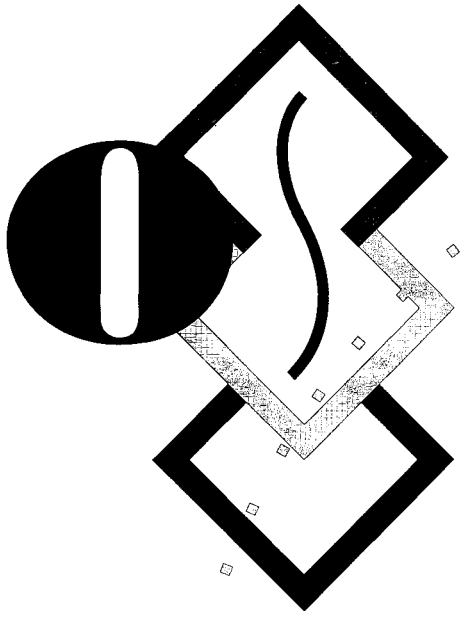


CONVEX

adb (Assembly Language Debugger) User's Guide

Sixth Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

CONVEX adb (Assembly-Language Debugger) User's Guide

Order No. DSW-009

Sixth Edition
March 1994

CONVEX Press
Richardson, Texas
United States of America

CONVEX
adb (Assembly-Language Debugger) User's Guide

Order No. DSW-009

Revision information for

CONVEX
adb (Assembly-Language Debugger)
User's Guide

Edition	Document No.	Description
Sixth	710-025030-000	Divided the information in Chapter 1, "Introduction to adb," into two new chapters—Chapter 1, "Overview," which contains terminology and explains concepts, and Chapter 2, "adb command syntax," which lists all adb commands and their syntax. Added Chapter 4, "Quick start for new users," which contains instructions for performing frequently-used adb tasks. Deleted Chapter 7, "Putting it all together." This information is now included in other chapters. Added reference pages to Chapter 9, "adb command reference."
Fifth	740-002630-203	Released with CONVEX UNIX 7.0, October 1988.
Fourth	740-02630-201	Released with CONVEX UNIX 6.1, October 1987.
Third	740-02030-000	Release with CONVEX UNIX 6.0, November 1986.
Second	740-00630-000	Released with CONVEX UNIX 2.9, September 1985.
First	740-00630-000	Released with CONVEX UNIX 1.0, February 1985. Initial release.

Contents

Using this book xv
Purpose and audience xv
Organization xv
Notational conventions xvii
Command syntax xvii
General conventions xvii
Associated documents xix
Ordering documentation xix
Technical assistance xix
Reader response xx
The contact utility xx
Acknowledgments xx

1 Overview 1
Description 2
Debugging concepts and terminology 3
Register set 3
Breakpoints 6
Call stack 6
Current frame, current position, and dot 6
Symbols 7
Current process 7
Job 8
Thread 8
Current thread 9
Pipes, sockets, and shared memory 9

2 command syntax 11
Overview of adb commands 12
: commands 14
: command descriptions 14
: command syntax 15
$ commands 16
$ command descriptions 16
$ command syntax 18
? commands 20
? command descriptions 20
? command syntax 21
/ commands 22

/ command descriptions 22
/ command syntax 23
= commands 24
= command descriptions 24
= command syntax 25
?, /, and = command syntax 26
) and ! commands 28

3 Invoking adb 29
Invoking the adb debugger 30
Executable file: objfile 30
Core file: corefile 31
Executable and core file modification 31
Input directory 31
Kernel mapping 32
Invoking adb: command syntax 33
Syntax example 33
adb address expressions 35
Specifying actual addresses 36
Specifying symbolic addresses 37
adb arithmetic operators 39
Data display formats 40
Getting help 42
Quitting adb 45

4 Quick start for new users 47
Displaying a stack backtrace 48
Modifying register values 49
Scalar registers 49
Vector registers 49
Address registers 50
Disassembling an object file (or single stepping) 51
Patching a binary 52
Examining a core file 53

5 Basic adb debugging 55
Displaying information 56
Displaying information from an objfile 56
Displaying information from a corfile 57
Displaying instructions 57
Displaying program variables 59
Displaying job-related information 60
Displaying breakpoints 64
Displaying registers 65
Displaying address registers 65

Displaying hardware communication registers 67
Displaying scalar registers 69
Displaying vector registers 70
Displaying internal and global variables 71
Moving around in memory 73
Modifying program data 75
Modifying memory 75
Modifying registers 77
Manipulating breakpoints 80
Setting breakpoints 80
Specifying an ignore count 81
Specifying a breakpoint handler 81
Deleting breakpoints 82
Executing program instructions 84
Starting a process 84
Executing single instructions ("stepping") 86
Continuing program execution 87
Killing the current process 88
Interpreting stack backtraces 89
Executing commands from a file 93

6 Advanced adb use 95
Core files 96
Signal handling 99
Kernel debugging 103
Segment mapping 104
Internal variables 106

7 Multithreaded debugging 107
What is a multithreaded program? 108
Terminology 108
Multiple threads 109
Registers 112
Multithreaded debugging commands 115
Jobs and threads 115
Groups of commands 115
Commands that apply to the current thread 116
Commands that apply to all threads in a job 117
Commands that apply to a job 118
Commands that change or display program data 118
Commands that change or display the `adb` environment 119
Example of multithreaded debugging 120
Example `adb` multithreaded debugging session 125

8 Multiprocess debugging 137
What is a multiprocess program? 138
Components of a ConvexOS process 138
`fork()` system call 140
Multiprocess debugging commands 143
Example of multiprocess debugging 146
Example `adb` multiprocess debugging session 148

9 adb command reference 153
Organization of reference pages 153
Page format 154
List of reference pages 156

A Using `adb`—C program examples 403

B Using `adb`—FORTRAN program example 409

Figures

Figure 1 Allocation of register sets 5
Figure 2 Executing commands from a .adbrc file 32
Figure 3 `adb` online help 43
Figure 4 Displaying information from an object file 56
Figure 5 Displaying information from a core file 57
Figure 6 Displaying instructions with `?i` 57
Figure 7 Displaying 10 instructions with `?i` 58
Figure 8 Displaying 10 instructions with `?ia` 58
Figure 9 Displaying program variables with `j?w` 59
Figure 10 Displaying values 59
Figure 11 Displaying different parts of an array 60
Figure 12 Displaying the status of all active jobs 61
Figure 13 Displaying status of a job with two threads 61
Figure 14 Using the `$?` command 62
Figure 15 Using the `$?` command—multiple threads 62
Figure 16 Displaying the name of a executable file with `$?` .. 63
Figure 17 Displaying breakpoints 64
Figure 18 Setting a new breakpoint 64
Figure 19 Displaying the value of specific registers 66
Figure 20 Displaying the value of all address registers 66
Figure 21 Displaying register values—by number or name ... 66
Figure 22 Hardware register contents, 17th register 68
Figure 23 Displaying a single scalar register 69
Figure 24 Displaying all scalar registers 69
Figure 25 Displaying the first five elements of vector register 70
Figure 26 Command output from `$l` 71
Figure 27 Command listing from `$e` 72
Figure 28 Moving to new location using a logical address 73
Figure 29 Moving to new location using a symbolic address 74
Figure 30 Two ways of specifying the same address 74
Figure 31 Assigning values directly to memory 76
Figure 32 Changing the value of `valid_answer` 77
Figure 33 Debugging the core image 77
Figure 34 Assigning values to address, hardware, scalar registers 79
Figure 35 Assigning a value to a vector register 79
Figure 36 Setting a breakpoint at the `_main` routine 81
Figure 37 Specifying an ignore count 81
Figure 38 Specifying a breakpoint handler 82
Figure 39 Displaying information with `:r` 82
Figure 40 Deleting a single breakpoint using `:d` 83
Figure 41 Deleting all breakpoints using `:D` 83

Figure 42 List of breakpoints currently set 85
Figure 43 Stopping execution at a specific breakpoint 85
Figure 44 Using `:s` to execute an instruction 86
Figure 45 Using `$j` to execute one instruction for two threads 87
Figure 46 Using `$j` to execute one instruction for each thread 87
Figure 47 Using the `$c` command 89
Figure 48 Sample stack backtraces and code comparison 90
Figure 49 Sample source code and assembly code comparison 91
Figure 50 Stack backtrace from currently executing file 92
Figure 51 Looking at a stack backtrace from the core file 92
Figure 52 `$?` command 96
Figure 53 `/i` command 97
Figure 54 Using `$?` to look at the state of a core file 97
Figure 55 Using `$c` to look at a stack backtrace 98
Figure 56 Sample signal handler program in C 100
Figure 57 Using `:r` to continue execution 101
Figure 58 Using `:r` to continue execution 101
Figure 59 Using `:r` after execution resumes 102
Figure 60 Invoking `adb` in kernel mode 103
Figure 61 Using `$m` to produce a map listing 104
Figure 62 Processes in a single-processor system 110
Figure 63 Processes in a multiprocessor system processes ... 111
Figure 64 A multithreaded process and its registers 114
Figure 65 The C main program 120
Figure 66 Hand-coded `sum_in_parallel` assembly-language subprogram 121
Figure 67 The effects of executing instruction `spawn _thread_proc, fp` 123
Figure 68 Example ConvexOS process 139
Figure 69 Effects of process P calling `fork()` 141
Figure 70 A sample C program that uses `fork()` 147
Figure 71 C program with pointer bug 403
Figure 72 Sample `adb` session for core dump program 404
Figure 73 C program to decode tab stops 405
Figure 74 Sample `adb` session for tab conversion 407
Figure 75 FORTRAN program with bug 409
Figure 76 Sample `adb` session for FORTRAN program 410

Tables

Table 1	Allocation of register sets	4
Table 2	`adb` command set	12
Table 3	`:` command descriptions	14
Table 4	Syntax for `:` commands	15
Table 5	`$` command descriptions	16
Table 6	Syntax for `$` commands	18
Table 7	`?` command descriptions—for use with object files	20
Table 8	`/` command descriptions—for use with core files	22
Table 9	`=` command descriptions	24
Table 10	Syntax for `?`, `/`, and `=` commands	26
Table 11	`)` commands	28
Table 12	Common address expressions	35
Table 13	Monadic operators	39
Table 14	Dyadic operators	39
Table 15	Data display format values	40
Table 16	Radix values	41
Table 17	Displaying SP, AP, FP, PC, PSW, and dot	67
Table 18	`adb` internal variables	71
Table 19	Register assignment formats	78
Table 20	Commands that execute program instructions	84
Table 21	Internal variables	106
Table 22	Terminology used in multithreaded programs	108
Table 23	Commands that apply to the current thread	116
Table 24	Commands that apply to all threads in a job	117
Table 25	Commands that apply to a job	118
Table 26	Commands that change or display program data	118
Table 27	Commands that change or display the `adb` environment	119
Table 28	`C`ommands for debugging multiple processors	144
Table 29	`adb` commands that apply to the current job	145

Using this book

Purpose and audience

The *CONVEX adb User's Guide* describes the adb debugging tool used with the ConvexOS operating system. adb is an object-code debugger that does not require any special support from a compiler or loader.

This guide includes:

- Overall adb debugging and instructions for using adb
- A collection of adb command reference pages

Organization

This guide addresses the experienced programmer who is interested in debugging programs at the assembly-language level. This audience also includes those programmers who only need to examine a stack trace from a failed C or FORTRAN program. This guide describes the functions and operations of the adb debugger and includes command formats and symbols.

Specifically, this guide is organized into the following chapters and appendixes:

- Chapter 1 introduces concepts and terminology necessary to use adb, and provides a debugging overview.
- Chapter 2 presents the adb command set and syntax.
- Chapter 3 explains how to invoke adb and use the command line options. The chapter also describes the adb command syntax, expressions, and mathematical operators.
- Chapter 4 is a quick start for new users; it contains instructions for performing common adb tasks.
- Chapter 5 describes how to use adb to debug a program. It presents information in a task-oriented fashion.
- Chapter 6 describes more advanced debugging tasks and techniques.

- Chapter 7 explains how to debug multithreaded code.
- Chapter 8 explains how to debug multiprocess code.
- Chapter 9 is a command reference for `adb`. For each `adb` command, the chapter includes a description for the command, its options, parameters, examples, and a list of related commands, if applicable.
- Appendix A contains sample debugging sessions of C language programs.
- Appendix B contains a sample debugging session FORTRAN language program.

Notational conventions

This section discusses notational conventions used in this book.

Command syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
   ①       ②      ③     ④       ⑤
```

1. `COMMAND` must be typed as it appears.
2. *input_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. *[output_file]* enclosed in brackets indicates an optional file name supplied by the user.

General conventions

In general, the following conventions are used in this guide:

- Italics:
  - Designate user-supplied variables in a command-line example
  - Introduce new and important terms
  - Identify variables in mathematical equations
  - Indicate document titles
- `Constant-width font` designates:
  - System output in screens and examples
  - Command names and options
  - System calls
  - Data structures and types
  - Directives, program statements, display examples, printout examples, and error messages returned
- **`Bold, constant-width font`** designates user input in screens and examples, and must be typed exactly as it appears.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).

- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form `adb`(1), where the name of the man page is followed by its section number enclosed in parentheses.
- The ConvexOS prompt is printed as a percent sign (`%`) and the VMS prompt is printed as a dollar sign (`$`).

Associated documents

Using this software may require information not specific to the tasks described in this document.

For more information on the ConvexOS operating system, you can order these books from CONVEX Computer Corporation:

- *ConvexOS Primer* (DSW-133). This book introduces new users to the ConvexOS operating system.
- *ConvexOS Man Pages for Users* (DSW-331). This book is one of three standard references that contain man pages for the ConvexOS operating system.
- *ConvexOS Man Pages for Programmers* (DSW-332). This book is one of three standard references that contain man pages for the ConvexOS operating system.

Ordering documentation

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

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

Include the order number or the exact title, as listed on the front cover.

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384
- Outside continental U.S., contact local CONVEX office.

The `contact` utility

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

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

To use `contact` requires:

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

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

Acknowledgements

The author wishes to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book.

Overview

1

This chapter describes:

- The CONVEX adb debugger and its basic features
- Concepts and terms used when debugging object code

Description

The CONVEX adb debugger is an object-code debugger. It provides a controlled environment for executing your programs. It enables you to start and stop a program at any point, display and change the values of program variables, and examine and modify the state of the process stack. Because adb debugs code at the assembly-language level, you can use it to debug executable code produced by any CONVEX compiler or assembler.

The adb debugger executes a program under its control, so it always knows the state of the program and the values of all symbols. If the program fails while being debugged, control is returned to adb rather than to the ConvexOS shell.

You can use the adb debugger to:

- Create a process from an executable file
- Interactively debug programs at assembly-language level
- Examine a core file (usually called core) from a failed program
  - by examining the state of the program when it failed
  - by isolating the cause of that failure
- Start, stop, and continue process execution at any point
- Execute a program one instruction at a time
- Examine and modify program variables, registers, and the stack
- Examine the values of program varaibles
- Display and modify the values of machine registers
- Set breakpoints where you want execution to stop
- Step process execution line-by-line
- Display the assembly instructions generated from the program

You can also store several adb commands in a file to be executed immediately when adb is invoked. This lets you bring the program to a known state before you start debugging interactively.

Debugging concepts and terminology

Before you can use the adb debugger effectively, you need to understand the following basic debugging concepts and terminology:

- Register set
- Breakpoint
- Call stack
- Current frame
- Current position
- Dot
- Symbols
- Current process
- Job
- Thread
- Current thread
- Current job
- Communication

Each of these concepts is explained in the following sections.

Register set

The *register set* consists of address, scalar, hardware communication, vector, program counter (PC), processor status word (PSW), and other internal registers. You can examine and modify these registers to affect the program being debugged.

The register set consists of the following register names and types:

- A0-A7—eight 32-bit address registers: a0, a6, and a7 registers are SP (stack pointer), AP (address pointer), and FP (frame pointer), respectively.
- S0-S7—eight 64-bit scalar registers
- H0-H63—sixty-four hardware communication registers (these registers are only supported in the C200 Series machines)
- V0-V7—eight 128-element registers, where each element is 64 bits
- PC—Program counter
- PSW—Program status word

Some of the address registers have special names. The A0, A6, and A7 registerfs are SP (stack pointer), AP (argument pointer), and FP (frame pointer), respectively.

Hardware communication register sets allow communication between multiple threads within a process.

On a single-processor machine, programs can access the following register sets:

- Address
- Scalar
- Vector

On multiple-processor machines, programs can access the following register sets:

- Address
- Scalar
- Vector
- Hardware communication

The two components of multiprocess programs, processes and threads, each access these registers in a slightly different way. Each process within a program has a unique copy of each of these register sets. Each thread within a process has a unique copy of the address, scalar, and vector registers, and shares the hardware communication registers belonging to the process that created the threads.

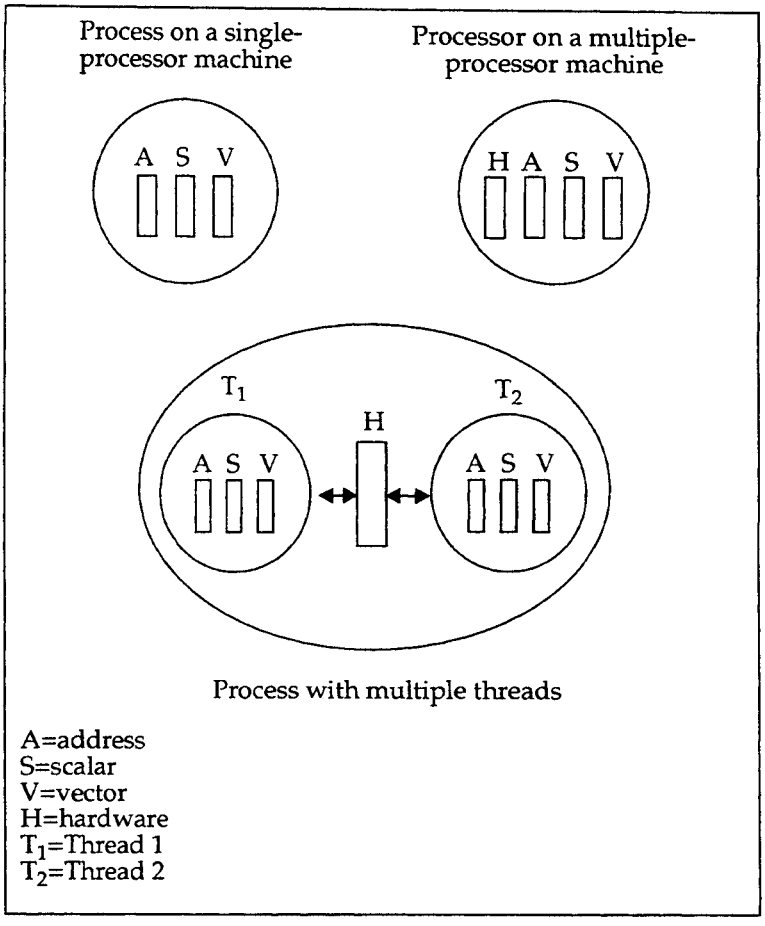
Table 1 displays the relationships between CPUs, processes, threads, and register sets.

Table 1
Allocation of register sets

Register set	Single Processor	Multiple processors	
		process	thread
Address	yes	yes	yes
Scalar	yes	yes	yes
Vector	yes	yes	yes
Communication	no	yes	shared

This scheme of register allocation and use is shown in Figure 1.

Figure 1 Allocation of register sets



Breakpoints

Breakpoints are traps you place at specific locations in your code where you want adb to halt execution of your process. When the location of a breakpoint is reached, execution stops and returns control to you. Using breakpoints, you can look at the state of the program, display program variables, assign new values to variables, and perform other actions to help you see what the program is doing.

While a program is executing, adb does not accept commands or input. When the program is ready to accept input or the debugger is ready to accept commands, it reads what you have typed. Use caution when typing ahead, because typing errors can have a major impact on the state of the program.

For information on how to display breakpoints, refer to "Displaying breakpoints," on page 64. For information on how to set breakpoints, refer to "Setting breakpoints," on page 80. For information on how to delete breakpoints, refer to "Deleting breakpoints," on page 82.

Call stack

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

When adb stops a program at a breakpoint or after executing a single instruction, the subroutine that contains the point where the program execution stops is at the top of the call stack.

Current frame, current position, and dot

When a program stops, adb initializes three positions: the *current frame*, the *current position*, and *dot*.

The *current frame* is the contents of the general register set: *A0* through A7, S0 through S7, PC, and PSW. It represents the routine in which you are currently located and it is the top entry on the call stack.

The *current position* is the location pointed to by the PC register, and it represents the next instruction to be executed. The current position is located in the current frame.

dot is a virtual pointer that is used to move around memory without changing the current position. When you display instructions, for example, *dot* is moved to the address of the instruction, but the current position remains unchanged.

Symbols

A *symbol* is a logical name that represents a value in memory. Symbolic names can only be used in `adb` for programs that contain a symbol table. All executable programs have symbol tables unless specifically removed by the `-s` option of the loader (`ld`) or by the ConvexOS `strip` command. Symbolic names in `adb` can represent program routines and global variables. Local variables and parameters passed to a routine are placed on the stack and can only be accessed by their addresses.

The value of a symbol is taken from the symbol table in the executable file.

Symbols provide an easy way to access major parts of a program. For example, you can identify the start of the main program with its name (`main`) instead of its machine address.

`adb` recognizes two types of symbols: external and internal. External symbols are part of the executable program and its libraries. Internal symbols are those used internally by `adb`; these symbols are not part of the executable program. External symbols are differentiated from internal symbols by underscores. For C language programs, external symbols are prefixed with an underscore (`_sub1`); symbols in FORTRAN must begin and end with an underscore (`_sub1_`). `adb` automatically prefixes an underscore to a symbol if it is omitted when the symbol does not appear in the symbol table as entered. If an external symbol has the same name as an internal or hidden variable, you must prefix the underscore to access the symbol.

Current process

A process is a collection of instruction streams within a single logical address space. The *current process* is the program currently being executed and usually corresponds to the executable file specified on the `adb` command line.

Job

A *job* is a term that refers to a process, along with its associated debugging information. A job includes the current process, its threads, its breakpoints, and other information used by adb to debug the program.

The meaning of the current job changes slightly, depending on what type of program you are debugging. For a single-process, single-thread program, the current job is the only job available (the main program). It coincides with the current process and the current thread. Any commands that operate on threads or processes operate on the current job.

When you are debugging a single-process, multithreaded program, there is still only one job, but the job has separate execution streams (threads) under its control. At any given time, precisely one thread is current. Hence, there is a current job (the main program) and a current thread, that is a subset of the current job. In this scenario, the current job and current thread are never the same. Commands that operate on threads operate on the current thread. You can use adb commands to monitor, change, and control the current thread. Any command that operates on processes operates on the current job (the main program).

When you are debugging a program with multiple processes, the current job refers to any active job, including the main program (job 0). Commands that operate on specific jobs operate only on the current job and do not affect any part of the executable image outside the active process. As with multithreaded debugging, you can monitor and control individual processes and the threads within a process.

Thread

A *thread* is a stream of execution, or sequence of instructions, that is fetched and executed by a single CPU. A process consists of one or more threads, each of which can execute on a different CPU. Memory files, signals, and other process attributes are generally shared amoung threads in a given process. This enables the threads to join together in performing a particular task. Each thread has its own call stack and machine registers.

When you debug a program using adb, every job has a corresponding thread. In nonparallel programs, the job is the same as the thread. When a job uses multiple threads to achieve parallel execution, one job can have several threads, each executing simultaneously. When debugging such a program, you have access only to one thread at a time, and that thread is called the current thread. Non-current threads can execute simultaneously with the current thread. adb offers many commands to support thread-level debugging.

When a program creates multiple processes, adb treats each process as a separate job. Each job can have one or more threads associated with it. Even though there are multiple jobs (each of which has threads), there is still only one current thread. The current thread is one of the threads in the current job.

Current thread

The *current thread* is the thread currently being monitored.

Pipes, sockets, and shared memory

Portions of code that run in parallel must be able to communicate with each other. The most common ways to allow communication are to use *pipes*, *sockets*, and *shared memory*. Code that does not run in parallel does not use this type of communication, because, at any given time, the program has access to all data available to the program. After any instruction is executed, the exact state of the program can be predicted.

When you start running parts of a program independently of each other, however, the exact state of a program cannot be reliably predicted after every instruction. One thread or process may be working on data needed by another thread or process, or a thread or process may try to access common data simultaneously, which may produce artificial or questionable results. Different methods have been devised to control communication between parts of a program, depending on whether the program uses multiple threads or multiple processes.

When a parallel program uses multiple threads, the threads communicate with each other by using special high-speed hardware communication registers. While each thread uses its own scalar, address, and vector registers, it shares the hardware communication registers with other threads executing in parallel. Data is transferred between threads by writing and reading it to and from the hardware registers. To prevent two

threads from accessing data stored in the same register at the same time, each hardware communication register is equipped with a lock bit to guarantee mutually exclusive access to the register. You can monitor or modify the data exchanged between threads while debugging a program by manipulating the hardware communication registers with adb commands.

When a parallel program uses multiple processes, communication between processes must be handled through a more indirect medium, because, unlike threads, processes do not share registers. The tools you use to allow communication between processes are pipes, sockets, and shared memory. Pipes are connections between the standard output of one program and the standard input of another program. Sockets are communication endpoints. To establish communication between processes, data is written to a file using a pipe and socket, then, when another process needs the data, it reads the file. Shared memory is an area of memory accessible to all processes running in a program. The adb commands you can use to access these data structures are introduced in Table 28, "Commands for debugging multiple processors," on page 144.

command syntax 2

This chapter describes the adb command set and provides:

- Tables of all commands and their descriptions
- Tables that show syntax for each command

Overview of adb commands

In general, requests to adb are of the form

(adb)[*prefix argument*]command[*parameter*][;]

where

(adb)	Is the prompt.
prefix argument	Can be [*address*] or [,*count*], or both, [*address*][,*count*]. If *address* is present, *dot* is set to *address*. Initially, *dot* is set to 0. For most commands, *count* specifies how many times the command will be executed. The default *count* is 1.
command	A verb and a modifier.
parameter	One or more parameters listed in the command syntax tables on the following pages in this chapter.
[;]	Separates consecutive commands on a single command line.

For information on how to invoke adb, refer to Chapter 3, "Invoking adb," on page 29.

adb commands consist of a verb followed by a modifier or list of modifiers. Verbs in the adb command set can be divided into seven broad categories, as shown in Table 2.

Table 2
adb command set

Category	Description
:	Manage process execution
$	Display information about the program
?	Display and modify memory locations in *objfile*
/	Display and modify memory locations in *corefile*
=	Display and modify memory locations at *address*
)	Execute an extended command
!	Use a regular shell command within adb

For example, the `:r` command (where `:` is the verb and `r` is the modifier) starts program execution, and the `$b` command (where `$` is the verb and `b` is the modifier) displays currently set breakpoints. The `_main/wu` command displays the value at the memory location in the core file pointed to by the symbolic name `_main`. The `wu` modifiers specify the format of the display.

The sections on the following pages describe each command (verb and modifier) with its decription and syntax.

: commands

In general, commands that begin with a : manage pocess execution.

: command descriptions

Table 3 lists : commands and their descriptions.

Table 3
: command descriptions

Command	Description
`:b`	Set breakpoint
`:C`	Continue execution for all threads in current job
`:c`	Continue program execution of current thread
`:D`	Delete all currently set breakpoints
`:d`	Delete breakpoint
`:e`	Change the exit disposition of current job to RELEASE
`:f`	Bring a job or thread to foreground
`:i`	Toggle inherit mode
`:k`	Kill the current job
`:m`	Toggle scheduling mode between DYNAMIC and FIXED
`:r`	Run *objfile* as a process
`:S`	Execute one instruction for each thread in current job
`:s`	Execute a single instruction in current thread

: command syntax

Table 4 shows syntax for : commands. To use the chart, start at the (adb) prompt at the left, and read left to right, selecting one item from each column. If any cell is empty, the syntax for that command does not require an item from that column.

Table 4
Syntax for : commands

Prompt	Prefix arguments	Verb	Modifier	Parameters
(adb)	[*address*][,*count*]	:	b	*command*
	[*address*][,*count*]	:	c	[*signal*]
	[*address*][,*count*]	:	C	[*signal*]
	[*address*]	:	d	
		:	D	
		:	e	
	[*job*][,*thread*]	:	f	
		:	i	
		:	k	
		:	m	
	[*address*][,*count*]	:	r	[*program-args*]
	[*address*][,*count*]	:	s	[*signal*]
	[*address*][,*count*]	:	S	[*signal*]

Consider this example and explanation:

```
(adb)  subl_,10:C0t14
  ①    ②     ③④⑤   ⑥
```

① `(adb)` is the prompt

② `subl_` is an *address*

③ `,10` is *count*, which specifies the number of times to execute command.

④ `:` is the command verb

⑤ `C` is the command modifier, which in this case continues execution for all threads in the current job.

⑥ `0t14` is the signal that is sent to the process as execution resumes.

Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

$ commands

Commands that begin with a $ are generally commands that affect data display.

$ command descriptions

Table 5 lists $ commands.

Table 5
$ command descriptions

Command	Description
`$<`*f*	Read and execute `adb` commands from file *f*.
`$<<`*f*	Read and execute `adb` commands from file *f*, then return to original file
`$>`*f*	Write `adb` output to file *f*
`$?`	Display PID and signals and the general registers
`$a=`	Modify value of an address register
`$a?`	Display the contents of an address register
`$b`	Display currently set breakpoints
`$c`	Display C stack backtrace
`$e`	Display names and values of all external variables
`$f[`*mode*`]`	Display or set the floating-point format
`$g`	Get new symbol or core file names
`$h=`	Assign value to a hardware communication register
`$h?`	Display contents of a hardware communication register
`$i`	Display names and values of all nonzero internal variables
`$j`	Display status of all jobs
`$k`	Change current kernel memory mapping (kernel debugging only)

Table 5 (continued)
$ command descriptions

Command	Description
`$l`	Set limit for symbol matches
`$m`	Display address maps
`$n`	Display or set maximum number of processors allocated for a process
`$o`	Toggle operating modes between chained and sequential
`$q`	Quit `adb`
`$R`	Display registers for all threads in current job
`$r[`*w*`]`	Display contents of general registers
`$s=`	Modify a scalar register
`$s?`	Display contents of a scalar register
`$t[`*symbol*`]`	Search for a symbol in the current job's symbol table
`$v=`	Modify a vector register
`$v?`	Display contents of a vector register
`$X`	Set default output radix to *address* and report the new value
`$x`	Set default input radix to *address* and report the new value

Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

$ command syntax

Table 6 shows syntax for $ commands. To use the chart, start at the (adb) prompt at the left, and read left to right, selecting one item from each column. If any cell is empty, the syntax for that command does not require an item from that column.

Table 6
Syntax for $ commands

Prompt	Prefix arguments	Verb	Modifer	Parameters
(adb)		$	<	*filename*
		$	<<	*filename*
		$	>	*filename*
		$	?	
		$	a[*reg*]=	*format value*
		$	a[*reg*]?	*format value*
		$	b	
	[*address*][, *count*]	$	c	
		$	e	
		$	f	[*mode*]
		$	g	
		$	h[*reg*]=	*format value*
		$	h[*reg*]?	*format value*
		$	i	
	[*job*]	$	j	
	[*address*][, *thread*]	$	k	
	[*limit*]	$	l	
		$	m	
	[*count*]	$	n	
		$	o	
		$	q	
		$	r	[*w*]
		$	R	[*w*]
		$	s[*reg*]=	*format value*
		$	s[*reg*]?	*format* [*radix*]
		$	t	[*symbol*]
		$	v[*reg*][:*element*]=	*format value*
		$	v[*reg*][:*starting-location*[*count*]?	*format* [*radix*]
	radix	$	x	
	radix	$	X	

Consider this example and explanation:

```
(adb) 8$n
  ①   ②③④
```

① `(adb)` is the prompt.

② `8` is *count*, which specifies the maximum number of processes that will be allocated to a process.

③ `$` is the command verb.

④ `n` is the command modifier, which in this case displays or sets the maximum number of processors allocated to *count*.

Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

? commands

When you use the ? command verb, locations starting at *address* in an object file are displayed or modified, according to which command modifier you use. *dot* is incremented by the sum of the increments for each modifier.

? command descriptions

The ? command displays and modifies the executable image, starting at *address*. When you use `adb` to modify an executable file (usually a file that ends with a .o), you must specify the `-w` option on the `adb` command line. These commands are listed in Table 7.

Table 7
? command descriptions—for use with object files

Command	Description
`?`	Display data from *objfile*
`?=`	Write information into *objfile* memory
`?a`	Display value of *dot* in symbolic form
`?b`	Display a byte specified by *radix*
`?c`	Display addressed character
`?C`	Display addressed character using ^X and ^?
`?f`	Display 32-bit value as floating-point number
`?F`	Display double precision floating point
`?g`	Search for a pattern in the *objfile*
`?h`	Display a halfword specified by *radix*
`?l`	Display a longword specified by *radix*
`?m`	Modify segment map parameters in the *objfile*
`?n`	Display *newline*
`?P`	Display addressed value in symbolic form
`?r`	Display a space
`?s`	Display addressed characters until 0 is reached
`?S`	Display string using ^X escape convention

Table 7 (continued)
? command descriptions—for use with object files

Command	Description
`?t`	Tabs to next tab spot; preceded by integer
`?w`	Display a word specified by *radix*
`?"..."`	Display enclosed string
`?^`	*dot* is incremented by current increment
`?+`	*dot* is incremented by 1; nothing is displayed
`?-`	*dot* is decremented by1; nothing is displayed
`?`*newline*	Repeat previous command with *count* of 1

? command syntax

Table 10, "Syntax for ?, /, and = commands," on page 26, illustrates command syntax for all destination commands. Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

/ commands

When you use the / command verb, locations starting at *address* in a core file are displayed or modified, according to which command modifier you use. *dot* is incremented by the sum of the increments for each modifier.

/ command descriptions

The / command displays and modifies portions of the coredump image (usually a file called *corfile*), starting at *address*. These commands are listed in Table 8.

Table 8
/ command descriptions—for use with core files

Command	Description
/	Display data from *corefile*
/=	Write information into *corefile* memory
/a	Display value of *dot* in symbolic form
/b	Display a byte specified by *radix*
/c	Display addressed character
/C	Display addressed character using ^X and ^/
/f	Display 32-bit value as floating-point number
/F	Display double precision floating point
/g	Search for a pattern in the *corefile*
/h	Display a halfword specified by *radix*
/l	Display a longword specified by *radix*
/m	Modify segment map parameters in the *corefile*
/n	Display *newline*
/P	Display addressed value in symbolic form
/r	Display a space
/s	Display addressed characters until 0 is reached
/S	Display string using ^X escape convention
/t	Tabs to next tab spot. Preceded by integer.

Table 8 (continued)
/ command descriptions—for use with core files

Command	Description
/w	Display a word specified by *radix*
/"..."	Display enclosed string
/^	*dot* is incremented by current increment.
/+	*dot* is incremented by 1; nothing is displayed.
/-	*dot* is decremented by1; nothing is displayed.
/*newline*	Repeat previous command with *count* of 1.

/ command syntax

Table 10, "Syntax for ?, /, and = commands," on page 26, illustrates command syntax for all destination commands. Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

= commands

When you use the = command verb, locations starting at *address* itself are displayed or modified, according to which command modifier you use. *dot* is incremented by the sum of the increments for each modifier.

= command descriptions

= commands display and modify the value of *address* itself. These commands are listed in Table 9.

Table 9
= command descriptions

Command	Description
=a	Display value of *dot* in symbolic form
=b	Display a byte specified by *radix*
=c	Display addressed charcter
=C	Display addressed character using ^X
=f	Display 32-bit value as floating-point number
=F	Display double precision floating point
=i	Display machine instructions
=h	Display a halfword specified by *radix*
=l	Display a longword specified by *radix*
=n	Display *newline*
=P	Display addressed value in symbolic form
=s	Display address character until 0 is reached
=S	Display string using ^X
=w	Display a word specified by *radix*
=Y	Display four bytes in date format
=^	*dot* is incremented by current increment.
=+	*dot* is incremented by 1; nothing is displayed
=-	*dot* is decremented by 1; nothing is displayed

= command syntax

Table 10 illustrates command syntax for all destination commands. Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

?, /, and = command syntax

Table 10 shows syntax for ?, /, and = command verbs.

Table 10
Syntax for ?, /, and = commands

Prompt	Prefix arguments	Verb	Modifier	Parameters
(adb)	[*address*][,*count*]	[?,/,=]		*format* [*radix*]
	[*address*]	[?,/,=]	=	*format value* [...]
		[?,/,=]	b	*radix*
		[?,/,=]	h	*radix*
		[?,/,=]	w	*radix*
		[?,/,=]	l	*radix*
		[?,/,=]	f	
		[?,/,=]	F	
		[?,/,=]	c	
		[?,/,=]	C	
		[?,/,=]	s	
		[?,/,=]	S	
		[?,/,=]	Y	
		[?,/,=]	i	
		[?,/,=]	a	
		[?,/,=]	p	
		[?,/,=]	t	
		[?,/,=]	r	
		[?,/,=]	n	
		[?,/,=]	"..."	
		[?,/,=]	^	
		[?,/,=]	+	
		[?,/,=]	-	
		[?,/,=]	newline	
		[?,/,=]		
		[?/]	g	*format value mask*
			=	*format value*
			m	*b e f* [?/]

Consider this example and explanation:

```
(adb) ,5/g
  ①   ②③④
```

① `(adb)` is the prompt

② `,5` is *count*, which specifies the number of times to execute `command`.

③ `/` is the verb

④ `g` is the modifier

Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

) and ! commands

) commands are used for extended commands, which are listed in Table 11.

Table 11
) commands

Command	Description
)comment	Make a non-executable remark
)help	Display the adb help file
)static	Toggle the static symbol—ON and OFF
)status	Display status of various adb modes

The only) command with a syntax option is)comment, which is used in the following manner:

(adb)) [insert your comment here]

The ! command is used to execute common ConvexOS shell commands from within adb. A shell is created to execute the rest of the line following the !. If the SHELL environment variable is not set, /bin/sh is used.

Refer to Chapter 9, "adb command reference," on page 153 for detailed syntax information and examples for each command.

Invoking adb

3

This chapter explains how to use the CONVEX adb debugger. Topics discussed in this chapter include:

- Invoking the adb debugger
- adb command syntax
- adb address expressions
- adb arithmetic operators
- Data display formats
- Getting help
- Terminating the adb debugger

Invoking the adb debugger

Invoke the adb debugger by using the following command at the ConvexOS shell prompt:

```
% adb [-w] [-k] [-I directory] [objfile [corefile]]
```

where

`-w`	Opens (and creates, if necessary) *objfile* and *corefile* for reading and writing so that you can modify files using adb.
`-k`	Specifies that ConvexOS kernel memory mapping is to be used. You must use this option when *corefile* is a ConvexOS crashdump or /dev/mem.
`-I` *directory*	Specifies the directory where files can be read when using the $< or $<< commands if the files are not located in the current directory. If you don't specify this option, adb looks only in the current directory for the files.
objfile	Is a ConvexOS executable file. The default file is a.out.
corefile	Is the name of a core-image file produced by the operating system when a job terminates abnormally. The default file is core.

Details of the command syntax are described in the following sections.

Executable file: *objfile*

An *executable* file is one whose executable image has been created by the CONVEX loader (ld). The file can be generated from the fc, vc, cc, or ada compilers. The executable file can also be generated using an assembly-language program. If you don't specify an *objfile*, adb searches for the file a.out in the current directory.

Occasionally, you may want to execute adb without an executable image when an a.out file exists in the current directory. To force adb to ignore the a.out file, use a hyphen as the executable file name, as in

```
adb -
```

This command line starts adb without loading an executable image.

Core file: *corefile*

The *corefile* is a file created by the ConvexOS system when an executable program aborts. The core file contains the state of the program and its data when it failed. By debugging the core file, you can see exactly what caused the program to abort and dump core. If you don't specify a core file, adb looks for the file core in the current directory.

Occasionally, you may want to execute adb but ignore core in the current directory. To force adb to ignore the core file, use a hyphen as the core file name, as follows:

```
% adb a.out -
```

This command line starts adb without loading a core file. In this case, you must name the *objfile* even though it is the default name. If you do not, adb assumes the hyphen replaces the *objfile* instead of the *corefile*.

You can also invoke adb with a core file only. To do this, use a hyphen as the name of the executable file, as in

```
% adb - core
```

This command line starts adb without loading the executable image. All commands now apply to the core file. If you do not load the corresponding executable file, you cannot use symbolic names when debugging the core file.

Executable and core file modification

Normally you cannot change files that contain the executable program or the core image while using adb. When debugging, you are only modifying the representations in memory. If you need to modify these files (when doing a patch, for instance), you can use adb to change the images. To do this, you must specify the -w option on the adb command line. For example:

```
% adb -w example core
```

This command invokes adb with an executable file and a core file and enables you to modify the files.

Input directory

By default, adb reads commands interactively from the adb command line. However, you may store a series of adb commands in a file and instruct adb to read and execute the commands from the file.

adb looks in your home directory for a .adbrc file. If it finds one, it executes the adb commands contained in it before displaying the first prompt. You can use this file to store commands that you use every time you debug a program (setting a breakpoint at _sub1, for example). A sample .adbrc file might contain the commands:

```
_sub1:b

:r
```

The commands in the above .adbrc file execute every time you invoke adb. When the adb prompt appears, the program will already be stopped at _sub1 as shown in Figure 2.

Figure 2 Executing commands from a .adbrc file

```
% adb a.out
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
job 0: running
job 0/0: breakpoint _sub1:    sub.w #0,a0
(adb)
```

The -I option tells adb in which directory the (.adbrc) files are located, so you won't need to specify a full path name when you execute one of these commands. adb looks for the files in the current directory first; if it can't find them there, it looks in the directory /usr/lib/adb. Using this option, you can force adb to look in other directories for the files instead of looking in /usr/lib/adb.

Two commands affected by this option are the $< and $<< commands.

Kernel mapping

The -k option changes the internal mapping to let you debug the ConvexOS kernel while it is running. It also allows you to debug from kernel core images generated by the crashdump utility (on the SPU).

Invoking adb: command syntax

The general form for the adb command (at the (adb) prompt) is

[*address*] [, *count*] [command] [*parameter*][*format*][*radix*][;]

where

[*address*]	Identifies the starting location of the code to be affected by the command. The *address* value is represented by a valid adb expression, as isted in Table 12 '<$paratext>" on page 35. For a few selected commands, the *address* parameter is interpreted as the job specification.
[, *count*]	Specifies the number of times to execute *command*. The *count* value is represented by a valid adb expression. If unspecified, *count* defaults to 1. For a few selected commands, the *count* parameter is interpreted as a thread specification. A repetition count is valid in the command, e.g., (adb) _sub1,5?4i.
[command]	A verb and a modifier.
[*parameter*]	One or more parameters.
[*format*]	A letter that defines the format (size) of the data.
[*radix*]	A letter that defines the number base in which to display the data.
[;]	Separates consecutive commands on a single command line.

Multithreaded debugging techniques are described in Chapter 5; they are slightly different and more complicated than those used to debug processes running on machines with a single CPU. The job and thread portions of the command syntax are explained in that chapter. Multiprocessor debugging techniques are described in Chapter 6.

Syntax example

To illustrate an example of adb command syntax, consider the ?i command, which lists the assembly instruction at the specified address of the executable file. To list five assembly instructions starting with the subroutine _sub1, use the command:

```
(adb) _sub1,5?i
```

In this command, `_subl` is the address for the command, 5 is the number of times to execute the command, and the `?i` is the command to execute. The output from the command is similar to the following:

```
_subl:    sub.w       #24,a0
          ld.w        #0,s0
          st.w        s0,_stop_program
          mov         a0,a6
          pshea       0
```

The starting address for the `?i` command is referenced by the symbolic name `_subl`. The next section provides more information about specifying addresses for `adb` commands.

adb address expressions

The current position (called *dot*) always represents the current memory address. Unless given another address, adb executes commands starting at dot. When you specify an address, dot is set to that location and the command, if any, is executed.

Addresses can consist of decimal, octal, and hexadecimal integers alone or combined with one or more arithmetic operations to change the location. Addresses an also be represented symbolically if the executable file contains a symbol table.

Some of the more common address expressions are listed in Table 12.

Table 12
Common address expressions

Address	Description
.	Value of *dot*
+	Value of *dot* incremented by current increment
^	Value of *dot* decremented by current increment
"	Last address typed
integer	adb interprets integers based on the number's prefix. Integer prefixes include: 0o or 0O (zero o) Indicates an octal integer 0t or 0T Indicates a decimal integer 0x or 0X Indicates a hexadecimal integer If you do not specify a prefix, the default radix determines how the integer is interpreted. The default radix is hexadecimal; use the $x command to change the radix.
'cccc'	The ASCII value of up to 4 characters. A backslash ("\") may be used to escape a single quote ("'").
< *name*>	Value of *name*, which is either a variable name or a register name. To see which registers are used by the program, use the $r command.
symbol	Sequence of alphanumeric and underscore (_) characters. The *symbol* may not begin with a digit. If the symbol is an external symbol, adb automatically prefixes an underscore.

Table 12 (continued)
Common address expressions

Address	Description
_symbol	True name of an external symbol in a C program. If the *symbol* is unique, you need not use the underscore as a prefix because `adb` adds it automatically. If the symbol matches an internal symbol, you must prefix the underscore so that it is interpreted as an external symbol.
*_symbol_*	True name of an external symbol in a FORTRAN program. The prefixed underscore is added automatically by `adb`; you must, however, append the underscore to prevent the symbol from being interpreted as an internal or external C symbol.
(*exp*)	Value of the expression, *exp*, which is a combination of one or more of the above expressions and arithmetic operators.

The *integer*, *<name*, and *exp* expressions may also be used to specify values. For example, 0t10 can represent the numerical value 10. Similarly, the *exp* expression (0x9+0x7) can represent the numerical value 10 (expressed in hexadecimal format).

Address expressions fall into one of two categories: actual addresses and symbolic addresses. The following sections describe how to specify these addresses.

Specifying actual addresses

When you specify an actual address, you move dot to the location identified by a number representing the machine address. The number can be a named integer or it can be a calculated number.

The command

```
(adb) 0x80015000?i
```

lists the instruction stored at hexadecimal address 80015000 in the object file. The `0x` indicates that the rest of the number is to be interpreted as a hexadecimal number. This is not necessary unless you change the default input radix, because hexadecimal is the standard default. The 0 prefix is required, however, if the address begins with a hexadecimal letter (a-f).

You can also specify an actual address by calculating a number, usually by adding or subtracting a number from a known address. For example, the command

```
(adb) 0x80015000+0t17?i
```

displays the instruction stored at the address 80015011. The more common method of moving dot relative to the current address is to use a command similar to the following:

```
(adb).-0xa?i
```

This command moves dot back 10 bytes from the current address and displays the instruction at that location.

Specifying symbolic addresses

Another method of specifying an address is to use a symbolic reference. When an executable file is created, a symbol table is added to the end of the file. The symbol table contains a map of the program symbols (variables, routines, functions, and subprograms) with their respective addresses. You can move dot to the beginning of a routine by providing the routine name, rather than its actual machine address.

For example, the command

```
(adb) _sub1?i
```

moves *dot* to the address corresponding to the `_sub1` symbol in a C language program and displays the instruction.

To use a symbolic address in a FORTRAN program, you must both prefix and append an underscore to the symbol name. For example, the command

```
(adb) _GET_RESPONSE_?i
```

moves *dot* to the address corresponding to the FORTRAN subprogram.

The underscores help `adb` determine what kind of symbol to expect. You can have symbol names in assembly-language, C, and FORTRAN programs, each of which treats symbol names differently. Symbols in an assembly-language program are stored in the symbol table exactly as used in the program. Symbols used in a C language program have underscores prepended to them, while symbols used in FORTRAN programs have underscores both prepended and appended to the name of the symbol.

For example, a symbol named "is_done" is stored as "is_done" if it appears in an assembly-language program, as "_is_done" if it appears in a C language program, and as "_is_done_" if it appears in a FORTRAN program.

adb knows that the C compilers prefix an underscore to symbol names. When you specify a symbol name, adb first looks for the symbol as you entered it. If it doesn't find the symbol, adb prefixes an underscore and searches the symbol table again. Only when both searches fail is an error message displayed. adb doesn't automatically append an underscore to search for a FORTRAN symbol. You must append the underscore to FORTRAN symbols; if missing, adb automatically prefixes an underscore to symbols that end with an underscore.

If you know that a symbol is unique in a C or assembly-language program, you need not use the underscore prefix because adb adds it automatically. However, it is good practice to type the underscore at the beginning of every symbol. Problems may occur when a global symbol appears in the source of a C program in an assembly-language routine linked to the C routine. The two symbols are distinct, but if you forget to use an underscore in this case, the assembly-language symbol is referenced, rather than the C symbol.

adb arithmetic operators

The adb debugger lets you perform arithmetic operations with expressions. For example, you can negate the value of a register, add or subtract two variables, and perform bit manipulations. Spaces may not appear in adb arithmetic expressions.Two types of operators are supported: monadic and dyadic. Monadic operators are those that operate on a single operand (expression); negation is an example of a monadic operator. Dyadic operators are those that operate on two operands (expressions); multiplication is an example of a dyadic operator.

Table 13 lists monadic operators; Table 14 lists dyadic operators.

Table 13
Monadic operators

Monadic operator	Description
* *exp*	Contents of the location addressed by *exp* in the core file
@ *exp*	Contents of the location addressed by *exp* in the object file
- *exp*	Negate the value of *exp*
~ *exp*	Perform a bitwise complement of *exp*
# *exp*	Perform a logical negation of *exp*

Table 14
Dyadic operators

Dyadic operator	Description
e1+e2	Add two integer expressions
e1-e2	Subtract two integer expressions
*e1*e2*	Multiply two integer expressions
e1%e2	Divide two integer expressions (result is an integer)
e1 & e2	Perform a logical AND on two expressions
e1 | e2	Perform a logical OR on two expressions
e1 #e2	Round *e1* up to the next multiple of *e2*

Data display formats

The format portion of the command syntax specifies the manner in which the information is displayed. Data display formats are listed in Table 15.

Table 15
Data display format values

Format	Value
`b`	Display byte of data; optional radix
`h`	Display halfword (16 bits) of data; optional radix
`w`	Display word (32 bits) of data; optional radix
`l`	Display longword (64 bits) of data; optional radix
`f`	Display a 32-bit single-precision value as a floating-point number (same as using `wf`)
`F`	Display as a 64-bit double-precision floating-point number (same as using `lf`)
`c`	Display the data as a character
`C`	Display the data as a character, using an escape convention where control characters are indicated by the form `^x` and delete character is shown as `^?`
`s`	Display the data as a character string (terminated by a zero character)
`S`	Display the data as a string using an escape convention (see C)
`Y`	Display 4 bytes of data in date format
`i`	Display the data as machine instructions
`a`	Display the value of *dot* in symbolic form
`P`	Display the addressed value in symbolic form using the same rules described in modifier a
`t`	Move to the next tab spot
`r`	Display a space
`n`	Display a newline
string	Display the enclosed *string* (must be enclosed by double quotes)

The *radix* portion of the command syntax specifies the number base for the display. A *radix* is valid only for the `b`, `h`, `w`, and `l` modifiers. *radix* may assume any of the values listed in Table 16.

Table 16
Radix values

Radix	Description
x	Hexadecimal (default)
t	Signed decimal
u	Unsigned decimal
q	Signed octal
o	Unsigned octal
f	Floating point (valid only for words and longwords)

To see the value of the variable *total_frames* (declared as an integer in a C program), you can use one of these commands:

```
(adb) _total_frames/w
(adb) _total_frames?wo
(adb) _total_frames?wt
```

These display the value of the variable in hexadecimal (?w), octal (?wo), and decimal (?wt) formats, respectively. When the value of *total_frames* is 44 in decimal, the output from these commands, in order, is as follows:

```
_total_frames: 2c
_total_frames: 54
_total_frames: 44
```

You can display the values stored in an array variable in a similar manner. Use the *count* parameter to specify the number of elements in the array to display. For example, the command

```
(adb) _branch,5?wt
```

moves *dot* to the beginning of the array and displays the next five words in decimal format. In this case, *branch* is an array of integers (which are one word in length). Each element is displayed as shown below:

```
_branch:  0     100   500   1000
          1500
```

Getting help

adb provides an online help command that displays brief descriptions of the adb commands. For each command, the help displays the command name, its parameters, and a brief description of what the command does.

The information is contained in a single file and the `)help` command displays the file one page at a time.

To access online help, enter:

```
(adb) )help
```

The information is displayed as shown in Figure 3.

Figure 3 adb online help

```
(adb) )help

Adb Helpfile
"$" commands
------------
$b              display breakpoints in the current job.

[adr][,cnt]$c   display a stack backtrace for the current thread. if adr is
                specified, the trace begins at that address. if cnt is
                specified, ony cnt frames are displayed. the default for adr is
                the current value of the frame pointer; the default for cnt is
                all frames.

$e              display the values of all non-zero global variables in the
                current job.

$g              get new symbol/core file names. this will change the names of
                both the symbol and core files in the current job and reset the
                data associated with it. a file name of "-" should be used if
                the file should be closed and not reopened. a null file name
                (carriage-return only) will cause the file to be unchanged.

                invoking this command will cause the file mappings to be reset
                even if neither of the files are changed.

$h[r]?f[x]      display hardware (communications) registers. works the same way
                that the $a, $s, and $v register commands work.

$hr=f value     modify hardware registers. same as $a, $s, and $v. the format f
                must be "l".

$hr[l|u]        lock/unlock hardware register.

$i              display the values of all non-zero internal adb variables.

[job]$j         display the status of job. the default for job is all active
                jobs.

$m              display address mappings in the current job.

[cnt]$n         display the maximum number of processors that may be allocated
                to a process. if cnt is specified, the maximum is set to its
                value. this value is a global value, not a per-process value.
                setting this value while a process is running has no effect.

$o              toggles the operating mode of the current job from sequential to
                chained or vice-versa, depending on the operating mode.

$r[w]           display registers from the current thread in the current job. if
                w is specified, symbols are displayed in a "wide" format, where
                A registers are lined up with S registers.

$R[w]           display registers from all threads in the current job. w is
                handled the same as for $r.

$t [symbol]     searches the symbol table in the current job for symbol, and
                display it in an nm(1) style output format. symbol may be either
                a symbol name or a symbol value.if symbol is not specified, all
                symbols are displayed.

$vl?f[x]        display the vector length register. works the same way that the
                $a, $s, and $v register commands work.
```

Figure 3 (continued) adb online help

```
$vl=f value       modify the vector length register. same as $a, $s, and $v.
                  the format f must be "b".

$vml?f[x]         display the lower 64 bits of the vector merge register. works
                  the same way that the $vl register command works.

$vml=f value      modify the lower 64 bits of the vector merge register. same
                  as $vl. the format f must be "l".

$vmu?f[x]         display the upper 64 bits of the vector merge register. works
                  the same way that the $vl register command works.

$vmu=f value      modify the upper 64 bits of the vector merge register. same
                  as $vl. the format f must be "l".

$vs?f[x]          display the vector stride register. works the same way that
                  the $vl register command works.

$vs=f value       modify the vector stride register. same as $vl. the format f
                  must be "w".

radix$X           set the output radix. this determines the radix in which
                  numbers are displayed. the legal values for radix are 0
                  (default), 8 (octal), 10 (decimal), and 16 (hexadecimal).
                  remember that the value of radix is determined by the
                  current input radix.


":" commands
------------

adr:b             set a breakpoint at adr in the current job.

[adr][,cnt]:c     continue the current thread in the current job.

[adr][,cnt]:C     continue all threads in the current job.

adr:d             delete the breakpoint at adr in the current job.

:D                delete all breakpoints in the current job.

:e                change the exit dispostion of the current job to RELEASE.
                  that is, do not terminate the job when exiting adb - continue
                  it.

[job][,thd]:f     brings job/thd into the debug foreground. that is, job
                  is made the current job, thd is made the current thread
                  in job. if neither job nor thd are specified, the current
                  job and thread are displayed. the default for job is the
                  current job; the default for thd is the previous current
                  thread for the job.

:i                toggle inherit mode in the current job. when inherit mode
                  is set, this allows the debugging of child processes.

:k                terminate (kill) the current job.

:m                toggle scheduling mode in the current job. scheduling mode may
                  be either fixed or dynamic. it is initially set to fixed.
```

Figure 3 (continued) adb online help

```
[adr][,cnt]:r   run a process, starting at adr and stopping when cnt break-
                points have been hit. any arguments specified after the :r
                will be passed to the process. the default for adr is the
                normal entry point for the process. the default for cnt is 1.

[adr][,cnt]:s   single step the current thread.

[adr][,cnt]:S   single step all threads.


")" commands
----------------

comment         introduces a comment line

help            displays this file

static          toggles the static symbol usage flag. when set, all references
                to symbol names will include static symbols.

status          display status about things like input radix, output radix,
                inherit mode, etc.
```

adb uses the paging mechanism specified by the PAGER environment variable. If the variable is not defined, adb uses more(1) to display the information.

Quitting adb

To quit adb, at the (adb) prompt enter:

```
(adb)$q
```

or

```
(adb) CTRL-d
```

The shell prompt is then displayed.

Quick start for new users

4

This chapter presents instructions and examples on how to perform five simple, frequently-used tasks with adb. You can use these instructions to do the following:

- Display a stack backtrace
- Modify values of scalar, vector, and address registers
- Disassemble an object file
- Patch a binary
- Examine a core file

The C program that converts tabs to spaces in Appendix A is used where necessary in the examples in this chapter.

Displaying a stack backtrace

adb is frequetly used to display and examine stack backtraces. A stack backtrace shows the current state of the program being debugged and how the program got to that state.

To look at a stack backtrace, use the following steps. In the following procedure, a.out is an executable file created by compiling the tabs program shown in Appendix A.

Step 1 Enter **adb a.out** at the shell prompt.

Step 2 Enter an appropriate address name and `:b` to set a breakpoint. (in this case, `settab:b`).

Step 3 Enter **:r** to start the program executing.

Step 4 Enter **$c** to display the stack backtrace, as shown below.

```
% adb a.out
(adb)settab:b
(adb):r
job 0: running
Job 0/0: breakpoint _settab:    sub.w     #24, a0
(adb) $c
_settab(80008270) from _main+0x24 [ap = ffffcab0]
_main(1,ffffcacd, ffffcae4) from _ap$envret+0x26
                  [ap = ffffcad0]
(adb)
```

Stack backtraces show the order in which routines are called; the last routine called is listed at the top of the output and the first routine called is listed last. In the example above, the `_main` routine was the first routine called, and the `_settab` routine was the last routine called.

Stack backtraces also display values passed to routines, and locations from which routines are called. Using this kind of output, you can determine the probable cause of a core dump, and get a reasonably good idea of which instruction caused a failure.

For more information on examining stack backtraces, refer to Chapter 5, "Interpreting stack backtraces" on page 89.

Modifying register values

Using adb, you can modify the values of scalar, vector, and address registers.

Scalar registers

To modify the value of a scalar register, you can use one of the following examples:

Example 1 Use the `$s=` command to modify the value of scalar register S5, then use `$s?` to verify the change. The value is expressed as a decimal number (the `0t` prefix).

```
(adb) $s5=l 0t1234567
s5=000000000012d687
(adb) $s5?lt
s5=        1234567
(adb)
```

Example 2 Another way to use the `$s=` command to modify a scalar register is shown below. The value is expressed as a hexadecimal number, which is the default.

```
(adb) $s6=l 0ff7fed54
s6=00000000ff7fed54
(adb) $s6?l
s6=00000000ff7fed54
(adb)
```

For more information regarding modification of scalar machine registers, refer to the `$s=` reference page on page 245.

Vector registers

To modify the values of a vector register, use the `$v=` command:

Example Assign 0 to the first element of vector register 0.

```
(adb) $v0:0=l 0
v0[000]=000000000000000
(adb)
```

For more information regarding modification of vector machine registers, refer to the `$v=` reference page on page 253.

Address registers

To modify the value of an address register, use the `$a=` command.

Example 1 Assume you want to change the value of address register FP from 26 to 11. Use the `$fp=`command, as shown below.

```
(adb) $Ffp=w0t11
FP=000000000a
(adb)
```

Example 2 Another way to use the `$a=` command is shown below. Assume you want to assign the value 11 to the A2 register. Enter `$a2=w0x11`.

```
(adb) $a2=w0t11
FP=000000000a
(adb)
```

For more information regarding the modification of vector machine registers, refer to the `$a=` reference page on page 201.

Disassembling an object file (or *single stepping*)

You might need to disassemble an object file to:

- Locate or examine a bug
- Study an algorithm

When you disassemble an object file, you step through the file, looking at machine instructions in the current thread one at a time. This process is often referred to as *single stepping*. Use the following instructions:

Step 1 Invoke `adb` with the name of the object file.

Step 2 Enter `settab:b` to set a breakpoint and `:r` to start the program executing.

Step 3 Enter :s to step forward and display the next machine instruction.

```
% adb tab.o
(adb)settab:b
(adb):r
job 0: running
job 0/0: stopped at _settab+0x4: ld.w    #0,s0
(adb) :s
job 0: running
job 0/0: stopped at _settab+0x8: st.w    s0,-4(fp)
(adb) :s
job 0: running
job 0/0: stopped at _settab+0xc: ld.w    -4(fp),s0
(adb) :s
job 0:
job 0/0: stopped at _settab+0xc: ld.w    -4(fp),s0
(adb)
```

By disassembling an object file in this manner, you can examine each machine instruction and make changes if needed.

For more information, refer to

- Reference page ":s" on page 187
- Reference page ":S" on page 189
- "Example adb multithreaded debugging session" on page 125.

Patching a binary

You might need to patch a binary in order to fix a bug or modify an application. You can patch a binary on many kinds of files, including:

- a.out
- object.o
- lib.a
- /vmunix

Note

In order for `adb` to inspect or reassign (patch) values in data segments, the target file must be linked.

To patch a binary, follow these instructions:

Step 1 Invoke `adb` on the file in question, using the `adb -w` option which allows writing in a file.

Step 2 Search for the value in question using the `?g` command. In the screen example below, the search is for a `calls` instruction (represented by the halfword 2140).

```
% adb -w a.out
Convex Debugger ($Date 88/09/21 15:51:31$)
Use ')help' for help.
(adb) _runpcs?gh2140
_runpcs+0x50
(adb)
```

Another way to search is to use the `adb ,`*x*`?i` command, which displays *x* number of lines of machine instructions.

Step 3 Assign a new value to replace the old one, using the `?=` (for object files) or `/=` (for core files) command.

```
(adb) _runpcs?=123
_runpcs:        5ac05b80=       123
(adb)
```

For more information on patching binaries, refer to these reference pages:

- "?g" on page 275
- "/g" on page 321
- "?=" on page 261
- "/=" on page 307

Examining a core file

You might need to examine a core file to determine why a program failed.

Follow these instructions:

Step 1 Enter `adb - core` at the ConvexOS prompt.

Step 2 Enter `$?` at the (adb) prompt. The `$?` command displays registers and the reason (shown by a signal) why the program failed.

```
% adb - core
Convex Debugger ($Date 88/09/21 15:51:31$)
Use ')help' for help.
(adb)
.
.
(adb) $?
job 0:no process
job 0:coredump of 'rlogin', version 10.0
job 0:coredump created Mon Sep 28 10:55:31 1992
job 0/0: stopped by a bus error (unknown)

job 0/0: register display

pc=80016efc
ps=02108000 (EF, DZE, FE)
sp=ffffca2c a1=00000050 a2=ffffca40 a3=ffffcac2
a4=80024218 a5=00000014 ap=ffffca40 fp=ffffcac2
s0=000000000000003 s1=00000000000000d s2=000000000000002 s3=000000000000001
s4=000000000000001 s5=000000000000004 s6=000000000024218 s7=000000000024600
```

Step 3 Enter `$c` to display the stack backtrace.

```
(adb)$c
0x80016ef8(0,ffffca5b,1) from 0x80001eb0 [ap = ffffca40]
0x80001e86() from 0x80001be6 [ap = ffffca70]
0x80001a0c(0) from 0x800019c0 [ap = ffffca8c]
0x80001434(1,ffffcadc,ffffcae4) from 0x80001108 [ap = ffffcacc]
(adb)
```

Looking at a core file in this manner, you can determine the order in which routines were called, the values passed to each routine, the location from where each routine was called, and learn which instruction caused the failure. For more information on examining core files, refer to:

- "Displaying information from a corfile" on page 57
- "Core files" on page 96

- Reference page "$?" on page 199
- Reference page "$c" on page 209

Basic adb debugging

5

This chapter explains how to use the CONVEX adb debugger. It explains many of the basic tasks you perform when debugging a program at the assembly-language level. Commands are introduced as they relate to specific tasks. Examples are provided for each command to help you see what each command does.

Topics discussed in this chapter include:

- Displaying information
- Moving around in memory
- Modifying program data
- Manipulating breakpoints
- Executing program instructions
- Interpreting stack backtraces
- Executing commands from a file

Displaying information

Because adb controls execution of a process, it has access to program instructions, program data, status of all processes, and several other pieces of information that are useful when debugging a program. Among the information you can display are

- Data in the object file
- Data in the core file
- adb-specific information
- Contents of CONVEX registers
- Internal and global variables
- Assembly-language instructions

Displaying information from an *objfile*

The ? command displays data from the executable image, *objfile*, that you may have specified when you invoked adb. The ? command displays the contents of memory at a specified location; it has the format

[*address*] ? modifier[*parameters*]

Modifers and parameters for ? commands are listed in the section titled "? commands" on page 20.

The *address* specifies the area of memory from which to display data. You can specify an actual or a symbolic *address* (0x8000196c or _main, for example). You can also specify a relative *address* by adding or subtracting values from a known *address* (_main+3a, for example). For a more complete description of addresses, see the section titled "adb address expressions" on page 35.

For example, to display a byte, specified by radix *x*, from an object file, use the ?b command, as shown in Figure 4.

Figure 4 Displaying information from an object file

```
(adb)?bx
_settab:                15
(adb)
```

For more information, refer to the section titled "? commands" on page 20, or to Chapter 9, "adb command reference" on page 153.

Displaying information from a *corfile*

The / command displays data from the core image, which you may have specified when you invoked `adb`. It displays the contents of memory at a specified location, and has the form:

[*address*] / `modifier`[*parameters*]

Modifers and parameters for / commands are listed in the section titled "/ commands" on page 22.

The *address* specifies the area of memory from which to display data. You can specify an actual or a symbolic *address* (0x8000196c or `_main`, for example). You can also specify a relative *address* by adding or subtracting values from a known *address* (`_main+3a`, for example). For a more complete description of addresses, see the section titled "adb address expressions" on page 35.

For example, to display the value of *dot* in symbolic form from a core file, use the `/a` command, as shown in Figure 5.

Figure 5 Displaying information from a core file

```
(adb)/a
```

For more information, refer to the section titled "/ commands" on page 22, or to Chapter 9, "adb command reference" on page 153.

Displaying instructions

To display the instruction at the current address, use the `?i` command as shown in Figure 6. The address is assumed to be dot.

Figure 6 Displaying instructions with `?i`

```
(adb)?i
 0x80001954:    sub.w #0,a0
(adb)
```

The value of dot is 0x80001954 (the current location), and the instruction stored at that location is `sub.w #0,a0`. Figure 7 shows how to use the `?i` command to display 10 instructions starting from `_main`.

Figure 7 Displaying 10 instructions with `?i`

```
(adb) _main,0t10?i
_main:                  sub.w       #0,a0
                        sub.w       s0,s0
                        st.w        s0,0x8000a858
                        mov         a0,a6
                        pshea       0x0
                        calls       0x80001160
                        add.w       #4,a0
                        ld.w        12(fp),a6
                        ld.w        0x8000a858,s0
                        eq.w        #0,s0
(adb)
```

You can also combine formats to display additional information. Figure 8 shows that you can display the instructions along with their corresponding addresses by combining the `a` and `i` formats.

Figure 8 Displaying 10 instructions with `?ia`

```
adb) _main,0t10?ia
_main:                  sub.w       #0,a0
_main+0x2:              sub.w       s0,s0
_main+0x4:              st.w        s0,0x8000a858
_main+0xa:              mov         a0,a6
_main+0xc:              pshea       0x0
_main+0x10:             calls       0x80001160
_main+0x16:             add.w       #4,a0
_main+0x18:             ld.w        12(fp),a6
_main+0x1c:             ld.w        0x8000a858,s0
_main+0x22:             eq.w        #0,s0
_main+0x26:
(adb)
```

In Figure 7 and Figure 8, the number of instructions is specified as `0t10` to force `adb` to interpret the 10 as a decimal number rather than a hexadecimal one. In Figure 8, specifying the `a` format causes the value of *dot* to be displayed in addition to each instruction.

The length in bytes of the last instruction displayed can be calculated by subtracting the previous instruction address from the current instruction address. In Figure 8, the length of the `eq.w #0,s0` instruction is 4 bytes (0x26 minus 0x22).

Displaying program variables

You can also use the `?` and `/` commands to display values of global variables. If you know the type of a global variable, you can display the appropriate amount of memory for the variable. Then, specify the appropriate *address* and *format* for the command to display the value of a variable.

To display the value of a variable, you can use a symbolic address (the variable name) and the appropriate format. For example, if the variable *j* is defined as type `int`, you know it uses one word of memory. You can display a variable's value as shown in Figure 9.

Figure 9 Displaying program variables with `j?w`

```
(adb) _j?w
_j:210
(adb)
```

In Figure 9, `adb` displays the value of *j* in the *corefile* as a hexadecimal number. By default, `adb` expects input and displays output in a hexadecimal format. You can override these defaults individually as desired by specifying the appropriate radix. You can also change the default settings for input and output using the `$x` and `$X` commands. See the section titled "Overview of adb commands" on page 12.

The variable `lesson_name` is a character string. Figure 10 shows how to display its value.

Figure 10 Displaying values

```
(adb) _lesson_name?s
_lesson_name: file_structure
(adb)
```

The variable branch used inFigure 11 is defined as an array [0..3] of integers. As such it uses 16 bytes of memory (four 4-byte integers). Figure 11 shows how to display different parts of the array. The first command displays the value of the first array element. The second command displays the third element of the array; the command uses relative addressing. The third command displays all elements of the array (starts at _branch and displays four words of memory in decimal format). In each case the t format is used to display the integer in decimal format.

Figure 11 Displaying different parts of an array

```
(adb) _branch?wt
_branch:    0
(adb) _branch+8?wt
_branch+0x8: 521
(adb) _branch,4?wt
_branch:    0          521        525        528
(adb)
```

The relative address _branch+0x8 was calculated by adding the size of the first two elements of the array to the address of the beginning of the array's space in memory.

Displaying job-related information

The adb debugger maintains information about the state of the job being debugged. You can display the status of a job and determine a job's PID and the signal that caused the job to stop. These functions are performed by the following two commands:

```
$j
$?
```

The $j command displays the status of the current job. The status display includes

- Job number
- PID number
- Current job and thread
- List of the modes
- Current state
- Name of the executable image
- Name of the core image

The format of the command is

[*job*] $j

where *job* is the job ID number. With no *job* parameter, the $j command displays the status of all active jobs. The latter instance is shown in.

Figure 12 Displaying the status of all active jobs

```
(adb) $j
      job   pid    c      state
      ---   -----  -      ------------------------
      0     10634  *      active, sequential, terminate on quit, native fpmode
                   *      thread 0: at breakpoint _main
file '?': a.out
(adb)
```

In Figure 12, the current job is job 0, and its PID is 10634. The * in the c column indicates that it is the active job. The state of the job is active, running in sequential mode and native floating-point mode, and the job will be terminated if adb quits while the job is still active.

The display also shows that the job has one thread (thread 0), and that it is the current thread. The job is stopped in thread 0 because a breakpoint was reached at _main. Finally, the output shows that the executable image for the job is a.out (the ? file). The core file is not being debugged, which is determined by its absence in the listing.

Figure 13 displays the status for a job with two threads. Thread 0 is the current thread and thread 1 is in the background (the *s in the c column).

Figure 13 Displaying status of a job with two threads

```
(adb) $j
      job   pid    c      state
      ---   -----  -      ------------------------
      0     443    *      active, sequential, terminate on quit, native fpmode
                   *      thread 0: at breakpoint _thread
                          thread 1: at breakpoint _thread
                          file '?': a.out
(adb)
```

The example in Figure 13 is shown for completeness. Debugging programs with multiple threads requires advanced debugging techniques. For more information on threads, see Chapter 7, "Multithreaded debugging" on page 107.

The `$?` command displays job number, PID for the current job, and values of the general registers (address and scalar). Figure 14 illustrates this command.

Figure 14 Using the `$?` command

```
(adb) $?
job 0: process 6522
job 0/0: register display
pc=80001172 (_main+0x2)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd78 a1=ffffcd8c a2=ffffcda0 a3=ffffcdf0 tt=00000137
a4=00000000 a5=00000030 ap=ffffcd8c fp=ffffcd78
s0=fffffffe007fffc s1=0000000000080000 s2=0000000c00000000 s3=0000000000000000
s4=00000000fffffffc s5=0000000000000080 s6=00000000e0000000 s7=0000000000070000
(adb)
```

For jobs that are executing multiple threads, the `$?` command displays information for the threads, as shown in Figure 15.

Figure 15 Using the `$?` command—multiple threads

```
(adb) $?
job 0: process 24685
job 0/0: register display

pc=80001500 (_thread+0x22)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES) sp=ffffcd54
a1=80006078 a2=000f4240 a3=80008808 tt=0011c586
a4=0001b68d a5=803d9130 ap=ffffcd68 fp=ffffcd54
s0=000000000001b68d s1=0000000000027100 s2=0000046a00000002 s3=000000dd00000000
s4=0000000500000044 s5=0000000000000000 s6=0000000000000000 s7=0000000000000000

job 0/1: register display

pc=800014e0 (_thread+0x2)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd54 a1=0000000a a2=00003f00 a3=00224721 tt=00000008 a4=00003c8d
a5=001a8618 ap=ffffcd68 fp=ffffcd54
s0=0000000000000001 s1=0000000200000001 s2=0000000000000001 s3=0000000200000001
s4=00000000001ba978 s5=0000000000000001 s6=0000000000000001 s7=0000000000000000
(adb)
```

Figure 15 illustrates that each thread in a job has its set of address and scalar registers.

If you are debugging a core dump, the `$?` command also displays the name of the executable file that caused the core dump, the version of the executable program, and the date and time the core dump was created. This command is shown in Figure 16.

Figure 16 Displaying the name of a executable file with `$?`

```
(adb) $?
job 0: no process
job 0: coredump of 'core.e', version 7.0.0.7
job 0: coredump created Tue Jul 5 09:59:11 1988
job 0/0: stopped by a floating point exception (integer divide by zero)

job 0/0: register display

pc=ffffd08a
ps=02000080 (EF,RES)
sp=ffffcd0c a1=ffffcd9c a2=ffffcdb0 a3=ffffcdfc
a4=00000008 a5=00000040 ap=ffffcd9c fp=ffffcd0c
s0=0000000000000000 s1=000000000000000a s2=0000000000000000 s3=00000000ffffffff
s4=0000000080054990 s5=000000008004f3c0 s6=00000000e0000000 s7=0000000000000000
(adb)
```

Displaying breakpoints

The $b command lists all breakpoints set in the current job. The breakpoint listing shows where each breakpoint is set, which occurrence will cause it to stop execution, and commands to execute when the breakpoint is reached. The list is ordered according the order in which you set the breakpoints, with the most recently set breakpoint listed first.

To display breakpoints, enter $b as shown in Figure 17.

Figure 17 Displaying breakpoints

```
(adb) $b
job 0: breakpoint display
count bkpt      command
3     _Get_response:s;$r
1     _main
(adb)
```

In this example, the job has two breakpoints. One breakpoint is set at _main; adb will stop execution the first time the program reaches the instruction at _main. The other breakpoint is set at the entrance to the _Get_response routine. adb will stop execution the third time the instruction is executed, then it will execute the :s and $r commands to execute one more instruction and display the general registers.

When you set a breakpoint at an address where one is already set, adb supersedes the previous breakpoint with the new one. With the breakpoints set as in the previous screen, setting another breakpoint at _Get_response changes the listing as shown in Figure 18.

Figure 18 Setting a new breakpoint

```
(adb) _Get_response:b
(adb) $b
job 0: breakpoint display

count bkpt      command
1     _Get_response
1     _main
(adb
```

adb will now stop execution the first time the program reaches _Get_response and will not execute any adb commands.

The `$b` command only displays breakpoints set for the current job. The section titled "Setting breakpoints" on page 80 explains how to set breakpoints. To display breakpoints set for other jobs, you must first bring the other job to the foreground.

Displaying registers

The `adb` debugger lets you look at the registers used by `adb` and the executing program. CONVEX supports four sets of registers: address, hardware communication, scalar, and vector. All CONVEX machines support address, scalar, and vector registers; however, hardware communication registers are supported only on CONVEX machines that have multiple CPUs. The commands to display these register sets include:

`$a?` (address)

`$h?` (hardware communication)

`$s?` (scalar)

`$v?` (vector)

You can also look at the PC and PSW registers and the special register *dot*.

Each of these commands is explained in the following sections.

Displaying address registers

The `$a?` command displays the contents of an address register. CONVEX supports eight 32-bit address registers (0-7). All data stored in these registers is 32 bits long, although you can display any portion (byte, halfword, word) of the data.

The format of the command is

`$a` [*reg*]`?` *format* [*radix*]

where:

reg	Number of the address register you want to display.
format	One of the letters defining the format (size) of the data. Since address registers contain only 32 bits of data, you can not use the `l` format.
radix	A letter that defines the number base in which to display the data. If no *radix* is specified, the default output radix is used.

The specific *format* and *radix* values are described in "Data display formats" on page 40.

Figure 19 shows how to display the value of the fifth address register. The *format* (w) displays 32 bits of data and the *radix* (x) displays it in hexadecimal.

Figure 19 Displaying the value of specific registers

```
(adb) $a4?wx
a4=000eb66a
(adb)
```

If you don't specify a register, adb displays the contents of all address registers, as shown in Figure 20. Spaces between parts of the command syntax are discretionary.

Figure 20 Displaying the value of all address registers

```
(adb) $a?wx
sp=ffffcd54 a1=ffffffff a2=00000011 a3=00000000
a4=000eb66a a5=803d9130 ap=ffffcd68 fp=ffffcd54
(adb)
```

Three of the registers are not identified by numbers (a0, a6, a7). These three address registers are the SP, AP, and FP registers respectively. You can display the value of one of these registers either by specifying the number or the name of the register, as shown in Figure 21.

Figure 21 Displaying register values—by number or name

```
(adb) $a7?wx
fp=ffffcd54
(adb) $fp?wx
fp=ffffcd54
(adb)
```

You can display the SP, AP, FP, PC, PSW, and *dot* by using the commands shown in Table 17.

Table 17
Displaying SP, AP, FP, PC, PSW, and *dot*

Register	Command
SP	`$sp?`*format radix*
AP	`$ap?`*format radix*
FP	`$fp?`*format radix*
PC	`$pc?`*format radix*
PSW	`$psw?` *format radix*
dot	`$dot?` *format radix*

Displaying hardware communication registers

The `$h?` command displays the values of the hardware communication registers. CONVEX supports sixty-four 64-bit user-addressable hardware communication registers (numbered zero through 63). All data stored in these registers is 64 bits long, though you can display any portion (byte, halfword, word, longword) of the data.

The format of the command is

`$h` [*reg*]`?` *format radix*

where:

reg — Number of the hardware communication register you want to display.

format — One of the letters defining the format (size) of the data.

radix — One of the letters defining the number base in which to display the data.

Specific *format* and *radix* values are described in the section titled "Data display formats" on page 40.

Figure 22 shows how to display the contents of the 17th (specified as a hexadecimal number) hardware communication register. The `l` and `x` parameters display a longword of data in hexadecimal format.

Figure 22 Hardware register contents, 17th register

```
(adb) $h0x10?lx
job 0: communication register display

h[16]=000000001a37f640 (0)
(adb)
```

The 0 in the display indicates that the register is not locked; a 1 appears when the register is locked. When a hardware communication register is locked, it means that the register has valid data stored in it.

Displaying scalar registers

The `$s?` command displays the values of the scalar registers. CONVEX supports eight 64-bit scalar registers (numbered zero through 7). All data stored in these registers is 64 bits long, though you can display any portion (byte, halfword, word, longword) of the data.

The format of the command is

`$s` [*reg*]`?` *format radix*

where:

reg	Number of the scalar register you want to display.
format	A letter that defines the format (size) of the data.
radix	A letter that defines the number base in which to display the data.

The specific *format* and *radix* values are described in the section titled "Data display formats" on page 40.

Figure 23 shows how to display the contents of the sixth scalar register. The `w` and `x` parameters display the data as a word in hexadecimal.

Figure 23 Displaying a single scalar register

```
(adb) $s5?wx
s5=00000001
(adb)
```

If you don't specify a register, the values for all scalar registers are displayed, as shown in Figure 24, where the values are displayed in the decimal number base (`t` radix specifier).

Figure 24 Displaying all scalar registers

```
(adb) $s?wt
s0=0    s1=130000     s2=2     s3=0
s4=68   s5=1          s6=0     s7=0
(adb)
```

Displaying vector registers

The `$v?` command displays the values of the vector registers. CONVEX supports eight 128-element vector registers (numbered zero through 7), where each element is 64 bits. All data stored in these registers is 64 bits long, though you can display any portion (byte, halfword, word, longword) of the data.

The format of the command is

`$v` [*reg*][: *element*][, *count*] `?` *format radix*

where

reg	Number of the vector register you want to display.
element	Element in the register at which you want to start displaying data.
, count	Number of elements you want to display.
format	A letter that defines the format (size) of the data.
radix	A letter that defines the number base in which to display the data.

Specific *format* and *radix* values are described in the section titled "Data display formats" on page 40.

Figure 25 shows how to display the first five elements of a vector register. The 1 specifies the second vector register, the 0 specifies the starting element, and the 5 specifies the number of elements to display. The `l` and `x` parameters display the data as longwords in hexadecimal.

Figure 25 Displaying the first five elements of vector register

```
(adb) $v1:0,5?lx
job 0: vector register display for thread 0
vl=00000080 vs=00000008
vm=00000000000000000000000000000000
v1[000]=401994d2972e9d86
v1[001]=403cee0adada1e2c
v1[002]=4024c4eee884ffd3
v1[003]=bfd0adc8cc73e3ea
v1[004]=3fe10fdcbbdcb7fe
(adb)
```

Displaying internal and global variables

adb maintains two sets of system variables: internal variables and external (global) program variables. The $i and $e commands list the values of these variables, respectively.

Internal variables are used by adb to track various locations in the program, and are shown in Table 18.

Table 18
adb internal variables

Variable	Location in the program
b	Base address of the data segment
d	Length of the data segment
e	Entry point of the program
m	Magic number (identifies the type of file)
s	Length of the stack
t	Length of the text segment

The $i command displays the values only for those internal variables with nonzero values. For example, Figure 26 shows the display from a $i command.

Figure 26 Command output from $1

```
(adb) $i
job 0: internal variable display
b = 0x80008000
d = 0x4000
e = 0x80001000
m = 0x181
s = 0x2000
t = 0x7000
(adb)
```

In Figure 26, the display shows that for job 0, the base address for the data segment is 0x80008000, the length of the data segment is 0x4000, the entry point of the program is 0x80001000, the magic number is 0x181, the length of the stack is 0x2000, and the length of the text segment is 0x7000.

The `$i` command displays this information only for the current job. To see this information for a different job, bring it to the foreground using the `:f` command. See the section titled "Overview of adb commands" on page 12. For more information on internal variables, refer to "Internal variables" on page 106.

The `$e` command displays the values of all known global variables. Global variables include any program variables declared globally, variables contained in library files included in the program, and variables that are part of system libraries automatically loaded.

Figure 27 illustrates a `$e` command listing. This listing shows all of the variables in one screen; unless you stop the scrolling, this listing will normally scroll off of the screen.

Figure 27 Command listing from `$e`

```
(adb) $e

job 0: external variable display

 _environ:          ffffcda0
mth$tstack:         e007fffc
mth$dontfork:       0
mth$soff_flag:      70000
_n:                 0
_threadcnt:         0
 _array:            1
 _pcnt:             0
_psum:              0
_errno:             0
__iob:              0
__sobuf:            0
__lastbuf:          80007704
__ctype_:           202020
_realloc_srchlen:   4
curbrk:             803da000
_end:               803da000
minbrk:             803da000
(adb)
```

In Figure 27, the variables beginning with `mth$` are loaded from a library, the variables between `_n` and `_errno` are global program variables, and the remainder of the variables are part of system libraries.

Moving around in memory

When you display data from the executable or core file, adb implicitly moves around in memory. For example, when you display an instruction, adb first moves *dot* to the specified address and displays the instruction located at the address. After the instruction is displayed, dot is located immediately following the location of the last instruction displayed.

Using adb, you have the ability to move dot to any location in the virtual address space of the executable or core images. In this way, you can look at or modify any portion of memory. You can specify a location in memory in one of three ways:

- Specify a logical address (0x80001a30)
- Specify a symbolic address (_main)
- Use a combination of addresses (symbolic and logical) and expressions (_main+0x11).

To move dot to a new location and memory (and do nothing else), type the address or address expression of the location. dot is moved to the specified address.

Figure 28 illustrates moving to different locations using logical addresses.

Figure 28 Moving to new location using a logical address

```
(adb) 0x80001172?ia
0x80001172:    sub.w s0,s0
0x80001174:
(adb) 0x800014e0?ia
0x800014e0:    inc.w   0x8022,a1
0x800014e8:
(adb) 0x800014b0?ia
0x800014b0:    sub.l s0,s0
0x800014b2:
(adb)
```

Moving around memory using symbolic addresses is shown in the next screen. After the first ?i command is executed to display the instruction at dot, adb automatically displays the instruction at the new address, as shown in Figure 29.

Figure 29 Moving to new location using a symbolic address

```
(adb) _main
adb
(adb) ?i
_main:sub.w    #4, a0
(adb) _thread
_thread:ld.w    #1, a1
(adb) _sum_in_parallel
_sum_in_parallel:ulk      0x8021
(adb)
```

Normally, you would combine the first two commands into a single one, `_main?i`. In the example, they are separated to illustrate that dot is moved when an address is specified by itself.

You can also move to new locations in memory by using an address in conjunction with the `adb` arithmetic operators. The operators are described in the section titled "adb address expressions" on page 35.

You can add values to or subtract values from an address, specify an address by multiplying two numbers, and combine multiple expressions to calculate an address. Figure 30 shows two ways of specifying the same address.

Figure 30 Two ways of specifying the same address

```
(adb) _main+0x6?i
_main+0x6:    div.l v7,s7,v4
(adb) _main+(0t3*0t2)?i
_main+0x6:    div.l v7,s7,v4
(adb)
```

Both commands move dot to the sixth byte in memory following `_main`. The first command adds 6 to the address of `_main`, while the second adds the product of 2 and 3 to the address of `_main`.

Modifying program data

In addition to displaying information, `adb` lets you change any part of a process. You can change the values of variables, registers, and the value at any location in memory.

Modifying memory

`adb` provides two commands for changing data in memory. The `?=` command modifies memory used by the executable image, and the `/=` command modifies memory used by the core image.

The format of the commands are

[*address*]`?=` *format value*

and

[*address*]`/=` *format value*

where

address Location in memory you want to start modifying.

format A letter that defines the format (size) of data you want to modify.

value What you want to store at this location in memory.

The specific *format* and *radix* values are described in the section titled "Data display formats" on page 40.

Figure 31 illustrates several commands that assign values directly to memory. The first command writes a longword (64 bits) of data starting four bytes after the address pointed to by `_main`. The second command assigns a word (32 bits) of data starting 22 bytes (22 as hexadecimal) after the address pointed to by `_thread`. The last command assigns one byte of data to an absolute address.

Figure 31 Assigning values directly to memory

```
(adb) _main+4?=l 0x8ff3
_main+0x4:   3638fffc3238fffc   =    8ff3
(adb) _thread+0x22?=w 0t123
_thread+0x22: 74fb4c62    =   7b
(adb) 0x80002123?=b 0x4
__doprnt$n+0x673:     10   =  4
(adb)
```

When you assign new values to memory, adb displays the old value on the left of the = and the new value on the right. The output radix in the above example is 16, so all values are displayed as hexadecimal numbers. The values 123 decimal and 7b hexadecimal are equivalent.

You can also use these commands to assign values to program variables. You can, for example, assign unexpected values to variables to see if the program can properly handle them. Likewise, if a program fails when a variable assumes a certain value, you can manipulate the variable and see how the program reacts.

To assign new values to global variables, specify the variable name as the address parameter. The command

```
(adb)_is_done?=w 0
```

assigns the value 0 (zero) to the integer variable _is_done.

Assume that a program has the following global variables (extracted from a $e command):

```
_branch:                    0
_valid_answer:             6a
_stop_lesson:               0
_lesson_name:        66696c65
```

The declarations for these variables in the C program are

```
int                 branch[4];
int              valid_answer;
int               stop_lesson;
char         lesson_name[80];
```

Figure 32 shows how to change the value of valid_answer. After the variable is changed, dot points to the first location modified. Therefore, no address parameter is needed to display the value of the variable.

Figure 32 Changing the value of `valid_answer`

```
(adb) _valid_answer?=w0x1
_valid_answer: 6a        =         1
(adb)?wt
_valid_answer: 1
(adb)
```

Figure 33 shows how to modify elements within an array. First, display all elements of the `branch` array. Then change the second and third elements. In this example, each element of the array uses four bytes of memory because the array was declared as integer. Then verify that the values are changed. In , the core image is being debugged.

Figure 33 Debugging the core image

```
(adb) _branch,4/wt _branch:   0 210 107   210
(adb) _branch+0x4/=w0t200 _branch+0x4: d2   = c8
(adb).+0x4/=w0t205 _branch+0x8: 6b    =   cd
(adb) _branch,4/wt _branch:    0   200   205    210
(adb)
```

The output radix in the previous example is 16, so all values are displayed as hexadecimal numbers. The values 210 decimal and d2 hexadecimal are equal.

In Figure 33, the first assignment added four bytes to the address of the beginning of the `branch` array and wrote four bytes of data to memory. Because the current position moves to the second element in the array, the next command simply adds four bytes to the current address (.).

Modifying registers

You can change the values of registers similarly to the way you display them. The commands for modifying registers include:

`$a=` (address)

`$h=` (hardware communication)

`$s=` (scalar)

`$v=` (vector)

Each register is of a single size. For example, all address registers are 32 bits long. For this reason, you are only permitted to assign values of the right size (format) to the registers. Table 19 shows the size of each register type and the format you must use to assign a value to the appropriate register.

Table 19
Register assignment formats

Register Name	Quantity	Size (bits)	Format Letter
address	8	32	`w`
communication	64	64	`l`
scalar	8	64	`l`
vector	8	64	`l`

Each vector register has 128 elements.

The formats for the commands to assign values to the address, hardware, scalar, and stack pointer registers are:

`$a` *reg = format value*

`$h` *reg = format value*

`$s` *reg = format value*

`$sp=`*format value*

Figure 34 shows examples of assignments for each of these register types. The format for the address register is `w` while it is `l` for the others. Also illustrated is the command to reassign of the stack pointer (the `$sp` command).

Figure 34 Assigning values to address, hardware, scalar registers

```
(adb) $a4=w 0x80001234
a4=80001234
(adb) $sp=w0x800021d6
sp=800021d6
(adb) $h10=l 0t1032
h[16]=0000000000000408 (0)
(adb) $h0t10=l 0t1032
h[10]=0000000000000408 (0)
(adb) $s4=l 0o3451
s4=0000000000000729
(adb)
```

Since there are eight vector registers, each consisting of 128 64-bit elements, you must specify both the register number and the specific element to which you want to assign a value. The format for the `$v=` command is

`$v` *reg* [: *element*] = *format value*

If you don't specify a specific element, `adb` assigns the value to element 000. Figure 35 illustrates how to assign a value to a vector register. The second command shows what happens when you don't specify an element parameter.

Figure 35 Assigning a value to a vector register

```
(adb) $v3:0t43=l 0x5a3e1

vl=00000005 vs=00000004
vm=0000000000000000000000000000001f

v3[043]=000000000005a3e1
(adb) $v4=l 0x800027e1

v4[000]=00000000800027e1
(adb)
```

Manipulating breakpoints

When you start a program under adb, it executes until it finishes. To stop a program before it completes, you can interrupt the program by pressing the INTERRUPT key (usually bound to CTRL-c). This stops the program immediately and saves the current state of the program. Unfortunately, by using this method you cannot reliably predict where the program stops.

In contrast, breakpoints provide a reliable means of controlling when, where, and if a program stops execution to return control to adb.

The next two sections describe how to set and delete breakpoints.

Setting breakpoints

When you set breakpoints in a program, you control exactly where the program stops. The breakpoint marks a spot in memory. When the program reaches the breakpoint location, adb stops the program and returns control of the program to you. You can then look at its current state, change program data, and control further execution.

The :b command sets a breakpoint in the current job. The format of the command is

[*address*] [, *count*] :b [*command*]

where:

address	Location of the breakpoint (default is dot).
, *count*	Number of times the breakpoint is ignored before the program stops (default is 1).
command	List of one or more adb commands to be executed when the breakpoint is reached.

Figure 36 sets a breakpoint at the _main routine and then lists the currently set breakpoints to verify that the breakpoint is set.

Figure 36 Setting a breakpoint at the `_main` routine

```
adb) _main:b
(adb) $b
job 0:breakpoint display
count bkptcommand
1     _main
(adb)
```

Specifying an ignore count

An *ignore count* is the number of times a breakpoint is skipped until it is triggered. Suppose that the program you are debugging fails in a specific routine, but not until the fifth time it is called. You can set a breakpoint at the routine to stop execution only when it is called the fifth time. This feature is illustrated inFigure 37.

Figure 37 Specifying an ignore count

```
(adb) _Display_next_frame,5:b
(adb) $b
job 0:breakpoint display
count bkptcommand
5     _Display_next_frame
1     _main
(adb)
```

In Figure 37, `adb` executes the program and monitors the number of times `_Display_next_frame` is called. When it is called the fifth time, `adb` stops program execution. Because this is close to the point where the program fails, you can now start monitoring and modify the program to isolate the problem.

Specifying a breakpoint handler

You may want to execute one or more debugging commands when you reach a breakpoint. Using the previous methods, you would set the breakpoint and then enter the commands when the breakpoint is reached. The *command* parameter of the `:b` command lets you specify commands to be executed automatically when the breakpoint is reached. Another term for this is specifying a "breakpoint handler."

In the previous scenario, you may execute a few machine instructions in `_Display_next_frame` to reach a known state and then display the values of some the program variables. This process is illustrated in Figure 38.

Figure 38 Specifying a breakpoint handler

```
(adb) _Display_next_frame,5:b5:s; _branch,4?wt; testch?c
(adb) $b
job 0: breakpoint display
count bkpt      command
5    _Display_next_frame,5:s; _branch,4?wt; testch?c
(adb)
```

In Figure 38, three commands are specified to be executed when the breakpoint is reached the fifth time. First, `5:s` executes five machine instructions. Second, `_branch,4?wt` displays four words of memory in decimal starting at `_branch`. Third, `testch?c` displays the value of `testch` as a character.

Once the breakpoint is set, the program can be executed using the `:r` command. When the breakpoint is reached, the information is displayed, as shown in Figure 39.

Figure 39 Displaying information with `:r`

```
(adb):r
job 0: running
job 0/0: stopped at _Clear_screen+0x2:     pshea mth$dontfork+0x90
_branch:   120      120      12000000      0
_testch:
(adb)
```

Deleting breakpoints

`adb` does not automatically delete breakpoints when they are reached; you must explicitly delete them. You can delete breakpoints either individually or as a group. The `:d` command deletes a single breakpoint, and the `:D` command deletes all breakpoints.

The syntax for commands that delete breakpoints are:

[*address*]`:d`

`:D`

To delete a single breakpoint, specify the address at which the breakpoint is set. If you do not specify an address, `adb` tries to delete a breakpoint located at the current address, and if it can't find one, it displays an error message. In Figure 40, the breakpoint display shows three currently set breakpoints, and then the `:d` command deletes two of them.

Figure 40 Deleting a single breakpoint using :d

```
(adb) $b
 job 0: breakpoint display

count bkpt      command
1     _Get_response
1     _main
4     _Display_next_frame,5:s; branch,4?wt; testch?c

(adb) _main:d
(adb) _Display_next_frame:d
(adb) $b
job 0: breakpoint display

count bkpt      command 1     _Get_response
(adb)
```

To delete all breakpoints, use the :D command as illustrated in Figure 41.

Figure 41 Deleting all breakpoints using :D

```
(adb) $b
job 0: breakpoint display

count bkpt      command

1     _Get_response
1     _main
4     _Display_next_frame,5:s; branch,4?wt; testch?c
(adb):D
(adb) $b
job 0: no breakpoints set
(adb)
```

Executing program instructions

Because executable programs are controlled by adb when you are debugging them, you must instruct adb to let the program execute. adb execution commands control the flow of the program, including normal execution, single stepping, and continuing execution after the program has stopped.

The commands in Table 20 execute program instructions.

Table 20
Commands that execute program instructions

Command	Description
:C	Continue execution for all threads in the current job
:c	Continue program execution
:r	Run *objfile* as a process
:S	Execute one instruction for each thread in the current job
:s	Execute a single instruction

The following sections describe these commands.

Starting a process

The :r command starts a program executing from its beginning. All executable images must be started with this command. If a program is stopped during execution, this command restarts the program from the beginning, not from where it stopped.

The format of the command is

[*address*] [, *count*] :r [*arguments*]

where:

address — Memory location to start execution when you don't want to start at the program's standard entry point.

, *count* — Number of breakpoints to skip before stopping (default is 1).

arguments — List of one or more arguments passed to the program as if it were executed from the shell.

To start program execution, enter:

```
(adb):r
```

The program executes. If no breakpoints are set and you do not interrupt the program, the program completes execution and returns control to adb.

Normally, you set one or more breakpoints before executing the program so that you can manipulate it before it finishes. Figure 42 illustrates a series of breakpoints that are currently set. The program :r command then executes the program and stops when the first breakpoint is reached.

Figure 42 List of breakpoints currently set

```
(adb) $b
job 0: breakpoint display
countbkpt       command 1
_main 1_Get_response
(adb):r
job 0: running
job 0/0: breakpoint _main:    sub.w #0,a0
(adb)
```

You can also instruct adb to ignore a specified number of breakpoints before it stops execution. The *,count* prefix tells the :r command to continue executing through the first *count* -1 breakpoints and stop the next time a breakpoint is reached. With this prefix argument, adb ignores the breakpoints, regardless of which ones or in which order the breakpoints are reached.

Figure 43 shows a list of several breakpoints. The ,4 count parameter instructs the :r command to stop when the fourth breakpoint is encountered.

Figure 43 Stopping execution at a specific breakpoint

```
(adb) $b
job 0: breakpoint display

count bkpt      command
1     _sum_in_parallel
1     _thread
1     _main
(adb),4:r
job 0: running
job 0/0: stopped at _thread+0x2a:         add.w a3,a4
job 0/1: breakpoint _thread: ld.w #1,a1
(adb)
```

Notice that execution does not stop at _main as it normally would.

Executing single instructions ("stepping")

You can also continue program execution one instruction at a time, also called stepping. By executing one instruction at a time, you can monitor what is happening to a program in minute detail.

The commands for executing single instructions are

- `:S` Execute one instruction for each thread in the current job
- `:s` Execute a single instruction

Both commands work the same unless multiple threads are active in the current job.

The formats for the commands are:

[*address*] [, *count*] `:S` [*signal*]

[*address*] [, *count*] `:s` [*signal*]

where:

address	Location of the instruction to be executed (default is *dot*).
, *count*	Number of instructions to execute (default is 1).
signal	Number of a ConvexOS signal you want to pass to the program.

In Figure 44, one instruction is executed. After executing `:s`, the current position moves forward two bytes (the length of the instruction).

Figure 44 Using :s to execute an instruction

```
(adb):s
job 0: running
job 0/0: stopped at _thread+0x2:        inc.w 0x8022,a1
(adb)
```

Figure 45 and Figure 46 illustrate the difference between the commands when a job has multiple threads. The process is in the same state in each example when the command is executed.

In Figure 45, the job status display shows that job 0 is executing two threads. The only instruction executed is in thread 0, which is the current thread as indicated by the * in the c column.

Figure 45 Using `$j` to execute one instruction for two threads

```
(adb) $j
job     pid       c         state
---     -----     -         ------------------------
0       18489     *         active, sequential, terminate on quit, native fpmode
                  *         thread 0: stopped at _thread+0x38
                            thread 1: at breakpoint _thread
                            file '?': a.out
                            file '/': core

(adb):s
job 0: running
job 0/0: stopped at _thread+0x3a:         br    _thread+0x18
(adb)
```

In Figure 46, one instruction is executed for each thread.

Figure 46 Using `$j` to execute one instruction for each thread

```
adb) $j
job     pid       c         state
---     -----     -         ------------------------
0       18489     *         active, sequential, terminate on quit, native fpmode
                  *         thread 0: stopped at _thread+0x38
                            thread 1: at breakpoint _thread
                            file '?': a.out
                            file '/': core
(adb):S
job 0: running
job 0/0: stopped at _thread+0x18:         ld.w #1,a4
job 0/1: stopped at _thread+0x2:         inc.w 0x8022,a1
(adb)
```

If you are not using a CONVEX machine with multiple CPUs, the `:s` and `:S` commands work identically, because there is only one thread to execute.

Continuing program execution

Two commands let you continue program execution from where the execution stops. Unlike the single instruction commands, these commands resume execution until another breakpoint is reached or the process finishes.

The commands for continuing program execution are:

`:C` Continue execution for all threads in the current *job*

`:c` Continue program execution

Both commands work the same unless multiple threads are active in the current job.

The formats for the commands are

[address] [, count] :C *signal*

[address] [, count] :c *signal*

where

address Location of the instruction to be executed.

, count Number of instructions to execute.

signal Number of a ConvexOS signal you want to pass to the program.

Killing the current process

The :k command terminates the current process. This is useful when you do not want to wait for a program to finish executing. The command kills the current job and all of its threads.

Interpreting stack backtraces

A stack backtrace shows the current state of the program and how the program got there. Stack backtraces have many uses. You can use them to determine the order in which routines are called, the values passed to each routine, and the location from which each routine is called. You can also use them to determine the cause of a core dump and get a reasonably good idea of which instruction caused the dump.

The `$c` command displays a stack backtrace, as shown in Figure 47.

Figure 47 Using the `$c` command

```
(adb) $c
0xffffd08a(2,0) from _fraction+0x10 [ap = ffffcd58]
_fraction(2,0) from _swap+0x52 [ap = ffffcd58]
_swap(2,0) from _main+0x3e [ap = ffffcd78]
_main(1,ffffcda8,ffffcdb0) from start+0x80 [ap = ffffcd9c]
(adb)
```

Backtraces contain several pieces of information. Figure 48 contains a sample program and the stack backtrace generated from the program that dumped core.

Figure 48 Sample stack backtraces and code comparison

Source code	Stack backtrace

`fraction (x,y)`
`int x, y;`
`{`
`int z;`
`z = x / y;`
`printf("%d/%d=%d n",x,y,z);`
`}`
 	
`swap (a,b)`
 	
`int a, b;`
`{`
`int temp;`
 	
`fraction (a,b);`
`temp = a;`
`a = b;`
`b = temp;`
`fraction (a,b);`
`}`
 	
`main ()`
 	
`int i, j;`
 	
`for (i=2; i>=0; i--)`
`{`
`for (j=0; j<=2; j++)`
`swap (j,i);`
`}`
	`0xffffd08a(2,0) from _fraction+0x10 [ap = ffffcd58]`
	`_fraction(2,0) from _swap+0x52 [ap = ffffcd58]`
	`_swap(2,0) from _main+0x3e [ap = ffffcd78]`
	`_main(1,ffffcda8,ffffcdb0) from start+0x80 [ap = ffffcd9c]`

The stack backtrace shows that `_main` called `_swap`, which called `_fraction`. While in the `_fraction` routine, the signal handler (0xffffd08a) was called because the program generated a ConvexOS signal when it tried to divide by zero. The numbers in the parentheses represent the arguments passed to each routine. In this example, the numbers 2 and 0 were passed to the routines called from `_main`. You can verify this by looking at the code on the left.

Reading the backtrace from left to right, the display shows the name of the called routine, the arguments passed to it, the location from which it was called, and the location of the routine's arguments. Look at the line beginning with `_fraction`. From this line, you can tell that the routine `_fraction` was passed two arguments (2 and 0), that it was called from an instruction 82 (52 hex) words into the `_swap` routine, and that the routine arguments are located at ffffcd58.

To illustrate how the routines are called, look at Figure 49, which illustrates the assembly instructions corresponding to the `_swap` routine.

Figure 49 Sample source code and assembly code comparison

Source code	Assembly-language code

`int a, b;`
`{`
`int temp;`
 	
`fraction (a,b);`
`temp = a;`
`a = b;`
`b = temp;`
`fraction (a,b);`
`}`
	`_swap:  sub.w  #4,a0`
	`_swap+0x2:  ld.w  4(ap),s0`
	`_swap+0x6:  psh.w  s0`
	`_swap+0x8:  ld.w  0(ap),s0`
	`_swap+0xc:  psh.w  s0`
	`_swap+0xe:  mov  a0,a6`
	`_swap+0x10:  pshea  0x2`
	`_swap+0x14:  calls  _fraction`
	`_swap+0x1a:  add.w  #12,a0`
	`_swap+0x1e:  ld.w  12(fp),a6`
	`_swap+0x22:  ld.w  0(ap),s0`
	`_swap+0x26:  st.w  s0,-4(fp)`
	`_swap+0x2a:  ld.w  4(ap),s0`
	`_swap+0x2e:  st.w  s0,0(ap)`
	`_swap+0x32:  ld.w  -4(fp),s0`
	`_swap+0x36:  st.w  s0,4(ap)`
	`_swap+0x3a:  ld.w  4(ap),s0`
	`_swap+0x3e:  psh.w  s0`
	`_swap+0x40:  ld.w  0(ap),s0`
	`_swap+0x44:  psh.w  s0`
	`_swap+0x46:  mov  a0,a6`
	`_swap+0x48:  pshea  0x2`
	`_swap+0x4c:  calls  _fraction`
	`_swap+0x52:  add.w  #12,a0`
	`_swap+0x56:  ld.w  12(fp),a6`

The address referenced in the stack backtrace (`_swap+0x52`) immediately follows the instruction to call `_fraction`. In the example, the `_fraction` routine was called twice from `_swap`. The stack backtrace helped you determine which call to the routine failed. In larger programs where a single routine might be called several times, using the stack backtrace to locate the proper instance can save you a lot of time.

The location from which the routine is called is only an approximation in the stack backtrace display. When adb executes an instruction, it determines the length of the instruction, which also moves the location of dot. For this reason the address of the calling instruction does not correspond to its actual location. Depending on what kind of action is taken by the operating system (handling a ConvexOS signal, for example), the distance between the actual calling instruction and the displayed address may vary by more than a single instruction.

Figure 50 shows a stack backtrace taken from a file currently executing in adb. The $c command displays the backtrace from the executable image if it is running. If the executable image is not running, adb displays the stack backtrace from the core file if it is present.

Figure 50 Stack backtrace from currently executing file

```
(adb) $c
_Get_response() from _Determine_branch+0x74 [ap = ffffcd68]
_Determine_branch() from _main+0x104[ap = ffffcd8c]
_main(1,ffffcdac,ffffcdb4) from start+0x80[ap = ffffcda0]
(adb)
```

Figure 51 shows how to look at the stack backtrace from the core file.

Figure 51 Looking at a stack backtrace from the core file

```
% adb prog1 core
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $c
0xffffd08a(2,0) from _fraction+0x10[ap = ffffcd58]
_fraction(2,0) from _swap+0x52[ap = ffffcd58]
_swap(2,0) from _main+0x3e[ap = ffffcd78]
_main(1,ffffcda8,ffffcdb0) from start+0x80[ap = ffffcd9c]
(adb)
```

Immediately after starting adb, you can view the stack backtrace from the core file by issuing the $c command. Because no processes are currently running in adb, there is no stack backtrace for the executable program; hence, the core file is used.

Once you start program execution, you can no longer view the stack backtrace from the core file.

Executing commands from a file

Occasionally you may want to execute a series of commands repeatedly while trying to debug a program, or perhaps you want to execute commands to bring the program to a known state before debugging interactively. The `$<` and `$<<` commands let you do this; you can store several debugging commands in a text file and execute them using one command. When executed, the `$<` and `$<<` commands get their input from a file instead of the keyboard.

The formats of the commands are as follows:

`$<` *file*

`$<<` *file*

You can include any `adb` commands in *file*. A sample file might look like this:

```
_main:b
:r
$r
:s
```

When executed, this file sets a breakpoint at `_main`, starts program execution, displays the general registers when the breakpoint is reached, and then executes one machine instruction.

Both commands read the `adb` commands from the specified file and execute them. The difference between them is that the `$<<` command returns to the calling file for continued execution while the `$<` does not. The `$<<` command is usually used when calling an additional command file from within a command file.

Compare these two command files:

```
_main:b          _main:b
:r               :r
$< file2         $<< file2
_index?wt        _index?wt
```

If you were to execute the file on the left, the `_index?wt` command would never get executed because, when `adb` started executing the commands stored in file2, it would never return to this file. The file on the right, however, would execute `_index?wt` after executing the commands in file2.

Advanced adb use

6

This chapter explains some of the more advanced uses of adb, including:

- Core files
- Signal handling
- Kernel debugging
- Segment maps
- Internal variables

Core files

A core file is a copy of the memory image of a process that has terminated under abnormal conditions. Common reasons for abnormal termination include memory violations, illegal instructions, and floating-point exceptions. adb may be used with a core file to help determine why a process failed.

The following figures demonstrate how adb is used with a core file. adb is invoked with the *objfile* that caused the problem and the *corefile*; this means that the *corefile* can be debugged using the symbols found in *objfile*. The *corefile* can be debugged without the *objfile* by using "-" as the name of the *objfile*, but the lack of *objfile* symbols makes this task more difficult.

The first step is to invoke adb with the executable file (*objfile*) and corresponding core file and to use the $? command to display information about the *corefile* in Figure 52. The $? command shows that the program fails because of a bus error and that the failure occurs near the instruction at _main +0xe (the PC register in the general register set).

Figure 52 $? command

```
% adb example core
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $?
 job 0: no process
job 0: coredump of 'example', version 7.0.0.7
job 0: coredump created Fri Aug 12 10:17:15 1988
job 0/0: stopped by a bus error (inward ring address reference)
job 0/0: register display

pc=8000117e (_main+0xe)
ps=80108000 (C,XF,DZE,FE)
sp=ffffce64 a1=ffffce78 a2=ffffce8c a3=ffffcebc
a4=00000000 a5=ffffc9a4 ap=ffffce78 fp=ffffce64 s0=00000000e00000ff
s1=0000000000000000 s2=0000000000000000 s3=0000000000000000
s4=0000000080038838 s5=0000000080038180 s6=00000000e0000000
s7=0000000000000000
(adb)
```

The next step is to examine the instructions around the location of the failure using the /i command as shown in Figure 52. Here, the command repeats the /i command with a count of 1. Two instructions before the instruction referenced by the pc is a load indirect instruction (ld.b @_ptr,s0) through a zero pointer. This instruction causes the failure. By looking in the source code, you can determine why the pointer is zero and fix the problem.

Figure 53 /i command

```
(adb) _main/i
_main:   sub.w #0,a0
(adb)RETURN
 _main+0x2: ld.b @_ptr,s0
(adb)RETURN
 _main+0x8: st.b s0,_ch
(adb) RETURN
_main+0xe: ld.b _ch,s0
(adb) _ptr/wx
_ptr:   0
(adb) $q%
```

The $c command is also useful in debugging core files. It displays the stack backtrace from the core file. The problem in the previous example occurs in the _main routine, so $c is not used.

Figure 52 illustrates how the $c command can help you isolate a problem. The first step is to invoke adb and look at the state of the core file.

Figure 54 Using $? to look at the state of a core file

```
% adb example core
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $?
job 0: no process
job 0: coredump of 'example', version 7.0.0.7
job 0: coredump created Wed Aug 17 15:13:18 1988
job 0/0: stopped by a bus error (inward ring address reference)
job 0/0: register display
pc=8000213c (_linkread+0x15c)
ps=80908000 (C,XF,SC,DZE,FE)
sp=ffffbf8c a1=00000000 a2=0000000b a3=ffffc010
a4=00000000 a5=ffffc9a4 ap=ffffbfb4 fp=ffffbfa0
s0=0000000000000006 s1=0000000000000000 s2=0000000000000000
s3=00000000ffffffff
s4=0000000000000007 s5=00000000ffffcec0 s6=00000000ffffc431
s7=00000000ffffceb9
(adb)
```

The next step is to look at the stack backtrace of the core file, as shown in Figure 52.

Figure 55 Using `$c` to look at a stack backtrace

```
(adb) $c
_linkread(ffffc42c,ffffc010,400) from _name1+0x7ea [ap = ffffbfb4]
_name1(ffffc42c,ffffceb9) from _name+0xa2 [ap = ffffc424]
_name(ffffceb4,ffffc949) from _main+0x19a [ap = ffffc940]
_main(0,ffffce78,ffffce7c) from start+0x80 [ap = ffffce64]
(adb) 80002130?i _linkread+0x150: sub.w a1,a1
(adb)RETURN
 _linkread+0x152:st.w a1,-4(fp)
(adb)RETURN
_linkread+0x156:ld.b@-4(fp),s0
(adb)RETURN
_linkread+0x15a:cvtb.ws0,s0
(adb)RETURN
_linkread+0x15c:psh.ws0
(adb)
```

The stack backtrace reveals that the program fails in the `_linkread` routine, which is called from `_name1`, which is called from `_name`, which is called from `_main`.

By looking at the instructions surrounding the PC, you can determine that the program fails at the `ld.b @-4(fp),s0` instruction.

For more information concerning stack backtraces, refer to the section titled "Interpreting stack backtraces" on page 89.

Signal handling

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

If a signal is sent to a program that is being debugged, the program stops, and the signal is actually delivered to `adb`. When the signal is received, `adb` displays the reason for the stoppage and enters command mode. When the program is single-stepped or continued, the signal is passed on to the program by `adb`.

A different signal may be passed to a debugged program using the same single-step and continue commands:

`:s` *signal*

`:c` *signal*

signal is the number of the signal to be passed (refer to the sigvec (2) man page). Signal handling can be particularly useful when testing interrupt-handling routines. If a signal should not be passed to a program, single-step or continue it with a signal of 0. The signal number is interpreted according to the current input radix.

When you pass a signal using the `:s` command, it can appear as if the signal does not get passed to the program because of the way the ConvexOS kernel handles signals. When the kernel receives one or more signals, it starts with signal 1 and moves sequentially through the rest of the signals until it finds a match with the signal passed with the command; then, it handles the signal.

When you issue a `:s` command, the command itself generates a signal (SIGTRAP, 5). If a signal is passed with the `:s` command, both the SIGTRAP signal and the passed signal are sent to the kernel. Because most signal numbers are greater than 5, the kernel handles the SIGTRAP signal to stop program execution after a single instruction. So, as long as you single step, the signal you send with a `:s` command does not get passed to the program.

The C program shown in Figure 52 acts as a signal handler. If at any time the program receives a SIGTERM signal (signal 15) or a SIGUSR1 signal (signal 30), it calls the appropriate subroutine. Because the program executes an infinite loop, you can stop it by typing **CTRL-c**.

Figure 56 Sample signal handler program in C

```
#include <signal.h>

int i;

sigusr1()
{
        printf("received signal SIGUSR1; i =%d n", i);
        return;
}

sigterm()
{
        printf("received signal SIGTERM; i =%d n", i);
        return;
}
main()

{
        i = 0;
        signal(SIGUSR1, sigusr1); /* call sigusr1 on a SIGUSR1 (30) */
        signal(SIGTERM, sigterm); /* call sigterm on a SIGTERM (15) */
        for (;;) {        /* loop until INTERRUPT */
                ++i;
                if (i > 32767)   /* keep from exceeding MAXINT */
                        i = 0;
        }
}
```

The following figures demonstrate how to use the `:c` and `:s` commands to change a signal before passing it to the executable program. In each of these figures, a ^C indicates that a **CTRL-c** was typed while the program was executing.

In Figure 52, `adb` is invoked and execution is started with the `:r` command. While the program is running, a **CTRL-c** is typed and execution is suspended. The `:c` command then continues execution without changing the signal. When the executable program receives the SIGINT signal, the program terminates.

Figure 57 Using `:r` to continue execution

```
% adb signal
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) :r
job 0: running
^Cjob 0/0: stopped by an interrupt
job 0/0: stopped at _main+0x5c: lt.w #32767,s0
(adb) :c
job 0: running
job 0: terminated
job 0/0: stopped by an interrupt all processes
terminated
(adb)
```

In Figure 52, the same program is rerun and stopped in the same way. This time the `:c` command continues execution, but changes the signal to a SIGUSR1 signal. The appropriate subroutine is called, and execution continues until stopped by another interrupt.

Figure 58 Using `:r` to continue execution

```
(adb) :r
job 0: running
^Cjob 0/0: stopped by an interrupt
job 0/0: stopped at _main+0x56: ld.w _i,s0
(adb) :c 0t30
job 0: running received signal SIGUSR1; i = 19210
^Cjob 0/0: stopped by an interrupt
job 0/0: stopped at _main+0x5c: lt.w #32767,s0
(adb)
```

You can also instruct `adb` to continue execution without passing any signal to the program. The program then executes as if the signal never occurred. This is illustrated in Figure 52.

Figure 59 Using `:r` after execution resumes

```
(adb) :r
job 0: running
^Cjob 0/0: stopped by an interrupt
job 0/0: stopped at _main+0x50: st.w s0,_i
(adb) :c 0
job 0: running
^Cjob 0/0: stopped by an interrupt
job 0/0: stopped at _main+0x56: ld.w _i,s0
(adb)
```

Kernel debugging

adb may be used to examine and modify the ConvexOS kernel while it is running. adb must first be invoked in kernel mode, as shown in Figure 52.

Figure 60 Invoking adb in kernel mode

```
% adb -k /vmunix /dev/mem
sdr[0] = 8014f000 sdr[1] = 00000000 sdr[2] = 00000000 sdr[3] = 8014f200
sdr[4] = 00000000 sdr[5] = 00000000 sdr[6] = 00000000 sdr[7] = 00000000
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb)
```

The -k option invokes kernel mode. Because /vmunix is used only to access the symbol names, /dev/mem should be treated as the running copy of the program and "/" debugging commands should be used instead of "?" commands. The eight sets of hexadecimal numbers displayed in response to this command are the segment descriptor registers (sdr) for the kernel. See the *CONVEX Architecture Reference Manual (C Series)*[1] for further information about them.

One adb command is useful in changing kernel debug state, namely

[*address*][, *thread*] **$k**

where:

address Address of a new set of segment description registers (sdr) that are to be used to remap the kernel

thread Specifies which thread memory is to be examined.

Debugging the kernel is mostly useful for examining and patching kernel variables and data structures, mainly because none of the ":" commands work when debugging the kernel.

[1]Order number DHW-300

Segment mapping

ConvexOS has only one executable file format and core file format. These formats may contain several different types of segments, including text, data, and bss (block storage segment). adb processes the segments in these files on start-up and provides access to them through a series of segment maps. To display the maps, use the following command:

```
(adb) $m
```

A typical map listing for an object and a core file is shown in Figure 52.

Figure 61 Using `$m` to produce a map listing

```
(adb) $m
job 0: address map display
? map       'a.out'
b0 = 80001000 e0 = 8000a000 f0 = 00001000 r x text
b1 = 8000a000 e1 = 8000d000 f1 = 0000a000 rw data
/ map       'core'
b0 = 80000000  e0 = 80001000  f0 = 00006000  rw  data
b1 = 80001000  e1 = 8000a000  f1 = 00007000  r x text
b2 = 8000a000  e2 = 8000d000  f2 = 00010000  rw  data
b3 = 8000d000  e3 = 8000e000  f3 = 00013000  rw  bss
b4 = ffffb000  e4 = ffffd000  f4 = 00014000  rw  bss
b5 = ffffd000  e5 = ffffe000  f5 = 00016000  r x text
(adb)
```

Each line in the map listing consists of five fields, namely

- b—Beginning address.
- e—Ending address.
- f—Offset in the file of the segment.
- Permissions that the segment has when loaded.
- Type of segment (text, data, or bss) into memory.

Two initial settings of mappings are made, one for *objfile* ("?") and one for *corefile* ("/"). Both mappings are suitable for normal files. If either file is not of the kind expected, then for that file, b is set to 0, e is set to the maximum file size, and f is set to 0; in this way, the whole file may be examined with no address translation. Otherwise, given an address, A, adb calculates the location of the address in the file (or in memory) as shown:

$$b < A < e => \text{address } (A - b) + f$$

If A does not fall into one of the segment ranges, it is not considered a legal address and an error occurs.

Any of the segment maps may be modified using the command:

[?/] **m** *n b e f*

n is an integer that indicates the segment map to be modified and is required. If less than three expressions are given, then the remaining map parameters are left unchanged.

Internal variables

adb maintains a set of 36 internal variables, named 0 through 9, and a through z. Each variable is a signed, 64-bit integer. These variables are not used by adb; they are provided as "scratchpad" space for adb scripts (such as may be processed by a $< command), but you can use them whenever you desire. Some of these variables are initialized by adb upon start-up from *corefile*. If *corefile* is not specified or does not appear to be a default core file (core), then these values are set from *objfile*, as listed in Table 21.

Table 21
Internal variables

Value	Purpose
b	Base address of the data segment
d	Size of the data segment
e	Entry point
m	Magic number
s	Size of the stack segment
t	Size of the text segment

Other variables change from command to command. These are:

0 Last value printed

9 Count on the last $< or $<< command

Other variables may be used for any desired purpose. The values of these variables may be used in the address field of a command, such as:

```
<b
```

This command is evaluated as the base address of the data segment. The value of a variable may be changed by an assignment request such as:

```
0o2000>b
```

This command sets b to octal 2000.

Multithreaded debugging 7

This chapter explains how to debug multithreaded code. CONVEX multiprocessor systems allow you to create and execute parallel programs by providing support for multiple streams of execution within a single process. Each stream, or thread, executes independently of other instruction streams.

Topics covered in this chapter are:

- Multithreaded programs
- `adb` commands that control and debug multithreaded programs
- Debugging a multithreaded program

What is a multithreaded program?

A *multithreaded program* is an application in which a task is subdivided and solved by several independent streams of execution running 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 processors; many threads can run on one processor or on many processors.

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

Terminology

Common terms used to discuss multithreaded programs are listed in Table 22.

Table 22
Terminology used in multithreaded programs

Term	Description
CPU	One physical processing unit. Each CPU in the configuration operates independently as a 64-bit CONVEX supercomputer.
Multiprocessor	A CONVEX supercomputer consisting of memory system, I/O system, peripherals, and one or more CPUs.
Complex	The entire set of physical CPUs in the system configuration.
Subcomplex	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 executing at the same time in a process can never exceed the number of CPUs within the processor configuration.
Mutual exclusion	A protocol that guarantees that only one thread obtains access to a resource at a given time.
Hardware communication register	A high-speed register set used for communication between the threads of a process. Threads communicate by sending and receiving data through the hardware communication registers. A hardware-maintained lock bit is associated with each hardware communication register; the lock bit guarantees mutually exclusive access to the register.

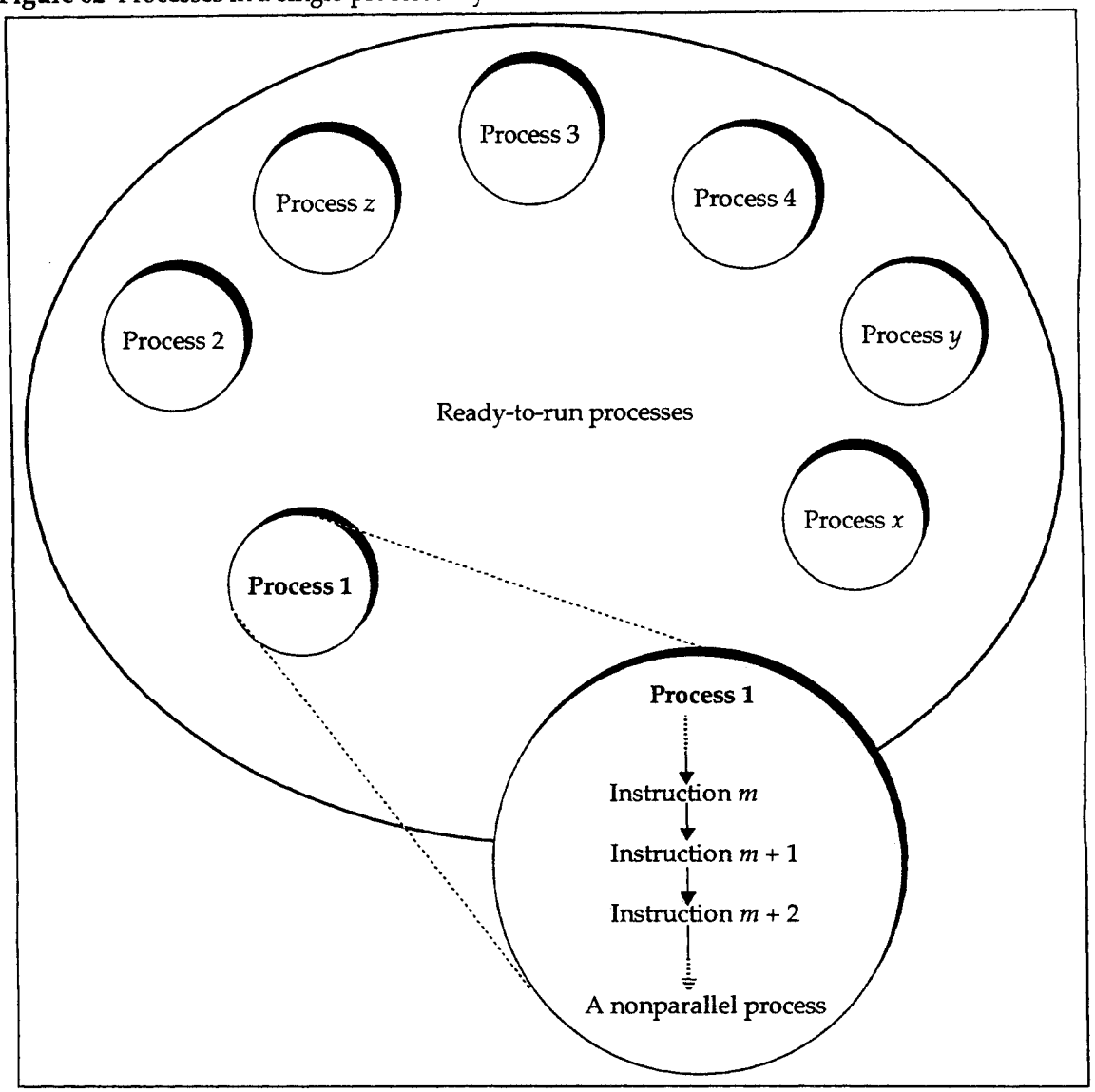
Multiple threads

Multithreaded applications are radically different from standard, nonparallel programs. To effectively debug multithreaded code, you must understand how multithreaded programs execute in the 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.

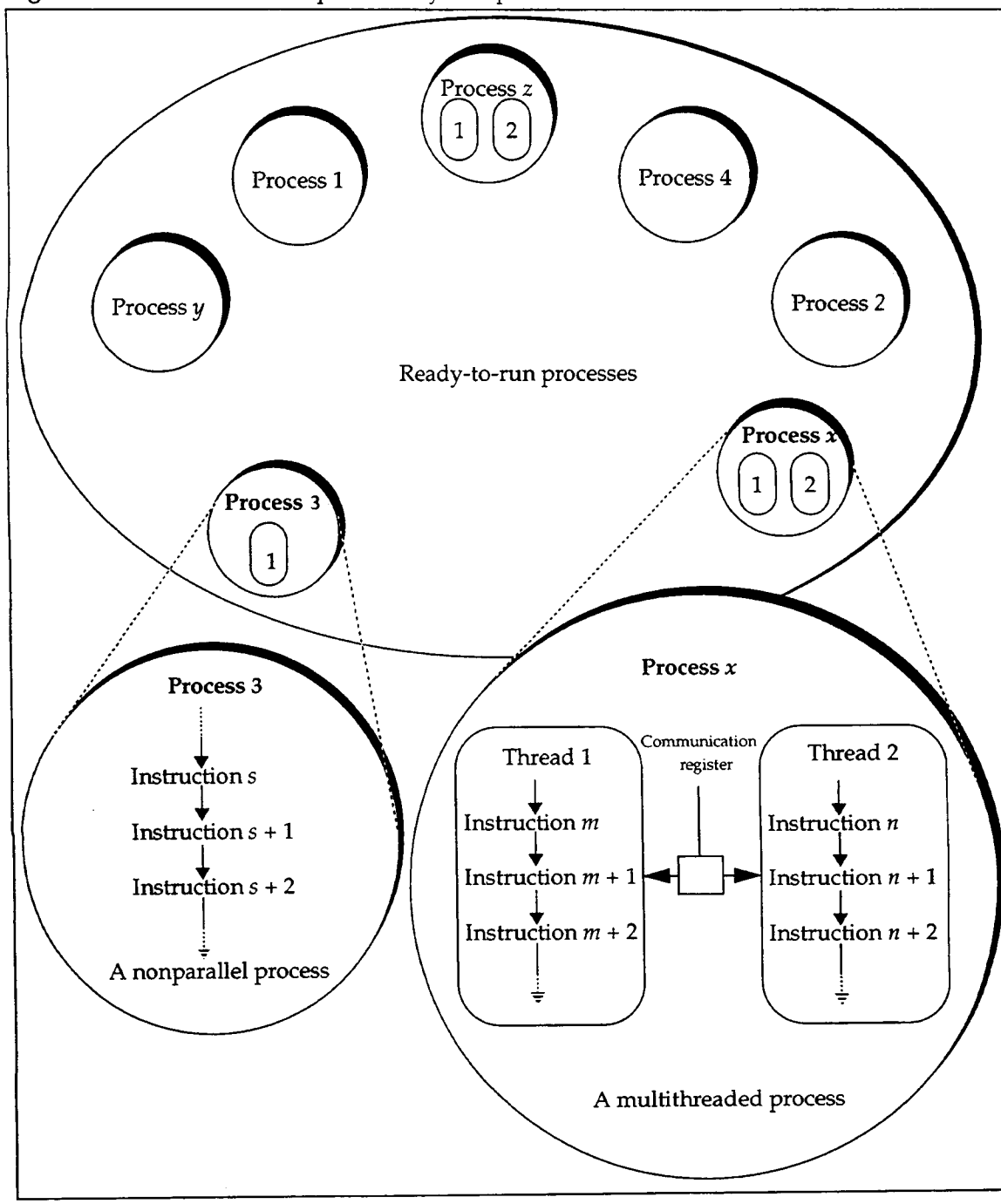
Figure 62 shows an overview of a single-processor system. The single-processor system maintains a set of ready-to-run processes; a process is an instance of a running program. Each process has one stream of execution as shown in the detail of Process 1. The state of process 1 at instruction *m* is defined by the set of instructions executed prior to instruction *m*.

Figure 62 Processes in a single-processor system



A multiprocessor system supports multiple streams of execution. Programs execute 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 63 displays an overview of a multiprocessor system.

Figure 63 Processes in a multiprocessor system processes



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*, and so on.
- Thread 2 is executing the sequence of instructions *n*, *n+1*, *n+2*, and so on.

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 hardware communication registers (represented in Figure 59 by the small box). Hardware 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 of the hardware communication registers.

Figure 59 displays other processes 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 and control the way the threads interact.

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)
- Program 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

As was explained earlier, each thread executes independently of and in parallel with other threads in the same process. Threads must synchronize or exchange information among themselves. Hardware communication registers are the medium for communication between threads of the same process. Within a process there are one hundred twently-eight 64-bit hardware communication registers; all threads within a process share the same hardware communication register set.

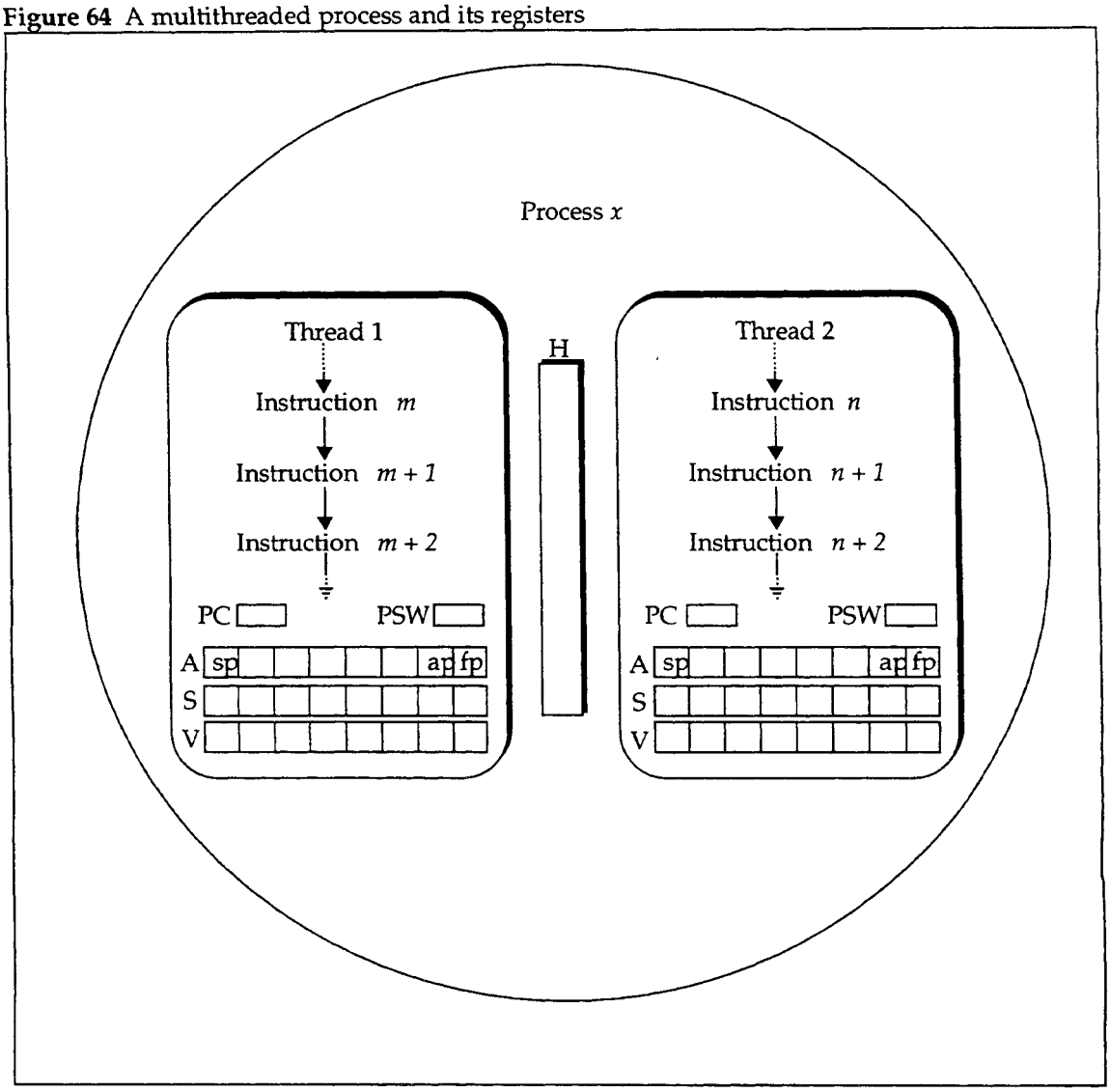
The 128 hardware communication registers are divided as follows:

- 32 registers reserved for use by the hardware
- 32 registers reserved for use by operating system reserved registers
- 64 registers accessible by user programs; user-accessible registers are referenced by addresses 0x8000 through 0x803f (the first register is at location 0x8000).

Figure 64 shows the detail of Process *x* from Figure 63 with all registers drawn. The array of boxes labeled H represents the process' user-accessible hardware communication registers. Each thread has its own program counter, status word, and sets of address, scalar, and vector registers, but hardware 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 addresses into thread-specific physical addresses. Two thread-private memory segments will never map onto the same physical addresses.

Figure 64 A multithreaded process and its registers



Multithreaded debugging commands

adb lets you control multithreaded programs by supporting thread-specific commands. You can set process breakpoints, monitor individual threads, alter hardware communication registers, alter thread memory and thread-specific registers, and display thread stack backtraces.

Jobs and threads

In adb there are two basic entities to control: *jobs* and *threads*. An adb job is an envelope enclosing one process. A job contains all of the threads in a process and all the information needed to control and manage that process. Job information includes a list of breakpoints, the names of the files being debugged, and the process' unique process ID.

Running a job under adb, one thread is always the current thread. Commands always apply to the current thread; the current thread can be changed to any other thread using an appropriate adb command. The current thread is denoted by an asterisk in adb output displays.

Groups of commands

The set of adb commands that are used in debugging multithreaded programs can be divided into five groups:

- Commands that apply to the current thread
- Commands that apply to all threads
- Command that apply to a job
- Commands that affect data display
- Commands that change internal adb modes

These commands are discussed, by group, in the following sections.

Commands that apply to the current thread

Commands that apply to the current thread are listed in Table 23. Full descriptions of each command can be found in the adb(1) man page or in Chapter 9, "adb command reference" on page 153.

Table 23
Commands that apply to the current thread

Command	Description
`$a=`	Modify value of an address register
`$a?`	Display contents of an address register
`$c`	Display a call-stack backtrace
`$h=`	Assign value to a hardware communication register
`$h?`	Display contents of a hardware communication register
`$j`	Display status of all threads
`$r`	Display contents of general registers
`$s=`	Modify a scalar register
`$s?`	Display contents of a scalar register
`$v=`	Modify a vector register
`$v?`	Display contents of a vector register
`:c`	Continue execution of current thread
`:s`	Execute a single instruction in current thread

Commands that apply to all threads in a job

Table 24 lists commands that apply to all threads in a job. Full descriptions of each command can be found in the adb(1) man page or in Chapter 9, "adb command reference" on page 153.

Table 24
Commands that apply to all threads in a job

Command	Description
`$b`	Display currently set breakpoints
`$j`	Display status of all threads
`$h=`	Assign value to a hardware communication register
`$h?`	Display registers for all threads in current job
`:b`	Set a breakpoint in all threads
`:d`	Delete a breakpoint from all threads
`:e`	Change the exit disposition of current job to RELEASE
`:f`	Make a thread current
`:k`	Kill current job
`:C`	Continue execution for all threads in current job
`:D`	Delete all currently set breakpoints
`:S`	Execute one instruction for each thread in current job

Commands that apply to a job

Commands that apply to a job are listed inTable 25. Full descriptions of each command can be found in the adb(1) man page or in Chapter 9, "adb command reference" on page 153.

Table 25
Commands that apply to a job

Commands	Descriptions
`$?`	Display process ID, the signal that caused job stoppage, and all thread register sets as with `$R`
`:i`	Toggle inherit mode
`:k`	Kill current job
`:r`	Run *objfile* as a process

Commands that change or display program data

Commands that change or display program data are listed in Table 26. Full descriptions of each command can be found in the adb(1) man page or in Chapter 9, "adb command reference" on page 153.

Table 26
Commands that change or display program data

Command	Description
`$e`	Display the names and values of all external variables
`?modifier`	Display locations in the object file according to modifier
`/modifier`	Display locations in the core file according to modifier
`=modifier`	Display a value according to modifier

Commands that change or display the adb environment

Table 27 lists commands that change or display the adb environment. Full descriptions of each command can be found in the adb(1) man page or in Chapter 9, "adb command reference" on page 153.

Table 27
Commands that change or display the adb environment

Command	Description
$f	Display or set floating-point format
$g	Get new symbol or core file names
$i	Display names and values of all nonzero internal variables
$l	Set limit for symbol matches
$m	Display address maps
$n	Display or set maximum number of processors allocated for a process
$o	Toggle operating modes between chained and sequential
$q	Terminate debugging session
$t	Search for a symbol in the current job's symbol table
$x	Modify default input radix
$X	Modify default output radix
)help	Display a help file containing information about adb
)status	Display status of various adb modes, including current input radix, current output radix, inherit mode, and operating mode

Example of multithreaded debugging

The example C program in this section sums the elements of an array of integers. To solve the problem in parallel, the program divides itself into multiple threads. Each thread reads elements from the array and adds the elements to a common sum. Access to the array is controlled by a shared array index resource; the common sum is another shared resource.

The program uses hardware communication registers for shared resources. The CONVEX multiprocessor system provides special assembly-language instructions to ensure that, *at any* given time, only one thread is accessing or altering a hardware communication register.

A listing of the C main program is shown in Figure 65. A listing of the hand-coded assembly-language subprogram is shown in Figure 65.

Figure 65 The C main program

```
#include <stdio.h>

#define ASIZE  1000000

int array[ASIZE];
int threadcnt;
long long psum = 0LL;

/*
* main() - initialize an array of integers and call subroutine
* sum_in_parallel to compute the sum of the array elements in
* parallel.
*/
main()
{       int i;
        int sum_in_parallel();
        /* initialize the array */
        for (i = 0; i < ASIZE; i++) {
              array[i] = 1;
        }
        /* sum in parallel */
        threadcnt = sum_in_parallel(array, ASIZE, &psum);
        /* print answer */
        printf("%d threads: psum =%ll n", threadcnt, psum);

        exit(0);
}
```

Figure 66 Hand-coded `sum_in_parallel` assembly-language subprogram

```
;
; sum_in_parallel(a,n,asum): compute "asum = sum of n elements in array a."
;
; sum_in_parallel spawns as many parallel threads as possible. each
; thread reads elements from the array - access is controlled via a
; common index counter maintained in a communication register. each
; element of the array is added to a global sum also maintained in a
; communication register.
;
; when all elements have been processed, the threads execute a join
; so that the sum_in_parallel function returns to the caller single threaded
; again.
;
; sum_in_parallel returns the sum in argument asum and returns to the
; calling program the number of threads spawned.
;
;
; communication register usage:
; registers x8000 - x801f are reserved by the compiler and runtime systems.
; register INDEX (0x8020) will be used as the array index - it is a
;resource - the hardware controls access to it.
; register TOTAL (0x8021) will be used as the total - it is a resource -
;the hardware will control access to it.
; register THREADS (0x8022) will be used to count the number of threads
;spawned.
;

INDEX   = 0x8020
TOTAL   = 0x8021
THREADS = 0x8022

              .globl              _sum_in_parallel
              .globl              _thread_proc
              .text
_sum_in_parallel:
                                  ; initialize TOTAL
              ulk TOTAL           ;clear lock bit
              sub.l s0,s0         ;s0 = 0;
              snd.l s0,TOTAL      ; TOTAL = 0, set lock bit
                                  ; initialize THREADS
              ulk THREADS         ;clear lock bit
              snd.l s0,THREADS    ;THREADS = 0, set lock bit
                                  ; initialize INDEX
              ulk INDEX           ;clear lock bit
              ld.w #-1,a1         ; a1 = -1
              snd.w a1,INDEX      ; place a -1 in INDEX, set lock bit
              spawn_thread_proc,fp; spawn other threads
```

Figure 66 (Continued)
Hand-coded `sum_in_parallel` assembly-language subprogram

```
_thread_proc:                   ; each thread executes this procedure
                                ; add one to the thread count
          ld.w #1,a1            ;a1 = 1
          inc.w THREADS,a1      ;atomic "THREADS=THREADS+1; a1=THREADS"
          bra.f _thread_proc    ; if access to THREADS failed, try again
          ld.w  8(ap),a1        ; a1 = address of result
          ld.w 4(ap),a2         ; a2 = n
          ld.w 0(ap),a3         ; a3 = array address

L3:                             ; compute next index, place the index in a4
          ld.w #1,a4            ;    a4 = 1
          inc.w INDEX,a4        ;atomic "INDEX=INDEX+a4; a4=INDEX"
          bra.f L3              ; if access to INDEX failed, try again le.w
          a4,a2                 ; access successful
          jbra.f L5             ; if INDEX > n, then (branch to exit)
          mul.w  #4,a4;         ; a4 = 4 * a4 (where 4 is the size of an
                                array element
          add.w                 a3,a4; a4 = a3 + a4, the address of the int

L4:       ld.w                  (a4),s0; s0 = (a4)
          inc.l TOTAL,s0        ; TOTAL = TOTAL + (s0)
          brs.f L4              ; if access to TOTAL failed, try again
          br L3                 ; restart the loop
L5:
          join                  ; now do joins until single threaded again
          get.l  TOTAL,s0       ; s0 = TOTAL
          st.l  s0,(a1)         ; asum = s0
          get.l  THREADS,s0     ; s0 = THREADS (return the number of threads)
rtn                             ; will restore correct sp for calling routine
```

The C main program creates an array of integers and calls `sum_in_parallel (array, ASIZE, &psum)`, where `array` is an array of integers, *ASIZE* is the number of elements in the array, and `&psum` is the address where the result should be stored. The assembly-language subprogram `sum_in_parallel` computes the sum in parallel and returns the number of threads used in the computation.

The assembly-language subprogram uses the spawn instruction to divide the problem into as many threads as there are processors available. The `spawn` instruction has two arguments: an address and an address register. The address is used as the initial program counter (pc) of each thread; the contents of the address register (typically the frame pointer register) are used as the initial stack pointer (sp) for each thread.

The effect of executing the assembly instruction `spawn _thread_proc, fp` is shown in Figure 67.

Figure 67 The effects of executing instruction `spawn _thread_proc, fp`

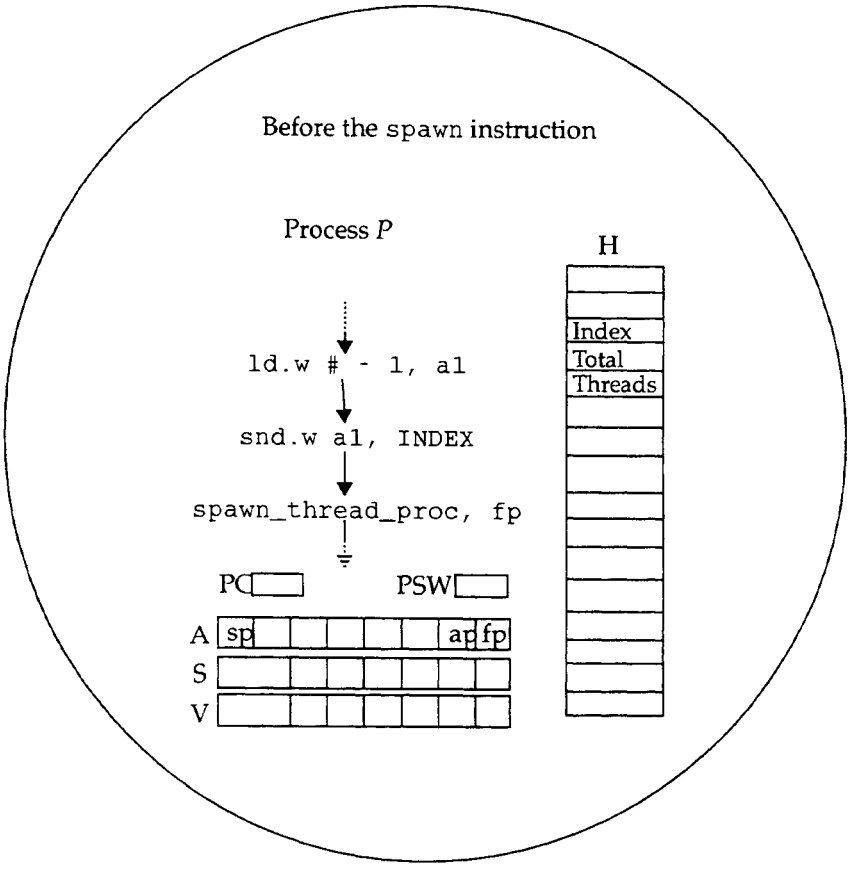
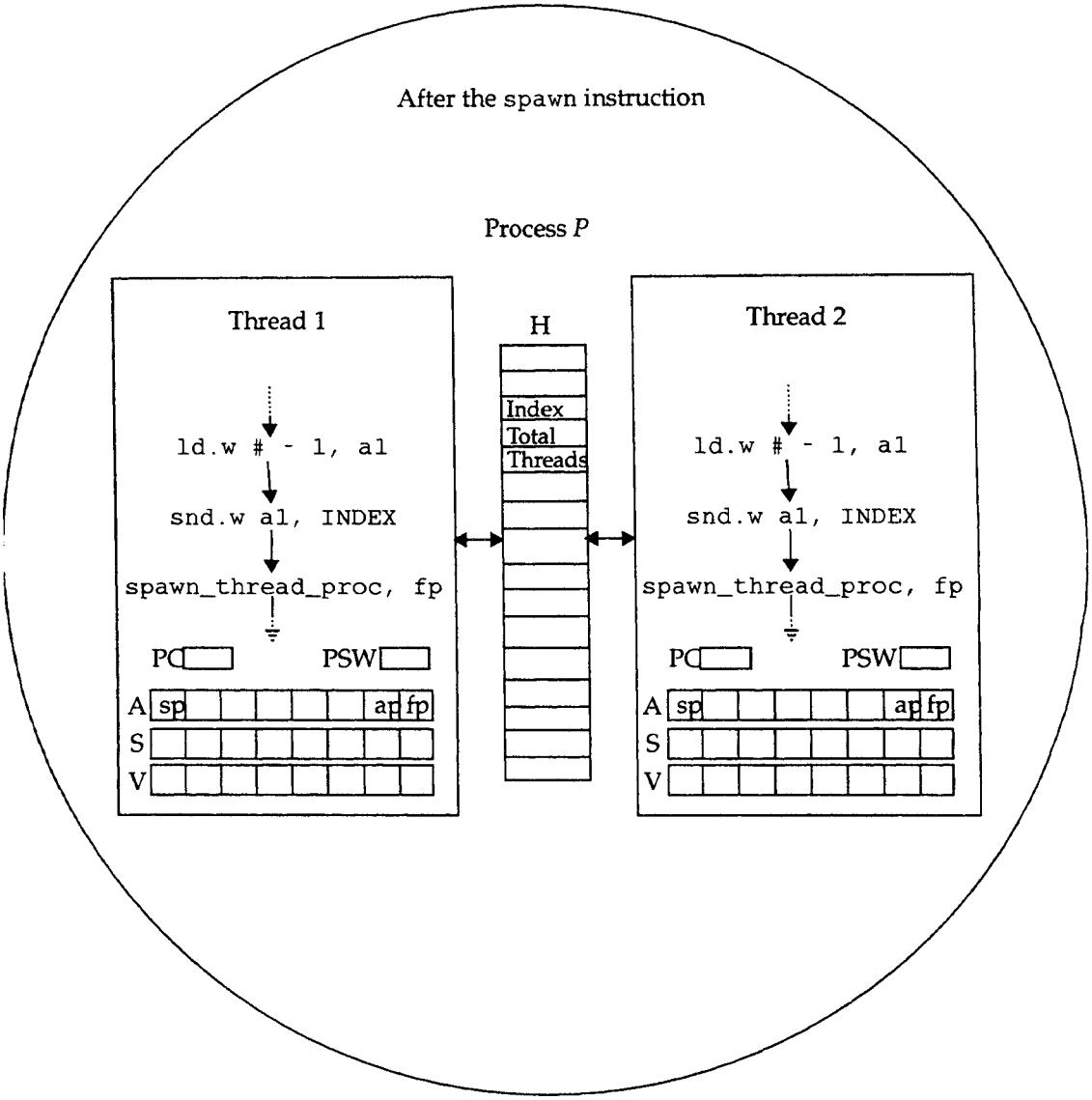


Figure 67 (Continued) The effects of executing instruction `spawn _thread_proc, fp`



Example adb multithreaded debugging session

This section presents an example of adb commands and output. The program segments presented in Figure 65 and Figure 65 are used in this example.

Step 1 Create the executable using the cc compiler and the assembler.

```
% cc -c main.c
% as suminparallel.s
% cc suminparallel.o main.o
```

Step 2 Invoke adb to debug the file a.out.

```
% adb a.out
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb)
```

Step 3 Display the status of important modes in adb using the)status command.

```
(adb) )status
inherit mode: OFF
operating mode: sequential
exit disposition: terminate
input radix: 16
output radix: default
static symbols: NO
maximum processors: 2
help file: /usr/lib/adb/helpfile
(adb)
```

Step 4 Set a breakpoint at the beginning of procedure sum_in_parallel using the :b command. Then display all breakpoints set in the job with the $b command.

```
(adb) _sum_in_parallel:b
(adb) $b
job 0: breakpoint display
count   bkpt     command
1       _sum_in_parallel
(adb)
```

Step 5 Run the program. Use `:r`.

```
(adb) :r
job 0: running
job 0/0: breakpoint _sum_in_parallel: ulk 0x8021
(adb)
```

Step 6 Display the status of all threads with the `$j` command.

```
(adb) $j
job     pid     c       state
---     -----   -       ------------------------
0       15234   *       active, sequential, terminate on quit, native fpmode
                *       thread 0: at breakpoint _sum_in_parallel
                        file '?': a.out
                        file '/': core
(adb)
```

Program execution has stopped at the breakpoint set at location `_sum_in_parallel`.

Step 7 Display the call stack backtrace with the `$c` command.

```
(adb) $c
_sum_in_parallel (80007808, 186a0, 80069288) from _main+0x54 [ap = ffffcd18]
_main(1, ffffcd48, ffffcd50) from start+0x80 [ap = ffffcd3c]
(adb)
```

Step 8 Single step the current thread. Use `:s`.

```
(adb):s
job 0: running
job 0/0: stopped at _sum_in_parallel+0x8:    sub.l s0,s0
(adb):s
job 0: running
job 0/0: stopped at _sum_in_parallel+0xa:    snd.l s0,0x8021
(adb):s
job 0: running
job 0/0: stopped at _sum_in_parallel+0x12:   snd.l s0,0x8022
(adb):s
job 0/0: stopped at _sum_in_parallel+0x1a:   ulk   0x8020
(adb):s
job 0: running
job 0/0: stopped at _sum_in_parallel+0x22:   ld.w #65535,a1
(adb):s
job 0: running
job 0/0: stopped at _sum_in_parallel+0x26:   snd.w a1,0x8020
(adb):s
job 0: running
job 0/0: stopped at _sum_in_parallel+0x2e:   spawn _thread_proc,a7
(adb)
```

Step 9 Display the general registers with the `$r` command; display the `INDEX`, `TOTAL` and `THREADS` hardware communication registers with the `$h` command.

```
(adb) $r
job 0/0: register display

pc=8000122e (_sum_in_parallel+0x2e)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=ffffffff a2=ffffcd50 a3=ffffcda8 tt=001fdb27
a4=00000000 a5=00000030 ap=ffffcd18 fp=ffffcd04
s0=0000000000000000 s1=0000000003d09000 s2=0000000000000000 s3=00000000ffffffff
s4=00000000fffffffc s5=0000000000000080 s6=00000000e0000000 s7=0000000000060000

(adb) $h20?wx
job 0: communication register display

h[32]=ffffffff (1)
(adb) $h21?lx
job 0: communication register display

h[33]=0000000000000000 (1)
(adb) $h22?lx
job 0: communication register display

h[34]=0000000000000000 (1)
(adb)
```

In `adb`, hardware communication registers are specified by their offset from 0x8000, the base address of all hardware communication registers. The `INDEX` hardware communication register is located at address 0x8020, and its offset is 20 base 16 or 32 base 10 (0x8020 through 0x8000).

The command `$h20?lx`, shown above, instructs `adb` to print the `INDEX` hardware communication register as a long hexadecimal value.

Step 10 List eight instructions starting at the address `_thread_proc`.

```
(adb) _thread_proc,0x8?ia
_thread_proc:       ld.w #1,a1
_thread_proc+0x2:   inc.w 0x8022,a1
_thread_proc+0xa:   bra.f _thread_proc
_thread_proc+0xc:   ld.w 8(ap),a1
_thread_proc+0x10:  ld.w 4(ap),a2
_thread_proc+0x14:  ld.w 0(ap),a3
_thread_proc+0x18:  ld.w #1,a4
_thread_proc+0x1a:  inc.w 0x8020,a4
_thread_proc+0x22:
(adb)
```

The output shows the address of the instruction and the instruction itself. `0x8022` is the address of the `THREADS` hardware communication register; `0x8020` is the address of the `INDEX` hardware communication registers.

Step 11 Set a breakpoint, using `:b`, at the start of procedure `_thread_proc`, then list all breakpoints set in the current job, with `$b`.

```
(adb) _thread_proc:b
(adb) $b
job 0: breakpoint display
count   bkpt     command
1       _thread_proc
1       _sum_in_parallel
(adb)
```

Step 12 Continue execution of the program with the `:c` command. The next breakpoint in the process is at `_thread_proc`. The assembly-language instruction `_spawn_thread_proc,fp` adds additional threads to the process.

```
(adb) :c
job 0: running
job 0/0: breakpoint _thread_proc:       ld.w #1,a1
job 0/1: breakpoint _thread_proc:       ld.w #1,a1
(adb)
```

The output shows that there are two threads: `job 0/0` and `job 0/1`. Each thread has stopped at the breakpoint at location `_thread_proc`.

All threads stop executing whenever any thread is stopped by a breakpoint or interrupt. Threads may stop at different locations.

Step 13 Display the status of all the threads.

```
(adb) $j
job     pid     c       state
---     -----   -       ------------------------
0       19980   *       active, sequential, terminate on quit, native fpmode
                *       thread 0: at breakpoint _thread_proc
                        thread 1: at breakpoint _thread_proc
                        file '?': a.out
(adb)
```

The column labeled c stands for current; the current thread is denoted by a * in this column. As shown, there are two threads: thread 0 is current and is stopped at breakpoint `_thread_proc`. Thread 1 has been added to the process and is also stopped at breakpoint `_thread_proc`. The line `file '?': a.out` specifies that the file associated with the "?" output display command is a.out.

Step 14 Single step the current thread with the `:s` command.

```
(adb):s
job 0: running
job 0/0: stopped at _thread_proc+0x2:          inc.w 0x8022,a1
(adb):s
job 0: running
job 0/0: stopped at _thread_proc+0xa:          bra.f _thread_proc
(adb):s
job 0: running
job 0/0: stopped at _thread_proc+0xc:          ld.w 8(ap),a1
(adb)
```

Step 15 Using `$h?`, display the `THREADS` hardware communication register.

```
(adb) $h22?wx
job 0: communication register display
h[34]=00000001 (1)
(adb)
```

The current thread has added 1 to the `THREADS` counter.

Step 16 Make thread 1 the current thread with the `:f` command.

```
(adb),1:f
job 0/1: current job/thread
(adb)
```

adb confirms that thread 1 is now the current thread.

Step 17 Show the register sets of all threads in the current job using the `$R` command.

```
(adb) $R
job 0/0: register display

pc=80001242 (_thread_proc+0xc)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=00000001 a2=ffffcd50 a3=ffffcda8 tt=00212366
a4=00000000 a5=00000030 ap=ffffcd18 fp=ffffcd04
s0=0000000000000000 s1=0000000003d09000 s2=0000000000000000 s3=0000000000000078
s4=00000000fffffffc s5=0000000000000080 s6=00000000e0000000 s7=0000000000060000

job 0/1: register display

pc=80001238 (_thread_proc+0x2)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=0000000a a2=03b50000 a3=000e0080 tt=00000007
a4=0001ffff a5=001cd5aa ap=ffffcd18 fp=ffffcd04
s0=0000000000000007 s1=0000000000000001 s2=0000000080001236 s3=ffffffffffffffff
s4=000000000000000f s5=0000000000000000 s6=0000000000000001 s7=0000000000000006
(adb)
```

Step 18 Single step the current thread (thread 1) using `:s`.

```
(adb):s
job 0: running
job 0/1: stopped at _thread_proc+0x2:       inc.w 0x8022,a1
(adb):s
job 0: running
job 0/1: stopped at _thread_proc+0xa:       bra.f _thread_proc
(adb):s
job 0: running job 0/1: stopped at _thread_proc+0xc:       ld.w 8(ap),a1
(adb)
```

Step 19 Using `$h?`, display the `THREADS` hardware communication register.

```
(adb) $h22?lx
job 0: communication register display
h[34]=0000000000000002 (1)
(adb)
```

Thread 1 has also added 1 to the `THREADS` counter.

Step 20 Using `$j`, display the status of all jobs and all threads.

```
(adb) $j
job     pid     c       state
---     -----   -       -------------------------
0       19980   *       active, sequential, terminate on quit, native fpmode
                        thread 0: stopped at _thread_proc+0xc
                *       thread 1: stopped at _thread_proc+0xc
                        file '?': a.out
(adb)
```

Step 21 Single step all threads in the current job with the `:S` command; repeat several times.

```
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x10:     ld.w 4(ap),a2
job 0/1: stopped at _thread_proc+0x10:     ld.w 4(ap),a2
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x14:     ld.w 0(ap),a3
job 0/1: stopped at _thread_proc+0x14:     ld.w 0(ap),a3
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x18:     ld.w #1,a4
job 0/1: stopped at _thread_proc+0x18:     ld.w #1,a4
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x1a:     inc.w 0x8020,a4
job 0/1: stopped at _thread_proc+0x1a:     inc.w 0x8020,a4
(adb):S
 job 0: running
job 0/0: stopped at _thread_proc+0x22:     bra.f _thread_proc+0x18
job 0/1: stopped at _thread_proc+0x22:     bra.f _thread_proc+0x18
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x24:     le.w a4,a2
job 0/1: stopped at _thread_proc+0x24:     le.w a4,a2
(adb)
```

Step 22 Display the registers of all threads in the job with the `$R` command.

```
(adb) $R
job 0/0: register display

pc=8000125a (_thread_proc+0x24)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=80005078 a2=000f4240 a3=80007808 tt=002123c9
a4=00000000 a5=00000030 ap=ffffcd18 fp=ffffcd04
s0=0000000000000000 s1=0000000003d09000 s2=0000000000000000 s3=0000000000000078
s4=00000000fffffffc s5=0000000000000080 s6=00000000e0000000 s7=0000000000060000

job 0/1: register display

pc=8000125a (_thread_proc+0x24)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=80005078 a2=000f4240 a3=80007808 tt=00000097
a4=00000001 a5=001cd5aa ap=ffffcd18 fp=ffffcd04
s0=0000000000000007 s1=0000000000000001 s2=0000000080001236 s3=ffffffffffffffff
s4=000000000000000f s5=0000000000000000 s6=0000000000000001 s7=0000000000000006
(adb)
```

The value of address register `a4` in thread 0 differs from the value of the same register in thread 1. In this iteration, Thread 0 will process element 0 (`a4=0`) and thread 1 will process element 1 (`a4=1`). The value of the `INDEX` hardware communication register at any time is the number of elements in the array already processed. At the start of each iteration, the threads add one to the `INDEX` register, record that value, and process that element.

Step 23 Single step all threads in the current job using `:S`.

```
(adb):S
 job 0: running
job 0/0: stopped at _thread_proc+0x26:     bra.f _thread_proc+0x3c
job 0/1: stopped at _thread_proc+0x26:     bra.f _thread_proc+0x3c
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x28:     mul.w #4,a4
job 0/1: stopped at _thread_proc+0x28:     mul.w #4,a4
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x2a:     add.w a3,a4
job 0/1: stopped at _thread_proc+0x2a:     add.w a3,a4
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x2c:     ld.w 0(a4),s0
job 0/1: stopped at _thread_proc+0x2c:     ld.w 0(a4),s0
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x30:     inc.l 0x8021,s0
job 0/1: stopped at _thread_proc+0x30:     inc.l 0x8021,s0
(adb)
```

Step 24 Using `$R`, display the registers of all threads in the current job.

```
(adb) $R
$R
job 0/0: register display

pc=80001266 (_thread_proc+0x30)
ps=03909080 (EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=80005078 a2=000f4240 a3=80007808 tt=00212420 a4=80007808 a5=00000030
ap=ffffcd18 fp=ffffcd04
s0=0000000000000001 s1=0000000003d09000 s2=0000000000000000 s3=0000000000000078
s4=00000000fffffffc s5=0000000000000080 s6=00000000e0000000 s7=0000000000060000

job 0/1: register display

pc=80001266 (_thread_proc+0x30)
ps=03909080 (EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd04 a1=80005078 a2=000f4240 a3=80007808 tt=000000e8 a4=8000780c a5=001cd5aa
ap=ffffcd18 fp=ffffcd04
s0=0000000000000001 s1=0000000000000001 s2=0000000080001236 s3=ffffffffffffffff
s4=000000000000000f s5=0000000000000000 s6=0000000000000001 s7=0000000000000006
(adb)
```

Step 25 Single step all threads in the job twice with `:S`.

```
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x38:      brs.f _thread_proc+0x2c
job 0/1: stopped at _thread_proc+0x38:      brs.f _thread_proc+0x2c
(adb):S
job 0: running
job 0/0: stopped at _thread_proc+0x3a:      br    _thread_proc+0x18
job 0/1: stopped at _thread_proc+0x3a:      br    _thread_proc+0x18
(adb)
```

Step 26 Using `$h?`, display the TOTAL hardware communication register.

```
(adb) $h2l?lx
job 0: communication register display
h[33]=0000000000000002 (1)
(adb)
```

Step 27 Continue execution of all threads in the job with the `:C` command; quit `adb` with `$q`.

```
(adb):C
job 0: running
2 threads: psum = 1000000
job 0: terminated
all processes terminated
(adb) $q
```

Multiprocess debugging

8

New processes are created using the `fork()` system call. `fork()` creates a new process by copying the executable image of the calling process. Using `fork()` and other system calls, you can create and manage multiple processes within their application.

Topics covered in this chapter are:

- The `fork()` operation and multiprocess programs
- Commands to control and debug multiprocess programs
- Debugging a multiprocess program

What is a multiprocess program?

A *multiprocess* program is an application that replicates its image during execution using the `fork()` system call. `fork()` creates a new process, called the child process, by copying the calling parent process.

A process is an instance of a running program. Each process exists and executes independently of all other processes in the system. At all times, ConvexOS maintains a list of runnable processes; newly created processes are added to the list, and terminated processes are removed.

The following section describes the components of a ConvexOS process.

Components of a ConvexOS process

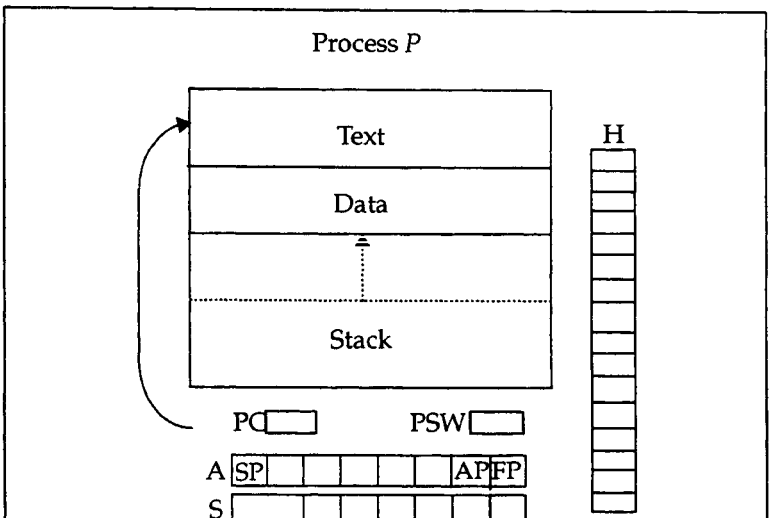
A ConvexOS process is composed of the following parts:

- A unique numeric *process identifier* or PID
- Program text. Program text is often shared and is write-protected to prevent accidental destruction of program instructions.
- A data segment that is readable and writable by the user. The data segment is neither shared with, nor accessible by, other users. This area begins where the program text ends and may be increased or decreased by the user using library routines such as `malloc()` or `free()`.
- An unshared stack segment that grows down from the top of memory. This area grows and shrinks in the course of program execution.
- A collection of one or more threads that share the text, data, and stack segments of the process. Threads are described in the previous chapter. To summarize, each thread is a stream of execution, can have its own thread-private memory, and has its own complement of machine registers:
  - Program counter register (PC)
  - Processor 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.
  - 128 hardware communication registers; each hardware communication register is 64 bits wide.
- By default, a process has only one thread of execution.

The text, data, and stack segments of a process are collectively referred to as the *address space* of the process. The address space of each process is distinct from that of other processes within the system.

Figure 68 shows the components of a ConvexOS process, *P*. The program counter register (PC) points into the text segment at some instruction in the instruction sequence.

Figure 68 Example ConvexOS process



P is the process' PID. The process' stack begins at a high address in memory and grows toward lower addresses; the `dotted` arrow indicates the direction of growth. Process *P* is single-threaded; it has one set of machine registers, which includes the PC, the PSW, address registers, scalar registers, vector registers, and the hardware communication registers. (If a process has multiple threads, the PC, PSW, address, scalar and vector registers are replicated, one copy per thread. Refer to Figure 64, "A multithreaded process and its registers," in Chapter 7.)

`fork()` system call

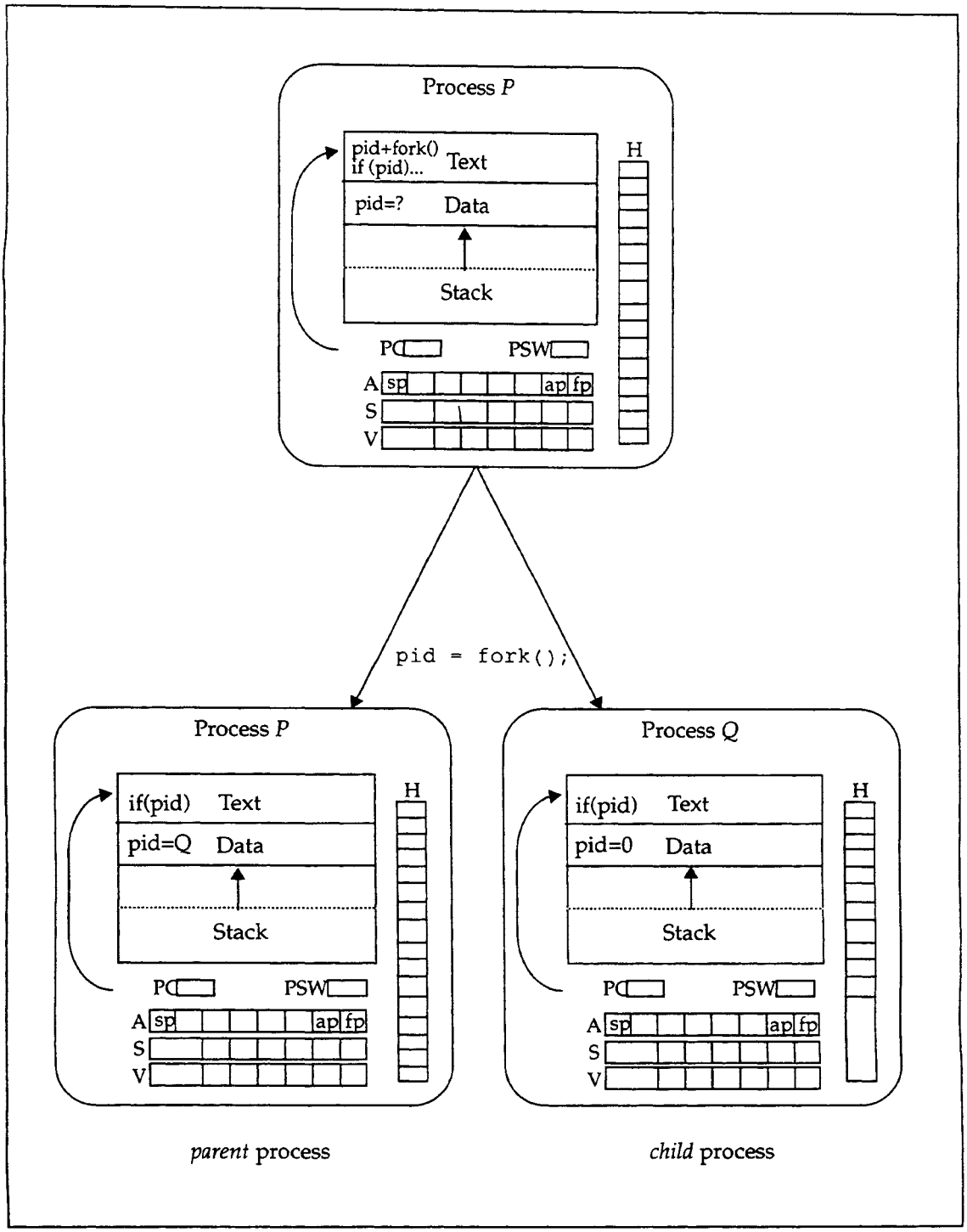
`fork()` causes the creation of a new process. The new process (child process) is an exact copy of the original calling process (parent process) except for the following:

- The child process has its own unique PID. When a process is created, it is assigned a process id not in use by another process at that time.
- The child process has a different parent process ID or PPID from its parent. The PPID of a process indicates the process ID that created the process.
- The child process has its own copies of the parent process' open file descriptors. This resource is the only one shared between child and parent; operations on a descriptor by parent or child affect both the parent and the child.

On successful completion, `fork()` returns a value of 0 to the child process and returns the PID of the child process to the parent process. See the fork (2) man page for a full description of the `fork()` system call.

Figure 69 shows the effects of process *P* executing the instruction `pid = fork()`.

Figure 69 Effects of process *P* calling `fork()`



In Figure 69, process *P* is the parent process of a new child process, *Q*. Initially, as shown at the top of the figure, the program counter of process *P* point to the instruction `pid=fork()`, which is the next instruction to be executed. The value of variable *pid* (in the data segment) is undefined (=?).

The system call `fork()` is executed and process *Q* is created by copying process *P* (shown in the lower half of the figure); the variable *pid* is assigned the value *Q* in the parent process and the value 0 in the child process. The program counters of both processes are the same and point to the first instruction after the `fork()`.

The child process *Q* is a new process and is a copy of its parent's image, including open files (not shown). Immediately after `fork()`, the child process' registers contain the same values as its parent's registers. The data segment is the same size as the parent's and contains the same values (except for the return value of `fork()`). The child's stack is an exact copy of the parent's stack. In both the parent and child, execution continues with the instruction immediately following the `fork()` system call.

Multiprocess debugging commands

adb allows you to debug multiprocess programs by creating multiple processes and managing each process as a distinct adb job. An adb job is an envelope enclosing one process; to debug a multiprocess program, you must manage many jobs.

Debugging multiprocess programs is similar to debugging multithreaded programs. Instead of many threads, a multiprocess program has many jobs.

As processes are distinct from each other, so are adb jobs. Each job includes:

- A process. A job's associated PID is displayed in all job output or can be manually displayed using the `$?` command.
- One or more process threads. This chapter will only consider single-threaded processes. In general, an adb job may contain any number of threads, up to the maximum defined by the processor configuration.
- A list of process breakpoints. Process breakpoints apply to all threads within a process.

When debugging a multiprocess program, one job is always the *current* job. (Within a job, one thread is always the current thread.) When a job is current, job-specific commands apply to that job; the current job can be changed to another job using an adb command. In output displays, the current job (and current thread) are denoted by an asterisk.

The previous chapter presented commands for debugging multithreaded programs. In many cases, those commands also apply to multiprocess programs. Commands that debug multiprocess programs are listed in Table 28.

Table 28
Commands for debugging multiple processors

Command	Function
`$j`	Display status of all jobs and all threads
`$h=`	Assign value to a hardware communication register
`$h?`	Display contents of a hardware communication register
`$n`	Display or set maximum number of processors allocated for a process
`$R`	Display registers of all threads within current job
`$C`	Continue execution of all threads within current job
`:e`	Change the exit disposition of the current job to RELEASE
`:f`	Make a job the current job. This command can also change the current thread.
`:S`	Execute one instruction in each thread within the current job
`)help`	Display a help file containing information about `adb`
`)status`	Display status about `adb` modes, including current input radix, current output radix, inherit mode, and operating mode

For fuller descriptions of these commands, see the adb(1) man page or Chapter 9, "adb command reference."

Table 29 lists adb commands that are applicable to the current job.

Table 29
adb commands that apply to the current job

Command	Function
$b	Display currently set breakpoints
$j	Display status of all threads
$h=	Assign value to a hardware communication register
$h?	Display contents of a hardware communication register
$R	Display registers for all threads in the current job
$?	Display the PID, the signal that caused job stoppage, and all thread register sets as with $R
:b	Set a breakpoint in the current job; breakpoints apply to all threads within the job
:d	Delete a breakpoint from the current job.
:e	Change exit disposition of the current job to RELEASE
:f	Make a job and/or thread current
:k	Kill current job
:i	Toggle inherit mode
:r	Run *objfile* as a process
:C	Continue execution for all threads in the current job
:D	Delete all currently set breakpoints
:S	Execute one instruction for each thread in current job

Example of multiprocess debugging

The example C program inFigure 70 is a very simple multiprocess application. The program is written in C and uses the `fork()`, `wait()`, and `exit()` system calls to create, manage, and terminate processes.

After `fork()` has been called, the parent process executes a subprogram called `parent()` that is representative of parent code found in applications. The child process executes a subprogram called `child()` that has some simple work for the child process to do.

The subprogram `parent()` uses the `wait()` system call to wait for the child process to terminate. The `wait()` system call blocks the parent until the child has finished; execution in the parent resumes when the call to `wait()` has returned. The PID of the child is the return value of `wait()`.

Figure 70 shows the example C program.

Figure 70 A sample C program that uses `fork()`

```
#include <stdio.h>
#include <sys/wait.h> int pid; /* process id returned from fork() */
union wait status;             /* the status of the child when it completes */
 int parent();                 /* work for the parent process to do */
int child();                   /* work for the child process to do */
int fork();                    /* fork a new process */
int wait();                    /* wait for the child process to finish */
int getpid();                  /* get the process' own pid */
/*
 * main() - execute a fork to create a new process. the parent will wait
 * for the child to complete. parent and child do simple work.
 */
main()
{
        printf("main: about to fork... \n");
        /*
         * fork a new process.
         * if fork returns -1, then an error occurred.
         * if fork returns a 0, then call child().
         * if fork returns nonzero, then call parent()
         */
        pid = fork();          /* two processes now...  */
        if (pid == -1) {       /* fork failed */
        perror("fork failed");
        exit(1);
}
        if (pid == 0) {        /* child process */
        child();
        exit(0);
}
        if (pid > 0) {         /* parent process */
        parent();
        wait(&status);
}
        exit(0);               /* all done! */
}
/*
 *parent() - the parent's work. print a message with the parent's process id(pid
 * using getpid() and the process id of the child (from the global variable pid.
 */
int parent()
{
        printf("I am the parent process. \n");
        printf("The parent process id is %d \n", getpid());
        printf("The child process id is %d \n", pid);
        return;
}
/*
 * child() - the child's work. print a message with the child's process
 * id using getpid()
 */
int child()
{
        printf("I am the child process - my process id is %d n", getpid());
        return;
}
```

Example adb multiprocess debugging

This section presents an example of adb commands and output. The C program in Figure 70 is used as the sample program.

Step 1 Create the executable using the C compiler.

```
% cc -o example fork.c
%
```

Step 2 Invoke adb to debug the executable example.

```
% adb example
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use )help for help.
(adb)
```

Step 3 Show the status of important modes in adb.

```
(adb))status
inherit mode: OFF
operating mode: sequential
exit disposition: terminate
input radix: 16
output radix: default
static symbols: NO
maximum processors: 1
help file: /usr/lib/adb/helpfile
(adb)
```

Step 4 Toggle inherit mode on. When inherit mode is set, child processes created by the debugged process will inherit all debug states, thus enabling them to be debugged. Use `:i`.

```
(adb):i
job 0 inherit mode: SET
(adb)
```

Step 5 Set process breakpoints at `_main`, the start of the program, and at parent and child. Child processes inherit the parent's breakpoints. Breakpoints are set using the `:b` command.

```
(adb) _main:b
(adb) _parent:b
(adb) _child:b
```

Step 6 Display the current breakpoints with the `$b` command.

```
(adb) $b
job 0: breakpoint display
count bkpt       command
1     _child 1     _parent 1     _main
(adb)
```

Step 7 Begin execution of the program with `:r`. The program executes until it encounters a breakpoint.

```
(adb):r
job 0: running
job 0/0: breakpoint _main:    sub.w #0,a0
(adb)
```

Step 8 Display the status of all jobs using the `$j` command. The current job is denoted by an asterisk in the column labeled c. The output also shows a single thread, thread 0; thread 0 is the current thread in the current job.

```
(adb) $j
job    pid   c     state
--------------------------------
0      978   *     active, sequential, terminate on quit, native fpmode, inherit
             *     thread 0: at breakpoint _main
                   file '?': example
(adb)
```

Step 9 Continue execution of the current job using the `:c` command. The program prints a message and then forks.

```
(adb):c
job 0: running main: about to fork...
job 1: new process, pid 980
job 0/0: breakpoint _parent: sub.w #0,a0
(adb)
```

Step 10 Display the status of all jobs using `$j`. The display shows two jobs: job 1 enclosing process 980, the new process, and job 0 enclosing the parent process 978. Job 0 is the current job; thread 0 is the current thread within job 0. Thread 0 is the current thread in job 1.

```
(adb) $j
job     pid     c       state
---     -----   -       -------------------------
0       978     *       active,sequential,terminate on quit,native fpmode,
                        inherit
                *       thread 0: at breakpoint _parent
                        file '?': example
1       980             active,sequential,terminate on quit, native fpmode
                *       thread 0: stopped at 0x80005332
(adb)
```

Step 11 Make job 1 the current process, using `:f`.

```
(adb) 1:f
job 1/0: current
job/thread
(adb)
```

Step 12 Continue execution of the current job with the `:c` command. Execution stops at the next breakpoint at `_child`.

```
(adb):c
job 1: running
job 1/0: stopped at _child+0x2: mov   a0,a6
(adb)
```

Step 13 Continue execution of the current job again. When the child terminates, the remaining job becomes the current job.

```
(adb):c
job 1: running I am the child process - my process
id is 980
job 1: terminated
job 0: new current process terminated
(adb)
```

Step 14 Display the status of all jobs with the `$j` command. The child process has finished; the parent process is now current and is stopped at the breakpoint at `_parent`.

```
(adb) $j
job   pid   c     state
---   -----  -    ------------------------
0     978   *     active, sequential, terminate on quit, native fpmode, inherit
            *     thread 0: at breakpoint _parent
                  file '?': example
(adb)
```

Step 15 Continue execution of the current job using `:c`. When a child process completes, the parent process receives a SIGCHLD *signal* indicating that the child has exited. Signals delivered to any process are reported to the user; it is up to the user to pass the indicated signal to the process being debugged. In this case, the signal is ignored and not passed to the parent.

```
(adb):c
job 0: running
job 0/0: stopped by a child status change
job 0/0: breakpoint _parent: sub.w #0,a0
(adb)
```

Step 16 Continue execution of the current job again.

```
(adb):c
job 0: running
I am the parent process.
The parent process id is 12210
The child process id is 12213
(adb)
```

Step 17 Quit `adb`.

```
(adb) $q
%
```

adb command reference

9

This chapter explains adb commands. It consists of a series of "reference pages," each of which describes a single adb command. A reference page may be longer than one page in length.

Organization of reference pages

To help you locate a command description, the command name is located on the top outer corner of each page. Reference pages are arranged alphabetically under the following categories:

- : commands
- $ commands
- ? commands
- / commands
- = commands
-), **RETURN**, >, and ! commands

Page format

Each reference page is designed to be easily scanned. The top of each page contains the name of the current reference page.

Each reference page begins with the name of the command along with a brief description. This information is immediately above a dark line, so it is easy to see. For example, the first page of the `:d` command looks like this:

:d
delete breakpoint

Syntax	[*address*] `:d`
Parameters	None
Prefix arguments	If no address is specified, `adb` tries to delete a breakpoint set at *dot*.
Description	The `:d` command deletes a single breakpoint. If no address precedes the command, `adb` deletes the breakpoint at the current address, if possible. When no breakpoint is set at the specified address, the message `no breakpoint set` is displayed.
Example	List the currently set breakpoints and delete the breakpoint set at *_main*.

```
(adb) $b
job 0: breakpoint display
count bkpt      command
1     _Get_response
1     _main
(adb) _main:d
(adb) $b
job 0: breakpoint display
count  bkpt      command
1     _Get_response
(adb)
```

The main body of each reference page is in a two-column format. The left column contains the name of the reference section (syntax, description, and so on). The right column contains the text for the sections.

Each reference page consists of seven sections. The first section, which appears at the top of each new reference page, contains the name of the command and a one-line summary of the command.

The next six sections are as follows:

Syntax	The syntax, or rules, for the command.
Parameters	A description of each of the parameters specified in the command syntax.
Prefix arguments	A description of the arguments that precede the command itself.
Description	A detailed description of the command and how the debugger uses it.
Example	A series of examples illustrating each of the command's options. Not every reference page contains an example.
Related commands	A list of commands that are related to the particular command. The list of commands includes the command name and a brief description of the command.

List of reference pages

Commands that begin with a : include:

:b	:e	:r
:c	:f	:S
:C	:i	:s
:d	:k	
:D	:m	

Commands that begin with a $ include:

$<	$g	$q
$<<	$h?	$r
$>	$h=	$R
$?	$i	$s?
$a=	$j	$s=
$a?	$k	$t
$b	$l	$v=
$c	$m	$v=
$e	$n	$x
$f	$o	$X

Commands that begin with a ? include:

?	?g	?S
?=	?h	?t
?a	?i	?w
?b	?m	?"..."
?c	?n	?^
?C	?P	?+
?f	?r	?-
?F	?s	

Commands that begin with a / include:

/	/g	/S
/=	/h	/t
/a	/i	/w
/b	/m	/"..."
/c	/n	/^
/C	/P	/+
/f	/r	/-
/F	/s	

Commands that begin with a = include:

=a	=m	=w
=c	=n	=Y
=C	=p	=^
=f	=r	=+
=F	=s	=-
=i	=t	

Miscellaneous commands include:

```
)comment
) help
) static
) status
RETURN
>
!
```

Reference pages are arranged alphabetically under the following categories:

- : commands
- $ commands
- ? commands
- / commands
- = commands
-), **RETURN**, >, and ! commands

Syntax

[*address*] [, *count*] : b [*command*]

Parameter	Meaning
command	The *command* parameter specifies the command (or series of commands) to be executed when the breakpoint is reached.
Prefix argument	**Meaning**
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
, *count*	The *,count* parameter specifies the number of breakpoints to ignore before stopping. `adb` ignores the first *count* -1 breakpoints and stops the *count*th time the breakpoint is reached.

Description

The `:b` command sets a breakpoint at the current or a specified address. Immediately before the instruction stored at the specified address is executed, the breakpoint is reached, and `adb` performs some activity.

By default, `:b` sets a breakpoint at the current instruction address. To set a breakpoint at a different address, precede the command with the address. The address may be an actual address or a symbolic one, provided the executable image contains symbol table information.

You can also precede the command with a *,count* value, which specifies the number of times `adb` is to ignore the breakpoint before stopping the program. `adb` ignores the first *count*-1 times the breakpoint is reached and then suspends program execution the next time the breakpoint is reached.

If you specify a command to be executed when the breakpoint is reached, `adb` executes the specified command or series of commands then stops.

If you do not specify a command, program execution stops when the breakpoint is reached and adb waits for command input. The program also stops if the command sets the value of dot to zero.

Examples

1. Set a breakpoint at the current address; then, display the currently set breakpoints to make sure the new breakpoint is set.

```
(adb) :b
(adb) $b
job 0: breakpoint display
count     bkpt      command
1         _thread+0xa
(adb)
```

2. Set a breakpoint at the beginning of the _main routine; then display the currently set breakpoints to make sure the new breakpoint is set.

```
(adb) _main:b
(adb) $b
job 0: breakpoint display
count     bkpt      command
1         _main
(adb)
```

3. Set another breakpoint at _thread that stops the fifth time the routine is called; then, display the breakpoints to make sure the new breakpoint is set.

```
(adb) _thread,5:b
job 0: breakpoint display
count     bkpt      command
5         _thread
1         _main
(adb)
```

4. Set a breakpoint at _sum_in_parallel that displays the general registers when the breakpoint is reached; then display the breakpoints to make sure the new breakpoint is set.

```
(adb) _sum_in_parallel:b $r
(adb) $b
job 0: breakpoint display
count     bkpt      command
1         _sum_in_parallel        $r
5         _thread
1         _main
(adb)
```

Related commands

`$b`	Display currently set breakpoints.
`:C`	Continue execution for all threads in the current job.
`:c`	Continue program execution.
`:d`	Delete breakpoint.
`:D`	Delete all currently set breakpoints.
`:r`	Run objfile.
`:S`	Execute one instruction for each thread in the current job.
`:s`	Execute a single instruction.

Syntax

[*address*] [*count*] : c [*signal*]

Parameter	Meaning
signal	Number corresponding to a ConvexOS signal. Interrupts are asynchronous (usually abnormal) events that occur while a program is executing and which generate special signals. Signals can be generated by input from the keyboard, by a program error (for example, a bus error), by a request from another program (kill), or by a process that is stopped because it tries to access its control terminal while in the background. Some signals normally terminate a process if no action is taken.

Prefix argument	Meaning
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,count	The *,count* parameter specifies the number of breakpoints to ignore before stopping. `adb` ignores the first *count* -1 breakpoints and stops the *count* th time the breakpoint is reached.

Description

The `:c` continues program execution that has been suspended by a breakpoint or some ConvexOS signal. Unlike the `:r` command, which restarts the program, `:c` resumes execution from the point where it stopped.

The *address* parameter lets you resume execution at a specific address other than the current one where the program stopped. adb moves the current *address* to that address and resumes program execution using the current state of the program.

The *count* parameter specifies the number of breakpoints to skip before stopping program execution again. The program is stopped on the *count*th time a breakpoint is reached. The count is incremented when any breakpoint is reached, not just a particular one.

To continue execution and pass a signal to the program, include the *signal* parameter. This is useful in debugging programs that do explicit signal handling.

Examples

1. Continue execution after the breakpoint.

```
(adb):r
job 0: running
job 0/0: breakpoint _sum_in_parallel:      ulk   0x8021
(adb):c
job 0: running
job 0/0: breakpoint _thread: ld.w #1,a1
(adb)
```

2. Continue execution until the second breakpoint is reached. In the program, the first routine called from `_main` is `Get_lesson`; it is skipped because the `,2:c` command instructed adb to stop after the second breakpoint.

```
(adb) $b
job 0: breakpoint display
count bkpt      command
1     _Get_lesson
1     _Create_frame_index
1     _main
(adb) :r
job 0: running
job 0/0: breakpoint _main:    sub.w  #0,a0
(adb) ,2:c
job 0/0: breakpoint _Create_frame_index:    sub.w  #0,a0
(adb)
```

3. Run a program then stop it with a CTRL-c. Pass it a user-defined signal (SIGUSR1, which is signal number 30) instead of the interrupt signal. `0t30` is use to specify the decimal 30..

```
(adb) :r
job 0: running ^C
job 0/0: stopped by an interrupt
job 0/0: stopped at _main+0x56: ld.w  _i,s0
(adb) :c 0t30
job 0: running received signal SIGUSR1; i = 19210
job 0: running
job 0: terminated
(adb)
```

Related commands

`$b`	Display the currently set breakpoints.
`:b`	Set breakpoint.
`:C`	Continue execution for all threads in the current job.
`:S`	Execute one instruction for each thread in the job.
`:s`	Execute a single instruction.

Syntax

[*address*] [,*count*] : C [*signal*]

Parameter	Meaning
signal	Number (0 through 31) corresponding to a ConvexOS signal. Interrupts are asynchronous (usually abnormal) events that occur while a program is executing and which generate special signals. Signals can be generated by input from the keyboard, by a program error (for example, a bus error), by a request from another program (kill), or by a process that is stopped because it tries to access its control terminal while in the background. Some signals normally terminate a process if no action is taken.

Prefix argument	Meaning
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,count	The *,count* parameter specifies the number of breakpoints to ignore before stopping. `adb` ignores the first *count* -1 breakpoints and stops the *count*th time the breakpoint is reached.

Description

The :C command continues execution of all threads in the current job.

The *address* parameter lets you resume execution at a specific address other than the current one where the program stopped. adb moves the current *address* to that address and resumes program execution using the current state of program.

The *count* parameter specifies the number of breakpoints to skip before stopping program execution again. The program is stopped on the *count* th time a breakpoint is reached. The *count* is incremented when any breakpoint is reached, not just a particular one.

To continue execution and pass a signal to the program, include the *signal* parameter. This is useful in debugging programs that do explicit signal handling.

Examples

1. Run the program and continue both threads.

```
(adb) :r
job 0:running
job 0/0: stopped at _thread+0x1a:inc.w
job 0/0: stopped at _thread+0x2:
(adb) :C
job 0: terminated
all processes terminated
(adb)
```

2. Continue execution until the second breakpoint is reached. In the program, the first routine called from _main is Get_lesson; it is skipped because the ,2:C command instructed adb to stop after the second breakpoint.

```
(adb) $b
job 0: breakpoint display
count     bkpt     command
1         _Get_lesson
1         _Create_frame_index
1         _main
(adb) :r
job 0: running
job 0/0: breakpoint _main:    sub.w  #0,a0
(adb) ,2:C
job 0/0: breakpoint
_Create_frame_index:     sub.w  #0,a0
(adb)
```

Related commands

`$b`	Display currently set breakpoints.
`:b`	Set breakpoint.
`:c`	Continue program execution.
`:S`	Execute one instruction for each thread in the job.
`:s`	Execute a single instruction.

Syntax

[*address*] `:d`

Parameter

None

Prefix argument	Meaning
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35. If no *address* is specified, `adb` tries to delete a breakpoint set at dot.

Description

The `:d` command deletes a single breakpoint. If no address precedes the command, `adb` deletes the breakpoint at the current address, if possible. When no breakpoint is set at the specified address, the message

```
no breakpoint set
```

is displayed.

Examples

1. List the currently set breakpoints and delet the breakpoint set at `_main`.

```
(adb) $b
job 0: breakpoint display
count  bkpt       command
1      _main
1      _Get_response
(adb) _main:d
(adb) $b
job 0: breakpoint display
count  bkpt       command
1      _Get_response
(adb)
```

2. Delete the breakpoint at dot. dot is not at the address of one of the breakpoints, as shown by the ?i command.

```
(adb) $b
job 0: breakpoint display
count  bkpt       command
1      _main
1      _Get_response
(adb) ?i
_Clear_screen:  sub.s    #0, a0
(adb):d
no breakpoint set
(adb)
```

Related commands

`$b`	Display currently set breakpoints.
`:b`	Set brealpoint.
`:D`	Delete all currently set breakpoints.

:D

Delete all currently set breakpoints

Syntax

```
:D
```

Parameter

None

Prefix argument

None

Description

The `:D` command deletes all currently set breakpoints in the current job.

Examples

List the currently set breakpoints; then delete all of them.

```
(adb) $b
job 0: breakpoint display
count  bkpt       command
1      _main
1      _Get_response
(adb) :D
(adb) $b
job 0: no breakpoints set
(adb)
```

Related commands

`$b`	Display currently set breakpoints.
`:b`	Set breakpoint.
`:d`	Delete breakpoint.

:e

Change the exit disposition of the current job to RELEASE

Syntax

```
:e
```

Parameter

None

Prefix argument

None

Description

The `:e` command changes the exit disposition of a process running under `adb`. By default, `adb` terminates an active process when it exits. You can use this command to force `adb` to let the process continue running when you exit `adb`. The operating system then handles the process.

Active processes are those that have started executing in `adb` and have not yet terminated. If a process is currently stopped at a breakpoint or after a single instruction step, it is an active program.

This is a one-way switch. Once you execute this command, you cannot reverse it and cause `adb` to terminate an active process when you exit `adb`.

Examples

Change the exit disposition of the current process.

```
(adb):e
job 0 exit disposition: release on quit
(adb)
```

Related commands

None

Syntax

[*job*] [, *thread*] : f

Parameter

None

Prefix argument	Meaning
job	The *job* argument specifies which job to bring to the foreground. If no *job* is specified, it is not changed.
thread	The *thread* argument specifies which *thread* of the current (or specified) job to bring to the foreground.

Description

The : f command brings a job and/or thread to the foreground, thus making it the current thread.

The concept of foreground in adb is similar to the concept of foreground in ConvexOS. When adb is debugging multiple processes (jobs and threads), it places one process in the foreground and puts the others in the background. While all of the processes can be executing, you can only interact with the current (foreground) process.

When you bring a thread to the foreground, you can display information about it, modify its registers and variables, and change its characteristics.

When you specify a *job*, it becomes the current job. Specifying a *thread* makes it the current thread in the job. When you specify a thread without a job, adb brings the thread in the current job to the foreground. When you specify a job without a thread, adb brings the last current thread for the specified job to the foreground.

If neither *job* nor *thread* is specified, adb displays the current job and thread.

Examples

Display the job status; then, bring thread 1 to the foreground.

```
(adb) $j
job   pid   c     state
---   ----- -     ------------------------
0     6646  *     active, sequential, terminate on quit, native fpmode
            *     thread 0: stopped at _thread+0x2a
                  thread 1: at breakpoint _thread
                  file '?': a.out
(adb) ,1:f
job 0/1: current job/thread
(adb)
```

Related commands

`$j` Display status of all jobs.

Syntax

```
:i
```

Parameter

None

Prefix argument

None

Description

The `:i` command toggles inherit mode. When inherit mode is set, any child processes created by the process being debugged inherit the debug state; that enables you to debug the child processes.

A child process is one which is forked by the main process (a new job). If inherit mode is turned off (`CLEAR`), the child process runs of its own accord. You are unable to stop, modify, or regulate it.

You can use this command to find out what mode you are in.

Examples

Toggle the inherit mode to SET. Toggle it back to CLEAR.

```
(adb):i
job 0 inherit mode: SET
(adb):i
job 0 inherit mode: CLEAR
(adb)
```

Related commands

None

:k

kill the current job

Syntax

```
:k
```

Parameter

None

Prefix argument

None

Description

The `:k` command kills the current job. All threads active in the job are also killed.

If you need to kill a job running in the background, bring it to the foreground using the `:f` command; then kill it.

Examples

Terminate the current process.

```
(adb):k
job 0: killed
(adb)
```

Related commands

`$j`	Display status of all jobs.
`:f`	Bring a job or thread to the foreground.

:m

Toggle scheduling mode between DYNAMIC and FIXED

Syntax

```
:m
```

Parameter

None

Prefix argument

None

Description

The `:m` command toggles the scheduling mode between FIXED and DYNAMIC in the current process. When scheduling mode is set to DYNAMIC, CPUs are allocated to a process on an as-needed basis; when scheduling mode is set to FIXED, all CPUs are allocated to the process each time the process is scheduled for execution by the operating system.

The scheduling mode is initially set to FIXED. You can use this command to find out what mode you are in.

Examples

Toggle the scheduling mode to DYNAMIC and then back to FIXED.

```
(adb) :m
job 0 scheduling mode: dynamic
(adb) :m
job 0 scheduling mode: fixed
(adb)
```

Related commands

None

:r

Run *objfile* as a process

Syntax

[*address*] [, *count*] : r [*program-arguments*]

Parameter	Meaning
program-arguments	The *program-arguments* contain any arguments that you would normally include on the command line when you executed the program from the ConvexOS shell. Program arguments are separated by spaces.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,*count*	The ,*count* parameter specifies the number of breakpoints to ignore before stopping. adb ignores the first *count* -1 breakpoints and stops the *count*th time the breakpoint is reached.

Description

The :r commands runs or reruns the program. The command starts the program from the beginning, even if the program was stopped during execution.

The *address* argument lets you specify a specific address for execution to begin. Normally you would start execution at the standard entry point.

The count argument tells adb to ignore the first *count* -1 breakpoints it encounters. It stops program execution on the *count* th breakpoint. The breakpoint count is incremented each time any breakpoint is reached.

Examples

1. Start a new process.

```
(adb) $j
(adb) :r
job 0: running
job 0/0: breakpoint _main:    sub.w  #4,a0
(adb)
```

2. Rerun a job currently executing.

```
(adb) :c
job 0: running
job 0/0: stopped at _thread+0x38:     brs.f  _thread+0x2c
job 0/1: stopped at _thread+0x2:      inc.w  0x8022,a1
(adb) :r
job 0: running
job 0/0: breakpoint _thread:  ld.w  #1,a1
(adb)
```

3. Rerun the program and stop when the second breakpoint is reached. In the program, the first routine called from main is `Get_lesson`; it is skipped because the `,2:r` command instructed `adb` to stop after the second breakpoint.

```
(adb) $b
job 0: breakpoint display
count bkpt      command
1     _Get_lesson
1     _Create_frame_index
1     _main
(adb) ,2:r
job 0: running
job 0/0: breakpoint _Create_frame_index:     sub.w  #0,a0
(adb)
```

Related commands

`$j` Display status of all jobs.

Syntax

[*address*][,*count*] : s *signal*

Parameter	Meaning
signal	A *signal* is the number corresponding to a ConvexOS signal Interrupts are asynchronous (usually abnormal) events that occur while a program is executing and which generate special signals. Signals can be generated by input form the keyboard, by a program error (for example, a bus error), by a request form another program (`kill`), or by a process that is stopped because it wants to access its control terminal while in the background. Some signals normally terminate a process if no action is taken.
Prefix argument	**Meaning**
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (*_main*, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,count	The *,count* parameter specifies the number of breakpoints to ignore before stopping. `adb` ignores the first *count* -1 breakpoints and stops the *count*th time the breakpoint is reached.

Description

The : s command executes (steps) one assembly-language instruction in the current thread.

Examples

1. Execute a single instruction in the current thread.

```
(adb :s
job ): running
job 0/0: stopped at _thread+0x2: inc.w 0x8022, a1
(adb)
```

2. With two threads active, execute a single instruction in the current thread only.

```
(adb) :c
job 0: running
job 0/0 stopped at _thread+0x2: i
job 0/1: stopped at _thread+0x2: inc.w 0x8022, a1
(adb) :s
job 0: running
job 0/0: stopped at _thread+0x26 bra.f _thread+0x3c
(adb)
```

3. With two threads active, change threads and execute a single instruction.

```
(adb) :c
job 0: running
job 0/0 stopped at _thread+0x2: le.w a4, a2
job 0/1: stopped at _thread+0x2: inc.w 0x8022, a1
(adb) ,1:f ; :s
job 0/1: stopped at _thread+0x24 le.w a4, a2
job 0: running
job 0/1:stopped at _thread+0x2: inc.w 0x8022, a1
(adb)
```

Related commands

`:C`	Continue execution of all threads in the current job.
`:c`	Continue program execution.
`$R`	Display registers for all threads in the current job.
`$r`	Display contents of general registers.
`:S`	Execute one instruction for each thread in the current job.

:S

Execute one instruction for each thread in the current job

Syntax

[*address*] [, *count*] : S [*signal*]

Parameter	Meaning
signal	Number corresponding to a ConvexOS signal. Interrupts are asynchronous (usually abnormal) events that occur while a program is executing and which generate special signals. Signals can be generated by input from the keyboard, by a program error (for example, a bus error), by a request from another program (kill), or by a process that is stopped because it wants to access its control terminal while in the background. Some signals normally terminate a process if no action is taken.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,count	The *,count* parameter specifies the number of breakpoints to ignore before stopping. `adb` ignores the first *count* -1 breakpoints and stops the *count*th time the breakpoint is reached.

Description

The : S command executes one assembly-language instruction for each thread in the current job.

When you specify an *address*, `adb` starts execution for all threads at that point. If a *signal* is specified, `adb` sends it to all threads.

Examples

1. Execute one instruction for each thread in the current job. The breakpoint is reached and shows that two threads are executing.

```
(adb) :c
job 0: running
job 0/0: stopped at _thread+0x38:    brs.f  _thread+0x2c
job 0/1: stopped at _thread+0x2:     inc.w  0x8022,a1
(adb) :S
job 0: running
job 0/0: stopped at _thread+0x3a:    br   _thread+0x18
job 0/1: stopped at _thread+0x2:     inc.w  0x8022,a1
(adb)
```

2. Run the program, stopping at a breakpoint at `_main`, then execute four instructions.

```
(adb) :r
job 0: running
job 0/0: breakpoint _main:   sub.w  #0,a0
(adb) ,4:S
job 0: running
job 0/0: stopped at _main+0xc: pshea  0x0
(adb)
```

Related commands

`:C`	Continue execution for all threads in the current job
`:c`	Continue program execution
`$R`	Display registers for all threads in the current job
`$r`	Display contents of general registers
`:s`	Execute a single instruction

Syntax

$< *file*

Parameter	Meaning
file	*file* is the name of a file containing adb commands.

Prefix argument

None

Description

The $< command reads debugger commands from the specified file and executes them on the current executable image. This is often used to get a program to a known state before starting interactive debugging.

A command file might contain the following commands

```
_main:b
:r
$r
:c
```

This would set a breakpoint at *_main* (*_main*:b), run the program (:r), list the contents of the registers when the breakpoint is reached ($r), and continue program execution (:c).

If the command input file contains another $< command, adb begins reading the commands in the new file. Any commands following the $< in the original file are ignored.

Examples

1. Start adb and execute the commands stored in the file in1. The file in1 contains the following:

```
_main:b
_Get_lesson:b,5?ia
:r
$r
:c
```

```
% adb cai
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $<in1
job 0: running
job 0/0: breakpoint _main:   sub.w  #0,a0
job 0/0: register display

pc=80001a2e (_main+0x2)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd8c a1=ffffcda0 a2=ffffcdb4 a3=ffffce00
tt=000000eb a4=00000000 a5=00000030 ap=ffffcda0
FP=ffffcd8c s0=00000000e0000000 s1=0000000000000000
s2=0000000c00000000 s3=0000000000000000
s4=00000000fffffffc s5=0000000000000080
s6=00000000e0000000 s7=0000000000000000 j
ob 0: running
_Get_lesson: sub.w  #0,a0
_Get_lesson+0x2:                mov   a0,a6
_Get_lesson+0x4:                pshea  0x0
_Get_lesson+0x8:                calls  _Clear_screen
_Get_lesson+0xe:                add.w  #4,a0
_Get_lesson+0x10:
job 0/0: breakpoint
_Get_lesson:        sub.w  #0,a0
(adb)
```

2. Execute the commands stored in the file *in1*. Commands stored in the file *in2* are read first. The files *in1* and in2 contain the same commands as *in1* in the previous example, but they are split as follows:

```
(in1)                (in2)
_main:b              :r
 _Get_lesson,5?ia    $r
$<in2
:c
```

```
% adb cai
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $<inl
job 0: running
job 0/0: breakpoint _main:   sub.w  #0,a0
job 0/0: register display  pc=80001a2e (_main+0x2)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd8c a1=ffffcda0 a2=ffffcdb4 a3=fffce00
tt=0000010c
a4=00000000 a5=00000030 ap=ffffcda0 FP=ffffcd8c
s0=00000000e0000000 s1=0000000000000000
s2=0000000000000000
s3=0000000000000000 s4=00000000fffffffc
s5=00000000fffce1b
s6=00000000e0000000 s7=0000000000000000
(adb)
```

The `:c` command in the first file was ignored. Therefore, the breakpoint at `_Get_lesson` was not reached, and the instructions were not displayed.

Related commands

`$<<`	Read and execute `adb` commands from a file, then return to original file
`$<`	Write `adb` output to a file

$<<

Read and execute adb commands from a file, then return to original file

Syntax

$ << *file*

Parameter	Meaning
file	*file* is the name of a file containing adb commands.

Prefix argument

None

Description

The $ << command reads debugger commands from the specified file and executes them on the current executable image. This is often used to get a program to a known state before starting interactive debugging.

Unlike the $ < command, when adb encounters this command within a command file, it executes the commands in the file and then returns to the original file and continues executing those commands.

A command file might contain these commands

```
_main:b
$<<foo
:r
:c
```

This sets a breakpoint at _main, reads the command file foo, executes the program, and continues after the breakpoint is reached.

When the *,count* prefix is 0, the $ < command is ignored. When no *file* is specified, adb terminates the current input stream.

Examples

Start adb and executed the commands stored in the file *in1*, which also executes the commands stored in the file *in2*. The files contain the following:

```
(in1)                    (in2)
 _main:b                 :r
_Get_lesson:b,5?ia       $r
$<<in2
:c
```

```
% adb cai
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $<in1
job 0: running
job 0/0: breakpoint _main:   sub.w  #0,a0
job 0/0: register display pc=80001a2e (_main+0x2)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd8c a1=ffffcda0 a2=ffffcdb4 a3=ffffce00
tt=000000eb
 a4=00000000 a5=00000030 ap=ffffcda0 FP=ffffcd8c
s0=00000000e0000000
s1=0000000000000000 s2=0000000c00000000
s3=0000000000000000
s4=00000000fffffffc s5=0000000000000080
s6=00000000e0000000 s7=0000000000000000
job 0: running
_Get_lesson:  sub.w  #0,a0
_Get_lesson+0x2:        mov   a0,a6
_Get_lesson+0x4:        pshea  0x0
_Get_lesson+0x8:        calls  _Clear_screen
_Get_lesson+0xe:        add.w  #4,a0
_Get_lesson+0x10: job 0/0: breakpoint
_Get_lesson:        sub.w  #0,a0
(adb)
```

adb executed the :c command after returning from *in2*.

Related commands

$<	Read and execute adb commands from a file.
$>	Write adb output to a file.
$<<	Read and execute adb commands from a file, then return control to original file.

Syntax

$> *file*

Parameter	Meaning
file	*file* is the name of the file to contain the debugger output.

Prefix argument

None

Description

The $> command writes adb debugging output to the specified file, overwriting any data already in the file. If the file does not exist, adb creates it. All adb command output following the $> command is directed to the file.

When no file is specified, adb displays the output on the terminal. This lets you stop writing the output to the file and resume displaying output on the terminal screen.

Examples

1. Start appending adb output to the file *out1* and execute some debugging commands.

```
(adb) $>out1
(adb) :r
(adb) :s
(adb) ,4?ia
(adb)
```

2. Stop writing to the output file and begin displaying adb output to the terminal. List the contents of *out2* using the shell escape command (!) and the ConvexOS `cat` command.

```
(adb) $>
(adb) !cat out2
job 0: running
job 0/0: breakpoint _main:    sub.w  #0,a0
job 0: running
job 0/0: stopped at _main+0x2: sub.w  s0,s0
_main+0x2:    sub.w  s0,s0
_main+0x4:    st.w   s0,_stop_program
_main+0xa:    mov    a0,a6
_main+0xc:    pshea  0x0
_main+0x10:
!
(adb)
```

The ! on the last line indicates the end of the shell command.

Related commands

`$<`	Read and execute adb commands from a file.
`$<<`	Read and execute adb commands from a file, then return control to the original file.

$?

Display PID, signals, and the general registers

Syntax

```
$?
```

Parameter

None

Prefix argument

None

Description

The `$?` command displays the PID and the signals that stopped or terminated the program. It also displays the values of the general registers (address and scalar).

Once the program begins execution (`:c`, `:r`, `:s` commands), the PID is set.

If you are debugging a core dump, the `$?` command also displays the name of the executable program that caused the dump, the version of the executable program, and the date and time the core dump was created.

Examples

Display the program information after executing a single machine instruction.

```
(adb) $?
job 0: process 24574
job 0/0: register display
pc=8000120a (_main+0x2)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd88 a1=0000000c a2=ffffcd78 a3=ffffcdfc
tt=000000f3
a4=00000000 a5=00000030 ap=ffffcd9c FP=ffffcd88
s0=00000000e0000000 s1=0000000000000000
s2=0000000000000302
s3=0000000000000000 s4=00000000fffffffc
s5=0000000000000080
s6=0000000000000004 s7=0000000000000000
(adb)
```

Related commands

`$j`	Display status of all jobs.
`:f`	Bring a job or thread to the foreground.

Syntax

`$a` *reg* = *format value*

Parameter	Meaning

reg	The *reg* parameter is the number of the address register (0-7). The SP (stack pointer), AP (address pointer), and FP (frame pointer) are equivalent to the following address registers:
	`SP = A0`
	`AP = A6`
	`FP = A7`
	If *reg* is unspecified, all registers are displayed.

format	The *format* specifies the size of the data. Address registers store only words of data (format `w`).
value	The *value* is the actual data to be stored in the address register.

Prefix argument

None

Description

The `$a=` command modifies the value of an address register.

The format for address register values must be `w`.

Examples

1. Assign the value 10 (decimal) to the FP register.

```
(adb) $fp = w 0t10
FP=0000000a
(adb)
```

2. Assign the value 10 (hexadecimal) to the A2 register.

```
(adb) $a2 = w 0x10
a2=00000010
(adb)
```

3. Assign the value 10 (octal) to the A5 register.

```
(adb) $a5 = w 0o10
a5=00000008
(adb)
```

Related commands

`$a?`	Display the contents of an address register.
`$h=`	Assign value to a hardware communication register.
`$s=`	Modify a scalar register.
`$v=`	Modify a vector register.

Syntax

`$a` [*reg*]`?` *format* [*radix*]

Parameter	Meaning

reg	The *reg* parameter is the number of the address register (0-7). The SP (stack pointer), AP (address pointer), and FP (frame pointer) are equivalent to the following address registers:
	`SP = A0`
	`AP = A6`
	`FP = A7`
	If *reg* is unspecified, all registers are displayed.

format	The *format* specifies the manner in which the information is displayed. The format can assume one of the following values: `d` Display a byte of data (may specify an optional radix) `h` Display a halfword of data (may specify an optional radix) `w` Display a word of data (may specify an optional radix) `f` Display a 32-bit value as a floating-point number (same as using `wf`)
radix	The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The radix may assume one of the following values: `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default *radix* value is hexadecimal (`x`). To set a new default *radix* use the `$x` command.

Prefix argument

None

Description

The `$a?` command displays the value of an address register. To display the contents of a single register, specify the register number as the *reg* parameter. If you do not specify a register, the contents of all address registers are displayed.

The amount of data displayed depends on the format and radix selected. You cannot use `1` format because the address registers are only 32 bits long.

The A0, A6, and A7 registers are SP, AP, and FP, respectively. You can use these numbers to see the contents of the registers. You can also display the sp, ap, fp, pc, psw, and the pseudo-register dot by using the following commands:

SP	`$sp?` *format*
AP	`$ap?` *format*
FP	`$FP?` *format*
PC	`$pc?` *format*
PSW	`$psw?` *format*
Dot	`$dot?` *format*

Examples

1. Display the contents of all address registers as words in the default (hexadecimal) radix.

```
(adb) $a?w
sp=ffffcd4c a1=80008790 a2=80008790 a3=8002f1e9
a4=00000062 a5=00000062 ap=ffffcd60 fp=ffffcd4c
(adb)
```

2. Display the contents of the A4 register as a byte with a decimal radix.

```
(adb) $a4?bu
a4=098
(adb)
```

3. Display the contents of the stack pointer (SP) as a halfword with a signed octal radix.

```
(adb) $sp?hq
sp=-31264
(adb)
```

4. Display the processor status word (PSW) as a floating-point number.

```
(adb) $psw?wx
ps= 02108000
(adb)
```

Related commands

`$a?`	Display the contents of an address register.
`$h=`	Assign value to a hardware communication register.
`$s=`	Modify a scalar register.
`$v=`	Modify a vector register.

Syntax

```
$b
```

Parameter

None

Prefix argument

None

Description

The `$b` command displays all currently set breakpoints and their associated counts and commands.

Examples

Display the currently set breakpoints.

```
(adb) $b breakpoints
count      bkpt              command
1     _Get_response           $e
1     _Determine_branch
(adb)
```

Related commands

`:b`	Set breakpoint.
`:d`	Delete breakpoint.
`:D`	Delete all currently set breakpoints.

Syntax

[*address*] [, *count*] `$c`

Parameter

None

Prefix argument	Meaning
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,count	The *,count* is the topmost number of call frames to be displayed.

Description

The `$c` command displays a backtrace through the C or FORTRAN routine calls. The list shows, from bottom to top, the routines that were called to get the program to its current state.

For each routine, the values of its arguments are displayed to help you determine where the error occurred.

Consider the following sample backtrace:

```
_read(0,8001e000,10000) from __filbuf+0x142 [ap = ffffcd78]
_filbuf(80008494) from _Cont_frame+0xb0 [ap = ffffcd9c]
_Cont_frame() from _main+0x118 [ap = ffffcdb4]
_main(1,ffffcdd4,ffffcddc) from start+0x80 [ap = ffffcdc8]
```

The first line is interpreted as follows:

`_read`	The name of the last subroutine called. The subroutines are listed in the reverse order in which they were called (`_main` was the first routine called in the example).

`(0,8001e000,10000)`	The arguments that were passed.
`from __filbuf+0x142`	The return address where the program returns after calling the subroutine.
`[AP=ffffcd78]`	The argument pointer indicating the location of the arguments.

The remaining lines are interpreted in a similar manner.

If you specify a *count*, only the topmost *count* number of call frames is displayed.

Examples

1. Display the C backtrace after the process receives an interrupt signal.

```
(adb) $c
_read(0,8001e000,10000) from __filbuf+0x142[ap= ffffccfc]
_filbuf(80008494) from _gets+0x54 [ap = ffffcd20]
_gets(8000a864) from _Get_response+0x50 [ap = ffffcd40]
_Get_response() from _Determine_branch+0x74[ap= ffffcd58]
_Determine_branch() from _main+0x104 [ap = ffffcd7c]
_main(1,ffffcd9c,ffffcda4) from start+0x80[ap = ffffcd90]
(adb)
```

2. Display the top two call frames of a C backtrace.

```
(adb) ,2$c
_Get_response() from _Determine_branch+0x74 [ap =
ffffcd54]
_Determine_branch() from _main+0x104 [ap = ffffcd78]
(adb)
```

Related commands

None

$e

Display the names and values of all external variables

Syntax

```
$e
```

Parameter

None

Prefix argument

None

Description

The `$e` command displays the names and values of all known global variables. The values are displayed in the current output radix. To change the output radix, use the `$X` command.

The global variables that `adb` displays include all variables declared globally in the program, all variables contained in libraries included in the program, and variables used by the ConvexOS operating system to run the program. These are the only variables you can manipulate in `adb`.

Examples

Display the external variables.

```
(adb) $e
_environ:  ffffcda0
_i:                             1
_j:                             0
_string:                        a002e09
_stop_program:                  0
__iob:                          1
_next_frame:                    0
_out_file:                      0
_answer:                        0
_branch:                        0
 _valid_answer:                 64
 _list_file:                    0
_stop_lesson:                   0
_lesson_name:                   66696c65
_last_frame:                    0
_total_frames:                  2c
 _testch:                       62008000
(adb)
```

Related commands

`$i`	Display the names and values of all nonzero internal variables.
`$X`	Modify default output radix.

$f

Display or set the floating-point format

Syntax

`$f` [*mode*]

Parameter	Meaning
mode	The *mode* is either `auto`, `ieee`, or `native`.

Prefix argument

None

Description

The `$f` command displays and changes the floating-point mode used by `adb`. With no parameters, the command displays the current modes used by `adb` and the program being debugged. When you supply one of the parameters, the command sets the floating-point mode for `adb` to the new value. The floating-point mode allows `adb` to translate internal bit representations of floating-point numbers (single and double precision) to the correct external value. For instance, if the floating-point numbers of a program are in `ieee` format, you must to set the mode to `ieee` so that `adb` would interpret the floating-point values correctly.

At the start of an `adb` debugging session, the floating-point mode of the debugger is the same as that of the program you are debugging. 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 `adb`.

`adb` supports three modes: `native` (for the native format), `ieee` (for the `IEEE` format), and `auto`. The `auto` mode automatically synchronizes the mode of `adb` with the mode of the program being debugged. This ensures that when `adb` prints a floating-point value, it is interpreted according to the mode used by the program at that stage of its execution.

For example, suppose the double-precision floating-point value is 0x3ff0000000000000. The modes affect the display of the value as shown below.

`ieee`	1.0
`native`	0.25

You cannot change the mode to ieee if you don't have the IEEE hardware on your system.

If the program being debugged is a dual-mode program, `adb` starts in `auto` mode.

Examples

1. Display the current floating-point modes.

```
(adb) $f
adb: native
a.out: native
(adb)
```

2. Change the numerical representation of floating-point mode to `ieee` and verify the change.

```
(adb) $f ieee
(adb) $f
adb: ieee
a.out: native
(adb)
```

3. Set the `adb` mode such that it always matches the floating-point mode used by the program. Then verify the change.

```
(adb) $f auto; $f
adb: auto
a.out: native
(adb)
```

Related commands

None

Syntax

```
$g
```

Parameter

None

Prefix argument

None

Description

The `$g` command gets new names for symbol (executable) and core files. It changes the names of *objfile* and *corefile* for the current job and resets the dates associated with each.

The command prompts for each name, at which time you can

- Enter the name of the new file to make it current.
- Type a hyphen to close the file and prohibit future referencing.
- Press **RETURN** to leave the file unchanged.

Examples

1. Change the names of the executable and core files.

```
(adb) $g
old symbol table: a.out
new symbol table: sort
old core file: core
new core file: newcore
(adb)
```

2. Change the name of the executable file only.

```
(adb) $g
old symbol table: sort
new symbol table: a.out
old core file: newcore
new core file:
core file unchanged
(adb)
```

3. Close the core file.

```
(adb) $g
old symbol table: a.out
new symbol table: symbol table unchanged
old core file: newcore
new core file: -
(adb)
```

Related commands

None

$h?

Display the contents of a hardware communication register

Syntax

`$h[`*reg*`]?`*format*`[`*radix*`]`

Parameter	Meaning
reg	The *reg* parameter is the number of the communication register (0-63). If *reg* is unspecified, all registers are displayed.
format	The *format* specifies the manner in which the information is displayed. The format can assume one of the following values. `d` Display a byte of data (may specify an optional radix) `h` Display a halfword of data (may specify an optional radix) `w` Display a word of data (may specify an optional radix) `f` Display a 32-bit value as a floating-point number (same as using `wf`) `l` Display a longword of data (may specify an optional radix) `F` Display a 64-bit value as a flaoting point number. Same as using `-lt`.
radix	The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The radix may assume one of the following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default *radix* value is hexadecimal (`x`). To set a new default *radix* use the `$x` command.

Prefix argument

None

$h?

Description

The `$h?` command displays the value of a hardware communication register. To display the value of a single register, specify the register number as the *reg* parameter. If you do not specify a register, the contents of all hardware communication registers are displayed.

The communication registers are part of a high-speed register set used for communication between the threads of a process. Each communication register is visible as a 64-bit register. Threads within a process communicate by sending and receiving data through these communication registers.

Examples

Display the value stored in the H1 register in hexadecimal base.

```
(adb) $h1?lx
job 0: communication register display

h[01]=00000000000badd3 (0)
(adb)
```

Display one word of the contents of the H1 register in decimal.

```
(adb) $h1?wt
job 0: communication register display

h[01]=  765395 (0)
(adb)
```

Related commands

`$a=`	Modify value of an address register.
`$a?`	Display the contents of an address register.
`$h=`	Assign a value to a hardware communication register.
`$r`	Display the contents of general register.
`$s=`	Modify a scalar register.
`$s?`	Display the contents of a scalar register.
`$v=`	Modify a vector register.
`$v?`	Display the contents of a vector register.

$h=

Assign value to a hardware communication register

Syntax

`$h` *reg= format value*

Parameter	Meaning
reg	The *reg* identifies the hardware communication register to display. ConvexOS systems support 64 user-modifiable hardware communication registers.
format	The *format* determines the amount of data to be stored, For hardware communication registers, you must use the `l` format.

Prefix argument

None

Description

The hardware communication registers are part of a high-speed register set used for communication between the threads of a process. Each communication register is visible as an addressable 64-bit register. Threads within a process communicate by sending and receiving data through these communication registers.

Examples

Assign the value 12345 (decimal) to the H4 register.

```
(adb) $h4=l 0t 12345
job0: communication register display

h[04]=00000000003-39 (0)
(adb)
```

Related commands

`$a=`	Modify value of an address register.
`$a?`	Display the contents of an address register.
`$h?`	Display the contents of a hardware communication register.
`$r`	Display the contents of general register.
`$s=`	Modify a scalar register.
`$s?`	Display the contents of a scalar register.
`$v=`	Modify a vector register.
`$v?`	Display the contents of a vector register.

$i

Display the names and values of all nonzero internal variables

Syntax

```
$i
```

Parameter

None

Prefix argument

None

Description

The `$i` commands displays the values of all `adb` internal variables for the current job and thread. Only those variables that have nonzero values are displayed. `adb` has the following internal variables:

- `b` Base address of the data segment
- `d` Length of the data segment
- `e` Entry point of the program
- `m` Magic number (identifies the type of file)
- `s` Length of the stack
- `t` Length of the text segment

To look at the internal variables of another job or thread, first make it the current thread using the `:f` command.

The values are displayed in the current output radix. To change the output radix, use the `$X` command.

Examples

1. Display the values of the internal variables.

```
(adb) $i
job 0: internal variable display
b = 0x80008000
d = 0x4000
e = 0x80001000
m = 0x181
s = 0x2000
t = 0x7000
(adb)
```

2. Change the output radix to decimal and display the values of the internal variables.

```
(adb) 0t10$X
output radix = 10 (base 10)

(adb) $i

job 0: internal variable display
b = 2147516416
d = 16384
e = 2147487744
m = 385
s = 8192
t = 28672
(adb)
```

Related commands

`$e`	Display the names and values of all external variables.
`:f`	Bring a job or thread to the foreground.

Syntax

[*job*]`$j`

Parameter

None

Prefix argument	Meaning
job	The *job* is the job number assigned by `adb`. This is the number that appears in a job display listing (`$j` command). This is only necessary if you want to see the status of a job other than the current one.

Description

The `$j` command displays the status of the current jobs. The status display includes

- Job number
- PID number
- Current job and thread
- List of the modes
- Current state
- Name of the executable image
- Name of the core image

A status of a program with one thread might look similar to the following:

```
job pid   c  state
--- ------  ------------------------
0   9606  *  active, sequential, native fpmode
          *  thread 0: stopped at start
             file '?': a.out
             file '/': core
```

In this status sample, the executable file is a.out, and the core image is core. An * in the c column indicates the current job and thread.

For programs running multiple threads, the `$j` command displays the information for each job. The following status sample shows the output from a multithreaded program.

```
job pid   c  state
--- ------  ------------------------
0   443   *  active,sequential,terminateon quit,native fpmode
          *  thread 0: at breakpoint
             thread 1: at breakpoint _thread
             file '?': a.out
```

Examples

1. Display the status of the current job.

```
(adb) $j
job   pid   c     state
 ---  -----  -    ------------------------
0     2421  *     active, sequential, terminate on quit, native fpmode
            *     thread 0: stopped at _read+0xe, signal = 2
                  file '?': cai
                  file '/': core
(adb)
```

2. Display the status of job 0.

```
(adb) 0$j
job   pid   c     state
---   -----  -    ------------------------
0     2910  *     active, sequential, terminate on quit, native fpmode
            *     thread 0: stopped at _thread+0x28
                  thread 1: stopped at _thread+0x22
                  file '?': a.out
                  file '/': core
(adb)
```

Related commands

`$?` Display PID, signals, and the general register.

$k

Change current kernel memory mapping (kernel debugging only)

Syntax

[*address*] [, *thread*] `$k`

Parameter

None

Prefix argument	Meaning
address	The *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
thread	The *thread* is the number corresponding to the *thread* for which you want to change the first-level page tables.

Description

The `$k` command changes the current kernel memory mapping. The address argument specifies the address of the new set of segment description registers (SDR) to map. You can only execute this command if you use the `-k` option when invoking `adb`.

Examples

1. Change the current kernel memory mapping. In this example, the `-k` option was not used on the `adb` command line.

```
(adb) $k
not debugging kernel
(adb)
```

2. Change the current kernel memory mapping for thread 0.

```
(adb),0$k
job 0: kernel thread id set to 0
(adb)
```

Related commands None

Syntax

[*limit*] `$l`

Parameter

None

Prefix argument	Meaning
limit	The *limit* is the number of bytes distant from a symbol name for `adb` to display addresses symbolically or absolutely.

Description

The `$l` command sets a value to limit the distance from which `adb` matches a symbol. When you display instructions, `adb` tries to list the address of each instruction as an offset from a known symbol. For example, if you specify the address to the `?i` command, and list five instructions, the display would look similar to the following:

```
_thread:      ld.w      #1,a1
_thread+0x2:  inc.w     0x8022,a1
_thread+0xa:  bra.f     _thread
_thread+0xc:  ld.w      8(ap),a1
_thread+0x10: ld.w      4(ap),a2
_thread+0x14:
```

The symbol is `_thread`. The next four instructions to be executed are addressed by their distance in memory from `_thread`. The `inc.w` instruction is located two bytes after the symbol. The bra.f instruction is located ten instructions after `_thread`. Thus, you know that the inc.w instruction occupies eight bytes in memory.

By changing the value for limit matches, you can tell `adb` to use only a symbolic address if the instruction falls within a certain range of memory. If an instruction is located past the limit value, `adb` displays the actual address for the instruction. The same example as above looks like this if the limit is set to nine bytes:

```
_thread:     ld.w  #1,a1
_thread+0x2:   inc.w  0x8022,a1
0x800014e8:    bra.f  _thread
0x800014ea:    ld.w  8(ap),a1
0x800014ee:    ld.w  4(ap),a2
0x800014f2:
```

In this case, the address for the `bra.f` instruction is displayed as an actual address because it is located ten bytes after `_thread` in memory.

By setting the limit to 0, you can force adb to always display actual addresses. Setting the limit to 1 causes adb to display symbolic addresses only for the instruction at the symbolic address.

Examples

1. Set the limit for symbol matches to 10 (hexadecimal) and display six instructions starting at _main.

```
(adb) 0x10$l
(adb) _main,6?ia
_main:      sub.w #4,a0
_main+0x2:    sub.w s0,s0
_main+0x4:    st.w  s0,-4(FP)
_main+0x8:    ld.w  -4(FP),s0
_main+0xc:    le.w  #1000000,s0
0x80001182:    brs.t
0x800011a6 0x80001184:
(adb)
```

2. Set the limit for symbol matches to 1 and display six instruction starting at _main.

```
(adb) 1$l
(adb) _main,6?ia
_main:      sub.w  #4,a0
0x80001172:    sub.w  s0,s0
0x80001174:    st.w  s0,-4(FP)
0x80001178:    ld.w  -4(FP),s0
0x8000117c:    le.w  #1000000,s0
0x80001182:    brs.t  0x800011a6
0x80001184:
(adb)
```

3. Set the limit for symbol matches to zero and display six instructions starting at _main.

```
(adb) 0$l
(adb) _main,6?ia
0x80001170:    sub.w  #4,a0
0x80001172:    sub.w  s0,s0
0x80001174:    st.w  s0,-4(FP)
0x80001178:    ld.w  -4(FP),s0
0x8000117c:    le.w  #1000000,s0
0x80001182:    brs.t  0x800011a6
0x80001184:
(adb)
```

Related commands

None

$m

Display the address maps

Syntax

`$m`

Parameter

None

Prefix argument

None

Description

The `$m` command displays memory mapping information for the executable and core files.

Examples

Display the current memory mapping.

```
(adb) $m
job 0: address map display
       ? map       'a.out'
b0 = 80001000  e0 = 80006000  f0 = 00001000  r x text
b1 = 80006000  e1 = 80008000  f1 = 00006000  rw  data
      / map       'core'
b0 = 00000000  e0 = ffffffff  f0 = 00000000      none
(adb)
```

Related commands

None

$n

Display or set maximum number of processors allocated for a process

Syntax

[*count*] `$n`

Parameter

None

Prefix argument	Meaning
count	The *count* specifies the maximum number of processors a program can use while you are debugging it. The number must be in the range 1 < *count* < physical number-of-processors.

Description

The `$n` command displays or sets the maximum number of processors (CPUs) that a job may use while you are debugging it. With no *count* specified, `adb` displays the current number of processors allowed.

When you specify a *count*, the maximum number of processors is set to that value. If you specify a number greater than the number of processors available on the system, the value is unchanged.

The value is a global value, not a per-process value. Setting this value while a process is running does not affect the current process.

Examples

1. Display the maximum number of processors for the current process.

```
(adb) $n
maximum number of processors: 1
(adb)
```

2. Change the maximum number of processors to 1.

```
(adb) 1$n
maximum number of processors: 1
(adb)
```

Related commands

None

$o

Toggle the operating modes between chained and sequential

Syntax

`$o`

Parameter

None

Prefix argument

None

Description

The `$o` command toggles the operating mode between chained and sequential, which refers to the hardware execution mode for the program being debugged. It changes the mode of the current program and sets the mode for any new programs to be debugged. To return to the previous mode, use the command again.

By default, `adb` starts in sequential mode.

In sequential mode, `adb` executes one instruction at a time, waiting for the current instruction to finish before starting the next. This is the preferred mode to use while debugging.

In chained mode, however, `adb` uses the pipelining ability of the machine. Thus, it may start executing the next instruction before the current one finishes. In some cases, the current location cannot be reliably predicted when the program stops execution. For this reason, the `:s` command may not work as expected.

Examples

Toggle the operating mode; then change it back.

```
(adb) $o
job 0 operating mode: chained
(adb) $o
job 0 operating mode: sequential
(adb)
```

Related commands

None

$q

Terminate debugging session

Syntax

```
$q
```

Parameter

None

Prefix argument

None

Description

The `$q` command terminates the `adb` debugger. An alternate way to end an `adb` session is by entering **CTRL-d** at the `(adb)` prompt.

Examples

Terminate the debugging session and return to the system prompt.

```
(adb) $q
%
```

Related commands

None

$r

Display general registers for active thread in current job

Syntax

```
$r
```

Parameter

None

Prefix argument

None

Description

The `$r` command displays the contents of the general registers (address and scalar) in hexadecimal format for the current job.

For programs using multiple threads, the `$r` command displays registers from the active thread in the current process.

Examples

1. Display the registers for the current thread.

```
(adb) $r
job 0/0: register display
pc=800014e0 (_thread+0x2)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd54 a1=ffffffff a2=00000012 a3=00000000 tt=000eb61e
a4=000eb6e7 a5=803d9130 ap=ffffcd68 FP=ffffcd54
s0=0000000000000000 s1=0000000000015f90 s2=0000046000000002
s3=000000d600000000 s4=0000000500000044 s5=0000000000000000
s6=0000000000000000 s7=0000000000000000
(adb)
```

2. Change the current thread to thread 1 and display the registers.

```
(adb) ,1:f;$r
job 0/1: register display
pc=800014e0 (_thread+0x2)
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd54 a1=0000000a a2=00e60000 a3=000e0080 tt=00000005
a4=0001ffff a5=001cd5a8 ap=ffffcd68 FP=ffffcd54
s0=0000000000000007 s1=0000000000000000 s2=00000000800014de
s3=ffffffffffffffff s4=000000000000000f s5=0000000000000001
s6=0000000000000001 s7=ffffffff00000006
(adb)
```

Related commands

`$R` Display general registers for all threads in current job.

$R

Display general registers for all threads in the current job

Syntax

```
$R
```

Parameter

None

Prefix argument

None

Description

The `$R` command displays the values of the general registers for all threads in the current job. For jobs with a single thread, the `$R` command is equivalent to the `$r` command.

To view the registers used by another job, first bring it to the foreground using the `:f` command.

Examples

Display the contents of the general registers. This job uses two threads.

```
(adb) $R

job 0/0: register display
pc=8000150a ps=03909080 (EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd54 a1=80006078 a2=000f4240 a3=80008808
tt=000f2de4
a4=800193e4 a5=803d9130 ap=ffffcd68 FP=ffffcd54
s0=00000000000042f7 s1=00000000000186a0
s2=0000044200000002
s3=000000d600000000 s4=0000000500000045
s5=0000000000000003
s6=0000000000000000 s7=0000000000000000

job 0/1: register display
pc=800014e0
ps=83909080 (C,EF,SEQ,SC,DZE,FE,SQS,RES)
sp=ffffcd54 a1=0000000a a2=00003f00 a3=00224721
tt=00000010
a4=00003e0d a5=001a8618 ap=ffffcd68 FP=ffffcd54
s0=0000000000000004 s1=0000000200000001
s2=0000000000000001
s3=0000000200000001 s4=0007bc06001ba978
s5=0000000000000001
s6=0000000000000001 s7=0000000000000000
(adb)
```

Related commands

`$j`	Display status of all jobs.
`$r`	Display general registers for active thread in current job.
`:f`	Bring a job or thread to the foreground.

Syntax

`$s` [*reg*]`?` *format* [*radix*]

Parameter	Meaning
reg	The *reg* parameter is the number of the address register (0-7). If *reg* is unspecified, all registers are displayed.
format	The *format* specifies the manner in which the information is displayed. The format can assume one of the following values: `d` Display a byte of data (may specify an optional radix) `h` Display a halfword of data (may specify an optional radix) `w` Display a word of data (may specify an optional radix) `f` Display a 32-bit value as a floating-point number (same as using `wf`) `l` Display a longword of data (may specify an optional radix). `F` Display a 64-bit values as a floating-point number. Same as using `-lf`)
radix	The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The radix may assume one of the following values: `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default *radix* value is hexadecimal (`x`). To set a new default *radix* use the `$x` command.

Prefix argument

None

Description

The `$s?` command displays the values of one or all of the scalar registers. To display the value of a single register, specify a register number as the *reg* parameter. If you don't specify a register, the values of all scalar registers are displayed.

The amount of data displayed depends on the format and radix selected. Because the size of the scalar registers is 64 bytes (longword), you can display the value in any format.

Examples

1. Display the values of all scalar registers as longwords in the default radix (hexadecimal).

```
(adb) $s?l
s0=0000000000000064 s1=0000000000000001
s2=0000000c00000000
s3=0000000000000077 s4=000000008000a280
s5=000000000012d687
s6=00000000ff7fed54 s7=0000000080008189
(adb)
```

2. Display the values of all scalar registers as halfwords using an unsigned decimal radix.

```
(adb) $s?hu
s0=00100 s1=00001 s2=00000 s3=00119
s4=41600 s5=54919 s6=60756 s7=33161
(adb)
```

3. Display the value of the s4 register as a word using a signed octal radix.

```
(adb) $s4?wq
s4=-17777656600
(adb)
```

4. Display the value of the s4 register as a word using a floating-point radix.

```
(adb) $s6?wf
s6=  -8.504635e+37
(adb)
```

Related commands

`$a=`	Modify value of an address register.
`$a?`	Display the contents of an address register.
`$h=`	Assign value to a hardware communication register.
`$h?`	Display the contents of a hardware communication register.
`$r`	Display the contents of general registers.
`$s?`	Display the contents of a scalar register.
`$v=`	Modify a vector register.
`$v?`	Display the contents of a vector registere.

Syntax

`$s` *reg* = *format value*

Parameter	Meaning
reg	The *reg* parameter is the number of the address register (0-7). If *reg* is unspecified, all registers are displayed.
format	The *format* determines the amount of data to be stored. For scalar registers, you must use the `l` format.
value	The *value* is the number you want stored in the register.

Prefix argument

None

Description

The `$s=` command modifies the value of a scalar register. You can only store longwords of data in a scalar register, so the format must always be `l`.

Scalar registers are multipurpose registers that are used by the compiler to perform intermediate calculations; they can also be used as temporary variables. The S0 register stores return values from functions.

Examples

1. Modify the value of scalar register S5 and verify the change. The value is expressed as a decimal number (the 0T prefix).

```
(adb) $s5=l 0t1234567
s5=000000000012d687
(adb) $s5?lt
s5=        1234567
(adb)
```

2. Modify the scalar register S6 and verify the change. The value is expressed as a hexadecimal number (the default).

```
(adb) $s6=l 00000000ff7fed54
s6=00000000ff7fed54
(adb) $s6?l
s6=00000000ff7fed54
(adb)
```

3. Assign a value with a w format to scalar register S5. It is an illegal assignment.

```
 (adb) $s5=w 0t12345
bad format
(adb)
```

Related commands

`$a=`	Modify value of an address register.
`$a?`	Display the contents of an address register.
`$h=`	Assign value to a hardware communication register.
`$h?`	Display the contents of a hardware communication register.
`$r`	Display the contents of general registers.
`$s?`	Display the contents of a scalar register.
`$v=`	Modify a vector register.
`$v?`	Display the contents of a vector register.

$t

Search for a symbol in the current job's symbol table

Syntax

`$t` [*symbol*]

Parameter	Meaning
symbol	A *symbol* is the name of a function or variable used in the executable program or any libraries included when loading the program.

Prefix argument

None

Description

The `$t` command searches the symbol table of the current job for the specified *symbol*. Symbols include function and variable names used in the program itself and any libraries included when loading the program. The values of static symbols are also displayed.

With no parameters, `$t` displays the values of all symbols in the symbol table.

Examples

1. Display the values of all program symbols.

```
(adb) $t
.
.
.
80009960 D __sctab
800073d8 T _tolower
80002088 T __cvt
80006910 T _ioctl
80002096 T __cvt$n
(adb)
```

2. Display the value of the variable `_stop_program`.

```
(adb) $t _stop_program
8000a858 B _stop_program
(adb)
```

3. Display the symbol information for the `_Get_lesson` routine.

```
(adb) $t _Get_lesson
80001238 T _Get_lesson
(adb)
```

Related commands

`$e`	Display the names and values of all external variables.
`$i`	Display the names and values of all nonzero internal variables.

Syntax

`$v` [*reg*][: *starting-location*[, *count*]] `?` *format* [*radix*]

Parameter	Meaning
reg	The *reg* parameter is the number of the address register (0-7). If *reg* is unspecified, all registers are displayed.
starting-location	The *starting-location* specifies at which element of the register to start displaying data. Each vector register contains 128 64-bit elements [0-127]. This value is interpreted according to the current input radix. If *starting-location* is unspecified, display starts at element 0.
count	The *count* specifies the number of elements to display. This value is interpreted according to the current input radix. If *count* is unspecified, all elements after *starting-location* are displayed.
format	The *format* specifies the manner in which the information is displayed. The format can assume one of the following values.: `b` Display a byte of data (may specify an optional radix) `h` Display a halfword of data (may specify an optional radix) `w` Display a word of data (may specify an optional radix) `l` Display a longword of data (may specify an optional radix) `f` Display a 32-bit value as a floating-point number (same as using `wf`) `F` Display a 64-bit value as a floating-point number. Same as using `lf`).

radix

The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The *radix* may assume one of the following values:

`x`	Hexadecimal (default)
`t`	Signed decimal
`u`	Unsigned decimal
`q`	Signed octal
`o`	Unsigned octal
`f`	Floating-point (valid only for words and longwords; same as using `wf`)
	Display a 64-bit value as a floating-point number (same as using `lf`)

The default *radix* value is hexadecimal (`x`). To set a new default radix use the `$x` command.

Description

The `$v?` command displays the contents of one or more elements in a vector register. Each register contains 128 elements. To display the value for a single element, you must specify the starting location and a count of 1.

If you do not specify a register, the values for all vector registers are displayed. If you do not specify a starting location, the display of the register starts with element 0 (zero). If you don't specify the count, `adb` displays all elements between the starting location and element 127.

Examples

1. Display the first 16 elements of vector register 0. The current input radix is hexadecimal.

```
(adb) $v0:0,10?lx
job 0: vector register display for thread 0
vl=00000040 vs=00000008 vf=00000120
vm=00000000000000000000000000000000
v0[000]=00000a0000140008
v0[001]=63756e6261746368
v0[002]=0000000000000a01
v0[003]=0010000666756c6c
v0[004]=6673000000000a02
v0[005]=001400087072696e
v0[006]=7475696400000000
v0[007]=00000a0300100005
v0[008]=7364617465000000
v0[009]=00000a04000c0003
v0[010]=7369670000000a05
v0[011]=0010000474696d65
v0[012]=0000000000000a06
v0[013]=0010000474647070
v0[014]=0000000000000a07
v0[015]=0010000574647265
(adb)
```

2. Display the last 10 elements of vector register 0. The display begins at the 118th element and goes to the 127th element. You must use `0t` to specify a decimal.

```
(adb) $v0:0t118?lx
job 0: vector register display for thread 0
vl=00000040 vs=00000008 vf=00000120
vm=00000000000000000000000000000000
v0[118]=000009fb000c0002
v0[119]=68670000000009fc
v0[120]=0010000472706364
v0[121]=00000000000009fd
v0[122]=000c000373756e00
v0[123]=000009fe000c0002
v0[124]=6331000000000ff
v0[125]=001c00096d6b6772
v0[126]=6461746573000000
v0[127]=0000000000000000
(adb)
```

3. Display element 0 of each of the vector registers as words in decimal.

```
(adb) $v:0,1?wt
job 0: vector register display for thread 0
vl=00000040
vs=00000008
vm=00000000000000000000000000000000
v0[000]=  1310728
v1[000]=        0
v2[000]=        0
v3[000]=-1391165441
v4[000]=        0
v5[000]=        0
v6[000]=        0
v7[000]=        0
(adb)
```

Related commands

`$a=`	Modify value of an address register.
`$a?`	Display the contents of an address register.
`$h=`	Assign value to a hardware communication register.
`$h?`	Display the contents of a hardware register.
`$r`	Display the contents of general register.
`$s=`	Modify a scalar register.
`$s?`	Display the contents of a scalar register.
`$v=`	Modify a vector register.

Syntax

`$v` *reg* [:*element*] = *format value*

Parameter	Meaning
reg	The *reg* parameter specifies the vector register [0-7] to be modified.
element	The *element* specifies at which element [0-127] in the vector register you want to start modifying data.
format	The *format* determines the amount of data to be stored in the vector register element. For vector registers, you must use format `l`.
value	The *value* is the number you want stored in the register.

Prefix argument

None

Description

The `$v=` command modifies the value of an element in a vector register. If you do not specify an element number, the value is assigned to element 0.

Examples

Assign 0 to the first element of vector register 0.

```
(adb) $v0:0=l 0
v0[000]=000000000000000
(adb)
```

Related commands

`$a=`	modify the value of an address register
`$a?`	display the contents of an address register
`$h=`	assign value to a hardware communication register
`$h?`	display the contents of a hardware communication register
`$r`	display contents of general register
`$s=`	modify a scalar register
`$s?`	display the contents of a scalar register
`$v?`	display the contents of a vector register

$x

Modify default input radix

Syntax

radix `$x`

Parameter

None

Prefix argument	Meaning
radix	The *radix* is the number base used to display output data. Acceptable values include: `0` (change to the original value) `8` (octal) `10` (decimal) `16` (hexadecimal).

Description

The `$x` command changes the number base used to input data to `adb`.

When you specify a new value, `adb` interprets it in the current input radix. For example, if the current input radix is 16 and you type

```
(adb)10$x
```

the radix will not be changed because 10 is 16 in hexadecimal. You can use a prefix letter to force interpretation in a specific number base. Both of the following commands change the input radix from 16 to 10.

```
(adb)0xa$x
```

```
(adb)0t10$x
```

The `0t` prefix tells `adb` to interpret the following number as a decimal number. Other valid prefixes include `0x` and `0o`.

Examples

Change the input radix to decimal then to hexadecimal. After the input radix is changed to decimal, the 16 in the next command is interpreted as decimal.

```
(adb) 0t10$x
default input radix = 10 (base 10)
(adb) 16$x
default input radix = 16 (base 10)
(adb)
```

Related commands

`$X` Modify default output radix.

$X

Modify default output radix

Syntax

radix `$X`

Parameter	Meaning
None	

Prefix argument	Meaning
radix	The *radix* is the number base used to display output data. Acceptable values include: `0` (change to the original value) `8` (octal) `10` (decimal) `16` (hexadecimal)

Description

The `$X` command changes the number base used to display `adb` output.

When you specify a new value, `adb` interprets it in the current input radix. For example, if the current input radix is 16 and you type

```
(adb)10$X
```

the *radix* will not be changed because 10 is 16 in hexadecimal. You can use a prefix letter to force interpretation in a specific number base. Both of the following commands change the output radix to 10 (decimal) when the input radix is 16.

```
(adb)0xa$X
```

```
(adb)0t10$X
```

The `0t` prefix tells `adb` to interpret 10 as a decimal number. Other valid prefixes include `0x` (hexadecimal) and `0o` (octal).

Examples

Display an address register, then change the output radix and redisplay the register.

```
(adb) $a1?w
a1=0000000a
(adb) 0t10$X
output radix = 10 (base 10)
(adb) $a1?w
a1=     10
(adb)
```

Related commands

`$x` Modify default input radix.

Syntax

[*address*] [, *count*]? *format* [*radix*]

Parameter	Meaning
format	*format* specifies the manner in which the information is to be displayed. The *format* can assume any of the values described in the section titled "Data display formats" on page 40.
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of the following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (`x`). To set a new default radix use the `$X` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?` command displays data from *objfile* memory starting at dot. By specifying an address, you can move the location of dot before displaying data.

Examples

1. Display one instruction at the current location of *dot*.

```
(adb) ?i
_main+0x18:    ld.w  12(FP),a6
(adb)
```

2. Display five instructions starting at *_main*.

```
(adb) _main,5?i
_main:      sub.w  #0,a0
sub.w  s0,s0
st.w   s0,_stop_program
mov    a0,a6
pshea  0x0
(adb)
```

3. Display the value of `_total_frames` as a decimal number. `_total_frames` is an integer type.

```
(adb) _total_frames?wt
_total_frames: 44
(adb)
```

4. Display the value of `_lesson_name` as a string.

```
(adb) _lesson_name?s
_lesson_name:  file_structure
(adb)
```

5. Display the values in the four-element array `_branch` as decimal numbers. `_branch` is an array of integers.

```
(adb) _branch,4?wt
_branch:    0      100     500     1000
(adb)
```

Related commands

`/` Display data from *corefile*.

Syntax

[*address*]?= *format value* [...]

Parameter	Meaning
format	The *format* parameter specifies the amount of data to write. Valid values for *format* areas follows: b Byte h Halfword w Word l Longword
value	The *value* is the data to be written to memory.
Prefix argument	**Meaning**
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The?= command writes data to the object file memory starting at dot. You can write any number of *values* to memory; each value uses the amount of memory specified by the *format*. If a *value* is smaller than the specified *format*, the high-order bits are padded with zeros. If a *value* is larger than the specified *format*, the value is truncated.

You can use the *address* prefix argument to change the location of dot before writing the data.

Examples

1. Assign three values to the object file memory starting at `_main`.

```
(adb) _main?=w 123 456 789
_main:        5ac05b80    =      123
_main+0x4:    36408000    =      456
_main+0x8:    a8585086    =      789
(adb)
```

2. Assign five values to the object file memory starting at `_main`. The format is `b` and that some of the values are larger than a byte.

```
(adb) _main?=b 0 1 1234
_main:        0     =     0
_main+0x1:    0     =     1
_main+0x2:    1     =     34
adb)
```

Related commands

`/=` Write information into *corefile* memory.

?a

Display value of *dot* from *objfile* in symbolic form

Syntax

[*address*][,*count*]`?a`

Parameter	Meaning
Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?a` command displays the value of *dot* from the object file in symbolic form.

Examples

Display *dot*'s value in symbolic form twice at address `_settab`.

```
(adb) _settab,2?a
_settab:        _settab:        _settab:
```

Related commands

None

?b

Display a byte, specified by *radix*, from *objfile*

Syntax

[*address*][,*count*]`?b` *radix*

Parameter	Meaning
radix	The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The radix may assume one of the following values: `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default *radix* value is hexadecimal (`x`). To set a new default *radix* use the `$x` command.
Prefix argument	**Meaning**
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?b` command displays a byte specified by *radix* in the object file.

Examples

Display a halfword at addess `_settab` using the `?b` command with a *b* radix.

```
(adb) _settab?bb
_settab:                    15   80
(adb)
```

Related commands

None

?c

Display a character specified by *address*, from *objfile*

Syntax

[*address*][*count*]?c

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The ?c command displays a character from the corefile addressed by *address*.

Examples

Display a character using the ?c command wih an address of _fputc.

```
(adb)fputc?c
14bx14b:
(adb)
```

Related commands

None

Syntax

[*address*][*count*]?C

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The ?C command displays a character addressed by *address*. Unlike the ?c command, however, the ?C command uses the standard escape convention where control characters are displayed as ^X and the delete character is displayed as ^?.

Examples

Display a character using the ?c command wih an address of _fputc.

```
(adb)fputc?c
14bx14b:                              ^U
(adb)
```

Related commands

None

?f

Display 32-bit value as a floating-point number from *objfile*

Syntax

[*address*][*count*]`?f`

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?f` command displays a 32-bit floating-point number from the core file.

Examples

Display the double-precision floating-point at address `_settab`, three times, using the `?f` command.

```
(adb) _settab,3?f
_settab:    1.29245837e-26  5.36347859e-29  6.81785936e-09
(adb)
```

Related commands

None

?F

Display double-precision floating-point from *objfile*

Syntax

[*address*][*count*]`?F`

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?F` command displays a double-precision floating-point number from the object file, specified by *address*.

Examples

Display the double-precision floating-point at address `_settab`, five times, using the `?F` command.

```
(adb) _settab,5?F
_settab:    9.96742373303358e-206 4.27380644960922e-48
            9.96742373303358e-206 4.27380644960922e-48
            0.06742373303358e-206
(adb)
```

Related commands

None

Syntax

[*address*]`?g` *format value* [*mask*]

Parameter	Meaning
format	The *format* specifies the size of data to match. The format can assume one of the following values: `b` Display a byte of data `h` Display a halfword of data `w` Display a word of data `l` Display a longword of data
value	The *value* is the string of digits you want to find. The size of *value* must match the *format* selected.
mask	The *mask* is a digit string that is ANDed with the *value* before the search is performed. Each character you type is converted to binary before ANDing it; the size of each binary representation depends on the selected *format*. For example, a mask of 0012 with a format of *h* is converted to 0000000000010010. If *mask* is omitted, a value of -1 is used.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `?g` command searches *objfile* memory for the first occurrence (after dot) for a pattern. The search is performed as follows:

1. Grab the format and size of data at the specified address.
2. AND it with the mask.

3. Compare the resulting value with the specified value.
4. Increment the address by the format and size and repeat the process if the comparison fails.

Examples

Search for a `calls` instruction (represented by the halfword, 2140). List the instruction to verify the search; list the next instruction to move dot past the `calls` instruction; and repeat the search.

```
% adb example -
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) _runpcs?gh 2140
_runpcs+0x50
(adb).?i
_runpcs+0x50:  calls  _printf
(adb)  RETURN
_runpcs+0x56:  add.w  #12,a0
(adb) ?gh 2140
_runpcs+0xf4
(adb).?i
_runpcs+0xf4:  calls  _validthread
(adb)  RETURN
_runpcs+0xfa:  add.w  #12,a0
(adb)
```

Related commands

`/g` Search for a pattern in the *corefile*.

?h

Display a halfword specified by *radix* from *objfile*

Syntax

[*address*][*count*]`?h`[*radix*]

Parameter	Meaning
radix	
Prefix argument	**Meaning**
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?h` command displays a halfword specified by *radix* in the object file.

Examples

Display a halfword at addess `_settab`, three times, using the `?h` command, with a radix of *b*.

```
(adb) ,3?hb
_settab+0xc:    3238    ff      fc1f    88
(adb)
```

Related commands

None

?i

Display machine instructions from *objfile*

Syntax

[*address*][*count*]`?i`

address — *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address.

You can use symbolic names only if the program executable contains a symbol table.

For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

count — The *count* parameter specifies the number of times to display data.

Description

The `?i` command displays machine instructions in the object file at *address*.

Examples

Display machine instructions at address `_settab` six times, using the `?i` command.

```
(adb)_settab,6?i
                sub.w       #24,a0
                ld.w        #0,s0
                st.w        s0,-4(fp)
                ld.w        -4(fp),s0
                lt.w        #80,s0
                brs.t       0x270
(adb)
```

Related commands

None

?m

Modify segment map parameters in the *objfile*

Syntax

`?m` *n b e f* [`?`/]

Parameter	Meaning
n	Refers to the number of the segment map to be modified. Valid numbers for the segment maps can be found using the `$m` command.
b	Identifies the beginning address of the segment.
e	Identifies the ending address of the segment.
f	Represents the offset in the file of the segment.

Prefix argument

None

Description

The `?m` command changes the values for the beginning and ending addresses and the file offset of a core file segment map. The *n* parameter specifies which segment map to modify. The valid map numbers can be found using the `$m` command, which also shows the current values for *b*, *e*, and *f*, as shown in the following display.

```
(adb) $m
   ? map       'a.out'
b0 = 80001000  e0 = 80006000  f0 = 00001000  r x text
b1 = 80006000  e1 = 80008000  f1 = 00006000  rw  data
```

The *objfile* has two segment maps (0,1), and for each segment map, the values for *b*, *e*, and *f* are displayed.

When you terminate the command with `a?` or `/`, you force `adb` to use the identified (*objfile* or *corefile*, respectively) file for all subsequent requests. For example, the command `?m1?` instructs `adb` to use the object file for all requests, even if the command specifies the core file. After issuing this command, the command `/=w 35` actually writes data to the object file, not the core file.

Examples

Assign new values to segment map 0 in the object file; then, verify with the `$m` command.

```
(adb) ?m0 80002000 80009000 00003000
(adb) $m
job 0: address map display
 ? map       'a.out'
b0 = 80002000  e0 = 80009000  f0 = 00003000  r x text
b1 = 80008000  e1 = 8000a000  f1 = 00008000  rw  data
/ map       'core'
b0 = 00000000  e0 = ffffffff  f0 = 00000000      none
(adb)
```

Related commands

`/ m` Modify segment map parameters in the *corefile*.

?n

Display a newline from *objfile*

Syntax

[*address*][*count*]`?n`

address — *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address.

You can use symbolic names only if the program executable contains a symbol table.

For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

count — The *count* parameter specifies the number of times to display data.

Description

The `?n` command displays a newline in the object file.

Examples

Use the `?n` command to display a newline, at address `_settab`, from an object file.

```
(adb)_settab?n
_settab:
(adb)
```

Related commands

None

?P

Display the addressed value from *objfile* in symbolic form

Syntax

[*address*][*count*]`?P`

address — *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address.

You can use symbolic names only if the program executable contains a symbol table.

For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

count — The *count* parameter specifies the number of times to display data.

Description

The `?P` command displays the value at *address* in symbolic form.

Examples

Use the `?P` command to display the value at address `_settab` in symbolic form.

```
(adb)?P
_settab:                    0x15800018
(adb)
```

Related commands

None

?r

Display a space from *objfile*

Syntax

[*address*][*count*]`?r`

address — *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address.

You can use symbolic names only if the program executable contains a symbol table.

For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

count — The *count* parameter specifies the number of times to display data.

Description

The `?r` command displays a space from the object file.

Related commands

None

?s

Display addressed characters from *objfile* until 0 is reached

Syntax

[*address*][*count*]?s

address — *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address.

You can use symbolic names only if the program executable contains a symbol table.

For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

count — The *count* parameter specifies the number of times to display data.

Description

The ?s command is used with theobject file, and displays addressed characters until 0 is reached.

Related commands

None

?S

Display string from *objfile* using ^X escaqpe convention

Syntax

[*address*][*count*]?S

address — *address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address.

You can use symbolic names only if the program executable contains a symbol table.

For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

count — The *count* parameter specifies the number of times to display data.

Description

The `?S` command displays a string from the object file using the ^X escape convention.

Examples

Usingthe `?S` command, display a string from the object file at address `_settab`, ten times.

```
(adb)_settab,10?S
_settab:    ^U^@^X^Q^H68^?|28^?|^_^HPw428^?|68^?t^Q^H^H68^?p28^?p29^?tQ^J_^B
]^B[^Q^[^Iw^F^Q^H68^?lq^E^Q^H^A68^?l*=^?|28^?l\e6h28^?|68^?x28^?|^T^H^A68^?|2
8^? |^_^H
(adb)
```

Related commands

None

?t

Tabs to the next stop in *objfile*

Syntax

[*integer*]`?t`

integer — *integer* represents any integer.

Description

The `?t` command tabs to the next stop in the object file.

Related commands

None

?w

Display a word from *objfile* specified by *radix*

Syntax

[*address*][,*count*]?w[*radix*]

Parameter	Meaning
radix	The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The radix may assume one of the following values: `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default *radix* value is hexadecimal (`x`). To set a new default *radix* use the `$x` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?w` command displays a word specified by *radix* in the object file.

Examples

Display a word at addess `_settab`, three times, using the `?w` command.

```
(adb) _settab,3?w
_settab:    15800018    11880000    3638fffc
(adb)
```

Related commands

`?h` Display halfword in object file.

Syntax

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

Parameter	Meaning
...	*string* represents any existing string in the object file.
Prefix argument	**Meaning**
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?"..."` command displays the string eclosed in quotes from the object file.

Examples

Display a string frm the object file using the `?"..."` command.

```
(adb) ?"putg.c"
0x1fc;                          putg.c
(adb)
```

Related commands

/"..." Display enclosed string from the object file.

Syntax

[*address*][,*count*]?^

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?"..."` command causes *dot* in the object file to be incremented by the current increment.

Related commands

`?+`	*dot* in *objfile* is incremented by one.
`?-`	*dot* in *objfile* is decremented by one.

Syntax

[*address*][,*count*]?+

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?+` command causes *dot* in the object file to be incremented by one.

Related commands

`?^`	*dot* in *objfile* is incremented by the current increment.
`?-`	*dot* in *objfile* is decremented by one.

Syntax

[*address*][,*count*]?-

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?-` command causes *dot* in the object file to be decremented by one.

Related commands

`?+`	*dot* in *objfile* is incremented by one.
`?^`	*dot* in *objfile* is incremented by the current increment.

Syntax

[address][, count] / format [radix]

Parameter	Meaning
format	The *format* specifies the manner in which the information is to be displayed. The *format* can assume any of the values described in the section titled "Data display formats" on page 40.
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of he following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (x). To set a new default radix use the `$x` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
,count	The *,count* parameter specifies the number of times to display data.

Description

The / command displays data from *corefile* memory starting at dot. By specifying an *address*, you can move the location of dot before displaying data.

/

Examples

1. Display one instruction at the current location of dot.

```
(adb) /i
_main+0x18:    ld.w  12(FP),a6
(adb)
```

2. Display five instructions starting at `_main`.

```
(adb) _main,5/i
_main:      sub.w  #0,a0
sub.w    s0,s0
st.w     s0,_stop_program
mov      a0,a6
pshea    0x0
(adb)
```

3. Display the value of `_total_frames` as a decimal number. `_total_frames` is an integer type.

```
(adb) _total_frames/wt
_total_frames: 44
(adb)
```

4. Display the value of `_lesson_name` as a string.

```
(adb) _lesson_name/s
_lesson_name:  file_structure
(adb)
```

5. Display the values in the four-element array *_branch* as decimal numbers. `_branch` is an array of integers.

```
(adb) _branch,4/wt
_branch:    0      100     500     1000
(adb)
```

Related commands

`?` Display data from *objfile*.

/=

write information into corefile memory

Syntax

[*address*] /= *format value...*

Parameter	Meaning
format	The *format* parameter specifies the amount of data to write. Valid values for *format* include: `b` Byte `h` Halfword `w` Word `l` Longword
value	The *value* is the data to be written to memory.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The /= command writes data into the object file memory starting at dot. You can write any number of value *s* to memory; each value uses the amount of memory specified by the *format*. If a *value* is smaller than the specified *format*, the high-order bits are padded with zeros. If a *value* is larger than the specified *format*, the *value* is truncated.

You can use the *address* prefix argument to change the location of dot before writing the data.

Examples

1. Assign three values to the core file memory starting at *_main*.

```
(adb) _main/=w 123 456 789
_main:        5ac05b80    =      123
_main+0x4:    36408000    =      456
_main+0x8:    a8585086    =      789
(adb)
```

2. Assign five values to the core file memory starting at *_main*. The format is `b` and some of the values are larger than a byte.

```
(adb) _main/=b 0 1 1234
_main:        0     =     0
_main+0x1:    0     =     1
_main+0x2:    1     =     34
(adb)
```

Related commands

`?=` Write information into *objfile* memory.

/a

Display value of *dot* from *corfile* in symbolic form

Syntax

[*address*][,*count*]/a[*radix*]

Parameter	Meaning
radix	The *radix* specifies the number base for the display. A radix is only valid for the `b`, `h`, and `w` formats. The radix may assume one of the following values: `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default *radix* value is hexadecimal (`x`). To set a new default *radix* use the `$x` command.
Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/a` command displays the value of *dot* from the core file in symbolic form.

Examples

Display *dot*'s value in symbolic form twice at address `_settab`.

```
(adb) _settab,2/a
_settab:        _settab:        _settab:
```

Related commands

None

/b

Display a byte, specified by *radix*, from the corefile

Syntax

[*address*][,*count*]/b[*radix*]

Parameter	Meaning
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of he following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (`x`). To set a new default radix use the `$X` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/b` command displays a byte specified by *radix* from the corefile.

Examples

Display a halfword at addess `_settab` using the /b command with a *b* radix.

```
(adb) _settab/bb
_settab:                    15   80
(adb)
```

Related commands

None

/c

Display a character specified by *address*, from *corfile*

Syntax

[*address*][*count*]/c

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/c` command displays a character addressed by *address*.

Examples

Display a character using the `?c` command wih an address of `_fputc`.

```
(adb)fputc/c
14bx14b:
(adb)
```

Related commands

None

/C

Display addressed character from *corfile* using ^X

Syntax

[*address*][*count*]/C

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The /C command displays a character addressed by *address*. Unlike the ?c command, however, the ?C command uses the standard escape convention where control characters are displayed as ^X and the delete character is displayed as ^?.

Examples

Display a character using the /c command wih an address of _fputc.

```
(adb)fputc/c
14bx14b:                              ^U
(adb)
```

Related commands

None

/f

Display 32-bit value as a floating-point number from *corfile*

Syntax

[*address*][*count*]`/f`

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/f` command displays a 32-bit floating-point number from the object file.

Examples

Display the double-precision floating-point at address `_settab`, three times, using the `/f` command.

```
(adb) _settab,3/f
_settab:    1.29245837e-26  5.36347859e-29  6.81785936e-09
(adb)
```

Related commands

None

Syntax

[*address*][*count*]/F

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The /F command displays a double-precision floating-point from the core file.

Examples

Display the double-precision floating-point at address _settab, five times, usigthe ?F command.

```
(adb) _settab,5/F
_settab:    9.96742373303358e-206 4.27380644960922e-48
            9.96742373303358e-206 4.27380644960922e-48
            0.06742373303358e-206
(adb)
```

Related commands

None

Syntax

[*address*] /g *format value* [*mask*]

Parameter	Meaning
format	The *format* specifies the size of data to match. The format can assume one of the following values. `b` Display a byte of data `h` Display a halfword of data `w` Display a word of data `l` Display a longword of data
value	The *value* is the string of digits you want to find. The size of *value* must match the *format* selected.
mask	The *mask* is a digit string that is ANDed with the *value* before the search is performed. Each character you type is converted to binary before ANDing it; the size of each binary representation depends on the selected *format*. For example, a mask of 0012 with a format of `h` is converted to 0000000000010010. If *mask* is omitted, a value of -1 is used.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `/g` command searches *corefile* memory for the first occurrence (after *dot*) for a pattern. The search is performed as follows:

1. Grab the format and size of data at the specified address.
2. AND it with the mask.
3. Compare the resulting value with the specified value.
4. Increment the address by the format and size and repeat the process if the comparison fails.

Examples

Search for a `calls` instruction (represented by the halfword, 2140). List the instruction to verify the search; list the next instruction to move dot past the `calls` instruction; and repeat the search.

```
% adb example -
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) _runpcs/gh 2140
_runpcs+0x50
(adb)./i
_runpcs+0x50:  calls  _printf
(adb)  RETURN
_runpcs+0x56:  add.w  #12,a0
(adb) /gh 2140
_runpcs+0xf4
(adb)./i
_runpcs+0xf4:  calls  _validthread
(adb)  RETURN
_runpcs+0xfa:  add.w  #12,a0
(adb)
```

Related commands

`?g` Search for a pattern in the *objfile*.

/h

Display a halfword specified by *radix* from *corfil*

Syntax

[*address*][*count*]/h

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The /h command displays a halfword specified by *radix* from the core file.

Examples

Display a halfword at addess _settab, three times, using the /h command.

```
(adb) _settab,3/h
_settab:    1580      18      1188
(adb)
```

Related commands

None

Syntax

[*address*][*count*]`/l`

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/l` command displays the contents of *dot* as machine instructions from the core file.

Examples

Display machine instructions at address `_settab` six times, using the `/i` command.

```
(adb)_settab,6/i
                sub.w       #24,a0
                ld.w        #0,s0
                st.w        s0,-4(fp)
                ld.w        -4(fp),s0
                lt.w        #80,s0
                brs.t       0x270
(adb)
```

Related commands

None

/m

modify segment map parameters in the *corefile*

Syntax

/mn b e f [?/]

Parameter	Meaning
n	Refers to the number of the segment map to be modified. Valid numbers for the segment maps can be found using the `$m` command.
b	Identifies the beginning address of the segment.
e	Identifies the ending address of the segment.
f	Represents the offset in the file of the segment.

Prefix argument	Meaning
None	

Description

The `/m` command changes the values for the beginning and ending addresses and the file offset of a core file segment map. The *n* parameter specifies which segment map to modify. The valid map numbers can be found using the `$m` command, which also shows the current values for *b*, *e*, and *f*, as shown in the following display:

```
 / map       'core'
b0 = 80000000  e0 = 80001000  f0 = 00006000  rw  data
b1 = 80001000  e1 = 80008000  f1 = 00007000  r x text
b2 = 80008000  e2 = 8000a000  f2 = 0000e000  rw  data
b3 = 8000a000  e3 = 8000c000  f3 = 00010000  rw  bss
b4 = ffffb000  e4 = ffffd000  f4 = 00012000  rw  bss
b5 = ffffd000  e5 = ffffe000  f5 = 00014000  r x text
```

The *corefile* has six segment maps (0 through 5) and for each segment map, the values for *b*, *e*, and *f* are displayed.

When you terminate the command with a `?` or `/`, you force `adb` to use the identified (*objfile* or *corefile*, respectively) file for all subsequent requests. For example, the command `/m1?` instructs `adb` to use the object file for all requests, even if the command specifies the core file. After issuing the `/m1?` command, the command `/=w 35` actually writes data to the object file, not the core file.

Examples

Assign new values to segment map 0 in the object file; then verify with the `$m` command.

```
(adb) /m0 80002000 80009000 00003000
(adb) $m
job 0: address map display
 ? map       'core.e'
b0 = 80001000  e0 = 80005000  f0 = 00001000  r x text
b1 = 80005000  e1 = 80007000  f1 = 00005000  rw  data     /
 map       'core'
b0 = 80002000  e0 = 80009000  f0 = 00003000  rw  data
b1 = 80001000  e1 = 80008000  f1 = 00007000  r x text
b2 = 80008000  e2 = 8000a000  f2 = 0000e000  rw  data
b3 = 8000a000  e3 = 8000c000  f3 = 00010000  rw  bss
b4 = ffffb000  e4 = ffffd000  f4 = 00012000  rw  bss
b5 = ffffd000  e5 = ffffe000  f5 = 00014000  r x text
(adb)
```

Related commands

`?m` Modify segment map parameters in the *objfile*.

/n

Display a newline from *corfile*

Syntax

[*address*][*count*]/n

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The /n command displays a newline in the object file.

Examples

Use the /n command to display a newline, at address _settab, from an object file.

```
(adb)_settab/n
_settab:
(adb)
```

Related commands

None

/P

Display the addressed value from *corfil* in symbolic form

Syntax

[*address*][*count*]`/P`

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/P` command displays the value at *address* in symbolic form.

Examples

Use the `/P` command to display the value at address `_settab` in symbolic form.

```
(adb)/P
_settab:                    0x15800018
(adb)
```

Related commands

None

/r

Display a space from *corfile*

Syntax

[*address*][*count*]/r

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/r` command displays a space from the core file.

Related commands

None

/s

Display addressed characters from *corfile* until 0 is reached

Syntax

[*address*][*count*]/s

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/s` command is used with an object file, and displays addressed characters until 0 is reached.

Related commands

None

/S

Display string from *corfile* using ^X escaqpe convention

Syntax

[*address*][*count*]/S

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/S` command displays a string from the core fileusing the ^X escape convention...

Examples

Usingthe `/S` command, display a string from the core file at address `_settab`, ten times.

```
(adb)_settab,10/S
_settab:     ^U^@^X^Q^H68^?|28^?|^_^HPw428^?|68^?t^Q^H^H68^?p28^?p29^?tQ^J_^B
]^B[^Q^[^Iw^F^Q^H68^?1q^E^Q^H^A68^?1*=^?|28^?1\e6h28^?|68^?x28^?|^T^H^A68^?|2
8^? |^_^H
(adb)
```

Related commands

None

/t

Tabs to the next stop in *corfile*

Syntax

[*integer*]?t

Parameter	Meaning
integer	*integer* represents any integer.

Description

The /t command tabs to the next stop in the core file.

Related commands

None

/w

Display a word from *corfile* specified by *radix*

Syntax

/wradix

Parameter	Meaning
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of he following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (`x`). To set a new default radix use the `$x` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `/w` command displays a word specified by *radix* in the core file.

Examples

Display a halfword at addess `_settab`, three times, using the `/w` command.

```
(adb) _settab,3/w
_settab:    15800018    11880000    3638fffc
(adb)
```

Related commands

`?h` Display halfword in object file.

Syntax

[*address*][,*count*]/[...]

Parameter	Meaning
...	*string* represents a ...

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The /"..." command displays the string eclosed in quotes from the object file.

Examples

Display a string frm the object file using the /"..." command.

```
(adb) ?"putg.c"
0x1fc;                              putg.c
(adb)
```

Related commands

?"..." Display enclosed string from the object file.

dot in *corfile* is incremented by current increment

Syntax

[*address*][,*count*]/^

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?"..."` command causes *dot* in the core file to be incremented by the current increment.

Related commands

/+	*dot* in *corefile* is incremented by one.
/-	dot in *corefile* is decremented by one.

/+

dot in *corfile* is incremented by 1

Syntax

[*address*][,*count*]/+

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?+` command causes *dot* in the core file to be incremented by one.

Related commands

`?^`	*dot* in *corefile* is incremented by the current increment.
`?-`	*dot* in *corefile* is decremented by one.

Syntax

[*address*][,*count*]/-

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?-` command causes *dot* in the object file to be decremented by one.

Related commands

`?+`	*dot* in *corefile* is incremented by one.
`?^`	*dot* in *corefile* is incremented by the current increment.

=a

Display value of *address* itself in symbolic form

Syntax

[*address*]=a

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The =a command displays the value of *address* from the core file in symbolic form.

Examples

Using the =a command, display the value of *address* in symbolic form at `_settab`.

```
(adb) _settab=a
0x8000162c:
(adb)
```

Related commands

None

=C

Display value of *address* as a character

Syntax

[*address*][*count*]=c

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `=c` command displays a character at *address*.

Examples

Display a character using the `=c` command.

```
(adb)=c
14bx14b:                              u
(adb)
```

Related commands

None

=C

Display addressed character at *address* using ^X

Syntax

[*address*][*count*]=C

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `=C` command displays a character at *address*. Unlike the `?c` command, however, the `?C` command uses the standard escape convention where control characters are displayed as ^X and the delete character is displayed as ^?.

Examples

Display a character using the `=C` command.

```
(adb)=C
14bx14b:                                u    ^U
(adb)
```

Related commands

None

=f

Display 32-bit value as a floating-point number at *address*

Syntax

[*address*][*count*]=f

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The =f command displays a 32-bit floating-point number at *address*.

Examples

Display the double-precision floating-point at address _settab using the =f command.

```
(adb) _settab=f
              Rop0x162c
(adb)
```

Related commands

None

Syntax

[*address*][*count*]=F

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `=F` command displays a double-precision floating-point number at *address*.

Examples

Display the double-precision floating-point at address `_settab` using the `=F` command.

```
(adb) _settab=F
              0.00000000000000000e+00
(adb)
```

Related commands

None

=h

Display a halfword specified by *radix* at *address*

Syntax

[*address*][*count*]=h[*radix*]

Parameter	Meaning
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of he following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (`x`). To set a new default radix use the `$X` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The `?=` command displays a halfword specified by *radix* at *address*.

Examples

Using the =h command, display a halfword at *addess*, with a radix of *b*.

```
(adb)=hb
8000          80
(adb)
```

Related commands

None

=i

Display as machine instructions at *address*

Syntax

[*address*]=i

Parameter	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The =i command displays machine instructions at *address*.

Examples

Display machine instructions at address _settab, using the =i command.

```
(adb)_settab,3=i
                mov         v0,s0,s0
(adb)
```

Related commands

None

=n

Display a newline at *address*

Syntax

[*address*][*count*]=n

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.
count	The *count* parameter specifies the number of times to display data.

Description

The =n command displays a newline at *address*.

Examples

Use the =n command to display a newline at address _settab.

```
(adb)_settab=n
_settab:
(adb)
```

Related commands

None

=P

Display value of *address* in symbolic form

Syntax

[*address*]=P

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The =P command displays the value at *address* in symbolic form.

Examples

Use the =P command to display the value at address _settab in symbolic form.

```
(adb)_settab=P
                    0x8000162c
(adb)
```

Related commands

None

=r

Display a space at *address*

Syntax

[*address*]=r

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `=r` command displays a space at *address*.

Related commands

None

=S

Display addressed characters at *ddress* until 0 is reached

Syntax

[*address*]=s

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (_main, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The ?s command displays characters at *address* until 0 is reached.

Related commands

None

=t

Tabs to the next stop at *address*

Syntax

[*integer*][*address*]=t

Parameter

None

Prefix argument	Meaning
integer	*integer* represents any integer.
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `=t` command tabs to the next stop after *address*.

Related commands

None

=W

Display a word at *address* specified by *radix*

Syntax

[*address*]=w[*radix*]

Parameter	Meaning
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of he following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (`x`). To set a new default radix use the `$x` command.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The =w command displays a word specified by *radix* at *address*.

Examples

Display a word at addess `_settab` using the `=w` command.

```
(adb) _settab=w
8000162c                        80
(adb)
```

Related commands

None

=Y

Display four bytes in date format

Syntax

=Y[*radix*]

Parameter	Meaning
radix	The *radix* parameter defines the number base to use if a `b`, `h`, `l`, or `w` format is used. The *radix* may assume one of he following values. `x` Hexadecimal (default) `t` Signed decimal `u` Unsigned decimal `q` Signed octal `o` Unsigned octal `f` Floating-point (valid only for words and longwords) The default radix value is hexadecimal (`x`). To set a new default radix use the `$X` command.

Description

The `=Y` command displays four bytes of data in date format, specified by *radix*.

Examples

Display four byes of data in date format using the `=Y` command.

```
(adb) =Y
            2038 Jan 18 22:48:44
(adb)
```

Related commands

None

=^

dot at *address* is incremented by current increment

Syntax

[*address*]=^

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `/^` causes *dot* at address to be incremented by the current increment. Nothing is displayed.

Related commands

None

=+

dot at *address* is incremented by 1

Syntax

[*address*]=+

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `/+` causes *dot* at address to be incremented by one. Nothing is displayed.

Related commands

None

=-

dot at *address* is decremented by 1

Syntax

[*address*]=-

Parameter

None

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The `/-` causes *dot* at address to be decremented by one. Nothing is displayed.

Related commands

None

)comment

Make a (non-executed) remark

Syntax

) comment

Parameter	Meaning
comment	*Any remark made.*

Prefix argument

None

Description

The) comment command makes a non-executed remark, and is usually used in adb command scripts.

)help

Display command help file

Syntax

```
)help
```

Parameter

None

Prefix argument

None

Description

The `)help` command accesses the online help utility for `adb`. It displays a brief description of each `adb` command. The `adb` help file resides in /usr/lib/adb/helpfile.

The output is piped through the utility defined in the ConvexOS PAGER environment variable. If the PAGER environment variable is not set, `adb` pipes the output through `more`.

Examples

Display the help file by typing `)help`. The command line disappears as the help file is displayed.

```
Adb Helpfile

"$" commands
------------

$b              display breakpoints in the current job.

[adr][,cnt]$c   display a stack backtrace for the current thread. if adr is
                specified, the trace begins at that address. if cnt is
                specified, only cnt frames are displayed. the default for
                adr is the current value of the frame pointer; the default
                for cnt is all frames.

$e              display the values of all non-zero global variables in the
                current job.

$g              get new symbol/core file names. this will change the
                names of both the symbol and core files in the current
                job and reset the data associated with it. a file name
                of "-" should be used if the file should be closed and
                not reopened. a null file name (carriage-return only)
                will cause the file to be unchanged.

                invoking this command will cause the file mappings to
                be reset even if neither of the files are changed.

$h[r]?f[x]      display hardware (communications) registers. works the
                same way that the $a, $s, and $v register commands work.

$hr=f value     modify hardware registers. same as $a, $s, and $v. the
                format f must be "l".

$hr[l|u]        lock/unlock hardware register.

$i              display the values of all non-zero internal adb variables.

[job]$j         display the status of job. the default for job is all
                active jobs.

$m              display address mappings in the current job.

[cnt]$n         display the maximum number of processors that may be allocated
                to a process. if cnt is specified, the maximum is set to it's
                value. this value is a global value, not a per-process value.
                setting this value while a process is running has no effect.
```

$o	toggles the operating mode of the current job from sequential to chained or vice-versa, depending on the operating mode.
$r[w]	display registers from the current thread in the current job. if w is specified, symbols are displayed in a "wide" format, where A registers are lined up with S registers.
$R[w]	display registers from all threads in the current job. w is handled the same as for $r.
$t [symbol]	searches the symbol table in the current job for symbol, and display it in an nm(1) style output format. symbol may be either a symbol name or a symbol value. if symbol is not specified, all symbols are displayed.
$vl?f[x]	display the vector length register. works the same way that the $a, $s, and $v register commands work.
$vl=f value	modify the vector length register. same as $a, $s, and $v. the format f must be "b".
$vml?f[x]	display the lower 64 bits of the vector merge register. works the same way that the $vl register command works.
$vml=f value	modify the lower 64 bits of the vector merge register. same as $vl. the format f must be "l".
$vmu?f[x]	display the upper 64 bits of the vector merge register. works the same way that the $vl register command works.
$vmu=f value	modify the upper 64 bits of the vector merge register. same as $vl. the format f must be "l".
$vs?f[x]	display the vector stride register. works the same way that the $vl register command works.
$vs=f value	modify the vector stride register. same as $vl. the format f must be "w".
radix$X	set the output radix. this determines the radix in which numbers are displayed. the legal values for radix are 0 (default), 8 (octal), 10 (decimal), and 16 (hexadecimal). remember that the value of radix is determined by the current input radix.

```
":" commands
------------
adr:b             set a breakpoint at adr in the current job.
[adr][,cnt]:c     continue the current thread in the current job.
[adr][,cnt]:C     continue all threads in the current job.
adr:d             delete the breakpoint at adr in the current job.
:D                delete all breakpoints in the current job.
:e                change the exit dispostion of the current job to RELEASE.
                  that is, do not terminate the job when exiting adb - continue
                  it.

[job][,thd]:f     brings job/thd into the debug foreground. that is, job
                  is made the current job, thd is made the current thread
                  in job. if neither job nor thd are specified, the current
                  job and thread are displayed. the default for job is the
                  current job; the default for thd is the previous current
                  thread for the job.

:i                toggle inherit mode in the current job. when inherit mode
                  is set, this allows the debugging of child processes.

:k                terminate (kill) the current job.

:m                toggle scheduling mode in the current job. scheduling mode may
                  be either fixed or dynamic. it is initially set to fixed.

[adr][,cnt]:r     run a process, starting at adr and stopping when cnt break-
                  points have been hit. any arguments specified after the :r
                  will be passed to the process. the default for adr is the
                  normal entry point for the process. the default for cnt is 1.

[adr][,cnt]:s     single step the current thread.

[adr][,cnt]:S     single step all threads.


")" commands
----------------

comment           introduces a comment line

help              displays this file

static            toggles the static symbol usage flag. when set, all references
                  to symbol names will include static symbols.

status            display status about things like input radix, output radix,
                  inherit mode, etc.
```

Related commands

`)static`	Toggle the static symbol usage flag.
`)status`	Display information about `adb`.
`)comment`	Make a non-executable remark.

)static

Toggle the static symbol usage flag

Syntax

```
)static
```

Parameter

None

Prefix argument

None

Description

The `)static` command toggles the static symbol usage flag. Normally, `adb` doesn't recognize static symbols in a program, because there can be multiple occurrences of the symbol in different files. In this case, `adb` has no way to let you know which one it is using.

When the static symbol is YES, static symbols are included in commands that use symbol names. Static symbols do not have to be unique, and using a static symbol name may generate in incorrect or unexpected result.

By default, `adb` has the toggle set to NO.

Examples

Toggle the static symbol usage flag to YES; then toggle it back.

```
(adb) )static
use static symbols: YES
(adb) )static
use static symbols: NO
(adb)
```

Related commands

`)help`	Display command help file.
`)status`	Display information about `adb`.
`)comment`	Make a non-executable remark.

)status

Display information about adb

Syntax

```
)status
```

Parameter

None

Prefix argument

None

Description

The)status command displays general information about adb, including the current:

- Inherit mode
- Operating mode
- Exit disposition
- Input radix
- Output radix
- Static symbol flag
- Maximum number of processors
- Help file

Examples

Display the current adb status.

```
(adb) )status
inherit mode: OFF
operating mode: sequential
exit disposition: terminate
input radix: 16
output radix: default
static symbols: NO
maximum processors: 2
help file: /usr/lib/adb/helpfile
(adb)
```

Related commands

`$n`	Display or set the maximum number of processors allocated for a process.
`$o`	Toggle the operating modes between chained and sequential
`$x`	Modify default input radix.
`$X`	Modify default output radix.
`:e`	Change the exit disposition of the current job to RELEASE.
`:i`	Toggle the inherit mode.
`)help`	Display the command help file.
`)static`	Toggle the static symbol usage flag.
`)comment`	Make a non-executable remark.

RETURN

Repeat the previous data formatting command

Syntax

RETURN

Parameter

None

Prefix argument

None

Description

Pressing the **RETURN** key repeats the previous data formatting command with a count of 1.

Examples

List five instructions; then list the next instruction.

```
(adb) _main,5?i
_main:
sub.w   #4,a0
sub.w   s0,s0
st.w    s0,-4(FP)
ld.w    -4(FP),s0
le.w    #1000000,s0
(adb)   RETURN
_main+0x12:    brs.t   _main+0x36
(adb)
```

Related commands

None

>

Assign *dot* to a variable or named register

Syntax

[*address*] > *name*

Parameter	Meaning
name	The *name* parameter is the name of an internal or external variable or the name of a register.

Prefix argument	Meaning
address	*address* represents a location in the file. The *address* can be either a number equivalent to the actual machine address or a symbolic name. A symbolic name is a program symbol (`_main`, for example) that has been mapped to an actual address. You can use symbolic names only if the program executable contains a symbol table. For more information concerning addresses, refer to the section titled "adb address expressions" on page 35.

Description

The > command stores the value of dot in the named variable or register. Variables consist of the internal variables (`b`, `d`, `e`, `m`, `s` `t`, and 0 through 9). Registers consist of the address, hardware communication, scalar, and vector registers.

By specifying an address you can move the location of dot before assigning its value.

Examples

Store the location of `_main` in the `S1` register. Then, display the contents of `S1` and `_main` to verify.

```
(adb) _main>s1
(adb) $s1?w
s1=80001954
(adb) _main?w
_main:      5ac05b80
(adb) 80001954?w
_main:      5ac05b80
(adb)
```

>

Related commands None

Syntax

! *shell-command*

Parameter	Meaning
shell-command	A *shell-command* is any valid ConvexOS command (`cat`, for example).

Prefix argument

None

Description

The ! command executes a ConvexOS shell command from within `adb`. `adb` uses the shell defined in the SHELL environment variable. If the SHELL environment variable is undefined, `adb` uses the Bourne shell (/bin/sh).

Examples

1. List the contents of the directory while in `adb`.

```
(adb) !ls
a.out       file_structure thread
cai.c       test.exe     thread.e
!
(adb)
```

2. Send mail while in `adb`.

```
(adb) !mail smith
Subject: just testing
I wanted to see if I could send mail while debugging my
program in adb.
Cc:
!
(adb)
```

Related commands

None

Using adb—C program examples

A

Analyzing a core dump

One of the most common uses of adb is to analyze the contents of a core file produced when a program terminates abnormally. The core file contains an image of the program's state at the time of the termination (the contents of the program's data and stack segments).

Dereferencing a null pointer is a common fault that results in a bus error and a core dump. This error occurs because the offending program attempts to reference location 0 (null), which does not exist. Figure 71 illustrates this error. The program is supposed to determine the length of the string "foo," but the pointer to "foo" was not properly initialized.

Figure 71 C program with pointer bug

```
main()
{
 int i;
char *ptr;
char *foo = "0123456789abcdef";

for (i = 0; *ptr++; ++i);
printf("string length = %d n", i);
}
```

The adb command sequence and corresponding output in Figure 72 illustrate how to use adb to determine the source of the problem.

Figure 72 Sample adb session for core dump program

```
% adb example1
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) $c
_main(1,ffffcdac,ffffcdb4) from start+0x80 [ap = ffffcda0]
(adb) $r
job 0/0: register display

pc=80001198 (_main+0x28)
ps=80908080 (C,XF,SC,DZE,FE,RES)
sp=ffffcd80 a1=00000000 a2=ffffcdb4 a3=fffffce00
a4=00000008 a5=00000040 ap=ffffcda0 fp=ffffcd8c
s0=0000000000000000 s1=0000000000000000 s2=0000000000000000
s3=00000000ffffffff s4=0000000080055c70 s5=00000000800503c0
s6=00000000e0000000 s7=0000000000000000
(adb) 80001198?i
_main+0x28:   eq.w  #0,s0
(adb) _main+20,5?i
 _main+0x20:
sub.w  #1,a1
ld.b  0(a1),s0
cvtb.w s0,s0
eq.w  #0,s0
brs.t  _main+0x3c
(adb) $q
```

The `$c` command displays a procedure backtrace that shows where in the stack the problem occurred. Because this is a simple example, this command is not very helpful. However, more sophisticated programs would benefit from this command. The `$r` command shows the values of all general registers, including the pc. Displaying the instruction at the location referenced by the pc doesn'tt show anything useful, so the next command displays a range of instructions around that location. Two instructions before the one referenced by the pc, we see the ld.b instruction through register a1. Referring back to the register display, it notes the contents of a1 as 0. This is a dereferenced null pointer. Examining the source code, you find that ptr was not initialized to the beginning of the string.

Using breakpoints

The C program in Figure 73 demonstrates how to use breakpoints for monitoring and controlling program execution. The program converts tabs to blanks, and was adapted from the book *Software Tools* by Kernighan and Plauger.

Figure 73 C program to decode tab stops

```
#include <stdio.h>

#define  MAXLINE  80
#define  YES      1
#define  NO       0
#define  TABSP    8
int      tabs[MAXLINE];
int      col, *ptab;
int      c;

main()
{
        ptab = tabs;
        settab(ptab);       /* set initial tab stops */
        col = 1;

        while ((c = fgetc(stdin)) != EOF) {
                switch((char)c) {
                case ' \t':/* tab */
                        while (tabpos(col) != YES) {
                                fputc(' ', stdout);
                                col++;

                        }
                        break;
                case ' \n':/* newline */
                        fputc(' \n', stdout);
                        col = 1;
                        break;
                default:
                        fputc((char)c, stdout);
                        col++;
                }
        }

        exit(0);
}
tabpos(col)       /* return YES if col is a tab stop */
int col;
{
        if (col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}
settab(tabp)      /* set initial tab stops */
int *tabp;
{
        int i;
        for (i = 0; i <= MAXLINE; i++)
                tabs[i] = (i % TABSP) ? NO : YES;
        return;
}
```

Figure 74 shows the output generated by adb as the execution of the tabs program is analyzed. The first step is to set breakpoints at the start of the functions settab, fgetc, and tabpos. Display the breakpoints with the $b command. This shows the three breakpoints and a count of 1 for each. Set the first breakpoint at the entry point to the settab routine, and display a few instructions from the routine with the ?i command.

The next step is to run the program, directing input from a file called data. This file contains a few test lines of text, with tabs interspersed between the words. Execution stops at the breakpoint in settab. Delete the breakpoint at settab with the :d command. Note that you must specify the breakpoint to be deleted.

Continue execution to the next breakpoint with the :c command. Once the program has stopped, you can use adb requests to display the contents of memory. Use the $c command to display a stack trace, or examine the first few locations in the tabs array with the /o command. At this point in the program, settab has been called and should have set every eighth location of the tabs array to a 1.

Again, continue the program with the :c command. Execution stops at the breakpoint in the tabpos routine, because there is a tab at the beginning of the first line of data. The program breakpoints in tabpos several times until the program has changed the tab into equivalent blanks. Once you decide that tabpos is working, you can remove the breakpoint.

The next routine to be examined is fgetc. Set a breakpoint for fgetc with the :b command, but add two things: a count of three, so that adb executes the breakpoint twice and stops on the third occurrence; and a command to be executed each time the program hits the breakpoint. Note that this overwrites the previous breakpoint set at fgetc. The command to be executed at the breakpoint displays the value of the variable col, which is the current output column position.

Continue the program two more times, verifying that characters are being read and processed. Then delete the breakpoint at fgetc. Once this is done, continue execution of the program until completion. Output from the program is displayed at the end.

Figure 74 Sample adb session for tab conversion

```
% adb example2
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) settab:b
(adb) fgetc:b
(adb) tabpos:b
(adb) $b
job 0: breakpoint display

count  bkpt      command
  1    _tabpos
  1    _fgetc
  1    _settab
(adb) settab,4?i
_settab:
        sub.w  #4,a0
        sub.w  s0,s0
        st.w  s0,-4(fp)
        ld.w  -4(fp),s0
(adb) :r <data
job 0: running
job 0/0: breakpoint _settab:  sub.w  #4,a0
(adb) settab:d
(adb) :c
job 0: running
job 0/0: breakpoint _fgetc:   ld.w  @0(ap),s0
(adb) $c
_fgetc(80003124) from _main+0x46 [ap = ffffcd88]
_main(1,ffffcdac,ffffcdb4) from start+0x80 [ap = ffffcda0]
(adb) tabs,0t24?wq
_tabs:    1        0        0        0
          0        0        0        0
          1        0        0        0
          0        0        0        0
          1        0        0        0
          0        0        0        0
(adb) :c
job 0: running
job 0/0: breakpoint _fgetc:   ld.w  @0(ap),s0
(adb) tabpos:d
(adb) fgetc,3:b col/wo
(adb) $b
job 0: breakpoint display
count  bkpt      command
 3     _fgetc    col/wo
(adb) :c
job 0: running
_col:      14
_col:      15
_col:      16
job 0/0: breakpoint _fgetc:   ld.w  @0(ap),s0
```

Figure 72 (Continued) Sample adb sessions for tab conversion

```
(adb) :c
job 0: running
_col:       17
_col:       20
_col:       21
job 0/0: breakpoint _fgetc:   ld.w  @0(ap),s0
(adb) fgetc:d
(adb) :c
job 0: running
This is a test of the detab program. It converts tabs
to spaces.
job 0: terminated
all processes terminated
(adb)
```

Using adb—FORTRAN program example

B

Analyzing a core dump

One of the most common uses of adb is to analyze the contents of a core file produced when a program terminates abnormally. The core file contains an image of the program's state at the time of the termination (the contents of the program's data and stack segments).

A divide-by-zero trap is a common fault that results in a floating-point exception and a core dump. The program in Figure 75 illustrates this error.

Figure 75 FORTRAN program with bug

```
program runit
dimension a(10),b(10),c(10)
data a/0,1,2,3,4,5,6,7,8,9/, b/2,1,9,8,7,6,5,4,3,0/
do i = 1,10
      c(i) = (a(i) * b(i)) / (b(i) - a(i))
write(*,*) ' a(i) =',a(i),'b(i)=',b(i),' c(i)=',c(i)
enddo
stop
end
```

The adb command sequence and corresponding output in Figure 76 illustrate how to use adb to determine the source of the problem.

Figure 76 Sample adb session for FORTRAN program

```
% adb example3
Convex Debugger ($Date: 88/06/10 15:37:38 $)
Use ')help' for help.
(adb) :r
job 0: running
a(i) =      0.    b(i) =  2.000000    c(i) =      0.
job 0/0: stopped by a floating point exception (floating divide by zero)
job 0/0: stopped at _MAIN__+0x104:        pshea  _MAIN__+0x210
(adb) $r
job 0/0: register display

pc=8000127c (_MAIN__+0x104)
ps=03119080 (EF,SEQ,DZE,FDZ,FE,SQS,RES)
sp=ffffcd18 a1=8001a8c4 a2=80079135 a3=8004c046 tt=00000a22
a4=80024004 a5=00000004 ap=8000134c fp=ffffcd50
s0=4010000040800000 s1=0000000080000000 s2=3fd9999900000000 s3=0000000000000000
s4=4044000000000000 s5=3fbeb85100000030 s6=0000000000000000 s7=2061286980022610
(adb) 8000127a,5?ai
_MAIN__+0x102: _MAIN__+0x102: div.s  s2,s1
_MAIN__+0x104: pshea  _MAIN__+0x210
_MAIN__+0x10a: pshea  _MAIN__+0x214
_MAIN__+0x110: st.w  s1,-2147336192(a5)
_MAIN__+0x116: mov   a0,a6
(adb) _MAIN__+100:b
(adb) _MAIN__+102:b
(adb) :r
job 0: running
job 0/0: breakpoint _MAIN__+0x100:        psh.w  a4
(adb) $r
job 0/0: register display

pc=8000127a (_MAIN__+0x102)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd1c a1=8001a8c4 a2=80079135 a3=8004c046 tt=00000699
a4=80024000 a5=00000000 ap=8000134c fp=ffffcd50
s0=4010000000000000 s1=0000000000000000 s2=3fe9999941000000 s3=0000000000000000
s4=4044000000000000 s5=3fbeb85100000030 s6=0000000000000000 s7=00000000800790fc
(adb) :c
job 0: running
job 0/0: breakpoint _MAIN__+0x102:        div.s  s2,s1
(adb) $r
job 0/0: register display

pc=8000127c (_MAIN__+0x104)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd18 a1=8001a8c4 a2=80079135 a3=8004c046 tt=000006b0
a4=80024000 a5=00000000 ap=8000134c fp=ffffcd5
s0=4010000000000000 s1=0000000000000000 s2=3f9999941000000 s3=0000000000000000
s4=404400000000000 s5=3fbeb85100000030 s6=0000000000000000 s7=00000000800790
```

Figure 72 Sample adb sessions for FORTRAN program

```
(adb) :c
job 0: running
a(i) =      0.   b(i) =  2.000000   c(i) =      0.
job 0/0: breakpoint _MAIN__+0x100:        psh.w  a4
(adb) $r
job 0/0: register display

pc=8000127a (_MAIN__+0x102)
ps=03109080 (EF,SEQ,DZE,FE,SQS,RES)
sp=ffffcd1c a1=8001a8c4 a2=80079135 a3=8004c046 tt=00000a18
a4=80024004 a5=00000004 ap=8000134c fp=ffffcd50
s0=4010000040800000 s1=0000000040800000 s2=3fd9999900000000 s3=0000000000000000
s4=4044000000000000 s5=3fbeb85100000030 s6=0000000000000000 s7=2061286980022610
(adb) :c
job 0: running
job 0/0: breakpoint _MAIN__+0x102:        div.s  s2,s1
(adb)
```

Run the progm `a.out` with the `:r` command. The program stops at `MAIN__+104` when it encounters a divide-by-zero trap. This means that the previous instruction `MAIN__+102` caused the error.

Next, display value of the register state, including the pc where the divide-by-zero trap occurred. Examining the location of the pc and the next 5 lines (`8000127a,5?ai`) shows that `MAIN__+102` is a divide instruction to divide S1 by S2, and store the result in S1.

Set a breakpoint on the instruction before the error and and one after the error in order to examine the registers at each step. Do this with the :b command. Running the program again (`:r`) causes a stop at `MAIN__+100`, the instruction before the divide instruction. The registers show that S1 is zero and S2 is some finite value.

The `:c` command continues execution of the program until the next breakpoint at `MAIN__+54`. The registers show that S1 and S2 are both finite and that the divide will proceed without an error in this iteration.

The `:c` command continues execution until the divide. Here, the S2 register is zero. This is causing the divide by zero trap.

Because the sample program is simple, it is apparent where the divide occurs and which variables are in S1, S2, and S3. In more complex programs, you would probably need to do an in-depth analysis using the more advanced `adb` tools.

Index

! command 401
$ 16, 16, 191, 193, 193, 195
$< command 196, 198
$> command 16, 196, 197
$? command 16, 53, 199, 224
$a= command 16, 50, 201, 218, 219, 243, 246
$a? command 16, 202, 203, 205, 218, 219, 243, 246
$b command 16, 161, 207
$c command 16, 48, 209
$e command 16, 211, 222, 248
$f command 16, 213
$g command 16, 215
$h= command 16, 144, 202, 205, 219, 243, 246
$h? command 16, 144, 217, 218, 219, 243, 246
$i command 16, 211, 221, 248
$j command 16, 144, 199, 223, 240
$k command 16, 225
$l command 17, 227
$m command 17, 229
$n command 17, 144, 231
$o command 17, 233
$q command 235
$q commands 17
$R command 17, 239
$r command 17, 218, 219, 237, 240, 243, 246
$s= command 17, 49, 202, 205, 218, 219, 245
$s? command 17, 218, 219, 241, 243, 246
$t command 17, 247
$v=command 17, 202, 205, 218, 219, 243, 246, 253
$v? command 17, 218, 219, 243, 246, 249
$X command 17, 211, 257
$x command 17, 255
)comment command 385
)help command 144, 385, 387
)static command 393
)status command 144, 395
/ command 305
/- command 349
/+ command 347
/= command 307
/^ command 345
/"..." command 343
/a command 309
/b command 311
/C command 315
/c command 313
/F command 319
/f command 317
/g command 321
/h command 323
/i command 325
/m command 327
/n command 329
/P command 331
/r command 333
/S command 337
/s command 335
/t command 339
/w command 341
:b command 32, 159, 207
:C command 161, 167
:c command 161, 163
:D command 161, 173, 207
:d command 161, 171, 207
:e command 144, 175
:f command 144, 177, 199, 222, 240
:i command 179
:k command 181
:l command 227
:m command 183, 257
:r command 32, 161, 185
:S command 161, 189
:s command 51, 161, 187
=- command 383
=+ command 381
=^ command 379
=a command 351
=C command 355
=c command 353
=F command 359
=f command 357
=h command 361
=i command 363
=n command 365
=P command 367
=r command 369
=s command 371
=t command 373

=w command 375
=Y command 377
> command 399
? command 20, 22, 33, 259
?- command 303
?+ command 301
?= command 20, 22, 261
?^ command 299
?"..." command 297
?a command 263
?b command 265
?C command 269
?c command 267
?F command 273
?f command 271
?g command 20, 22, 52, 275
?h command 277
?i command 279
?m command 20, 22, 281
?n command 283
?P command 285
?r command 287
?S command 291
?s command 289
?t command 293
?w command 295

A

actual addresses 36
adb
  arithmetic operators 39
  basic debugging 55
  command overview 12
  command syntax 30, 33
  commands that change environment 119
  debugging concepts 3
  help 42
  invoking 29
  online help, screen example 43
  terminating 45
  terminology 3
adb address expressions 35
  (exp) 36
  + 35
  . (dot) 35
  <name 35
  ^ 35
  _ symbol 36
  _ symbol _ 36
  " 35
  integer 35
  symbol 35
adb command syntax
  , *count*, description 33
  address, description 33
  command, description 33
address
  expressions 35
  maps, displaying 229
  registers 4
    displaying 65
    displaying the contents of 16, 203
    displaying the contents of a 202, 205
    modifying value of 16, 201
  specifying
    actual 36
    symbolic 37
address registers
  modifying value of 50
arithmetic operators 39
assembly instructions, displaying 2
assigning
  dot to a variable or named register 399
  register formats 78
  value of internal variable 106
  value to
    address register 79
    hardware communication register 79, 116, 144, 219
    scalar register 79
    vector register 79
  values directly to memory 75
assigning values to hardware communication registers 16
assistance xix
associated documents xix
audience xv

B

backtraces, interpreting stack 89
binary, patching—instructions 52
breakpoint
  deleting 82, 171
    all currently set 173
  description 6
  displaying 64, 207
    currently set 16
  manipulating 80
  setting 80, 159

C

call stack, description 6
chained operating mode 233
changing
  current kernel memory mapping 16, 225
  exit disposition 144
  exit disposition of current job to RELEASE 175
colon (:) commands, *see* individual entries listed under commands

commands
! 401
$ 16, 16, 191, 193, 193, 195
$< 196, 198
$> 16, 196, 197
$? 16, 53, 199, 224
$a= 16, 50, 201, 218, 219, 243, 246
$a? 16, 202, 203, 205, 218, 219, 243, 246
$b 16, 161, 207
$c 16, 48, 209
$e 16, 211, 222, 248
$f 16, 213
$g 16, 215
$h= 16, 144, 202, 205, 219, 243, 246
$h? 16, 217, 218, 219, 243, 246
$i 16, 211, 221, 227, 248
$j 16, 144, 199, 223, 240
$k 16, 225
$l 17
$m 17, 229
$n 17, 144, 231
$o 17, 233
$q 17, 235
$R 17, 239
$r 17, 218, 219, 237, 240, 243, 246
$s= 17, 49, 202, 205, 218, 219, 245
$s? 17, 218, 219, 241, 243, 246
$t 17, 247
$v+ 243, 246
$v= 17, 202, 205, 218, 219, 253
$v? 17, 218, 219, 243, 246, 249
$X 17, 211, 257
$x 17, 255
)comment 385
)help 144, 385, 387
)static 393
)status 144, 395
/ 305
/- 349
/+ 347
/= 307
/^ 345
/"..." 343
/a 309
/b 311
/C 315
/c 313
/F 319
/f 317
/g 321
/h 323
/i 325
/m 327
/n 329
/P 331
/r 333
/S 337
/s 335
/t 339
/w 341
:b 32, 159, 207
:C 161, 167
:c 161, 163
:D 161, 173, 207
:d 161, 171, 207
:e 144, 175
:f 144, 177, 199, 222, 240
:i 179, 227
:k 181
:m 183, 257
:r 32, 161, 185
:S 161, 189
:s 51, 161, 187
=- 383
=+ 381
=^ 379
=a 351
=C 355
=c 353
=F 359
=f 357
=h 361
=i 363
=n 365
=P 367
=r 369
=s 371
=t 373
=w 375
=Y 377
> 399
? 20, 22, 33, 259
?- 303
?+ 301
?= 20, 22, 261
?^ 299
?"..." 297
?a 263
?b 265
?C 269
?c 267
?F 273
?f 271
?g 20, 22, 52, 275
?h 277
?i 279
?m 20, 22, 281
?n 283
?P 285
?r 287
?S 291
?s 289
?t 293
?w 295

adb 33
debugging multithreaded programs 115
executing from a file 93
ld 30
multiprocess debugging 143
RETURN 98, 397
that apply to
all threads in a job 117
current thread 116
that change
program data 118
the adb environment 119
that display
adb environment 119
program data 118
compilers—fc, vc, cc, ada 30
complex, description 108
ConvexOS
components of a process 138
debugging while running 32
program failure returns control to 2
core dump
$? command 199
$c command 97
determining the cause of 96
core file
description 31, 96
displaying information 53, 56, 305
examining, instructions 53
get core filenames 215
modification 31
writing information into 307
CPU (Central Processing Unit), description 108
current
breakpoints, deleting all 173
executable image 191
frame, description 6
job
changing exit disposition to RELEASE 175
continue execution for all threads in 167
displaying registers for all threads 239
executing one instruction for each thread in 189
killing 181
searching for symbol in 247
kernel memory mapping 225
position, description 6
process
description 7
killing 88
thread
changing 144
commands that apply to 116
description 9

D

debugging
multiprocess 137–151
commands 143–145
example 146–151
multithreaded 107–135
see also multithreaded debugging
deleting
breakpoints 82, 171, 207
all 173
currently set breakpoints 173, 207
disassembling an object file 51
displaying
adb environment 119
address
maps 17, 229
register contents 16, 202, 203, 205, 218, 219
registers 65
all jobs 144
all threads 144
assembly instructions 2
breakpoints 64
corefile data 305
current memory mapping 229
currently set breakpoints 16, 207
data from *objfile* 20, 22
external variable
names 16, 211, 222
values 16, 211, 222
floating-point format 16, 213
general registers 16, 218, 219, 224
contents 17, 237
global variables 71
hardware communication registers 16, 67, 144, 218, 219
contents 217
help command 387
information 56
with)status 395
instructions 57
internal variables 71
names and values 221
job status, all jobs 223
job-related information 60
maximum number of processors allocated for a process 231
name of nonzero internal variables 16, 221
objfile data 259
PIDs 16, 199, 224
processors, maximum number allocated for a process 144
program
data 118
variables 59
registers 65

for all thread in current job 239
for all threads in current job 17
scalar registers 69
contents 218, 219, 241
signals 16, 199, 224
stack backtracing 16, 48, 209
status
of adb modes 144
of all jobs 16, 223
values of
nonzero internal variables 221
vector registers 70
contents 17, 218, 219, 249
dollar ($) commands, *see* individual entries listed under commands
dot
assigning to variable or named register 399
description 6
dual-mode program 213
dyadic operators, table 39
DYNAMIC scheduling mode 183

E

executable
file
described 30
modification 31
image, current 191
executing
commands from a file 16, 93, 191–198
continue threads in current job 167
one instruction for each thread 189
program
continuing 163
instructions 84
shell command (!) 401
single instructions 86, 187
external variables, displaying names and values of all 211

F

file
executing commands from 16, 93, 191, 193, 195, 196, ??–198
getting core file names 215
getting new symbol file names 215
objfile 185
reading commands from 191
resume displaying output on screen 197
writing adb output to 193, 197
FIXED scheduling mode 183
flag, toggling the static symbol usage flag 393
floating-point format, displaying 213
`fork()` 137, 140
formats
displaying floating-point 213

G

getting
adb help 42
core file names 215
new symbol file names 215
global variables, displaying 71

H

hardware communication registers
assigning value to 144, 202, 205, 219
displaying 67
contents 144
hardware execution mode 233
help xix
help, getting adb help 42

I

information
displaying adb, overview 56
supplemental xix
inherit mode 179
input directory, description 31
instructions
displaying 57
executing single 86
internal variables 106
assigning values to 106
displaying nonzero 16
list of adb 221
interpretating, stack backtraces 89
invoking adb 30

J

job
bringing to foreground 177, 222
commands that apply to 118
continue execution for all threads in current 167
current, description 8
description 8
displaying
job-related information 60
status of all jobs 16, 199, 223
killing current 181

K

kernel
  debugging 103
  mapping 16, 32, 229
killing
  the current job 181
  the current process 88

L

ld command 30
logical names, *see* symbols

M

mapping, kernel 32
  changing current 225
  displaying current 229
memory
  changing kernel mapping 225
  mapping information 229
  modifying 75
  moving around in 73
mode
  chained 17, 233
  DYNAMIC 183
  FIXED 183
  inherit 179
  sequential 17, 233
modifying
  core file 31
  executable file 31
  input radix, default 255
  memory 75
  output radix, default 211, 257
  program data 75
  registers 49, 77
  scalar registers 17, 202, 205, 218, 219, 245
  segment map parameters in *objfile* 281
  value of address registers 16, 201, 218, 219
  vector register 17, 253
    contents 218, 219
monadic operators, table 39
multiple threads, description 109
multiprocess debugging 137–151
  commands 143
  example debugging session 148
multiprocess program
  description 138
  see also multithreaded programs
multiprocessor, description 108
multithreaded debugging 107–135
  example session 125–135
multithreaded programs 108–113
  commands 115–119
  debugging commands 115
  jobs 115
  multiple threads 109
  registers 112
  threads 115
mutual exclusion, description 108

N

nonzero internal variables, displaying names and values 221
notational conventions xvii

O

objfile
  description 30
  disassembling 51
  displaying information 56
  memory, writing information into 261
  modifying segment map parameters in 281
  searching for pattern in 275
operating modes, toggling between chained and sequential 233
operators
  dyadic (table) 39
  monadic (table) 39
ordering documents xix

P

patching a binary, instructions 52
PID
  displaying 16, 199, 224
process
  current, description 7
  description 108
  displaying maximum number of processors allocated 231
  id, displaying, *see* PID, displaying 199
  killing the current 88
  running *objfile* as 185
  starting a 84
processors, displaying maximum number allocated 231
program
  data
    commands that change and display 118
    modifying 75
  execution, continuing 87, 163
  instructions, executing 84
  variables, displaying 59
purpose of document xv

R

radix
  modifying default radix input 255
  modifying default radix output 257
reading commands from a file 191–195
reference pages 153–401
  list of 156–157
register set
  address 4
  CONVEX 3
  hardware communication 4
  scalar 4
  vector 4
registers
  address 201
  displaying 65
    hardware communication 67
    scalar 69
    vector 70
  general 199
  modifying 49, 77
  scalar 202, 205
  vector 202, 205
RELEASE, changing exit disposition to 144, 175
RETURN command 397
RETURN command 98

S

scalar registers 4
  displaying 17, 69
  modifying 17, 202, 205
    quick intructions 49
scheduling modes 183
segment mapping 104–105
sequential operating mode 231
setting
  breakpoints 159, 207
  floating-point format 16, 213
  limit for symbol matches 17, 227
shell command (!), executing 401
signal handling 99–102
signals
  displaying 199
  that stop or terminate a program 199
single stepping, *see* stepping 51
slash (/) commands, *see* individual entries listed under commands 153
specifying
  actual addresses 36
  symbolic addresses 37
stack backtraces
  displaying 209
    quick instructions 48
  interpreting 89
stepping, executing single instructions 51, 86, 127
  all threads in a job 134
subcomplex, description 108
symbol matches
  setting limits for 17, 227
symbolic addresses 37
symbols
  description 7
  searching the current job 247
  static symbol usage flag 393
system calls, `fork()` 137, 140

T

TAC xix
technical assistance xix
terminating adb 235
terminating adb 45
thread
  bringing to foreground 177
  continue execution for all in current job 167
  current 115
  description 8, 108
  displaying registers for all in current job 239
  executing one instruction for each in the current job 189
typographic conventions xvii

U

using this book xv

V

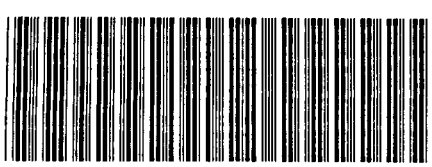
variables
  displaying
    all external 211
    global 71
    internal 71
  internal 106
    displaying name and values 221
vector
  displaying registers 17, 70
  registers
    modifying 17, 202, 205, 253
    modifying,
      instructions 49
virtual pointer, *see* dot

W

writing
adb output to a file 16, 193, 197
information into *corefile* memory 307
information into *objfile* memory 261

ORDER NUMBER
DSW-009

DOCUMENT NUMBER
710-025030-000



CONVEX PRESS