Sdb: A Symbolic Debugger

Howard P. Katseff

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

Sdb is a symbolic debugging program currently implemented for the language C on the UNIX/32V[†] Operating System. Sdb allows one to interact with a debugged program at the C language level. When debugging a core image from an aborted program, sdb reports which line in the C program caused the error and allows all variables, including array and structure elements, to be accessed symbolically and displayed in the correct format.

One may place breakpoints at selected statements or single step on a line by line basis. To facilitate specification of lines in the program without a source listing, a mechanism for examining the source text is also included in sdb.

Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines which provide formatted printout of structured data.

September 28, 1987

[†]UNIX is a trademark of Bell Laboratories

Sdb: A Symbolic Debugger

Howard P. Katseff

Bell Laboratories
Holmdel, New Jersey 07733

1. Introduction

This document describes a symbolic debugger, sdb, as implemented for C programs on the UNIX/V32† Operating System. Sdb is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

2. Examining core images

In order to use sdb, it is necessary to compile the C program with the '−g' flag. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the debug flag is specified, sdb can be used to obtain a trace of the called procedures at the time of the abort and interactively display the values of variables.

2.1. Invoking sdb

A typical sequence of shell commands for debugging a core image is:

```
% cc −g foo.c −o foo
% foo
Bus error − core dumped
% sdb foo
main:25:     x[i] = 0;
*
```

The program foo was compiled with the '−g' flag and then executed. An error occurred which caused a core dump. Sdb is then invoked to examine the core dump to determine the cause of the error. It reports that the Bus error occurred in procedure main at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. Sdb then prompts the user with a '*' indicating that it awaits a command.

It is useful to know that sdb has a notion of current procedure and current line. In this example, they are initially set to 'main' and '25' respectively.

In the above example sdb was called with one argument, 'foo'. In general it takes three arguments on the command line. The first is the name of the executable file which is to be debugged; It defaults to a.out when not specified. The second is the name of the core file, defaulting to core and the third is the name of the directory containing the source of the program being debugged. Sdb currently requires all source to reside in a single directory. The default is the working directory. In the example the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a procedure which was not compiled with the debug flag. In this case, sdb prints the procedure name and the address at which the error occurred. The current line and procedure are set to the first line in main. Sdb will complain if main was not compiled with '−g' but debugging can continue for those routines compiled with the debug flag.

---

†UNIX is a trademark of Bell Laboratories

2.2. Printing a stack trace

It is often useful to obtain a listing of the procedure calls which led to the error. This is obtained with the **t** command. For example:

```
*t
sub(2,3)    [foo.c:25]
inter(16012)    [foo.c:96]
main(1,2147483584, 2147483592)    [foo.c:15]
```

This indicates that the error occurred within the procedure sub at line 25 in file foo.c. Sub was called with the arguments 2 and 3 for inter at line 96. Inter was called from main at line 16. Main is always called by the shell with three arguments, often referred to as *argc*, *argp* and *envp*. Arguments in the call trace are always printed in decimal.

2.3. Examining variables

Sdb can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflg/
```

causes sdb to display the value of variable errflg. Unless otherwise specified, variables are assumed to be either local to or accessible from the current procedure. To specify a different procedure, use the form

```
*sub:i/
```

to display variable i in procedure sub. Section 3.2 will explain how to change the current procedure.

Sdb normally displays the variable in a format determined by its type as declared in the C program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are

b one byte
h two bytes (half word)
l four bytes (long word)

The lengths are only effective with the formats **d**, **o**, **x** and **u**. If no length is specified, the word length of the host machine, four for the DEC VAX-11/780†, is used. There are a number of format specifiers available:

c character
d decimal
u decimal unsigned
o octal
x hexadecimal
f 32 bit single precision floating point
g 64 bit double precision floating point
s Assume variable is a string pointer and print characters until a null is reached.
a Print characters starting at the variable's address until a null is reached.

As an example, variable i can be displayed in hexadecimal with the following command

```
*i/x
```

Sdb also knows about structures, one dimensional arrays and pointers so that all of the following commands work.

---

†DEC and VAX are trademarks of Digital Equipment Corporation

∗array[2]/
∗sym.id/
∗psym–>usage/
∗xsym[20].p–>usage/

The only restriction is that array subscripts must be numbers. Note that, as a special case

∗psym–>/d

displays the location pointed to by psym in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

∗1024/

displays location 1024 in decimal. As in C, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both of

∗02000/
∗0x400/

It is possible to intermix numbers and variables, so that

∗1000.x/

refers to an element of a structure starting at address 1000 and

∗1000–>x/

refers to an element of a structure whose address is at 1000.

The address of a variable is printed with the ‘=’ command, so

∗i=

displays the address of i. Another feature whose usefulness will become apparent later is the command

∗./

which redisplays the last variable typed.

3. Source file display and manipulation

Sdb has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided which perform context searches within the source files of the program being debugged and to display selected portions of the source files. The commands are similar to those of the UNIX editor ed and ex [1]. Like these editors, sdb has a notion of current file and line within the file. Sdb also knows how the lines of a file are partitioned into procedures, so that it also has a notion of current procedure. As noted in other parts of this document, the current procedure is used by a number of sdb commands.

3.1. Displaying the source file

Four command exist for displaying lines in the source file. They are useful for perusing through the source program and for determining the context of the current line. The commands are

w Window. Print a window of 10 lines around the current line.

z Print 10 lines starting at the current line. Advance the current line by 10.

control-D Scroll. Print the next 10 lines and advance the current line by 10. This command is used to cleanly display longs segments of the program.

There is also a **p** command which prints the current line. When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but is also used as input by some sdb commands.

3.2. Changing the current source file or procedure

The **e** command is used to change the current source file. Either of the forms

```
*e procedure
*e file.c
```

may be used. The first causes the file containing the named procedure to become the current file and the current line becomes the first line of the procedure. The other form causes the named file to become current. In this case the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current procedure and file named to be printed.

3.3. Changing the current line in the source file

As mentioned in section 3.1, the **z** and **control-D** commands have a side effect of changing the current line in the source file. This section describes other commands which change the current line.

There are two commands for searching for regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string which matches the regular expression and the second searches backwards. The trailing '/' and '?' may be omitted from these commands. Regular expression matching is identical to that of ed.

The **+** and − commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a newline advances the current line by one and typing a number causes that line to become the current line in the file. These commands may be catenated with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints 10 lines.

4. A controlled environment for program testing

One very useful feature of sdb is breakpoint debugging. After entering the debugger, certain lines in the source program may be specified to be *breakpoints.* The program is then started with a sdb command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and sdb reports which breakpoint the program is stopped at. Now, sdb commands may be used to display the trace of procedure calls and the values of variables. If the user is satisfied that the program is working correctly to this point, some breakpoints can be deleted and others set, and then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. Sdb can be requested to execute the next line of the program and them stop. This feature is especially useful for testing new programs, so they can be verified on a statement by statement basis. Note that if an attempt is made to single step through a procedure which has not been compiled with the '−g' flag, execution proceeds until a statement in a procedure compiled with the debug flag is reached.

The current implementation of single stepping is rather slow. While this is not a problem when stepping through a single statement, it may result in long delays while stepping through procedures not compiled with the debug flag. This problem is partially alleviated with the **n** command which quickly single steps until the positionally next statement is reached.

4.1. Setting and deleting breakpoints

Breakpoints can be set at any line in a procedure which contains executable code. The command format is:

```
*12b
*proc:12b
*proc:b
```

The first form sets a breakpoint at line 12 in the current procedure. The line numbers are relative to the beginning of the file, as printed by the source file display commands. The second form sets a breakpoint at line 12 of procedure proc and the third sets a breakpoint at the first line of proc.

Breakpoints are deleted similarly with the commands:

```
∗12d
∗proc:12d
∗proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the user. If the line begins with a ‘y’ or ‘d’, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **b** command. Beware that breakpoints do strange things if the debugged program is being run elsewhere at the same time.

4.2. Running the program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell. The command

```
∗r args
```

runs the program with the given arguments, as if they had been typed on the shell command line.

Execution is continued after a breakpoint with the **c** command and single stepping is accomplished with **s**. The **n** command is used to run the program until it reaches the positionally next statement.

Program execution can also be stopped with the RUBOUT key. The debugger is entered as if a breakpoint was encountered so that execution may be continued with **c**, **s** or **n**.

4.3. Calling procedures

It is possible to call any of the procedures of the program from the debugger. This feature is useful both for testing individual procedures with different arguments and for calling a procedure which prints structured data in a nice way. There are two ways to call a procedure:

```
∗proc(arg1, arg2, ...)
∗proc(arg1, arg2, ...)/
```

The first simply executes the procedure. The second is intended for calling functions: It executes the procedure and prints the value that it returns. The value is printed in decimal unless some other format is specified. Arguments to procedures may be integer, character or string constants, or values of variables which are accessible from the current procedure.

An unfortunate bug in the current implementation is that if a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), static variables are reinitialized before the procedure is restarted. This makes it impossible to use a procedure which formats data from a dump.

5. Other commands

To exit the debugger, use the **q** command.

The **!** command is identical to that in ed and is used to have the shell execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

```
∗variable!value
```

which sets the variable to the given value. The value may be a number, character constant or the name of another variable.

Acknowledgments

I would like to thank Bill Joy and Chuck Haley for their comments and constructive criticisms.

Reference

[1] William N. Joy, Ex Reference Manual, Computer Science Division, University of California, Berkeley, November 1977.

Appendix 1. Example of usage.

Appendix 2. Manual pages.

6. Introduction.

A symbolic debugger, sdb, has been implemented for the UNIX/32V operating system. This document describes modifications made to the C compiler to generate additional information about the compiled program and to the assembler and loader to process the information. It also describes information recognized by the assembler, the loader and sdb which are intended for use by compilers for other languages such as F77.

7. The C Compiler

The C compiler was modified to generate additional symbol table information describing a compiled program. Two new types of symbol table entries are made. One describes the variables, giving their class (local, register, parameter, global, etc.), their declared type in the program and their address or offset. An additional entry is made for structures giving their size. The other type of entry provides a mapping between the source program and the object program. There is an entry for each source line, procedure and source file giving their addresses in the object file. All line numbers are relative to the beginning of the source file.

All entries are generated with the new assembler pseudo-operation ‘.stab’. It always takes 12 arguments of which the first eight usually represent the name of the symbol as declared in the C program. An underscore is *not* prepended to the name as in some other symbol table entries. A typical entry would be

```
.stab   'e,'r,'r,'f,'l,'g,0,0,046,0,05,_errflg
```

For expository convenience, names in .stab entries will be listed as one word instead of eight separate characters.

7.1. External symbols defined with .comm

The following entry is made for each external symbol which is defined with a .comm pseudo-op.

```
.stab   name,040,0,type,0
```

The type is a 16-bit value describing the variable’s declared type. This field is described in section 2.13. The debugger determines the variable’s address from the entry made with the .comm. It assumes that the name for this entry is _name.

7.2. Symbols defined within .data areas

The following entry is made for each symbol which is defined as a label in a data area.

```
.stab   name,046,0,type,address
```

The type is the variable’s declared type. The address is given symbolically as the label.

7.3. Symbols defined with .lcomm

The following entry is made for each symbol which is defined with a .lcomm pseudo-op.

```
.stab   name,048,0,type,address
```

The type is the variable’s declared type. The address is given symbolically as the label. The specification of an octal constant with an 8 occurs for historical reasons.

7.4. Register symbols

The following entry is made for each variable whose value is in a register.

```
.stab   name,0100,0,type,register
```

The type is the variable’s declared type. The register is the register number assigned to the variable.

7.5. Local non-register symbols

The following entry is made for each local, non-register variable.

.stab name,0200,0,type,offset

The type is the variable's declared type. The offset is a positive number indicating its offset in bytes for the frame pointer.

7.6. Parameter symbols

The following entry is made for each procedure parameter.

.stab name,0240,0,type,offset

The type is the variable's declared type. The offset is a positive number indicating its offset in bytes from the stack pointer.

7.7. Structure elements

The following entry is made for each structure element.

.stab name,0140,0,type,offset

The type is the element's declared type. The offset is its offset within the structure in bytes.

7.8. Structure symbols

An additional entry is made for structures giving their size in bytes. It immediately follows their defining .stab entry. It is of the form

.stab name,0376,0,0,length

7.9. Common blocks

The following sequence of entries is used to describe elements of Fortran equivalence and common blocks. The first is of the form

.stab 0,0342,0,0,0

The entries for each element of the block should then appear as if they were structure elements. Finally, one of the following two entries is used depending on the type of common or equivalence block. If the block is defined as a .globl symbol, use the entry

.stab name,0344,0,0,0

where name is the name of the block defined in the .globl statement. It the block is defined in some other way, use

.stab 0,0348,0,0,address

7.10. Brackets

Since C is a block-structured language, it is necessary to know the extent of each block containing symbol definitions. An entry is made for each right and left bracket which encloses a block with definitions. The following entries are for left and right brackets respectively.

.stab 0,0300,0,nesting level,address
.stab 0,0340,0,nesting level,address

The nesting level is the static nesting level of the block. It is currently ignored by the debugger. The address is the address of the first byte of code for the block for the left brackets and the first byte following the block for right brackets.

7.11. Procedures

The following entry is made for each procedure.

```
.stab name,044,0,linenumber,address
```

The linenumber is the number of the first line of the procedure in the source file. The address is the address of the first byte of the procedure.

7.12. Lines

The following entry is made for each line in the source program.

```
.stab 0,0104,0,linenumber,address
```

The linenumber is its number. The address is the address of the first byte of code for the line. For each block of the program, the linenumber entries for that block should follow the entries for the variables of that block.

7.13. Source files

The following entries are made for each source file.

```
.stab name1,0144,0,0,address
.stab name2,0144,0,0,address
...
.stab namen,0144,0,0,address
```

Each entry contains 8 successive bytes of the name of the source file. The name is terminated by a null byte. All bytes following this one should also be null. The address is the address of the first byte of code for the first procedure of the file.

7.14. Included source files

The following entry is made for each included source file which generates code.

```
.stab name1,0204,0,0,address
.stab name2,0204,0,0,address
...
.stab namen,0204,0,0,address
```

This entry should appear each time the file is included. A similar entry giving the name of the original file should be made at the end of the include. The format of the name is identical to that for files. This feature is heavily used by programs generated by yacc and lex.

7.15. Format of types.

This 16 bit quantity type describes the declared type of a variable. We use the same scheme as in S.C. Johnson's Portable C Compiler [Johnson, 1978]. The type is divided into the following fields:

```
struct {
short basic:4;
d1:2,
d2:2,
d3:2,
d4:2,
d5:2,
d6:2,
}
```

There are four derived types:

0 none
1 pointer
2 function
3 array

They are indicated in the two bit fields d1, d2, d3, d4, d5 and d6. The four bit field basic indicates the basic type as follows:

0 undefined
1 function argument
2 character
3 short
4 int
5 long
6 float
7 double
8 structure
9 union
10 enumerated type
11 member of enumerated type
12 unsigned character
13 unsigned short
14 unsigned
15 unsigned long

8. The assembler and loader

Each .stab pseudo-operation generates one entry in the symbol table. The entry is of the form:

```
struct {
charname[8];
chartype;
charother;
shortdesc;
unsigned value;
}
```

The loader uses the four least significant bits of the type field to determine how to relocate the .stab entry. The following are currently used.

0 none
4 text
6 data

It is necessary for the assembler and loader to preserve the order of symbol table entries produced by .stab pseudo-ops.

Reference

Johnson, S.C., "A Portable Compiler: Theory and Practice", *Proc. 5th ACM Symp. on Principles of Programming Languages*, January 1978.

Appendix

The following definitions are extracted from the file /usr/include/a.out.h.

```
struct nlist { /* symbol table entry */
        char   n_name[8];  /* symbol name */
        char   n_type;             /* type flag */
        char   n_other;
        short  n_desc;
        unsigned n_value; /* value */
};

        /* values for type flag */
#define       N_UNDF    0            /* undefined */
#define       N_ABS     02           /* absolute */
#define       N_TEXT    04           /* text */
#define       N_DATA    06           /* data */
#define       N_BSS     08
#define       N_TYPE    037
#define       N_FN      037          /* file name symbol */

#define       N_GSYM    0040         /* global sym: name,,type,0 */
#define       N_FUN     0044         /* function: name,,linenumber,address */
#define       N_STSYM 0046           /* static symbol: name,,type,address */
#define       N_LCSYM 0048           /* .lcomm symbol: name,,type,address */
#define       N_RSYM    0100         /* register sym: name,,register,offset */
#define       N_SLINE   0104         /* src line: ,,linenumber,address */
#define       N_SSYM    0140         /* structure elt: name,,type,struct_offset */
#define       N_SO      0144         /* source file name: name,,,address */
#define       N_LSYM    0200         /* local sym: name,,type,offset */
#define       N_SOL     0204         /* #line source filename: name,,,address */
#define       N_PSYM    0240         /* parameter: name,,type,offset */
#define       N_LBRAC  0300          /* left bracket: ,,nesting level,address */
#define       N_RBRAC  0340          /* right bracket: ,,nesting level,address */
#define N_BCOMM 0342                 /* begin common: name,,, */
#define N_ECOMM 0344                 /* end common: name,,, */
#define N_ECOML 0348                 /* end common (local name): ,,,address */
#define       N_LENG    0376         /* second stab entry with length information */

#define       N_EXT     01           /* external bit, or'ed in */

#define       FORMAT    "%08x"

#define       STABTYPES        0340
```

Howard P. Katseff

-- --

Copy to
R.W. Lucky
C.S. Roberts % cat testdiv2.c main() { int i; i = div2(-1); printf("-1/2 = %d0, i); } div2(i) { int j; j = i>>1; return(j); }

% cc -g testdiv2.c

% a.out -1/2 = -1

% sdb No core image # Warning message from sdb */^div2 # Search for procedure "div2" 6: div2(i) { # It starts at line 6 *z # Print the next few lines 6: div2(i) { 7: int j; 8: j = i>>1; 9: return(j); 10: } *div2:b # Place a breakpoint at beginning of div2 div2:8 b # Sdb echoes proc name and line number *r # Run the procedure Breakpoint at # Execution stops just before line 8 div2:8: j = i>>1; *t # Print trace of subroutine calls div2(-1) [testdiv2.c:8] main(1,2147483380,2147483388) [testdiv2.c:3] *i/ # Print i -1 *s # Single step div2:9: return(j); # Execution stops just before line 9 *j/ # Print j -1 *8d # Delete the breakpoint *div2(1)/ # Try running div2 with different args 0 *div2(-2)/ -1 *div2(-3)/ -2 *q # Exit sdb