

HP 9000 Computer Systems
ADB Tutorial



HP Part No. 92432-90005
Printed in U.S.A. June 1991

First Edition
E0691

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

Edition	Date
First Edition	June 1991

Preface

This tutorial describes the use of ADB, a program that you can use to debug assembly language programs on Precision Architecture RISC (PA-RISC) machines. It also presents the ADB command format, and explains how to debug C programs, set breakpoints, and use maps. A complete command summary is provided following the tutorial.

This manual assumes that you, the reader, are experienced in assembly language programming. In addition, you should have a working knowledge of the HP-UX operating system. Consult the following manuals for additional details on related subjects:

- *HP-UX Reference: HP 9000 Computers, 3 volumes* (B1864-90000)
- *Assembly Language Reference Manual* (92432-90001)
- *Software Tools*
Kernigham and Plauger
Addison-Wesley Publishing Company
1976

Conventions

UPPERCASE

In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either upper or lowercase. For example:

COMMAND

can be entered as any of the following:

command Command COMMAND

It cannot, however, be entered as:

comm com_mand comamnd

italics

In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *FileName* with the name of the file:

COMMAND *FileName*

punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*FileName*):(*FileName*)

{ }

In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either ON or OFF:

COMMAND { ON }
 { OFF }

[]

In a syntax statement, brackets enclose optional elements. In the following example, OPTION can be omitted:

COMMAND *FileName* [OPTION]

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select OPTION or *Parameter* or neither. The elements cannot be repeated.

COMMAND *FileName* [OPTION
 Parameter]

Conventions (continued)

[...] In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *Parameter* zero or more times. Each instance of *Parameter* must be preceded by a comma:

[, *Parameter*] [...]

In the example below, you only use the comma as a delimiter if *Parameter* is repeated; no comma is used before the first occurrence of *Parameter*:

[*Parameter*] [, ...]


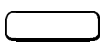

| ... | In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select A, AB, BA, or B. The elements cannot be repeated.

$\left\{ \begin{array}{l} A \\ B \end{array} \right\} | \dots |$

... In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

△ In a syntax statement, the space symbol △ shows a required blank. In the following example, *Parameter* and *Parameter* must be separated with a blank:

(*Parameter*) △ (*Parameter*)

 The symbol  indicates a key on the keyboard. For example,  represents the carriage return key.

base prefixes The prefixes %, #, and \$ specify the numerical base of the value that follows:

%num specifies an octal number.
#num specifies a decimal number.
\$num specifies a hexadecimal number.

If no base is specified, decimal is assumed.

Contents

1. ADB Tutorial	
Invoking ADB	1
Using ADB Interactively	3
Displaying Information	5
Debugging C Programs	8
Debugging a Core Image	8
Setting Breakpoints	11
Advanced Breakpoint Usage	15
Maps	18
Variables and Registers	19
Formatted Dumps	21
Patching	22
Debugging Already Running Processes	25
System Dependencies	27
Command Summary	28
Breakpoint and Program Control	28
Calling the Shell	28
Assignment to Variables	28
Formatted Printing Commands	29
Additional Printing Commands	29
Format Summary	30
Expression Summary	31
Monadic Operators	32

Index

Figures

1. C Program with a Pointer Bug	8
2. ADB Output from the Program in Figure 1	9
3. C Program to Decode Tabs	11
4. ADB Output from C Program Shown in Figure 3	13
5. ADB Output for Map Command	18
6. Simple C Program to Illustrate Patching	22
7. ADB Output Illustrating Patching	24

Tables

1. Expression-building Operators	3
2. Commonly Used ADB Commands	4
3. Commonly Used Format Commands	5
4. ADB Variables	19
5. Breakpoint and Program Control	28
6. Calling the Shell	28
7. Assignment to Variables	28
8. Formatted Printing Commands	29
9. Additional Printing Commands	29
10. Format Summary	30
11. Expression Components	31
12. Dyadic Operators	31
13. Monadic Operators	32

ADB Tutorial

ADB is a debugging program that operates on assembly language programs. It allows you to look at object files and “core” files that result from aborted programs, to print output files in a variety of formats, to patch files, and to run programs with embedded breakpoints. This tutorial provides examples of these and other ADB features.

Invoking ADB

You invoke ADB by executing the `adb(1)` command. The syntax is:

```
adb [-w] [-k] [-I dir] [-P pid] [objfile [corefile]]
```

where:

- `-w` Permits writing to the object file.
- `-k` Tells ADB that the object and core files are kernel files so ADB can perform the appropriate memory mapping.
- `-I dir` Specifies a directory (*dir*) that contains commands for ADB.
- `-P pid` “Adopt” an already running process for debugging.
- objfile* Names an executable object file.
- corefile* Names a core image file.

Normally, you invoke ADB by typing:

```
adb a.out core
```

or more simply:

```
adb
```

because the default setting for the object file is `a.out` and the core file is `core`.

Supplying a minus sign (-) for a file's name means "ignore this argument," as in:

```
adb a.out -
```

To write to the object file while ignoring the core file, you could type:

```
adb -w a.out -
```

To debug a currently running process, invoke ADB by typing:

```
adb -Ppid a.out
```

The `pid` or "process identifier" can be obtained using the `ps(1)` command.

Because ADB intercepts keystrokes, you cannot use a quit signal to exit from ADB. You must use the explicit ADB request `$q` or `$Q` (or `CONTROL D`) to exit from ADB.

For details on invoking the ADB command, see the `adb(1)` page in the *HP-UX Reference manual*.

Using ADB Interactively

You work interactively with ADB by entering requests.

The general form for a request is:

[*address*] [,*count*] [*command*] [*modifier*]

ADB maintains a current address, called “dot”. This address is similar in function to the current pointer in the HP-UX editor, *vi(1)*. When you supply an *address*, ADB sets dot to that location. ADB then executes any *command* you entered *count* times.

You can enter the *address* and *count* values as expressions. You create these expressions from symbols within the program you are testing and from decimal, octal, and hexadecimal integers. Table 1 lists the different operators for forming expressions.

Table 1. Expression-building Operators

Operator	Operation
+	Addition
-	Subtraction or Negation
*	Multiplication
%	Integer division
~	Unary NOT
&	Bitwise AND
	Bitwise Inclusive OR
#	Round up to next multiple

ADB performs arithmetic operations on all 32 bits.

ADB “remembers” the last radix set. You can change the current radix with the **\$o**, **\$d**, or **\$x** commands. During startup, the default radix is hexadecimal. If you change the radix to decimal, all subsequent input and output of integers are interpreted as decimal until another radix specifier is used.

Table 2 lists some commonly used ADB commands and their meaning.

Table 2. Commonly Used ADB Commands

Command	Description
?	Prints contents from <i>objfile</i> .
/	Prints contents from <i>corefile</i> .
=	Prints value of “dot” (.).
:	Breakpoint control.
\$	Miscellaneous requests.
;	Request separator.
!	Escapes to shell.
CONTROL C	Terminates any ADB command.

Displaying Information

You can request ADB to examine locations in either the object file or the core file. The `?` request examines the contents of the object file, while the `/` request examines the core file. Once you initiate a process (using either the `:r` or `:e` command), both `?` and `/` refer to locations in the address space of the running process.

Following either the `?` or `/` request, you can specify a format that ADB should use to print this information. Table 3 lists some commonly used format commands.

Table 3. Commonly Used Format Commands

Command	Description
<code>c</code>	One byte as a character.
<code>b</code>	One byte as a hexadecimal value.
<code>x</code>	Two bytes in hexadecimal.
<code>X</code>	Four bytes in hexadecimal.
<code>d</code>	Two bytes in decimal.
<code>f</code>	Four bytes in single floating point.
<code>F</code>	Eight bytes in double floating point.
<code>i</code>	HP Precision Architecture instruction.
<code>s</code>	Null-terminated character string.
<code>a</code>	Print in symbolic form.
<code>n</code>	Print a newline.
<code>r</code>	Print a blank space.
<code>^</code>	Backup dot.

For example, to print the first hexadecimal element of an array of long integers named `ints`, you would type the request:

```
ints/X
```

Note The array that is declared must be global. ADB does not recognize local variables.

This request sets the value of dot to the symbol table value of `ints`. It also sets the value of the dot increment to four. The “dot increment” is the number of bytes that ADB prints in the requested format.

In another example, to print the first four bytes as a hexadecimal number then the next four bytes as a decimal number, you would type the request:

```
ints/XD
```

In this case, ADB still sets dot to `ints` but the dot increment is now eight bytes.

The newline command is a special command that repeats the previous command. The newline command also uses the value of dot increment, but the command may not always have meaning. In this context, however, it means to repeat the previous **command** using a **count** of *one* and an **address** of *dot plus dot increment*. So, in this case, the newline command sets dot to `ints+0x8` and prints the two long integers: the first as a hexadecimal number and the second as a decimal number. You can also repeat the newline command as often as desired. For example, you could use this technique to scroll through sections of memory.

Using this example to illustrate another point, you can print the first four bytes in long hexadecimal format and the next four bytes in byte hexadecimal format, by typing the request:

```
ints/X4b
```

As this example shows, you can precede any format command with a decimal repeat character.

Furthermore, you can use the **count** parameter of an ADB request to repeat the entire format command a specific number of times. For example, to print three lines using the above format, you would type the request:

```
ints,3/X4bn
```

(The `n` at the end of the command prints a carriage return that makes the output easier to read.)

In this example, ADB sets the value of dot to `ints+0x10`, rather than `ints`. This happens because each time ADB re-executes the format command, it sets *dot* to *dot plus dot increment*. Therefore, the value of dot is the value that dot had at the beginning of the last execution of the format command. Dot increment is the size of the requested format (in this case, eight bytes). A newline command at this time would set dot to `ints+0x18` and print only one repetition of the format, because the **count** value is reset to one.

To verify the current value of dot, you can type the request:

```
.=a
```

The `=` command can print the value of an address in any format.

You can also use the = command to convert from one base to another. For example, you can print the value “0x32” in octal, hexadecimal, and decimal notation by typing:

```
0x32=oxd
```

ADB “remembers” complicated format requests for each of the ?, /, and = commands. For example, after entering the previous request, you can print the value “0x64” in octal, hexadecimal, and decimal notation by typing:

```
0x64=
```

Then, because the last entered / command was `ints/X4b`, you can type:

```
ints/
```

to print four bytes in long hexadecimal format and four bytes in byte hexadecimal format.

Although the two commands `main,10?i` and `main?10i` may appear to be identical, two important differences exist. The first is that the number “10” is represented in different bases. This happens because a repeat factor (`10i`) represents a decimal constant, while a `count` value (`,10`) can be an expression, and is therefore, by default, a hexadecimal number.

The second difference is that entering a newline command after the first request would print one line, while a newline command after the second request would print another ten lines.

Debugging C Programs

The following examples illustrate various features of ADB. Certain parts of the output such as machine addresses may depend on the hardware being used, as well as how the program was linked whether *shared*, *unshared*, or *demand loaded*.

Debugging a Core Image

The C program listed in Figure 1 shows some of the useful information that you can obtain from a core file. This program attempts to calculate the square of the variable `ival` by calling the function `sqr` with the address of that integer. An error occurs, however, because the program passes the integer's value rather than its address. Therefore, executing the program produces a bus error that generates a core file.

```
int ints[]=    {1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0,
                1,2,3,4,5,6,7,8,9,0};

int ival;
main()
{
    register int i;
    for(i=0;i<10;i++)
    {
        ival = ints[i];
        sqr(ival);
        printf("sqr of %d is %d\n",ints[i],ival);
    }
}

sqr(x)
int *x;
{
    *x *= *x;
}
```

Figure 1. C Program with a Pointer Bug

To isolate the problem assuming the object file is `a.out`, you can invoke ADB by entering the command:

```
adb
```

You can then request a C backtrace of the subroutines that this program calls by typing:

```
$c
```

This request allows you to check the validity of the parameters that the program passes. The stack trace shows that the segmentation violation occurred within the procedure `sqr()`. (See Figure 2.)

```

$c
sqr() from main+30
main() from _start+18
_start() from $START$+30
$r
pcoqh 0xD2B      sqr+7
pcoqt 0xD2F      sqr+0xB
rp     0xCEB      main+33

arg0  1          arg1  68023130      arg2  68023138      arg3  0
sp    68023250    ret0  0          ret1  4E          dp
r1    40001800    r3    0          r4    40010954    r5
r6    68023644    r7    499E       r8    0           r9    3
r10   0xFFFF88000  r11   1880      r12   4000       r13   0xA
r14   0xFFFFB400  r15   2A000     r16   29C00      r17   25000
r18   1BB58       r19   1         r20   1513EF8    r21
r22   0           r31   1         sar   12          sr0   122
sr1   0           sr2   0         sr3   0          sr4   122
sqr+4?
sqr+4:          0xE601095      = ldws  0(r19),r21
<r19=X
1

sqr,3?ia
sqr:
sqr:          or      arg0,r0,r19
sqr+4:        ldws   0(r19),r21
sqr+8:        ldws   0(r19),arg1
sqr+0c:

```

Figure 2. ADB Output from the Program in Figure 1

In general, ADB does not know the location of arguments passed to subroutines on HP-UX systems. The first four arguments are usually passed in registers, but compilers may transfer the argument contents to another register or copy these value to the stack. While these software conventions assist in program execution performance, they make assembly-level debugging more adventuresome.

The system maintains registers that point to the head of the program counter queue (`pcoqh`) and to the tail of this queue (`pcoqt`). To print these register values and an interpretation of the instructions at those locations, you can type the request:

```
$r
```

Because the `rp` register often points to a subroutine's return address, its value is also referenced and symbolically interpreted. The two lower bits in these registers contain "instruction privilege level" information that ADB usually ignores. Other registers include `arg0`, `arg1`, `arg2`, and `arg3`, which are often used to pass arguments to subroutines; `dp` (data pointer), which points to the beginning of text; `sp`, the stack pointer; and `ret0` and `ret1`, which hold function return values. Note that all values are given as hexadecimal numbers (the default base for integer values).

The `pcoqh` register indicates that the program failed at `sqr+4` (remember to ignore the lower two bits). To print the actual instruction that failed, you can list the instruction and its offset by typing the request:

```
sqr+4?i
```

or:

```
<pcoqh?i
```

This request shows that the instruction that failed was:

```
sqr+4:      ldws    0(r19),r21
```

This instruction uses general register 19 (`r19`) as a pointer, and loads the contents of the memory location to which it points (offset by 0) into general register 21 (`r21`).

But what was the value of `r19` when the program crashed? To print the value of `r19` as a 4-byte hexadecimal value, you can type the request:

```
<r19=X
```

You find that its value is one. Therefore, the segmentation violation occurs because memory address 1 is not part of the data space.

Refer to Table 8, “Formatted Printing Commands”, for more information.

How did `r19` get this value? You can print three instructions, beginning at `sqr`, by typing the request:

```
sqr,3?ia
```

This shows that the first instruction copied the first argument (`arg0`) into `r19`. This means that the value of the first argument was one; in other words, the program is passing the value—rather than the address—of the integer `ival` in `main()`.

You can print the values of all external variables at the time a program crashes, by typing:

```
$e
```

Setting Breakpoints

The C program shown in Figure 3, which changes tabs into blanks, is adapted from a program in the book *Software Tools*.

```
#include <stdio.h>
#define MAXLINE 80
#define YES 1
#define NO 0
#define TABSP 8

FILE *stream;
int tabs[MAXLINE];
char ibuf[BUFSIZ];

main(argc, argv)
int argc;
char **argv;
{
    int col, *ptab;
    char c;

    setbuf(stdout, ibuf);
    ptab = tabs;
    settab(ptab); /* Set initial tab stops */
    col = 1;
    stream = fopen(argv[1], "r");
    while((c = getc(stream)) != EOF) {
        switch(c) {
            case '\t': /* TAB */
                while(tabpos(col) != YES) {
                    putchar(' '); /* put BLANK
                    col++ ;
                }
                break;
            case '\n': /*NEWLINE */
                putchar('\n');
                col = 1;
                break;
            default:
                putchar(c);
                col++ ;
        }
    }
}
```

Figure 3. C Program to Decode Tabs

```

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
    if(col > MAXLINE)
        return(YES);
    else
        return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
    int i;

    for(i = 0; i<= MAXLINE; i++)
        (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}

```

Figure 3. C Program to Decode Tabs (continued)

After compiling the program into an object file called `expand`, trying to run the program produces a segmentation violation. So, to run the program under ADB control, you can enter the command:

```
adb expand
```

In this case, asking for a stack trace yields little information, so you set breakpoints in the two subroutines and the library routines `setbuf` and `fopen` by typing:

```

setbuf:b
settab:b
fopen:b
tabpos:b

```

In general, you can set breakpoints in a program by using requests of the form:

```
address[,count] :b [request]
```

where:

- count* Is an optional modifier which specifies the number of times that ADB should skip this breakpoint before stopping.
- request* Is an optional command that ADB executes when it encounters this breakpoint.

Figure 4 lists an interactive session with ADB for the program listed in Figure 3.

```

adb expand
$c
main() from _start+18
_start() from $START$+30
setbuf:b
settab:b
fopen:b
tabpos:b
$b
breakpoints
count  bkpt          command
1      tabpos
1      fopen
1      settab
1      setbuf
:r
expand: running (process 18958)
breakpoint  setbuf:      stw      rp,-14(sp)
:c
expand: running
breakpoint  settab:      ldo      38(sp),sp
:c
expand: running
breakpoint  fopen:       stw      rp,-14(sp)
:c
expand: running
segmentation violation
stopped at  main+64:     stws     r21,0(r19)
setbuf:d
settab:d
:r
expand: running (process 18965)
breakpoint  fopen:       stw      rp,-14(sp)
<rp:b <ret0=X
$b
breakpoints
count  bkpt          command
1      tabpos
1      fopen
1      main+4C      <ret0=X
:c
expand: running
0
breakpoint  main+4C:     addil   1000,dp

```

Figure 4. ADB Output from C Program Shown in Figure 3

```

fopen:b <arg0/s; <arg1/s
$b
breakpoints
count  bkpt          command
1      main+4C      <ret0=X
1      tabpos
1      fopen        <arg0/s; <arg1/s
:r
expand: running (process 18966)
0:
40000000:      r
breakpoint    fopen:          stw      rp,-14(sp)
:r expand.c
expand: running (process 18968)
68023007:      expand.c
40000000:      r
breakpoint    fopen:          stw      rp,-14(sp)
:c
expand: running
40000040
breakpoint    main+4C:        addil   1000,dp
:c
expand: running
#include <stdio.h>
#define MAXLINE 80
breakpoint    tabpos:         ldo     50(r0),r19
:d*
:c
...

tabs/80X
tabs:
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
               ...

```

Figure 4. ADB Output from C Program Shown in Figure 3 (continued)

You can print the location of each breakpoint by typing:

```
$b
```

Notice that the display lists a `count` field. ADB bypasses a breakpoint “count - 1” times before it stops execution. A `command` field indicates which requests ADB should execute each time it encounters that breakpoint.

To run the program, type:

```
:r
```

ADB informs you that it has encountered a breakpoint at `setbuf`, and it prints the instruction at that address.

To continue executing the program from that breakpoint, type:

```
:c
```

After breaking and continuing two more times, the program encounters the segmentation violation.

Advanced Breakpoint Usage

At this point, you should ensure that the call to `fopen` succeeded. First, delete the breakpoints at `setbuf` and `settab` by typing:

```
setbuf:d  
settab:d
```

Now you can run the program again. When ADB executes the breakpoint at `fopen`, you set a breakpoint at the return from `fopen` by typing:

```
<rp:b <ret0=X
```

This sets a breakpoint at the address to which `rp` points. Remember that, by convention, this register is a return pointer: it points to the address to which the program returns after execution of the procedure. Additionally, you tell ADB to print the value of `ret0` when it encounters the breakpoint. This register contains the 32-bit return value from `fopen`. Note that 64-bit values are returned in `ret0` and `ret1`, combined, while larger values are returned in the address to which `ret0` points.

To verify that the previous breakpoint commands have been registered, you can list all the breakpoints by typing:

```
$b
```

This displays a breakpoint at `main+4C` as well as the command that you want ADB to execute when it encounters this breakpoint. Then, when you give the command to continue, ADB encounters the breakpoint at `main+4C`. Before issuing the breakpoint message, however, ADB executes the command associated with that breakpoint.

In this case, the return value is zero, which indicates that the `fopen` call failed. The *HP-UX Reference* manual lists several possible causes for this failure; one of which is incorrect arguments. Although at this point it is too late to find the file name and type arguments to `fopen` as both are passed as pointers to character strings, you can examine them at the procedure entry point.

You can run the program once again, wait for ADB to encounter the breakpoint at `fopen`, and then print the argument registers. For illustrative purposes let's print the arguments with breakpoint commands by typing:

```
fopen:b <arg0/s; <arg1/s
```

Note that this request overrides the previous breakpoint at this address. The semicolon is necessary to separate the two commands.

When you run the program again, ADB suspends the program at the `fopen` breakpoint and prints:

```
0:
40000000:      r
breakpoint    fopen:      stw    rp,-14(sp)
```

The displayed values refer to the contents of `arg0` and `arg1` and the strings to which they point.

At this point, you may realize that the first argument is a null pointer; no arguments are being passed to the program! Because the program performs no error checking, there is no way to determine if the call to `fopen` returned a `FILE` pointer. A better program design, therefore, would be to test whether an argument was passed to the program; and, if not, use standard input as the stream.

Arguments and redirection of standard input and output are passed to a program as follows:

```
:r arg1 arg2 ... <infile >outfile
```

Note ADB does not perform “wild-card” expansion on its arguments.

If you now run the program with a file name as an argument, as in:

```
:r expand.c
```

you find that a pointer to the string “`expand.c`” is being passed as the first argument. Upon continuing execution, `fopen` correctly returns a non-null value. Now when you continue again, the program successfully reaches the next breakpoint at `tabpos`.

A number of breakpoints at `tabpos` occurs before the program terminates normally. With confidence that the program is working correctly, you can remove all breakpoints with:

```
:d*
```

and continue with:

```
:c
```

Unfortunately, however, you soon realize that the program does not work; multiple tabs seem to have the same effect as one tab. At this point, it would make sense to check whether the `tabs` array is initialized correctly, and if the columns marker, `col`, is being set correctly. You can set a breakpoint at any point past `settab` (`fopen` suffices) and examine the `tabs` array by typing the request:

```
tabs/80X
```

The array looks correct, so examine the value of `col` and the operation of the subroutine `tabpos`.

To print the value of `col` (`arg0`) at every call to `tabpos`, you can type the request:

```
tabpos,-1:b <arg0=D
```

The *count* argument of “-1” is an artifice; the breakpoint is not really executed “*count*-1” times before stopping as the manual page states. Rather, it is executed until *count* is decremented to 0. Upon continuation, all that prints to the screen is the output from the program `expand`, and the value in decimal of `col` at the time of the call. The bug should become apparent at this point.

You can regain ADB’s attention prematurely with an interrupt signal. Any HP-UX signals that act on ADB itself such as quit, interrupt and stop signals are also received by the program being debugged. The process enters a stopped state before it actually receives the signal, and ADB is notified. See the *ptrace(2)* and *wait(2)* manual pages for more information. The signal is then passed to the process being debugged when you type the request:

```
:c
```

You can override this result by passing another signal number as an argument. In particular, you can pass no signal to a process by typing the request:

```
:c 0
```

Maps

HP-UX supports several executable file formats such as *shared*, *unshared*, and *demand-loaded* that tell the loader how to load a program file. Currently, only shared text files are supported on PA-RISC systems. In shared files, instructions are separated from data, and the text space or instructions are shared when several users are running the process concurrently. Note that once a breakpoint is executed, a private copy of the program's text is used by ADB.

ADB uses knowledge of file formats to translate addresses both symbolic and numeric to locations in the executable and core files. A map command is available that prints out the file format mapping:

```
$m
```

ADB uses the **b**, **e**, and **f** fields (known as a triple) to map addresses into file addresses. The **f1** field in the executable map (the “?” map) is the length of the header at the beginning of the file. The **f2** field is the displacement from the beginning of the file to the data. The **b** field is the beginning of the virtual address of a memory segment and the **e** field is the end of the virtual address of a memory segment. The **?*** request tells ADB to use only the second part of the map in the **a.out** file when translating addresses. The user-modifiable map for the core file also has two triples. The initial map for the core file has as many triples as there are core segments in the core file (see *core(4)*). Figure 5 shows ADB output for the map command.

```
? map      'c0358'
b1 = 1000          e1 = 3114          f1 = 4000
b2 = 40000000     e2 = 400003E0        f2 = 7000
/ map      'c0358.core'
Kernel: b = 68FA89C0   e = 68FA89FC   f = 10
Exec:   b = 68FA897C   e = 68FA89C0   f = 5C
Core:   b = 68FA896C   e = 68FA8970   f = 0xB0
Data:   b = 40000000   e = 40003000   f = 0xC4
Registers:      b = 68FA8AB8   e = 68FA8C40   f = 30D4
Stack:  b = 68FAC000   e = 68FBE000   f = 326C
/ map (inactive)      'c0358.core' from 'c0358'
b1 = 0              e1 = -1         f1 = 0
b2 = 0              e2 = -1         f2 = 0
```

Figure 5. ADB Output for Map Command

Variables and Registers

ADB provides a set of variables for programmers to use. Each variable name consists of a single letter or digit. For example, to set the variable “5” to the hexadecimal value 32, you use the “greater than” sign (>) as follows:

```
0x32>5
```

You can then use this variable in other requests. For example, to print the value of the variable “5” in hexadecimal format, you use the “less than” sign (<) as follows:

```
<5=X
```

ADB sets the value of other variables. These variables are listed in Table 4.

Table 4. ADB Variables

Variable	Description
0	Last value printed.
9	Count for a \$< command.
b	Base address of data segment.
d	Data segment length.
e	Entry point.
m	Execution type: 0x107 (non-shared) 0x108 (shared) 0x10b (demand loaded)
s	Stack length.
t	Text length.

These variables are helpful when you want to know whether the file under examination is an executable file or core image file. ADB reads the header of a core image file to find the values for these variables. If the second file specified with the `adb` command does not appear to be a core file or if the `adb` command omits this file, ADB uses the header of the executable file instead.

You can use variables for such purposes as counting the number of times a routine is called. For example, to count the number of times that the routine `tabpos` is called in the program listed in Figure 3, you would type the requests:

```
0>5
tabpos,-1:b <5+1 >5
:r
<5=U
```

The “0>5” command sets the variable 5 to zero.

The “`tabpos,-1:b <5+1 >5`” command sets a breakpoint at `tabpos`. Because the `count` field is -1, the process never stops at this breakpoint, but ADB executes the breakpoint requests every time it reaches this breakpoint. Finally, this command increments the value of the variable 5 by 1.

The “`:r`” command causes the process to run to termination, and the “`<5=U`” command prints the value of the variable as an unsigned decimal value.

You can print the values of all nonzero variables by typing:

```
$v
```

Note ADB uses the `a` register to determine how many arguments to print with a stack trace. For more information see the section on “Anomalies” later in this tutorial.

You can also set the values of individual registers in the same way you set variables. For example, to set the value of the register `r1` to hexadecimal 32, you would type:

```
0x32>r1
```

Or, to print the value of the register `r1` in hexadecimal format, you would type:

```
<r1=X
```

You can print the value for every register by typing the request:

```
$R
```

And, you can print the value of the registers of general interest by typing:

```
$r
```

Formatted Dumps

You can combine ADB formatting commands to provide elaborate displays. The following examples illustrate this.

To print four octal half-words followed by their ASCII interpretation from the data space of the core image file, you would type:

```
<b,-1/4o4^8Cn
```

The first part of this request, broken down, has the following meanings:

- <b Gives the base address of the data segment.
- ,-1/ Prints from the base address to the end of file. The negative count field lets ADB loop until it detects an error condition or the end of the file.

The format request modifier (4o4^8Cn) has the following meaning:

- 4o Prints four octal half-words locations.
- 4^ Backs up the current address four locations to the original start of the field.
- 8C Prints eight consecutive characters using an escape convention that prints each character in the range 0 to 037 as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
- n Prints a newline.

To stop the printing at the end of the data segment, where <d provides the data segment size in bytes, you would modify the previous request as follows:

```
<b,<d%8/4o4^8Cn
```

Formatting requests can also be read from script files. The script files can be specified as the standard input for ADB:

```
adb a.out < script_file
```

Alternately, a script file can be invoked within a debugging session with the ADB command:

```
$<script_file
```

Patching

You can patch files with ADB by using the “write” request (`w` or `W`). You often use this request in conjunction with the “locate” request (`l` or `L`). The syntax for both requests is:

```
[?/] [lL] value
[?/] [wW] value
```

The `l` request matches on two bytes, and the `w` request writes two bytes; whereas the `L` request matches on four bytes, and the `W` request writes four bytes. The `value` field for both requests is an expression, so decimal and octal numbers, as well as character strings, are supported.

To modify a file, you must invoke ADB with the `-w` flag; for example:

```
adb -w objectfile corefile
```

When you invoke ADB with this option, ADB creates the *objectfile*, if necessary, and opens that file for both reading and writing. ADB only opens *corefile* for reading, however.

Note Once a subprocess has been initiated with a `:r` or `:e` command, write requests alter the subprocess’ address space, not the *objectfile*.

For example, consider the C program shown in Figure 6. The `write` command takes three arguments: a file descriptor, a character buffer, and a count of the number of bytes to write. As currently written, the count value for the number of bytes to write was calculated incorrectly.

```
main()
{
    write(1, "Hello world\n", 11);
}
```

Figure 6. Simple C Program to Illustrate Patching

You could set a breakpoint at the call to the `write` procedure and set the argument to the correct value by typing the command:

```
0d12>arg2
```

However, you would have to do this every time you wanted to run the program.

Assuming that you had lost the source file for this “valuable” piece of code, you could patch the object code using ADB.

You call ADB with the command:

```
adb -w hello -
```

Then you can find which instruction to modify by printing the first eight instructions of `main`.

```
main,8?i
```

You find the required instruction is at `main+18` (hexadecimal).

```
main+18:      ldo      0xB(r0),arg2
```

This instruction loads the contents of `r0` (which is always zero), plus the immediate value `0xB` (decimal 11) into `arg2`, the third argument to the write statement.

You can change the instruction with:

```
main+18?W 34180018
```

Broken down, this request has the following meanings:

<code>main+18</code>	Sets the value of dot.
<code>?W</code>	Writes four bytes in <i>objectfile</i> .
<code>34180018</code>	The hexadecimal value to write.

Note that ADB prints the old and new value when you request a write. When you reprint the instruction, you see that you patched it correctly. This sort of patching requires a knowledge of the machine-level format, or a willingness to experiment. Remember that if you had started the process with `:r` or `:e` before you issued the write command, the patch would have been made in the process' address space, not in the object file itself. Refer to Figure 7.

```
main,8?ia
main:
main:          stw    rp,-14(sp)
main+4:        ldo    30(sp),sp
main+8:        ldo    1(r0),arg0
main+0xC:      addil  0,dp
main+10:       ldo    0(r1),arg1
main+14:       bl     write,rp
main+18:       ldo    0xB(r0),arg2
main+1C:       ldw    -44(sp),rp
main+20:
main+18?W 34180018
main+18:       34180016      =      34180018
main+18?i
main+18:       ldo    0xC(r0),arg2
:r
hello: running (process 1576)
Hello world
process terminated
$q
```

Figure 7. ADB Output Illustrating Patching

Debugging Already Running Processes

The `-P` option allows ADB to “adopt” an errant process as if it had been originally run under the control of the debugger. The user can then examine it, and detach from it when debugging is completed. After ADB detaches from the process, the program resumes execution, no longer under the control of ADB.

Note that the effective user ID of the tracing process must match the effective user ID of the traced process; however, this is not necessary if the effective user ID of the tracing process is the superuser.

Consider the C program below. After the write statement, the program goes into an infinite loop.

```
main()
{
    write(1, "Hello world\n", 12);
    for(;;)
        ;
}
```

If this program runs in the background like this:

```
a.out &
4326
Hello world
```

You can debug pid 4326 by typing:

```
adb -P4326 a.out
```

Single stepping through the program reveals that this program seems to be looping infinitely:

```
:s
a.out: running
stopped at      main+1C:      b,n      main+1C
:s
a.out: running
stopped at      write:      ldil     -40000000,r1  (nullify)
:s
a.out: running
stopped at      main+1C:      b,n      main+1C
:s
a.out: running
stopped at      write:      ldil     -40000000,r1  (nullify)
:s
a.out: running
stopped at      main+1C:      b,n      main+1C
$q
```

Running `ps(1)` after exiting ADB shows that the process continues executing after ADB detaches from it.

```
ps
  PID TTY          TIME COMMAND
 5428 ttyq6        0:01 ps
 4326 ttyq6       11:14 a.out
 4126 ttyq6        0:03 csh
```

Note It is not possible to use the `-w` and `-P` options together. It is an error to open a file for writing when it is already open for execution.

System Dependencies

Below is a list of some system dependencies of which you should be aware.

- To increase run-time execution speed, stack-frame context is kept to a minimum. In particular, the previous stack pointer and the return pointer are not necessarily written to memory locations on the stack itself. Additional information necessary to perform stack unwinds resides in the object file. Because of this, if the object and core files being debugged are not from the same program, the stack unwind for the core file fails.
- Arguments to procedures are not all passed on the user stack. By convention, the first four arguments are passed in registers. Usually, general registers 23-26 known mnemonically as `arg3`, `arg2`, `arg1`, and `arg0` are used, but the floating point registers 4-7 (`fr4`, `fr5`, `fr6`, and `fr7`) may also be used. Arguments five and beyond are passed on the stack, and space is left on the stack to store the arguments passed in registers, if the compiler (or assembly language coder) sees fit to do so; but the compiler might decide to save the argument in another register rather than on the stack.

At procedure entry time, ADB has the information available to discern which argument registers have valid data. Beyond this point, however, ADB has no way of determining where the compiler has decided to store the arguments unless the procedure was compiled with the `-g` option to produce symbolic debug information. ADB also has no way of determining the number of arguments passed to a procedure. By default, it prints four arguments during a stack trace if it knows the location of the arguments.

You can force ADB to print more than four arguments by changing the value of the “`a`” variable. See the section on “Variables and Registers” for more details. ADB then prints the contents of the stack locations where these arguments are stored, although this data is “garbage” when no argument is passed which corresponds to that location.

- At entry to a procedure, the user stack is in a known state; that is, the location of arguments and return pointer is known in relation to the current stack pointer value. The first few instructions after procedure entry are a dialogue to save the return pointer on the stack, increment the stack pointer, save the old stack pointer on the stack, and save registers. All of these steps are optional, at the discretion of the compiler or assembly language coder. If you set a breakpoint beyond the entry to the procedure, but before the stack has been incremented if that is to be done, stack traces give incorrect information. Once again, ADB uses Additional information in the instruction space to determine where the return pointer and previous stack pointer were stored.

Command Summary

Breakpoint and Program Control

Table 5. Breakpoint and Program Control

Command	Description
:b	Set breakpoint at dot.
:c	Continue running program.
:d	Delete breakpoint.
:k	Kill the program being debugged.
:r	Run object file under ADB control.
:s	Single step through program.

Calling the Shell

Table 6. Calling the Shell

Command	Description
!	Call shell to read remainder of line.

Assignment to Variables

Table 7. Assignment to Variables

Command	Description
> <i>name</i>	Assign dot to variable or register name.

Formatted Printing Commands

Table 8. Formatted Printing Commands

Command	Description
? <i>format</i>	Print from object file according to format.
/ <i>format</i>	Print from core file according to format.
= <i>format</i>	Print the value of dot.
?w <i>expression</i>	Write <i>expression</i> to object file.
/w <i>expression</i>	Write <i>expression</i> to core file.
?1 <i>expression</i>	Locate <i>expression</i> in object file.
/1 <i>expression</i>	Locate <i>expression</i> in core file.

Additional Printing Commands

Table 9. Additional Printing Commands

Command	Description
\$b	Print current breakpoints.
\$c	Print stack trace.
\$d	Set default radix to address argument.
\$e	Print external variables.
\$f	Floating-point registers as single precision.
\$F	Floating-point registers as double precision.
\$m	Print ADB segment maps.
\$r	Print general registers.
\$s	Set offset for symbol match.
\$v	Print ADB variables.
\$w	Set output line width.

Format Summary

Table 10. Format Summary

Command	Description
a	Value of dot in symbolic form.
b	One byte in hexadecimal.
B	One byte in octal.
c	One byte as a character.
d	Two bytes in decimal.
D	Four bytes in decimal.
f	Four bytes in single precision floating point.
F	Eight bytes in double precision floating point.
i	HP Precision Architecture instruction.
o	Two bytes in octal.
O	Four bytes in octal.
n	Print a newline.
r	Print a blank space.
s	Character string terminated by null.
nt	Move to next <i>n</i> space tab.
u	Two bytes as unsigned integer.
U	Four bytes as unsigned integer.
x	Hexadecimal number.
X	Four bytes as a hexadecimal number.
Y	Date.
~	Backup dot.
"..."	Print string.

Expression Summary

An expression consists of an operator and an operand (or operands). An operand can consist of the following components.

Expression Components

Table 11. Expression Components

Component	Examples
Decimal integer	0d256, 0t256
Octal integer	0277, 0o277
Hexadecimal integer	0xff, 0xC0
Symbols	flag, main
Variables	<b
Registers	<arg0, <rp
(expression)	Expression grouping

Dyadic Operators

Table 12. Dyadic Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise AND
	Bitwise OR
#	Round up to next multiple

Monadic Operators

Table 13. Monadic Operators

Operator	Operation
~	NOT
*	Contents of location
-	Negate integer value

Index

A

ADB

- command format, 1-3
 - commands, 1-4
 - command summary, 1-28
 - debugger, 1-1
 - features, 1-1
 - interactive requests, 1-3
 - invoking, 1-1
 - location of arguments, 1-9
 - syntax, 1-1
 - variables, 1-19
- additional printing commands, 1-29
- address, 1-3
- numeric, 1-18
 - symbolic, 1-18
- advanced breakpoint usage, 1-15
- ASCII interpretation, 1-21
- assignment to variables, 1-28

B

- breakpoints, 1-11
- program control, 1-28

C

- calling the shell, 1-28
- commands
- additional printing, 1-29
 - assignment to variables, 1-28
 - breakpoints and program control, 1-28
 - calling the shell, 1-28
 - format, 1-5
 - format summary, 1-30
 - formatted printing, 1-29
- command summary, 1-28
- common format commands, 1-5
- components
- expression, 1-31
- core file, 1-2
- count, 1-3
- C program
- debugging, 1-8
 - patching example, 1-22
 - to decode tabs, 1-12
 - with a pointer bug, 1-8

- current address
- dot, 1-3

D

- debugging
- core image, 1-8
 - C programs, 1-8
 - running processes, 1-25
- displaying information, 1-5
- dot
- value of, 1-5
- dyadic operators, 1-31

E

- executable file formats
- shared, unshared, demand loaded, 1-8, 1-18
- expression
- components, 1-31
 - operand, 1-31
 - operator, 1-31
- expression-building operators, 1-3
- expression summary
- dyadic operators, 1-31
 - expression components, 1-31
 - monadic operators, 1-32

F

- format
- commands, 1-5
 - summary, 1-30
- formatted
- dumps, 1-21
 - printing commands, 1-29

I

- interrupt signal, 1-17

L

- loader, 1-18
- locate request, 1-22

M

- map addresses, 1-18
- file addresses, 1-18
- map command output, 1-18

maps, 1-18
monadic operators, 1-32

N

newline command, 1-6

O

object file, 1-2
operators
 dyadic, 1-31
 monadic, 1-32

P

patching, 1-22
 corefile, 1-22
 C program, 1-22
 objectfile, 1-22
printing commands
 additional, 1-29
 formatted, 1-29
procedure entry time, 1-27
processes running
 debugging, 1-25
process identifier, 1-2

R

registers
 setting the value, 1-20
requests, 1-3
 locate and write, 1-22
running processes
 debugging, 1-25

S

setting breakpoints, 1-11
 request, 1-12
shell
 calling, 1-28
stack-frame context, 1-27
stack unwinds, 1-27
summary
 ADB commands, 1-28
 additional printing commands, 1-29
 assignment to variables, 1-28
 breakpoints and program control, 1-28
 calling the shell, 1-28
 dyadic operators, 1-31
 expression components, 1-31
 format, 1-30
 formatted printing commands, 1-29
 monadic operators, 1-32
system dependencies, 1-27
 arguments to procedures, 1-27

U

user stack, 1-27
 known state, 1-27

V

variable name, 1-19
variables and registers, 1-19

W

write request, 1-22