NAME action – execute a debugging action

SYNOPSIS

```
void action()

void action(line)
int line ;

void action(function)
void (*func)() ;

void action(filename, line)
char *filename ;
int line ;
```

OPTIONS terse_list (unset)

DESCRIPTION

Action allows users to execute statements at different locations in a program without having to modify the source code. It can be used to make temporary changes to a program, such as adding debugging code, printf statements, and consistency checks. Actions cannot be set within compiled code.

Action prompts the user for a single statement to execute; multiple statements can be executed by enclosing them within a block. The action may use any global, formal, or automatic variables that are in scope at the location where the action is set.

If action is called without an argument, the action is executed at every line in every function. The action is not performed if the user is executing statements or expressions directly in the environment or at a break level. If the argument is a line number, a function, or a file and a line number, the action will be executed at the specified location only.

Actions can be listed with the status command and deleted with the delete command.

OPTIONS

If the **terse_list** option is set, the list command will not display the complete text of an active action when listing the location where the action was triggered.

EXAMPLE

The example below sets an action that prints the value of total every time line 10 of file main.c is reached. If the value of total equals 1000, the stop command is called and execution is suspended. Notice that a block is used because the action consists of more that one statement.

```
1 -> action("main.c", 10)
Setting action at "f.c", line 3, main().
Use braces for multiple statments.
Enter<Ctrl-D> to quit, <Ctrl-C> to redo.
action -> {
action +>   printf("Value = %d\n", total) ;
action +>   if (total == 0) stop();
action +> }
Action #1 set.

2 ->
```

SEE ALSO status, delete

NAME badptr − display information about bad pointers

SYNOPSIS

```
void badptr()
```

DESCRIPTION `Badptr` describes all known bad pointers, the value they currently hold, and the location at which they became invalid.

A pointer is considered invalid if it contains an illegal address or if it points to an address outside the bounds of the object to which it was originally assigned. Most invalid pointers are reported when they are created, but some invalid pointers are reported only when they are used. This behavior prevents spurious warnings when pointers are set to values such as `0`, `1`, or `-1`

SEE ALSO dump

NAME

catch – trap signals before reaching program
ignore – allow signals to reach program

SYNOPSIS

```
void catch()

void catch(sig_num)
int sig_num ;

void ignore()

void ignore(sig_num)
int sig_num ;
```

DESCRIPTION

Catching a signal with the `catch` command will trap the signal when it is generated and break execution of the program. Ignoring a signal with the `ignore` command will pass the signal directly to the user's program and execute a signal handler if it has been specified.

Calling either `catch` or `ignore` without an argument will list the signals that are currently caught or ignored. The following signals are ignored by default:

signal #14 - SIGALRM
signal #19 - SIGCONT
signal #20 - SIGCHLD
signal #28 - SIGWINCH

BUGS

If a signal is caught and a break level is generated, the signal is consumed. Ignoring the signal while at the break level and continuing execution will not pass the signal to the user's program.

Ignoring either a SIGINT or SIGTSTP signal will interfere with the ability to abort or suspend execution with <Control-C> and <Control-Z>.

NAME

cont – continue execution from a break level
reset – return to a previous break level

SYNOPSIS

```
void cont()
/* can enter <Control-D> as a shortcut */

void cont(continuation_value)
/* the type of the continuation_value is determined dynamically.

void reset()

void reset(level)
int level ;
```

DESCRIPTION

Cont exits from a break level and continues execution of the interrupted program.

If the break level was created by the stop command or by the user entering a <Control-C> during execution, cont is called without an argument. A shortcut for doing this is to enter a <Control-D> at the break level.

If the break level was created due to a warning or error, cont can be called with a continuation value as its argument. The type of the continuation value is specified in the error or warning message when the break level is generated. (The message can be redisplayed with the whereami command). Cont substitutes the continuation value for the expression which caused the error or warning when it continues execution.

Reset returns to a previous break level without continuing execution from the current break level.

If reset is called without an argument (or if level equals 0), the user is returned to the top level of the environment. If level has a positive value, the user is returned to that break level. If level has a negative value, the user is returned the specified number of break levels.

BUGS

Not all errors can be continued from by supplying a continuation value to cont.

SEE ALSO

stop, where, whereami

NAME contents – display information about loaded files

SYNOPSIS

```
void contents()

void contents(module_name)
char *module_name ;
```

DESCRIPTION

Contents displays information about loaded files. If it is called without an argument, it will prompt the user to list loaded user files, loaded library files, or all loaded files.

If the argument is the name of a file, it will prompt the user to list functions, variables, or types defined in the file. If the argument is a library, all files archived within the library are listed.

The contents of a file that has been linked from a library can be displayed by specifying the name of the library followed by the name of the member file enclosed within parentheses. For example the contents of the file printf.o, which was linked from the library /lib/libc.a, can be displayed by entering:

```
1 -> contents("/lib/libc.a(printf.o)");
```

SEE ALSO load

NAME dump – display information about an address

SYNOPSIS

```
void dump(address)
void *address ;
```

DESCRIPTION

Dump displays information about the data located at an address. It is especially useful for determining to what variable or address a pointer points.

Dump provides the name and size of the variable or function located at the address. If the address refers to allocated data, dump displays the size of the allocated data, and, if available, the type of data most recently stored there.

Dump indicates if the address is being watched by a tag debugging action, or if the address contains a bad pointer.

In the example below, dump is used to display information about a value stored in a pointer.

```
1 -> int i, *ptr;
2 -> ptr = &i ;
(int *) 0x125278
3 ->
4 -> dump(ptr) ;
address = 0x125278,name = 'i'
size= 4, type = int
5 ->
```

SEE ALSO badptr, calloct

NAME edit – edit a file

SYNOPSIS

```
void edit()

void edit(line_number)
int line_number ;

void edit(function)
void (*function)() ;

void edit(file_name)
char *file_name ;
```

OPTIONS

```
edit_name = "vi"
edit_options = "+%d %s"
edit_line_file = ":n +%d %s\n"
edit_line = "%dG"
edit_query = "\x01b:f\n\x01b:\n"
win_edit (unset on ascii terminals)
win_edit_opts (unset on ascii terminals)

auto_reload (set)
edit_jobs = 2
```

DESCRIPTION

Edit a file by invoking the editor specified by the `edit_name` option. This option is initialized to the shell environment variable **EDITOR**. If the environment variable is missing, the default value "vi" **is used instead.**

If `edit` is called without an argument, the file at the current list location is edited. If a line number is supplied as the argument, the current list file is edited at that line number.

If a function is supplied as the argument, the file containing the function will be edited at the beginning of the function. If a file name is supplied as the argument, the specified file will be edited at the beginning of the file.

The `win_edit` option indicates whether or not to create a new window for the edit job. When **saber** is running in a windowing environment, this option automatically set. If `win_edit` is not set, **saber** will be suspended while the editing job is active.

If the file to be edited is already open in an active or suspended editing job, **saber** will reuse the editing job and update the file to the new location. If **saber** needs to start a new editing job and the maximum number of edit jobs are active (determined by the `edit_jobs` option), **saber** will reuse the least recently used editing job.

If the user enters a <Control-Z> (suspend signal) to the editor, **saber** will reload the file being edited. If the file does not end with a ".c" suffix, **saber** executes a general load command that checks all files for reloading.

It is important that, before interacting with **saber**, the user leaves the editor in a state that can accept queries about the file and line being edited. This means that the editor **vi** should not be left in Q mode and the editor **emacs** should not be reading input from the mini-buffer.

Suspended editing jobs can be listed with the jobs command and resumed with the fg command.

OPTIONS

The edit_jobs option indicates the number of new editing jobs to create. If the editor is **emacs**, the option is set to 1. If the editor is **vi**, the option is set to 2 on a windowing system and 5 on a non-windowing system.

The option edit_options indicates how to start the editor, and the edit_line, and edit_line_file, options describe the editor command sequence used to move to a new line and move between files.

If the win_edit option is not set, then **saber** examines the auto_reload option to see if files should be automatically reloaded when an editing jobs is finished or suspended. If the auto_reload option is not set, **saber** will prompt the user before reloading files.

SEE ALSO

fg, jobs

NAME

english – describe a C type expression in English

SYNOPSIS

```
void english(typeof_code)
int typeof_code ;
```

DESCRIPTION

English describes a C type expression in English. The argument to english must be the value returned by the **saber** typeof operator.

EXAMPLE

The example below indicates how to correctly call the english command with a C type expression. Notice that the type expression is identical to a *cast* in the C language. The type expression is equivalent to the complete declaration for a variable excluding the name of the variable itself.

```
1 -> char *(*func)() ;
2 ->
3 -> english( typeof(char *(*)()) ) ;
pointer to function returning pointer to char.
(void)
4 ->
```

The example below contains a macro that automatically gets the type code returned by the typeof operator before calling english.

```
-> #define eng(type) english(typeof(type))
```

SEE ALSO

typeof

NAME

freealloc − free all allocated memory

SYNOPSIS

int freealloc();

DESCRIPTION

Freealloc frees all memory allocated by the standard library functions **malloc**, **calloc**, and **realloc**. Any memory allocated by the **saber** commands malloct and calloct will also be freed. Freealloc returns the number of bytes freed.

Freealloc is automatically called by the run and reinit commands, but not by the start command.

Freealloc should be called only when the user has no more use for allocated memory. Any pointers to allocated data will no longer be valid after the call.

SEE ALSO

calloct, malloct, run, start, reinit

NAME

help – information about **saber** commands

SYNOPSIS

```
void help()

void help(name)
void (*name)() ;

void help(name)
char *name ;
```

OPTIONS

win_help (unset on ascii terminals)
win_help_opts (unset on ascii terminals)

DESCRIPTION

Help displays information about **saber** commands. If help is called without an argument, it displays a summary of all commands.

If the argument to help is the name of a command, a complete description of the command is displayed. The name of a command can be passed to help directly or it can be placed within a string.

If the win_help option is set, the help information will be displayed in a separate window. This option is always unset on ascii terminals.

BUGS

If the name of a **saber** command is changed with the rename command, help must be called with the original name of the command contained within a string. Help will not recognize the new name of the command nor the old name of the command if it is not enclosed within a string.

NAME

history – list previously entered input
##, #N, #-N – redo previous entered input

SYNOPSIS

```
void history()

void history(number)
int number ;
```

DESCRIPTION

History lists previous lines of input to **saber**. If history is called with an argument, then the specified number of lines are listed.

To redo a previous line of input, use #N where N is the number of the input line to be redone. If N is negative, then the N'th previous line is redone. To redo the most recent input line, use ##, which is equivalent to entering #-1.

When a line is redone, the terminating newline is omitted, allowing the user to make changes to the line before it is executed again.

EXAMPLE

The following example indicates how to redo the previous line of input, the 10th line of input, and the 5th previous line of input.

```
-> ##     /* Redo previous line. */
-> #10    /* Redo line 10. */
-> #-5    /* Redo current line minus 5. */
```

NAME

fg – restart a suspended editing job
jobs – list all suspended editing jobs
killjob – kill a suspended editing job

SYNOPSIS

```
void fg()

void fg(jobnum)
int jobnum ;

void jobs()

void killjob(jobnum)
int jobnum ;
```

DESCRIPTION

These three **saber** commands mimic the C Shell commands **jobs**, **fg**, and **kill**.

Jobs displays all suspended editing sessions begun with the edit command. The most recently suspended job is marked by a + sign , and the job suspended previous to that is marked by a - sign.

Fg restarts the most recently suspended editing job. If a job number is supplied as an argument, the specified job is restarted.

Killjob kills the indicated job by sending a SIGTERM signal (#15). If the editing job does not terminate within three seconds, a SIGKILL signal (#9) is sent to the editing job.

BUGS

Killing a job with killjob sends a non-catchable kill signal to the edit job if the terminate signal does not work. This may prevent the editing job from performing any cleanup actions before it quits.

NAME

link – link files from libraries
unres – list all unresolved references

SYNOPSIS

```
int link()

int link(function)
void (*function)() ;

int link(var_name)
char *var_name ;

void unres()

void unres(func)
void (*func)() ;

void unres(var_name)
char *var_name ;
```

DESCRIPTION

Link searches all attached libraries to resolve undefined variables and functions. If it is called without an argument, all files required to resolve all undefined references will be loaded from attached libraries. Link returns the number (possibly 0) of undefined variables remaining after the linking process.

If a function is supplied as the argument, link will resolve only the undefined variables and functions used by the specified function. If a variable name is supplied as the argument, link will resolve only the undefined variables and functions used as initialization values for the variable.

Unres lists all undefined variables and functions which are referenced by user code. If a function is supplied as the argument, unres will list all undefined variables and functions referenced by that function. If a variable name is supplied as the argument, unres will list all undefined variables and functions used as initialization values for the variable.

SEE ALSO

xref

NAME

list – display source code lines

SYNOPSIS

```
void list()

void list(beg_line)
int beg_line ;

void list(beg_line, end_line)
int beg_line, end_line ;

void list(function)
void *function ;

void list(file_name)
char *file_name ;
```

OPTIONS

page_list (unset (10 lines))
terse_list (unset)
win_pager more
win_list (unset)
win_list_opts (unset)

DESCRIPTION

List displays lines of source code. The source line corresponding to the current execution location will have a `` *´´ preceding the line number. Breakpoints are displayed with a `` B´´ preceding the line. Signals and errors are noted after the source line on which they occurred.

Actions are listed in full above the line on which they are set. Each line of the action is listed with a <An> preceding the text of the action, where n represents the action number. Tagactions are displayed after the source line that triggered them. Each line of a tagaction is listed with a *<Tn>* preceding the text of the action, where *n* represents the tagaction number.

If list is called without an argument, it begins listing at the current list location. The list location is set to the first line of a file when it is loaded, the execution location when a break level is entered, or the last line displayed by the most recent list command.

If the argument is a line number, list begins listing at the specified line number in the current list file. If the number is negative, listing begins at the current list location minus the specified number. If list is called with a starting and ending line number, all lines between the two are listed.

If the argument is a function, listing starts at the beginning of the function. If the argument is a filename, listing starts at the top of the file.

OPTIONS

If the win_list option is set, the listing will appear in a separate window. The win_pager option indicates the program to use to list in this window. The win_list_opts option may be set to customize where the window should appear.

If the page_list option is set, list will display the specified number of lines; otherwise 10 lines are displayed.

If the terse_list option in set, actions that execute everywhere and the text of tagactions are not listed when displaying the locations at which they were triggered.

NAME load – load or reload files

SYNOPSIS

```
int load()

int load(files_and_options)
char *files_and_options ;
```

OPTIONS

```
path (unset)
load_options (unset)
batch_load  (unset)
page_load   (unset)
src_err = 3
terse_load (unset)
ansi (unset)

auto_replace  (unset)
auto_compile  (unset)
```

DESCRIPTION

Load loads new files into **saber** or reload files that have been modified since they were loaded. Load will accept C source files, relocatable object files, and libraries which have been archived (with **ar**) and ranlibed (with **ranlib**). It returns 0 if all files were successfully loaded, and -1 if an error occurred.

Load can be called with the name of one or more files to be loaded. Libraries can be loaded by specifying either the complete pathname of the library file, or by using the -l option (the same option used by the C compiler).

To list all currently loaded files, use the contents command.

Loading source files

Source files are translated according to the provisions of the ANSI Draft Proposed C Standard (X3J11/85-044), which extends and modifies the language described by Kernighan and Ritchie in *The C Programming Language*, (Prentice-Hall, 1978). See the portability reference manual page for differences between **saber** and the C compiler, **cc**.

Compiled files and libraries

Object code files compiled with or without the -g compiler switch (for debugging) can be loaded. Object code files compiled with debugging information should be used whenever possible.

Loading a library makes the contents of the library available to **saber**. The individual functions and variables within the library are dynamically linked when they are needed to satisfy references to undefined variables.

Reloading outdated files

If load is called without arguments, it reloads any outdated files. If load is called with the name of a file that is already loaded, the file is reloaded if it has been modified.

A source code file will be reloaded if it or any of the files it includes have been modified since it was loaded. A compiled file or library will be reloaded if it has been modified (i.e. re-compiled or re-archived) since it was loaded. If only the source file for the compiled file has been modified, **saber** will prompt the user to remake the compiled file before reloading it.

Load automatically resets to the top level before reloading a file. Therefore, the user cannot reload a file at a break level and continue execution.

SWITCHES

The load command accepts all switches used with the C compiler, although only the switches listed below are meaningful. Any switches listed in the argument string to load are processed when a file is loaded. If no switches are given, then the switches specified by the load_options option are used. The following switches are currently recognized:

−D*macro_name***[=***definition***]**

Causes *macro_name* to become defined as if a #define *directive had occurred. If definition is not supplied, then "1" is used.*

−U*macro_name*

Causes the predefined *macro_name* to become undefined as if a #undef *directive had occurred.*

−I*directory_name*

Adds *directory_name* to the list of directories to search for files specified by the #include *preprocessor directive. When the name of a file is surrounded by double quotes, it is first sought in the directory of the file being read, then in directories specified by* -I *options, and finally in the* /usr/include*directory. If a file name is quoted with angle brackets, it is only sought in directories specified by* -I *options and the* /usr/include*directory.*

−L*library_path*

Adds *library_path* to the list of directories to search for libraries. When a library is loaded using the -lx format, then a file called libx.a is sought first in the directories specified by -L options, then in the standard directories /lib, /usr/lib, and /usr/local/lib.

−w

Suppresses all load time warnings. This option is equivalent to setting the terse_load option for the duration of loading the file. If this switch is used when a library is loaded, warnings will be suppressed when modules are linked from the library.

−G

Ignores debugging information, produced by the −g switch of the compiler, when loading compiled files. This allows the user to load compiled files for which **saber** has trouble reading the debugging information. If this switch is used when a library is loaded, debugging information will be ignored when linking from the library.

Predefined macros

The following macros are predefined by the load command:

__FILE__:	name of the file being read
__LINE__:	line number of the file being read
__FUNC__:	name of the function being read
__SABER__:	always defined to "1"
SABER1_2:	defined to "1" in **saber** version 1.2
SABER:	always defined to "1"

If the ansi option is set, the following macros are also defined:

```
__DATE__:   date the file was read ("Mmm dd yyyy" )
__TIME__:   time the file was read ("hh:mm:ss")
__STDC__:   always defined to "1"
```

The remaining options listed here are usually defined in the system-wide initialization file if appropriate:

```
sun:       defined to "1" on a Sun
mc68000:   defined to "1" on a Sun
mc68020:   defined to "1" on a Sun3
vax:       defined to "1" on a VAX and MicroVAX
unix:      defined to "1" under Unix and Ultrix
ultrix:    defined to "1" under Ultrix
```

Note that the macros __SABER__ and SABER are not predefined by the C compiler, **cc**. The macro __STDC__ is defined by C compilers and interpreters that conform to the ANSI Draft Standard.

OPTIONS

The path option indicates which directories should be searched for files. When load is trying to locate a file, it will look in directories specified by the path option before looking in the current working directory. If the current working directory is changed (e.g. with the **chdir** function), the path option should be changed so that any relative paths are still correct.

If the batch_load option is set, the user is not prompted for an action when an error or warning is detected.

If the terse_load option is set, warnings will not be reported as files are loaded. Errors are always reported.

The page_load option governs the number of lines of violations that are displayed before prompting for more. This option is meaningful only if the batch_load option is not set. If page_load is not set, a full screenful is displayed.

The src_err option determines the number of source lines listed when an error or warning is reported.

If the auto_replace option is set, **saber** will automatically replace a source code module with the object code (or vice-versa) without issuing a ’replace’ prompt.

If the auto_compile option is set, **saber** will automatically recompile outdated object code modules before reloading them. If the object code file does not exist, **saber** will automatically try to create it. If a makefile exists, **saber** will execute a **make** command to create it, otherwise **saber** will use **cc**.

BUGS

Loading an object file without debugging information may cause spurious warnings since initialized variables can be grouped together without correct type or size information.

If **saber** rejects loading an object file that was compiled with debugging information (i.e. due to a type redeclaration), try loading the file with the -G switch.

When a module is reloaded, it is first unloaded then loaded from scratch. If a fatal error occurs during the loading process, the module is left unloaded.

Occasionally, linking libraries may produce spurious warnings about size or type redeclarations. It is usually safe to continue past these warnings.

SEE ALSO ansi, contents, portability, unload

NAME

malloct – allocate memory with type checking
calloct – allocate cleared memory with type checking

SYNOPSIS

```
void *malloct(size)
int size ;

void *calloct(nelem, typeof_code)
unsigned int nelem ;
unsigned int typeof_code ;
```

OPTIONS

save_memory (unset)

DESCRIPTION

Malloct and calloct allocate memory on which run-time type checking is performed. If the save_memory option is unset, the standard C library functions **malloc** and **calloc** use malloct and calloct to allocate memory, thereby silently providing type-checking for all allocated memory.

The library functions **free**, **cfree**, and **realloc** can be used with memory allocated by malloct and calloct.

Malloct returns a pointer to a block of memory of size bytes. Whenever data is stored in this memory, **saber** notes the type of the data. Later, when the memory is used, **saber** checks that the type used to examine the data is consistent with the type used to store it.

Calloct returns a pointer to a block of memory of size nelem times the size of the type indicated by typeof_code. The memory is initialized to the value 0 and **saber** notes that it should be used to hold data of the type specified by typeof_code.

Typeof_code is the type code value yielded by the **saber** operator typeof. If typeof_code is the code for the type void, the allocated memory will have no type associated with it, and exactly nelem bytes are allocated.

OPTIONS

If the save_memory option is set, the library functions **malloc** and **calloc** will not use malloct and calloct to allocate memory. This reduces the amount of memory used for allocated data by almost one half and, depending on the application, can improve execution speed.

EXAMPLE

In the example below, the user allocates an array to hold 100 pointers to integers.

```
1 -> ptr = (int **) calloct(100, typeof(int *)) ;
(int **) 0x136dfc < 0x136dfc : 0x136f8c >
2 ->
```

If data of a different type is stored in this memory, **saber** will issue a type mismatch warning.

SEE ALSO typeof, freealloc,

NAME options – options for configuring **saber**

DESCRIPTION The options described below let the user configure **saber** for many different types of programming projects. Options can be display with the printopt command, and changed with the setopt and unsetopt commands.

GENERAL OPTIONS

path (default is unset)
A string containing a list of directories to search when loading, editing, or listing files. The list is searched in order. The directory names should be separated by spaces, and the directories can be specified as absolute or relative pathnames.

term (default is set to the environment variable TERM)
A string containing the name of the terminal being used.

LOAD OPTIONS

load_options (default is unset)
A string containing switches to be passed to the load command when loading files. These switches should be the same as those specified to the C compiler when compiling files, such as "-I../ifiles -Dmacro". This option contains the default switches for load; therefore, it is not used if the user specifies switches in the argument to load. See the load reference manual page for a complete description of available switches.

sys_load_options (set in the system .saberinit file)
A string containing switches that are always passed to the load command when the user loads files. This option should be used to specify system characteristics, such as the location of libraries or system header files.

batch_load (default is unset)
If set, the load command will not stop at each error and warning to prompt the user for an action to perform. This option is useful when loading a group of files at the beginning of a **saber** session.

RELOAD OPTIONS

auto_reload (default is set)
If set, the edit and run commands will automatically reload any modules that are out of date. If the option is not set, the user will be prompted before reloading any files.

auto_replace (default is unset)
If set, the load command will automatically replace a source file when called with the name of the compiled file, or replace a compiled file when called with the

name of the source file. If the option is not set, the user will be prompted before replacing any loaded files.

auto_compile (default is unset)
If set, the load command will automatically recompile any out-dated object files before reloading them. If a makefile exists in the directory of the source file, **saber** will execute a **make** command to bring the .o file up to date; otherwise **saber** will compile the file with **cc** -c -g and any other flags specified in the load_options option. If the option is not set, the user will be prompted before recompiling.

RUN-TIME OPTIONS

program_name (default is "a.out")
This string becomes the first argument to a program (argv[0]) when the commands run or start are used.

batch_run (default is unset)
If set, the user will not be prompted on how to proceed with run-time errors and warnings as they occur.

MEMORY OPTIONS

save_memory (default is unset)
If set, extra memory for run-time type checking will not be used when the user allocates memory. This flag should be set when memory is scarce or for portions of a program that allocate very large arrays.

sbrk_size (default is 524288)
This option specifies the amount of memory that can be allocated by the **sbrk** library function. Programs that only use the **malloc** and **free** functions will not be affected by this option. Only programs that call **sbrk** directly (usually programs that perform their own memory allocation) will be restricted to allocating the amount of memory specified by this option. The upper limit for this option can be determined by the amount of swap space available on the system.

OUTPUT

print_string — (default is 20)
An integer that indicates how many characters of a string are printed. If more characters can be displayed, a '...' will follow the string.

print_pointer — (default is set)
If set, pointers are displayed with their low and high boundaries.

terse_list — (default is unset)
If set, tagactions and actions that execute everywhere will not be displayed when listing the source line at which they were triggered.

terse_run — (default is unset)
If set, all warnings will be ignored when a program is executing.

terse_load — (default is unset)
If set, all warnings will be ignored when loading a file.

terse_suppress — (default is unset)
If set, the suppress command will not echo the name of the violation being suppressed.

terse_where — (default is unset)
If set, the where command will not list the formal arguments of each function on the execution stack.

PAGING OPTIONS

page_cmds — (default is the size of the terminal's screen)
An integer indicating the number of lines of output displayed by a **saber** command before a *more* prompt is issued. If the option is unset, command output will not be paginated.

page_list — (default is 10 lines)
An integer that indicates how many lines of source code the list command displays before before a *more* prompt is issued. If the option is unset, source code listings will not be paginated.

page_load — (default is the size of the terminal's screen)
An integer that indicates how many lines of error and warning messages the load command displays before a *more* prompt is issued. This option is meaningful only when the batch_load option is set. If the option is unset, the messages will not be paginated.

SOURCE LISTING OPTIONS

src_err — (default is 3)
An integer that indicates how many lines of source code are displayed when an error or warning is reported.

src_stop — (default is 3)
An integer that indicates how many lines of source code are displayed when a

	break level is created for the first time.
src_step	(default is 1) An integer that indicates how many lines of source code the step and next command display after stepped execution of a statement.

EDITING OPTIONS

edit_name	(default is set to environment variable EDITOR) This string contains the name of the editor to start when editing a file. If the environment variable **EDITOR** does not exist, the option is set to "vi"**.**
edit_line	(default depends on editor) This string describes the editor command sequence to move to a specific line of a file. The sequences '%d' be substituted with the line number. This option must be changed before any files are edited.
edit_line_file	(default depends on editor) This string describes the editor command sequence to move between files. The sequences '%d' and '%s' will be substituted with the line number and filename respectively. This option must be changed before any files are edited.
edit_query	(default depends on editor) This string describes the editor command sequence to request the name of the file currently being edited. This option must be changed before any files are edited
edit_options	(default depends on editor) This string describes the command line arguments that are used to invoke the editor. The sequences '%d' and '%s' will be substituted with the line number and filename respectively. If the editor normally creates its own window for execution, it must be told not to with a command line switch specified in this flag. Changing the value of this flag must be done before the first request to edit.
edit_jobs	(default is 2 or 1 if emacs) An integer that indicates the number of simultaneous editing jobs **saber** will manage. If the maxmum number of jobs has been reached, **saber** will begin to reuse editing jobs. This option must be changed before any files are edited.

WINDOW OPTIONS

win_pager	(default is set to environment variable PAGER) A string that indicates the paginating program to invoke when source listings and help screens are displayed in a separate window. If the environment variable PAGER does not exist, the program "more" is used.
win_creator	(default depends on windowing system) A string that indicates the program to invoke to create a new window (usually **xterm** or **shelltool**).
win_display	(default set to environment variable DISPLAY) This option is used by X windows to indicate where to create the window. It is initialized to the DISPLAY environment variable.
win_x	(set if using X windows)

win_no_raise (default is unset)
If unset, the **saber** workspace window is automatically raised after files have been loaded. This is to prevent it from being buried underneath editing windows.

win_timeout (default is 60 seconds)
A number indicating the number of seconds **saber** waits for a window to be created.

COMMAND WINDOW OPTIONS

win_edit (default is set when using X or Sunview)
If set, a new window is created when editing a file.

win_edit_opts (default is unset)
A string containing options for the window creation program when opening a window for editing. This option can be used to specify the size and location of a window.

win_list (default is set when using X or Sunview)
If set, a new window is created when listing a file.

win_list_opts (default is unset)
A string containing options for the window creation program when opening a window for listing. This option can be used to specify the size and location of a window.

win_help (default is set when using X or Sunview)
If set, a new window is created when displaying help information.

win_help_opts (default is unset)
A string containing options for the window creation program when opening a window for displaying help information. This option can be used to specify the size and location of a window.

win_redo (default is set)
If set, a new window is created when the user selects to redo an line of input. If set when using an ascii terminal, a new edit job is started for this text.

win_redo_opts (default is unset)
A string containing options for the window creation program when opening a window for redoing a line of input. This option can be used to specify the size and location of a window.

win_io (default is set when using X or Sunview)
If set, a new window is created for input and output when a program is executed using the `run` command.

win_io_opts (default is unset)
A string containing options for the window creation program when opening a window for program input and output. This option can be used to specify the size and location of a window.

MISCELLANEOUS FLAGS

logfile (default is a temporary file name)
This option holds the name of the file that **saber** uses to record the environment session. It is periodically used by **saber**, so care should be taken not to delete it. It holds a recording of all user input into the environment.

echo (default is unset)
If set, **saber** will echo the input stream after the preprocessing has taken place. This is similar to calling the C compiler with the -E switch.

ansi (default is unset)
If set, preprocessing and function prototype conversion are done in strict conformance with the ANSI Draft Standard. This is not always compatible with the standard C compiler. For more information on preprocessing semantics and function prototypes, see the *portability* and *ansi* manual page.

SEE ALSO

setopt, unsetopt, printopt, getopt

NAME

quit – quit from **saber**
suspend – suspend **saber** and return to the shell

SYNOPSIS

void quit()

void suspend()

DESCRIPTION

Quit exits **saber** and returns the user to the shell. If there are any active editing jobs, **saber** will prompt the user for confirmation to quit. If the user requested a log file on the command line, the name of the log file is displayed as part of the exit message.

Suspend will suspend **saber** and return the user to the shell. The same effect can be produced by entering a <Control-Z> to the environment. Suspend is useful for creating scripts that will suspend **saber** or for situations when entering a <Control-Z> does not produce a stop signal

NAME

rename − rename a **saber** command to avoid name conflicts

SYNOPSIS

```
void rename()

void rename(old_name, new_name)
char *old_name ;
char *new_name ;
```

DESCRIPTION

Rename changes the name of a **saber** command to prevent name conflicts between **saber** commands and variables or functions defined by the user. When renaming commands, it is preferable to make a minor change in the name, such as prepending an underscore to the beginning of the name.

The conflicting command name should be renamed before any files are loaded to avoid accidental use of the **saber** command to resolve external references in user code. Rename will not rename a command if it has already been used to resolve a reference within the user's program.

The command name is changed from old_name to new_name (both names must be enclosed within strings). After a command has been renamed, new_name must be used to refer to the command.

If rename is called without arguments, it lists all **saber** commands that have been renamed.

BUGS

Help does not recognize renamed commands. If a command is renamed, help must be called with the original name of the command contained within a string.

SEE ALSO

xref, whatis

NAME

run – initialize variables and execute main
start – execute main without initializing variables
reinit – reinitialize all variables

SYNOPSIS

```
void run()

void run(command_line)
char *command_line ;

void start()

void start(command_line)
char *command_line ;

void reinit()

void reinit(variable_name)
char *variable_name ;
```

OPTIONS

program_name = "a.out"
auto_reload (set)
win_io (unset on ascii terminals)
win_io_opts (unset on ascii terminals)

DESCRIPTION

Run initializes all variables (by calling reinit), processes command line arguments, executes main (by calling start), and closes open files when main returns. If run is called from a break level, it first resets to the top level of the environment.

Start processes command line arguments and executes main without resetting the break level, initializing variables, or closing files when main returns.

Both run and start use the program_name option for the first argument to main (argv[0]) and the command line for the remaining arguments. If a command line is not supplied, the command line arguments from the previous call to run or start are reused. If the argument is an empty string, main is called with only the program name as an argument.

The command line can contain the input and output redirection symbols shown below. Pipelining input and output is not supported.

> file
>& file
: redirect standard output to a file. If the file does not exists, create it. If the file exists, truncate it to zero length before writing to it. If the & symbol is used, both standard output and standard error are redirected to the file.

>> file
>>& file
: append standard output to the end of a file. If the file does not exists, create it. If the & symbol is used, both standard output and standard error are redirected to the file.

< file — redirect standard input to a file.

Reinit reinitializes all global variables, frees allocated memory, closes open files, and reinitializes all signal handlers. If reinit is called with a variable name, only the specified variable will be initialized.

If reinit is called from a break level, errors may be introduced when global variables are reinitialized to their initial values. In addition, reinit will free allocated memory which may still be used.

OPTIONS

If the win_io option is set and the run command is called, then program input and output will be redirected to a separate window or terminal. If **saber** is running in a windowing environment using X windows or Sunview, a new window will appear automatically. If **saber** is running in a non-windowing environment, the user will be prompted for the name of another terminal to use for the input and output.

If the auto_reload flag is set, run and start will automatically reload any modified files before executing the program. If the option is unset, the user will be prompted before reloading any files.

SEE ALSO

freealloc

NAME

save – save the current session
restore – restore a saved session

SYNOPSIS

```
int save()

int save(file_name)
char *file_name ;

int restore()

int restore(file_name)
char *file_name ;
```

DESCRIPTION

Save saves the current session in the file saber.save or in a file specified by the user. Save prompts the user for a comment to be included with the saved session.

Restore loads a saved session from the specified file or from the file saber.save if a filename is not specified. Any user comments in the save file are displayed when the session is restored.

BUGS

Information about open files is not saved with the session. Thus, saving a session while files are open may create problems when the session is restored, unless the exact same files have been opened again. Also, open files which are not open in the restored session might be closed improperly and data could be lost.

The state of the terminal is neither saved nor restored with a session.

NAME

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

SYNOPSIS

```
void setopt()

void setopt(option)
char *option ;

void setopt(option, value)
char *option ;
/* value is determined by the option */

void unsetopt()

void unsetopt(option)
char *option ;

void printopt()

void printopt(option)
char *option ;

char *getopt(option)
char *option ;
```

DESCRIPTION

These commands let the user examine and change **saber** options. A list of all the options is provided on the options reference page.

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.

Printopt prints 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.

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.

SEE ALSO

options

NAME

status – display all breakpoints, actions, and tags.
delete – delete a breakpoint, tag, or action

SYNOPSIS

```
void status()

void delete()

void delete(number)
int number ;
```

DESCRIPTION

Status displays all actions, breakpoints, and tags, along with the number of each debugging item. Delete deletes an action, breakpoint, or tag.

If delete is called without an argument, it prompts the user to delete all debugging items; otherwise, it deletes the specified debugging item.

If the debugging item to be deleted is currently active on the execution stack, it will be *zombied* instead of being deleted. When execution continues, the zombied item cannot be activated anew, but it will not be deleted until it has completed executing. When displayed by the status command, zombied items are listed with the letter `` Z´´ preceding the debugging item number.

SEE ALSO

action, stop, tag, tagaction,

NAME

step – step execution by statement, enter functions
next – step execution by line, do not enter functions
stepout – continue execution until function returns

SYNOPSIS

```
void step()

void step(number)
int number ;

void next()

void next(number)
int number ;

void stepout()
```

OPTIONS

src_step 1
src_stop 3

DESCRIPTION

Step executes a single *statement* and stops execution. If the statement is a function, the function is entered and execution will be stopped on the first line of the entered function. If a line contains multiple statements, execution will move to the next statement on the line.

Next executes an entire *line* before stopping execution, regardless of the number of statements on the line. Next does not enter functions.

If an argument is supplied to either step or next, the specified number of statements or lines are executed before stopping.

stepout continues execution until the current function returns. Execution will stop before the next statement in the calling function is executed.

OPTIONS

The number of source lines listed after a step or next command is determined by the src_step option. The number of source lines listed after a stepout command is determined by the src_stop option.

BUGS

Step, next and stepout will not stop within compiled code.

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.

NAME stop – set a breakpoint

SYNOPSIS

```
int stop()

int stop(line_number)
int line_number ;

int stop(function)
void (*func)() ;

int stop(filename, line_number)
char *filename ;
int line_number ;
```

OPTIONS src_stop 3

DESCRIPTION

Stop sets a breakpoint in user code. When the breakpoint is encountered, execution is interrupted and a break level is created.

If the argument is a line number, the breakpoint is set at that line in the current file. If the argument is a function, the breakpoint is set on the first line of that function. If a filename and line number are passed as arguments, the breakpoint is set at that line in the specified file.

If stop is called without an argument, it will immediately interrupt execution and create a break level. This allows the user to create conditional breakpoints by combining stop within other debugging commands, such as action or tagaction .

OPTIONS

The src_stop option indicates how many lines are listed when execution is stopped. The letter `` B´´ appears before the line at which the breakpoint is set.

BUGS

Stop cannot set breakpoints in compiled code.

SEE ALSO status, delete, cont, action, tagaction

NAME

suppress – suppress reporting of a warning
unsuppress – reactivate reporting of a warning

SYNOPSIS

```
int suppress()

int suppress(number)
int number ;

int suppress(number, function)
int number ;
void (*function)() ;

int suppress(number, location)
int number ;
char *location ;

int suppress(number, line)
int number ;
int line ;

int unsuppress()

int unsuppress(number)
int number ;

int unsuppress(number, function)
int number ;
void (*function)() ;

int unsuppress(number, location)
int number ;
char *location ;

int unsuppress(number, line)
int number ;
int line ;
```

OPTIONS

terse_suppress (unset)

DESCRIPTION

Suppress suppresses the reporting of warnings and errors (collectively known as violations). If suppress is called without arguments, all of the currently suppressed violations are listed. After the listing, **saber** prompts to save all of the suppressions to a file (the default is the startup file). All suppressed violations (except those suppressed by default) will be appended to this file with a comment explaining the violation number and the suppression range.

If suppress is called with the number of a violation as the only argument, reporting of that violation will be suppressed everywhere.

If a second argument is supplied, it indicates the region in which reporting of the warning should be suppressed. If the region is a file name (a string), reporting is suppressed for the entire file. If the region is a line number (an integer), reporting is suppressed at that line in the current file. If the region is a file name followed by a colon followed by a line number (all part of one string, e.g. "foo.c:10"), then the violation is suppressed at the indicated line number in the specified file. If the region is a function name followed by an open/close parenthesis (all part of one string, e.g. "main()"), then the violation is suppressed while over this indicated function.

If the region is a function (either as specified above, or passed as a function pointer), the exact behavior depends on the type of violation being suppressed. If the violation occurs when functions are called, reporting is suppressed wherever the specified function is called. A list of this category of violations is shown below:

#29 Losing information during call to function.
#30 Calling function with too many arguments.
#31 Calling function with too few arguments.
#32 Signed vs. unsigned type mismatch during call to function.
#33 Benign type mismatch during call to function.
#34 Serious type mismatch during call to function.
#35 Passing illegal enumeration value to function.
#36 Function call is passing a bad pointer.
#37 Function call is passing a bad pointer in a structure/union.
#38 Calling function which contains unresolved references.
#39 Calling compiled function which contains unresolved references.
#340 Function *'name'* was redeclared.

For all other warnings, reporting is suppressed while inside the specified function.

Unsuppress reactivates the reporting of suppressed violations. If the warning is suppressed within a region, unsuppress must be called with the region as the second argument. If unsuppress is called without arguments, it will prompt the user to reactivate each suppressed warning.

OPTIONS

If the terse_suppress option is set, suppress will not echo the name of the violation it is suppressing.

NAME

tag – stop execution when an address is modified
tagaction – execute an action when an address is modified

SYNOPSIS

```
int tag(address)
void *address ;

int tag(address, size)
void *address ;
int size ;

void tagaction(address)
void *address ;
int size ;

void tagaction(address, size)
void *address ;
int size ;
```

OPTIONS

save_memory (unset)
terse_list (unset)

DESCRIPTION

Tag stops execution when an address is modified. Tagaction executes a user defined action when an address is modified.

All global variables may be tagged. Dynamically allocated data may be tagged if it was allocated when the save_memory flag was unset or if it was allocated with the malloct or calloct commands.

Function arguments and automatic variables of a function can be tagged only while within the function — their addresses are undefined when outside the function. The tag must be set from a break level at which execution has stopped inside the function. The tag will be deleted automatically when the function returns.

The number of bytes that are tagged is determined by the type of the data pointed to by address. If the data's type is int, four bytes will be tagged. If the argument is the name of an array, the first element of the array is tagged. If the argument is the address of the array, the entire array is tagged.

```
1 -> int array[10] ;
2 ->
3 -> tag(array) ; /* Tag first element only. */
4 -> tag(&array) ; /* Tags entire array. */
```

The user can specify the number of bytes to tag with a second argument to tag or tagaction .

The value of the tagged address is changed before the tag is triggered. Tags are not triggered if the address is modified while executing statements directly in the environment or while executing within compiled code.

Tagaction prompts the user for a single statement to execute when the tagged address is modified. Multiple statements can be executed by placing the statements within a block. Only global variables may be used within the body of the action.

Tags are deleted automatically if a variable becomes undefined when a file is unloaded. Only one tag action can be associated at an individual address.

BUGS

Tags are not triggered if the address is modified while executing within compiled code.

Only global variables may be used in the body of a tagaction. To print the value of an local variable, the user must find the absolute address of the variable, and then dereference it according to its type. E.g. If the address of a local integer variable is 0x12345, then the tagaction might read:

```
printf("Value of variable = %d\n", *(int *)0x12345) ;
```

SEE ALSO

status, delete, stop

NAME touch – mark address as initialized and valid

SYNOPSIS

```
void touch(address, size)
char *address ;
int size ;
```

DESCRIPTION

Touch marks memory as initialized and valid in order to prevent "garbage value" and "type mismatch" violations from occurring when the memory is used. The memory from address to address+ size is marked.

The example below shows how touch can be used to suppress "type mismatch" warnings when the value 99 is stored as an int, but examined as a char.

```
1 -> char c, *ptr ;
2 -> ptr = (char *) malloc( sizeof(int) ) ;
3 ->
4 -> *(int *)ptr = 99 ;
5 -> touch(ptr, sizeof(int)) ;
6 ->
7 -> c = *ptr ;
```

SEE ALSO suppress

NAME trace – trace program execution

SYNOPSIS

```
int trace()

int trace(level)
int level ;
```

DESCRIPTION

Trace displays source code lines as they are executed. Each line is displayed once regardless of the number of statements on the line.

The possible arguments to trace are shown below:

0: No 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 statements executed within an action or tagaction will not be traced.

NAME typeof – yields a **saber** type code

SYNOPSIS

typeof(type)
typeof expr

DESCRIPTION

The typeof operator yields an unsigned integer representing **saber**'s type code for the given type or expression. Typeof parses exactly like the C sizeof operator.

The calloct, english, and udpf functions require the type code provided by typeof.

SEE ALSO calloct, english, udpf

NAME

udpf – specify a user-defined printing function

SYNOPSIS

```
void (*udpf(typeof_value, func))()
unsigned typeof_value ;
void (*func)() ;
```

DESCRIPTION

Udpf specifies a user-defined function for printing values of a given type. When an expression is evaluated in the environment, a default printing function is used to display the type and value of that expression is printed. The user can use udpf to designate an alternate printing function to print this information for a specific type. This is especially useful for examining abstract data types such as linked lists.

The printing function is called whenever a value of the given type is displayed. The printing function should accept one argument of the type it is should print. The example below lists a printing function for displaying a linked list.

```
/* Definition of a singly linked list. */
struct NODE
{
   int value ;
   struct NODE *next ;
} ;

/* A function to print a linked list. */
void list_printer(node)
struct NODE *node ;
{
   printf("{") ;
   for( ; node ; node = node->next)
      printf("%d", node->value) ;
   printf("}");
   fflush(stdout); /* Flush buffered output. */
}
```

When the function is called, it is passed the value to be printed. To be consistent with the default printing facility, it should either write to **stderr** or call **fflush** before returning. Otherwise, some of the output may be buffered and printed at the wrong time.

In the example below, the user sets list_printer as the printing function for values of type struct NODE *:

```
-> udpf(typeof(struct NODE *), list_printer) ;
```

If the second argument to udpf is nil, the default printing function is reinstated. If the second argument equals 1, the values of the specified will not be printed. The example below shows how to suppress the printing of values of type void.

```
-> udpf(typeof(void), 1) ;
```

Udpf returns a pointer to the previous user-defined printing function. The value -1 is returned if an invalid typeof code or an illegal function address is specified.

BUGS

User defined print functions cannot be set for values of type structor union; Instead, print functions for pointer to structor pointer to union must be used.

If an error occurs in a user-defined print function, doing a stack trace may trigger an infinite loop of internal errors.

SEE ALSO

typeof

NAME unload – unload files

SYNOPSIS

```
int unload()

int unload(list_of_files)
char *list_of_files;
```

OPTIONS path (unset)

DESCRIPTION Unload unloads files from **saber**. It automatically resets to the top level of the environment; therefore, the user cannot unload a file at a break level and then continue execution.

All functions and static variables defined within an unloaded file become undefined. Global data variables, macro definitions, structures, unions, enumerators, enumeration constants, and type definitions will not become undefined if they are declared or defined in any other loaded module.

Breakpoints, tags, and actions are deleted if they are set on variables or functions which become undefined when a file is unloaded.

Unloading an attached library will detach the library and unload the individual modules that have been dynamically linked from the library.

If unload is called without an argument, it prompts the user to unload user files, library files, or all files.

If an error occurs while unloading, unload will stop unloading the file and return an error value of -1.

OPTIONS The search path specified by the path option is used to determine the complete pathname of modules to unload.

SEE ALSO load

NAME

up – move up the execution stack
down – move down the execution stack

SYNOPSIS

```
void up()

void up(n)
int n ;

void down()

void down(n)
int n ;
```

DESCRIPTION

Up moves the current scope location up the execution stack, away from the current break level and toward the top level of the environment. Down moves the current scope location down the execution stack, away from the top level of the environment and toward the current break level.

If an argument is supplied, both up and down will move the specified number of levels along the execution stack.

The scope location is the point at which all variables, type, and macros are scoped. When a break level is generated, the scope location is set to the point at which execution was interrupted.

Use the where command to display the execution stack. Use the whereami command to list the execution and scope locations.

SEE ALSO

where, whereami

NAME

version – display **saber** version information
gripe – send a message to the authors

SYNOPSIS

```
void version()

void gripe()

void gripe(filename)
char *filename ;
```

DESCRIPTION

Version displays the following items of information: the version number, the release date, the directory containing help manual pages, the name of the log file, and the electronic mailing address for the authors of **saber**.

Version also displays the address and phone number for Saber Software.

Gripe sends an electronic message to the authors of **saber**. This command should be used to report bugs and offer suggestions. If possible, bug reports should include examples of the source code that produced the problem.

To mail a file, call the gripe command with the name of the file as an argument.

The electronic mailing address is set when **saber** is installed. If gripe messages are not being delivered properly, it may be necessary to specify a different mailing address using the -g command line option.

NAME

whatis – display all uses of a name

SYNOPSIS

```
void whatis(name)
char *name ;
```

DESCRIPTION

Whatis displays all uses of the specified name for functions, data variables, structures, unions, enumerators, type definitions and macro definitions.

Whatis displays first all uses of the name within scope at the current scope location, followed by all uses of the name not within scope. The order of the listing represents the order in which the specified name is resolved when it is used.

The example below displays the output of whatis for the name test when it is used for a variable and a macro definition.

```
1 -> int test ;
2 -> #define test 100
3 ->
4 -> whatis("test");
'test' is declared as:
#define test 100
extern int test; /* defined */
(void)
5 ->
```

NAME

where – display the execution stack
whereami – display execution and scope locations

SYNOPSIS

```
void where()

void where(n)
int n ;

void whereami()
```

OPTIONS

where_terse (unset)

DESCRIPTION

Where displays a trace of the execution stack, starting from the location where execution has stopped. If n is supplied, only the top n functions on the stack are displayed.

Whereami displays the current execution location (i.e. the point at which execution was stopped when a break level is entered). If an error or a warning caused the most recent break level, whereami displays the error or warning number, and if execution can be continued, the arguments that can be passed to cont are also displayed.

The scope location (i.e. the point to which all variables, functions, types, and macros are scoped) is displayed if it is different from the execution location. The scope location can be changed to different locations on the execution stack with the up and down commands.

OPTIONS

If the where_terse option is set, functions on the execution stack are listed without their formal parameters. Object code functions loaded without debugging information are always listed without formal parameters.

SEE ALSO

down, up

NAME xref – cross reference a function or variable

SYNOPSIS

```
void xref(function)
void (*function)() ;

void xref(variable_name)
char *variable_name ;
```

DESCRIPTION

Xref cross references a function or variable, listing where it is used, what variables and function it uses, and where it is declared and defined.

If the argument is a function, xref lists all functions which call it and all variables which use it for an initialization value. This is followed by a list of functions and global variables used within the body of the specified function. Xref also lists all the files in which the function is defined.

If the argument is a variable name, xref lists all functions in which it is used. If the variable is initialized, xref lists all functions and variables which are used in the initialization value. All files that declare or define the variable are also listed.

Xref displays only static references that exist in source or compiled code files. Dynamically created references, such as assigning the address of a variable to a pointer in the environment, are not listed. Formal parameters and automatic variables cannot be cross referenced.