SABER-C USER'S GUIDE

Version 1.2

This Manual contains proprietary information which is the sole property of Saber Software, Inc. This Manual is furnished to authorized users of Saber-C solely to facilitate the user of Saber-C as specified in written agreements.

Saber-C is a trademark of Saber Software, Inc.

Copyright 1986, 1987 by Saber Software, Inc.
All rights reserved.

CHAPTER 1: Getting Started

To start Saber-C, enter the command **saber** to the shell.

```
% saber

        Saber-C (Version 1.2)
   Copyright (C) 1986, 1987 by Saber Software

Attaching: /lib/libc.a

Do you want instructions? n

1 ->
```

When **saber** starts up, it looks in the **saber** help directory for a file called .saberinit. This is the system startup file which was created when **saber** was installed on the system. It contains general system information and instructions. In the example shown above, the system startup file contains a command to load the standard C library /lib/libc.a.

After reading the system startup file, **saber** looks for a file called .saberinit in the current working directory and the user's home directory (in that order). If this file exists, it is read for the user's startup instructions. Instructions on how to create a .saberinit file are provided in the chapter on customizing **saber**.

If the .saberinit file is not found, **saber** prompts the user for instructions. This message can be eliminated by creating an empty .saberinit file in your home directory.

1.1. Command Line Switches

After the startup files have been read, the command line switches are processed in the order that they are seen. Any files that are listed on the command line will be loaded after the switches have been processed. The general format for **saber**'s command line switches is shown below. The arguments enclosed within square brackets are optional.

```
% saber -d -flog_name -gmail_address -hhelp_path -nnumber
        -s[startup_file] -S[startup_file] -Dname[=def] -Uname
        -Iheader_path -llib_name -Llib_path -G -w filename ...
```

Here is a brief explanation of each switch:

-d

Should be used when working on a dumb terminal or when running **saber** as a subprocess under another program, such as **emacs**. This switch turns off terminal dependent output. Prompts will not be highlighted, and raw mode input will be disabled so that a <Return> will be required when responding to a prompt.

-f*log_name*

Saves a copy of all input typed by the user in a file called *log_name*. All input is usually save in a temporary logfile that is deleted when the user exits **saber**.

-g*mail_address*

Specifies the the mailing address for the gripe command. This switch is necessary only if the default mailing address does not work. Use the version command to display the mailing address.

-h*help_path*

Specifies the directory in which on-line documentation and utility programs are located. This switch is necessary only if the help directory pathname was not setup properly when **saber** was installed. Use the version command to display the current pathname of the help directory.

-r*number*

Sets the size of the run-time stack to *number* nested function calls. The default size is approximately 1000 nested function calls The default size may need to be increased when executing highly recursive programs.

-s[*startup_file***]**

If *startup_file* is supplied, it is read at startup instead of the .saberinit, file, which is the default startup file. If a filename is not specified, the default startup file is ignored

-S[*startup_file***]**

If *startup_file* is supplied, it is read at startup instead of the default system startup file. If a filename is not specified, the system startup file is ignored.

-D*name***[=***def***]**

-U*name*

-I*header_path*

-l*lib_name*

-L*library_path*

-G

-w

These switches are passed to the load command along with the names of any files that are specified on the command line. For more information, consult the chapter on loading files or the description of the load command in the *Reference Manual*.

CHAPTER 2: The Saber-C Environment

Saber supports an interactive environment in which users can execute any part of the C language. The environment is completely modeless, allowing users to intermix **saber** commands and C statements without having to switch contexts. The commands associated with the environment are:

```
##          — redo the previous line of input
#N          — redo the Nth previous line of input

history     — display previously entered input
```

Although it is possible to define variables and functions in the environment, the user should refrain from entering C source code in the environment that is likely to be modified or debugged. It is preferable to write programs in a file using an editor and load the files into **saber** for testing and debugging.

2.1. Entering Input

All input to **saber**'s environment must consist of proper C expressions or statements. Every expression or statement must be terminated by a semi-colon. Each expression or statement is *evaluated* as soon as the user enters a semi-colon followed by a newline.

```
1 -> int i ;
2 -> i = 100 ;
(int) 100
3 ->
```

When expressions are evaluated, they produce a value which **saber** displays along with the type of the value. C statements do not produce a value, so **saber** displays the type void. For example, the while statement on line 8 below changes the value of the variable i, but the statement itself does not result in a value.

```
4 -> /* expressions result in a value */
5 -> i = (123 + 456) / 2 ;
(int) 289
6 ->
7 -> /* statements do not result in a value */
8 -> while( i < 1000) i++ ;
(void)
9 -> i;
(int) 1000
10 ->
```

2.2. Understanding the Input Prompts

Saber uses four prompts to indicate different input conditions. All prompts are preceded by the current line number.

The -> prompt is used when **saber** is expecting input from the user. If the user enters an incomplete expression or statement — usually because of a missing semi-colon or an incomplete block — the prompt changes to +> .

```
1 -> 123 +
2 +> 456 +
3 +> 789;
(int) 1368
4 ->
```

The *> prompt indicates **saber** is in the middle of reading a comment. The #> prompt indicates **saber** is reading the excluded portion of a #if**, #ifdef**, or #ifndef **preprocessor directive.**

2.3. Interrupting and Suspending Saber

If the user types a <Control-C> (interrupt signal) while entering input, **saber** will discard all input from the beginning of the statement or expression, even if the statement or expression spans more than one line.

```
1 -> 123 + 456 +
2 +> 789 +
3 +> ^C*** interrupted ***

3 ->
```

If the user types a <Control-Z> (suspend signal), **saber** will be suspended and the user will be placed in the shell. Upon resuming **saber**, the user will be returned to the exact location at which the <Control-Z> was entered. Any incomplete input will be preserved.

2.4. Executing Commands

Saber commands are implemented as C functions which can be called directly from the environment. The user executes a command by calling it as a function, and command options or switches are passed as arguments to the function.

```
1 -> load("-DDEBUG test.c") ;
Loading: -DDEBUG test.c
(int) 0
2 ->
```

Even if the command is called without any options, it must be followed by an open parentheses, a close parentheses, and a semi-colon.

```
2 -> load();
load: All loaded modules are up-to-date.
(int) 0
3 ->
```

Many of the commands accept a variable number of arguments, or variable types of arguments. For example, the argument to the `list` command can be a line number, which is specified by using an integer; a function, which is specified by using the name of a function; or a file, which is specified by using a string containing the name of the file.

```
1 -> list(10);
/* list lines starting a line 10 */
(void)
2 ->
3 -> list(main) ;
/* list lines starting at function 'main' */
(void)
4 ->
5 -> list("test.c") ;
/* list lines from the top of file 'test.c' */
(void)
6 ->
```

The `list` command can also be called without any arguments or with two arguments.

```
7 -> list() ;
/* list lines from the end of the previous listing */
(void)
8 ->
9 -> list(10,20) ;
/* list from line 10 to line 20 */
(void)
10 ->
```

Commands that accept a variable number of arguments should be called only from the environment or from within source code. These commands will not work properly if they are executed from within compiled code. The one exception to this rule is the command `stop`, which can be used to insert breakpoints into a program before it is compiled.

2.5. Defining Variables and Types

Variables and types can be defined and used directly in the environment at any time.

```
1 -> int i = 1234;
2 -> struct X { int a ; char *ptr; } entry ;
3 ->
```

The type and value of a variable can be displayed by evaluating it, which is done by entering the name of the variable followed by a semi-colon.

```
4 -> i ;
(int) 1234
5 ->
6 -> entry ;
(structX) =
{
 int a = 0;
 char *ptr= 0x0;
}
6 ->
```

2.6. Defining Functions

Functions can be defined directly in the environment. The user must specify the return type of the function when entering the definition. For example, the following definition of the function `add` does not specify the return type of the function.

```
/* missing return type before function name */
add(x, y) int x, y ;
{
   return x + y ;
}
```

If this definition occurs within a file, **saber** will treat it just as the compiler does — the function's return value will be set to the default type `int`. However, if the definition is entered directly in the environment, **saber** will assume the user wants to execute the function because the return type is missing.

```
1 -> add(x, y) int x, y ;
Error#277: 'add' is undefined.
2 ->
```

In order to define the function in the environment, the definition must be entered with a return type specified.

```
3 -> int add(x, y) int x, y ;
4 +> {
5 +>     return x + y ;
6 +> }
7 ->
```

2.7. Examining and Redoing Previous Input

All input to the environment can be displayed by using the `history` command.

```
1 -> (123 + 456) ;
(int) 579
2 -> int abc  ;
3 -> double xyz ;
4 -> history() ;
   1: (123 + 456) ;
   2: int abc  ;
   3: double xyz ;
   4: history() ;
(void)
5 ->
```

A previous line of input can be redone by using the notation `#N` , where N is the number of the input line to be redone. If N is a negative number, then the N'th previous line is redone. If the user enters `##` , the most recent line of input is redone (this is equivalent to entering `#-1`).

```
6 -> #1
7 -> (123 + 456) ;
(int) 579
8 -> ##
9 -> (123 + 456) ;
(int) 579
10 ->
```

When a line is redone, it is first echoed on the screen so that the user can modify the line before it is executed again.

2.8. Clearing Previous Input

Previous input to the environment can be cleared by unloading the environment.

```
1 -> int i = 99 ;
2 -> i ;
(int) 99
3 -> unload("environment")
(int) 0
4 -> i ;
Error#277: 'i' is undefined.
5 ->
```

2.9. Saving and Editing Previous Input

All input to the environment is saved to a temporary logfile. The logfile is used merely to record input; therefore, changing the logfile will not affect the state of **saber**'s environment. The name of the logfile can be displayed by calling the printopt command with the logfile option as the argument.

```
1 -> printopt("logfile") ;
logfile             "/tmp/saber.a05289"
--file containing a record of commands entered into the
environment.  For more information about option 'logfile',
enter 'help("options")'
(void)
2 ->
```

This file will be deleted when the user exits **saber**. In order to save the contents of the logfile, the file name display by printopt should be copied while **saber** is active or suspended. A permanent logfile can be specified by using the -f command line switch when starting **saber**.

The logfile should not be edited or modified while **saber** is executing. However, functions that have been defined in the environment can be edited with the edit command. The user will be prompted for a file in which to copy the text of the function.

```
1 -> int test() { return-1 ; }
2 -> edit(test) ;
Appending  test() (defined  in the environment)  to a  file.
Please specify a filename  (<Return> to abort): test.c
/* Function test() copied  to file "test.c" */
3 ->
```

Since the function test is not deleted from the environment, loading the file test.c will result in a function redefinition error. The user must either unload the environment (thereby unloading the definition of test and any other functions or variables) or change the name of test in the file test.c.

CHAPTER 3: Break Levels

When a **saber** session is begun, the user is placed at the *top* level of the environment. When execution is interrupted by the user or by a violation in the user's program, a *break level* of the environment is generated. The commands associated with the break levels are:

```
cont        — continue execution from a break level
reset       — return to a previous break level

where       — display the execution stack
up          — move up the execution stack
down        — move down the execution stack

whereami    — display execution and scope locations
```

3.1. Generating Break Levels

A break level is generated when execution is interrupted due to one of the following reasons: the user chooses to stop execution at an error or warning, the user enters a <Control-C> (interrupt signal) during execution, a breakpoint is encountered while executing the program, or a signal occurs that is not being caught by the program.

In the example below, the user has interrupted execution of the `while` statement by entering a <Control-C>, resulting in the generation of break level #1 .

```
1 -> int i = 0 ;
2 ->
3 -> while( i < 9999) i++ ;
^C  ** interrupting execution **

Stopped in top level, line 3.
   2:
*  3: while( i < 9999) i++ ;
   *** signal #2 (SIGINT) occurred ***
(break 1) 4 ->
```

Break levels preserve the location where execution was interrupted. If execution is interrupted again while at the break level, a new break level is generated.

```
(break 1) 4 -> int j ;
(break 1) 5 -> while( j < 12345) j++ ;
^C **interrupting execution **

Stopped in break level #1, line 5
   4: int j ;
*  5: while( j < 12345) j++ ;
   ***signal #2 (SIGINT) occurred ***
(break 2) 6 ->
```

The prompt identifies the new break level as number 2. Each break level is numbered according to how many levels deep it is from the top level.

3.2. Continuing from a Break Level

Execution can be continued from a break level by using the cont command. Execution will resume from the point at which it was interrupted.

If execution was interrupted because the user entered a <Control-C> or because a breakpoint was encountered, cont should be called without an argument. A <Control-D> can be used as a shortcut for calling cont without an argument.

In the two examples shown above, both break level 1 and break level 2 were generated when the user entered a <Control-C>. Therefore, execution can be resumed from both break levels by entering <Control-D> or calling cont without an argument.

```
(break 2) 7 -> <Control-D>

Resuming execution.
(void)
(break 1) 9 -> /* 1st while loop done: j equals 12345*/
(break 1) 10 -> j ;
(int) 12345
(break 1) 11 ->
(break 1) 12 -> cont() ;

Resuming execution.
(void)
13 -> /* 2nd while loop done: i equals 9999*/
14 -> i ;
(int) 9999
15 ->
```

If the break level was generated because the user selected the break option at a violation, the message displayed when the break level was generated specifies what value can be supplied as the argument to

cont.

If an argument is supplied to cont, its value will be substituted for the expression containing the violation. If no argument is supplied to cont (or a <Control-D> is used), the value of the original expression will be used. If the violation is an error, the user must provide a substitute value to safely continue execution.

In the example below, the user attempts to dereference and use a null pointer. The last line of the error message indicates that in order to continue execution, cont should be called with an integer (int).

```
1 -> int i, *ptr = 0 ;
2 ->
3 -> i = *ptr ;
Error#67: Dereferencing a pointer that is out of bounds.
   Pointer = 0x0, low bound = 0x0, high bound = 0x0.
To continue, use 'cont(int)'.
(break 1) 4 ->
```

The value supplied to cont will be used instead of dereferencing the variable ptr. When execution resumes and the assignment is completed, the value supplied to cont is stored in i.

```
(break 1) 5 -> cont(123) ;
(int) 123
6 ->
7 -> i ;
(int) 123
8 ->
```

3.3. Reseting from a Break Level

The reset command resets from a break level and returns to a previous break level or to the top level without continuing execution.

If reset is called without arguments, it resets to the top level.

```
(break 3) 50 -> reset() ;

Resetting to top level.
51 ->
```

If reset is called with a positive argument, it will treat the argument as the number of the break level to which to return. If the argument is negative, reset will back up the specified number of break levels.

```
(break 10) 100 -> reset(6) ;

Resetting to break level 6.
/* execution location listing not shown */
(break 6) 101 -> reset(-4) ;

Resetting to break level 2.
/* execution location listing not shown */
(break 2) 102 ->
```

3.4. Displaying the Execution Stack

The *execution stack* consists of all the functions and previous break levels which are in the middle of execution. The `where` command displays the execution stack, starting from the location where execution has stopped in the current break level.

```
(break 1) 11 -> where() ;

break level #1, line 11, caused by error #66 (Division)
test(num = (int) 1, size = (int) 100) at "main.c", line 44
top level, line 3
(void)
(break 1) 12 ->
```

In the example above, the user is at line 11 of break level 1, which was created when an error was detected at line 44 of function `main`, which was called from line 3 of the top level environment.

3.5. Execution and Scope Location

Saber maintains two concepts of location while at a break level. The *execution* location is the point at which execution has stopped. The *scope* location is the point at which all variables, types, and macros are scoped.

When a break level is generated, both the execution and the scope location are identical — all variables, types, and macros are scoped to the point at which execution was interrupted. The scope location can be changed to another function on the execution stack with the `up` or `down` commands.

The `whereami` command will display the execution location and the scope location if it is different from the execution location. In the example below, `whereami` does not print the scope location because it has not been changed from the execution location.

```
(break 1) 9 -> whereami() ;
Execution stopped at "environment", line 3.
   2:
*  3: while( i < 1000) i++ ;
   ***signal #2 (SIGINT) occurred ***
   4:
(void)
(break 1) 10 ->
```

CHAPTER 4: Errors and Warnings

Saber performs both static and dynamic error checking. Static error checking detects syntactic violations which occur in user input or within source files. Dynamic error checking detects run-time violations which occur during the execution of a program.

If the violation is serious, **saber** will report it as an *error* , which requires the user to remedy the situation. If the violation is minor, **saber** will report it as a *warning* from which the user may continue. The commands associated with errors and warnings are:

```
suppress     — suppress reporting of a warning or error
unsuppress   — reactivate reporting of a warning or error
```

To get a listing of all the errors and warnings, call the help command with the argument "errors."

4.1. Message Layout

Both error and warning messages are displayed in a similar format. The example below contains a sample warning message.

```
"test.c",line 11, test(): Dereference (Error#67)
   10: {
*  11:  return *++ptr ;
   12: }
Dereferencing a pointer that is out of bounds.
Pointer = 0x4, low bound = 0x0, high bound = 0x0.
Options: break/quit/edit/reload [b] ?
```

The first line of the message displays the location of the violation followed by the number of the violation. The next three lines list the source code at the location of the violation. The line containing the violation is preceded by a star. A description of the error or warning follows the source code listing.

The last line of the message contains a set of options from which the user can select a specific action. The default option, which is the option that is selected if the user responds with a <Space> or a <Return>, is enclosed within the square brackets. The options available to the user depend on the context and severity of the violation.

4.2. Warnings

Warnings notify the user of non-fatal or non-critical violations from which execution may continue. The example below displays a warning that occurred while loading the file test.c.

```
"test.c",line 10, main(): Unused automatic (Warning #286)
   9: {
*  10: int x ;
   11: int y ;
Automatic variable 'x' was not used.
Options: continue/suppress/terse/quit/abort/edit/reload[c] ?
```

The user is given seven options to choose from, with the default option set to continue. Selecting `` c´´ will continue loading the file.

Selecting `` s´´ will suppress reporting of the warning. A subsequent options line appears, from which the user must select the range over which the warning should be suppressed.

Selecting `` t´´ will set the load_terse option for the remainder of the loading process. This will prevent the reporting of any warnings while the remaining files are being loaded.

Selecting `` q´´ will quit loading this file.

Selecting `` a´´ will unload this file and abort the loading process so that other files passed in the same call to load will not be loaded.

Selecting `` e´´ will start the editor at the location of the violation.

Selecting `` r´´ will reload the file if it has been modified.

4.3. Errors

Errors notify the user of violations from which the user is advised not to continue. The example below displays a error that occurred while executing the function test.

```
"test.c",line 14, test(): Formal variable  (Error#62)
   14:  x = 100;
*  15:  y = 100;
   16:  z = 100;

Formal variable  'y' cannot be used as an lvalue.
   Function called  with too few  arguments.
Options: break/edit/quit [b] ?
```

The user is given three options, with the default option set to **break**. Since errors report serious violations, both **continue** and **suppress** are excluded from the options line.

Selecting `` e´´ will start the editor at the location of the violation.

Selecting `` b´´ will interrupt execution at the location of the error and generate a break level.

Selecting `` q´´ will quit execution of the function and return to the environment.

4.4. Suppressing Warnings and Errors

The reporting of both warnings and errors can be suppressed by using the **suppress** command. The different ways to use the **suppress** command that are discussed in this section are also available when the user selects **suppress** when a warning is reported.

If the **suppress** command is called with the number of the violation (displayed on the first line of every message), the violation will be suppressed everywhere.

```
1 -> suppress(240);
Violation #240 (Too many  macro  arguments) suppressed everywhere.
(int) 0
2 ->
```

This can also be accomplished by selecting `` e´´ for **everywhere** on the suppress options line.

An optional second argument can be used to specify the range over which the reporting is to be suppressed. If the argument is a file name, the reporting is suppressed throughout the file, but nowhere else. If the argument is the name of a function, the reporting is suppressed within the specified function only. If the argument is a line number, the reporting is suppressed only at that line in the file.

```
3 -> suppress(345,"test.c") ;
Violation #345 (Member redefined) suppressed
   in file: test.c.
(int) 0
4 ->
5 -> suppress(346,main) ;
Violation #346 (Tag name redefined) suppressed
   in function: main().
(int) 0
6 ->
7 -> suppress(347,10) ;
Violation #347 (Prototype usage mismatch) suppressed
   at line 10, file: test.c.
(int) 0
8 ->
```

The same violation can be simultaneous suppressed over different ranges.

If `suppress` is called without arguments, it will list all suppressed violations and prompt the user to save the listing to a file.

```
9 -> suppress();
Num   Description              Range
240   Too many macro arguments    everywhere
345   Member redefined            file: test.c
346   Tag name redefined          function: main()
347   Prototype usage mismatch    line 10, file: test.c

Save suppressionsto a file [n] ? n
(int) 0
10 ->
```

If the suppression list is saved to a file, it can be placed in the startup file or in a script so that unwanted warnings will be suppressed prior to loading and executing programs.

4.5. Unsuppressing Violations

Reporting of violations can be reactivated by using the `unsuppress` command. If the violation was suppressed over a particular range, `unsuppress` must be called with the range as the second argument.

```
11 -> unsuppress(288);
unsuppress:Violation #288 is not suppressed everywhere.
(int) -1
12 ->
13 -> unsuppress(288,10) ;
Violation #288 unsuppressed.
(int) 0
14 ->
```

If unsuppressis called without arguments, it will prompt the user to reactivate each suppressed violation, one at a time.

CHAPTER 5: Execution

Saber supports two modes of execution: direct execution of functions from the environment and controlled execution of a program by calling the function `main`. In addition, execution can be stepped when continuing from a break level.

The commands related to execution are:

```
run       — initialize variables and execute main
start     — execute main without initializing variables
reinit    — reinitialize all global variables.

step      — step execution by expression, enter functions
next      — step execution by line, do not enter functions
stepout   — continue execution until function returns
```

5.1. Incremental Execution

Saber allows functions to be used as soon as they have been defined. A function can be defined by entering it directly in the environment, by loading a file containing the defining instance, or by linking from a library.

Saber allows functions to be executed even if they contain references to undefined functions or variables. The function will be executed as long as it does not actually encounter the undefined reference. If the function's execution path avoids the unresolved references altogether, execution will not be interrupted.

In the example below, the user declares two external variables, `x` and `y`, and one library function, `printf`. Both variables and the library function are used in function `test`.

```
1 -> extern int x, y, printf() ;
2 ->
3 -> void test()
4 +> {
5 +>    printf("x = %d\n", x) ;
6 +>    printf("y = %d\n", y) ;
7 +> }
8 ->
```

The declarations on the first line of the example merely specify information about the types of the variables

and function; External declarations do not define variables or functions.

When test is executed, all attached libraries are searched to satisfy undefined variables and functions. In the current example, only printf can be loaded from an attached library; both x and y remain undefined.

```
8 -> test() ;
Linking from /lib/libc.a ...Linking completed.
Warning #38: Calling function test() which contains
   unresolved references.
Options: break/continue/quit/edit/suppress [b] ?b
To continue, use 'cont()' or <Control-D>.
(break 1) 9 ->
```

Since the violation is only a warning, the user can continue execution or suppress the warning. In this example, the user chose to interrupt execution and generate a break level.

One option at this point is to define the variables and resume execution. The user defines the variable x and continues execution by exiting the break level. Execution continues until the undefined variable y is encountered.

```
(break 1) 9 -> int x ;
(break 1) 10 -> <Control-D>
x = 0

"environment", line 6, test(): Undefined variable. (Error#60)
   5:  printf("x = %d\n", x) ;
*  6:  printf("y = %d\n", y) ;
   7: }
Undefined variable: 'y'.
To continue, use 'cont(int)'.
If you define the variable, use 'cont()' or <Control-D> to retry.
(break 1) 11 ->
```

When execution reached the undefined variable y, an error message is reported. The end of the message indicates that cont can be called with an integer value, or that y can be defined and execution can be continued.

The user continues execution by supplying the value -99 to the cont command. This value is used instead of the value in y.

```
(break 1) 16 -> cont(-99) ;
y = -99
(void)
17 ->
```

5.2. Incremental Execution of Compiled Code

If a compiled function containing references to undefined variables is executed, a warning is produced when **saber** is about to execute the function. If execution is continued without defining all unresolved references, **saber** will **NOT** generate addition warning or error messages, even if the compiled code is using undefined variables and functions.

Always avoid executing compiled functions which contain references to undefined variables; the most likely result will be a segmentation fault while the function is executing.

To insure that a compiled function is fully resolved, call the link command with the name of the function as the argument. Link will search all attached libraries for any unresolved references and it will list the names of any references it can not resolve.

5.3. Executing a program

The run and start commands let users execute programs with command line arguments. Run initializes all global variables (by calling reinit), processes command line arguments, calls main, and closes all input/output buffers and file descriptors after main returns. Run will also reset to the top level if the current execution point is at a break level.

Start is similar to run, except that it does not reinitialize variables, nor does it close input/output buffers and file descriptors when execution is completed. (Note: input/output buffers and file descriptors will always be closed if the program calls the library function exit to terminate execution.) Also, start does not reset to the top level of the environment.

The example below shows how command line arguments can be specified with the run command.

```
1 -> int main( argc, argv)
2 +> int argc ; char *argv[ ] ;
3 +> {
4 +>  printf("argc = %d, argv[ 1] = %s\n", argc, argv[ 1]) ;
5 +>  exit(0) ;
5 +> }
6 ->
7 -> run("user") ;

Executing: a.out user
Linking from /lib/libc.a ...Linking completed.
argv = 2, argv[ 1] = user
Program exiting with return status= 0
Resetting to top level.
8 ->
```

Both run and start construct arguments for main from the command line argument passed to them. If

this argument is missing, command line arguments from the most recent call to either command are reused. The string pointed to by argv[0] is determined by the value of the program_name option.

The command line may contain the input and output redirection symbols recognized by the shell. Pipes are not allowed in the command line. The example below indicates how to call a program that gets its input from the file raw_data .

```
-> run("< raw_data") ;
Executing: a.out < raw_data
```

If the auto_reload option is set, then run will automatically reload any outdated modules before beginning execution. If run is called when **saber** is executing in a windowing environment, a new window will be opened to handle input and output for the program. This can be disabled by unsetting the win_io option.

5.4. Stepping Execution

Execution can be stepped by statement or by line from a break level. The step command will continue execution until the next statement is reached. If execution is stopped on a line containing multiple statements, step will execute the next statement only. The example below shows execution stopped at a source line containing three statements. When step is used, only the first statement is executed, and execution is stopped at the second statement on the same line.

```
Stopped in "environment", line 11, foo().
   10:
*B11: puts("x=1") ; puts("y=2") ; puts("z=3") ;
   12: puts("done") ;
(break 1) 20 ->
(break 1) 21 -> step() ;
x=1
Stopped in "environment", line 11, foo().
*B11: puts("x=1") ; puts("y=2") ; puts("z=3") ;
(break 1) 22 ->
```

The next command continues execution until a new source code line is reached. When next is called on the example used above, the remaining statements on the line are executed.

```
(break 2) 22 -> next() ;
y=2
z=3
Stopped in "environment", line 12, foo().
*B12: puts("done") ;
(break 2) 23 ->
```

Step will enter function calls, while next does not. Any breakpoints set within the function will be triggered even if the user does not enter into the function. If a numerical argument is passed to step or next, it is taken as the number of statements or lines to execute before stopping.

The stepout command will continue execution until the current function returns. This is particularly useful when the user accidentally enters a function by using step instead of next.

If step or next is used at a line calling a compiled function, execution will not stop until the next line of source code is reached. If step, next, or stepout is issued from a break level within a compiled function, execution will not stop until the next line of source code is encountered.

The number of source lines listed when stepping execution is determined by the value of the src_step option.

CHAPTER 6: Loading and Unloading Files

Files can be loaded or included into **saber**. Included files are processed directly by the environment, as if the user had entered the contents of the file. Loaded files are stored internally as modules which can be subsequently unloaded or reloaded. The file commands are:

```
#include      — include contents of a source file

load          — load or reload files
unload        — unload files

link          — load library functions
```

6.1. Including Files

Files can be included into **saber** by using the #include **preprocessor directive. Including a file is identical to entering the contents of the file directly to the environment; therefore, only files containing C code can be included.**

The example below shows a script file in which the user has defines a number of macros and types.

```
% cat script

/* File script: contains macros and types for saber */
#define S step() ;
#define N next() ;
#define W where() ;

typedef int *pointer;
structloc { int x, y ; } ;
```

Once the file has been included into **saber**, the macros and types defined within it can be used in the environment.

```
1 -> #include "script"
2 ->
3 -> pointer ip = 0 ;
4 -> ip ;
(int *) 0x0
5 ->
6 -> W

top level, line 6.
(void)
7 ->
```

Once a file has been included, it can be unloaded only by unloading the entire environment module. Therefore, files that are included should not contain code that will be debugged or modified. Included files should be used to execute startup commands, declare commonly used types, or define macros.

6.2. Loading Files

The load command loads source code, compiled code, and library files into **saber** and creates a *module* with the same name as the loaded file. The module represents the file which was loaded and all other files included within it (e.g. a C source file and the header files it includes).

```
1 -> load("main.c") ;
Loading: /usr/saber/main.c
(int) 0
2 ->
```

Load looks for the file in the directories specified by the path option, or in the current working directory in the option is not set. In the example below, the user sets the path option so that load will look for files in the test directory first and then the current directory (specified by the `` .´´ character).

```
1 -> setopt("path", "/usr/saber/test.") ;
(void)
2 ->
3 -> load("abc.c") ;
Loading: /usr/saber/test/abc.c
(int) 0
4 ->
```

If the path option is not set, the file abc.c can also be loaded by specifying the complete path to load.

Information on all the switches and options for the load command can be found in the *Reference Manual*.

6.3. Reloading Files

If load is called with the name of a previously loaded file, the file will be reloaded if it has been modified since it was loaded. If load is called without any arguments, it will check all loaded files to see if any are outdated and require reloading.

```
4 -> load("abc.c") ;
/usr/saber/test/abc.cis up to date.
(int) 0
5 ->
6 -> load() ;
All loaded files are up to date.
(int) 0
7 ->
```

A source code file is reloaded if the source file itself or any of the header files it includes have been modified since the file was loaded. An object file is reloaded if the file has been recompiled since it was loaded.

If the auto_compile option is set, an object code file will be recompiled if the source file has been modified. If the option is unset, the user will be prompted to recompile the file. Load will attempt to use **make** to recompile the file, or, if a makefile is not found, the file will be compiled by calling the C compiler with the switches specified in the load_options option.

6.4. Swapping Files

If load is called with the name of a source file for which the object file is already loaded (or vice-versa), the user will be prompted to replace the previously loaded file with the newly specified file. If the auto_replace option is set, the file is replaced automatically without prompting the user.

6.5. Unloading Files

Any loaded file can be unloaded by calling the unload command with the name of the file. If unload is called without any arguments, it will prompt the user to unload all modules defined by the user, unload all modules defined in libraries, unload everything, or unload modules one by one.

```
13 -> unload("abc.c") ;
Unloading: /usr/saber/test/abc.c
(int) 0
14 ->
```

If a library is unloaded, all the modules that have been linked from the library will also be unloaded.

6.6. Loading Compiled Code with Debugging Information

Compiled files with debugging information (i.e. compiled with the -g compiler switch) should be used whenever possible. Only the -w and the -G switches are meaningful when loading compiled files.

The listing below shows the beginning of the file sample.c, which will be compiled and loaded into **saber**.

```
/* file: sample.c */
#define BUFLEN 512
char buffer[BUFLEN];

structentry {
   char *task;
   structentry *next ;
} ;

structentry head, test() ;
```

When compiled files are loaded, all variables, functions, and types will be visible in the environment. Only macros and other preprocessor information is not available.

For instance, if sample.o is loaded, the user can examine the variable head but not the macro BUFLEN.

```
1 -> load("sample.o")  ;
Loading:  sample.o
(int) 0
2 ->
3 -> head  ;
(struct entry) =
{
 char *task = 0x0;
 struct entry *next = 0x0;
}
4 ->
5 -> BUFLEN;
Error#277: 'BUFLEN' is undefined.
6 ->
```

Since the file is compiled with debugging information, the source code file can be listed, edited, and displayed when a violation occurs.

6.7. Loading Compiled Code without Debugging Information

If an object code file without debugging information is loaded, no information about the types of variables or functions is available. For example, if the file `sample.o` is compiled without debugging information and loaded, the type `structentry` could not be used in the environment.

```
1 -> load("sample.o")  ;
Loading:  sample.o
(int) 0
2 ->
3 -> struct entry xyz  ;
Error#281: Tag  'entry' is currently undefined.
4 ->
```

If the type for a variable defined in compiled code is unavailable, **saber** assigns the generic type <*data*> to the variable. Before variables of type <*data*> can be used, a type must be assigned by the user or by another file.

```
5 -> head ;
Error#283: Variable 'head' has undefined type <data>.
6 ->
7 -> structentry { char *task; structentry *next ; } ;
8 -> structentry head ;
9 -> head ;
(structentry) =
{
 char *task= 0x0;
 structentry *next = 0x0;
}
10 ->
```

It is important that the type assigned to compiled data matches the type used when originally defining the data.

If the type for a function defined in compiled code is unavailable, **saber** assigns the generic type <*text*> to the function. If the function is used before its type is declared, it will be given a return type of int.

6.8. Attaching Libraries

If load is called with the name of a library, it *attaches* the library, thereby making the contents of the library available to **saber**. All loaded libraries must be processed by **ar** and **ranlib** before they are loaded.

If the library resides in the directories /lib, /usr/lib, or /usr/local/lib, the notation -lx can be used, where x is an abbreviation for /lib/libx.a, /usr/lib/libx.a, or /usr/local/lib/libx.a, (The syntax for this option is the same as for the -l switch of the **ld** and **cc** commands.)

```
1 -> load("-lm") ;
Attaching: /usr/lib/libm.a
(int) 0
2 ->
```

The individual functions within the library are dynamically linked when they are needed to satisfy an undefined reference during execution.

```
3 -> double sqrt() ;
4 ->
5 -> sqrt(16.0);
Linking from /usr/lib/libm.a...Linkingcompleted.
(double) 4.00000e+01
6 ->
7 -> sqrt(25.0);
(double) 5.00000e+01
8 ->
```

Once a library function has been loaded into the environment, it does not have to be loaded again for the remainder of the session.

Libraries may be detached by calling the unload function with the name of the library. The library and all modules that have been linked from the library will be unloaded.

6.9. Using the link command

The link command will search all attached libraries to satisfy all undefined references. Link lists all undefined references which cannot be found in any of the attached libraries.

```
1 -> extern int i, j ;
2 -> int *ip = &i ;
3 ->
4 -> link() ;
Undefined variables:
   int i
(int) 1
5 ->
```

Note that even though the variable j is undefined, it is not listed by link because there are no references to it.

CHAPTER 7: Handling Files

Saber provides several commands to display and modify files.

```
contents      — display information about loaded files

list          — display source code lines

edit          — edit a file
```

7.1. Listing Loaded Files

The contents command lists the names of all loaded modules, or the contents of one individual module. If contents is called without an argument, it prompts the user for which subset (or all) of the loaded modules to list. In the example below, the user opted to list all loaded files.

```
5 -> contents() ;
Options: user/library/all/quit[u] ?a
saber: object code
environment: source code
/lib/libc.a: archived library
(void)
6 ->
```

The files saber and environment are loaded when **saber** starts up. The file /lib/libc.a is loaded as a result of a load command in the system startup file.

If contents is called with the name of a loaded file as an argument, it will list the variables, functions and types defined in the file. In the example below, the user lists the contents of the environment.

```
1 -> struct S { int i, j ; } info ;
2 -> int xyz ;
3 -> void test() { return ; }
4 ->
5 -> contents("environment") ;
Options: functions/variables/types/all [f] ?a

Contents: environment: source code
struct S { ... } ;
struct S info ;
int xyz ;
void test() ;
(void)
6 ->
```

If `contents` is called with the name of a library as its argument, it will list all the modules archived in the library.

7.2. Listing Source Code

The `list` command displays source code lines. If no argument is given to `list`, listing begins from the current *list location*. The list location is set to either the first line of a file when it is loaded, the line where execution was interrupted when entering a break level, or the last line displayed by a previous `list` command.

```
-> list() ;
/* begin listing at the current list location */
```

If a line number is supplied as the argument, listing will begin at the specified line number in the current file. If the line number is negative, listing will begin from the current list location minus the specified number of lines. If both a starting and ending line number are specified, the source code between the two lines will be displayed.

```
-> list(10);
/* list lines starting a line 10 */

-> list(-5);
/* list from list location - 5 */

-> list(10,30) ;
/* list from line 10 to line 30 */
```

If the argument to `list` is a function, listing will begin at the beginning of the function. If the argument is a filename, listing will begin at the beginning of the file.

List displays the number of lines specified by the page_list option. If more lines can be displayed, list prompts the user for more. If **saber** is executing in a windowing system, the user can specify that the output of listbe sent to a separate window by setting the win_list option.

The source line corresponding to the current execution location will have a * preceding the line number. Lines at which breakpoints are set are displayed with a Bpreceding the line. Signals and errors are noted after the source line at which they occurred.

Actions are listed in full above the line on which they are set. Each line of the action is preceded by the notation <An> , where n represents the debugging reference number. Tags and tagactions are displayed after the source line that triggered them. Each line of a tagaction is preceded by the notation <Tn>, where n represent the debugging reference number.

7.3. Editing Files

The edit command allows users to edit source files. If edit is called without an argument, it uses the list location to determine which file to edit.

If the argument is a line number, editing will begin at that line in the file specified by the current list location. If the argument is the name of a file, the editor will begin on the first line of the specified file. If edit is called with the name of a function, the editor will begin on the first line of the function.

The edit command will use the editor specified by the edit_name option, which is set to the environment variable **EDITOR**, or the editor **vi** if the environment variable does not exist.

The edit_jobs option sets the maximum number of new edit jobs that **saber** should start. If the editor is **emacs**, the value will be set to 1. If the editor is **vi**, the value will be set to 2 on a windowing system or to 5 on a non-windowing system.

Saber keeps track of the files that are currently open so that windows can be reused whenever possible. The jobs command lists the currently open files. If a request is issued to edit a new file, and there are already edit_jobs number of jobs active, **saber** will ask one of the existing edit jobs to switch to the new file. **Saber** uses a 'least recently used' algorithm to determine which file to switch out of.

For example, users who prefer to have only one **emacs** job active at a time can set the edit_jobs option to 1. **Saber** will reuse the one emacs job for every new request to edit. The 'switch to a new file' request in emacs will open a new buffer for each new file that is edited.

Saber uses the string specified by the edit_options option to determine the command line arguments for the editor. Normally, this option is set to a sequence such as "+%d %s" to indicate that the editor should start at line "%d", in file "%s". **Saber** will fill in these two fields with the starting line number and filename. Versions of editors that create their own window must be told not to do so (with gnuemacs, the -nx or -nw switch).

The edit_line_file option describes the editor command sequence used to move to another file at another line. The sequences "%d" and "%s" must be present somewhere, and **saber** will fill them in with the proper line number and filename. This is how **saber** tells the editor to switch files if it cannot start another editing job.

The edit_query option describes the editor command sequence to report the current file name. This command sequence must force the editor to echo a string in the following format: '%s"%s"%s\n', where the '%s' inside the quotation marks represents the current filename. The first '%s' string and the last '%s' string may contain garbage. Each request to edit and the jobs command cause **saber** to send this query string the each active editing job. If this option is not set, the <Control-Z> feature on a windowing system will not work properly.

Saber attempts to initialize all of these options properly for the **vi** and **gnuemacs** editors so that the user does not have to set their values.

When moving between the editor and **saber**, the user must leave the editor is a state such that the command specified by the edit_line_file and the edit_query options will be accepted. When using **vi**, the editor should not be left in the ex mode. When using **gnuemacs**, the editor should not be left in the minibuffer. **Saber** will report a 'query timed out' error if sending the edit_query string does not produce the correct response. If this should happen, it is usually sufficient to change the state of the editor and retry the query.

7.4. Editing and Windows

The win_edit option determines whether **saber** will place editing jobs in separate windows. If the option is unset, **saber** will be suspended while an editing job is active. If the win_edit option is set, edit jobs will be placed in a separate window and **saber** will remain active. The win_edit_opts can be set to specify options when the edit windows are created.

On a non-windowing system (or if the win_edit option is not set), entering a <Control-Z> to the editor will suspend the edit job and return control to **saber**. If the edit was started on an option line or a previously loaded file is being edited, a prompt will appear asking if the file should be reloaded. To avoid this prompt, unset the auto_reload option. In a manner similar to the **csh**, the fg command can be used to resume suspended editing jobs. Loading successive files will reuse previously suspended editing jobs if an edit is requested (subject to the edit_jobs option).

On a windowing system, **saber** is not suspended while editing. Entering a <Control-Z> in the edit window does not suspend the edit job, but rather tells **saber** to load (or reload) the current file. If **saber** was waiting on an option line, it would be as if a 'r' (for reload) was entered. If **saber** is waiting for user input at a break level, (including the top level of the environment) it would be as if the command 'load("filename");' was entered. The file should be saved before pressing <Control-Z>.

A special exception occurs if the file being edited does not end in a '.c' extension, in which case the command 'load()' is executed.

If an error or warning is detected while the file is being reloaded, the edit job is updated automatically to the location of the violation.

7.5. Editing in the Environment

Saber provides a convenient way to edit statements in the environment. If a syntax violation in the environment extends over one line, the user is prompted to edit the lines of input containing the violation. If the user selects the edit option, the text of the violation will be copied to a temporary file, which is then loaded into the editor.

After correcting the input error and saving the file, the user can resubmit the contents of the editor by entering a <Control-Z>. **Saber** will echo each line of the file as it is read. If another violation occurs, the user is placed back in the temporary file. If the user quits the editor instead of entering <Control-Z>, the contents of the editor will not be submitted.

The win_redo option, if set, indicates that a new editor should be started on this temporary file. If the option is unset, a previous editing job is reused if possible.

CHAPTER 8: Debugging Commands

The debugging commands available with **saber** fall into two categories: commands that execute when a certain location is reached, and commands that execute when a certain address is modified. The debugging commands are:

stop	— stop execution at a line (breakpoint)
action	— execute an action at a line or everywhere
tag	— stop execution when address is modified
tagaction	— execute an action when address is modified
trace	— trace program execution
status	— list all breakpoints, tags, and actions
delete	— delete a breakpoint, tag, or action

8.1. Setting Breakpoints

The stop command is used to interrupt execution at a specific location, otherwise known as setting a *breakpoint* . The location is specified by the argument supplied to stop.

If the argument is a line number, the breakpoint will be placed at the specified line number. The line number should refer to a source line within a function at the current list location. The list location is set to either the first line of a file when it is loaded, the line where execution was interrupted when the user entered a break level, or the last line displayed by a previous list command.

```
5 -> list('test.c') ;
   1: int i ;
   2:
   3: foo(x)
   5: int x ;
   4: {
   6:  if( (x % 2) == 1 ) x++ ;
   7:  i = x/2 ;
   8:  returnx + i ;
   9: }
(void)
6 -> stop(7) ;
Breakpoint #1 set at "test.c", line 7, foo().
(int) 1
7 ->
```

If stop is called with a file name and a line number, the breakpoint is set at that line in the specified file. If the first argument to stop is the name of a function, the breakpoint will be placed on the first line of the function.

If stop is called without an argument, it will immediately interrupt execution and generate a new break level. This allows users to create conditional breakpoints by combining stop with other debugging commands, such as action or tagaction. Also, users can set permanent breakpoints by inserting calls to stop within the source code of functions.

8.2. Setting Conditional Breakpoints

The action command performs a user-specified action. The location for the action is specified in the same manner as it is to the stop command.

If action is called without any location argument, it will perform the specified action at every executed statement. This allows users to set actions which are constantly monitoring the execution of a program. In order to avoid spurious messages, statements that are executed directly by the user or by other debugging commands are not monitored.

The action can consist of one or more C statements. If more than one statement is necessary, the entire action must be placed within a block.

The example below sets a multi-statement action on line 8 of function foo. The action uses the global variable i.

```
8 -> action(8) ;
Setting action at "test.c", line 8, foo().
Use braces for multiple statements. <Ctrl-D> to quit,
   <Ctrl-C> to redo.
action -> {
action +>    printf("i = %d\n", i) ;
action +>
action +>    if( i == 0 ) stop() ;
action +> }
Action #1 set.
(int) 1
9 ->
```

In addition to printing the value of i, the action will also interrupt execution with the stop command if i equals zero.

In the example below, foo is called with the value 10. Since i does not equal zero, only the call to printf is executed.

```
10 -> foo(10) ;
i = 5
(int) 11
11 ->
```

In the next example, foo is called with the value 0. Since i equals zero, the call to stop will be executed, and execution will be interrupted.

```
18 -> foo(0) ;
i = 0
Stopped in action #1, line 4, set in
foo(x = (int) 0) at "test.c", line 8
   7:  i = x/2 ;
<A1>    1: {
<A1>    2:  printf("i = %d, i) ;
<A1>    3:
*<A1>   4:  if( i == 0 ) stop() ;
<A1>    5: }
    8:  returnx + i ;
    9: }
```

8.3. Setting Watchpoints

The tag command interrupts execution when a specified address is modified, which is also known as setting a *watchpoint*. The addresses of global variables, allocated data, formal parameters, and automatic variables can be tagged.

Execution will not be interrupted if the tagged address is modified in compiled code. To avoid spurious messages, the tag will not trigger if the tagged address is modified by the user in the environment.

The argument to tag specifies the address to be watched. The example below tags the variable abc by calling tag with the address of the variable.

```
1 -> int abc ;
2 ->
3 -> tag(&abc) ;
Tag #1 set on 4 bytes beginning at address 0xd9616.
(int) 1
4 ->
```

The number of bytes tagged equals the size of the type of data being tagged. In the example above, four bytes were tagged because the type of abc is int, and the size of an int is four bytes.

This default behavior can be changed by providing a second argument to tag which indicates the number of bytes to tag. In the example below, the user tags only one byte of the variable xyz.

```
5 -> int xyz ;
6 ->
7 -> tag(&xyz, 1) ;
Tag #2 set on 1 bytes beginning at address 0xd9620.
(int) 2
```

8.4. Setting Conditional Watchpoints

The tagaction command allows users to specify an action which will be performed when an address is modified. The address and the number of bytes tagged is specified in the same manner as tag. The action is specified in the same manner as action.

The example below shows how to set a tagaction which will print the value of the variable ctr whenever it is modified.

```
1 -> int ctr ;
2 ->
3 -> tagaction(&ctr) ;
Use braces for multiple statements. <Ctrl-D> to quit,
   <Ctrl-C> to redo.
tag action -> printf("ctr = %d\n", ctr) ;
Tag #1 set on 4 bytes beginning at address 0xd962a.
(int) 1
```

Since the action does not contain a call to **stop**, execution will not be interrupted when **ctr** is modified.

8.5. Tracing Program Execution

The **trace** command displays each source code line as it is executed. Each line it displayed only once, regardless of the number of statements on the line.

Trace is called with a value between 0 and 5. The effect of each value is shown below:

0: Turn off tracing.
1: Print function, line number for each line executed.
2: Print file, function, line number for each line executed.
3: Print the source line as it is being executed.
4: Levels 1 and 3 combined.
5: Levels 2 and 3 combined.

Calling **trace** without an argument will print the current tracing level.

Compiled functions and source code executed within debugging actions cannot be traced.

8.6. Listing and Deleting Debugging Items

The **status** command lists all active debugging items by number.

```
10 -> status() ;
(1) Breakpoint: file 'test.c', function foo(), line 6.
(2) Breakpoint: file 'test.c', function main(), line 27.
(3) Tag: addresses 0xd9616 - 0xd9619 (location 'x').
(void)
11 ->
```

A debugging item can be deleted by calling the **delete** command with the reference number listed by **status**

```
12 -> delete(1) ;
Breakpoint number #1 deleted.
(void)
13 ->
```

If delete is called without arguments, the user will be asked for confirmation to delete all debugging items.

CHAPTER 9: Information Commands

Saber's information commands display information on commands, variables, and functions.

```
help      — display information about commands

whatis    — display all uses of a name
xref      — cross-reference a function or variable
unres     — list unresolved references

dump      — display information about an address
badptr    — display information about bad pointers
```

9.1. Getting Information on Saber Commands

The help command displays information about **saber** commands. If help is called without any arguments, it displays a summary of all commands. If help is called with the name of a command, it displays information about the specified command.

```
1 -> help() ;
/* summary of commands not listed here */
2 ->
3 -> help("load") ;
/* information about load not listed here */
4 ->
```

If **saber** is executing in a windowing environment, the output of the help command will be displayed in a separate window. This can be disabled by unsetting the win_help option.

9.2. Getting Information on Names

The whatis command displays all the uses of a name for a function, data variable, structure, union, enumerator, type definition or macro definition. The name passed to whatis must be enclosed in quotes.

In the example below, the name x is used for a variable and a macro.

```
1  -> int x ;
2  -> #define x    99.99
3  ->
4  -> whatis("x");
'x' is currently defined as:
#define x 99.99 /* macro. */
extern int x; /* defined */
5 ->
```

Whatis lists uses of the name which are in scope, followed by uses of the name which are not in scope. The order of the listing indicates how the name would be resolved when it is used. In the list shown above, any use of the name x would be treated as a macro.

9.3. Cross-referencing a Variable or Function

The xref command cross-references a variable or function, listing where it is declared, where it is referenced, and what variables and functions it references.

It the argument is a function, xref lists all functions which call it and all variables which use it for an initialization value. Xref also lists all functions and global variables used in the body of the function itself.

In the example below, the function abc is referenced by the variable fptr and by the function main. The function abc references both functions foo and fum.

```
1 -> extern int abc(), foo(), fum() ;
2 ->
3 -> int (*fptr)() = &abc ;
4 -> void main() { abc() ; }
5 ->
6 -> int abc() { foo() ; fum() ; }
7 ->
```

The xref listing for function abc is shown below.

```
8 -> xref(abc)  ;
Referenced by:
   int (*fptr)()
   void main(/* void */)

References to:
   int foo()  (undefined)
   int fum()  (undefined)

Declared in:
   environment: defined
(void)
9 ->
```

If xref is called with the name of a variable (variable names must be quoted), it will list all functions in which the variable is used and all variables which use the named variable for an initialization value. If the named variable is initialized, all functions and variables it uses for its initialization value are also listed.

The example below displays the cross-reference listing for the initialized variable fptr.

```
10 -> xref("fptr")  ;
Referenced by:
   (nothing)

References to:
   int abc(/* void */) (unresolved)

Declared in:
   environment: initialized
(void)
11 ->
```

Xref only recognizes static references between objects. If the reference was dynamically created, such as by assigning the address of a variable to a pointer, it will not be listed. In the example below, the function foo does not appear in the cross-reference of the variable new_ptr because the reference was created dynamically.

```
12 -> int (*new_ptr)() ;
13 -> new_ptr = foo ;
14 ->
15 -> xref("new_ptr") ;
Referenced by:
   (nothing)

References to:
   (nothing)

Declared in:
   environment: defined
(void)
16 ->
```

Formal arguments and automatic variables cannot be cross-referenced.

9.4. Locating Unresolved References

The unres command lists unresolved references. An *unresolved reference* exists when a defined variable or function uses an undefined variable or function. In the example below, the defined function main uses the undefined functions test and foobar; therefore, main contains two unresolved references.

If unres is called without any arguments, it lists all unresolved references.

```
1 -> extern int test() , foobar(), fum() ;
2 ->
3 -> int main() { test() ; foobar() ; }
4 ->
5 -> unres() ;
int test() (undefined)
int foobar() (undefined)
(void)
6 ->
```

Since the function fum is not used by main, it is not listed as an unresolved reference.

If unres is called with the name of a function or the name of a variable (variable names must be quoted), it will list only undefined variables referenced in the named function or variable. The example below shows the listing of unresolved references for the function foo.

```
7 -> int foo() { comp() ; }
8 ->
9 -> unres(foo) ;
int comp()  (undefined)
(void)
10 ->
```

9.5. Getting Information About Addresses

The dump command displays information about an address. Dump will display the name, size and type of the object associated with the address.

In the example below, dump is used to display information about the address stored in the variable ptr, which points to the fourth element in the array many .

```
1 -> int *ptr ;
2 -> int many[10] ;
3 ->
4 -> ptr = &many[4] ;
(int *) 0959d4a <0x959c4 : 0x959ec >
5 ->
6 -> dump(ptr) ;
dump (0x959d4): int many[index  4]
Size = 4, contains type: int
(void)
7 ->
```

All valid addresses, including allocated data, formal arguments and automatic variables can be examined with dump.

9.6. Getting Information on Pointers

The badptr command lists all pointers which currently contain invalid addresses. The current value of the pointer and the location at which it became invalid are listed.

```
10 -> badptr() ;
Bad pointers:
   Bad pointer at address 0x11b56e, holds value of 0x13fe60.
      pointer went bad at line 31 in main.

   Bad pointer at address 0x11b55a, holds value of 0x11b578.
      pointer went bad at line 31 in main.
```

When a pointer goes bad, it is placed on the list even if a violation is not reported — a violation is reported only if the incorrect pointer value is used, or the new value for the pointer is a clear error. Once the pointer is assigned a valid address, it is removed from the list.

CHAPTER 10: Miscellaneous Commands

jobs	— list all stopped jobs
fg	— restart a stopped job
killjob	— kill a stopped job
suspend	— suspend **saber**
save	— save the current session
restore	— restore a saved session
catch	— catch signals before reaching program
ignore	— allow signals to reach program
rename	— rename a **saber** command
gripe	— send message to authors of **saber**
version	— print version information

10.1. Job Control

Entering <Control-Z> while **saber** is executing user code suspends **saber**, returning control to the shell. If the user has defined a signal handler for the <Control-Z> signal, and **saber** has been instructed to ignore the signal (with the ignore command), the signal handler will be called instead of returning to the shell.

The suspend command will suspend **saber** and return control to the shell. It can be placed in user code, or it can be used when the <Control-Z> cannot be issued.

In non-windowing environments, **saber** provides job control during editing similar to what is available with the **csh** shell. If <Control-Z> is entered while editing, the editing job is suspended and control returns back to **saber**. The jobs command produces a list of suspended editing jobs.

To resume a suspended job, call the fg command with the job number listed by jobs. If fg is not passed any arguments, it will resume the most recently suspended job. To terminate a suspended job, call killjob with its number.

In a windowing environment, entering <Control-Z> will not suspend the editing job. The chapter on handling files contains additional information on this subject.

10.2. Saving and Restoring a saber Session

The save and restore commands are used to save and restore a session.

If save is called with a file name, the session is saved in the specified file; otherwise, the session is saved in saber.save. Save prompts the user for a comment to be included with the saved session.

If restore is called with a file name, it loads the saved session from the specified file; otherwise, the session is loaded from saber.save. Any user comments are displayed when the session is restored.

Save is most frequently used right after all files have been loaded and linked. At any later point in this, or a completely new session with **saber**, invoking restore will return the environment back to a known starting point.

Save and restore can also be used to assist in debugging and testing user code. For instance, to test several execution paths through a function, save the session at a break level before the paths diverge. To test each execution path, restore the saved session and continue execution. This can be repeated as many times as necessary, until all execution paths have been tested.

Depending upon the nature of the session with **saber**, the exact environment may not be successfully restored. For instance, files that were open when the environment was saved and then later closed, will not be open when the session is restored.

10.3. Signal Processing

All signals are caught directly by **saber**. When a signal occurs, execution is stopped and a break level is created. The ignore command is used to specify which signals should be ignored by **saber** and passed directly to the user's program. The catch command is used to specify which ignored signals should be caught by **saber**. Calling catch or ignore without an argument will list all currently caught or ignored signals respectively.

10.4. Name Conflicts

If the name of a command conflicts with the name of a user-defined data variable or function, the name of the **saber** command can be changed using the rename command. In the example below, the name of the contents command is changed to _contents.

```
1 -> rename("contents", "_contents") ;
'contents' renamed to '_contents'
(void)
2 ->
```

Once the name has been changed, the new name must be used to execute the command.

```
3  -> contents() ;
Error#271: line 3:
   2:
*  3: contents() ;
'contents' is undefined
4  ->
5  -> _contents() ;
     /* Listing not shown */
```

Conflicting command names should be renamed before any files are loaded (usually in the initialization file .saberinit), thereby avoiding any accidental use of a **saber** command to resolve external references in user code.

10.5. Sending Mail to Saber Software

To report bugs, ask questions, and offer suggestions to the authors of **saber**, use the gripe command. If possible, bug reports should include examples of code which produce the bug. If the gripe messages are not being delivered properly, it may be necessary to specify a different mailing address with the -g command line option when **saber** is started.

10.6. Version information

The version command lists information about the current version of **saber**, including the gripe mailing address and the pathname for the help directory. The version command also displays the address and phone number for Saber Software.

```
1 -> version() ;
   Saber Software
   P.O. Box 2581
   Cambridge, Ma. 02238
   (617) 576-5893
Version 1.2 (SUN)
Date: 8/11/87
Help page directory: '/usr/local/lib/Saber'
Mailer: '/usr/ucb/mail'
Gripe address: 'mit-eddie!mit-athena!saber-bugs'
Saber Number: 10-0
Expiration date:
Log file = '/tmp/saber.a01955'
Comment:
(void)
2 ->
```

CHAPTER 11: Options and Customization

Saber can be customized by changing the default settings of options. The commands used to examine and change options are:

setopt	— set an option
unsetopt	— unset an option
printopt	— print the value of an option
getopt	— return the value of an option

11.1. Examining and Changing Options

A list of all options can be displayed printopt command, and changed with the setopt and unsetopt commands.

Setopt sets an option to a value. The value is determined by the type of the option. Boolean options, which can be only set or unset, do not require a value. All other options require the value to be either an integer or a string. If setopt is called without an argument, it displays all options that are set.

Unsetopt unsets an option.

Printoptprints the value of an option and a short description of its function. If printopt is called without an argument, it displays a listing of all options. To view detailed description of all options, call the help command with the topic "options"as the argument.

Getopt returns the value of an option in a string. This command can be used to save the value of an option before changing it with setopt.

11.2. Using An Initialization File

Saber searches the current working directory and the user's home directory for the initialization file, .saberinit. If the file exists, it will be included after the standard C library has been attached, but before any files listed on the command line are loaded.

The initialization file can be used to define macros, set options, initialize variables, or execute commands. A sample initialization file is shown below:

```
% cat .saberinit

/* Define macros for common commands. */
#define S         step() ;
#define N         next() ;
#define cd(path)    chdir(#path) ;
#define pwd         system("pwd") ;

/* Specify paths and load options */
setopt("path", ". ../test /usr/fred/src");
setopt("load_options", "-DLIMIT=10 -Iheaders") ;

/* make suppress terse and suppress warnings */
setopt("terse_suppress") ;
suppress(342,"stop") ;
unsetopt("terse_suppress") ;
```

Since the initialization file is included into the environment, it should contain only **saber** commands and C code that the user does not intend to debug or reload.

The first four lines of this sample startup file define macros that can be used instead of entering the full name of commands. The next two lines set the path and load_options options. The last three lines suppress a warning with the suppress command silenced so that the message printed by the suppress command is not displayed.

11.3. Saving Keystrokes

All **saber** commands are implemented as C functions, requiring the user to type parentheses, quotes, and semi-colons. However, since the environment includes a full C preprocessor, the user can define macros for frequently used commands.

For instance, if the following macro is defined, the user can simply type N to single step executing to the next line.

```
#define N next() ;
```

Many **saber** commands take a string as an argument. The user can avoid typing double quotes around these arguments by using the new # preprocessor operator. When a macro parameter is followed by a #, its text becomes quoted. (See the ansi page in the *Reference Manual*).

For example, the macro defined below allows the user to call load with a file name without placing the name within quotes.

```
1 -> #define ld(file) load(# file);
2 ->
3 -> ld(source.c)
Loading: source.c
(int) 0
4 ->
```

If a particular series of commands is to be performed often, then place those commands into a file and use the #include directive to include that file into the environment. When a file is included into the environment, it's contents are treated just as if they were entered by the user from the keyboard.

11.4. Useful Library Functions

All standard library functions may be called directly from **saber**. Two particularly useful library functions are **system** and **chdir**.

The **system** function executes commands passed to it within a string, as if the commands were entered to the shell. **System** is a convenient way to execute commands, such as **ls** or **grep**, without exiting **saber**.

```
1 -> system("ls") ;
a.out x.c y.c z.c
2 ->
```

The **chdir** function changes the current working directory. It is called with a string containing the new directory, just as it would be specified to the **cd** command in the shell. **Saber** will notify the user when the current working directory has changed.

```
1 -> chdir("..") ;
wd now: '/usr/saber/'
```

Some helpful macros for common uses of these library functions are shown below:

```
#define cd(path)    chdir(#path) ;
#define pwd         system("pwd") ;
#define ls          system("ls");
#define date        system("date") ;
```

The cd macro uses the # preprocessor operator described in the previous section.

11.5. Assert

The `assert` macro (found in `/usr/include/assert.h`) is used for verification of conditions while a program is running. A method to specify an alternative **assert** macro for use with **saber** is shown below:

```
#ifdef SABER/* Use this version with saber. */
#define assert(expr) { if(!(expr)) stop(); }
#else /* Otherwise, use the standard version. */
#include <assert.h>
#endif /* SABER*/
```

The **saber** version of `assert` will interrupt execution and generate a break level (by calling `stop`) when it detects a failed assertion. This allows the user to inspect variables and examine the execution stack at the location of the assertion failure.

11.6. Using Blocks

Blocks are useful for performing operations in the environment without producing adverse side effects or conflicting with global variables.

All automatic variables declared within the block are local to the block — they will cease to exist at the end of the block. These variables can be used for storing values and performing calculations, without affecting global variables, even of the same name.

```
1 -> /* A loop to test function 'f()' */
2 ->
3 -> {
4 +>     int i ;
5 +>     for(i = 0 ; i < 10 ; i++)
6 +>         printf("f(%d)=%d\n", i, f(i)) ;
7 +> }
```

Expressions and statements placed within a block will not be executed until the block is ended with a closing brace. The values produced by the expressions and statements within the block are not displayed. Only the `void` value produced at the end of the block is displayed.

Table of Contents

Chapter 1: Getting Started 1
- **1.1: Command Line Switches** 1

Chapter 2: The Saber-C Environment 3
- **2.1: Entering Input** 3
- **2.2: Understanding the Input Prompts** 4
- **2.3: Interrupting and Suspending Saber** 4
- **2.4: Executing Commands** 5
- **2.5: Defining Variables and Types** 6
- **2.6: Defining Functions** 7
- **2.7: Examining and Redoing Previous Input** 8
- **2.8: Clearing Previous Input** 8
- **2.9: Saving and Editing Previous Input** 9

Chapter 3: Break Levels 10
- **3.1: Generating Break Levels** 10
- **3.2: Continuing from a Break Level** 11
- **3.3: Reseting from a Break Level** 12
- **3.4: Displaying the Execution Stack** 13
- **3.5: Execution and Scope Location** 13

Chapter 4: Errors and Warnings 15
- **4.1: Message Layout** 15
- **4.2: Warnings** 16
- **4.3: Errors** 16
- **4.4: Suppressing Warnings and Errors** 17
- **4.5: Unsuppressing Violations** 18

Chapter 5: Execution 20
- **5.1: Incremental Execution** 20
- **5.2: Incremental Execution of Compiled Code** 22
- **5.3: Executing a program** 22
- **5.4: Stepping Execution** 23

Chapter 6: Loading and Unloading Files 25
- **6.1: Including Files** 25
- **6.2: Loading Files** 26
- **6.3: Reloading Files** 27
- **6.4: Swapping Files** 27
- **6.5: Unloading Files** 27
- **6.6: Loading Compiled Code with Debugging Information** 28
- **6.7: Loading Compiled Code without Debugging Information** 29

6.8: Attaching Libraries 30
6.9: Using the link command 31

Chapter 7: Handling Files 32
7.1: Listing Loaded Files 32
7.2: Listing Source Code 33
7.3: Editing Files 34
7.4: Editing and Windows 35
7.5: Editing in the Environment 35

Chapter 8: Debugging Commands 37
8.1: Setting Breakpoints 37
8.2: Setting Conditional Breakpoints 38
8.3: Setting Watchpoints 39
8.4: Setting Conditional Watchpoints 40
8.5: Tracing Program Execution 41
8.6: Listing and Deleting Debugging Items 41

Chapter 9: Information Commands 43
9.1: Getting Information on Saber Commands 43
9.2: Getting Information on Names 43
9.3: Cross-referencing a Variable or Function 44
9.4: Locating Unresolved References 46
9.5: Getting Information About Addresses 47
9.6: Getting Information on Pointers 47

Chapter 10: Miscellaneous Commands 49
10.1: Job Control 49
10.2: Saving and Restoring a saber Session 50
10.3: Signal Processing 50
10.4: Name Conflicts 50
10.5: Sending Mail to Saber Software 51
10.6: Version information 51

Chapter 11: Options and Customization 53
11.1: Examining and Changing Options 53
11.2: Using An Initialization File 53
11.3: Saving Keystrokes 54
11.4: Useful Library Functions 55
11.5: Assert 56
11.6: Using Blocks 56