15. sdb—THE SYMBOLIC DEBUGGER

Introduction

This chapter describes the symbolic debugger, **sdb**(1), as implemented for C language and Fortran 77 programs on the operating system. The **sdb** program 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.

The **sdb** program allows interaction 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 to be accessed symbolically and to be displayed in the correct format.

When executing, breakpoints may be placed at selected statements or the program may be single stepped 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 printouts of structured data.

Using sdb

To use **sdb** to its full capabilities, it is necessary to compile the source program with the **–g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **–g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

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

```
cc –g prgm.c –o prgm
prgm
Bus error - core dumped
sdb prgm
main:25:        x[i] = 0;
*
```

The program **prgm** was compiled with the **–g** option and then executed. An error occurred, which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The

sdb program then prompts the user with an *, which shows that it is waiting for a command.

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

Here **sdb** was called with one argument, **prgm**. In general, it takes three arguments on the command line. The first is the name of the executable file that 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 list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the **–g** option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the **–g** option, **sdb** will print an error message, but debugging can continue for those routines that were compiled with the **–g** option.

Figure 15-1, at the end of the chapter, shows a more extensive example of **sdb** use.

Printing a Stack Trace

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

```
*t
sub(x=2,y=3)          [prgm.c:25]
inter(i=16012)        [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c) [prgm.c:15]
```

This indicates that the program was stopped within the function **sub** at line 25 in file **prgm.c**. The **sub** function was called with the arguments **x**=2 and **y**=3 from **inter** at line 96. The **inter** function was called from **main** at line 15. The **main** function is always called by a startup routine 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.

Examining Variables

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

```
*errflag/
```

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

```
*sub:i/
```

to display variable **i** in function **sub**. FORTRAN 77 users can specify a common block variable in the same way, provided it is on the call stack.

The **sdb** program supports a limited form of pattern matching for variable and function 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 function **sub** beginning with **y**, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command:

```
**:*/
```

displays the variables for each function on the call stack.

The **sdb** program 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 length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can

be used with the **s** or **a** formats to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly 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 starting at the address pointed to by the variable until a null is reached.
- **a** Print characters starting at the variable's address until a null is reached.
- **p** Pointer to function.
- **i** Interpret as a machine-language instruction.

For example, the variable **i** can be displayed with:

```
*i/x
```

which prints out the value of **i** in hexadecimal.

sdb also knows about structures, arrays, and pointers so that all the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

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

```
*psym[0]
```

displays the structure 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 language, numbers may also be specified in octal or hexadecimal, so the above command is equivalent to both:

```
*02000/
```

and:

```
*0x400/
```

It is possible to mix numbers and variables so that the command:

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and the command:

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the type *1000.x/ and *1000–>x/, the **sdb** program uses the structure template of the last structured referenced.

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

```
*i=
```

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

```
*./
```

which redisplays the last variable typed.

Source File Display and Manipulation

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. There are facilities that perform context searches within the source files of the program being debugged and that display selected portions of the source files. The commands are similar to those of the operating system text editor **ed**(1). Like the editor, **sdb** has a notion of current file and line within the current file. **sdb** also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands 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	Prints the current line.
w	Window; prints a window of ten lines around the current line.
z	Prints ten lines starting at the current line. Advances the current line by ten.
control-d	Scrolls; prints the next ten lines and advances the current line by ten. 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 indicates its relative position in the file, but it is also used as input by some **sdb** commands.

Changing the Current Source File or Function

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

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. 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 function and file named to be printed.

Changing the Current Line in the Source File

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands that search 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**(1).

The + and – commands may be used to move the current line forward or backward 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 combined with the display commands so that the command:

```
*+15z
```

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

A Controlled Environment for Program Testing

One useful feature of **sdb** is breakpoint debugging. After entering **sdb**, breakpoints can be set at certain lines in the source program. The program is then started with an **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 the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; 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. If an attempt is made to single step through a function that has not been compiled with the **–g** option, execution proceeds until a statement in a function compiled with the **–g** option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the **–g** option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function compiled with the **-g** option. The command format is:

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

The first form sets a breakpoint at line 12 in the current file. 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 function **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 **d** command:

```
*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:

```
*12b 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 variation of the above command. There are two forms:

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

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

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 within **sdb** 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 the user.

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

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command that continues but passes the signal that 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, the command:

```
*17 g
```

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

The **s** command is used to run the program for a single statement. 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 functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine level instruction at a time while ignoring the signal that stopped the program. Its uses are similar to the **s** command. There is also an **I** command that causes the program to execute one machine level instruction at a time, but also passes the signal that stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to print structured data.

There are two ways to call a function:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

An unfortunate bug in the current implementation is that if a function 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 function is started. This makes it impossible to use a function that formats data from a dump.

Machine Language Debugging

The **sdb** program 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. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function **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 ten instructions.

Absolute addresses may be specified instead of line numbers by appending a **:** to them, so that the command:

```
*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 function. It is also possible to set or delete a breakpoint by specifying its absolute address the command:

```
*0x1024:b
```

sets a breakpoint at address 0x1024.

Manipulating Registers

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

```
*r3%
```

displays the value of register **r3**.

Other Commands

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

The **!** command (when used immediately after the * prompt) is identical to that in **ed**(1) and is used to have the shell execute a command. The **!** can also be used to change the values of variables or registers when the program is stopped at a breakpoint. This is done with the command:

```
*variable!value
*r3!value
```

which sets the variable or the named register to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

An sdb Session

An example of a debugging session using **sdb** is shown in Figure 15-1. Comments (preceded by a pound sign, #) have been added to help you see what is happening.

```
sdb myoptim - .:../common# enter sdb command
Source path: .:../common
No core image
*window:b              # set a breakpoint at start of window
0x2462 (window:1459+2) b
*r < m.s > out.m.s     # run the program
Breakpoint at
0x2462 in window:1459: window(size, func) register int size;
boolean(*func)(); {
*t                     # print stack trace
window(size=2,func=w2opt)    [optim.c:1459]
peep()    [peep.c:34]
pseudo(s=.def^Imain;^I.val^I.;^I.scl^I-1;^I.endef)    [local.c:483]
yylex()    [local.c:229]
main(argc=0,argv=0x1FFFE43,-1073610300)    [optim.c:227]
*z                     # print 10 lines of source
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461:    extern NODE *initw();
1462:    register NODE *pl;
1463:    register int i;
1464:
1465:    TRACE(window);
1466:
1467:    /* find first window */
1468:
*s                     # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                     # step
window:1465:     TRACE(window);
*s                     # step
window:1469:     wsize = size;
*s                     # step
window:1470:     if ((pl = initw(n0.forw)) == NULL)
*S                     # step through procedure call
window:1475:     for (opf = pf->back; ; opf = pf->back) {
*pl                    # show variable pl
0x86b38
*x                     # print the register contents
  r0/ 0x86b38          r1/ 0                r2/ 0x8796c
  r3/ 0x85830          r4/ 0x1FFFB8F        r5/ 0x1FFF907
  r6/ 0x1FFFB87        r7/ 0x86b38          r8/ 2
  ap/ 0x1FFFD23        fp/ 0x1FFFCF7        sp/ 0x1FFFCF7
 psw/ 0x2004           pc/ 0x24b0
0x24b0 (window:1475):      MOVW    0x80d8c,%r0  [-0x7f77f274,%r0]
```

Figure 15-1. Example of **sdb** Usage (Sheet 1 of 2)

```
*pl[0]                  # dereference the pointer
pl[0].forw/ 0x86b6c
pl[0].back/ 0x86ac8
pl[0].ops[0]/ mov.w
pl[0].uniqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
*pl->forw[0]            # dereference the pointer
pl->forw[0].forw/ 0x86ca0
pl->forw[0].back/ 0x86b38
pl->forw[0].ops[0]/ call
pl->forw[0].uniqid/ 0
pl->forw[0].op/ 9
pl->forw[0].nlive/ 3584
pl->forw[0].ndead/ 4099
*pl!pl->forw            # replace pl with pl->forw
*pl                     # show pl
0x86b6c
*c                      # continue
Breakpoint at
0x2462 in window:1459: window(size, func) register int size;
boolean (*func)(); {
*s                      # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                      # step
window:1465:    TRACE(window);
*size                   # show function argument size
3
*D                      # delete all breakpoints
All breakpoints deleted
*c                      # continue
Process terminated
*q                      # quit sdb
$
```

Figure 15-1. Example of **sdb** Usage (Sheet 2 of 2)