SDB—A Symbolic Debugger

H. P. Katseff

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

Sdb is a symbolic debugging program currently implemented for the languages C and F77 on the UNIX† operating system. *Sdb* allows one to interact with a debugged program at the source language level. When debugging a "core image" from an aborted program, *sdb* reports which line in the source 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, *sdb* provides a mechanism for examining the source text.

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.

1. INTRODUCTION

This document describes a symbolic debugger, *sdb*, as implemented for C and F77 programs on the UNIX 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 source 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;
*
```

† UNIX is a trademark of Bell Laboratories.

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.

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(x=2,y=3)        [foo.c:25]
inter(i=16012)      [foo.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c)        [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 x=2 and y=3 from *inter* at line 96. *Inter* was called from *main* at line 15. *Main* is always called by the shell with three arguments, often referred to as *argc*, *argv* and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

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*. F77 users can specify a common block name in the same manner. Section 3.2 will explain how to change the current procedure.

Sdb supports a limited form of pattern matching for variable and procedure names. The symbol "*" is used to match any sequence of characters of a variable name and "?" to match any single character. Consider the following commands:

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with "x", the second prints the values of all two letter variables in procedure *sub* beginning with "y", and the last prints all variables. In the first and last examples, only variables accessible from the current procedure are printed. The command

```
**:*/
```

displays the variables for each procedure on the call stack.

Sdb normally displays the variable in a format determined by its type as declared in the source 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, (i.e., 4 for the DEC VAX-11/780) is used. A numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed.

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.
- **p** pointer to procedure.
- **i** interpret as a machine-language instruction.

As an example, the 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.

```
*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 editors *ed* [1] and *ex* [2]. 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 the source program and for determining the context of the current line. The commands are:

p	Print the current line.
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 long segments of the program.

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 that change the current line.

There are two commands for searching for instances of 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 that 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 new-line 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 then 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.

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
*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. The last sets a breakpoint at the current line.

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 and the **D** command deletes all breakpoints. It is sometimes desirable to have *sdb* automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command:

```
*12 b t;x/
```

causes both a trace-back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a special case of the above command. There are two forms:

```
*proc: a
*proc:12 a
```

The first prints the procedure name and its arguments each time it is called and the second prints the source line each time it is about to be executed.

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. If no arguments are specified, then the arguments from the last execution of the program are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs or the program terminates. In all cases, after an appropriate message is printed, control returns to *sdb*.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12 c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command which continues, but passes the signal which stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example,

```
*17 g
```

continues at line 17 of the current procedure. A use for this command is to avoid executing a section of code which is known to be bad. The user should not attempt to continue execution in a different procedure than that of the breakpoint.

The **s** command is used to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command, but does not stop within called procedures. It is often used when one is confident that the called procedure works correctly, but is interested in testing the calling routine.

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), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a dump.

5. MACHINE LANGUAGE DEBUGGING

Sdb has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. *Sdb* can also be used to display or modify the contents of the machine registers.

5.1. Displaying Machine Language Statements

To display the machine language statements associated with line 25 in procedure *main*, use the command

```
*main:25?
```

The **?** command is identical to the / command except that it displays from text space. The default format for printing text space is the **i** format, which interprets the machine language instruction. The **control-d** command may be used to print the next 10 instructions.

Absolute addresses may be specified instead of line numbers by appending a ":" to them so that

```
*0x1024:?
```

displays the contents of address **0x1024** in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line **0x1024** in the current procedure. It is also possible to set or delete a breakpoint by specifying its absolute address:

```
*0x1024:b
```

sets a breakpoint at address **0x1024**.

5.2. Manipulating Registers

The **x** command prints the values of all the registers. Also, individual registers may be named instead of variables by appending a '%' to their name, so that

```
*r3%!22
```

changes the value of register **r3** to 22.

6. 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. If the variable is of type float or double, the value can also be a floating-point constant.

ACKNOWLEDGEMENT

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

REFERENCES

[1] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).

[2] Joy, W. N. *Ex Reference Manual,* Computer Science Division, University of California, Berkeley, November 1977.

Appendix 1. EXAMPLE OF USAGE

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
        int i;
        i = div2(-1);
        printf("-1/2 = %d\n", 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"
7: div2(i) {                 # It starts on line 7
*z                           # Print the next few lines
7: div2(i) {
8:      int j;
9:      j = i>>1;
10:     return(j);
11: }
*div2:b                      # Place a breakpoint at the beginning of "div2"
div2:9 b                     # Sdb echoes proc name and line number
*r                           # Run the procedure
a.out                        # Sdb echoes command line executed
Breakpoint at                # Executions stops just before line 9
div2:9:  j = i>>1;
*t                           # Print trace of subroutine calls
div2(i=-1)   [testdiv2.c:9]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58)   [testdiv2.c:4]
*i/                          # Print i
-1
*s                           # Single step
div2:10:   return(j);        # Execution stops just before line 10
*j/                          # Print j
-1
*9d                          # Delete the breakpoint
*div2(1)/                    # Try running "div2" with different arguments
0
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```