A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

ABSTRACT

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine ‘‘core’’ files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine "core" and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at ‘‘core’’ files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The filename minus (−) means ignore this argument as in:

adb − core

ADB has requests for examining locations in either file. The **?** request examines the contents of *objfile*, the **/** request examines the *corefile*. The general form of these requests is:

address ? format

† UNIX is a trademark of Bell Laboratories.

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

sets dot to octal 126 and prints the instruction at that address. The request:

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the **?** or **/** requests, the current address can be advanced by typing newline; it can be decremented by typing ˆ.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, –, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ˜ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name;* ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP 11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
ˆ	**backup dot**

(Format letters are also available for "long" values, for example, ‘**D**’ for long decimal, and ‘**F**’ for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets ‘dot’ to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command	**Meaning**
?	**Print contents from *a.out* file**
/	**Print contents from *core* file**
=	**Print value of "dot"**
:	**Breakpoint control**
$	**Miscellaneous requests**

;	**Request separator**
!	**Escape to shell**

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp.* Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

adb a.out core

The first debugging request:

$c

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

$r

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

$e

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by **?** whereas the map for *core* file is referenced by **/**. Furthermore, a good rule of thumb is to use **?** for instructions and **/** for data when looking at programs. To print out information about the maps type:

$m

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp.* This is done by:

***charp/s**

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

main.argc/d

prints the decimal *core* image value of the argument *argc* in the function *main.*

The request:

***main.argv,3/o**

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main.* Note that these values are the addresses of the arguments to main. Therefore:

0177770/s

prints the ASCII value of the first argument. Another way to print this value would have been

***"/s**

The " means ditto which remembers the last address typed, in this case *main.argc* ; the ***** instructs ADB to use the address field of the *core* file as a pointer.

The request:

.=o

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.–10/d

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f, g,* and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

$c

will fill a page of backtrace references to *f, g,* and *h.* Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

,5$C

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f.* Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h,* type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with **$C** or the occurrence of a variable in the most recent call of a function. It is possible with the **$C** request, however, to print the stack frame starting at some address as **address$C.**

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

adb a.out –

Breakpoints are set in the program as:

address:b [request]

The requests:

settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

$b

The display indicates a *count* field. A breakpoint is bypassed *count −1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

settab,5?ia

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

settab,5?i

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the **?** command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

:r

To delete a breakpoint, for instance the entry to the function *settab,* one types:

settab+4:d

To continue execution of the program from the breakpoint type:

:c

Once the program has stopped (in this case at the breakpoint for *fopen),* ADB requests can be used to display the contents of memory. For example:

$C

to display a stack trace, or:

tabs,3/8o

to print three lines of 8 locations each from the array called *tabs.* By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs.*

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

:c

See Figure 6b. *Getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

ibuf+6/20c

When we continue the program with:

:c

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

tabpos+4:d

If the program is continued with:

:c

it resumes normal execution after ADB prints the message

a.out:running

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

:c

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

:c 0

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

settab+4:b settab,5?ia *

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

getc+4,3:b main.c?C *

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these statements must be written as:

settab+4:b settab,5?ia;0
getc+4,3:b main.c?C;0
settab+4:b settab,5?ia; ptab/o;0

Note that **;0** will set dot to zero and stop at the breakpoint.

semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

settab+4:b .,5?ia
fopen+4:b

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

settab+4:b settab,5?ia; ptab/o *

could be entered after typing the above requests.

Now the display of breakpoints:

$b

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f, g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

adb ex3 –

Suppose we enter the following breakpoints:

```
h+4:b     hcnt/d;  h.hi/;  h.hr/
g+4:b     gcnt/d;  g.gi/;  g.gr/
f+4:b     fcnt/d;  f.fi/;  f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b     fcnt/d;  f.a/;  f.b/;  f.fi/
g+4:b     gcnt/d;  g.p/;  g.q/;  g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a*, *b* and *fi*.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b     fcnt/d;  f.a/"a="d;  f.b/"b="d;  f.fi/"fi="d
```

In this format the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  :r arg1 arg2 ... <infile >outfile

  This request kills any existing program under test and starts the *a.out* afresh.

- The program being debugged can be single stepped by:

  :s

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

  address:r

- The count field can be used to skip the first *n* breakpoints as:

  ,n:r

  The request:

  ,n:c

  may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

  address:c

- The program being debugged runs as a separate process and can be killed by:

  :k

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc -n pgm.c**, whereas a 411 file is produced by **cc -i pgm.c**. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the *a.out* file. The **?*** request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the **?*** operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file).

The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

$$\mathbf{b1 \le A \le e1 \Rightarrow file\ address = (A - b1) + f1}$$
$$\mathbf{b2 \le A \le e2 \Rightarrow file\ address = (A - b2) + f2}$$

A user can access locations by using the ADB defined variables. The **$v** request prints the variables initialized by ADB:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

<b

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

02000>b

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

<b,–1/4o4ˆ8Cn

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b	The base address of the data segment.
<b,–1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4ˆ8Cn** is broken down as follows:

4o	Print 4 octal locations.
4ˆ	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in

the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.

n Print a newline.

The request:

<b,<d/4o4ˆ8Cn

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

adb a.out core < dump

to read in a script file, *dump,* of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,–1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

symbol + offset

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request **=** can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

=3n"C Stack Backtrace"

that spaces three lines and prints the literal string. The request **$v** prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

<b,–1/8ona

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

adb dir –

=n8t"Inum"8t"Name"
0,–1? u8t14cn

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer’s Manual Section V) could be dumped with the following set of requests:

adb /dev/src –
02000>b
?m <b
<b,–1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

072 = odx

will print

072 58 #3a

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

’a’ = co

prints

a 0141

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write*, **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate*, **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

?l value

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

adb –w file1 file2

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

adb −w ex7 –
?l 'Th'
?W 'The '

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

?l 'Th'; ?s

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

adb a.out –
:s arg1 arg2
flag/w 1
:c

The **:s** request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the **w** request causes *flag* to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if **?** is used for text (instructions) and **/** for data.
3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
        char    cc;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((fcreat(argv[1],obuf)) < 0){
                printf("%s : not found\n", argv[1]);
                exit(8);
        }
        charp = ´T´;
printf("debug 1 %s\n",charp);
        while(cc=  *charp++)
                putc(cc,obuf);
        fflush(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
        argc:       02
        argv:       0177762
        cc:         02124
$r
ps      0170010
pc      0204    ~main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
~main+0152:     mov     _obuf,(sp)
$e
savr5:      0
_obuf:      0
_charp:     0124
_errno:     0
_fout:      0
$m
text map   `ex1´
b1 = 0              e1    = 02360           f1 = 020
b2 = 0              e2    = 02360           f2 = 020
data map   `core1´
b1 = 0              e1    = 03500           f1 = 02000
b2 = 0175400        e2    = 0200000         f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx      Nh@x &_
~
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756: 2
*main.argv/3o
0177762: 0177770  0177776  0177777
0177770/s
0177770: a.out
*main.argv/3o
0177762: 0177770  0177776  0177777
*"/s
0177770: a.out
 .=o
                0177770
 .–10/d
0177756: 2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x−y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q−p;
        gr = q−p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
        x:      04452
        y:      04451
        hi:     ?
~g(04453,011124)
        p:      04453
        q:      011124
        gi:     04451
        gr:     ?
~f(02,04451)
        a:      02
        b:      04451
        fi:     011124
        fr:     04453
~h(04450,04447)
        x:      04450
        y:      04447
        hi:     04451
        hr:     02
~g(04451,011120)
        p:      04451
        q:      011120
        gi:     04447
        gr:     04450
fcnt/d
_fcnt:          1173
gcnt/d
_gcnt:          1173
hcnt/d
_hcnt:          1172
h.x/d
022004:         2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE 80
#define YES         1
#define NO          0
#define TABSP       8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

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

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != −1) {
                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++ ;
                }
        }
}

/* 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 6a: ADB output for C program of Figure 5

```
adb a.out −
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count   bkpt            command
1       ˜tabpos+04
1       _getc+04
1       _fopen+04
1       ˜settab+04
settab,5?ia
˜settab:        jsr     r5,csv
˜settab+04:     tst     −(sp)
˜settab+06:     clr     0177770(r5)
˜settab+012:    cmp     $0120,0177770(r5)
˜settab+020:    blt     ˜settab+076
˜settab+022:
settab,5?i
˜settab:        jsr     r5,csv
                tst     −(sp)
                clr     0177770(r5)
                cmp     $0120,0177770(r5)
                blt     ˜settab+076
:r
a.out: running
breakpoint      ˜settab+04:     tst     −(sp)
settab+4:d
:c
a.out: running
breakpoint      _fopen+04:      mov     04(r5),nulstr+012
$C
_fopen(02302,02472)
˜main(01,0177770)
        col:    01
        c:      0
        ptab:   03500
tabs,3/8o
03500:          01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
                01      0       0       0       0       0       0       0
```

Figure 6b: ADB output for C program of Figure 5

```
:c
a.out: running
breakpoint        _getc+04:         mov       04(r5),r1
ibuf+6/20c
__cleanu+0202:              This      is        a test    of
:c
a.out: running
breakpoint        ~tabpos+04:       cmp       $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count     bkpt              command
1         ~tabpos+04
3         _getc+04 main.c?C;0
1         _fopen+04
1         ~settab+04        settab,5?ia;ptab?o;0
~settab:          jsr       r5,csv
~settab+04:       bpt
~settab+06:       clr       0177770(r5)
~settab+012:      cmp       $0120,0177770(r5)
~settab+020:      blt       ~settab+076
~settab+022:
0177766: 0177770
0177744: @`
T0177744:         T
h0177744:         h
i0177744:         i
s0177744:         s
```

Figure 7: ADB output for C program with breakpoints

```
adb ex3 –
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:      0
0177732:    214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:      0
0177746:    1
0177750:    1
0177732:    214
_gcnt:      0
0177726:    2
0177730:    3
0177712:    214
_hcnt:      0
0177706:    2
0177710:    1
0177672:    214
_fcnt:      1
0177666:    2
0177670:    3
0177652:    214
_gcnt:      1
0177646:    5
0177650:    8
0177632:    214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:      0
0177746:    a = 1
0177750:    b = 1
0177732:    fi = 214
_gcnt:      0
0177726:    p = 2
0177730:    q = 3
0177712:    gi = 214
_hcnt:      0
0177706:    x = 2
0177710:    y = 1
0177672:    hi = 214
_fcnt:      1
0177666:    a = 2
0177670:    b = 3
0177652:    fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

407 files

```
a.out        hdr                  text+data
           |______| ________________________________________|
                   0                                        D

core         hdr                  text+data                           stack
           |_________| ________________________________________......| _________________|
                      0                                        D      S                 E
```

410 files (shared text)

```
a.out        hdr                  text                                data
           |______| ________________________________________| _______________________|
                   0                                        T  B                     D

core         hdr            data                   stack
           |_________| ____________________......| _________________|
                      B                    D      S                 E
```

411 files (separated I and D space)

```
a.out        hdr                  text                                data
           |______| ________________________________________| _______________________|
                   0                                        T  0                     D

core         hdr            data                   stack
           |_________| ____________________......| _________________|
                      0                    D      S                 E
```

The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D–B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    `map407´
b1 = 0              e1 = 0256           f1 = 020
b2 = 0              e2 = 0256           f2 = 020
data map    `core407´
b1 = 0              e1 = 0300           f1 = 02000
b2 = 0175400        e2 = 0200000    f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q


adb map410 core410
$m
text map    `map410´
b1 = 0              e1 = 0200           f1 = 020
b2 = 020000         e2 = 020116     f2 = 0220
data map    `core410´
b1 = 020000         e1 = 020200     f1 = 02000
b2 = 0175400        e2 = 0200000    f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q


adb map411 core411
$m
text map    `map411´
b1 = 0              e1 = 0200           f1 = 020
b2 = 0              e2 = 0116           f2 = 0220
data map    `core411´
b1 = 0              e1 = 0200           f1 = 02000
b2 = 0175400        e2 = 0200000    f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
char    str1[]  "This is a character string";
int     one     1;
int     number  456;
long    lnum    1234;
float   fpt     1.25;
char    str2[]  "This is the second character string";
main()
{
        one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,–1/8ona
020000:          0       064124    071551    064440    020163    020141    064143071141

_str1+016:  061541    062564    020162    072163    064562    063556    0      02

_number:
_number:    0710  0    02322     040240    0      064124    071551    064440

_str2+06:   020163    064164    020145    062563    067543    062156    061440060550

_str2+026:  060562    072143    071145    071440    071164    067151    0147  0

savr5+02:   0     0    0     0    0     0    0     0

<b,20/4o4ˆ8Cn
020000:          0       064124    071551    064440    @`@`This i
            020163    020141    064143    071141    s a char
            061541    062564    020162    072163    acter st
            064562    063556    0      02    ring@`@`@b@`

_number:    0710  0    02322     040240    H@a@`@`R@d @@
            0     064124    071551    064440    @`@`This i
            020163    064164    020145    062563    s the se
            067543    062156    061440    060550    cond cha
            060562    072143    071145    071440    racter s
            071164    067151    0147  0    tring@`@`@`
            0     0    0     0    @`@`@`@`@`@`@`@`
            0     0    0     0    @`@`@`@`@`@`@`@`
data address not found
<b,20/4o4ˆ8t8cna
020000:          0       064124    071551    064440         This i
_str1+06:   020163    020141    064143    071141         s a char
_str1+016:  061541    062564    020162    072163         acter st
_str1+026:  064562    063556    0      02         ring
_number:
_number:    0710  0    02322     040240         HR
_fpt+02:    0     064124    071551    064440         This i
_str2+06:   020163    064164    020145    062563         s the se
_str2+016:  067543    062156    061440    060550         cond cha
_str2+026:  060562    072143    071145    071440         racter s
_str2+036:  071164    067151    0147  0         tring
savr5+02:   0     0    0     0
savr5+012:  0     0    0     0
data address not found
<b,10/2b8tˆ2cn
020000:          0    0

_str1:      0124  0150      Th
            0151  0163      is
            040   0151       i
            0163  040       s
            0141  040       a
            0143  0150      ch
            0141  0162      ar
            0141  0143      ac
            0164  0145      te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir –
=nt"Inode"t"Name"
0,–1?ut14cn

        Inode Name
0:      652   .
        82    ..
        5971  cap.c
        5323  cap
        0     pp
```

```
adb /dev/src –
02000>b
?m<b
new map     `/dev/src´
b1 = 02000    e1 = 0100000000    f1 = 0
b2 = 0        e2 = 0             f2 = 0
$v
variables
b = 02000
<b,–1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:          flags  073145
        links,uid,gid   0163  0164  0141
        size    0162  10356
        addr    28770   8236  25956   27766   25455   8236  25956   25206
        times   1976 Feb 5 08:34:56    1975 Dec 28 10:55:15

02040:          flags  024555
        links,uid,gid   012   0163  0164
        size    0162  25461
        addr    8308  30050   8294  25130   15216   26890   29806   10784
        times   1976 Aug 17 12:16:51   1976 Aug 17 12:16:51

02100:          flags  05173
        links,uid,gid   011   0162  0145
        size    0147  29545
        addr    25972   8306  28265   8308  25642   15216   2314  25970
        times   1977 Apr 2 08:58:01    1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a) formatted printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

>*name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
*n***t**	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
ˆ	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

+	add
–	subtract
*****	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

~	not
*****	contents of location
–	integer negate