical operators (**not**, **and**, **or**), and the relational operators (<, <=, >, >=, =, ==, <>, !=).

The next screen illustrates four print commands.

```
(csd) print answer
"a"
(csd) print choice
1
(csd) print choice + 6
7
(csd) print choice + (choice * 2)
3
(csd)
```

The *print* command displays the current value of the selected expression. You must qualify non-unique variables. You can use the field reference operator (.) with pointers and records, making the C operator "->" unnecessary (although it is supported).

2.5.2.1 Index and Subranges

To display subranges of arrays, specify as an index a lower and upper bound separated by a colon. The next screen displays elements 1-4 from the following C-language structure:

```
struct frame_index
{
 int frame_number;
 int frame_location;
}

struct frame_index frame_map[MAXCHARS];
```

```
(csd) print frame_map[1:4]
(
(frame_number = 0, frame_location = 50)
(frame_number = 100, frame_location = 606)
(frame_number = 101, frame_location = 1580)
(frame_number = 102, frame_location = 2532)
)
(csd)
```

In FORTRAN, all indices are included with one set of parentheses. If one dimension of a multi-dimensional array element reference specifies a subrange, then all other dimensions must specify subranges, even if the desired subrange is only one element long. For example,

print a(1,1:4)

generates a syntax error; a(1:1,1:4) is the correct subrange reference.

In C, each index is enclosed in its own set of brackets, so you can give any combination of simple indices and subranges for a multi-dimensional C array. For example, for a three-dimensional array **ca**,

print ca [1] [2:3] [4:5]

is a legal print statement, which prints four elements of the array. Notice that an index and two subranges are used.

A subrange cannot be followed by qualifying information. For example, if **st** is an array of structures with member **ia**, the command

print st[1:5].ia

is not permitted.

2.5.2.2 Type Transfer

csd can display the contents of a variable in a form other than its declared type. For example, you can display an integer variable as though it were of type *float*. This process is called *type transfer*. Two methods are available for routines written in C, explicit and implicit types.

In the first method, the name of the type is explicit:

print *type* (*expression*)

or

print *expression* \ *type*

The value of *expression* is printed as though it were of type *type*. *csd* does not really convert the value as does type casting in C. If the lengths of the *expression* and the new *type* are different, a sufficient number of bytes is used to display the *expression*, beginning with the most significant byte of the expression value. The length of an expression that is a variable name is the length of that variable (if the expression is *count*, its length is the same as the length of its type).

Integer expressions occupy four bytes. Real expressions occupy eight bytes. Complex expressions occupy 16 bytes.

Valid *types* include the simple C types (int, float, char, double), user defined typedefs, and structure or union types. For a declared structure or union type, use $$*tag*, where *tag* is the structure or union tag or just the *tag* if its name is unique.

The following screen shows how to display the variable *testch*, which is declared as type char, as an integer.

```
(csd) print testch
'b'
(csd) print testch\int
1644167168
(csd)
```

In the second method, the type is implicit; that is to say, the type is implied from the type of another variable.

print *expression* \ *variable*

The *expression* is transferred to the type of *variable*.

For example, the following screen shows how to display *testch* (type char) as if it were the same type as *count* (type int).

```
(csd) print testch
'b'
(csd) print testch\count
1644167168
(csd)
```

Type transfer in FORTRAN is only supported by using the implicit method, as described above. For example, assume that *ivar* is a variable of type integer*4 and *fvar* is a variable of type real*4. Then the command

print *ivar* \ *fvar*

prints the value of *ivar* as though it were of type real*4.

The *print* command also provides the capabilities listed below:

- Based on the expression or variable type, *csd* automatically formats the value. This includes arrays and structures (in C). *csd* limits the data for a *print* statement to 40,000 bytes, or 5000 real*8 words. To display larger arrays use several *print* commands with successive subrange references.

- Dereferencing pointers (in C). Statements of the form

  print **pointer_variable*

  print the value pointed to by the *pointer_variable* in its declared type, rather than the value of the *pointer_variable*.

- Printing string literals. The command

  print "*string*"

  prints the *string* literal, thereby providing the capability to build command files to format output, construct table headings, and monitor processing of command files.

- Displaying the address of a symbol. You can determine the address of a particular variable with the *print* command by preceding the variable name with an ampersand (&). For example, to display the hexadecimal address of *main*, type

  print &main

  Addresses may be expressions made up of other addresses, the operators "+" and "–", and indirection (unary "*").

- Displaying contents of specific registers. The command

  print *$regname*

  displays the contents of the register *regname*. Appendix A contains a list of the register names recognized by *csd*.

Figure 2-1 illustrates the basic print capabilities using a C declaration.

Figure 2–1: Examples of the *print* Command

```
struct node {
  short field1;
  struct node *forw;
  char *code;
};

struct node *foo;
```

print foo	displays the hexadecimal value of the pointer
print *foo	displays structure data pointed to by *foo*
print foo->forw	displays the value of the *forw* pointer field
print *foo.forw	dereferences the *forw* pointer
print foo->code	displays in ASCII the characters pointed to by *code*
print foo, *foo	displays both the value of the pointer and its data
print &foo	displays the address of the variable *foo*

Character strings are printed in ASCII, addresses and pointers in hex, and variables in decimal. *csd* uses the same syntax as C to refer to variables. *csd* also uses the same type of structure-qualified field names as C.

2.5.3 Displaying Symbol Information: *which, whereis, whatis*

Use the *which* command to list the default environmental pathname associated with a particular variable. To use this command, type

which *variable*

The following screen shows the default environmental pathname for the variable *testch*. In the example, the variable is defined as a global variable in the program (specified by the $b0).

```
(csd) which testch
$b0.testch
(csd)
```

Occasionally a program may have multiple declarations of a variable, each having a distinct scope. A counter variable is an example of a variable that is sometimes declared within the routine that uses it. Use the *whereis* command to list all the environmental pathnames associated with a variable. To use this command, type

whereis *variable*

The next example shows all declarations of the variable *testch*. The three declarations occur in the routines *Get_response*, *Get_lesson*, and in the outer block of the program.

```
(csd) whereis testch
cbt.Get_response.testch cbt.Get_lesson.testch $b0.testch
(csd)
```

The *whatis* command displays the definition of a particular symbol. The format of the command is

whatis *variable* | *type* | *routine*

where

variable is the name of a program variable. The variable's type is displayed (for example, float or int in C; integer*2 or real*8 in FORTRAN).

type is a valid C language *type* including the simple C types (int, float, char, double), user defined typedefs, and structure or union types. For a declared structure or union, use $$*tag*, where *tag* is the structure or union tag, or just the *tag* if its name is unique.

routine is the name of a C or FORTRAN routine. The routine's declaration is displayed, including the names and types of the parameters.

The following screen shows examples of determining the declarations of various symbols in the executable program.

```
(csd) whatis frame_map
struct frame_index frame_map[200];
(csd) whatis Get_response
int Get_response()
(csd) whatis testch
char testch;
(csd)
```

From the previous listing, you can determine that *frame_map* is an 200-element array of type *frame_index*, that *Get_response* is a function that returns an integer value, and that *testch* is a character variable.

The C language and the loader permit a programming practice that may confuse a *csd* user. You can declare a global variable in each of several source files, and the variable can have a different type in each file. If you initialize the variable only once (or never), the loader allocates space for the largest type for the variable. (If you initialize the variable more than once, the loader complains of multiple definitions, even if the types agree). If each source module is compiled for debugging and the collection of files is loaded together, *csd* sees several different declarations for the same global variable when it reads the executable file. When you ask about such a variable with the *whatis* command, *csd* reports the type of the first declaration seen in the executable file. The declaration is determined by the order of the object (*.o*) files. Type transfers can be used to display the value of the variable in any desired form.

csd prints a warning message once for each extra type seen for such a global variable. The message displays the variable name and the source module containing the new declaration.

2.5.4 Displaying the Current Routine Name: *func*

The *func* command displays and changes the current routine to the *routine* specified, and the current source file to the pathname of the file containing the *routine*. The command has the following format

func [*routine*]

If you do not specify a routine, *csd* displays the current routine name. At startup, *csd* initializes the current function to the main routine.

You can use the *func* command to look at various routines in the program. For example, you can change the current routine and list source code; *csd* lists the code at the beginning of the new current routine. The next screen demonstrates how to use *func* to look at various parts of the code.

```
(csd) func
main
(csd) list
  360*          Create_frame_index ();
  361*          next_frame = 0;
  362*          in_file = fopen (lesson_name, "r");
  363*          stop_lesson = FALSE;
  364*          while (stop_lesson == FALSE)
  365           {
  366*              Display_next_frame ();
  367*              testch = getc (in_file);
  368*              switch (testch)
  369              {
(csd) func Get_response
(csd) list
  236
  237           it is called by Determine_branch routine.
  238   */
  239
  240   Get_response ()
  241*  {
  242   char testch;
  243*      valid_answer = FALSE;
  244*      while (valid_answer == FALSE)
  245       {
(csd)
```

Changing the current routine may also change the default environmental pathname for a variable (if the variable has multiple declarations). In the next screen, the first command shows the different declarations of *testch* and the second command shows which instance of *testch* is current. When the third command changes the current routine, the default pathname of *testch* also changes, as verified by the last command.

```
(csd) whereis testch
cbt.Get_response.testch cbt.Get_lesson.testch $b0.testch
(csd) which testch
$b0.testch
(csd) func Get_response
(csd) which testch
cbt.Get_response.testch
(csd)
```

When you change the current routine to one that is located in a different file, the current file automatically changes.

The C program in Figure 2-2 uses three different instances of the variable *i*. Figure 2-3 shows how to use different *csd* commands to change the current context during program execution.

Figure 2–2: Sample C Program Using Multiple Environments

```
1     main()
2     {
3        int i;  /* the first instance of i */
4        int j;
5
6        i = 5;
7        printf ("sample.main.i=%d\n",i);
8        for (j=1;j<i;j++)
9        {
10           int i;  /* the second instance of i */
11           i=i+j;
12           printf ("sample.main.$b1.i=%d\n",i);
13        }
14
15        i=foo(i);
16    }
17
18    foo(i)
19    int i;  /* the third instance of i */
20    {
21        i=i*100;
22        printf ("sample.foo.i=%d\n",i);
23        return(i);
24    }
```

Figure 2–3: Output Illustrating Environment in *csd*

```
% cc -g sample.c
% csd sample
Convex Symbolic Debugger
Type 'help' for help.
reading symbolic information ...
(csd) whereis i
 sample.foo.i sample.main.$b1.i sample.main.i
(csd) stop at 7
 [1] stop at 7
(csd) run
 [1]
stopped in main at line 7
   7*     printf("sample.main.i = %d0, i);
(csd) func
main
(csd) which i
sample.main.i
(csd) print i
5
(csd) print foo.i
"i" is not active
(csd) step 3
sample.main.i = 5
stopped in $b1.main at line 12
  12*     printf("sample.main.$b1.i = %d0, i);
(csd) which i
sample.main.$b1.i
(csd) print i
1
(csd) print main.i
5
(csd) stop in foo
[3] stop in foo
(csd) cont
sample.main.$b1 = 1
sample.main.$b1 = 3
sample.main.$b1 = 6
sample.main.$b1 = 10
[3] stopped in foo at line 20
  20* {
(csd) func
foo
(csd) which i
sample.foo.i
(csd) print i
5
(csd) func main
(csd) func
main
(csd) which i
sample.main.i
(csd) quit
%
```

2.5.5 Displaying the Current Source File Name: *file*

The *file* command displays and changes the current source file to *filename*. You can use this command to view source code in other files. The *file* command has the following format:

file [*filename*]

If you do not specify an argument, *csd* displays the current source file pathname. At startup, *csd* initializes the current file to the first file loaded.

The following screen illustrates the *file* command.

```
(csd) file
cbt.c
(csd) file cai.c
(csd) file
cai.c
(csd)
```

The current routine does not automatically change when you change the current file.

2.5.6 Displaying Register Contents: *regs, vregs*

To display the contents of all the scalar and address registers, use the *regs* command as shown in the following screen.

```
(csd) regs
pc=800012ae psw= 2108080
sp=ffffcd74 a1=800084a0 a2=8001d000 a3=8000d001
a4=00000000 a5=00000000 ap=ffffcd8c fp=ffffcd78
s0=0000000000000000 s1=000000008000d000 s2=0000000000000302 s3=0000000000000046
s4=000000008000a29c s5=0000000080008400 s6=00000000ffffcd74 s7=0000000080008139
(csd)
```

To display the contents of all the vector registers, type

vregs [*index*]

If the vector registers are empty, *vregs* displays all zeros. *csd* displays the output of these commands in hexadecimal. It also condenses duplicate array elements in the output as shown in the next screen. Elements 10-20 of vector register 5 contain the value 1.0 (double-precision floating-point format).

```
(csd) vregs
vl=00000040 vs=00000008
vm=00000000000000000000000000000000

v0[000] through v0[127] = 0000000000000000

v1[000] through v1[127] = 0000000000000000

v2[000] through v2[127] = 0000000000000000

v3[000] through v3[127] = 0000000000000000

v4[000] through v4[127] = 0000000000000000

v5[000] through v5[007] = 0000000000000000
v5[008]=0000000000000000, 0000000000000000, 4010000000000000, 4010000000000000
v5[012] through v5[019] = 4010000000000000
v5[020]=4010000000000000, 0000000000000000, 0000000000000000, 0000000000000000
v5[024] through v5[127] = 0000000000000000

v6[000] through v6[127] = 0000000000000000

v7[000] through v7[127] = 0000000000000000

(csd)
```

When the *vregs* command displays the values of the vector registers, it displays them in groups of four. That is why the eleven elements with non-zero values are displayed across three lines.

You can display the contents of a single vector register by specifying an *index*.

2.5.7 Displaying the Call Stack: *where*

csd automatically keeps track of where the program is executing. The current context is the current address, the current routine, and the current source file. When the program is not executing, you can examine the current context using the *where* command. To use this command, type

> **where** [*integer*] [>*filename*]

The *where* command displays a stack trace of the routine calls that led to the current program state. The output for each call displays the name of the routine, its arguments, the source line number containing the call, and the source file containing the routine.

The following screen shows how to use the *where* command to display the call stack. The display shows that the program is suspended in the routine, *Get_response*. It also shows that *Get_response* is called from *Determine_branch*, which is called from *main*.

```
(csd) where
Get_response, line 247 in "cbt.c"
Determine_branch, line 296 in "cbt.c"
main(0x1, 0xffffcdac, 0xffffcdb4), line 371 in "cbt.c"
(csd)
```

Use the optional *integer* parameter to display a specific number of runtime stack frames or routines. Thus, *where 1* displays the current address, current function, and current source file. You can also direct output from the *where* command into a file with the ">" symbol. For example, the command

where 3 > temp

stores the top three elements of the call stack in the file, *temp*.

2.5.8 Displaying a Core Dump: *dump, set*

The *dump* command can be used on active programs as well as programs that end abnormally. For instance, when running *csd* on a program, you can use *dump* to examine the variable values up to a set breakpoint. Or, if your program has aborted, you can use *dump* to examine the state of the program when the exception occurred.

To use the *dump* command, type

dump [> *filename*]

where *filename* is the file to which you redirect standard output.

There are several different listings that the *dump* command can generate. By default, the *dump* command automatically generates a listing consisting of a stack backtrace, the signal that caused the abort, and the approximate source code location of the exception. Output goes to *stdout*.

Available options for formatting output and their default values (in bold) are shown in Table 2-1. To set a corresponding *pmd* option for *dump*, type

set [deref_aregs | dump_lfmt | dumpsrc | dumpvregs = *logical_constant*]

where *logical_constant* is one of **true**, **false**, **.true.**, or **.false.**. Use the logical constant appropriate for the language.

When no arguments are specified, *csd* displays the current values for all arguments.

You can also set the *num_elements* option using the command

set *num_elements* = *number* [, *number* ...]

where *number* is an integer. You may specify a maximum of seven *numbers*.

Table 2-1: Dump Options

pmd Option	Equivalent *set* Command
-a	set deref_aregs = true/**false**
-d	set num_elements = n1, n2, ...
-l	set dump_lfmt = **true**/false
-s	set dump_lfmt = true/**false**
-S	set dumpsrc = **true**/false
-v	set dumpvregs = true/**false**

2.5.9 Assigning Values: *assign*

The *assign* command lets you assign a value to a variable while you are debugging a program. This is often useful when you notice that a variable has an unexpected value when you print it. You can assign it the value you think it should have, then continue debugging. Assigning specific values to program variables may also be useful when you want to test a specific part of a program.

The format of the *assign* command is

assign *variable* = *value*

where *variable* and *value* are of the same type.

The next screen shows how to assign values to integer and character variables. After the assignments are made, the new values are displayed using the *print* command to verify the assignments.

```
(csd) assign i = 1234
(csd) assign testch = 'H'
(csd) print i,testch
1234 'H'
(csd)
```

You can also assign values to array and structure elements, as demonstrated in the following screen.

```
(csd) assign branch[0] = 4321
(csd) assign frame_map[1].frame_location = 3000
(csd) print branch[0],frame_map[1].frame_location
4321 3000
(csd)
```

Figure 2-4 illustrates the *assign* command in a *csd* session for a FORTRAN program. Figure 2-5 illustrates the command being used while debugging a C program.

Figure 2–4: Example of the *assign* Command in FORTRAN

```
% csd a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...
(csd) list
    1           REAL A / 5.0 /
    2           INTEGER I / 4 /
    3           LOGICAL L / .TRUE. /
    4
    5*          WRITE (6,*) A, I, L
    6
    7*          WRITE (6,*) A, I, L
    8*          END
(csd) stop at 7
[1] stop at 7
(csd) run
    5.000000              4 T
[1] stopped in MAIN_ at line 7
    7*           WRITE (6,*) a, i, l
(csd) assign a = 25.0
(csd) assign i = 16
(csd) assign l = .false.
(csd) cont
   25.00000              16 F

execution completed
(csd) quit
%
```

Figure 2–5: Examples of the *assign* Command in C

```
% csd a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...
(csd) list 1 11
    1   main()
    2   { int i ;
    3     float f ;
    4
    5*    i = 1 ;
    6*    f = 5.0 ;
    7
    8*    printf (" %d, %f\n",i,f );
    9
   10*    printf (" %d, %f\n",i,f );
   11*  }
(csd) stop at 10
[1] stop at 10
(csd) run
 1, 5.000000
[1] stopped in main at line 10
   10*    printf (" %d, %f\n",i,f );
(csd) assign i = 100
(csd) assign f = 100.0
(csd) cont
 100, 100.000000

execution completed
(csd) quit
%
```

2.5.10 Displaying Raw Memory

To inspect memory contents, use either of the following machine-level requests:

address[*,count*]?*mode*
. [*,count*]?*mode*

The first format displays the contents of memory beginning with the address specified. The second format displays the contents of memory starting at the current memory location. In both formats, *count* specifies the number of items to be displayed. *mode* specifies how to display memory; if you omit it, *csd* uses the previous mode specified. The initial mode is "x". Preface hexadecimal addresses with "0x". To list assembly-language instructions, use mode "i". Table 2-2 contains the valid *csd* output modes.

To display ten assembly-language instructions starting at *main*, use the command shown in the next screen. The & preceding *main* instructs *csd* to look for the address of *main* and start displaying data from that location.

```
(csd) &main,10?i
0x80001a70    main+0          sub.w  #0,a0
0x80001a72    main+2          sub.w  s0,s0
0x80001a74    main+4          st.w   s0,write
0x80001a7a    main+a          mov    a0,a6
0x80001a7c    main+c          pshea  0
0x80001a80    main+10         calls         Get_lesson
0x80001a86    main+16         add.w  #4,a0
0x80001a88    main+18         ld.w   12(fp),a6
0x80001a8c    main+1c         ld.w   write,s0
0x80001a92    main+22         eq.w   #0,s0
(csd)
```

You can also display data from memory by specifying a hexadecimal address. The next example displays ten instructions starting at 0x80001a70 (the hexadecimal address for *main*).

```
(csd) 0x80001a70,10?i
0x80001a70    main+0          sub.w  #0,a0
0x80001a72    main+2          sub.w  s0,s0
0x80001a74    main+4          st.w   s0,write
0x80001a7a    main+a          mov    a0,a6
0x80001a7c    main+c          pshea  0
0x80001a80    main+10         calls         Get_lesson
0x80001a86    main+16         add.w  #4,a0
0x80001a88    main+18         ld.w   12(fp),a6
0x80001a8c    main+1c         ld.w   write,s0
0x80001a92    main+22         eq.w   #0,s0
(csd)
```

You can also display specific amounts (bytes, words, and so on) of data from memory. In the next example, the contents of the array *branch* are displayed; *branch* is a four-element array of integers, so each element is represented as a word in memory. The **d** mode instructs *csd* to display the data in decimal format.

```
(csd) &branch,4?d
8000a870:  4321 100 101 201
(csd)
```

csd supports several different modes for displaying data from memory; they are illustrated in Table 2-2.

Table 2-2: *csd* Output Modes

Mode	Meaning
b	Byte in octal (8 bits).
c	Byte as a character (8 bits).
d	Word in decimal (32 bits).
D	Long long word in decimal (64 bits).
f	Single-precision real number (32 bits).
F	Double-precision real number (64 bits).
g	Single-precision real number using C's printf %g format.
G	Double-precision real number using C's printf %g format.
i	Prints the machine instruction.
o	Word in octal (32 bits).
O	Long long word in octal (64 bits).
s	String of characters terminated by a null byte.
x	Word in hexadecimal (32 bits).
X	Long long word in hexadecimal (64 bits).

2.5.11 Searching for a Pattern: /, ?

You can search a source file for any given pattern while in *csd*. You can search a file in both forward and backward directions. The formats for these commands are:

/*regular expression*[/]   (for forward searches)
?*regular expression*[?]   (for backward searches)
/   (search forward for next occurrence)
?   (search backward for next occurrence)

The next screen shows how you can use the search commands to move through the source file. In this example, you want to search for the string, "stop". The first command searches for the first occurrence of "stop" after the current position (at the beginning of *main* in this case). The second and third commands repeat the search.

```
(csd) /stop
  358*      while (stop_program == FALSE)
(csd) /
  363*          stop_lesson = FALSE;
(csd) /
  364*          while (stop_lesson == FALSE)
(csd)
```

Use the *?* command to search backward for a string. The following example demonstrates using the *?* command to search backward for "stop_lesson = TRUE".

```
(csd) ?stop_lesson = TRUE
  142*      stop_lesson = TRUE;
(csd) ?
  128*      stop_lesson = TRUE;
(csd)
```

2.6 Controlling Program Execution

Executable programs are controlled by *csd* when you are debugging them. The *csd* execution commands control the flow of the program, including normal execution, single stepping, and continuing execution after it is suspended.

Use the following commands to execute program instructions.

run	execute a program
rerun	rerun a program
step	execute one source line, entering any called routines
next	execute one source line, treating called routines atomically
stepi	execute one assembly instruction, entering any called routines
nexti	execute one assembly instruction, treating called routines atomically
cont	resume program execution
source	execute *csd* commands stored in a file
call	execute a user-defined routine

2.6.1 Executing a Program: *run, rerun*

Once you have invoked *csd* you can use the *run* command to start executing the program to be debugged. If the program is currently executing, the command restarts the program from the beginning.

The *run* command has the following format:

run [*args*] [<*filename*] [>[&]*filename*]

where

args	are the arguments passed as command line arguments to the program being debugged.
<*filename*	is the name of the file from which data is read (*stdin*).
>*filename*	is the name of the file to which data is written (*stdout*). Error messages (*stderr*) are displayed on the screen.
>&*filename*	is the name fo the file to which all output is written (*stdout* and *stderr*).

The following screen shows how the *run* command restarts the program. The program is suspended at a breakpoint in *Get_response*. Breakpoints are set at *main* and *Get_response*. When the *run* command is executed, the program starts execution from the beginning of the program and stops when it enters the *main* routine (the location of the first breakpoint).

```
[2] stopped in Get_response at line 241
  241*  {
(csd) run
[1] stopped in main at line 355
  355*  {
(csd)
```

You can redirect standard input, standard output, and standard error for the program by using the *<filename*, *>filename*, and *>&filename* options, respectively. If you invoke the *run* command mor than once, the program's variables are re-initialized with each invocation.

For instance, suppose a program that summarizes accounting information for a specific user requires two input arguments: a user name and the file pathname. Start the program using the *run* command as follows:

run uname /usr/adm/wtmp

where *uname* is the user name, and */usr/adm/wtmp* is the pathname of the accounting file. In this example, the standard input and output are not redirected.

You can redirect the input and output as in

run <input.data >listing

In this example *input.data* is used as standard input when the program executes. The output from the program is written to *listing*.

To run the program again, either use the *run* command or the *rerun* command:

rerun [*args*] [*<filename*] [>[&]*filename*]

rerun is identical to *run*, except when no arguments are specified. If no arguments are specified, the argument list (if any) from the last *run* command is used.

In the next screen, the executable image is run using *data* as the input file. The *rerun* command executes the program again, using the same input file.

```
(csd) run < data
This is a       test of the     detab program.
It converts tabs        to spaces.

execution completed
(csd) rerun
This is a       test of the     detab program.
It converts tabs        to spaces.

execution completed
(csd)
```

2.6.2 Executing a Single Source Line: *step, next*

Use the *step* command to execute one or more high-level language source lines. The format of the *step* command is as follows:

step [*count*]

where *count* is an optional step count (the default is one source line). If *count* is greater than 1, the *step* command executes the specified number of lines without stopping for breakpoints. If you want to stop at breakpoints, use the *cont* command. You cannot *step* until the program has been started with a *run* command. The *step* command always stops on the next line at which you can set an eventpoint (marked with an asterisk).

The following example shows how the *step* command can be used to execute a series of single instructions. First, the *list* command shows the current position after the breakpoint is reached. Next, the two *step* commands execute the next two source lines (as verified by the line numbers). Finally, the last command, *step 3*, executes three source lines before suspending execution.

```
[2] stopped in Get_response at line 241
  241*  {
(csd) list
  242   char testch;
  243*       valid_answer = FALSE;
  244*       while (valid_answer == FALSE)
  245        {
  246*           valid_answer = TRUE;
  247*           printf (teststring);
  248*           gets (answer);
  249*           if ((answer[0] == 'a') || (answer[0] == 'A'))
  250            {
  251*               choice = 1;
(csd) step
stopped in Get_response at line 243
  243*       valid_answer = FALSE;
(csd) step
stopped in Get_response at line 244
  244*       while (valid_answer == FALSE)
(csd) step 3
stopped in Get_response at line 248
  248*           gets (answer);
(csd)
```

When the next instruction to be executed is a subroutine call, the *step* command steps *into* the subroutine. The following screen demonstrates this. The *list* command shows that the instruction at line 357 is a subroutine call. When the *step* command is issued at that line, the current position moves to the subroutine *Get_lesson*.

```
[1] stopped in main at line 355
  355*  {
(csd) list
  356*      stop_program = FALSE;
  357*      Get_lesson ();
  358*      while (stop_program == FALSE)
  359       {
  360*          Create_frame_index ();
  361*          next_frame = 0;
  362*          in_file = fopen (lesson_name, "r");
  363*          stop_lesson = FALSE;
  364*          while (stop_lesson == FALSE)
  365          {
(csd) step
stopped in main at line 356
  356*      stop_program = FALSE;
(csd) step
stopped in main at line 357
  357*      Get_lesson ();
(csd) step
stopped in Get_lesson at line 160
  160*  {
(csd)
```

The *next* command also lets you execute source lines. However, *next* treats a routine call as a single statement and stops at the next line in the calling routine (provided that a user-specified event doesn't occur diring execution of the routine), while *step* stops at the first line of the called routine. The format for the *next* command is as follows:

next [*count*]

where *count* is an optional step count (the default is one source line). If *count* is greater than 1, the *step* command executes the specified number of lines without stopping for breakpoints. If you want to stop at breakpoints, use the *cont* command. You cannot *step* until the program has been started with a *run* command. The *step* command always stops on the next line at which you can set an eventpoint (marked with an asterisk).

The following example is similar to the previous one, except that the *next* command is used instead of *step*. Notice that when the current position is at line 357 and *next* executes, the current position moves to line 358. The subroutine *Get_lesson* is executed and treated as a single instruction by *csd*.

```
[1] stopped in main at line 355
  355*  {
(csd) list
  356*      stop_program = FALSE;
  357*      Get_lesson ();
  358*      while (stop_program == FALSE)
  359       {
  360*          Create_frame_index ();
  361*          next_frame = 0;
  362*          in_file = fopen (lesson_name, "r");
  363*          stop_lesson = FALSE;
  364*          while (stop_lesson == FALSE)
  365           {
(csd) next
stopped in main at line 356
  356*      stop_program = FALSE;
(csd) next
stopped in main at line 357
  357*      Get_lesson ();
(csd) next
stopped in main at line 358
  358*      while (stop_program == FALSE)
(csd)
```

2.6.3 Executing a Single Instruction: *stepi, nexti*

Use the *stepi* command to execute instructions at the assembly-language level. Because source-level stepping makes little sense when you are debugging assembly-language modules, use the *nexti* command to step over routine calls, treating each routine as a single step. Both of these commands function similarly to their source-level counterparts, *step* and *next*. Both commands accept an optional step count. These commands are useful for debugging embedded assembly language modules or modules compiled without the *-g* or *-db* option. The formats for these commands are:

> **stepi** [*count*]
> **nexti** [*count*]

If no parameter is specified, one machine instruction is executed.

The following example illustrates the *stepi* command.

```
stopped in main at line 358
  358*      while (stop_program == FALSE)
(csd) stepi
stopped in main at 0x80001a92
0x80001a92    main+22          eq.w   #0,s0
(csd) stepi
stopped in main at 0x80001a96
0x80001a96    main+26          jmps.f main
(csd)
```

Using *stepi* when the current position is at a subroutine causes *csd* to step into the subroutine, as shown in the next example. Notice that both the hexadecimal and relative addresses change when stepping into *Get_lesson*.

```
(csd) stepi
stopped in main at 0x80001a7a
0x80001a7a    main+a            mov    a0,a6
(csd) stepi
stopped in main at 0x80001a7c
0x80001a7c    main+c            pshea  0
(csd) stepi
stopped in main at 0x80001a80
0x80001a80    main+10           calls          Get_lesson
(csd) stepi
stopped in Get_lesson at 0x80001280
0x80001280    Get_lesson+0      sub.w  #4,a0
(csd) stepi
stopped in Get_lesson at 0x80001282
0x80001282    Get_lesson+2      mov    a0,a6
(csd)
```

The next screen steps through the same area of code, except that it demonstrates the *nexti* command. Notice that the call to *Get_lesson* appears to have been skipped. Since the routine is executed as a "single instruction," the program stopped after it returned from *Get_lesson*.

```
(csd) nexti
stopped in main at 0x80001a7a
0x80001a7a    main+a            mov    a0,a6
(csd) nexti
stopped in main at 0x80001a7c
0x80001a7c    main+c            pshea  0
(csd) nexti
stopped in main at 0x80001a86
0x80001a86    main+16           add.w  #4,a0
(csd) nexti
stopped in main at 0x80001a88
0x80001a88    main+18           ld.w   12(fp),a6
(csd)
```

Note that source-level stepping makes little sense when you are debugging assembly-language modules.

2.6.4 Resuming Program Execution: *cont*

To resume execution of the program after a stop (such as encountering a breakpoint), type

cont

The next screen illustrates the *cont* command. In the example, two breakpoints are set: one at *main* and one at *Get_lesson*. Execution starts with the *run* command and stops when the program enters *main*. The *cont* command resumes execution, which is suspended when entering the *Get_lesson* routine.

```
(csd) run
[11] stopped in main at line 355
  355*  {
(csd) cont
[12] stopped in Get_lesson at line 160
  160*  {
(csd)
```

You cannot continue execution if a standard program exit or abnormal termination has occurred. You can use the *run* command, however, to start the program again. *csd* captures control when the program attempts to exit, thereby letting letting you examine the state of the program.

Another form of the command is

cont *signum* | *signame*

which lets you continue as if the signal *signum* or *signame* had occurred. A list of the signal names and their corresponding numbers is located in */usr/include/signal.h* and in the *signal*(3c) manual page. When specifying a *signame*, omit the SIG prefix.

For more information on signal handling, refer to section 2.6, "Handling Signals."

2.6.5 Executing Commands From a File: *source*

The *source* command reads and executes *csd* commands stored in a file. To use this command, type

source *filename*

on the command line. You can also use the *source* command to set the program to a known state from which debugging may begin.

The next screen example shows the *source* command executing the *csd* commands in the file *com-list*, which contains the following:

```
stop in main
run
next 3
list
```

```
(csd) source com-list
[1] stopped in main at line 19
   19*  {
stopped in main at line 22
   22*          signal(SIGUSR1, sigusr1);
   23           for (;;) {
   24*                  ++i;
   25*                  if (i > 32767)
   26*                          i = 0;
   27*          }
   28*  }
(csd)
```

Notice that a breakpoint is set at *main*; program execution starts and then stops at the breakpoint. Next, three instructions are executed (lines 19-21) and the source code is listed, starting at line 22 (the source line following the breakpointed line).

2.6.6 Executing a Compiled Routine: *call*

The *call* command executes a user-defined *routine* that is linked into the program being debugged. This command can be an invaluable interactive debugging tool. For instance, a program containing complex data structures linked together in a list can be tedious to examine with the print command, especially as the list grows in size. You could, however, write a small routine to traverse the list, link it into the program, and execute it automatically with the *call* command at any time during the debugging session. The form of the *call* command is

call *routine* ([*argument_list*])

If *routine* is compiled with a *-g* or *-db* option, *csd* verifies that the argument list has the correct number and types of arguments. If source information for the *routine* is unavailable, *csd* can't check the argument list. In this case, *csd* assumes that the argument list is correct and passes all arguments to the *routine* by value. Arrays and character string literals are passed by address.

Routines with no source information, including system and runtime library functions, are called by *csd* just as they are called by a C program. This may or may not work as desired. In particular, *csd* does not emulate the name translation and calling style conventions of the FORTRAN compiler (the *CONVEX FORTRAN User's Guide*). Therefore, you cannot call FORTRAN intrinsics from *csd*.

2.7 Handling Signals

In the UNIX environment, signals are defined as asynchronous events. Signals implement many different types of events, such as error exceptions, program interrupts, and program terminations.

Typically, programs that run in the UNIX environment contain signal catchers that instruct the program to take particular actions when receiving certain types of signals. For example, a signal catcher in a particular program might print a message when an error exception occurs. Unfortunately, because signals tend to occur asynchronously, programs containing signal handlers are difficult to debug. Difficulties arise because debugging demands that the program not run continuously. For example, signals sent when the program is paused at a breakpoint may be missed.

Two *csd* commands exist solely to simplify problems related to signals:

catch	display or change the list of signals to be caught by *csd*
ignore	display or change the list of signals to be ignored by *csd*

Another command that is useful when handling signals is the *cont* command, which resumes execution of a suspended program.

2.7.1 Specifying Signals to Catch: *catch*

The *catch* command traps signals before they are sent to the program. When a signal is received, *catch* prints a message and returns control to the command level. To use the *catch* command, type

catch [*signum* | *signame*]

at the command level. The argument *signum* is the number identifying the signal to be intercepted; *signame* is the name of the signal. A list of signal names and numbers is located in the include file */usr/include/signal.h* and in the *signal*(3c) manual page. When specifying a *signame*, omit the SIG prefix (refer to the SIGINT signal as INT).

Initially all signals are caught except SIGCONT, SIGCHLD, SIGALRM, SIGTSTP, SIGHUP, and SIGKILL. With no arguments, *catch* lists all signals to be caught, as shown in the following screen.

```
(csd) catch
INT QUIT ILL TRAP IOT EMT FPE BUS SEGV SYS PIPE TERM URG STOP
TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH LOST USR1 USR2
(csd)
```

2.7.2 Specifying Signals to Ignore: *ignore*

Use the *ignore* command to deactivate a signal.

ignore [*signum* | *signame*]

Like *catch*, the *ignore* command accepts the number or name of the signal to be ignored. The *ignore* command can also ignore any troublesome signal, even if it was not previously caught. When specifying a *signame*, omit the SIG prefix (refer to the SIGINT signal as INT).

Initially, the signals SIGCONT, SIGCHLD, SIGALRM, SIGTSTP, SIGHUP, and SIGKILL are ignored. With no arguments, *ignore* lists all signals to be ignored, as shown in the following screen.

```
(csd) ignore
HUP KILL ALRM TSTP CONT CHLD
(csd)
```

Because of the way *csd* controls execution of the program, it is not possible to debug applications using the signal SIGTRAP (see *signal* (3c)).

Because of the pipelined nature of the machine, the program counter (pc) may not correlate directly to the source line that caused the signal; normally the pc points to the next executable source line. If in doubt, stop the program on the source line of the previous executable statement and single step until the signal occurs (*step*, *next*, or *trace*, for example). The source line displayed for the signal is then correct.

2.7.3 Passing Different Signals: *cont*

If a signal is sent to a program that is being debugged, the program stops and the signal is really delivered to *csd.* When this happens, *csd* displays the signal that stopped execution and enters command mode. When the program is single-stepped or continued, the signal is passed on to the program by *csd.*

Using the *cont* command, you can pass a different signal to the program. The format of the command is

cont *signame* | *signum*

where *signame* and *signum* are the name or number of the signal to be passed (see *sigvec*(2)). This can be particularly useful when testing interrupt handling routines. If you don't to pass any signal to the program, continue execution with a signal of **0**.

The FORTRAN program shown in Figure 2-6 acts as a signal handler. If at any time the program receives a SIGTERM signal (#15) or a SIGUSR1 signal (#30), it calls the appropriate subroutine. Because the program executes an infinite loop, you can stop it by typing CTRL-C.

Figure 2–6: Sample Signal Handler Program in FORTRAN

```
      program main

      external termhandler
      external userhandler

      real x,y,z
      common /data/x,y,z

      i = signal(15,termhandler,-1)
      i = signal(30,userhandler,-1)

      write(*,*) 'Starting Loop'
10     continue
        x = x * 10.0
        y = y * 10.0
        z = z * 10.0
      goto 10
      end

      subroutine termhandler
      write(*,*) 'I am in the term handler routine'
      return
      end

      subroutine userhandler
      write(*,*) 'I am in the user handler routine'
      return
      end
```

The following screens demonstrate how to use the *cont* command to change a signal before passing it to the executable program. In each of these screens, a ^C indicates that a CTRL-C is typed while the program is executing.

In the following screen, *csd* is invoked and execution is started with the *run* command. While the program is running, a CTRL-C is typed and execution is suspended. The *cont* command then continues execution without changing the signal. When the executable program receives the SIGINT signal, the program terminates.

```
% csd signal
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...

(csd) run
Starting Loop
^C
Interrupt in MAIN_ at line 14
   14*                                              x = x * 10.0
(csd) cont
^C
Interrupt in MAIN_ at line 14
   14*                                              x = x * 10.0
(csd)
```

In the next screen, the same program is rerun and stopped in the same way. This time the *cont* command continues execution, but changes the signal to a SIGUSR1 signal. Notice that the appropriate subroutine is called, and execution continues until stopped by another interrupt.

```
(csd) run
^C
Starting Loop
^C
Interrupt in MAIN_ at line 14
   14*                                              x = x * 10.0
(csd) cont 30
I am in the user handler routine

Interrupt in MAIN_ at line 14
   14*                                              x = x * 10.0
(csd)
```

2.8 Manipulating Eventpoints

This section includes information on setting and deleting eventpoints using the *stop*, *trace*, and *when* commands.

For a statment that extends across multiple lines, you may specify an eventpoint on any line marked with an asterisk after the line number in a source listing. The *list* command displays this information, but it is also displayed whenever *csd* lists a source line after an event occurs (when a breakpoint is reached, for example).

When multiple statements appear on the same line, the eventpoint is located at the first statement only. No eventpoints can be specified for the remaining statements on the line. For example, in the following line of a source program, you can set an eventpoint at the statement $i = i + 1$ but not at $j = j * i$.

```
i = i + 1 ; j = j * i ;
```

csd maintains a list of all active events. When an eventpoint is set, *csd* displays it and its unique identification number. This number is called an *event index*. The list can be printed using the *status* command. An event can be removed from the list using the *delete* command.

Use the following commands to manipulate eventpoints.

status	display currently set eventpoints
stop	set a breakpoint at the source level
stopi	set a breakpoint at the assembly-instruction level
trace	set a tracepoint at the source level
tracei	set a tracepoint at the assembly-instruction level
when	execute *csd* commands when an event occurs
delete	delete an eventpoint

These commands are described in the following sections as they pertain to particular debugging tasks.

2.8.1 Displaying Eventpoints: *status*

To determine the unique *index* identifying the currently active eventpoints, use the *status* command. The format of the *status* command is

status [>*filename*]

csd routes output from the command to any filename used as an argument. If you do not specify a filename, output is routed to *stdout*. Output includes a list of eventpoints and their unique event indices.

The following screen contains output from a *status* command. Three breakpoints are currently set in the program and can be accessed by the appropriate index number.

```
(csd) status
[1] stop in main
[2] stop at 300
[3] stop in Get_response
(csd)
```

The breakpoint index numbers are incremented each time a new eventpoint is declared, but are not decremented when an eventpoint is deleted. After setting and deleting eventpoints several times during a debugging session, the index numbers may have gaps between them. The following example shows such a listing. In the *status* command output, four eventpoints are active: three breakpoints and a tracepoint. Notice that the index numbers are not sequentially ordered, indicating that some eventpoints have been deleted.

```
(csd) status
[1] stop in main
[3] stop in Get_response
[9]  at 290
[10] stop in Determine_branch
(csd)
```

2.8.2 Setting a Breakpoint: *stop, stopi*

Use the *stop* command to set a breakpoint. When *csd* encounters a breakpoint, it stops execution of the program and returns to command mode. *csd* commands are then used to display information for monitoring the progress of a program.

You can set breakpoints for the following events:

- at a specific line number
- when a particular routine is called
- when some user-defined condition (logical expression) is encountered
- when the value of a specific variable changes

The format of the *stop* command is

> **stop** *event* [**if** *condition*]

or

> **stop if** *condition*

where *event* can be

at *source-line-number*	suspend execution at the specified line number
in *routine*	suspend execution at the start of the specified routine
variable	suspend execution when the value of the specified variable changes

and *condition* is a logical expression. A logical expression may be a relational expression, or a parenthesized relational or logical expression connected by the logical operators **and** or **or**. The logical operator **not** is not supported because it is unnecessary; the sense of a relational can be reversed. For example, valid logical expressions are

```
i < 10
(i == 5) and (j > k)
((i+3 > 99) or (j <= k*m)) and (k != n-1)
```

All events can be modified to occur only when certain conditions are true by using the **if** *condition* option of the command. If you do not specify an *event*, program execution stops whenever *condition* is true.

When you set a breakpoint at a specific line number, *csd* stops program execution at the beginning of the line. When a breakpoint is set at a routine call, *csd* stops program execution at the beginning of the routine.

The following screen illustrates the use of the *stop* command to set a breakpoint at a specific line. The *list* command displays the area of code in which to set the breakpoint. You may only set breakpoints on lines that have an asterisk after the line number. The first *stop* command tries to set a breakpoint at line 331, which is an invalid line. The second *stop* command sets a breakpoint at line 336. The *status* command verifies that the breakpoints are set.

```
(csd) list 330,340
 330
 331   Display_next_frame ()
 332*  {
 333*      Clear_screen ();
 334*      last_frame = next_frame;
 335*      i = 1;
 336*      while (next_frame != frame_map[i] .frame_number)
 337       {
 338*          ++i;
 339*      }
 340*      fseek (in_file, frame_map[i] .frame_location, 0);
(csd) stop at 331
no executable code at line 331
(csd) stop at 336
[7] stop at 336
(csd) status
[7] stop at 336
(csd)
```

Note that there are minor differences between *cc* and *vc* (the two CONVEX C compilers); executable source lines are defined differently. For example, you can set a breakpoint at a label using *vc*, but cannot stop at the statement associated with that label. The reverse is true for *cc*.

To set a breakpoint at a line in a file other than the current file, precede the *source_line_number* with the name of the file in quotes and a colon, as in "*driver.f*":25.

You can also set a breakpoint at the entrance to a routine, as shown in the next screen.

```
(csd) stop in Get_response
[8] stop in Get_response
(csd) stop in Clear_screen
[9] stop in Clear_screen
(csd) status
[7] stop at 336
[8] stop in Get_response
[9] stop in Clear_screen
(csd)
```

csd also lets you set a breakpoint to occur when the value of some variable changes. The next example sets a breakpoint to occur when the value of *lesson_name* changes. The *status* command verifies that the breakpoint is set.

```
(csd) stop lesson_name
[10] stop lesson_name
(csd) status
[7] stop at 336
[8] stop in Get_response
[9] stop in Clear_screen
[10] stop lesson_name
(csd)
```

By specifying a logical condition in the *stop* command, you can be more exact when setting a breakpoint. For example, in the following screen a breakpoint is set in the *Get_response* routine, but only if the value of *answer[0]* is equal to "a".

```
(csd) stop in Get_response if answer[0] == 'a'
(csd) status
[20] stop if answer[0] = 'a' in Get_response
stopped in Get_response at line 249
  249*          if ((answer[0] == 'a') || (answer[0] == 'A'))
(csd)
```

Table 2-3 contains sample breakpoint specifications for a FORTRAN program.

Table 2-3: Sample Breakpoint Specifications

Specification	Result
stop at 35	Stops at line 35 in current source file.
stop at "sample.f":35	Stops at line 35 in source file *sample.f*.
stop at 35 if loop_count == 30	Stops at line 35 in current source file if the value of *loop_count* is equal to 30.
stop in f1	Stops execution when routine *f1* is called.
stop in MAIN_ if loop_count > 30	Stops in the main program when the value of *loop_count* is greater than 30.
stop xyz	Stops execution when the value of *xyz* changes.
stop if sample.MAIN_.i <= 10	Stops execution when the value of *i* <= 10, where *i* is defined in the main program in file *sample.f*.

Use the *stopi* command to set a breakpoint at a machine-instruction address, as follows:

stopi at *address* [**if** *condition*]

where *address* is the hexadecimal address of a machine instruction where you want the program to stop and **if** *condition* is the condition under which the program stops. To stop at a machine instruction when a variable changes its value, type

stopi *variable* [**if** *condition*]

The next screen demonstrates the two formats of the *stopi* command. The *print* command is executed to determine the hexadecimal address of *main+0xa*. The breakpoints are then set and the *status* command verifies that they are set.

```
(csd) print &main+0xa
0x80001a7a
(csd) stopi at 0x80001a7a
[10] stopi at 0x80001a7a
(csd) stopi testch
(csd) status
[1] stopi at 0x80001a7a
[2] stopi $b0.testch
(csd)
```

2.8.3 Setting a Tracepoint: *trace, tracei*

Tracepoints, like breakpoints, are used to instruct *csd* to print information that can be used to monitor the progress of the program. Unlike breakpoints, however, tracepoints do not return control to you when this information is displayed. Program execution continues until the program terminates normally or until it encounters a breakpoint, or fatal error, or is stopped.

To set tracepoints, use the command

trace [*event*] [**if** *condition*]

where *event* can be

in *routine* — display each source line only during execution of *routine*

source_line_number — display the specified source line immediately before it is executed

routine — display lines of information each time *routine* is called, including the name of the calling routine, the parameters passed, and the value returned if it is a function

expression **at** *source_line_number* — display the value of *expression* each time the specified source line is reached

variable [**in** *routine*] — display the name and value of the specified variable each time its value changes (while executing in *routine*, if specified)

If you do not specify any arguments, each source line is displayed before processing. This results in very slow program execution, because the program is stopped after every line and the conditions for all tracepoints are checked. The *condition* is evaluated before printing the tracing information. If the *condition* is false, *csd* does not display the tracing information.

The following screen shows how *csd* traces execution in the routine *Get_response*. The *trace* command sets the eventpoint in the routine. When the program runs, each line of the routine is displayed as it is executed.

```
(csd) trace in Get_response
[1] trace  in Get_response
(csd) run
trace:    243*     valid_answer = FALSE;
trace:    244*     while (valid_answer == FALSE)
trace:    246*         valid_answer = TRUE;
trace:    247*         printf (teststring);
trace:    248*         gets (answer);
trace:    249*         if ((answer[0] == 'a') || (answer[0] == 'A'))
trace:    254*         if ((answer[0] == 'b') || (answer[0] == 'B'))
trace:    259*         if ((answer[0] == 'c') || (answer[0] == 'C'))
trace:    264*         if ((answer[0] == 'd') || (answer[0] == 'D'))
trace:    269*         if ((answer[0] == 'q') || (answer[0] == 'Q'))
trace:    274*             valid_answer = FALSE;
trace:    275*     }
trace:    244*     while (valid_answer == FALSE)
trace:    246*         valid_answer = TRUE;
trace:    247*         printf (teststring);
trace:    248*         gets (answer);
```

Notice that some of the lines of code are missing (250-253, for example). The missing lines represent code in the source file that is not executed.

You can also display the value of an expression while tracing program execution. The following screen shows how to display the value of i*2 each time the variable is incremented.

```
(csd) trace (i*2) at 24
[2] trace i*2 at 24
(csd) run
at line 24: i*2 = 0
at line 24: i*2 = 2
at line 24: i*2 = 4
at line 24: i*2 = 6
at line 24: i*2 = 8
at line 24: i*2 = 10
at line 24: i*2 = 12
at line 24: i*2 = 14
at line 24: i*2 = 16
at line 24: i*2 = 18
at line 24: i*2 = 20
at line 24: i*2 = 22
at line 24: i*2 = 24
at line 24: i*2 = 26
at line 24: i*2 = 28
at line 24: i*2 = 30
at line 24: i*2 = 32
(csd)
```

You can also set conditional tracepoints using the **if** *condition* portion of the command. The next screen displays the same information as the previous screen, except that it only displays the values when *i* is between 10 and 20.

```
(csd) trace (i*2) at 24 if (i>10) and (i<20)
[3] if i > 10 and i < 20 { trace i*2 } at 24
(csd) run
at line 24: i*2 = 22
at line 24: i*2 = 24
at line 24: i*2 = 26
at line 24: i*2 = 28
at line 24: i*2 = 30
at line 24: i*2 = 32
at line 24: i*2 = 34
at line 24: i*2 = 36
at line 24: i*2 = 38
(csd)
```

Table 2-4 contains sample tracepoint specifications for a C program.

Table 2-4: Sample Tracepoint Specifications

Specification	Result
trace	Traces each line as it executes.
trace in s1	Traces each line of routine *s1* as it executes.
trace if xyz==10	Traces each line if *xyz=10.*
trace 100	Traces execution of line 100 in the current source file.
trace f2	Traces calls to routine *f2.*
trace xyz in main	Traces changes to variable *xyz* while the main program is executing.

Use the *tracei* command to turn on tracing at a machine-instruction address. To trace the execution of a machine instruction, type

tracei [*address*] [**if** *condition*]

To trace a variable before the execution of a machine instruction, type

tracei [*variable*] [**at** *address*] [**if** *condition*]

If no arguments are specified, all executed machine instructions are traced.

2.8.4 Executing Commands Conditionally: *when*

The *when* command executes a list of commands when a particular event occurs. The form of the *when* command is

when *event* { *command* ; ... }

where *event* can be

at *source_line_number*	execute the commands immediately before the specified *source_line_number* is executed
in *routine*	execute the commands when the program is executing instructions in *routine*
condition	execute the commands when a logical expression is true

Separate multiple commands with semicolons; a single command must end with a semicolon (;).

The following screen illustrates the *when* command by setting an eventpoint at line 371. When the program reaches line 371, *csd* displays the values of the *branch* array and continues program execution.

```
(csd) when at 371 {print branch;}
[13] print branch at 371
(csd) run
(0, 100, 500, 1000, 1500)
```

Program execution continues after the last *command* is executed unless it is **STOP** that stops execution and returns control to *csd*.

The next example shows how to use the keyword **STOP** to return control to *csd* after the commands are executed. The same code is used as in the previous example, and the *when* command is modified to include a **STOP**. Notice that after the array values are displayed, the *csd* prompt appears to indicate that *csd* is waiting for command input.

```
(csd) when at 371 {print branch; STOP;}
[14] { print branch; stop } at 371
(csd) run
(0, 100, 500, 1000, 1500)
[14] stopped in main at line 371
  371*                  Determine_branch ();
(csd)
```

2.8.5 Deleting Eventpoints: *delete*

Use the *delete* command to delete eventpoints, i.e., breakpoints, tracepoints and *when* commands. The command has the following format:

> **delete** *index* ...
> **delete all**

You can specify a list of events to delete or you can delete all the eventpoints currently set.

To determine the unique *index* identifying the currently active eventpoints, use the *status* command.

The following screen demonstrates the *delete* command. First, a *status* command is issued to determine currently set eventpoints. The *delete* command deletes the first two eventpoints. The last *status* command verifies that the two eventpoints are deleted.

```
(csd) status
[1] stop in main
[2] stop at 241
[3] trace  in Get_lesson
[4] print $b0.testch at 300
(csd) delete 1 2
(csd) status
[3] trace  in Get_lesson
[4] print $b0.testch at 300
(csd)
```

By using the **all** keyword, you can delete all currently set eventpoints, as illustrated in the next screen.

```
(csd) status
[1] stop in main
[2] stop at 241
[3] trace  in Get_lesson
[4] print $b0.testch at 300
(csd) delete all
(csd) status
(csd)
```

2.9 Manipulating the Call Stack

Use the following commands to manipulate the call stack.

where	display a stack trace of active routines
down	move down the call stack
up	move up the call stack
return	return from a routine on the call stack

2.9.1 Displaying the Call Stack: *where*

The call stack can be examined using the *where* command. To use this command, type

where [*integer*] [>*filename*]

The *where* command displays a stack trace of the routine calls that led to the current program state. The output for each call consists of the name of the routine, its arguments, the source line number containing the call, and the source file containing the routine.

The following screen demonstrates the *where* command. The listing shows the program executing in the *Determine_branch* routine, which is called from *main*.

```
(csd) where
Determine_branch, line 287 in "cbt.c"
main(0x1, 0xffffcdac, 0xffffcdb4), line 371 in "cbt.c"
(csd)
```

Use the optional *integer* argument to display a specific number of runtime stack frames or routines. Thus, the command *where 1* displays the current address, current function, and current source file. You can also direct output from this command into *filename* with the ">" symbol.

The next example lists the top two frames on the call stack.

```
(csd) where 2
Clear_screen, line 79 in "cbt.c"
Get_lesson, line 162 in "cbt.c"
(csd)
```

For more information about runtime stack frames, refer to the *CONVEX Architecture Reference.*

2.9.2 Moving Through the Call Stack: *down, up*

For every instance of a routine, a stack frame is pushed on the runtime stack and is popped when the routine returns. When routines recursively call one another, both routines are on the runtime stack at the same time. To move the current routine up (toward "main") or down the stack to resolve names, use the *up* or *down* commands. The formats for these commands are

up [*count*]
down [*count*]

where *count* is the number of levels to move on the runtime stack. If *count* is not specified, the default *count* is 1.

The following screen illustrates the effects of using *up* and *down* with a program that has multiple declarations of a variable name. The sample screen is based on the C program, *environ.c*, shown in Figure 2-7.

Figure 2–7: C Program Using Nested Calls

```
fraction (x,y)

int x, y;
{
  int z;
  z = x / y;
  printf ("The value of %d / %d is %d0,x,y,z);
}

swap (x,y)

int x, y;
{
  int temp;

  fraction (x,y);
  temp = x;
  x = y;
  y = temp;
  fraction (x,y);
}

main ()
{
  int x,y;

  for (x=1; x>=0; x--)
   {
    for (y=0; y<=2; y++)
      swap (y,x);
   }
}
```

Notice that there are three declarations of the variable *x*. When the program suspends execution, the call stack is displayed by the *where* command. The *up* and *down* commands move through the call stack. At each stage, the *which* command displays the particular instance of *x* currently being used.

```
(csd) where
fraction(x = 0, y = 1), line 7 in "environ.c"
swap(x = 0, y = 1), line 18 in "environ.c"
main(0x1, 0xffffcda8, 0xffffcdb0), line 32 in "environ.c"
(csd) which x
environ.fraction.x
(csd) up 2
main(0x1, 0xffffcda8, 0xffffcdb0), line 32 in "environ.c"
(csd) which x
environ.main.x
(csd) down
swap(x = 0, y = 1), line 18 in "environ.c"
(csd) which x
environ.swap.x
(csd)
```

Notice that as you move up or down the call stack, the environment of the program changes, as reflected by the changing instances of *x*. The current point of execution does not change when you move up or down the call stack.

2.9.3 Return From Routines on the Call Stack: *return*

The *return* command pops one or more runtime stack frames from the top of the runtime stack.

return [*routine*]

The program begins executing at the current program counter and stops at the next statement after it returns. If you specify an optional routine name, runtime stack frames are executed until the named *routine* is on the top of the runtime stack.

The *return* command is illustrated in the following screen. The program stopped in the *fraction* routine, as shown by the *where* command listing. The *return* command executed the remaining code in *fraction* and popped off the call stack, stopping at the next instruction in *swap*.

```
(csd) run
[1] stopped in fraction at line 7
    7*      z = x / y;
(csd) where
fraction(x = 0, y = 1), line 7 in "environ.c"
swap(x = 0, y = 1), line 18 in "environ.c"
main(0x1, 0xffffcda8, 0xffffcdb0), line 32 in "environ.c"
(csd) return
The value of 0 / 1 is 0
stopped in swap at line 19
   19*      temp = x;
(csd)
```

2.10 Customizing *csd* Behavior

Use the following commands to change the way *csd* works:

alias	define an alias for a *csd* command
unalias	cancel an alias definition
mode	display or change the current execution mode
fpmode	display or change the current floating-point mode
format	display or change the format of integers
use	display or change the directory search path

2.10.1 Using Command Aliases: *alias, unalias*

The *csd* command *alias* enables you to abbreviate or rename a *csd* command or a command *string*. You may also specify parameters within the command *string* for which *csd* substitutes values. The various forms of the *alias* command are

alias
alias *newname*
alias *newname oldname*
alias *newname* "*string*"
alias *newname* (*parameters*) "*string*"

Using the *alias* command with no arguments displays a list of active user defined aliases. If just *newname* is specified, its alias is printed. You can refer to a command by either the *oldname* or the *newname.*

The following example illustrates the *string* form of the *alias* command. To shorten the *stop* command and specify a parameter, type

alias s(n) "stop at n"

Then when you wish to stop at a particular line, all you need to enter is the shortened command and the line at which you wish to stop:

s(122)

csd expands **s(122)** to "stop at 122."

csd looks for lists of alias commands in a file in the current directory called *.csdinit.* Aliases in this file are defined each time you invoke *csd,* so they can be used from session to session. You can also keep the *.csdinit* file in your home directory, because *csd* looks there after searching the current directory, if the file is not found.

To remove an alias, use the *unalias* command in the following format

unalias *name*

where *name* is the alias for the command that you wish to delete.

The *csd* command set comes with a built-in set of aliases that you can use in place of the command name. These standard command aliases are shown in Table 2-5. You can also add to this list, and develop aliases for more complex functions.

Table 2-5: *csd* Standard Command Aliases

Alias	Command
a	assign
b	stop
c	cont
d	delete
e	edit
f	func
h	help
l	list
n	next
p	print
q	quit
r	run
s	step
t	trace
w	where
st	status
wi	whereis
W	which

2.10.2 Changing the Current Execution Mode: *mode*

The *mode* command sets the execution characteristics of your program to either sequential mode or chained mode. Sequential and chained are two methods of debugging: sequential mode is the normal method for debugging software; chained mode is normally used when a hardware bug is suspected. Only use chained mode as a last resort. The format of this command is

mode [sequential | chained]

Entering the *mode* command with no arguments displays the mode currently in effect. The default mode is sequential.

2.10.3 Changing the Current Floating-Point Mode: *fpmode*

csd supports two floating-point modes: native, which is CONVEX specific, and IEEE. The *fpmode* command sets and displays the floating-point mode.

The floating-point mode allows *csd* to translate internal bit representations of floating-point numbers (single and double precision) to the correct external value. When you assign floating-point values, the mode allows *csd* to translate your value into the correct internal bit representation. For instance, if the floating-point numbers of a program are in IEEE format, set the mode to *ieee* so that *csd* correctly interprets the floating-point values.

To display the floating-point mode for the debugger and the program you are debugging, type

fpmode

The output looks something like the following

```
  csd: native
a.out: native
```

If the program is dual mode, *csd* begins in *auto* mode. You cannot change the floating-point mode of the program you are debugging (only the program can), but you can change the floating-point mode of *csd*. To change the mode of *csd*, type

fpmode *mode*

where *mode* is *native* (for the native format), *ieee* (for IEEE format), or *auto*. The *auto* mode automatically synchronizes the mode of *csd* with the mode of the program being debugged. This ensures that when *csd* prints or assigns a floating-point value, it is interpreted according to the mode used by the program at that stage of its execution. At the start of a *csd* debugging session, the floating-point mode of the debugger is the same as that of the program you are debugging.

For example, suppose the hexadecimal representation of a double-precision floating-point value is 0x3ff0000000000000. If you print the value after typing

fpmode ieee

you get "1.0". If you print the value after typing

fpmode native

you get "0.25".

2.10.4 Changing the Current Integer Format: *format*

The *format* command sets the default print format for displaying integer values to either hexadecimal or decimal. The default format is decimal. To set the print format, type:

format [**hex** | **decimal**]

Typing the *format* command with no argument displays the print format currently in effect.

In the following screen, the *format* command displays the current integer format as decimal. The next command changes the integer format to hexadecimal. The *print* command illustrates that the format has changed (36 decimal is equal to 24 hexadecimal).

```
(csd) format
decimal
(csd) format hex
(csd) print 6*6
0x24
(csd)
```

2.10.5 Changing the Floating-Point Precision

csd displays floating-point numbers with a default precision of 6. To modify this value, type

set precision =*integer*

This value is used for all subsequent floating-point displays until you enter the next *set* command. Setting the precision to zero (0) restores the default value.

2.10.6 Changing the Directory Search Path: *use*

The *use* command manipulates the list of directories through which *csd* searches for source files and functions similarly to the *-I* option. The difference between *use* and the *-I* option is that the latter may be used only when *csd* is invoked, while *use* may be invoked as needed during the debugging session. To use this command, type

use [*directory* ...]

The previous list is erased, not appended, when a new list is supplied. Also, directories are searched in the order in which they appear.

If no argument is given, the list of directories is displayed.

The next screen demonstrates the *use* command. First, the current directory search path is displayed. Next, the search path is changed; and finally, the directory search path is displayed to verify the change.

```
(csd) use
.
(csd) use . /usr/smith /usr/jones/project
(csd) use
. /usr/smith /usr/jones/project
(csd)
```

2.11 Executing a Shell Command: *sh*

The *sh* command passes a command line to the shell for execution. (The shell used depends on the argument used with the SHELL variable in the *.login* file.) To use this command, type

sh *command-line*

where *command-line* is the command to be passed to the shell.

For example, you can list the files starting with "c" in the current directory and display the contents of the *commands* file, as illustrated in the next screen.

```
(csd) sh ls c*
  50 cai*        10 cbt.c       144 core        36 core.e*
  10 cai.c        2 commands      2 core.c
(csd) sh cat commands
stop in main
run
next 3
list
(csd)
```

2.12 Editing a Source File: *edit*

Use the *edit* command to invoke text-editing systems. The text-editing system invoked by default depends on your local environment. You can override the default by setting the EDITOR variable in the *.login* file to the name of the desired editor.

Forms of the *edit* command are

edit [*filename*]
edit *routine*

If you do not specify a *filename*, the editor is invoked on the current source file. If you specify a *routine*, the editor is invoked on the file that contains it.

Chapter 3
Debugging Optimized Code

This chapter explains how to debug optimized and vectorized code produced by the CONVEX FORTRAN and C compilers. It explains how optimization affects the mapping between source and assembly-language code and how it affects the way you debug a program at the source level. Several sample debugging sessions are included to illustrate the differences in debugging optimized code.

Topics covered in this chapter include

- effects of optimization
- problems with debugging optimized code
- tracking a bug in optimized code

csd provides limited support for debugging optimized and vectorized code produced by the CONVEX FORTRAN and CONVEX C compilers using the *-O0*, *-O1*, or *-O2* options. See Chapter 4, "Debugging Multithreaded Programs," for a discussion on debugging parallelized programs, i.e., programs compiled at the *-O3* level.

3.1 Effects of Optimization

Optimized code presents two major debugging problems: it drastically affects the correspondence between source statements and object code, and it makes it difficult to obtain the value of certain variables. With respect to affecting the map between source statements and object code locations, program optimization can:

- move code - as in hoisting code (as with code motion), sinking code (as with the incrementation performed at the end of a loop), and initialization of a variable globally allocated to a register. There is still a one-to-one mapping between source and object, but execution of the object code occurs at a different point than the source code indicates.

- merge identical code sequences - as in common subexpression elimination. There is now a many-to-one source-to-object mapping.

- replicate statements - as in inline substitution, loop unrolling, dynamic selection, loop distribution, and including executable statments in an include file. There is now a one-to-many source-to-object mapping.

- combine statements - as in vectorization. Statements may be collapsed into more powerful machine instructions (array operations or a sum reduction in a loop may be performed by a vector instruction, for example). Many statements within a group of statements now map to the same object code.

- delete statements - as in a redundant assignment or dead code. There is no object code for the source statement.

- rearrange code - as in instruction scheduling, which the compiler always performs at -O0 and higher, causes instructions for a statement to no longer be consecutive, but intermixed with instructions of other statements. Therefore, statments do not start in source order.

With respect to being able to obtain the value of a variable, program optimization can:

- eliminate variables - as in the iteration variable of a loop may be eliminated and expressed in terms of compiler introduced variables.

- change when a value is assigned - as in instruction scheduling can cause the assignment to a variable to be out of source order and therefore the value in memory not to be current at certain points in the computation.

- allocate variable to a register - as in global register allocation. Over a section of source code, the value will not be in memory.

- delete an assignment - as in a useless assignment after constant propagation and constant folding are performed.

The debugger is unaware of the optimizations that have been performed. Therefore, it cannot give the correct value of certain variables, nor be precise about where program execution stopped. However, there are specific statements at which you can set an eventpoint and be free of the effects of instruction scheduling. These statements are located at the start of a block of code in

which each statement is executed and after it has executed the flow of control changes. Such a block is called a *basic block*.

The first statement in a basic block represents an eventpoint; eventpoints are symbolized by an asterisk after their line number when listed by the *list* command. A breakpoint or tracepoint can only be set on an eventpoint line and an attempt to set one on a non-eventpoint line results in an error message. Within optimized code, therefore, the *step* and *trace* commands work on a basic block basis, not a source line basis.

Program execution may be suspended within a basic block when a fault occurs. The point where program execution stopped is called the *locus of execution.* (In terms of machine instructions, the locus of execution is the current value of the program counter. In terms of source statements, the locus of execution is within the statement corresponding to the last machine instruction executed.). When execution is suspended, the debugger reports the eventpoint line corresponding to the basic block containing the locus of execution.

To determine the exact source line in *routine* corresponding to the last instruction executed

1. Compile the routine with the same compiler option(s) used to create the compiler options.

2. Assemble the resulting *.s* file with the *-l* option of the *as* command and redirect *stdout* to a file.

3. Determine the relative address of the current routine from the output file of the *as* command. The compute the relative address of the locus of execution with the following command:

   print &pc – *&routine* + *offset*

4. Find the instruction corresponding to the resulting relative address. This is the next instruction to be executed. The immediately preceding instruction is the last instruction executed.

5. Examine the last instruction executed. The number after the semi-colon with a # prefix is the source line from which the machine instruction was generated. The source line reported for the start of the containing basic block should be the line number in the first preceding *.stabd* (last item).

3.2 Debugging an Optimized Routine

To help you better understand how optimization and vectorization affects the task of debugging a program, this section

- presents and explains an optimized FORTRAN subroutine
- illustrates how the optimization affects the way you debug the routine
- takes you step-by-step through the process of locating a bug (intentionally introduced) in the routine.

3.2.1 About the Routine

The example program is the *tred1* subroutine used in EISPACK. *tred1* reduces a real symmetric matrix to a symmetric tridiagonal matrix using orthogonal similarity transformations. For this example, *tred1* is compiled with the *-O2* and *-db* options of the *fc* command. The source for *tred1* is listed in Figure 3-1. The listing is generated from within *csd* to show line numbers and the beginnings of basic blocks (identified by asterisks).

The *fc* compiler vectorized all program loops except those beginning at lines 12, 50, and 76. The number of basic blocks is fewer than if the routine had been compiled with the *-no* and *-db* options.

Consider the basic block beginning at line 12. It initializes the induction variable and tests for loop termination. The iteration variable *ii* is replaced with a compiler-generated induction variable. The basic block beginning at line 13 consists of statements 13-17. The variables *i* and *l* are induction variables had are replaced with a compiler-generated induction variable. Therefore, you cannot monitor the values of *i*, *ii*, and *l* if execution is suspended within the 300 loop (lines 12-92).

The compiler allocated the variables *scale* and *h* to a scalar register when the program is executing between lines 15 and 39. Therefore. you cannot examine their values if the program stops between these lines.

The basic block beginning at line 39 consists of statements 39-45. The compiler used instruction scheduling scrambling for these lines of code. For example, *dsqrt* and *dsign* are computed first, the assignment to *e2* in line 39 is performed last, just before the test in line 45, and the variable *g* is held in a scalar register over the entire basic block.

Figure 3–1: Source of Optimized Subroutine, *tred1*

```
1*      subroutine tred1(nm,n,a,d,e,e2)
2  c
3       integer i,j,k,l,n,ii,nm,jp1
4       double precision a(nm,n),d(n),e(n),e2(n)
5       double precision f,g,h,scale
6  c
7*      do 100 i = 1, n
8*        d(i) = a(n,i)
9         a(n,i) = a(i,i)
10   100 continue
11  c    .......... for i=n step -1 until 1 do -- ..........
12*      do 300 ii = 1, n
13*        i = n + 1 - ii
14         l = i - 1
15         h = 0.0d0
16         scale = 0.0d0
17         if (l .lt. 1) go to 130
18  c    .......... scale row (algol tol then not needed) ..........
19*        do 120 k = 1, l
20*   120   scale = scale + dabs(d(k))
21  c
22*        if (scale .ne. 0.0d0) go to 140
23  c
24*        do 125 j = 1, l
25*          d(j) = a(l,j)
26           a(l,j) = a(i,j)
27           a(i,j) = 0.0d0
28   125   continue
29  c
30*   130   e(i) = 0.0d0
31         e2(i) = 0.0d0
32         go to 300
33  c
34*   140   do 150 k = 1, l
35*          d(k) = d(k) / scale
36           h = h + d(k) * d(k)
37   150   continue
38  c
39*        e2(i) = scale * scale * h
40         f = d(l)
41         g = -dsign(dsqrt(h),f)
42         e(i) = scale * g
43         h = h - f * g
44         d(l) = f - g
45         if (l .eq. 1) go to 285
46  c    .......... form a*u ..........
47*        do 170 j = 1, l
48*   170   e(j) = 0.0d0
49  c
50*        do 240 j = 1, l
51*          f = d(j)
52           g = e(j) + a(j,j) * f
53           jp1 = j + 1
54           if (l .lt. jp1) go to 220
55  c
56*          do 200 k = jp1, l
57*            g = g + a(k,j) * d(k)
58             e(k) = e(k) + a(k,j) * f
59   200      continue
60  c
61*   220     e(j) = g
62   240   continue
63  c    .......... form p ..........
64*        f = 0.0d0
65  c
66*        do 245 j = 1, l
67*          e(j) = e(j) / h
68           f = f + e(j) * d(j)
69   245   continue
```

```
70  c
71*        h = f / (h + h)
72  c    .......... form q ..........
73*        do 250 j = 1, 1
74*   250   e(j) = e(j) - h * d(j)
75  c    .......... form reduced a ..........
76*        do 280 j = 1, 1
77*          f = d(j)
78         g = e(j)
79  c
80*          do 260 k = j, 1
81*   260      a(k,j) = a(k,j) - f * e(k) - g * d(k)
82  c
83*   280    continue
84  c
85*   285    do 290 j = 1, 1
86*          f = d(j)
87         d(j) = a(1,j)
88         a(1,j) = a(i,j)
89         a(i,j) = f * scale
90   290   continue
91  c
92*   300 continue
93  c
94*      return
95      end
```

3.2.2 Problems with Debugging Optimized Code

This section illustrates the problems associated with debugging optimized code by proceeding step-by-step through the debugging process. The optimized routine, *tred1*, is used in this scenario. Recall that the source for *tred1* is shown in Figure 3-1.

In this scenario, you are trying to determine the values of *ii*, *i*, *l*, *h*, and *scale* on each iteration of the 300 loop (lines 12-92).

Step 1 Invoke *csd* and change the current context to the source file, *tred1.f*.

```
% csd a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...
(csd) file tred1.f
(csd)
```

Step 2 List the code for the relevant portion of the loop.

```
(csd) list 11, 18
   11   c     .......... for i=n step -1 until 1 do -- ..........
   12*        do 300 ii = 1, n
   13*           i = n + 1 - ii
   14            l = i - 1
   15            h = 0.0d0
   16            scale = 0.0d0
   17            if (l .lt. 1) go to 130
   18   c     .......... scale row (algol tol then not needed) ..........
(csd)
```

Step 3 Set a breakpoint at line 17 so you can check the values of the variables immediately after they are assigned in the loop. When you enter the command, *csd* displays an error message stating that there is no eventpoint at line 17. There is no asterisk at line 17 because it is inside the basic block beginning at line 13, and eventpoints can only be set at the beginning of basic blocks.

```
(csd) stop at 17
event point not permitted at line "tred1.f":17
(csd)
```

Step 4 Set a breakpoint at line 13 to check the value of *ii* before the assignment to *i* is made. Then run the program and examine the value of *ii* when the program stops. Displaying the value of *ii* at this point shows an incorrect value, because *ii* was transformed to a new induction variable when the routine was optimized.

```
(csd) stop at 13
[1] stop at "tred1.f":13
(csd) run
[1] stopped in tred1.tred1 at line 13 in file "tred1.f"
   13*           i = n + 1 - ii
(csd) print ii
0
(csd)
```

Step 5 Step one basic block, which advances to line 19 (not to line 14 as you might expect) because line 19 is the start of the next basic block and the next line at which an event can occur (signified by the asterisk). Displaying *i* and *l* shows incorrect results because the variables were also transformed into new induction variables.

```
(csd) step
stopped in tred1.tred1 at line 19 in file "tred1.f"
   19*           do 120 k = 1, l
(csd) print i,l
0 0
(csd)
```

Step 6 Set a breakpoint at the end of the loop (line 92). Continue execution and display the values of *h* and *scale* when execution is suspended. Note that the values are correct.

```
(csd) stop at 92
[4] stop at "tred1.f":92
(csd) cont
[4] stopped in tred1.tred1 at line 92 in file "tred1.f"
   92*    300 continue
(csd) print h,scale
0.00262695 251.562
(csd)
```

Step 7 Continue execution and step one line when execution is suspended at line 13. At this point you would expect the values of *h* and *scale* to be zero, but they are, instead, the values in memory at the end of the loop. The reason is that these variables are allocated to scalar registers within the loop. When needed, the values for the variables are taken from the registers. When the end of the loop is reached, the values will have been written into program memory.

```
(csd) cont
[1] stopped in tred1.tred1 at line 13 in file "tred1.f"
   13*           i = n + 1 - ii
(csd) step
stopped in tred1.tred1 at line 19 in file "tred1.f"
   19*           do 120 k = 1, l
(csd) print h,scale
0.00262695 251.562
(csd) q
%
```

3.2.3 Tracking a Bug in Optimized Code

This section illustrates a technique for tracking down a bug in optimized code. The same optimized routine, *tred1,* is used (see Figure 3-1). The error occurs in the following section of code.

```
63  c    .......... form p ..........
64        h = 0.0d0
65  c
66        do 245 j = 1, l
67          e(j) = e(j) / h
68          f = f + e(j) * d(j)
69   245    continue
```

For illustration purposes, the assignment to *f* at line 64 is mistyped as an assignment to *h*. You are trying to find the actual instruction at which the program aborts.

Step 1 Invoke *csd* and run the program. The program aborts because of floating-point exception at line 67, caused by an attempt to divide by zero.

```
% csd a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...
(csd) run

Floating point exception (floating divide by zero) in tred1.tred1
at line 67 in file "tred1.f"
   67*              e(j) = e(j) / h
(csd)
```

Step 2 Display the value of *h*.

```
(csd) print h
0.00262695
(csd)
```

Step 3 List the source code surrounding line 67. Note that *h* is assigned the value of zero on line 64; however, the line doesn't represent an eventpoint. The reason is that a constant propagation was performed on *h* by the compiler, which accounts for why the value of *h* displayed is not zero as expected, but the last value stored in memory.

```
(csd) list 63 73
   63   c      .......... form p ..........
   64              h = 0.0d0
   65   c
   66*             do 245 j = 1, l
   67*                e(j) = e(j) / h
   68                 f = f + e(j) * d(j)
   69      245     continue
   70   c
   71*             h = f / (h + h)
   72   c      .......... form q ..........
   73*             do 250 j = 1, l
(csd)
```

Step 4 Recompile the source code using the *-S* option to create the assembly code file, *tred1.s*. The redirection of *stderr* prevents the optimization information from being displayed on the screen. Then assemble *tred1.s* and store the output in *tred1.list* to get the relative locations for the instructions.

```
(csd) sh fc -O2 -db -S tred1.f >&optSummary
(csd) sh as -l tred1.s >tred1.list
(csd)
```

Step 5 Search the file *tred1.list* for the entry point of *tred1*, which is used in the calculation as the offset of the routine from *main*.

```
(csd) sh grep _tred1 tred1.list
000023  00000000                .stabs  "tred1:F21",0x24,0,0,_tred1_
000067  0000000c                .globl  _tred1_ ;ENTRY
000068                                  _tred1_:
000075  0000000c                .stabs  "tred1:F21",0x24,0,0,_tred1_
(csd)
```

Step 6 Determine the relative location of the pc (next instruction to be executed). Then search for the memory address in *tred1.list*. This line contains the next instruction to be executed.

```
(csd) print $pc-&tred1+0xc
0x580
(csd) sh grep 00000580 tred1.list
000561  00000580  3b0a0000     ld.l     0(a1),v2        ; #68, D
(csd)
```

Step 7 Display the ten lines surrounding line 561 using the *tail* and *head* commands. The error occurred two lines earlier at the *div.d* instruction.

```
(csd) sh tail +561 tredl.list | head
000555  0000056a  14c8ffffff80          add.w   #0xffffff80,s0  ; #69, -128
000556  00000570  5869                  add.w   a5,a1    ; #67
000557  00000572  14850400              add.w   #0x0000400,a5   ; #69
000558  00000576  3b28fc00              ld.l    -1024(a5),v0    ; #67, E
000559  0000057a  9e09                  div.d   v0,s1,v1        ; #67
000560  0000057c  3f29fc00              st.l    v1,-1024(a5)    ; #67, E
000561  00000580  3b0a0000              ld.l    0(a1),v2        ; #68, D
000562  00000584  9253                  mul.d   v1,v2,v3        ; #68
000563  00000586  7e8b                  sum.d   s3       ; #68, F
000564  00000588  1f880000              lt.w    #0x0000000,s0   ; #69, 0
(csd)
```

Note that in the above example, the line identified by *csd* as the line at which the error occurred corresponded with the same line number in the assembly code. The reason is that the assignment statement is located at the beginning of the basic block. Had lines 67 and 68 been switched, *csd* would still have reported the error as occurring at line 67, rather than line 68, because line 67 represents the beginning of the basic block containing the error. The assembly code would have shown that the error occurred in line 68.

Chapter 4
Debugging Multithreaded Programs

This chapter explains how to debug multithreaded code produced by the CONVEX FORTRAN and C compilers. It explains the concept of multithreaded code and introduces basic terminology used when discussing multithreaded code. The chapter also documents the commands that have been added to *csd* and the enhancements made to existing commands to enable you to debug multithreaded code.

Topics covered in this chapter include

- multithreaded programs
- multithreaded debugging commands
- sample debugging sessions

csd provides support for debugging parallelized code produced by the CONVEX FORTRAN and CONVEX C compilers using the *-O3* option. In particular, *csd* supports the debugging of both automatically parallelized loops and code specified to execute in parallel via directives when such code is implemented as multiple threads, i.e., multiple instruction streams that execute independently in parallel without explicit user synchronization. Since -O3 includes all optimizations performed at -O2, you must also contend with the problems associated with debugging optimized code (see Chapter 3, "Debugging Optimized Code").

A compiled program executes under *csd* as a single UNIX process; *csd* does not support the debugging of a multiprocess program. Therefore, multithread debugging refers to debugging a single process with multiple threads.

4.1 Multithreaded Programs

A *multithreaded program* is an application in which a piece of the program is sub-divided and solved by several, independent instruction streams executing in parallel. Each stream is called a *thread.* Threads can communicate with each other to synchronize and to exchange data. In the CONVEX multiprocessor system, threads are dynamically distributed between the CPUs; many threads can run on one CPU, or on many CPUs.

To help you better understand the difference between single-threaded and multithreaded programs, the next three sections introduce multithread terminology, the concept of multiple threads, and the use of registers in multiple threads.

4.1.1 Terminology

Some of the more common terms used to discuss multithreaded programs include

CPU — One physical processing unit. Each CPU in the configuration operates independently as a 64-bit CONVEX supercomputer.

multiprocessor — A CONVEX supercomputer consisting of the memory system, the I/O system, peripherals, and one or more CPUs.

complex — The entire set of physical CPUs in the system configuration.

sub-complex — Any non-empty, proper subset of a complex.

process — A collection of one or more streams of execution within a single logical address space.

thread — A stream of execution within a process. In the CONVEX multiprocessor environment, the total number of threads in a process can never exceed the number of CPUs within the multiprocessor configuration.

mutual exclusion — A protocol that guarantees that only one thread obtains access to a resource at a given time.

communication registers — A high-speed register set used for communication between the threads of a process. Threads communicate by sending and receiving data through the communication registers. A hardware-maintained *lock* bit is associated with each communication register; the lock bit guarantees mutually exclusive access to the register.

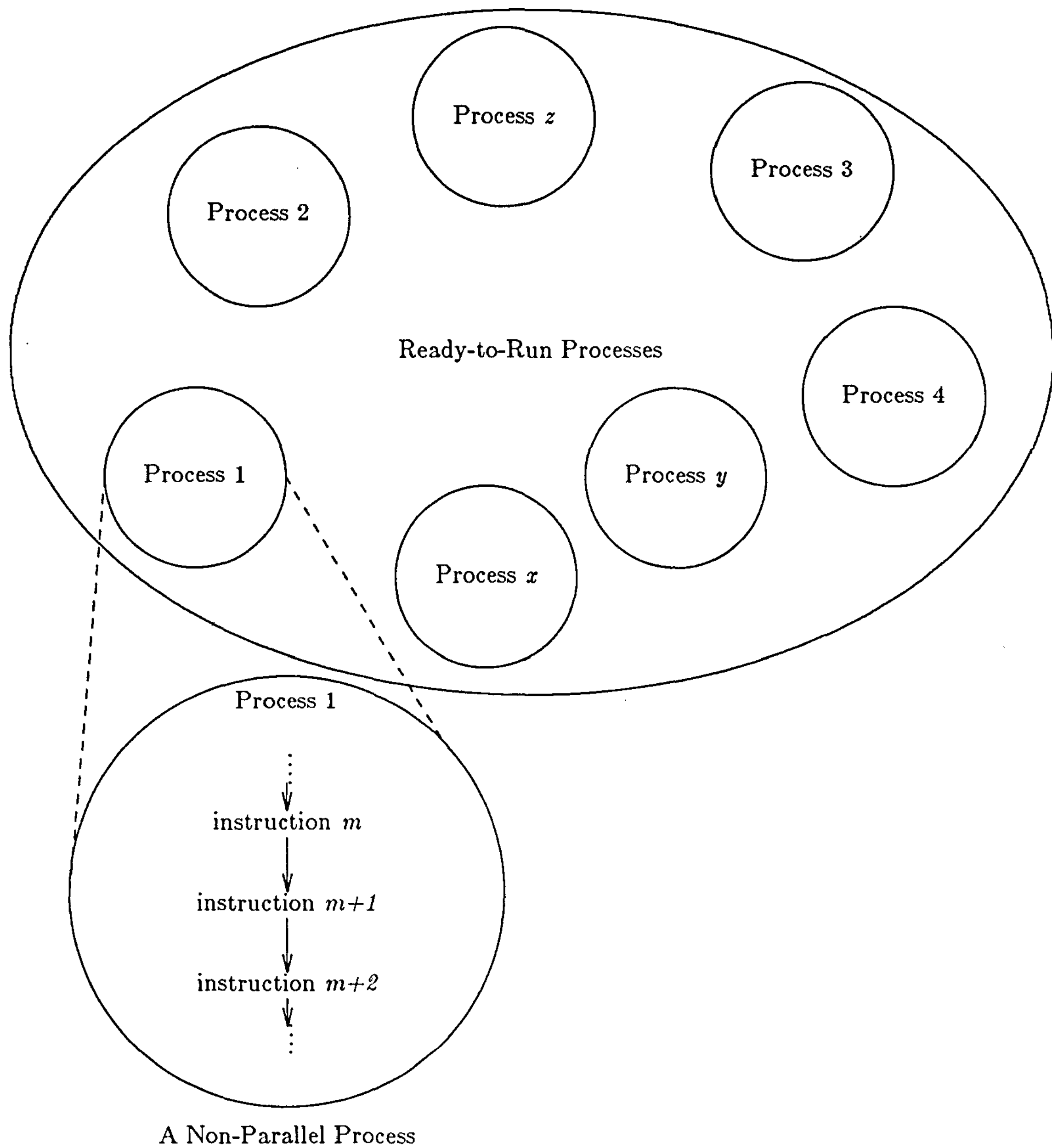
4.1.2 Multiple Threads

Multithreaded applications are radically different than standard, non-parallel programs. To debug multithreaded code effectively, you must understand how multithreaded programs execute in a multiprocessor environment.

In a single-processor system, a program executes in linear fashion according to the logical order of the source code statements. In a single-processor system, each process has a single stream of execution and, at any given time, the state of the program is defined by the set of program statements already executed.

Figure 4-1 shows an overview of a single-processor system. The single-processor system maintains a set of ready-to-run processes; a process is an instance of a running program. Each process has one stream of execution as shown in the detail of Process 1. The state of Process 1 at instruction m is defined by the set of instructions executed before instruction m.

Figure 4–1: Processes in a Single-Processor System



A Non-Parallel Process

A multiprocessor system supports multiple streams of execution. A program executes in parallel by dividing a problem among several independent threads. Each thread executes sequentially in an order defined by the source code and in parallel with other threads. The state of a parallel program is defined by the combined states and interactions of all the threads within a process. A parallel program can add new threads to the processing, and remove threads from the computation.

Figure 4-2 displays an overview of a multiprocessor system. As in single-processor systems, the multiprocessor system maintains a list of ready-to-run processes. In the multiprocessor system, however, each program may have one or more threads executing simultaneously. In the exploded view of Process *x*, there are two independent threads of execution:

- Thread 1 is executing the sequence of instructions *m, m+1, m+2 ...*
- Thread 2 is executing the sequence of instructions *n, n+1, n+2...*

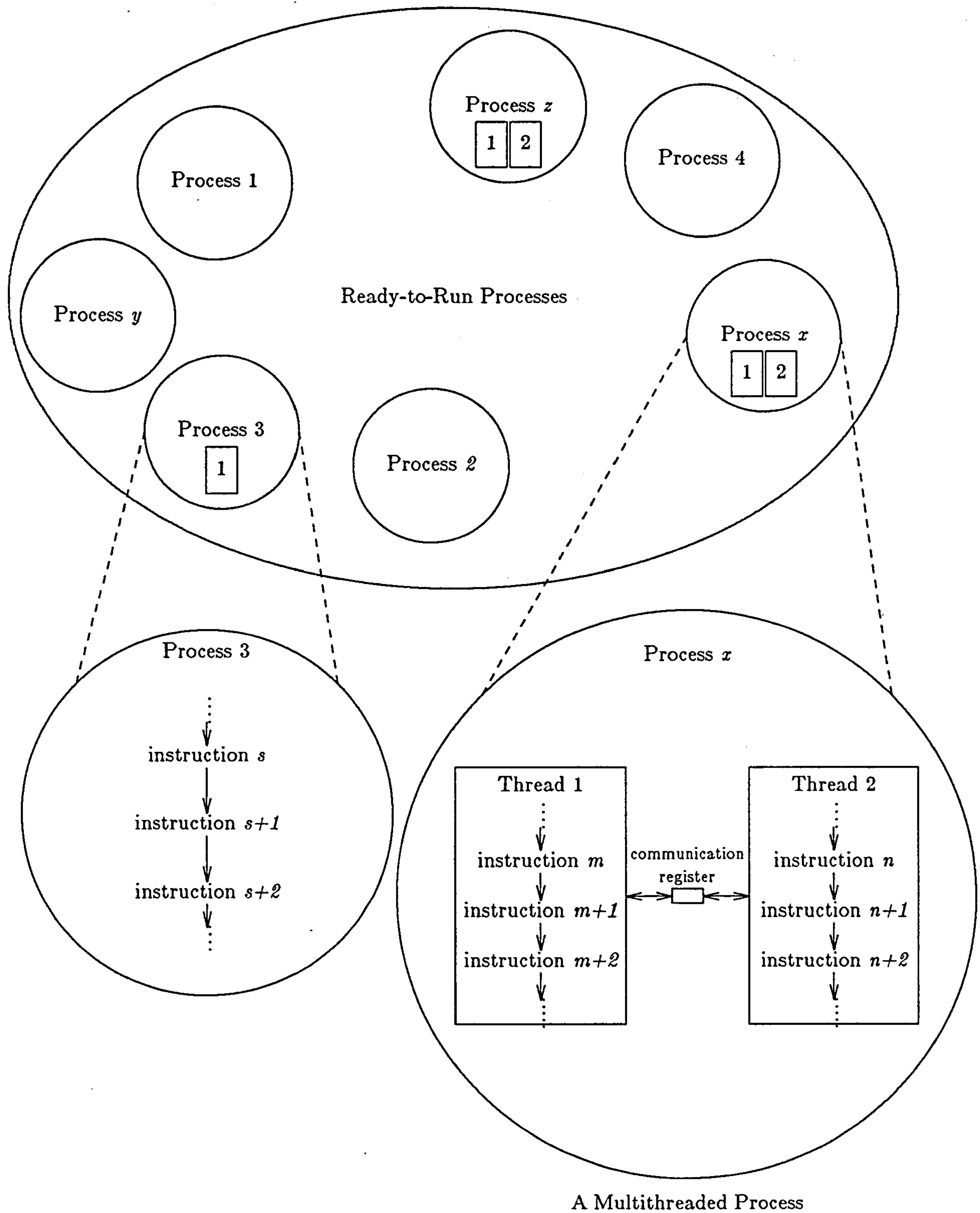
The state of Process *x* depends on the combined states of Thread 1 and Thread 2.

To share information, Thread 1 and Thread 2 use one or more *communication registers* (represented in Figure 4-2 by the small box). Communication registers are shared resources provided by the hardware; Thread 1 and Thread 2 use mutual exclusion primitives, also provided by the hardware, to control access to the communication register.

Figure 4-2 displays processes for other programs that might exist: Process 3 has a single thread; Process *z* has two threads.

At a given time, a program may have any number of threads executing in parallel, up to the maximum number of CPUs configured in the complex. To debug multithreaded programs, you must manipulate each thread independently and control the way the threads interact.

Figure 4–2: Processes in a Multiprocessor System



A Multithreaded Process

4.1.3 Registers

In a multithreaded process, each thread has its own complement of machine registers. The state of each thread is reflected in the thread's private register set. Each thread's register set includes

- program counter register (PC)
- status word register (PSW)
- address registers A0 through A7; each address register is 32 bits wide
- scalar registers S0 through S7; each scalar register is 64 bits wide
- vector registers V0 through V7; each vector register has 128 elements, and each element is 64 bits wide

Each thread executes independently of and in parallel with other threads in the same process. Threads must synchronize or exchange information among themselves. A new set of registers, *communication registers*, is the medium for inter-thread communication. Within a process there are 128, 64-bit communication registers; all threads within a process share the same communication register set.

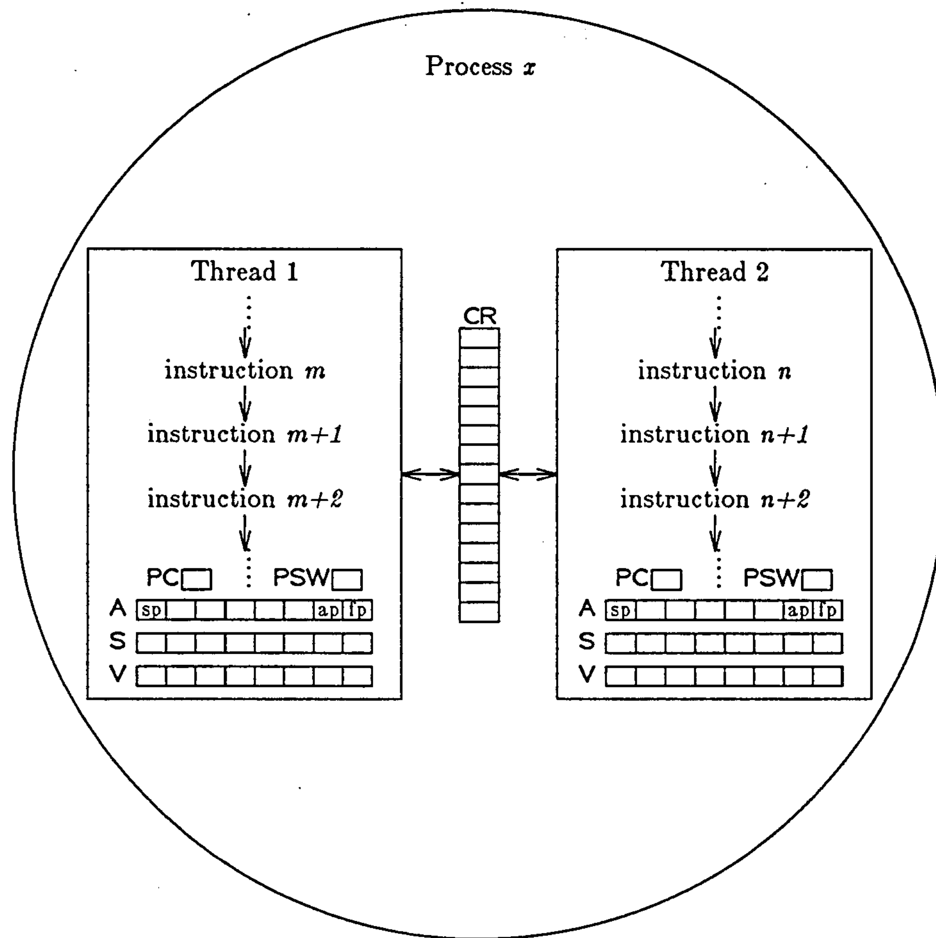
The 128 hardware communication registers are divided as follows:

- 32 registers reserved for use by the hardware
- 32 registers reserved for use by the operating system
- 64 registers accessible for use by user programs

Figure 4-3 shows the detail of Process *x* from Figure 4-2 with all registers drawn. The array of boxes labeled CR represents the process' user-accessible communication registers. Note that each thread has its own program counter, status word and sets of address, scalar, and vector registers, but the communication registers are shared among all the threads.

In addition to its private register set, a thread can also have private memory. Thread-private memory is allocated for each thread as the thread is created; thread-private memory is additional space for data, and can only be accessed by the owning thread. For different threads, thread-private memory is accessed using the same addresses; the virtual memory system translates thread-private logical addresses into unique thread-specific physical addresses.

Figure 4–3: A Multithreaded Process and Its Registers



4.2 Current Thread

When executing a multithreaded program under *csd*, one thread is identified as the *current thread*. Commands apply to the current thread. For example, the *where* command displays the stack frame only for the current thread. The expressions in the *print* and *assign* commands use variables that occur in the current thread; access to variables that reside in thread-private memory is transparent. The current thread can be changed to any other thread using the *thread* command.

When program execution is suspended, all threads are stopped. The fact that there are multiple threads is indicated by the display of multiple source lines that state where each thread is stopped. If program execution is suspended because a thread reached a set breakpoint, the thread becomes the current thread and is stopped *at* the source line where the breakpoint is set. The other threads are generally stopped within a basic block, that is, *near* an eventpoint line, which *csd* reports.

The display message contains as a prefix the *thread_id* enclosed in angular brackets. The current thread is identified by an asterisk preceding the *thread_id*. Possible *thread_id* numbers range from 0 to one less than the number of CPUs. When there is only one thread (the parallelized program

is run single-threaded (sequentially) or at the time program execution is suspended there are no multiple threads), the *<thread_id>* prefix does not appear in the display message. In this case, output is the same as for a non-parallelized program.

The debugger is unaware of threads created during program execution (unless the monitoring of thread creation is specified by the ***stop thread*** or ***trace thread*** commands), and therefore, they are not under debugger control. If such a thread terminates before program execution is suspended, the debugger will never know of its existence. This can even happen if you issue the *call* command, because an existing thread executes the *call* command, thereby allowing the hardware to create a thread.

If a parallelized program uses multiple threads, true concurrency occurs. In other words, each thread executes on its own CPU simultaneously at least once. However, once program execution is suspended, multiple threads may not execute concurrently, but in pseudo-parallel. For example, the threads may execute one after other on a single CPU in an order determined by the scheduling algorithm of the operating system.

4.3 A Multithread Debugging Philosophy

Assume you start with a program that executes correctly when compiled at the *-O2* level or lower. Then you compile the program at the *-O3* level with directives possibly inserted to force parallelization. The program now executes incorrectly. To determine the cause of the problem, first verify the parallelized program is correct when executed sequentially by issuing the *csh* command

limit concurrency 1

just before executing the program. If the program executes incorrectly, then either the compiler generated incorrect code or a directive was incorrectly inserted, that is, the code is not really capable of correctly executing in parallel.

If no directives were inserted to force parallelization, then the automatic parallelization performed by the compiler is incorrect. If directives were inserted, remove them and recompile at *-O3*. If the program still executes incorrectly, the fault lies with automatic parallelization.

If the fault lies with inserted parallelization directives, you can use *csd* to find the code that should not have been forced to be parallelized. When all threads of a parallelized section of code are stopped, you can direct your attention to an individual thread, examine thread local and shared data, and step or continue its execution. Controlling the order threads are executed overrides the operating system's scheduler.

Threads are created automatically by the hardware when a CPU becomes idle. Therefore, a program may use a single thread one time it is executed and use multiple threads when executed another time. When you debug a program containing parallel code, you want to be guaranteed that concurrency occurs so that the parallel portions are executed in parallel every time you debug the program. The *-f* option of *csd* guarantees concurrency by making the operating system allocate all CPUs to the program when it schedules a time slice for program execution. By using the *-f* option, you make sure that the program executes with multiple threads, so you can debug the program reliably.

One common error is to force a loop that calls a routine to be parallelized, and not to compile the called routine and all routines that can be called from it with the *-re* option. Another common error is to assume the local variables of a routine compiled with *-re* are initialized to 0; the local variables must be explicitly initialized by the routine.

Debugging parallel code requires the retention of much detailed information. To aid in remembering what transpired, record the debugging session using *script*(1).

Note: You can test if a program designed to execute in parallel runs correctly at lower optimization levels without removing directives inserted to force parallelization. At *-O2* and lower, the compiler silently ignores parallelization directives.

4.4 Commands for Debugging Multithreaded Code

csd lets you exert firm control over multithreaded programs by supporting thread-specific commands. You can set process eventpoints, monitor individual threads, examine user communication registers, examine thread-specific machine registers, and display thread stack backtraces.

4.4.1 New Commands

The new *csd* commands to support multithread debugging provide the ability to determine how many threads exist, to control how many threads should be permitted, to change the current thread, and to examine the user communication registers.

csd automatically keeps track of where the program is executing. When execution is suspended, you can use the *thread* command to determine which thread is the current thread or to change the current thread. The *thread* command has the following format:

thread [*thread_id*]

If *thread_id* is unspecified, the command display the number of the current thread. To change the current thread, specify the appropriate *thread_id* number.

The following screen demonstrates both forms of the *thread* command. The first command displays the number of the current thread, and the second command changes the current thread from thread 0 to thread 1.

```
(csd) thread
current thread: <0>
(csd) thread 1
current thread: <1>
(csd)
```

The *threads* command displays the *thread_id* numbers of all active threads. The command has the following format:

threads [*thread_limit*]

If no *thread_limit* is specified, the thread limit and thread numbers for all active threads are displayed, as shown in the next screen.

```
(csd) threads
thread limits: current = 2      next = 2
<0>:  0x8000130a    44*           call subprog (a, i)
<*1>:  0x800012b2    43*        do i = 1,10
(csd)
```

csd displays the value of the pc and the first line of the basic block for each active thread. Using the technique described in section 3.1, "Effects of Optimization," you can find the exact locus of execution by using the value of the pc. The technique is illustrated in section 3.2.3, "Tracking a Bug in Optimized Code."

To change the maximum number of threads to be created for the next *run* or *rerun* command, specify the appropriate number for the *thread_limit*. The command in the following screen changes the maximum number of threads to 1 for the next time you use the *run* or *rerun* command.

```
(csd) threads 1
thread limits: current = 2      next = 1
(csd)
```

Compiler-generated code uses the communication registers to hold values shared by threads. To examine the contents of the communication registers, use the *cregs* command whose format is:

cregs [*index*]

If *index* is not specified, the contents of all user communication registers and their lock bits are displayed; otherwise, the contents of the specified user communication register is displayed. Communication registers are number sequentially starting at 0. The number following the colon indicates the status of the lock bit (1=on, 0=off).

```
(csd) cregs
creg[00]=00001c00fffffffe:1, 0000000000000000:0, 0000000000000000:0,
creg[03] through creg[63] = 0000000000000000:0
(csd)
```

If the communication registers are empty, *cregs* displays all zeros. *csd* displays the output of these commands in hexadecimal, and condenses duplicate array elements in the output as shown in the next screen.

```
(csd) cregs
creg[00]=00001c00fffffffe:1, 0000000000000000:0, 0000000000000000:0,
creg[03] through creg[11] = 0000000000000000:0
creg[12] through creg[17] = 0003801000000008:1
creg[18] through creg[63] = 0000000000000000:0
(csd)
```

4.4.2 Enhancements to Old Commands

This section indicates which of the old *csd* commands are enhanced to support the debugging of multithreaded code. Old commands not mentioned apply to the current thread.

- **cont** [**all**] [*signum* | *signame*]
  Continue execution of **all** threads or the current thread without/with the specified signal.

- **mode** [**sequential** | **chained**]
  On a C2, both the SEQ and SQS bits in the PSW are set when the mode is set to **sequential**, and both reset when the mode is set to **chained**. On a C1, SEQ is set/reset when the mode is set to **sequential/chained**; there is no SQS bit on a C1. Recall that if SEQ is set, instructions are executed sequentially (no pipelining and overlap occurs). If SQS is set, stores to memory occur in instruction execution order, i.e., not in a non-sequential order.

- **next** [**all**] [*count*]
  Step **all** threads, or the current thread, *count* eventpoint line(s), treating called routines as a single step. When you specify the *all* option, threads are executed one at a time. Thread execution is numerically ordered by the *thread_id*. To change the order in which threads are executed, use a combination of thread numbers with the *next* command.

  If any thread traps out with a fatal error while it is executing, subsequent execution of that thread, or any other thread, is not possible. When a thread is stepped from eventpoint to eventpoint, the debugger may hang if the thread terminates. To get control back to the command level, type CTRL-C and proceed with the remaining threads.

- **regs** [**all**]
  Print the value of the scalar and address registers of the current thread or **all** threads.

- **step** [**all**] [*count*]
  Single step **all** threads or the current thread *count* eventpoint line(s), honoring eventpoints within a called routine. When you specify the *all* option, threads are executed one at a time. Thread execution is ordered numerically by the *thread_id*. To change the order in which threads are executed, use a combination of thread numbers with the *step* command.

  If any thread receives a fatal error while it is executing, neither it nor any other thread may execute. When a thread is stepped from eventpoint to eventpoint, the debugger may hang if the thread terminates. To get control back to the command level, type CTRL-C and proceed with the remaining threads.

- **stop threads** [**in** *routine*]
  Suspend execution when a new thread is created or an existing thread terminates, and display the source lines *at* or *near* which all threads were stopped. When a new thread is created, it becomes the current thread. When a thread terminates, the current thread remains the same unless the current thread terminated, in which case one of the remaining threads becomes the current thread. If a *routine* is specified, execution only stops when the thread is in that routine.

- **trace threads** [**in** *routine*]
  Trace the creation of a new thread or the termination of an existing thread. When a new thread is created, the source line at which the new thread begins execution is displayed. When an existing thread terminates, a message indicating which thread

terminated is displayed. If a *routine* is specified, thread tracing will be limited to execution in that routine.

- **vregs** [**all**] [*index*]
  Dispaly the values of all vector registers of the current thread or all threads. To display the values of a single vector register, specify the desired *index*.

4.5 Sample Multithreaded Routine

As an example, consider the subroutine, *tred1*, used in EISPACK. Notice that this is the same example used in Figure 3-1, "Source of Optimized Subroutine, *tred1*". *tred1* reduces a real symmetric matrix to a symmetric tridiagonal matrix using orthogonal similarity transformations. *tred1* is compiled at *-O3 -db*. The source is listed in Figure 4-4; the listing is generated from within *csd* to show the location of the basic blocks.

All loops are vectorized except the loops at lines 12, 50, and 76, and for the vectorized loops, their strip mine was parallelized. Notice the new eventpoint lines (asterisks appear on the *continue* statements at lines 10, 28, 37, 59, 69, and 90) that weren't there at the *-O2* level, when compared to the source listing in Figure 3-1. This is because parallelization introduces additional basic blocks.

Figure 4–4: Source of Parallelized Subroutine

```
1*      subroutine tred1(nm,n,a,d,e,e2)
2  c
3       integer i,j,k,l,n,ii,nm,jp1
4       double precision a(nm,n),d(n),e(n),e2(n)
5       double precision f,g,h,scale
6  c
7*      do 100 i = 1, n
8*        d(i) = a(n,i)
9         a(n,i) = a(i,i)
10*  100 continue
11  c   .......... for i=n step -1 until 1 do -- ..........
12*      do 300 ii = 1, n
13*        i = n + 1 - ii
14         l = i - 1
15         h = 0.0d0
16         scale = 0.0d0
17         if (l .lt. 1) go to 130
18  c   .......... scale row (algol tol then not needed) ..........
19*        do 120 k = 1, l
20*  120   scale = scale + dabs(d(k))
21  c
22*        if (scale .ne. 0.0d0) go to 140
23  c
24*        do 125 j = 1, l
25*          d(j) = a(l,j)
26           a(l,j) = a(i,j)
27           a(i,j) = 0.0d0
28*  125   continue
29  c
30*  130   e(i) = 0.0d0
31         e2(i) = 0.0d0
32         go to 300
33  c
34*  140   do 150 k = 1, l
35*          d(k) = d(k) / scale
36           h = h + d(k) * d(k)
37*  150   continue
38  c
39*        e2(i) = scale * scale * h
40         f = d(l)
41         g = -dsign(dsqrt(h),f)
42         e(i) = scale * g
43         h = h - f * g
44         d(l) = f - g
45         if (l .eq. 1) go to 285
46  c   .......... form a*u ..........
47*        do 170 j = 1, l
48*  170   e(j) = 0.0d0
49  c
50*        do 240 j = 1, l
51*          f = d(j)
52           g = e(j) + a(j,j) * f
53           jp1 = j + 1
54           if (l .lt. jp1) go to 220
55  c
56*          do 200 k = jp1, l
57*            g = g + a(k,j) * d(k)
58             e(k) = e(k) + a(k,j) * f
59*  200     continue
60  c
61*  220     e(j) = g
62   240   continue
63  c   .......... form p ..........
64*        f = 0.0d0
65  c
66*        do 245 j = 1, l
67*          e(j) = e(j) / h
68           f = f + e(j) * d(j)
69*  245   continue
```

```
70  c
71*       h = f / (h + h)
72  c   .......... form q ..........
73*       do 250 j = 1, l
74*  250   e(j) = e(j) - h * d(j)
75  c   .......... form reduced a ..........
76*       do 280 j = 1, l
77*         f = d(j)
78          g = e(j)
79  c
80*         do 260 k = j, l
81*  260      a(k,j) = a(k,j) - f * e(k) - g * d(k)
82  c
83*  280   continue
84  c
85*  285   do 290 j = 1, l
86*         f = d(j)
87         d(j) = a(l,j)
88         a(l,j) = a(i,j)
89         a(i,j) = f * scale
90*  290   continue
91  c
92*  300 continue
93  c
94*     return
95      end
```

4.6 Monitoring a Multithreaded Program

This section shows how to monitor and control the execution of a multithreaded program. The parallelized routine, *tred1*, called from another program. When the program enters *tred1*, various commands are used to illustrate the parallel execution. Through the following series of screen examples, you are taken step-by-step through the execution of the parallel routine. Because this program is executing on a complex with two CPUs, the maximum number of threads available to the program is two.

Step 1 Invoke *csd* with the *-f* option to ensure that the program will run in parallel. Then list a portion of the program to show the context of the call to *tred1*.

```
% csd -f a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...
(csd) list 11, 20
   11*                 call dranv (iseed,n,a(1,j))
   12            end do
   13
   14*           t0 = cputime ( 0.0 )
   15            call  tred1 (lda,n,a,w,e,e2)
   16*           t1 = cputime ( 0.0 )
   17            call  tqlrat (n,w,e2,ierr)
   18*           time = t1 - t0
   19            print 90,time,dasum(n,w,1),dnrm2(n,w,1)
   20      90    format (f10.4,2e18.8)
(csd)
```

Step 2 Set a breakpoint at line 14 (the beginning of the basic block) so you can step into *tred1* at the first instruction. Then run the program; execution stops when the breakpoint is reached.

```
(csd) stop at 14
[1] stop at "tred1t.f":14
(csd) run
[1] stopped in MAIN_ at line 14 in file "tred1t.f"
   14*          t0 = cputime ( 0.0 )
(csd)
```

Step 3 Display the current thread. Display the status of all threads. The display shows that the program may use two threads and displays the address of the current instruction.

```
(csd) thread
current thread: <0>
(csd) threads
thread limits: current = 2      next = 2
<*0>:  0x80001230    14*        t0 = cputime ( 0.0 )
(csd)
```

Step 4 Step into *tred1*. Then step two instructions to move into the loop. When the loop is entered, the program begins executing in parallel as indicated by the appearance of a thread number (<*0>).

```
(csd) step
stopped in tred1.tred1 at line 4 in file "tred1.f"
    1*        subroutine tred_1(nm,n,a,d,e,e2)
(csd) step
stopped in tred1.tred1 at line 7 in file "tred1.f"
    7*        do 100 i = 1, n
(csd) step
<*0> stopped in tred1.tred1 at line 8 in file "tred1.f"
    8*           d(i) = a(n,i)
(csd)
```

Step 5 Display all active threads. Thread 0 is executing in the inner and thread 1 is just entering the loop.

```
(csd) threads
thread limits: current = 2      next = 2
<*0>:  0x8000156c     8*           d(i) = a(n,i)
<1>:  0x800014d6     7*        do 100 i = 1, n
(csd)
```

Note: Thread 1 came into existence while stepping thread 0. When the process stops, the location of thread 1 is inderminate. In the above example. thread 1 may have been at line 8 when execution stopped.

Step 6 Execute one basic block in the current thread (thread 0).

```
(csd) step
<*0> stopped in tred1.tred1 at line 8 in file "tred1.f"
   8*    d(i) = a(n,i)
(csd)
```

Step 7 Execute one basic block in both threads. Both threads advance to the next instruction. It appears that both threads will execute the same instruction next, each with different data. While both threads will execute the same physical instruction, the data is different because each thread uses a different value of *i*. The integrity of the information is therefore maintained.

```
(csd) step all
<*0> stopped in tred1.tred1 at line 8 in file "tred1.f"
    8*          d(i) = a(n,i)
<1> stopped in tred1.tred1 at line 8 in file "tred1.f"
    8*          d(i) = a(n,i)
(csd)
```

Step 8 Execute one basic block in both threads. Three of the four strip mines are now complete.

```
(csd) step all
<*0> stopped in tred1.tred1 at line 10 in file "tred1.f"
    8*    d(i) = a(n,i)
<1> stopped in tred1.tred1 at line 10 in file "tred1.f"
   10*    100 continue
(csd)
```

When both threads execute, thread 0 gets the fourth strip mine and thread 1 finishes because there are no more strip mines to execute. Thread 1 waits until thread 0 finishes.

Step 9 Execute one basic block in the current thread (thread 0). Since the fourth strip mine is completed, thread 0 finishes because there are not more strip mines to execute.

```
(csd) step
<*0> stopped in tred1.tred1 at line 10 in file "tred1.f"
   10*    100 continue
(csd)
```

Step 10 Step in thread 0, which terminates the thread. Display the current thread and information about all threads.

```
(csd) step
thread terminated; new current thread is <1>
(csd) thread
current thread: <1>
(csd) threads
thread limits: current = 2       next = 2
<*1>:  0x800015ee    10*     100 continue
(csd)
```

Note: It is a common occurrence for the *thread_id* of the single thread prior to and immediately after a parallelized loop to change; the order in which threads terminate is indeterminate.

Step 11 Execute one basic block in the current thread (now thread 1), advancing to line 12. The thread number is not displayed because only one thread is currently active. Then execute one more instruction.

```
(csd) step
stopped in tred1.tred1 at line 12 in file "tred1.f"
   12*         do 300 ii = 1, n
(csd) step
stopped in tred1.tred1 at line 13 in file "tred1.f"
   13*            i = n + 1 - ii
(csd)
```

Step 12 Step one basic block in the current thread. Executing this line executes the source lines 13-17 as a basic block.

```
(csd) step all
stopped in tred1.tred1 at line 19 in file "tred1.f"
   19*            do 120 k = 1, l
(csd)
```

Step 13 Execute one instruction in the current thread (thread 1). When the instruction is executed, another thread (thread 0) is created. Thread 0 is now the active thread, but because of the pipelined nature of the machine, thread 1 is displayed.

```
(csd) step all
<1> stopped in tred1.tred1 at line 20 in file "tred1.f"
   20*    120    scale = scale + dabs(d(k))
(csd)
```

Step 14 Display the status of all active threads. The output shows that thread 0 is current. The *where* command shows that line 19 (in thread 0) is at the top of the call stack.

```
(csd) threads
thread limits: current = 2      next = 2
<*0>:  0x80001786    19*          do 120 k = 1, 1
<1>:  0x8000180e    20*    120    scale = scale + dabs(d(k))
(csd) where
tred1(nm=513, n=512, a=ARRAY, d=ARRAY, e=ARRAY, e2=ARRAY) line 19 in "tred1.f"
MAIN_, line 14 in "tred1t.f"
main(0x1, 0xffffce60, 0xffffce68) at 0x80003d0c
(csd)
```

Step 15 Change the current thread to thread 1. The output from the *where* command shows that line 20 (from thread 1) is now at the top of the call stack.

```
(csd) thread 1
current thread: <1>
(csd) where
tred1(nm=513, n=512, a=ARRAY, d=ARRAY, e=ARRAY, e2=ARRAY) line 20 in "tred1.f"
MAIN_, line 14 in "tred1t.f"
main(0x1, 0xffffce60, 0xffffce68) at 0x80003d0c
(csd)
```

Step 16 Display the contents of the communication registers.

```
(csd) cregs
creg[00]=0000000000000000:1, 00000002fffffff:1, 0000000000000000:0,
creg[03] through creg[63] = 0000000000000000:0
(csd) q
%
```

4.7 Debugging a Multithreaded Program

To illustrate tracking a bug involving multiple threads, suppose the basic block at line 39 in *tred1* is split into two parallel sections using the BEGIN_TASKS, NEXT_TASK, END_TASKS directives. The code is modified as follows:

```
38  c
39* C$DIR BEGIN_TASKS
40*        e2(i) = scale * scale * h
41         f = d(l)
42         g = -dsign(dsqrt(h),f)
43  C$DIR NEXT_TASK
44*        e(i) = scale * g
45         h = h - f * g
46         d(l) = f - g
47  C$DIR END_TASKS
48*        if (l .eq. 1) go to 285
```

For discussion purposes, lines 40-42 will be designated the first thread, and lines 44-46 the second thread. The threads that are created are not really independent; the second thread assigns to *f* and *h*, which the first thread uses. Depending on the order and the speed the threads execute, the first thread will either get the previous values of the variables or their new value.

Timing bugs are extremely difficult to isolate because it is difficult to reproduce the parallelism that occurred. In this case, the wrong values of *f* and *h* will be used whenever the second thread executes before the first thread. **To force this situation, the assembly code was modified because the generated code starts the threads in their textual order and the threads are too short to demonstrate the timing problem.** Note that you can achieve the same results by switching lines 40-42 with lines 44-46.

The bug manifests itself as a floating-point overflow at line 49, the first executable basic block after the two threads. Because of pipelining, a few instructions will have been executed after the instruction causing the exception. Therefore, some statement other than that at line 49 caused the exception.

To determine which of the two threads and which statement within the thread is at fault, you must determine the last instruction executed. You must examine the assembly code to find which instruction really executed last. This technique is similar to that described in Chapter 3, "Debugging Optimized Code."

The next series of screens take you step-by-step through the process of isolating the bug introduced to this routine.

Step 1 Invoke *csd* and run the program. The program terminates because of a floating-point exception.

```
% csd a.out
Convex Symbolic Debugger.
Type 'help' for help.
reading symbolic information ...
(csd) run

Floating point exception (floating overflow) in tred1.tred1 at line 48 in file "tred1.f"
   48*          if (l .eq. 1) go to 285
(csd)
```

The display shows that the error occurred at line 48. However, the code on this line (a logical expression) cannot generate this type of error so the error must have occurred elsewhere.

Step 2 List the code preceding the stated location of the error. Since line 48 is located after a section of parallelized code, it is likely that the error occurred during parallel execution of the two threads.

```
(csd) list 38,48
   38   c
   39*  C$DIR BEGIN_TASKS
   40*            e2(i) = scale * scale * h
   41             f = d(l)
   42             g = -dsign(dsqrt(h),f)
   43   C$DIR NEXT_TASK
   44*            e(i) = scale * g
   45             h = h - f * g
   46             d(l) = f - g
   47   C$DIR END_TASKS
   48*            if (l .eq. 1) go to 285
(csd)
```

Step 3 Recompile the source code using the *-S* option to create the assembly code file, *tred1.s*. The redirection of *stderr* prevents the optimization information from being displayed on the screen. Then assemble *tred1.s* and store the output in *tred1.list* to get the relative locations for the instructions.

```
(csd) sh fc -O3 -db -S tred1.f >&optSummary
(csd) sh as -l tred1.s >tred1.list
(csd)
```

Step 4 Search the file *tred1.list* for the entry point of *tred1*, which is used in the calculation as the offset of the routine from *main*.

```
(csd) sh grep _tred1 tred1.list
000023  00000000                        .stabs  "tred1:F21",0x24,0,0,_tred1_
000176  0000000c                        .globl  _tred1_ ;ENTRY
000177                          _tred1_:
000184  0000000c                        .stabs  "tred1:F21",0x24,0,0,_tred1_
(csd)
```

Step 5 Determine the relative location of the pc (next instruction to be executed). Then search for the memory address in *tred1.list*. This line contains the next instruction to be executed.

```
(csd) print $pc-&tred1+0xc
0x9c8
(csd) sh grep 000009c8 tred1.list
000957  000009c8                         .stabd  0x44,0,48
000959  000009c8  324000000078           ld.w    LUU+120,s0       ; #48, ?18a
(csd)
```

Step 6 Display the ten lines surrounding line 957 using the *tail* and *head* commands. The instruction is a label. Therefore, the last instruction(s) to be executed were either those immediately preceding the label or those preceding a branch to the label.

```
(csd) sh tail +957 tred1.list | head
000951  000009b2  2a39fe48               ld.w    -440(fp),a1      ; #39, ?151
000952  000009b6  55c1                   sub.d   s0,s1   ; #46
000953  000009b8  37090000               st.l    s1,0(a1)         ; #46, D
000954  000009bc  55dc                   sub.d   s3,s4   ; #45
000955  000009be  372a0000               st.l    s2,0(a5)         ; #44, E
000956  000009c2  374400000028           st.l    s4,LU+40         ; #45, H
000957  000009c8                         .stabd  0x44,0,48
000958                         L4d:
000959  000009c8  324000000078           ld.w    LUU+120,s0       ; #48, ?18a
000960  000009ce  324100000080           ld.w    LUU+128,s1       ; #48, ?c8c
(csd)
```

Step 7 Examine the registers to determine which register contains the reserved operand. The display shows that the reserved operand is stored in the s1 register.

```
(csd) regs
pc=80001dc0 psw=82548680
sp=          e007fffc a1=         00000000 a2=         00000003 a3=         00000080
a4=          00000100 a5=         80232fa0 ap=         8000132c fp=         ffffcde8
s0=4f2ea1ee2fce3122 s1=8000000000000000 s2=5e4d52d205508e52 s3=de4d52d200000001
s4=5e4d52d200000000 s5=cb1001fe8dee3ec9 s6=0000000000000c00
s7=3f93ec3400000000
(csd)
```

By looking at the assembly code (Step 6), you can determine that the instructions immediately preceeding 0x9c8 are not suspect because the floating-point instruction (sub.d) does not involve s1 in its calculations. Therefore, the error must be located near the code that branched to label L4d.

Step 8 Determine the location of the branch label L4d in the assembly code. Then display the instructions preceding the branch instruction (jbr).

```
(csd) sh grep L4d tred1.list
000939  0000098e  711d                         jbr     L4d                ; #43
000958                                L4d:
(csd) sh tail +936 tred1.list | head
000930  00000968  224000000000                 callq   _mth$d_sign        ; #42
000931  0000096e  334100000030                 ld.l    LU+48,s1           ; #44, SCALE
000932  00000974  65c0                         neg.d   s0,s0    ; #42
000933  00000976  5749                         mul.d   s1,s1    ; #40
000934  00000978  2a3dfdb8                     ld.w    -584(fp),a5        ; #40, ?c89
000935  0000097c  374000000020                 st.l    s0,LU+32           ; #42, G
000936  00000982  334200000028                 ld.l    LU+40,s2           ; #40, H
000937  00000988  5751                         mul.d   s2,s1    ; #40
000938  0000098a  37290000                     st.l    s1,0(a5)           ; #40, E2
000939  0000098e  711d                         jbr     L4d                ; #43
(csd)
```

The first floating-point instruction preceding the branch is a *mul.d* instruction involving s1. The comments in the code indicate that variables loaded into s1 and s2 are *scale* and *h*, respectively.

Step 9 Display the values of *scale* and *h*.

```
(csd) print scale,h
9.53331e+92 4.57703e+145
(csd)
```

Step 10 Display the source code again.

```
(csd) list 38,48
   38   c
   39*  C$DIR BEGIN_TASKS
   40*            e2(i) = scale * scale * h
   41             f = d(l)
   42             g = -dsign(dsqrt(h),f)
   43   C$DIR NEXT_TASK
   44*            e(i) = scale * g
   45             h = h - f * g
   46             d(l) = f - g
   47   C$DIR END_TASKS
   48*            if (l .eq. 1) go to 285
(csd)
```

Looking at thread 1 (40-42) shows that *scale * scale * h* will indeed overflow with the current values of *scale* and *h*. The error occurs at line 45 in thread 2 (44-46), because only the value of *h* is modified in this thread.

Chapter 5
Using Profilers

This chapter explains how to use profilers to measure program execution and how to use this information to improve the efficiency and speed of a program. CONVEX supports three profilers: *prof*, *gprof* and *bprof*, each of which are detailed in this chapter. Sample profiling sessions are included for each of the profilers described.

Some of the topics covered in this chapter include

- overview of profiling
- *prof*
- *gprof*
- *bprof*

Profiling is an important mechanism with which to measure the execution of programs. You can use the information obtained from profiling to optimize program execution and to achieve high levels of testing coverage. The *Consultant* package provides three different profiling utilities. *prof* and *gprof* are timing profilers that provide information about the amount of execution time spent in a particular routine. *bprof*, on the other hand, is a counting profiler that provides data about the number of times a particular statement was executed.

Each of the profilers provides you with detailed information about the execution characteristics of a program. At the highest level, *prof* is used to get an overall look at the program. At this level, it is easy to identify which routines in the program account for most of the execution time. Using the *gprof* utility then gives you hierarchical call graph information as well as execution times. In many cases, these two utilities may provide enough data to identify the processing bottlenecks right away. It may be necessary, however, to examine a particular routine at a still finer level, e.g., the execution flow on a statement-by-statement level. Use *bprof* in this case.

The remainder of this chapter discusses these three profiling utilities in greater detail.

5.1 Using *prof*

This section includes an overview of *prof* and a discussion of its basic capabilities. Examples of *prof* output are also included.

5.1.1 Capabilities Overview

prof reads the symbol table in the object file that you specify (*a.out* is the default), and correlates it with the profile file *mon.out* created when the object was executed. *prof* displays the following information (by default in descending order):

- The percentage of time spent executing in each routine.
- The number of times each routine was called.
- The number of milliseconds spent in each routine.

If you specify multiple profiles, the output can be the sum of the profiles.

5.1.2 Basic Operations

In order for *prof* to count the calls to a routine, you must first compile the file containing the routine with the *-p* compiler option. This is true whether you are using one of the C (*cc* or *vc*) or the FORTRAN (*fc*) compiler. You can measure the execution time within a routine without compiling it with the *-p* option, but you would not get a count of the calls and you must specify the *-p* option for any linking steps performed. When the program executes, it accumulates (internally) profile statistics. The program writes these profile statistics to a file called *mon.out* upon normal program termination (i.e., exit). Some statistics are lost if the program terminates abnormally (e.g., bus error). You can then use *prof* to examine the results.

The format of the *prof* command is as follows:

prof [-a] [-l] [-n] [-z] [-s] [*object* [**mon.out**[...]]]

where

-a	Reports all symbols, rather than just external symbols.
-l	Sorts the output by symbol value. This is the default sorting; *-n* overrides *-l*.
-n	Sorts output by the number of calls.
-z	Displays even routines that have zero usage according to call counts and accumulated time.
-s	Produces a summary profile file in *mon.sum*. This option is only useful when you specify multiple profiles.

If you use the *-l* and *-n* options on the same command line, *pmd* uses the one appearing second. Using only the *-n* option sorts output according the the number of calls only.

Figures 5-1 and 5-2 are sample FORTRAN and C programs with *prof* output.

Figure 5–1: Sample FORTRAN Program With *prof* Output

```
% cat b.f

#define N 200

        PROGRAM prog

        INTEGER i

        i= 0
        DO WHILE (i .LT. 10)
         WRITE(6,100)i
100      FORMAT('i is 'I)
         call gen()
         call process()
         i= i + 1
        END DO

        END


        SUBROUTINE gen()

        INTEGER dat(N)
        COMMON dat
        INTEGER i,j

        i= 0
        DO WHILE (i .LT. N)
         call ran(j)
         dat(i)= j
         i= i + 1
        END DO

        END


        SUBROUTINE process()

        INTEGER dat(N)
        COMMON dat
        INTEGER i,j

        i= 0
        DO WHILE (i .LT. N -1)
         call ran(j)
         dat(i)= -100 * dat(i) + j
         i= i + 1
        END DO

        END
% fc -o b -p b.f
% b
i is       0
i is       1
i is       2
i is       3
i is       4
i is       5
i is       6
i is       7
i is       8
i is       9

% prof b

 %time  cumsecs  #call  ms/call  name
  33.3   0.03              mcount
  22.2   0.05    10    2.00  _process_
```

```
22.2   0.07  3990   0.01  _ran_
11.1   0.08     5   1.43  _sbrk
11.1   0.09    13   0.77  _sigvec
 0.0   0.09     1   0.00  _MAIN__
 0.0   0.09     1   0.00  __findbuf
 0.0   0.09     1   0.00  __wrtchk
 0.0   0.09    10   0.00  __xflsbuf
 0.0   0.09    10   0.00  _bcopy
 0.0   0.09    10   0.00  _c_sfe
 0.0   0.09     3   0.00  _canseek
 0.0   0.09     3   0.00  _close
 0.0   0.09     1   0.00  _f_exit
 0.0   0.09     1   0.00  _f_init
 0.0   0.09     3   0.00  _fclose
 0.0   0.09     3   0.00  _fflush
 0.0   0.09    13   0.00  _find_unit
 0.0   0.09     3   0.00  _finode
 0.0   0.09    10   0.00  _fmt_bg
 0.0   0.09     3   0.00  _for$clos
 0.0   0.09    20   0.00  _for$do_fio
 0.0   0.09    10   0.00  _for$e_wsfe
 0.0   0.09    10   0.00  _for$j_icvt_a
 0.0   0.09    10   0.00  _for$s_wsfe
 0.0   0.09     4   0.00  _free
 0.0   0.09     7   0.00  _fstat
 0.0   0.09    10   0.00  _fwrite
 0.0   0.09    10   0.00  _gen_
 0.0   0.09     4   0.00  _gtty
 0.0   0.09     3   0.00  _ini_std
 0.0   0.09     4   0.00  _ioctl
 0.0   0.09     4   0.00  _isatty
 0.0   0.09     1   0.00  _main
 0.0   0.09     5   0.00  _malloc
 0.0   0.09     1   0.00  _profil
 0.0   0.09     1   0.00  _sigstack
 0.0   0.09     2   0.00  _t_runc
 0.0   0.09    10   0.00  _w_ed
 0.0   0.09    10   0.00  _w_ned
 0.0   0.09    10   0.00  _write
 0.0   0.09    10   0.00  _wrt_AP
 0.0   0.09    10   0.00  _wrt_IM
 0.0   0.09    10   0.00  _x_wend
 0.0   0.09     1   0.00  _getpagesize
 0.0   0.09    10   0.00  _memchr
```

Figure 5–2: Sample C Program With *prof* Output

```
% cat b.c
# define N 200
int   dat[N];
gen ()
{
    int   i;
    for (i = 0; i < N; i++)
    {
         dat[i] = rand ();
    }
}

process ()
{
    int   i;
    for (i = 0; i < N - 1; i++)
         dat[i] = -100 * dat[i] + rand();
}
main ()
{
    int   i;
    for (i = 0; i < 10; i++)
    {
         printf("i is %d\n", i);
         gen ();
         process ();
    }
}
^D

% cc -o b -p b.c
% b
i is 0
i is 1
i is 2
i is 3
i is 4
i is 5
i is 6
i is 7
i is 8
i is 9

% prof b

%time   cumsecs   #call   ms/call   name

  4.0     0.18     3990      0.05    _rand
 16.0     0.25                       mcount
 12.0     0.30       10      5.00    _process
  8.0     0.33       10      3.33    __doprnt
  8.0     0.37       10      3.33    _gen
  8.0     0.40       10      3.33    _write
  4.0     0.42        1     16.67    _main
  0.0     0.42       70      0.00    __flsbuf
  0.0     0.42        1      0.00    _fstat
  0.0     0.42        1      0.00    _gtty
  0.0     0.42        1      0.00    _ioctl
  0.0     0.42        1      0.00    _isatty
  0.0     0.42       10      0.00    _printf
  0.0     0.42        1      0.00    _profil
  0.0     0.42       30      0.00    __strout
  0.0     0.42        1      0.00    _getpagesize
  0.0     0.42        1      0.00    _malloc
  0.0     0.42        3      0.00    _sbrk
  0.0     0.42        9      0.00    udiv64
  0.0     0.42        9      0.00    urem64
```

5.2 Using *gprof*

This section includes an overview of *gprof* and its basic capabilities. A discussion about analyzing *gprof* output follows the examples.

5.2.1 Capabilities Overview

Like *prof*, *gprof* produces an execution profile of C or FORTRAN programs. *gprof* correlates the symbol table in the specified object file (*a.out* is the default) with the profile file (*gmon.out*) created when the program was executed. *gprof* first displays a flat profile, similar to the *prof* output (i.e., total execution times and call counts for each function). *gprof* then goes on to multiply these times along the call graph, find call cycles, and force calls into a cycle to share that cycle time. The next display listing shows the functions sorted according to the time they and their descendants represent. Below each function entry, *gprof* shows the direct call graph children and how their times are distributed to the function. A similar display above each function entry shows how the function's time and the time of its descendants is distributed to its direct-call graph parents.

gprof also displays call cycles, showing both the whole cycle and its members, and their contributions to the time and call counts of the cycle.

5.2.2 Basic Operations

In order for *gprof* to monitor calls to a routine, you must first compile the file containing the routine with the *-pg* compiler option. This is true whether you are using the FORTRAN (*fc*) or C (*cc* or *vc*) compiler. You must also specify the *-pg* option for any linking steps performed. When the program executes, it accumulates (internally) profile statistics. The program writes these statistics to a file called *gmon.out* on normal program termination (i.e., *exit*). Some statistics are lost if the program terminates abnormally (e.g., bus error). The *-pg* option also links in versions of the library routines that are compiled for profiling. You can then use *gprof* to examine the results.

The format of the *gprof* command is as follows:

> **gprof** [-a] [-b] [-c] [-e *name*] [-E *name*] [-f *name*]
> [-F *name*] [-s] [-z] [*object* [**gmon.out**[...]]]

where the options signify:

- **-a** Suppresses displaying of statically declared functions. This option displays static-function information (time samples, calls to and from other functions) that corresponds to the function loaded just before the static function in the object file.
- **-b** *gprof* prints extensive descriptions of each field in the profile. If you do not want to display these descriptions, specify the *-b* option.
- **-c** Examines the text space of the object file to find the static call graph of the program. This option indicates static-only parents or children with call counts of 0.
- **-e** *name* Suppresses displaying of the graph profile for the specified routine and all its descendants (unless it has other ancestors that are not suppressed). You may specify more than one *-e* option, each with a single *name*.
- **-E** *name* Suppresses displaying of the graph profile for the specified routine and all its descendants as in the *-e* option, but also excludes the time spent in *name* and its descendants from the total and percentage time computations.
- **-f** *name* Displays call graph profile entry of the specified routine and its descendants. You may specify more than one *-f* option, with one *name* per option.

-F *name* — Displays the call graph profile entry of the specified routine and its descendants, but uses only the times of the displayed routines in total time and percentage computations. You may specify multiple *-F* options, with one *name* per option. The *-F* option overrides the *-E* option.

-s — Produces a summary profile file called *gmon.sum*, which contains the sum of the profile information in all the specified profile files. You may specify *gmon.sum* in subsequent *gprof* executions (also with the *-s* option) to accumulate profile data over several runs of an object file.

-z — Displays even those routines that have zero usage according to call counts and accumulated time. This option is useful when used with the *-c* option for finding routines that never get called.

Figures 5-3 and 5-4 are sample FORTRAN and C programs with *gprof* output.

Figure 5–3: Sample FORTRAN Program With *gprof* Output

```
% cat b.f

#define N 200

   PROGRAM PROG

   INTEGER I

   I= 0
   DO WHILE(I .LT. 10)
    WRITE(6,100)I
100     FORMAT('I IS'I)
    CALL GEN()
    CALL PROCESS()
    I= I + 1
   END DO

   END


   SUBROUTINE GEN()

   INTEGER DAT(N)
   COMMON DAT
   INTEGER I,J

   I= 0
   DO WHILE(I .LT. N)
    CALL RAN(J)
    DAT(I)= J
    I= I + 1
   END DO

   END


   SUBROUTINE PROCESS()

   INTEGER I,J
   INTEGER DAT(N)
   COMMON DAT

   I= 0
   DO WHILE(I .LT. N - 1)
    CALL RAN(J)
    DAT(I)= -100 * DAT(I) + J
    I= I + 1
   END DO

   END


% fc -o b -pg b.fP
% b
I is      0
I is      1
I is      2
I is      3
I is      4
I is      5
I is      6
I is      7
I is      8
I is      9

% gprof -b b

granularity: each sample hit covers 4 byte(s) for 7.69% of 0.13 seconds
```

```
%time cumsecs seconds   calls name

 46.2  0.06  0.06     mcount
 23.1  0.09  0.03  2000 _ran_
 15.4  0.11  0.02     mcount_calladdr
  7.7  0.12  0.01    20 _for$do_fio
  7.7  0.13  0.01     1 _write
  0.0  0.13  0.00    13 _find_unit
  0.0  0.13  0.00    13 _sigvec
  0.0  0.13  0.00    10 _bcopy
  0.0  0.13  0.00    10 _c_sfe
  0.0  0.13  0.00    10 _fmt_bg
  0.0  0.13  0.00    10 _for$e_wsfe
  0.0  0.13  0.00    10 _for$j_icvt_a
  0.0  0.13  0.00    10 _for$s_wsfe
  0.0  0.13  0.00    10 _fwrite
  0.0  0.13  0.00    10 _gen_
  0.0  0.13  0.00    10 _process_
  0.0  0.13  0.00    10 _w_ed
  0.0  0.13  0.00    10 _w_ned
  0.0  0.13  0.00    10 _wrt_AP
  0.0  0.13  0.00    10 _wrt_IM
  0.0  0.13  0.00    10 _x_wend
  0.0  0.13  0.00     8 _malloc
  0.0  0.13  0.00     7 _fstat
  0.0  0.13  0.00     7 _sbrk
  0.0  0.13  0.00     4 _fflush
  0.0  0.13  0.00     4 _free
  0.0  0.13  0.00     4 _gtty
  0.0  0.13  0.00     4 _ioctl
  0.0  0.13  0.00     4 _isatty
  0.0  0.13  0.00     3 _canseek
  0.0  0.13  0.00     3 _close
  0.0  0.13  0.00     3 _fclose
  0.0  0.13  0.00     3 _finode
  0.0  0.13  0.00     3 _for$clos
  0.0  0.13  0.00     3 _ini_std
  0.0  0.13  0.00     2 _t_runc
  0.0  0.13  0.00     1 _MAIN__
  0.0  0.13  0.00     1 __findbuf
  0.0  0.13  0.00     1 __wrtchk
  0.0  0.13  0.00     1 __xflsbuf
  0.0  0.13  0.00     1 _f_exit
  0.0  0.13  0.00     1 _f_init
  0.0  0.13  0.00     1 _fseek
  0.0  0.13  0.00     1 _lseek
  0.0  0.13  0.00     1 _main
  0.0  0.13  0.00     1 _profil
  0.0  0.13  0.00     1 _sigstack

granularity: each sample hit covers 4 byte(s) for 14.29% of 0.07 seconds

                      called/total     parents
index %time    self descendents called+self    name   index
                      called/total     children

                                    <spontaneous>
[1]    71.4  0.00       0.05             start [1]
         0.00       0.05     1/1         _main [2]

---------------------------------------------

         0.00       0.05     1/1         start [1]
[2]    71.4  0.00       0.05      1      _main [2]
         0.00       0.04     1/1         _MAIN__ [3]
         0.00       0.01     1/1         _f_exit [11]
         0.00       0.00    13/13         _sigvec [24]
         0.00       0.00     1/5         _malloc [39]
         0.00       0.00     1/1         _sigstack [53]
         0.00       0.00     1/1         _f_init [50]
```

```
-----------------------------------------------

        0.00     0.04     1/1        _main [2]
[3]   57.1  0.00     0.04     1      _MAIN__ [3]
        0.00     0.03    10/10        _gen_ [5]
        0.01     0.00    10/20        _for$do_fio [7]
        0.00     0.01    10/10        _for$e_wsfe [14]
        0.00     0.00    10/10        _for$s_wsfe [29]
        0.00     0.00    10/10        _process_ [31]

-----------------------------------------------

        0.03     0.00  2000/2000       _gen_ [5]
[4]   42.9  0.03     0.00  2000      _ran_ [4]

-----------------------------------------------

        0.00     0.03    10/10        _MAIN__ [3]
[5]   42.9  0.00     0.03    10      _gen_ [5]
        0.03     0.00  2000/2000       _ran_ [4]

-----------------------------------------------

                                   <spontaneous>
[6]   28.6  0.02     0.00            mcount_calladdr [6]

-----------------------------------------------

        0.01     0.00    10/20        _MAIN__ [3]
        0.01     0.00    10/20        _for$e_wsfe [14]
[7]   14.3  0.01     0.00    20      _for$do_fio [7]
        0.00     0.00    10/10        _w_ned [33]
        0.00     0.00    10/10        _w_ed [32]
        0.00     0.00    10/10        _x_wend [36]

-----------------------------------------------

        0.00     0.00     1/4        _fseek [16]
        0.00     0.01     3/4        _fclose [13]
[8]   14.3  0.00     0.01     4      _fflush [8]
        0.00     0.01     1/1        __xflsbuf [10]

-----------------------------------------------

        0.00     0.01     3/3        _f_exit [11]
[9]   14.3  0.00     0.01     3      _for$clos [9]
        0.00     0.01     3/3        _fclose [13]
        0.00     0.00     2/2        _t_runc [15]
        0.00     0.00     3/4        _free [40]

-----------------------------------------------

        0.00     0.01     1/1        _fflush [8]
[10]  14.3  0.00     0.01     1      __xflsbuf [10]
        0.01     0.00     1/1        _write [12]

-----------------------------------------------

        0.00     0.01     1/1        _main [2]
[11]  14.3  0.00     0.01     1      _f_exit [11]
        0.00     0.01     3/3        _for$clos [9]

-----------------------------------------------

        0.01     0.00     1/1        __xflsbuf [10]
[12]  14.3  0.01     0.00     1      _write [12]

-----------------------------------------------

        0.00     0.01     3/3        _for$clos [9]
[13]  10.7  0.00     0.01     3      _fclose [13]
```

```
                0.00      0.01     3/4          _fflush [8]
                0.00      0.00     3/3          _close [45]
                0.00      0.00     1/4          _free [40]

-----------------------------------------------

                0.00      0.01    10/10          _MAIN__ [3]
[14]     7.1    0.00      0.01    10         _for$e_wsfe [14]
                0.01      0.00    10/20          _for$do_fio [7]

-----------------------------------------------

                0.00      0.00     2/2           _for$clos [9]
[15]     3.6    0.00      0.00     2         _t_runc [15]
                0.00      0.00     1/1           _fseek [16]

-----------------------------------------------

                0.00      0.00     1/1           _t_runc [15]
[16]     3.6    0.00      0.00     1         _fseek [16]
                0.00      0.00     1/4           _fflush [8]
                0.00      0.00     1/1           _lseek [51]

-----------------------------------------------

                0.00      0.00     3/13          _ini_std [47]
                0.00      0.00    10/13          _c_sfe [26]
[23]     0.0    0.00      0.00    13         _find_unit [23]

-----------------------------------------------

                0.00      0.00    13/13          _main [2]
[24]     0.0    0.00      0.00    13         _sigvec [24]

-----------------------------------------------

                0.00      0.00    10/10          _fwrite [30]
[25]     0.0    0.00      0.00    10         _bcopy [25]

-----------------------------------------------

                0.00      0.00    10/10          _for$s_wsfe [29]
[26]     0.0    0.00      0.00    10         _c_sfe [26]
                0.00      0.00    10/13          _find_unit [23]

-----------------------------------------------

                0.00      0.00    10/10          _for$s_wsfe [29]
[27]     0.0    0.00      0.00    10         _fmt_bg [27]

-----------------------------------------------

                0.00      0.00    10/10          _wrt_IM [35]
[28]     0.0    0.00      0.00    10         _for$j_icvt_a [28]

-----------------------------------------------

                0.00      0.00    10/10          _MAIN__ [3]
[29]     0.0    0.00      0.00    10         _for$s_wsfe [29]
                0.00      0.00    10/10          _c_sfe [26]
                0.00      0.00    10/10          _fmt_bg [27]

-----------------------------------------------

                0.00      0.00    10/10          _x_wend [36]
[30]     0.0    0.00      0.00    10         _fwrite [30]
                0.00      0.00    10/10          _bcopy [25]
                0.00      0.00     1/1           __wrtchk [49]

-----------------------------------------------
```

```
                0.00        0.00    10/10          _MAIN__ [3]
[31]     0.0    0.00        0.00    10           _process_ [31]

-----------------------------------------------

                0.00        0.00    10/10          _for$do_fio [7]
[32]     0.0    0.00        0.00    10           _w_ed [32]
                0.00        0.00    10/10          _wrt_IM [35]

-----------------------------------------------

                0.00        0.00    10/10          _for$do_fio [7]
[33]     0.0    0.00        0.00    10           _w_ned [33]
                0.00        0.00    10/10          _wrt_AP [34]

-----------------------------------------------

                0.00        0.00    10/10          _w_ned [33]
[34]     0.0    0.00        0.00    10           _wrt_AP [34]

-----------------------------------------------

                0.00        0.00    10/10          _w_ed [32]
[35]     0.0    0.00        0.00    10           _wrt_IM [35]
                0.00        0.00    10/10          _for$j_icvt_a [28]

-----------------------------------------------
                0.00        0.00    10/10          _for$do_fio [7]
[36]     0.0    0.00        0.00    10           _x_wend [36]
                0.00        0.00    10/10          _fwrite [30]

-----------------------------------------------

                0.00        0.00     1/7           __findbuf [48]
                0.00        0.00     3/7           _canseek [44]
                0.00        0.00     3/7           _finode [46]
[37]     0.0    0.00        0.00     7           _fstat [37]

-----------------------------------------------

                0.00        0.00     7/7           _malloc [39]
[38]     0.0    0.00        0.00     7           _sbrk [38]

-----------------------------------------------

                             3             _malloc [39]
                0.00        0.00     1/5           _main [2]
                0.00        0.00     1/5           __findbuf [48]
                0.00        0.00     3/5           _ini_std [47]
[39]     0.0    0.00        0.00     5+3         _malloc [39]
                0.00        0.00     7/7           _sbrk [38]
                             3             _malloc [39]

-----------------------------------------------

                0.00        0.00     1/4           _fclose [13]
                0.00        0.00     3/4           _for$clos [9]
[40]     0.0    0.00        0.00     4           _free [40]

-----------------------------------------------

                0.00        0.00     4/4           _isatty [43]
[41]     0.0    0.00        0.00     4           _gtty [41]
                0.00        0.00     4/4           _ioctl [42]

-----------------------------------------------

                0.00        0.00     4/4           _gtty [41]
[42]     0.0    0.00        0.00     4           _ioctl [42]

-----------------------------------------------
```

```
             0.00      0.00     1/4         __findbuf [48]
             0.00      0.00     3/4         _canseek [44]
[43]    0.0  0.00       0.00      4         _isatty [43]
             0.00      0.00     4/4         _gtty [41]

-----------------------------------------------

             0.00      0.00     3/3         _ini_std [47]
[44]    0.0  0.00       0.00      3         _canseek [44]
             0.00      0.00     3/7         _fstat [37]
             0.00      0.00     3/4         _isatty [43]

-----------------------------------------------

             0.00      0.00     3/3         _fclose [13]
[45]    0.0  0.00       0.00      3         _close [45]

-----------------------------------------------

             0.00      0.00     3/3         _ini_std [47]
[46]    0.0  0.00       0.00      3         _finode [46]
             0.00      0.00     3/7         _fstat [37]

-----------------------------------------------

             0.00      0.00     3/3         _f_init [50]
[47]    0.0  0.00       0.00      3         _ini_std [47]
             0.00      0.00     3/13         _find_unit [23]
             0.00      0.00     3/5         _malloc [39]
             0.00      0.00     3/3         _canseek [44]
             0.00      0.00     3/3         _finode [46]

-----------------------------------------------

             0.00      0.00     1/1         __wrtchk [49]
[48]    0.0  0.00       0.00      1         __findbuf [48]
             0.00      0.00     1/7         _fstat [37]
             0.00      0.00     1/5         _malloc [39]
             0.00      0.00     1/4         _isatty [43]

-----------------------------------------------

             0.00      0.00     1/1         _fwrite [30]
[49]    0.0  0.00       0.00      1         __wrtchk [49]
             0.00      0.00     1/1         __findbuf [48]

-----------------------------------------------

             0.00      0.00     1/1         _main [2]
[50]    0.0  0.00       0.00      1         _f_init [50]
             0.00      0.00     3/3         _ini_std [47]

-----------------------------------------------

             0.00      0.00     1/1         _fseek [16]
[51]    0.0  0.00       0.00      1         _lseek [51]

-----------------------------------------------

             0.00      0.00     1/1         _moncontrol [130]
[52]    0.0  0.00       0.00      1         _profil [52]

-----------------------------------------------

             0.00      0.00     1/1         _main [2]
[53]    0.0  0.00       0.00      1         _sigstack [53]

-----------------------------------------------
```

Figure 5–4: Sample C Program With *gprof* Output

```
% cat b.c
# define   N    200
int   dat[N];
gen ()
{
  int   i;
  for (i = 0; i < N; i++)
  {
  dat[i] = rand ();
  }
}

process ()
{
  int   i;
  for (i = 0; i < N - 1; i++)
  dat[i] = -100 * dat[i] + rand();
}
main ()
{
  int   i;
  for (i = 0; i < 10; i++)
  {
  printf("i is %d\n", i);
  gen ();
  process ();
  }
}

% cc -o b -pg b.c
% b
i is 0
i is 1
i is 2
i is 3
i is 4
i is 5
i is 6
i is 7
i is 8
i is 9

% gprof b
granularity: each sample hit covers 4 byte(s) for
2.63% of 0.38 seconds

flat profile:
%time          the percentage of the total running time of the program used by this
               function.
cumsecs        a running sum of the number of seconds accounted for by this function and
               those listed above it.
seconds        the number of seconds accounted for by this function alone. This is the
               major sort for this listing.
calls          the number of times this function was invoked, if this function is profiled,
               else blank.
name           the name of the function. This is the minor sort for this listing.


%time   cumsecs   seconds   calls   name

 55.3    0.21      0.21             mcount
 21.1    0.29      0.08      3990   _rand
  7.9    0.32      0.03        10   _process
  5.3    0.34      0.02        10   __doprnt
  5.3    0.36      0.02        10   _write
  2.6    0.37      0.01        70   __flsbuf
  2.6    0.38      0.01        10   _printf
  0.0    0.38      0.00        10   _gen
```

```
 0.0     0.38     0.00        1    _fstat
 0.0     0.38     0.00        1    _gtty
 0.0     0.38     0.00        1    _ioctl
 0.0     0.38     0.00        1    _isatty
 0.0     0.38     0.00        1    _main
 0.0     0.38     0.00        1    _profil
```

```
call graph profile:      The sum of self and descendants is the major      sort for this
listing.

   function entries:
index               the index of the function in the call graph listing, as an aid to locating
                    it (see below).
%time               the percentage of the total time of the program accounted for by this
                    function and its descendants.
self                the number of seconds spent in this function itself.
descendants         the number of seconds spent in the descendants of this function on behalf
                    of this function.
called              the number of times this function is called (other than recursive calls).
self                the number of times this function calls itself recursively.
name                the name of the function, with an indication of its membership in a cycle,
                    if any.
index               the index of the function in the call graph listing, as an aid to locating
                    it.

      parent listings:
self*               the number of seconds of this function's self time which is due to calls
                    from this parent.
descendants*        the number of seconds of this function's descendant time which is due to
                    calls from this parent.
called**            the number of times this function is called by this parent. This is the
                    numerator of the fraction which divides up the function's time to its
                    parents.
total*              the number of times this function was called by all of its parents. This
                    is the denominator of the propagation fraction.
parents             the name of this parent, with an indication of the parent's membership in
                    a cycle, if any.
index               the index of this parent in the call graph listing, as an aid in locating
                    it.
   children listings:
self*               the number of seconds of this child's self time which is due to being
                    called by this function.
descendant*         the number of seconds of this child's descendant's time which is due to
                    being called by this function.
called**            the number of times this child is called by this function. This is the
                    numerator of the propagation fraction for this child.
total*              the number of times this child is called by all functions. This is the
                    denominator of the propagation fraction.
children            the name of this child, and an indication of its membership in a cycle, if
                    any.
index               the index of this child in the call graph listing, as an aid to locating
                    it.

  * these fields are omitted for parents (or
  children) in the same cycle as the function. If
  the function (or child) is a member of a cycle,
  the propagated times and propagation denominator
  represent the self time and descendant time of the
  cycle as a whole.

  ** static-only parents and children are indicated
  by a call count of 0.
  cycle listings:
  the cycle as a whole is listed with the same
  fields as a function entry. Below it are listed
  the members of the cycle, and their contributions
  to the time and call counts of the cycle.

granularity: each sample hit covers 4 byte(s) for 5.88% of 0.17 seconds
```

```
                called/total parents index %time self descendants called+self name index
                called/total children

                          <spontaneous>      [1]        80.0        0.08            0.00
mcount_calladdr [1]

-----------------------------------------------

                          <spontaneous> [2]   20.0  0.00     0.02           start [2]
        0.00     0.02     1/1        _main [4]

-----------------------------------------------

        0.01     0.00  1990/3990     _process [6]
        0.01     0.00  2000/3990     _gen [5] [3]   20.0  0.02     0.00  3990      _rand
[3]

-----------------------------------------------

        0.00     0.02     1/1       start [2] [4]   20.0  0.00     0.02    1      _main [4]
        0.00     0.01    10/10       _gen [5]
        0.00     0.01    10/10       _process [6]
        0.00     0.00    10/10       _printf [16]

-----------------------------------------------

        0.00     0.01    10/10       _main [4] [5]   10.0  0.00     0.01    10      _gen [5]
        0.01     0.00  2000/3990      _rand [3]

-----------------------------------------------

        0.00      0.01      10/10        _main  [4]  [6]    10.0    0.00      0.01     10
_process [6]
        0.01     0.00  1990/3990      _rand [3]

-----------------------------------------------

        0.00      0.00      9/9        __doprnt  [15]  [7]     0.0   0.00      0.00      9
udiv64 [7]

-----------------------------------------------

        0.00      0.00      9/9        __doprnt  [15]  [8]     0.0   0.00      0.00      9
urem64 [8]

-----------------------------------------------

        0.00      0.00     80/80       __strout  [14]  [13]    0.0   0.00      0.00     80
__flsbuf [13]
        0.00     0.00     1/1       _fstat [18]
        0.00     0.00     1/1       _malloc [23]
        0.00     0.00     1/1       _isatty [22]

-----------------------------------------------

        0.00      0.00     30/30       __doprnt  [15]  [14]    0.0   0.00      0.00     30
__strout [14]
        0.00     0.00    80/80       __flsbuf [13]

-----------------------------------------------

        0.00      0.00     10/10       _printf  [16]  [15]    0.0   0.00      0.00     10
__doprnt [15]
        0.00     0.00    30/30       __strout [14]
        0.00     0.00    9/9        urem64 [8]
        0.00     0.00    9/9        udiv64 [7]

-----------------------------------------------

        0.00     0.00    10/10       _main [4] [16]   0.0  0.00     0.00    10      _printf
[16]
```

```
         0.00      0.00    10/10        __doprnt [15]
-----------------------------------------------

         0.00      0.00    3/3         _malloc [23] [17]   0.0  0.00      0.00    3       _sbrk
[17]

-----------------------------------------------

         0.00       0.00       1/1         __flsbuf  [13]  [18]     0.0    0.00       0.00       1
_fstat [18]

-----------------------------------------------

         0.00       0.00       1/1         _malloc  [23]  [19]     0.0    0.00       0.00       1
_getpagesize [19]

-----------------------------------------------

         0.00      0.00    1/1         _isatty [22] [20]   0.0  0.00      0.00    1       _gtty
[20]
         0.00      0.00    1/1         _ioctl [21]

-----------------------------------------------

         0.00      0.00    1/1         _gtty [20] [21]   0.0   0.00      0.00    1      _ioctl
[21]

-----------------------------------------------

         0.00       0.00       1/1         __flsbuf  [13]  [22]     0.0    0.00       0.00       1
_isatty [22]
         0.00      0.00    1/1         _gtty [20]

-----------------------------------------------

                      1          _malloc [23]
         0.00      0.00      1/1        __flsbuf  [13]  [23]    0.0    0.00       0.00        1+1
_malloc [23]
         0.00      0.00    3/3         _sbrk [17]
         0.00      0.00    1/1         _getpagesize [19]
                      1          _malloc [23]

-----------------------------------------------

         0.00      0.00      1/1         _moncontrol  [43]  [24]    0.0   0.00       0.00       1
_profil [24]

-----------------------------------------------
```

5.2.3 Analyzing *gprof* Output

The *gprof* output in Figure 5-3 is divided into two parts. The first part is a flat profile of the execution of each of the routines comprising the program. This is similar to the data obtained from *prof*. The data is ordered based on the percentage of total running time of the program used by a particular function. You can use this part of the output to obtain an overview of the execution characteristics of the program's components.

Use the second component of the listing, the call graph, for a more detailed analysis. This output is ordered in terms of decreasing percentage of total program time used, and is interpreted as follows

Each routine in the program has its own entry which is uniquely identified by an index number appearing on the far left of the entry. Use this index to easily locate the entry for a particular routine. The entry itself is divided into three separate fields. The first field begins with an index number and contains data pertaining to the function itself, including the number of times it is called, and the amount of time spent executing its descendants.

Immediately on top of this field are entries for each of the routines that called the function. These "parent listings" clarify the amount of time each parent used when calling the function. The routine responsible for the greatest amount of processing time could be a possible candidate for eventual optimization.

The remaining entry contains execution data for each routine called by the function. These "child entries" contain the amount of processing time spent in the child, and the total amount of time spent processing the child's dependents.

Using the sample output from the previous C example (Figure 5-4), interpret the call graph entry for the routine *_process* as follows.

The first field tells us that the index number for *_process* is 6, that 10.0 percent of the program's total execution time was spent in *_process* and its descendants, that 0.00 seconds were actually spent executing this routine, and that 0.01 seconds were spent executing its children. It also shows that *_process* was called exactly 10 times.

The parent listing indicates the routine *_main* was the only routine that called *_process*, accounting for all ten calls. Hence, *_main* also accounts for all of the 0.01 seconds spent executing *_process*, and all of the 0.05 seconds used by *_process*'s descendants.

The child listing shows that *_process* called only one routine, *_rand*, exactly 10 times and is responsible for 1990 of the 3990 calls to *_rand*.

5.3 Using *bprof*

This section includes an overview of *bprof* and a discussion of its basic capabilities. Examples of *bprof* output are also included.

5.3.1 Capabilities Overview

As stated earlier, *prof* and *gprof* are timing profilers. While they can provide you with much useful information, the data obtained may have several inconsistencies. For instance, it is possible for the time for one routine to get attributed to another routine. Also, many thousands of instructions may be executed during each sampling interval, making it impossible to get reliable counts for smaller subroutines. Finally, if the program is somehow dependent on the hardware clock, e.g., communications applications, the profiling sampling is not random.

As an alternative, *bprof* produces source-level execution count listings of C or FORTRAN routines. Knowing how many times a statement is executed enables you to easily determine which parts of the program are executed most frequently. This in turn helps in identification of processing bottlenecks and indicates which parts of the routine have not been executed.

A source line whose code is not executed has a blank count. A source line that corresponds to no executable code also has a blank count even though it occurs among other lines which have non-zero counts.

For example, the line containing the *do* statement in the following C program has a blank count.

```
do
{ ++i;
} while (i<10);
```

The profile data is taken from the *bprof* profile file (*bmon.out* default) that is created by programs compiled and linked with the *-pb* option of *vc*, *cc* and *fc*. *bprof* reads the symbol table in the named object file (*a.out* default) and correlates it with the *bprof* profile file. If you specify more than one profile file, the *bprof* output shows the sum of the information in the named files.

5.3.2 Basic Operations

The format of the *bprof* command is:

bprof [*options*] [*object* [**bmon.out**]]

where *options* are

-f Lists counting information by routine name. This option lists the routine name, the number of source lines executed in the routine, the sum total of all the lines executed, the number of source lines not executed, and the number of times the routine was called.

-m Lists counting information by module name. This option lists the number of routines in the module, the number of calls made to the routines, the number of source lines executed, the sum total of all the lines executed and the number of source lines not executed.

-s Produces a file named *bmon.sum* that contains the sum of all the specified profile data files.

-l Generates a source listing containing the number of times each source line was executed.

-I *directory* Adds the named directory to the list of directories searched when looking for a source file. If you do not specify a directory, *bprof* searches for files in the current directory and in the directory in which *a.out* is located. This option is only useful when used in conjunction with the *-l* option.

Note that you must specify one of the options when using *bprof*.

Figures 5-5 and 5-6 are sample FORTRAN and C programs with *bprof* output using the *-f* option.

Figure 5–5: Sample FORTRAN Program With *bprof* Output

```
% cat b.f

#define N 200

    PROGRAM PROG

    INTEGER I

    I= 0

    DO WHILE (I .LT. 10)
      WRITE(6,100)I
100     FORMAT('I IS 'I)
      CALL GEN()
      CALL PROCESS()
      I= I + 1
    END DO

    END


    SUBROUTINE GEN()

    INTEGER DAT(N)
    COMMON DAT,

    INTEGER I,J

    I= 0
    DO WHILE (I .LT. N)
      CALL RAN(J)
      DAT(I)= J
      I= I + 1
    END DO

    END


    SUBROUTINE PROCESS()

    INTEGER I,J
    INTEGER DAT(N)
    COMMON DAT

    I= 0
    DO WHILE (I .LT. N - 1)
      CALL RAN(J)
      DAT(I)= -100 * DAT(I) + J
      I= I + 1
    END DO

    END


% cc -pb -o b b.f
% b
I is 0
I is 1
I is 2
I is 3
I is 4
I is 5
I is 6
I is 7
I is 8
I is 9

% bprof -f b bmon.out
```

```
bprof - Convex source level profiler

SUMMARY BY ROUTINE

number of  number of      number not  sum of   routine name    file name
calls     lines executed  executed    lines
-----------------------------------------------------------------------------
    1         8          0        53   MAIN_        fig35.f
   10         8          0      8040    gen
   10         8          0      8000    process
```

Figure 5–6: Sample C Program With *bprof* Output

```
% cat b.c
# define   N    200
int   dat[N];
gen ()
{
   int   i;
   for (i = 0; i < N; i++)
   {
   dat[i] = rand ();
   }
}

process ()
{
   int   i;
   for (i = 0; i < N - 1; i++)
   dat[i] = -100 * dat[i] + rand();
}
main ()
{
   int   i;
   for (i = 0; i < 10; i++)
   {
   printf("i is %d\n", i);
   gen ();
   process ();
   }
}
^D

% cc -pb -o b b.c
% b
i is 0
i is 1
i is 2
i is 3
i is 4
i is 5
i is 6
i is 7
i is 8
i is 9

% bprof -f b bmon.out
bprof - Convex source level profiler

SUMMARY BY ROUTINE

number of  number of      number not  sum of   routine name    file name
calls     lines executed  executed    lines
--------------------------------------------------------------------------------
    10        5         0      4030    gen          fig36.c
    10        4         0      2020    process
     1        7         0        43   main
```

In Figures 5-5 and 5-6, the programs were compiled with the *-pb* compiler option to generate the basic *bprof* profile file *bmon.out*. After executing the program, *bprof* is run with the *-f* option.

The *-f* option breaks down the count listing by the three routines in the program. You can see how many times each routine was called, how many lines in each routine were executed, and how many lines were not executed.

Figures 5-7 and 5-8 show *bprof* output for the same programs, this time using the *-m* option.

Figure 5–7: *bprof* Output for FORTRAN Program, *-m* Option

```
% bprof -m b bmon.out
bprof - Convex source level profiler

SUMMARY BY MODULE

number of  number of  number of      number not  sum of   file name
routines   calls      lines executed  executed   lines
--------------------------------------------------------------------------------
    3        21         24         0      16093    fig35.f
```

Figure 5–8: *bprof* Output for C Program, *-m* Option

```
% bprof -m b bmon.out
bprof - Convex source level profiler

SUMMARY BY MODULE

number of  number of  number of      number not  sum of   file name
routines   calls      lines executed  executed   lines
--------------------------------------------------------------------------------
    3        21         16         0      6093    fig36.c
```

In the summary by module, there is only one module in the sample program. This program contains 3 routines, which are called a total of 21 times. There are 13 source lines executed, and 0 lines not executed.

Finally, *bprof* is run with the *-l* option. Figures 5-9 and 5-10 contain the output from the C and FORTRAN programs.

Figure 5–9: *bprof* Output From FORTRAN Program, *-l* Option

```
bprof - Convex source level profiler

        1  #define N 200
        2
        3   PROGRAM PROG
        4
        5   INTEGER I
        6
   1    7   I= 0
        8
   1    9   DO WHILE (I .LT. 10)
  10   10          WRITE(6,100)I
       11  100 FORMAT('I IS 'I)
  10   12          CALL GEN()
  10   13          CALL PROCESS()
  10   14          I= I + 1
  10   15        END DO
       16
   1   17        END
       18
       19
  10   20        SUBROUTINE GEN()
       21
       22   INTEGER DAT(N)
       23   COMMON DAT
       24
       25   INTEGER I,J
       26
  10   27        I= 0
  10   28        DO WHILE (I .LT. N)
2000   29          CALL RAN(J)
2000   30          DAT(I)= J
2000   31          I= I + 1
2000   32        END DO
       33
  10   34        END
       35
       36
  10   37        SUBROUTINE PROCESS()
       38
       39   INTEGER I,J
       40   INTEGER DAT(N)
       41   COMMON DAT
       42
  10   43        I= 0
  10   44        DO WHILE (I .LT. N - 1)
1990   45          CALL RAN(J)
1990   46          DAT(I)= -100 * DAT(I) + J
1990   47          I= I + 1
1990   48        END DO
       49
  10   50        END
       51
       52
```

Figure 5–10: *bprof* Output From C Program, *-l* Option

```
bprof - Convex source level profiler

        1 #define   N    200
        2 int   dat[N];
        3 gen ()
  10    4 {
        5    int   i;
  10    6    for (i = 0; i < N; i++)
        7    {
2000    8     dat[i] = rand ();
2000    9     }
  10   10 }
       11
       12 process ()
  10   13 {
       14    int   i;
  10   15    for (i = 0; i < N - 1; i++)
1990   16     dat[i] = -100 * dat[i] + rand();
  10   17 }
       18 main ()
   1   19 {
       20    int   i;
   1   21    for (i = 0; i < 10; i++)
       22    {
  10   23     printf("i is %d\n", i);
  10   24     gen ();
  10   25     process ();
  10   26     }
   1   27 }
```

The *-l* option produces a source listing of the program, numbers the lines, and shows how many times each line was executed.

When using *bprof* on C applications, multiple statements appearing on the same line are treated as a single source line. You can use the *indent*(1) utility to split these statements onto separate lines, giving you better visibility as to the execution of the statement.

For example, the two listings in Figure 5-11 were produced using the *-l* switch. The first contains multiple statements per source line, while the second has been run through *indent*. You can see that the second example contains more information.

Figure 5–11: Sample Source Listing Using *indent*

```
bprof - Convex source level profiler

     1  main()
1    2  {
     3    int flags;
     4    int j;
     5    int g[10];
     6
1    7    if (flags == 0) printf("statement10);
     8
1    9    for (j=0; j<10; j++) g[j]=j;
    10
1   11    for (j=0;j<10;j++) if (g[j] == 5)
    12    printf("statement20); else flags++;
1   13  }


bprof - Convex source level profiler


 1   1  main () {
     2    int  flags;
     3    int  j;
     4    int  g[10];
     5
 1   6    if (flags == 0)
     7      printf ("statement10);
     8
 1   9    for (j = 0; j < 10; j++)
10  10      g[j] = j;
    11
 1  12    for (j = 0; j < 10; j++)
10  13      if (g[j] == 5)
 1  14        printf ("statement20);
 1  15      else
 9  16        flags++;
    17
 1  18  }
```

Chapter 6
Post-Mortem Dump Utility (*pmd*)

This chapter describes *pmd*, a post-mortem dump analyzer. It describes basic *pmd* capabilities, how to invoke the utility, and how to reexamine *pmd* output using *csd*. There are also sample C and FORTRAN programs with *pmd* output.

Some of the topics covered in this chapter include

- capabilities overview
- using *pmd*
- *pmd* restrictions
- sample *pmd* session

6.1 Capabilities Overview

The *pmd* utility generates information about the program running under it, and prints the information to *stderr* if the program aborts and dumps core. The utility is structured so you can select how much of the data you want to print. It can also be used on programs containing optimized code. The post-mortem information generated in the most complete format is:

1. The signal that caused the program to abort.
2. A runtime stack backtrace to the approximate source line location where the program took the exception.
3. The contents of the machine registers.
4. A dump of the active local variables in each routine on the runtime stack (sorted alphabetically).
5. A dump of the global, or common, variables (sorted alphabetically).
6. The region of disassembled object code where the exception took place.
7. A summary of resources used by the program, such as execution time, elapsed time, percent of time spent in the CPU, size of shared memory and unshared memory, page faults, and swaps.

You can specify options on the *pmd* command line to limit the range of some of the information, or keep it from printing altogether. These options are discussed later in "Using pmd."

The information generated by *pmd* can be used as a "shortcut" in the debugging process. As soon as your program fails, you have access to detailed information showing you why. If you can spot the error in the *pmd* output, there is no reason to use a debugger. You can also narrow the area of study using this output, and speed up the process if using a debugger is necessary.

6.2 Using *pmd*

This section describes how to invoke *pmd*, the options available for formatting output, and how to use reexamine the output using *csd*.

6.2.1 Invoking *pmd*

For *pmd* to obtain information about the runtime stack backtrace, source line locations, and active variables, you must first compile your program with the appropriate compiler flag. If you are using the CONVEX FORTRAN compiler, *fc*, compile the program using the *-db* flag. If you are using one of the the C compilers, *vc* or *cc*, use the *-g* compiler flag.

Once you have compiled the program, invoke *pmd* as follows:

pmd [**-alsvS**] [**-d** *num*:**Inum**...] [**-t** *cpu-limit*] *program* [*arguments*]

where

-a specifies that the addresses within address registers a1 through a7 be dereferenced, and displayed in a set of commonly used formats. This means that the value of what the address points to is printed in hex, real*4, real*8, integer*4, and integer*8.

-d displays a specified number of array elements. The defaults for this option are 100 for the first dimension, 10 for the second dimension, 1 for the third dimension, and zero for dimensions 3-7. Dimensions must be separated by colons.

-l displays the post-mortem dump information in the long format. This format includes all the items listed in the previous section, "Capabilities Overview."

-s displays the post-mortem dump information in the short format. This format includes the signal that caused the abort, a runtime stack backtrace, and the approximate source code location of the exception. The short format is the default for *pmd.*

-S displays the region of source code lines where the program exception occurred.

-t specifies an execution limit, e.g., CPU seconds, when the program is invoked. You must specify the time limit as an argument on the *pmd* command line. For example, *pmd -t timelimit program.*

-v includes the contents of the machine's vector registers in the post-mortem dump listing. If the vector registers are empty, they appear as all zeros.

program is the name of the program you wish to run under *pmd.*

arguments are the arguments required by the program being executed.

You may specify more than one of the preceding options on the *pmd* command line. If you specify conflicting options (i.e, *-l* and *-s*,) however, *pmd* chooses the last conflicting option that you specify. For instance, if you specify *pmd -l -v -s*, you get the vector registers with the short format.

If you want to reexamine the *pmd* information, you can do so by invoking *csd* and executing the program with the *run* command (or you can specify a core file on the command line). When the program faults, enter

dump [> *filename*]

where *filename* is the name of file to which you redirect the output, if desired. You can also redirect *pmd* output using *stderr* by entering:

pmd *filename* >& *new.filename*

where *filename* is the file being executed and *new.filename* is the file to which you are redirecting the output.

6.3 Restrictions on *pmd*

In addition to using the correct compiler option (*-db* or *-g*), two more conditions must be met before *pmd* can generate a listing. First, you must have write permission in the directory where the program resides. This enables the creation of the core file, which is necessary for *pmd* to create a listing. Second, you must not exceed the maximum core file size specified by the "limit" parameter in your shell.

6.4 Optimization Level Restrictions

When you specify optimization when compiling your program, the location accuracy of source lines and active local variables becomes uncertain in the *pmd* listing. The higher the level of optimization, the higher the degree of uncertainty. You can, however, make the following assumptions about location accuracy:

- Regardless of the optimization level, memory locations of common block variables are guaranteed to be accurate at the call to a subprogram.
- The validity of memory location contents of active local variables cannot be guaranteed. In most code, however, the memory locations for local variables are likely to be accurate.
- The mapping of object code to source code is granular to the basic block. A basic block is a sequence of statements with no branches.
- The values of subprogram arguments at entry points to subprograms are accurate.

6.5 Sample Programs Run With *pmd*

The following C program (Figure 4-1) illustrates the use of *pmd* with the *-v* option specified. The program *sheep.c* is first compiled with the *-g* flag, then run with *pmd*. Since no optimization is specified, the vector registers are all zeros.

Figure 6–1: Sample C Program With *pmd* Output

```
% cat sheep.c

#include <stdio.h>

main()
{
  int main_var;

  main_var = 666;
  goat(1);
  exit(0);
}

goat(x)
int x;
{
  double goat_var;

  goat_var = -1.0;
  sheep(2);
}

sheep(y)
int y;
{
  char *cp = NULL;
  char *string = "This is a string";
  int i, j;
  int a[8], b[9][9];

  for (i = 0; i < 8; i++)
        a[i] = i;
  for (i = 0; i < 9; i++)
        for (j = 0; j < 9; j++)
          b[i][j] = (10*i) + j;
  i = a[8] / b[0][0];
  printf("%c", *cp);
}

% cc -g sheep.c
% pmd -v a.out

CONVEX post-mortem dump Report Wed Aug 20 13:53:45 1986
Signal Received: Floating exception (core dumped) --
        integer divide by zero trap



User time:            0.0
System time:           0.1
Total elapsed time:       0:00
Percent CPU:          0%
Shared memory size:     0 kbyte
Unshared memory size:    0 kbyte
Page faults:           1
Swaps:             0

 ---------------------
Vector Registers:  (contents displayed in hex)

vl=00000000 vs=00000000
vm=00000000000000000000000000000000

v0[000] through v0[127] = 0000000000000000

v1[000] through v1[127] = 0000000000000000

v2[000] through v2[127] = 0000000000000000
```

```
v3[000] through v3[127] = 0000000000000000

v4[000] through v4[127] = 0000000000000000

v5[000] through v5[127] = 0000000000000000

v6[000] through v6[127] = 0000000000000000

v7[000] through v7[127] = 0000000000000000

---------------------
Stack Backtrace:

exit.o(0x2) at 0xffffd068
sheep(y = 2), near line 35 in "sheep.c"
goat(x = 1), near line 18 in "sheep.c"
main(0x1, 0xffffce60, 0xffffce68),
near line 8 in "sheep.c"
---------------------
Approximate Area of Source Code:

  25    char *string = "This is a string";
  26    int i, j;
  27    int a[8], b[9][9];
  28
  29    for (i = 0; i < 8; i++)
  30    a[i] = i;
  31    for (i = 0; i < 9; i++)
  32    for (j = 0; j < 9; j++)
  33      b[i][j] = (10*i) + j;
  34    i = a[8] / b[0][0];
  35    printf("%c", *cp);
  36  }
  37
```

The next program (Figure 4-2) illustrates the use of *pmd* with both the *-v* and *-l* flags. The FORTRAN program is compiled with the *-db* option of the *fc* compiler.

Figure 6–2: FORTRAN Program With *pmd* Output

```
%cat demo.f

      program demo
      dimension x(10),z(10)
c
      do i=1,10
      z(i)=sum + x(i)
      enddo
      call sub1
      end
c
      subroutine sub1
      dimension y(10)
c
      do i=1,10
        y(i)=sum + y(i)
      enddo
      xyz=0
      call sub2(xyz)
      return
      end
c
      subroutine sub2(xyz)
      dimension y(10)
c
      do i=1,10
        y(i)=sum + y(i)
      enddo
      abort=abort/xyz
      return
      end

%fc -db demo.f
%pmd -v -l a.out

CONVEX post-mortem dump Report  Wed Aug 20 13:59:34 1986
Signal Received: Illegal instruction (core dumped) --
        privileged instruction trap



User time:           0.0
System time:          0.2
Total elapsed time:     0:00
Percent CPU:         0%
Shared memory size:     0 kbyte
Unshared memory size:    0 kbyte
Page faults:          2
Swaps:              0

 ---------------------
Scalar Registers: (contents displayed in hex)

pc=8000590c psw= 2000002
sp=80010344 a1=8000a050 a2=00000000 a3=ffffce9c
a4=00000000 a5=0f0a3a8c ap=80010358 fp=80010344
s0=0000000080001276 s1=0000000000000000 s2=ffc0000500000000
s3=0000000900000000 s4=0000000000000001 s5=0000000000000001
s6=00000000fffffff s7=000000008000a050
 ---------------------
Vector Registers: (contents displayed in hex)

vl=0000000a vs=00000004
vm=ffffffffffffffffffffffffffffffff

v0[000] through v0[127] = 0000000000000000

v1[000]=16dfce3200000000, 204e950500000000, 39a3507e00000000
v1[004]=367c69c300000000, 2df7d0b800000000, 034a914000000000
```

```
v1[008]=1d53440e00000000, 27c76e8e00000000, 0a065310ec545df5
v1[012]=117b0dfde057aed7, 2581e07ac928f8fe, 291f9793bf6b007b
v1[016]=108ce4cbd20156bb, 21ba3cc488386a89, 14cfa6c6f45c1d65
v1[020]=16a270977a391cc1, 325ef7e6d66bd9ac, 0f6fb83cface2bfa
v1[024]=25e1a21429b01373, 39325bc21cae8e07, 3198f2b29ed82ed8
v1[028]=1985cfeb0d525a07, 05006fc1b1171270, 0d0fa3228e8ab391
                .                       .                        .
                .                       .                        .
                .                       .                        .

v7[000] through v7[127] = 0000000000000000

- - - - - - - - - - - - - - - - - - - - -
Stack Backtrace:

abort at 0x8000590c
sigdie(0x8, 0x4, 0x80010388, 0x80001276) at 0x800015fc
exit.o(0x8000c370) at 0xffffd084
sub2(xyz = 0.0), near line 27 in "demo.f"
sub1, near line 18 in "demo.f"
MAIN_, near line 8 in "demo.f"
main(0x1, 0xffffce64, 0xffffce6c) at 0x80001248
- - - - - - - - - - - - - - - - - - - - - -
Dump of Active Variables:

abort at 0x8000590c

sigdie(0x8, 0x4, 0x80010388, 0x80001276) at 0x800015fc

exit.o(0x8000c370) at 0xffffd084

sub2(xyz = 0.0), near line 27 in "demo.f"
NAME    TYPE              CURRENT VALUE
abort   real*4 abort      abort = reserved operand
i       integer*4 i       i = 0
sum     real*4 sum        sum = 0.0
y       real*4 y(1:10)    y = ARRAY
- - - Arrays - - -
NAME        TYPE
y             real*4 y(1:10)
(1)     0.0,  0.0,  0.0,  0.0
(5) through (10)       0.0
                .
                .
                .
main(0x1, 0xffffce64, 0xffffce6c) at 0x80001248

- - - - - - - - - - - - - - - - - - - - - -
Approximate Area of Object Code:

ADDRESS     SYMBOL+OFFSET   INSTRUCTION
0x8000110e   sub2+1e        add.h       v5,v6,v4
0x80001110   sub2+20        ld.w        exit.o,s1
0x80001116   sub2+26        ld.w        @0(ap),s2
0x8000111a   sub2+2a        div.s       s2,s1
0x8000111c   sub2+2c        st.w        s1,exit.o
0x80001122   sub2+32        rtn
0x80001124   sub2+34        exit        0
0x80001128   sub2+38        exit        0
0x8000112c   sub2+3c        exit        1
0x80001130   sub2+40        mov   v0,s0,s0
0x80001132   sub2+42        add.h       v5,v6,v0
0x80001134   sub2+44        exit        0
0x80001138   main+0         sub.w       #36,a0
0x8000113c   main+4         ld.w        0(ap),s0
0x80001140   main+8         st.w        s0,exit.o
0x80001146   main+e         ld.w        4(ap),s0
0x8000114a   main+12        st.w        s0,exit.o
0x80001150   main+18        pshea       3c
0x80001154   main+1c        mov  a0,a6
0x80001156   main+1e        pshea       1
```

```
-------------------------
Approximate Area of Source Code:

  1      program demo
  2      dimension x(10),z(10)
  3  c
  4      do i=1,10
  5      z(i)=sum + x(i)
  6      enddo
  7      call sub1
  8      end
  9  c
 10       subroutine sub1
 11       dimension y(10)
 12  c
 13       do i=1,10
 14         y(i)=sum + y(i)
 15       enddo
 16       xyz=0
 17       call sub2(xyz)
 18       return
 19       end
 20  c
 21       subroutine sub2(xyz)
```

A
csd Summary

A.1 Source-Level Commands

Command	Description
alias	Print a list of all active user defined aliases and their current values.
alias *newname*	Print the value of alias *newname*
alias *newname cmd*	Equate *cmd* to *newname*
alias *newname* "*string*"	Equate *string* to *newname*
alias *newname*(*params*) "*string*"	Equate *string* to *newname* with parameters *params*
assign *var* = *exp*	Assign the value of the *exp* to *var*
call *routine*([*params*])	Call the user defined *routine* with optional parameters *params*
catch	Print a list of the caught signals
catch *signum* | *signame*	Catch signal *signum* or *signame*
cont	Continue execution
cont *signum* | *signame*	Continue execution with signal *signum* or *signame*
cont all	Continue execution for all threads
cont all *signum* | *signame*	Continue execution for all threads with signal *signum* or *signame*
cregs	Display the contents of the user communication registers
cregs *index*	Display the contents of the specified user communication register
delete *index* ...	Remove trace, stop or when of given *index*(es)
delete all	Remove all traces, stops, and whens
down	Move down the call stack one level
down *num*	Move down the call stack *num* levels
dump [> *filename*]	Print post-mortem dump information to *filename*
edit	Edit the current sourcefile
edit *filename*	Edit *filename*
edit *routine*	Edit the source file containing the *routine*
file	Print the name of the current source file
file *filename*	Change the current source file to *filename*
format	Print the current format for integers
format hex	Set the print format for integers to hexadecimal
format decimal	Set the print format for integers to decimal (default)
fpmode	Print the current floating-point mode
fpmode native	Set the current floating-point mode to native format
fpmode ieee	Set the current floating-point mode to IEEE format
fpmode auto	Automatically synchronize csd's floating-point mode with the program's mode
func	Print the name of the current routine
func *routine*	Change the current routine to *routine*
help	Print a summary of all commands
help *command*	Print help about *command*
ignore	Print a list of the ignored signals
ignore *signum* | *signame*	Ignore signal *signum* or *signame*
list	List 10 source lines in the current source file
list *first*[[,]*last*]	List source lines from *first* to *last* in the current source file
list *routine*	List the source to the *routine*
list eventpoints [*first*[[,]*last*]]	Display only those lines at which you can set an eventpoint
mode	Print the current execution mode

<table>
<tr><td>mode chained</td><td>Set the current execution mode to chained mode</td></tr>
<tr><td>mode sequential</td><td>Set the current execution mode to sequential (default)</td></tr>
<tr><td>next</td><td>Step one line, treating routines atomically</td></tr>
<tr><td>next count</td><td>Step count lines, treating routines atomically</td></tr>
<tr><td>next all</td><td>Execute one source line for all threads, treating routines atomically</td></tr>
<tr><td>next all count</td><td>Execute count source lines in each thread, treating routines atomically</td></tr>
<tr><td>print exp [, exp ...]</td><td>Print the value of the expressions</td></tr>
<tr><td>print type(exp)</td><td>Print the type cast value of exp (C only)</td></tr>
<tr><td>print exp \ type</td><td>Print the type cast value of exp (C only)</td></tr>
<tr><td>print exp \ var</td><td>Print the value of exp type cast to the type of var</td></tr>
<tr><td>quit</td><td>Exit csd</td></tr>
<tr><td>rerun [< filename] [> filename]</td><td>Begin execution of the program with the current arguments</td></tr>
<tr><td>rerun args [< filename] [> filename]</td><td>Begin execution of the program with new arguments</td></tr>
<tr><td>return</td><td>Return from routine on top of runtime stack</td></tr>
<tr><td>return routine</td><td>Return from routines until routine is on top of runtime stack</td></tr>
<tr><td>run [< filename] [> filename]</td><td>Begin execution of the program with no arguments</td></tr>
<tr><td>run args [< filename] [> filename]</td><td>Begin execution of the program with new arguments</td></tr>
<tr><td>set</td><td>Print the parameters used by dump when formatting listing output</td></tr>
<tr><td>set deref_aregs = true | false</td><td>Set dereferencing of address register values (default false)</td></tr>
<tr><td>set dump_lfmt = true | false</td><td>Set generation of detailed listing (default false)</td></tr>
<tr><td>set dumpsrc = true | false</td><td>Set displaying of stop location (default true)</td></tr>
<tr><td>set dumpvregs = true | false</td><td>Set displaying vector register values (default false)</td></tr>
<tr><td>set num_elements = num [, num...]</td><td>Set default array indices (default 0)</td></tr>
<tr><td>set precision = num</td><td>Set the precision for floating-point numbers to num (default 6)</td></tr>
<tr><td>sh cmd-line</td><td>Pass the command line to the shell for execution</td></tr>
<tr><td>source filename</td><td>Execute csd commands from filename</td></tr>
<tr><td>status [> filename]</td><td>Print trace, stop, and when statements currently in effect.</td></tr>
<tr><td>step</td><td>Single step one line, entering any called routines</td></tr>
<tr><td>step count</td><td>Single step count lines</td></tr>
<tr><td>step all</td><td>Execute one source line for all threads</td></tr>
<tr><td>step all count</td><td>Execute count instructions in each thread, entering any called routines</td></tr>
<tr><td>stop at line [if cond]</td><td>Stop execution at line in the current file</td></tr>
<tr><td>stop at "file":line [if cond]</td><td>Stop execution at line in the file</td></tr>
<tr><td>stop in routine [if cond]</td><td>Stop execution when the routine is called</td></tr>
<tr><td>stop if cond</td><td>Stop when condition true</td></tr>
<tr><td>stop var [if cond]</td><td>Stop when value of var changes</td></tr>
<tr><td>stop threads</td><td>Stop when a thread is created</td></tr>
<tr><td>stop threads [in routine]</td><td>Stop when a thread is created in the specified routine</td></tr>
<tr><td>thread</td><td>Display the current thread number</td></tr>
<tr><td>thread number</td><td>Change the current thread to number</td></tr>
<tr><td>threads</td><td>Display the thread limit and numbers for all active threads</td></tr>
<tr><td>threads number</td><td>Limit the child task of the next run or rerun command to number threads</td></tr>
<tr><td>trace</td><td>Trace each source line in the current file</td></tr>
<tr><td>trace line [if cond]</td><td>Trace execution of the line in the current file</td></tr>
<tr><td>trace "file":line [if cond]</td><td>Trace execution of the line in file</td></tr>
<tr><td>trace in routine [if cond]</td><td>Trace each source line while in the routine</td></tr>
<tr><td>trace routine [if cond]</td><td>Trace calls to the routine</td></tr>
</table>

trace *exp* **at** *line* [**if** *cond*]	Print *exp* when *line* in the current file is reached
trace *exp* **at** "*file*":*line* [**if** *cond*]	Print *exp* when *line* in *file* is reached
trace *var* [**in** *routine*] [**if** *cond*]	Trace changes to the variable while executing inside *routine*
trace threads	Detect the creation of a new thread and display its location
trace threads [**in** *routine*]	Detect the creation of a new thread and display its location while executing in the specified *routine*
unalias *name*	Remove the alias *name*
up	Move up the runtime stack one level
up *num*	Move up the runtime stack *num* levels
use	Print the directory search path
use *dir* ...	Change the directory search path
whatis *routine*	Print the declaration of *routine*
whatis *type*	Print the declaration of the *type* (C only)
whatis *variable*	Print the declaration of the *variable*
when at *line* { *cmd*; ... }	Execute command(s) when at *line* in the current file
when at "*file*":*line* { *cmd*; ... }	Execute command(s) when at *line* in *file*
when in *routine* { *cmd*; ... }	Execute command(s) when enter *routine*
when *cond* { *cmd*; ... }	Execute command(s) when condition true
where [> *filename*]	Print a stack trace of all active routines
where *num*	Print a stack trace of the top *num* active routines
whereis *identifier*	Print the fully qualified name of all symbols matching *identifier*
which *identifier*	Print the full default qualification of *identifier*
/*string*[/]	Search forward for *string* in the current file
//	Search forward for next occurrence
?*string*[?]	Search backward for *string* in the current file
??	Search backward for next occurrence

A.2 Machine Instruction-Level Commands

nexti	Step one machine instruction, treating routines atomically
nexti *count*	Step *count* machine instructions, treating routines atomically
nexti all	Step one machine instruction in each thread, treating routines atomically
nexti all *count*	Step *count* machine instructions in each thread, treating routines atomically
stepi	Single step one machine instruction
stepi *count*	Single step *count* machine instructions
stepi all	Step one source line in each thread, entering any called routines
stepi all *count*	Step *count* source lines in each thread, entering any called routines
stopi at *addr* [**if** *cond*]	Stop execution at location *addr*
stopi *var* [**if** *cond*]	Stop if *var* changes
tracei [**if** *cond*]	Trace execution of all machine instruction
tracei *addr* [**if** *cond*]	Trace execution of machine instruction at *addr*
tracei *var* [**at** *addr*] [**if** *cond*]	Trace changes to *var* at *addr*
regs	Print the value of all scalar and address registers
regs all	display the values of the scalar and address registers for all threads
vregs	Display the values of all vector registers
vregs *index*	Display the values of a single vector register
vregs all	Displaythe values of the vector registers for all threads
vregs all *index*	Display the values of a single vector register for all threads
.,count?mode	Print a memory range according to *mode*
address,count?mode	Print the memory range according to *mode*
&name	Address of *name*

Options for Output Mode:

b	byte in octal (8 bits)
c	byte as character (8 bits)
d	word in decimal (32 bits)
D	long long word in decimal (64 bits)
f	single-precision real (32 bits)
F	double-precision real (64 bits)
g	single-precision real in printf %g format
G	double-precision real in printf %g format
i	machine instruction
o	word in octal (32 bits)
O	long long word in octal (64 bits)
s	null-terminated character string
x	word in hex (32 bits)
X	long long word in hex (32 bits)

A.3 Standard Command Aliases

a	assign
b	stop
c	cont
d	delete
e	edit
f	func
h	help
l	list
n	next
p	print
q	quit
r	run
s	step
t	trace
w	where
st	status
wi	whereis
W	which

A.4 Register Names

Denote registers by entering $r, where r is the register name.

a0, sp	Address registers, Stack Pointer
a1 7=a2	
a3	
a4	
a5	
a6, ap	Argument Pointer
a7, fp	Frame Pointer
s0	Scalar registers
s1	
s2	
s3	
s4	
s5	
s6	
s7	
psw	Processor Status Word
pc	Program Counter

Note: Vector registers cannot be individually referenced; use the *vregs* command.

B

Reporting Problems

B.1 Introduction

The *contact* utility is the recommended way to report software and documentation problems to the Technical Assistance Center (TAC). It is an interactive tool that prompts you for the information necessary to report a problem to the TAC.

You must have a UNIX-to-UNIX Communications Protocol (UUCP) connection to the TAC to use *contact*. A UUCP system allows communication between UNIX systems by either dial-up or hard-wired communication lines. See *uucp*(1) or the entry in *info*(1) (online information system) for more information.

You must know the name and version number of the product involved. If you do not know the version number of the program or utility you are having trouble with, use the *vers* command. The syntax for the command is

vers *filename*

where *filename* is the the full pathname of the program. If you don't know the full pathname of the program, type

which *program*

For more information on these commands, see *vers*(1) and *which*(1) in the *CONVEX UNIX Programmer's Manual,* Part I.

B.2 Information Required to Report a Problem

contact requires the following information:

1. Your name, title, phone number, and corporate name.
2. The name and version of the product involved. Use the *vers* command if you don't know the version number of the program or utility.
3. A short (1 line) summary of the problem.
4. A detailed description of the problem. Include source code and a stack backtrace whenever possible. (See *adb*(1) or *csd*(1) for information on obtaining stack backtraces.) The more information provided, the quicker your problem can be isolated and solved.
5. The priority of the problem. You are shown a list of six levels from which to select.

6. Instructions on how to reproduce the problem, including the command syntax used, any flags invoked, or anything else you attempted to make your program run.

7. Any other comments about the problem or files you wish to submit.

You will have a chance to review your report before you submit it. You can edit the report if you find an error in what you have typed. If you change your mind and don't want to submit the report, you can abort the *contact* session; the file is saved in your home directory in a file named *dead.report*.

The following figure is a sample *contact* session. User input is in bold lettering, and the system response is in constant-width lettering.

Figure B–1: Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.14 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University (RETURN)
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed."  Happily, this is not true! (RETURN)
> (CTRL-D)

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 (RETURN)

Problem report submitted.
%
```

Index

. command 2-22
// command 2-24
/ command 2-24
?? command 2-24
? command 2-24

A

active routines 1-3
address registers 4-12
address registers, a0-a7 A-5
alias command 2-47
analyzing *gprof* output 5-18
argument pointer, ap A-5
array'index' 2-9
array'subrange 2-9
as command 3-3
assemble a program 3-3
assign command 2-20
assigning values 2-20
asterisk, signifying eventpoints 3-3
auto mode of floating-point 2-49

B

basic block'description of 3-2
basic block'suspending execution 3-3
bprof basic operations 5-20
bprof capabilities of 5-19
bprof command format 5-20
bprof command options 5-20
bprof sample output 5-20
bprof sample output'*-l* option 5-24
bprof sample output'*-m* option 5-24
bprof used with *indent*(1) 5-27
breakpoint 2-35
breakpoint commands'*delete* 2-44
breakpoint commands'*stop* 2-37
breakpoint commands'*stopi* 2-39
breakpoint'deleting 2-44
breakpoint'description of 1-3
breakpoints, setting 2-37
breakpoint'show valid lines for 2-7

C

call command 2-32
call stack'description of 1-3
capabilities of *gprof* 5-7
capabilities overview, *csd* 1-2
capabilities overview'*pmd* 6-2
catch command 2-33
catching UNIX signals 2-33
chained mode 4-12
code'executable'shown in listing 2-7
command aliases 2-47
command format'*bprof* 5-20
command format'*gprof* 5-7
command format'*prof* 5-3
command list execution 2-43
command options'*bprof* 5-20
command options'*-f* 4-15
command options'*gprof* 5-7
command options'*prof* 5-3
command'enhancements to 4-12
command'*limit concurrency 4-9*
commands'. 2-22
commands'*//* 2-24
commands'*/* 2-24
commands'*??* 2-24
commands'*?* 2-24
commands'*alias* 2-47
commands'*assign* 2-20
commands'breakpoint'*delete* 2-44
commands'breakpoint'*stop* 2-37
commands'breakpoint'*stopi* 2-39
commands'*call* 2-32
commands'*catch* 2-33
commands'*cont* 2-30
commands'*cont* 4-12
commands'*cregs* 4-11
commands'*delete* 2-44
commands'display'*cregs* 4-11
commands'display'*list* . 2-7
commands'display'*print* 2-8
commands'display'*regs* 4-12
commands'display'*thread* 4-10
commands'display'*threads* 4-10
commands'display'*vregs* 4-13
commands'display'*whereis* 2-12
commands'display'*which* 2-12
commands'*down* 2-45
commands'*dump* 2-19
commands'*edit* 2-51
commands'eventpoint'*stop* 2-37
commands'eventpoint'*stopi* 2-39
commands'eventpoint'*trace* 2-40
commands'eventpoint'*tracei* 2-42
commands'eventpoint'*when* 2-43
commands'executing at whenpoint 1-3
commands'execution'*cont* 2-30, 4-12
commands'execution'*next* 2-28, 4-12
commands'execution'*nexti* 2-29
commands'execution'*rerun* 2-26
commands'execution'*run* 2-25, 4-11
commands'execution'*source* 2-31
commands'execution'*step* 2-27, 4-12
commands'execution'*stepi* 2-29
commands'execution'*STOP* 2-43
commands'execution'*stop* 4-12
commands'execution'*trace* 4-12
commands'execution'*when* 2-43
commands'execution'*where* 2-18, 2-45
commands'*format* 2-50
commands'*fpmode* 2-49
commands'*func* 2-14
commands'*help* 2-4
commands'*ignore* 2-33
commands'*list* 2-6
commands'list'*whereis* 2-12
commands'list'*which* 2-12
commands'miscellaneous'*dump* 2-19
commands'miscellaneous'*edit* 2-51
commands'miscellaneous'*list* 2-6, 2-13
commands'miscellaneous'*sh* 2-51
commands'miscellaneous'*use* 2-50

commands'*mode* 2-49
commands'*mode* 4-12
commands'multithread debugging 4-10
commands'*next* 2-28
commands'*next* 4-12
commands'*nexti* 2-29
commands'*print* 2-8
commands'*quit* 2-4
commands'register'*regs* 2-17
commands'register'*vregs* 2-17
commands'*regs* 2-17
commands'*regs* 4-12
commands'*rerun* 2-26
commands'*return* 2-47
commands'*run* 2-25
commands'*run* 4-11
commands'search'// 2-24
commands'search'/ 2-24
commands'search'?? 2-24
commands'search'? 2-24
commands'*set* 2-19
commands'*set precision* 2-50
commands'*sh* 2-51
commands'*source* 2-31
commands'specifying initial 2-3
commands'stack'*down* 2-45
commands'stack'*return* 2-47
commands'stack'*up* 2-45
commands'*status* 2-36
commands'*step* 2-27
commands'*step* 4-12
commands'*stepi* 2-29
commands'step'*next* 2-28
commands'step'*nexti* 2-29
commands'step'*step* 2-27
commands'step'*stepi* 2-29
commands'*stop* 2-37
commands'*STOP* 2-43
commands'*stop* 4-12
commands'*stopi* 2-39
commands'*thread* 4-10
commands'*threads* 4-10
commands'*trace* 2-40
commands'*trace* 4-12
commands'*tracei* 2-42
commands'tracepoint'*delete* 2-44
commands'tracepoint'*trace* 2-40
commands'*unalias* 2-48
commands'*up* 2-45
commands'*use* 2-50
commands'*vregs* 2-17
commands'*vregs* 4-13
commands'*whatis* 2-13
commands'*when* 2-43
commands'*where* 2-18, 2-45
commands'*whereis* 2-12
commands'*which* 2-12
communication register 4-5
communication register'description of 4-2
communication register'displaying the 4-11
communication register'interaction between 4-7
communication register'lock bit 4-11
complex, description of 4-2
concepts'breakpoint 1-3
concepts'call stack 1-3
concepts'debugging 1-3
concepts'eventpoint 1-3
concepts'tracepoint 1-3
concepts'whenpoint 1-3
concurrency 4-9
cont command 2-30
cont command 4-12
contact, reporting problems B-1
context'current 1-5
context'triplet defining the 1-5
continuing program execution 4-12
CPU, description of 4-2
cregs command 4-11
csd command-line options'*-f* 2-2
csd command-line options'*-I dir* 2-2
csd command-line options'*-r* 2-2
csd'capabilities overview 1-2
csd'command-options'*-f* 4-15
csd'features 1-2
.csdinit file 2-3, 2-20, 2-48
csd'terminating 2-4
current'address'display line at 2-7
current'context 1-5
current'environmental pathname 1-6
current'file 1-5
current'line 1-5
current'routine 1-5
current'thread'changing the 4-10
current'thread'displaying the 4-10

D

debugging'concepts'breakpoint 1-3
debugging'concepts'call stack 1-3
debugging'concepts'environment 1-4
debugging'concepts'overview 1-3
debugging'concepts'scope 1-4
debugging'description of symbolic 1-2
debugging'multithread commands 4-10
debugging'optimized code 3-1
debugging'signals 2-34
debugging'uses of 1-6
debugging'vectorized code 3-1
delete command 2-44
display'commands'*cregs* 4-11
display'commands'*print* 2-8
display'commands'*regs* 4-12
display'commands'*thread* 4-10
display'commands'*threads* 4-10
display'commands'*vregs* 4-13
display'commands'*whereis* 2-12
display'commands'*which* 2-12
displaying'values 2-8
display'registers 4-12, 4-13
distinguishing between nonunique symbols 1-5
down command 2-45

dump command 2-19

E

edit command 2-51
effects of optimization 3-2
environment, definition 1-4
environmental pathname, current 1-6
error reporting B-1
event 2-35
event index 2-35, 2-44
eventpoint 1-3, 2-35, 3-3
eventpoint'commands'*stop* 2-37
eventpoint'commands'*stopi* 2-39
eventpoint'commands'*trace* 2-40
eventpoint'commands'*tracei* 2-42
eventpoint'commands'*when* 2-43
eventpoint'deleting 2-44
executable code'shown in listing 2-7
execution'commands'*cont* 2-30, 4-12
execution'commands'*next* 2-28, 4-12
execution'commands'*nexti* 2-29
execution'commands'*rerun* 2-26
execution'commands'*run* 2-25, 4-11
execution'commands'*source* 2-31
execution'commands'*step* 2-27, 4-12
execution'commands'*stepi* 2-29
execution'commands'*STOP* 2-43
execution'commands'*stop* 4-12
execution'commands'*trace* 4-12
execution'commands'*when* 2-43
execution'commands'*where* 2-18, 2-45
execution'continuing 4-12
execution'mode of 4-12
execution'monitoring program 1-3
execution'simultaneous thread 4-5
execution'suspending 1-7
execution'suspending program 1-3

F

-f csd command-line option 2-2
-f option 4-15
fault'instruction causing a 3-3
file'current 1-5
floating-point mode 2-49
floating-point mode'changing 2-49
floating-point mode'displaying 2-49
floating-point mode'types of'*auto* 2-49
floating-point mode'types of'*ieee* 2-49
floating-point mode'types of'*native* 2-49
floating-point values'assigning 2-49
floating-point values'printing of 2-49
format command 2-50
format'*bprof* 5-20
format'*gprof* 5-7
format'*prof* 5-3
fpmode command 2-49
frame pointer, fp A-5
func command 2-14

G

gprof analyzing output 5-18
gprof basic operations 5-7
gprof capabilities 5-7
gprof command format 5-7
gprof command options 5-7

H

help command 2-4

I

-I dir csd command-line option 2-2
ieee mode of floating-point 2-49
IEEE programs, debugging 2-49
ignore command 2-33
ignoring UNIX signals 2-33
indexing arrays 2-9
initial commands, specifying 2-3
internal representations, translating 2-49
invoking *pmd* 6-2
I/O redirection 2-25

L

learning tool, using debugger as a 1-6
limit concurrency command 4-9
line'current 1-5
list . command 2-7
list command 2-6
list command'asterisk 3-3
list command'show executable code 2-7
list commands'*whereis* 2-12
list commands'*which* 2-12
lock bit 4-11
locus of execution 3-3

M

machine-level stepping 2-29
memory, inspection of 2-22
miscellaneous commands'*dump* 2-19
miscellaneous commands'*edit* 2-51
miscellaneous commands'*list* 2-6
miscellaneous commands'*sh* 2-51
miscellaneous commands'*use* 2-50
miscellaneous commands'*whatis* 2-13
mode command 2-49
mode command 4-12
modes'chained 4-12
modes'sequential 4-12
module name 1-6
monitoring'a multithreaded program 4-15
monitoring'program execution 1-3
multiple threads 4-3
multiple threads'description of 4-2
multiprocessor, description of 4-2
multiprocessor system, overview of 4-5
multithread'commands for debugging 4-10
multithreaded program'monitoring a 4-15
mutual exclusion, description of 4-2

N

native mode of floating-point 2-49
next command 2-28
next command 4-12
nexti command 2-29
notes and limitations 1-7

O

optimization‘combining code 3-2
optimization‘effects of 3-2
optimization‘effects on variables 3-2
optimization‘merging code 3-2
optimization‘moving code 3-2
optimization‘rearranging code 3-2
optimization‘removing code 3-2
optimization‘replicating code 3-2
optimized code‘problems in debugging 3-2
options‘*bprof* 5-20
options‘*csd* 2-2
options‘*gprof* 5-7
options‘*prof* 5-3

P

parallel program 4-5
pc register 3-3
pmd capabilities overview 6-2
pmd‘invoking 6-2
pmd‘optimization level restrictions 6-4
pmd‘restrictions 6-3
pmd‘sample programs 6-4
pmd‘use of 6-2
post-mortem dump utility, *pmd* 6-1
print command 2-8
print command‘indexing 2-9
print command‘subrange 2-9
process‘description of 4-2
prof basic operations 5-2
prof capabilities 5-2
prof command format 5-3
prof options 5-3
prof output, sample 5-3
program counter, pc A-5
program status word, psw A-5
program‘monitoring a multithreaded 4-15
program‘running the 2-25
program‘stopping the 1-7
program‘suspending execution 1-3

Q

quit command 2-4

R

-r csd command-line option 2-2
redirection of I/O 2-25
register communication 4-5
register names A-5
register‘commands‘*cregs* 4-11
register‘commands‘*regs* 2-17
register‘commands‘*vregs* 2-17
register‘communication 4-7
register‘description of communication 4-2
register‘hardware 4-7
register‘machine 4-7
register‘one set per process 4-7
register‘pc 3-3
registers‘address 4-12
registers‘communication 4-11
registers‘communication‘displaying 4-11
registers‘communication‘lock bit 4-11
registers‘display 4-12, 4-13
registers‘scalar 4-12
registers‘vector 4-13
regs command 2-17
regs command 4-12
reporting problems B-1
rerun command 2-26
restrictions on *pmd* 6-3
return command 2-47
routine‘active 1-3
routine‘call stack 1-3
routine‘current 1-5
run command 2-25
run command 4-11
running the program 2-25

S

sample output‘*bprof* 5-20
sample program with *gprof* output 5-8
sample programs run with *pmd* 6-4
sample programs with *prof* output 5-3
scalar registers 4-12
scalar registers, s0-s7 A-5
scheduling mode 2-2
scope, definition 1-4
search commands‘// 2-24
search commands‘/ 2-24
search commands‘?? 2-24
search commands‘? 2-24
searching files 2-24
sequential mode 4-12
set command 2-19
set precision command 2-50
sh command 2-51
signal handling 2-32
signal handling‘examples of 2-34
signal handling‘multiple 2-34
signal handling‘sample program 2-34
signal‘catching UNIX 2-33
signal‘ignoring UNIX 2-33
signals 2-31
signals‘examples of debugging with 2-34
single processor system, overview of 4-3
source command 2-31
stack commands‘*down* 2-45
stack commands‘*return* 2-47
stack commands‘*up* 2-45
stack pointer, sp A-5
standard command aliases A-4
status command 2-36
step all threads 4-12
step command 2-27

step command 4-12
step commands'*next* 2-28
step commands'*nexti* 2-29
step commands'*step* 2-27
step commands'*stepi* 2-29
stepi command 2-29
stop command 2-37
STOP command 2-43
stop command 4-12
stopi command 2-39
stopping the program 1-7
stored files, executing 2-31
sub-complex, description of 4-2
subrange 2-9
suspending program execution 1-3
symbolic debugger, description of 1-2
symbols, distinguishing between nonunique 1-5
synchronization'between threads 4-7

T

terminating *csd* 2-4
terminology'communication register 4-2
terminology'complex" 4-2"
terminology'CPU 4-2
terminology'multiprocessor 4-2
terminology'mutual exclusion 4-2
terminology'process 4-2
terminology'sub-complex 4-2
terminology'thread 4-2
thread command 4-10
thread limit'changing the 4-10
thread'changing the limit 4-10
thread'description of 4-2
threads command 4-10
threads'continue all 4-12
threads'display registers for all 4-12, 4-13
threads'displaying all 4-10
threads'independent 4-5
threads'multiple 4-3
threads'simultaneous execution 4-5
threads'step all 4-12
threads'stop when created or terminated 4-12
threads'trace creation or termination 4-12
thread'synchronizing 4-7
trace command 2-40
trace command 4-12
tracei command 2-42
tracepoint 1-3, 2-35
tracepoint commands'*delete* 2-44
tracepoint commands'*trace* 2-40
tracepoint'deleting 2-44
tracepoints, setting 2-40
transferring types 2-10
trouble reports B-1
type transfer 2-10

U

unalias command 2-48
UNIX signals, catching 2-33
UNIX signals'ignoring 2-33
up command 2-45
use command 2-50
uses of debugging 1-6
using *pmd* 6-2
using the debugger as a learning tool 1-6

V

values'assigning 2-20
values'displaying 2-8
values'unexpected in variables 2-20
variables, unexpected values 2-20
variables'effects of optimization 3-2
vector registers 4-13
vers command B-1
version of software, how to find B-1
vregs command 2-17
vregs command 4-13

W

whatis command 2-13
when command 2-43
when event, deleting 2-44
whenpoint'description of 1-3
where command 2-18, 2-45
whereis command 2-12
which B-1
which command 2-12

Software Documentation

Index Enhancements

So that we can continue to provide better indexing in CONVEX documentation, please keep track of the words or phrases you look up in an index, but don't find. Then, list under which index entry you ultimately found the information you were seeking. You can mail one of these postage-paid forms to the CONVEX Software Documentation Department monthly, or you can submit the information to the Technical Assistance Center in the form of a bug report. You can get more forms by writing to CONVEX at the address below, or by calling us. You can also photocopy this form and mail it back in an envelope. Thank you for helping us to serve you better.

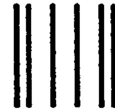
Name:____________________ **Company:**____________________

Phone:____________________ **Date:**____________________

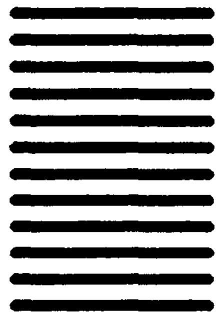
Manual Title/Rev. No.	Looked Up This Word	Found Information Under This Word

CONVEX Computer Corporation — P.O. Box 833851, Richardson, TX 75083-3851 (214)952-0200

(Fold Here First)



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)

CONVEX Consultant User's Guide
Document No. 740-002530-203, V8.0

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are most welcome and will help us continue in our efforts to generate quality customer documentation. (Please list the page number for questions and comments):

____

____

____

____

____

____

____

____

____

____

____

From:

Name________________________________Title________________________________

Company______________________________Date_________________________________

Address and Phone No.____

FOR ADDITIONAL INFORMATION/DOCUMENTATION:

Location	Phone Number
Texas	(214)952-0200
Continental USA	1(800)952-0379
Outside continental U.S.	Contact local office

Direct Mail Orders to:

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

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)