NAME

abort − generate a fault

SYNOPSIS

#include <stdlib.h>

void abort(void)

DESCRIPTION

Abort executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

SEE ALSO

sigaction(2), **exit**(2).

DIAGNOSTICS

Usually ‘‘abort − core dumped’’ from the shell.

BUGS

The abort() function does not flush standard I/O buffers. Use **fflush**(3).

NAME

abs − integer absolute value

SYNOPSIS

#include <stdlib.h>

int abs(int *i*)

DESCRIPTION

Abs returns the absolute value of its integer operand.

SEE ALSO

floor(3).

BUGS

Applying the *abs* function to the most negative integer generates a result which is the most negative integer. That is, abs(0x80000000) returns 0x80000000 as a result on a machine with 32-bit ints. Using the result in unsigned computations is sound however.

NAME

assert − program verification

SYNOPSIS

#include <assert.h>

void assert(int *expression***)**

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be true at this point in the program. It causes an **abort**(3) with a diagnostic comment on the standard output when *expression* is false (0). Compiling with the **cc**(1) option **−DNDEBUG** effectively deletes **assert** from the program.

DIAGNOSTICS

'Assertion "*expression*" failed: file *f* line *n*.' *F* is the source file and *n* the source line number of the **assert** statement.

NAME

atof, atoi, atol − convert ASCII to numbers

SYNOPSIS

#include <stdlib.h>

double atof(const char ****nptr*)**
int atoi(const char ****nptr*)**
long atol(const char ****nptr*)**

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional ‘e’ or ‘E’ followed by an optionally signed integer.

Atoi and **atol** recognize an optional string of spaces, then an optional sign, then a string of digits.

SEE ALSO

strtol(3), **strtod**(3), **scanf**(3).

BUGS

There are no provisions for overflow.

NAME

bstring, bcopy, bcmp, bzero, ffs − bit and byte string operations

SYNOPSIS

#include <sys/types.h>
#include <string.h>
#include <unistd.h>

void bcopy(const void * *src***, void *** *dst***, size_t** *length***)**
int bcmp(const void * *b1***, const void *** *b2***, size_t** *length***)**
void bzero(void * *dst***, size_t** *length***)**
int ffs(int *i***)**

DESCRIPTION

The functions **bcopy**, **bcmp**, and **bzero** operate on variable length strings of bytes. They do not check for null bytes as the routines in **string**(3) do.

Bcopy copies *length* bytes from string *src* to the string *dst*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* 0 bytes in the string *b1*.

Ffs find the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

BUGS

The **bcopy**, **bcmp**, and **bzero** functions are obsolete; new code should use **memmove**, **memcmp**, and **memset** respectively.

The **bcopy** routine takes parameters backwards from **memcpy**, **memmove**, and **strcpy**.

NAME

crypt − one-way password encryption function

SYNOPSIS

```
#define _MINIX_SOURCE 1
#include <unistd.h>

char *crypt(const char *key, const char *salt)
```

DESCRIPTION

The first use of **crypt()** is to encrypt a password. Its second use is to authenticate a shadow password. In both cases **crypt()** calls **pwdauth**(8) to do the real work.

Crypt() encrypts a password if called with a user typed key, and a salt whose first two characters are in the set [./0-9A-Za-z]. The result is a character string in the [./0-9A-Za-z] alphabet of which the first two characters are equal to the salt, and the rest is the result of encrypting the key and the salt.

If **crypt()** is called with a salt that has the form *##user* then the key is encrypted and compared to the encrypted password of *user* in the shadow password file. If they are equal then **crypt()** returns the *##user* argument, if not then some other string is returned. This trick assures that the normal way to authenticate a password still works:

```
if (strcmp(pw->pw_passwd, crypt(key, pw->pw_passwd))) ...
```

If *key* is a null string, and the shadow password is a null string or the salt is a null string then the result equals *salt*. (This is because the caller can't tell if a password field is empty in the shadow password file.)

The key and salt are limited to 1024 bytes total including the null bytes.

FILES

/usr/lib/pwdauth	The password authentication program

SEE ALSO

getpass(3), **getpwent**(3), **passwd**(5), **pwdauth**(8).

NOTES

The result of an encryption is returned in a static array that is overwritten by each call. The return value should not be modified.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ctime, localtime, gmtime, asctime, tzset − convert date and time to ASCII

SYNOPSIS

#include <sys/types.h>
#include <time.h>

void tzset(void)
char *ctime(const time_t **clock*)
char *asctime(const struct tm **tm*)
struct tm *localtime(const time_t **clock*)
struct tm *gmtime(const time_t **clock*)

DESCRIPTION

Tzset uses the value of the environment variable **TZ** to set up the time conversion information used by **localtime**.

If **TZ** does not appear in the environment, the **TZDEFAULT** file (as defined in <*tzfile.h*>) is used by **localtime**. If this file fails for any reason, the GMT offset as provided by the kernel is used. In this case, DST is ignored, resulting in the time being incorrect by some amount if DST is currently in effect. If this fails for any reason, GMT is used.

If **TZ** appears in the environment but its value is a null string, Greenwich Mean Time is used; if **TZ** appears and begins with a slash, it is used as the absolute pathname of the **tzfile**(5)-format file from which to read the time conversion information; if **TZ** appears and begins with a character other than a slash, it's used as a pathname relative to the system time conversion information directory, defined as **TZDIR** in the include file **tzfile.h**. If this file fails for any reason, the GMT offset as provided by the kernel is used, as described above. If this fails for any reason, GMT is used. See **TZ**(5) for a proper description of the **TZ** variable.

Ctime converts a time value, pointed to by *clock*, such as returned by **time**(2) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

Sun Sep 16 01:03:52 1973\n\0

Localtime and **gmtime** return pointers to structures containing the broken-down time. **Localtime** corrects for the time zone and possible daylight savings time; **gmtime** converts directly to GMT, which is the time UNIX uses. **Asctime** converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm {
        int tm_sec;     /* 0-59   seconds */
        int tm_min;     /* 0-59   minutes */
        int tm_hour;    /* 0-23   hour */
        int tm_mday;    /* 1-31   day of month */
        int tm_mon;     /* 0-11   month */
        int tm_year;    /* 0-     year − 1900 */
        int tm_wday;    /* 0-6    day of week (Sunday = 0) */
        int tm_yday;    /* 0-365  day of year */
        int tm_isdst;   /* flag:  daylight savings time in effect */
        long tm_gmtoff; /* offset from GMT in seconds */
        char **tm_zone; /* abbreviation of timezone name */
};
```

Tm_isdst is non-zero if a time zone adjustment such as Daylight Savings time is in effect.

Tm_gmtoff is the offset (in seconds) of the time represented from GMT, with positive values indicating East of Greenwich.

FILES

/usr/lib/zoneinfo	time zone information directory
/etc/localtime	local time zone file

SEE ALSO

time(2), **getenv**(3), **tzfile**(5), **TZ**(5), **environ**(7), **zic**(8).

NOTE

The return values point to static data whose content is overwritten by each call. The **tm_zone** field of a returned **struct tm** points to a static array of characters, which will also be overwritten at the next call (and by calls to **tzset**).

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii, toupper, tolower, toascii − character classification macros

SYNOPSIS

#include <ctype.h>

int isalpha(int *c***)**
...

DESCRIPTION

These macros classify characters by table lookup. Each is a predicate returning nonzero for true, zero for false. **Isascii** and **toascii** are defined on all integer values; the rest are defined only on the range of **unsigned char** and on the special value EOF (see **stdio**(3)).

isalpha	*c* is a letter
isupper	*c* is an upper case letter
islower	*c* is a lower case letter
isdigit	*c* is a digit
isxdigit	*c* is a hex digit
isalnum	*c* is an alphanumeric character
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or formfeed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, code 040(8) (space) through 0176 (tilde)
isgraph	*c* is a printing character, similar to **isprint** except false for space.
iscntrl	*c* is a delete character (0177) or ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200
tolower	*c* is converted to lower case. Return value is undefined if not **isupper**(*c*).
toupper	*c* is converted to upper case. Return value is undefined if not **islower**(*c*).
toascii	*c* is converted to be a valid ascii character.

SEE ALSO

ascii(7)

NAME

curses − screen/window management library

SYNOPSIS

cc demo.c -lcurses

DESCRIPTION

Curses is a library of screen and window management routines. It is modeled after the UNIX curses and ncurses libraries. Normally, programs written for curses should be easily ported to UNIX, and vice versa.

To use the routines, the function initscr() must first be called. This creates two 'windows' for the user: stdscr and curscr. Stdscr is the default window for the user to make changes on, and curscr reflects the current contents of the physical display screen. The user writes or edits the stdscr window to his liking, then calls the refresh() function to make curscr and the physical screen look like stdscr. When the user program terminates, it should call the endwin() function to restore things to normal.

There are all sorts of window manipulation routines available to the programmer: auxiliary windows may be created, edited, moved and deleted. The terminal may be set in many different modes, output text may be attributed with blink, blank, bold and reverse attributes. Screen colors may also be set, foreground and background. There are window-specific printf- and scanf-like routines, routines for scrolling, box-drawing, window overlaying, clearing routines etc.

For more and detailed information, see the library source codes. All curses functions are preceded by a complete description.

FUNCTIONS

Below is a list over the available functions, together with a brief description of what they do. In general, functions whose names start with 'w' differ from the one without 'w' (like wmove vs. move) signify that a specific window is used. Without a 'w', sdtscr is implied. The functions that start with 'mv' before the 'genereic' function name signify that a cursor motion should be made before the actual work. 'mv' and 'w' combine as expected.

Most routines that return an int will return the manifest constant ERR if there is a failure during execution. Routines that return a char actually return an int, so that ERR does not conflict with the character code 0xff. All characters from 0 to 0xff are allowed for usage with curses.

Some routines, like {mv}{w} printw() and {mv}{w}scanw() return a meaningful positive value if the operation is successful.

The curses package uses some predefined types, variables and manifest constants that are also available to the programmer. There are also a few globally accessible variables that should not be touched by the application program. Those untouchable variables have names starting with an underscore (_) to avoid conflicts. The user-accessible types, variables and constants are (there are a number of other constants defining character attribute names and function key names - consult <curses.h> for details):

(manifest constants)

TRUE	boolean true
FALSE	boolean false
ERR	unsuccessfull operation
OK	successfull operation

(types)

WINDOW	a window structure type
bool	boolean flag type

(variables)

WINDOW curscr	physical display image
WINDOW stdscr	default user drawing board
int LINES	terminal height
int COLS	terminal width

int NONL	\n causes CR and LF when TRUE

The following is an alphabetical list of the curses functions, together with their types, parameters and a short comment for each (win is a window, ch, vc, hc are characters, buf is a character buffer, attrs is an attribute bit map, bf is a boolean flag. Note that 'characters' in this context usually can have 16 bits):

int waddch(win,ch) int addch(ch) int mvaddch(y,x,ch) int mvwaddch(win,y,x,ch)	put char in stdscr
int waddstr(win,str) int addstr(str) int mvaddstr(y,x,str) int mvwaddstr(win,y,x,str)	put string in stdscr
void wattroff(win,attrs) void attroff(attrs)	clear attribute(s) in window
void wattron(win,attrs) void attron(attrs)	add attribute(s) in window
void wattrset(win,attrs) void attrset(attrs)	set window char attributes
int baudrate()	dummy for compatibility
void beep()	ring the bell or visible bell if no bell available
void flash()	flash terminal screen or rings bell if no visible bell available
void wbox(win,miny,minx,maxy,maxx,vc,hc) void box(win,vc,hc)	box in a window, with given characters
void cbreak()	set terminal cbreak mode
void wclear(win) void clear()	clear stdscr
void clearok(win,bf)	marks window for screen clear
int wclrtobot(win) int clrtobot() int mvclrtoeol(y,x) int mvwclrtobot(win,y,x)	clear from cursor to end of line and all lines down this line
int wclrtoeol(win) int clrtoeol() int mvclrtoeol(y,x) int mvwclrtoeol(win,y,x)	clear from cursor to end of line
int wdelch(win) int delch() int mvdelch(y,x) int mvwdelch(win,y,x)	delete a char in a window

int wdeleteln(win)	delete a line in a window
int deleteln()	
int mvdeleteln(y,x)	
int mvwdeleteln(win,y,x)	
void delwin(win)	delete a window or a subwindow
void doupdate()	update physical screen
void echo()	set terminal echo mode
int endwin()	cleanup and curses finitialization
void werase(win)	erase a window
void erase()	
int erasechar()	return char delete character
int fixterm()	dummy for compatibility
void flushinp()	kill pending keyboard input
int wgetch(win)	get char via a window
int getch()	
int mvgetch(y,x)	
int mvwgetch(win,y,x)	
int wgetstr(win,str)	get string via window to a buffer
int getstr(str)	
int mvgetstr(y,x,str)	
int mvwgetstr(win,y,x,str)	
void getyx(win,y,x)	get a window’s cursor position
int gettmode()	dummy for compatibility
void idlok(win,bf)	dummy for compatibility
WINDOW *initscr()	curses initialization (ret stdscr or NULL)
int winch(win)	get char at window cursor
int inch()	
int mvinch(y,x)	
int mvwinch(win,y,x)	
int winsch(win,ch)	insert character in a window
int insch(ch)	
int mvinsch(y,x,ch)	
int mvwinsch(win,y,x,ch)	
int winsertln(win)	insert new line in a window
int insertln()	
int mvinsertln(y,x)	
int mvwinsertln(win,y,x)	
void keypad(win,bf)	marks a window for keypad usage
int killchar()	return line delete character
char *longname()	returns terminal description string
void leaveok(win,bf)	marks window for cursor ’update leave’
void meta(win,bf)	marks window for meta
int move(y,x)	move cursor in stdscr
int mvcur(oldy,oldx,y,x)	move terminal cursor to <y,x>

int mvprintw(y,x,fmt,args)	move & print string in stdscr
int mvscanw(y,x,fmt,args)	move & get values via stdscr
int mvwin(win,y,x)	move window on physical screen
int mvwprintw(win,x,y,fmt,args)	move & print string in a window
int mvwscanw(win,y,x,fmt,args)	move & get values via a window
WINDOW *newwin(lines,cols,begy,begx)	create a new window
void nl()	set terminal cr-crlf mapping mode
void nocbreak()	unset terminal cbreak mod
void nodelay(win,bf)	marks window for no input wait
void noecho()	unset terminal echo mode
void nonl()	unset terminal cr-crlf mapping mode
void noraw()	unset raw terminal mode
void overlay(win1,win2)	overlay one window on another
void overwrite(win1,win2)	overwrite one window on another
int printw(fmt,args)	print string in stdscr
void raw()	set raw terminal mode
void refrbrk(bf)	set screen update break mode
void refresh()	refresh stdscr
int resetterm()	dummy for compatibility
int resetty()	restore terminal I/O modes
int saveoldterm()	dummy for compatibility
int saveterm()	dummy for compatibility
int savetty()	save terminal I/O modes
int scanw(fmt,args)	get values via stdscr
void scroll(win)	scroll scrolling region of a window
void scrollok(win,bf)	marks a window to allow scroll
void setcolors(A_COLOR(for,back))	sets the forground and background colors of stdscr
void set_curs(visibility)	0 for invisible, 1 for visible, 2 for good visible
int setsrcreg(miny,maxy)	define stdscr's scroll region
int setterm()	dummy for compatibility
int setupterm(term,fd,errret)	set up terminal
void standend()	start normal chars in stdscr
void standout()	start standout chars in stdscr
WINDOW *subwin(win,lines,cols,begy,begx)	create a sub-window in window win
int tabsize(ts)	set/get tabsize of stdscr
void touchwin(win)	mark a window as totally modified
char *unctrl(ch)	char-to-string converter
int wmove(win,y,x)	move cursor in a window
void wnoutrefresh(win)	create internal screen image
int wprintw(win,fmt,args)	print string in a window
void wrefresh(win)	refresh window
int wscanw(win,fmt,args)	get values via a window
void wsetcolors(win,A_COLOR(for,back))	sets the forground and background colors of the specified window
int wsetsrcreg(win,miny,maxy)	define a window's scrolling region
void wstandend(win)	start normal chars in window
void wstandout(win)	start standout chars in window
int wtabsize(win,ts)	set/get tabsize of a window

BUGS

Function keys are not available under the MINIX version.

NAME

directory, opendir, readdir, rewinddir, closedir, telldir, seekdir − directory routines

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname)
struct dirent *readdir(DIR *dirp)
void rewinddir(DIR *dirp)
int closedir(DIR *dirp)

#define _MINIX  1
#include <sys/types.h>
#include <dirent.h>

long telldir(DIR *dirp)
int seekdir(DIR *dirp, long pos)
```

DESCRIPTION

These routines form a system independent interface to access directories.

Opendir() opens the directory *dirname* and returns a pointer to this open directory stream.

Readdir() reads one entry from the directory as a pointer to a structure containing the field **d_name**, a character array containing the null-terminated name of the entry.

Rewinddir() allows the directory to be read again from the beginning.

Closedir() closes the directory and releases administrative data.

The Minix specific functions **telldir()** and **seekdir()** allow one to get the current position in the directory file and to return there later. **Seekdir()** may only be called with a position returned by **telldir()** or 0 (rewind). These functions should not be used in portable programs.

SEE ALSO

dir(5).

DIAGNOSTICS

Opendir() returns a null pointer if *dirname* can't be opened, or if it can't allocate enough memory for the **DIR** structure.

Readdir() returns null if there are no more directory entries or on error.

Closedir() and **seekdir()** returns 0 on success, -1 on error.

Telldir() returns -1 on error.

All of them set **errno** appropriately. **Readdir()** will only set **errno** on error, not on end-of-dir, so you should set **errno** to zero beforehand, and check its value if **readdir()** returns null.

NOTES

The return value of **readdir()** needs to be copied before the next operation on the same directory if it is to be saved.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

editline − command-line editing library with history

SYNOPSIS

```
char *
readline(prompt)
    char        *prompt;

void
add_history(line)
    char *line;
```

DESCRIPTION

Editline is a library that provides an line-editing interface with text recall. It is intended to be compatible with the *readline* library provided by the Free Software Foundation, but much smaller. The bulk of this manual page describes the user interface.

The *readline* routine returns a line of text with the trailing newline removed. The data is returned in a buffer allocated with *malloc*(3), so the space should be released with *free*(3) when the calling program is done with it. Before accepting input from the user, the specified *prompt* is displayed on the terminal.

The *add_history* routine makes a copy of the specified *line* and adds it to the internal history list.

User Interface

A program that uses this library provides a simple emacs-like editing interface to its users. A line may be edited before it is sent to the calling program by typing either control characters or escape sequences. A control character, shown as a caret followed by a letter, is typed by holding down the ''control'' key while the letter is typed. For example, ''^A'' is a control-A. An escape sequence is entered by typing the ''escape'' key followed by one or more characters. The escape key is abbreviated as ''ESC.'' Note that unlike control keys, case matters in escape sequences; ''ESC F'' is not the same as ''ESC f''.

An editing command may be typed anywhere on the line, not just at the beginning. In addition, a return may also be typed anywhere on the line, not just at the end.

Most editing commands may be given a repeat count, *n*, where *n* is a number. To enter a repeat count, type the escape key, the number, and then the command to execute. For example, ''ESC 4 ^f'' moves forward four characters. If a command may be given a repeat count then the text ''[n]'' is given at the end of its description.

The following control characters are accepted:

^A	Move to the beginning of the line
^B	Move left (backwards) [n]
^D	Delete character [n]
^E	Move to end of line
^F	Move right (forwards) [n]
^G	Ring the bell
^H	Delete character before cursor (backspace key) [n]
^I	Complete filename (tab key); see below
^J	Done with line (return key)
^K	Kill to end of line (or column [n])
^L	Redisplay line
^M	Done with line (alternate return key)
^N	Get next line from history [n]
^P	Get previous line from history [n]
^R	Search backward (forward if [n]) through history for text; must start line if text begins with an uparrow
^T	Transpose characters
^V	Insert next character, even if it is an edit command
^W	Wipe to the mark

^X^X	Exchange current location and mark
^Y	Yank back last killed text
^[	Start an escape sequence (escape key)
^]c	Move forward to next character ‘‘c’’
^?	Delete character before cursor (delete key) [n]

The following escape sequences are provided.

ESC ^H	Delete previous word (backspace key) [n]
ESC DEL	Delete previous word (delete key) [n]
ESC SP	Set the mark (space key); see ^X^X and ^Y above
ESC .	Get the last (or [n]’th) word from previous line
ESC ?	Show possible completions; see below
ESC <	Move to start of history
ESC >	Move to end of history
ESC b	Move backward a word [n]
ESC d	Delete word under cursor [n]
ESC f	Move forward a word [n]
ESC l	Make word lowercase [n]
ESC m	Toggle if 8bit chars display normally or with ‘‘M−’’ prefix
ESC u	Make word uppercase [n]
ESC y	Yank back last killed text
ESC v	Show library version
ESC w	Make area up to mark yankable
ESC nn	Set repeat count to the number nn
ESC C	Read from environment variable ‘‘_C_’’, where C is an uppercase letter

The *editline* library has a small macro facility. If you type the escape key followed by an uppercase letter, *C*, then the contents of the environment variable *_C_* are read in as if you had typed them at the keyboard. For example, if the variable *_L_* contains the following:

^A^Kecho ’^V^[[H^V^[[2J’^M

Then typing ‘‘ESC L’’ will move to the beginning of the line, kill the entire line, enter the echo command needed to clear the terminal (if your terminal is like a VT-100), and send the line back to the shell.

The *editline* library also does filename completion. Suppose the root directory has the following files in it:

bin vmunix
core vmunix.old

If you type ‘‘rm /v’’ and then the tab key. *Editline* will then finish off as much of the name as possible by adding ‘‘munix’’. Because the name is not unique, it will then beep. If you type the escape key and a question mark, it will display the two choices. If you then type a period and a tab, the library will finish off the filename for you:

rm /v[TAB]*munix*.TAB*old*

The tab key is shown by ‘‘[TAB]’’ and the automatically-entered text is shown in italics.

BUGS AND LIMITATIONS

Cannot handle lines more than 80 columns.

AUTHORS

Simmule R. Turner <uunet.uu.net!capitol!sysgo!simmy> and Rich $alz <rsalz@osf.org>. Original manual page by DaviD W. Sanderson <dws@ssec.wisc.edu>.

NAME

end, etext, edata − last locations in program

SYNOPSIS

```
extern int etext;
extern int edata;
extern int end, _end;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of **etext** is the first address above the program text, **edata** above the initialized data region, and **end** above the uninitialized data region. **_end** is the same as **end**, but in the implementers name space, i.e. for use in libraries.

When execution begins, the program break coincides with **end**, but it is reset by the routines **brk**(2), **malloc**(3), standard input/output (**stdio**(3)), etc. The current value of the program break is reliably returned by 'sbrk(0)', see **brk**(2).

SEE ALSO

brk(2), **malloc**(3).

NAME

execl, execv, execle, execlp, execvp, exec, environ − execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char ****name*, const char *****arg0***, ..., (char *) NULL)**
int execv(const char ****name*, char *const** *argv***[])**
int execle(const char ****name*, const char *****arg0***, ..., (char *) NULL, char *const** *envp***[])**
int execlp(const char ****name*, const char *****arg0***, ..., (char *) NULL)**
int execvp(const char ****name*, char *const** *argv***[])**

extern char *const *environ;

DESCRIPTION

These routines provide various interfaces to the **execve** system call. Refer to **execve**(2) for a description of their properties; only brief descriptions are provided here.

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. **execl** is useful when a known file with known arguments is being called; the arguments to **execl** are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A null pointer argument must end the argument list. (Note that the **execl*** functions are variable argument functions. This means that the type of the arguments beyond *arg0* is not checked. So the null pointer requires an explicit cast to type **(char *)** if not of that type already.)

The **execv** version is useful when the number of arguments is unknown in advance; the arguments to **execv** are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a null pointer.

When a C program is executed, it is called as follows:

int main(int *argc***, char *const** *argv***[], char *const** *envp***[]);**

exit(main(*argc***,** *argv***,** *envp***));**

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is directly usable in another **execv** because *argv*[*argc*] is 0.

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an =, and a null-terminated value. The array of pointers is terminated by a null pointer. The shell **sh**(1) passes an environment entry for each global shell variable defined when the program is called. See **environ**(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell **environ**, which is used by **execv** and **execl** to pass the environment to any subprograms executed by the current program.

Execlp and **execvp** are called with the same arguments as **execl** and **execv**, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment variable **PATH**. Under standard Minix, if a file is found that is executable, but does not have the proper executable header then it is assumed to be a shell script. **Execlp** and **execvp** execute **/bin/sh** to interpret the script. Under Minix-vmd this does not happen, a script must begin with **#!** and the full path name of the interpreter if it is to be an executable script.

SEE ALSO

execve(2), **fork**(2), **environ**(7), **sh**(1).

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see **a.out**(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is −1 and **errno** is set as for **execve**. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

NAME

exit, atexit − terminate a process after flushing any pending output

SYNOPSIS

#include <stdlib.h>

void exit(int *status***)**
int atexit(void (**func***)(void))**

DESCRIPTION

Exit first calls all functions registered by **atexit**, flushes all data buffered by the Standard I/O library, and finally terminates the process. **Exit** never returns.

Atexit registers the function *func* into a table of functions to be called on exit.

SEE ALSO

exit(2).

DIAGNOSTICS

Atexit returns 0 on success, −1 if **malloc** cannot allocate more memory for the list of registered functions.

NAME

fclose, fflush − close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose(FILE ****stream*)**
int fflush(FILE ****stream*)**

DESCRIPTION

Fclose causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

Fclose is performed automatically upon calling **exit**(3).

Fflush causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

SEE ALSO

close(2), **fopen**(3), **setbuf**(3).

DIAGNOSTICS

These routines return **EOF** if *stream* is not associated with an output file, or if buffered data cannot be transferred to that file.

NAME

ferror, feof, clearerr, fileno − stream status inquiries

SYNOPSIS

#include <stdio.h>

int feof(FILE ****stream*)**
int ferror(FILE ****stream*)**
int clearerr(FILE ****stream*)**
int fileno(FILE ****stream*)**

DESCRIPTION

Feof returns non-zero when end of file is read on the named input *stream*, otherwise zero. Unless cleared by **clearerr**, the end-of-file indication lasts until the stream is closed.

Ferror returns non-zero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by **clearerr**, the error indication lasts until the stream is closed.

Clearerr resets the error and end-of-file indicators on the named *stream*.

Fileno returns the integer file descriptor associated with the *stream*, see **open**(2).

Currently all of these functions are implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3), **open**(2).

NAME

fopen, freopen, fdopen − open a stream

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char *_filename_, const char *_type_)
FILE *freopen(const char *_filename_, const char *_type_, FILE *_stream_)
FILE *fdopen(int _fildes_, const char *_type_)

DESCRIPTION

Fopen opens the file named by *filename* and associates a stream with it. **Fopen** returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

"r" open for reading

"w" create for writing

"a" append: open for writing at end of file, or create for writing

In addition, each *type* may be followed by a "+" to have the file opened for reading and writing. "r+" positions the stream at the beginning of the file, "w+" creates or truncates it, and "a+" positions it at the end. Both reads and writes may be used on read/write streams, with the limitation that an **fseek**, **rewind**, or reading an end-of-file must be used between a read and a write or vice-versa.

Freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed.

Freopen is typically used to attach the preopened constant names, **stdin, stdout, stderr,** to specified files.

Fdopen associates a stream with a file descriptor obtained from **open**, **dup**, **creat**, or **pipe**(2). The *type* of the stream must agree with the mode of the open file.

SEE ALSO

open(2), **fclose**(3).

DIAGNOSTICS

Fopen and **freopen** return the pointer **NULL** if *filename* cannot be accessed, if too many files are already open, or if other resources needed cannot be allocated.

BUGS

Fdopen is not portable to systems other than UNIX.

The read/write *types* do not exist on all systems. Those systems without read/write modes will probably treat the *type* as if the "+" was not present. These are unreliable in any event.

NAME

fread, fwrite − buffered binary input/output

SYNOPSIS

```
#include <sys/types.h>
#include <stdio.h>

size_t fread(void *ptr, size_t itemsize, size_t nitems, FILE *stream)
size_t fwrite(void *ptr, size_t itemsize, size_t nitems, FILE *stream)
```

DESCRIPTION

Fread reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

If *stream* is **stdin** and the standard output is line buffered, then any partial output line will be flushed before any call to **read**(2) to satisfy the **fread**.

Fwrite appends at most *nitems* of data of the type of **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

SEE ALSO

read(2), **write**(2), **fopen**(3), **getc**(3), **putc**(3), **gets**(3), **puts**(3), **printf**(3), **scanf**(3).

DIAGNOSTICS

Fread and **fwrite** return 0 upon end of file or error.

NAME

fseek, ftell, rewind − reposition a stream

SYNOPSIS

#include <stdio.h>

int fseek(FILE ****stream*, long** *offset***, int** *ptrname***)**
long ftell(FILE ****stream*)**
void rewind(FILE ****stream*)**

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of **ungetc**(3).

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on UNIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an *offset* for **fseek**.

Rewind(*stream*) is equivalent to **fseek**(*stream*, 0L, 0).

SEE ALSO

lseek(2), **fopen**(3).

DIAGNOSTICS

Fseek returns −1 for improper seeks, otherwise zero.

NAME

getc, getchar, fgetc, getw − get character or word from stream

SYNOPSIS

#include <stdio.h>

int getc(FILE **stream*)
int getchar(void)
int fgetc(FILE **stream*)
int getw(FILE **stream*)

DESCRIPTION

Getc returns the next character from the named input *stream*.

Getchar() is identical to **getc**(**stdin**).

Fgetc behaves like **getc**, but is a genuine function, not a macro; it may be used to save object text.

Getw returns the next **int** from the named input *stream*. It returns the constant **EOF** upon end of file or error, but since that is a good integer value, **feof** and **ferror**(3) should be used to check the success of **getw**. **Getw** assumes no special alignment in the file.

SEE ALSO

clearerr(3), **fopen**(3), **putc**(3), **gets**(3), **scanf**(3), **fread**(3), **ungetc**(3).

DIAGNOSTICS

These functions return the integer constant **EOF** at end of file, upon read error, or if an attempt is made to read a file not opened by **fopen**. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return **EOF** until the condition is cleared with **clearerr**(3).

BUGS

Because it is implemented as a macro, **getc** treats a *stream* argument with side effects incorrectly. In particular, ‘getc(*f++);’ doesn’t work sensibly.

NAME

getcwd − get current working directory pathname

SYNOPSIS

#include <unistd.h>

char *getcwd(char *****pathname*, size_t** *len***)**

DESCRIPTION

Getcwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result. *Pathname* is a character array of length *len*.

DIAGNOSTICS

Getcwd returns a null pointer and sets **errno** if an error occurs. The error will reflect the system call errors that may occur if the path to the current directory is searched upwards to the root directory. The error **ERANGE** is returned if the result does not fit within *len* bytes.

NAME

getenv − value for environment name

SYNOPSIS

#include <stdlib.h>

char *getenv(const char ****name*)**

DESCRIPTION

Getenv searches the environment list (see **environ**(7)) for a string of the form *name=value* and returns a pointer to the string *value* if such a string is present, otherwise **getenv** returns the null pointer.

SEE ALSO

environ(7), **execve**(2).

NAME

getgrent, getgrnam, getgrgid, setgrent, endgrent, setgrfile − group file routines

SYNOPSIS

#include <grp.h>

struct group *getgrent(void)
struct group *getgrnam(const char *_name_)
struct group *getgrgid(gid_t _gid_)
int setgrent(void)
void endgrent(void)
void setgrfile(const char *_file_)

DESCRIPTION

These functions are used to obtain information from the group file. They return this information in a **struct group** as defined by <grp.h>:

```
struct group {
    char    *gr_name;       /* login name */
    char    *gr_passwd;     /* encrypted password */
    gid_t   gr_gid;         /* numeric group id */
    char    **gr_mem;       /* null-terminated list of group members */
};
```

Getgrent() reads the group file entry by entry. **Getgrnam()** scans the entire group file for the group with the given *name*. **Getgrgid()** looks for the first group with the given *gid*. The **setgrent()** and **endgrent()** functions are used to open and later close the group file. With **setgrfile()** one can specify the file to read other than the normal group file. This only sets the name, the next **setgrent()** call will open the file. Do not touch the file name while it is active. Use **setgrfile(NULL)** to revert back to the normal group file.

The usual way to scan the group file is (error checking omitted):

```
setgrent();
while ((gr = getgrent()) != NULL)
        if (appropriate_test(gr)) break;
endgrent();
```

The **gr** variable contains the entry that is wanted if non-NULL. The **getgrnam()** and **getgrgid()** functions are implemented as in this example, with error checking of course.

Getgrent() calls **setgrent()** if this has not yet been done. **Setgrent()** first calls **endgrent()** if the group file is still open. (Other implementations may simply rewind the file.)

FILES

/etc/group The group file database.

SEE ALSO

getgroups(2), **initgroups**(3), **getpwent**(3), **passwd**(5).

DIAGNOSTICS

Setgrent() has the same return value and error codes as the **open**(2) call it uses to open the group file. The **get**_xxx_**()** functions return NULL on end of file, entry not found, or error. You can set **errno** to zero before the call and check it after.

NOTES

All **get**_xxx_**()** routines return a pointer to static storage that is overwritten in each call.

Only **getgrnam()** and **getgrgid()** are defined by POSIX. The **_MINIX_SOURCE** macro must be defined before including <grp.h> to make the other functions visible. The **gr_passwd** field is also not defined by POSIX, but is always visible. Portable code cannot reliably detect errors by setting **errno** to zero. Under Minix it is better to make a **getgrent()** scan if you need to look up several group-id's or names, but portable code had better use several **getgrgid()** or **getgrnam()** calls. The **getgrent()** is

usually available on other systems, but may be very expensive. See **initgroups**(3) if you are after supplementary group id's.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

getlogin − get login name

SYNOPSIS

#include <unistd.h>

char *getlogin(void)

DESCRIPTION

Getlogin returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with **getpwnam** to locate the correct password file entry when the same user ID is shared by several login names.

If **getlogin** is called within a process that is not attached to a terminal, or if there is no entry in **/etc/utmp** for the process's terminal, **getlogin** returns a null pointer. A reasonable procedure for determining the login name is to first call **getlogin** and if it fails, to call **getpwuid**(**getuid**()).

FILES

/etc/utmp

SEE ALSO

getpwent(3), **utmp**(5), **ttyslot**(3)

DIAGNOSTICS

Returns a null pointer if the name cannot be found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getopt − get option letter from argv

SYNOPSIS

```
int getopt(argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind;
```

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option −− may be used to delimit the end of the options; **EOF** will be returned, and −− will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main(argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt(argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case ‘a’:
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case ‘b’:
                        if (aflg)
                                errflg++;
                        else
                                bproc();
                        break;
                case ‘f’:
                        ifile = optarg;
                        break;
```

```
                        case ‘o’:
                                ofile = optarg;
                                break;
                        case ‘?’:
                        default:
                                errflg++;
                                break;
                        }
                if (errflg) {
                        fprintf(stderr, "Usage: ...");
                        exit(2);
                }
                for (; optind < argc; optind++) {
                        .
                        .
                        .
                }
                .
                .
                .
}
```

HISTORY

Written by Henry Spencer, working from a Bell Labs manual page. Modified by Keith Bostic to behave more like the System V version.

BUGS

It is not obvious how ‘−’ standing alone should be treated; this version treats it as a non-option argument, which is not always right.

Option arguments are allowed to begin with ‘−’; this is reasonable but reduces the amount of error checking possible.

Getopt is quite flexible but the obvious price must be paid: there is much it could do that it doesn’t, like checking mutually exclusive options, checking type of option arguments, etc.

NAME

getpass − read a password

SYNOPSIS

#include <minix/minlib.h>

char *getpass(const char **prompt*)

DESCRIPTION

Getpass reads a password from the file **/dev/tty**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 32 characters, excluding the null.

SEE ALSO

crypt(3).

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpwent, getpwnam, getpwuid, setpwent, endpwent, setpwfile − password file routines

SYNOPSIS

#include <pwd.h>

struct passwd *getpwent(void)
struct passwd *getpwnam(const char *_name_)
struct passwd *getpwuid(uid_t _uid_)
int setpwent(void)
void endpwent(void)
void setpwfile(const char *_file_)

DESCRIPTION

These functions are used to obtain information from the password file. They return this information in a **struct passwd** as defined by <pwd.h>:

```
struct passwd {
    char    *pw_name;       /* login name */
    char    *pw_passwd;     /* encrypted password */
    uid_t   pw_uid;         /* numeric user id */
    gid_t   pw_gid;         /* numeric group id */
    char    *pw_gecos;      /* user full name and other info */
    char    *pw_dir;        /* user's home directory */
    char    *pw_shell;      /* name of the user's shell */
};
```

Getpwent() reads the password file entry by entry. **Getpwnam()** scans the entire password file for the user with the given *name*. **Getpwuid()** looks for the first user with the given *uid*. The **setpwent()** and **endpwent()** functions are used to open and later close the password file. With **setpwfile()** one can specify the file to read other than the normal password file. This only sets the name, the next **setpwent()** call will open the file. Do not touch the file name while it is active. Use **setpwfile(NULL)** to revert back to the normal password file.

The usual way to scan the password file is (error checking omitted):

```
setpwent();
while ((pw = getpwent()) != NULL)
        if (appropriate_test(pw)) break;
endpwent();
```

The **pw** variable contains the entry that is wanted if non-NULL. The **getpwnam()** and **getpwuid()** functions are implemented as in this example, with error checking of course.

Getpwent() calls **setpwent()** if this has not yet been done. **Setpwent()** first calls **endpwent()** if the password file is still open. (Other implementations may simply rewind the file.)

FILES

/etc/passwd The password file database.

SEE ALSO

cuserid(3), **getlogin**(3), **getgrent**(3), **passwd**(5).

DIAGNOSTICS

Setpwent() has the same return value and error codes as the **open**(2) call it uses to open the password file. The **get*xxx*()** functions return NULL on end of file, entry not found, or error. You can set **errno** to zero before the call and check it after.

NOTES

All **get*xxx*()** routines return a pointer to static storage that is overwritten in each call.

Only **getpwnam()** and **getpwuid()** are defined by POSIX. The **_MINIX_SOURCE** macro must be defined before including <pwd.h> to make the other functions visible. The **pw_passwd** and **pw_gecos**

fields are also not defined by POSIX, but are always visible. Portable code cannot reliably detect errors by setting **errno** to zero. Under Minix it is better to make a **getpwent()** scan if you need to look up several user-id's or names, but portable code had better use several **getpwuid()** or **getpwnam()** calls. The **getpwent()** is usually available on other systems, but may be very expensive.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

gets, fgets − get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets(char *_s_)
char *fgets(char *_s_, int _n_, FILE *_stream_)

DESCRIPTION

Gets reads a string into *s* from the standard input stream **stdin**. The string is terminated by a newline character, which is replaced in *s* by a null character. **Gets** returns its argument.

Fgets reads *n*−1 characters, or up through a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. **Fgets** returns its first argument.

SEE ALSO

puts(3), **getc**(3), **scanf**(3), **fread**(3), **ferror**(3).

DIAGNOSTICS

Gets and **fgets** return the constant pointer **NULL** upon end of file or error.

BUGS

Gets deletes a newline, **fgets** keeps it, all in the name of backward compatibility.

Gets is not present in the Minix-vmd C library for reasons that should be obvious. Use **fgets** instead.

NAME

getttyent, getttynam, setttyent, endttyent − interface to /etc/ttytab

SYNOPSIS

#include <ttyent.h>

struct ttyent *getttyent(void)
struct ttyent *getttynam(const char ***name*)**
int setttyent(void)
void endttyent(void)

DESCRIPTION

The **getttyent** functions provide an interface to the /etc/ttytab. (See **ttytab**(5)).

To read one of these files one calls **getttyent()** several times to read the entries in the table until NULL is returned for end-of-file.

Getttyname() searches the **ttytab** file for the given terminal device. It is equivalent to a call to **setttyent(),** several calls to **getttyent()** to locate the entry, and a final **endttyent()** to close the file.

Setttyent() opens or rewinds the ttytab database, and **endttyent() closes it. Getttyent()** opens the database if not already open, but does not close it.

The struct ttyent is defined by <ttyent.h> as follows:

```
struct ttyent {
    char    *ty_name;       /* Name of the terminal device. */
    char    *ty_type;       /* Terminal type name (termcap(3)). */
    char    **ty_getty;     /* Program to run, normally getty. */
    char    **ty_init;      /* Initialization command, normally stty. */
};
```

A valid entry has at least two strings, so both **ty_name** and **ty_type** are filled in. The optional **ty_getty** and **ty_init** may be NULL (field omitted), point to a pointer that is NULL (null lenght field, i.e. ""), or an array of strings terminated by a NULL (field present). For now no useful distinction can be made between a omitted field and an empty field, so treat both cases as an omission.

FILES

/etc/ttytab The terminal device database

SEE ALSO

ttyname(3), **ttyslot**(3), **ttytab**(5), **init**(8).

DIAGNOSTICS

Setttyent() has the same return value and error codes as the **open**(2) call it uses to open the ttytab file. The **get***xxx***()** functions return NULL on end of file, entry not found, or error. You can set **errno** to zero before the call and check it after.

NOTES

Getttyent() and **getttynam()** return a pointer to static storage that is overwritten in each call.

The Minix **struct ttyent** has only the **ty_name** and **ty_type** fields in common with the BSD implementation. This does not seem to be a problem, because most third party software that need to know about terminals only look at the **ty_name** field.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

g_h_b_n, gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent, herror − get network host entry

SYNOPSIS

#include <net/gen/netdb.h>

extern int h_errno;

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

struct hostent *gethostent()

sethostent(stayopen)
int stayopen;

endhostent()

herror(string)
char *string;

DESCRIPTION

Gethostbyname and *gethostbyaddr* each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, *named*(8), or broken-out fields from a line in */etc/hosts*. If the local name server is not running these routines do a lookup in */etc/hosts*.

```
struct    hostent {
          char    *h_name;        /* official name of host */
          char    **h_aliases;    /* alias list */
          int     h_addrtype;     /* host address type */
          int     h_length;       /* length of address */
          char    **h_addr_list;  /* list of addresses from name server */
};
#define   h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in h_addr_list; this is for backward compatiblity.

When using the nameserver, *gethostbyname* will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable ''HOSTALAIASES'' contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See *hostname*(7) for the domain search procedure and the alias file format.

Sethostent may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*. Otherwise, queries are performed using UDP datagrams.

Endhostent closes the TCP connection.

DIAGNOSTICS

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine *herror* can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

h_errno can have the following values:

HOST_NOT_FOUND	No such host is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error.
NO_DATA	The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

FILES

/etc/hosts

SEE ALSO

resolver(3), hosts(5), hostname(7), named(8)

CAVEAT

Gethostent is defined, and *sethostent* and *endhostent* are redefined, when *libc* is built to use only the routines to lookup in */etc/hosts* and not the name server.

Gethostent reads the next line of */etc/hosts*, opening the file if necessary.

Sethostent is redefined to open and rewind the file. If the *stayopen* argument is non-zero, the hosts data base will not be closed after each call to *gethostbyname* or *gethostbyaddr*. *Endhostent* is redefined to close the file.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

hton, htons, htonl, ntohs, ntohl − host to network byte order conversion

SYNOPSIS

```
#define _MINIX_SOURCE 1
#include <stddef.h>
#include <sys/types.h>

#include <net/hton.h>

u16_t htons(u16_t host_word)
u32_t htonl(u32_t host_dword)
u16_t ntohs(u16_t network_word)
u32_t ntohl(u32_t network_dword)
u16_t HTONS(u16_t host_word)
u32_t HTONL(u32_t host_dword)
u16_t NTOHS(u16_t network_word)
u32_t NTOHL(u32_t network_dword)
```

DESCRIPTION

These macros convert 16-bit and 32-bit quantities to and from the network byte order used by the TCP/IP protocols. The function of the macros is encoded in their name. **H** means host byte order, **n** means network byte order, **s** means a 16-bit quantity and **l** means a 32-bit quantity. Thus **htons** converts a 16-bit quantity from host byte order to network byte order. The difference between the lower case and upper case variants is that the lower case variants evaluate the argument at most once and the upper case variants can be used for constant folding. That is,

```
htonl(f(x))
```

will call f(x) at most once and

```
HTONS(0x10)
```

will be equivalent to 0x10 on a big-endian machine and 0x1000 on a little-endian machine.

SEE ALSO

ip(4).

AUTHOR

Philip Homburg (philip@cs.vu.nl)

NAME

malloc, free, realloc, calloc, alloca − memory allocator

SYNOPSIS

#include <sys/types.h>
#include <stdlib.h>
#include <alloca.h>

void *malloc(size_t *size***)**
void free(void ***ptr*)**
void *realloc(void ***ptr*, size_t** *size***)**
void *calloc(size_t *nelem***, size_t** *elsize***)**
void *alloca(size_t *size***)**

DESCRIPTION

Malloc and **free** provide a general-purpose memory allocation package. **Malloc** returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to **free** is a pointer to a block previously allocated by **malloc**; this space is made available for further allocation, but its contents are left undisturbed. A call with a null *ptr* is legal and does nothing.

Needless to say, grave disorder will result if the space assigned by **malloc** is overrun or if some random number is handed to **free**.

Malloc maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls **sbrk** (see **brk**(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. A call with a null *ptr* is legal and has the same result as **malloc(***size***)**.

Calloc allocates space for an array of *nelem* elements of size *elsize.* The space is initialized to zeros.

Alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2).

DIAGNOSTICS

Malloc, **realloc** and **calloc** return a null pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

NOTES

Other implementations of **malloc**, **realloc** or **calloc** may return a null pointer if the size of the requested block is zero. This implementation will always return a zero length block at a unique address, but you should keep in mind that a null return is possible if the program is run to another system and that this should not be mistakenly seen as an error.

BUGS

When **realloc** returns a null pointer, the block pointed to by *ptr* may be destroyed.

Alloca is machine dependent; its use is discouraged.

NAME

oneC_sum − One's complement internet checksum

SYNOPSIS

```
#define _MINIX_SOURCE 1
#include <stddef.h>
#include <sys/types.h>

#include <net/gen/oneCsum.h>

u16_t oneC_sum(u16_t prev, u16_t *data, size_t size)
```

DESCRIPTION

OneC_sum is used to calculate the one's complement checksum needed for IP network packets. A good document about the Internet Checksum is RFC-1071 (Computing the Internet checksum).

One_Csum expects three parameters:

prev The checksum of previous blocks of data that are to be included in the checksum. The value of prev in first call to oneC_sum should be 0.

data A pointer to the block of data. The data is interpreted as a series of 16 bit numbers in network byte order, but an odd number of bytes is also allowed.

size The size of the data in bytes.

SEE ALSO

ip(4).
RFC-1071

AUTHOR

Philip Homburg (philip@cs.vu.nl)

NAME

popen, pclose − initiate I/O to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen(const char *command, const char *type)
int pclose(FILE *stream)

DESCRIPTION

The arguments to **popen** are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by **popen** should be closed by **pclose**, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

SEE ALSO

pipe(2), **fopen**(3), **fclose**(3), **system**(3), **wait**(2), **sh**(1).

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or the shell cannot be accessed.

BUGS

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, for instance, with **fflush**, see **fclose**(3).

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf − formatted output conversion

SYNOPSIS

```
#include <sys/types.h>
#include <stdio.h>
#include <stdarg.h>

int printf(const char *format [, arg] ...);
int fprintf(FILE *stream, const char *format [, arg] ...);
int sprintf(char *s, const char *format [, arg] ...);
int snprintf(char *s, size_t n, const char *format [, arg] ...);
int vprintf(const char *format, va_list args);
int vfprintf(FILE *stream, const char *format, va_list args);
int vsprintf(char *s, const char *format, va_list args);
int vsnprintf(char *s, size_t n, const char *format, va_list args);
```

DESCRIPTION

Printf places output on the standard output stream **stdout**. **Fprintf** places output on the named output *stream*. **Sprintf** places 'output' in the string *s*, followed by the character '\0'. **Snprintf** (Minix-vmd only) is like **sprintf** except that no more than *n*−1 characters are written to *s* followed by a '\0'.

The **v*printf** functions can be used to make functions like the first four by using the **stdarg**(3) method to process the argument.

Each of these functions converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg*.

Each conversion specification is introduced by the character **%**. The remainder of the conversion specification includes in the following order

- Zero or more of following flags:
  - a '#' character specifying that the value should be converted to an ''alternate form''. For **c**, **d**, **s**, and **u** conversions, this option has no effect. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero. For **x**(**X**) conversion, a non-zero result has the string **0x**(**0X**) prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point only appears in the results of those conversions if a digit follows the decimal point). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be.
  - a minus sign '−' which specifies *left adjustment* of the converted value in the indicated field;
  - a '+' character specifying that there should always be a sign placed before the number when using signed conversions.
  - a space specifying that a blank should be left before a positive number during a signed conversion. A '+' overrides a space if both are used.
- an optional digit string specifying a *field width;* if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- an optional period '**.**' which serves to separate the field width from the next digit string;
- an optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;

- the character **l** specifying that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg*.
- a character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer *arg* supplies the field width or precision.

The conversion characters and their meanings are

dox The integer *arg* is converted to decimal, octal, or hexadecimal notation respectively.

X Like **x**, but use upper case instead of lower case.

f The float or double *arg* is converted to decimal notation in the style '[−]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e The float or double *arg* is converted in the style '[−]d**.**ddd**e**±dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double *arg* is printed in style **d**, in style **f**, or in style **e**, whichever gives full precision in minimum space.

c The character *arg* is printed.

s *Arg* is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer *arg* is converted to decimal and printed.

% Print a '%'; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by **printf** are printed by **putc**(3).

Examples

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

To print π to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

putc(3), **scanf**(3), **ecvt**(3), **stdarg**(3).

NAME

putc, putchar, fputc, putw − put character or word on a stream

SYNOPSIS

#include <stdio.h>

int putc(int *c***, FILE *****stream***)**
int putchar(int *c***)**
int fputc(int *c***, FILE *****stream***)**
int putw(int *w***, FILE *****stream***)**

DESCRIPTION

Putc appends the character *c* to the named output *stream*. It returns the character written.

Putchar(*c***)** is defined as **putc(***c***, stdout)**.

Fputc behaves like **putc**, but is a genuine function rather than a macro.

Putw appends word (that is, **int**) *w* to the output *stream*. It returns the word written. **Putw** neither assumes nor causes special alignment in the file.

SEE ALSO

fopen(3), **fclose**(3), **getc**(3), **puts**(3), **printf**(3), **fread**(3).

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, **ferror**(3) should be used to detect **putw** errors.

BUGS

Because it is implemented as a macro, **putc** treats a *stream* argument with side effects improperly. In particular ‘putc(c, *f++);’ doesn’t work sensibly.

Errors can occur long after the call to **putc**.

NAME

puts, fputs − put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts(char ****s*)**
int fputs(char ****s*, FILE ****stream***)**

DESCRIPTION

Puts copies the null-terminated string *s* to the standard output stream **stdout** and appends a newline character.

Fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

SEE ALSO

fopen(3), **gets**(3), **putc**(3), **printf**(3), **ferror**(3), **fread**(3).

BUGS

Puts appends a newline, **fputs** does not, all in the name of backward compatibility.

NAME

qsort − quicker sort

SYNOPSIS

#include <sys/types.h>
#include <stdlib.h>

void qsort(void **base*, size_t *nel*, size_t *width*, int (compar*)(const void *, const void *))**

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1).

NAME

rand, srand − random number generator

SYNOPSIS

#include <stdlib.h>

void srand(unsigned *seed***)**
unsigned rand(void)

DESCRIPTION

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to **RAND_MAX**.

The generator is reinitialized by calling **srand** with 1 as argument. It can be set to a random starting point by calling **srand** with whatever you like as argument.

SEE ALSO

random(3).

NAME

random, srandom, initstate, setstate − better random number generator; routines for changing generators

SYNOPSIS

#include <stdlib.h>

long random(void)
void srandom(unsigned *seed***)**
char *initstate(unsigned *seed***, char *****state***, int** *n***)**
char *setstate(char *****state*)**

DESCRIPTION

Random uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random number generator is very large, approximately $16\times(2^{31}-1)$.

Random/srandom have (almost) the same calling sequence and initialization properties as **rand/srand.** The difference is that **rand**(3) produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by **random** are usable. For example, ‘‘random()&01’’ will produce a random binary value.

Unlike **srand**, **srandom** does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like **rand**(3), however, **random** will by default produce a sequence of numbers that can be duplicated by calling **srandom** with **1** as the seed.

The **initstate** routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by **initstate** to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. **Initstate** returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate** routine provides for rapid switching between states. **Setstate** returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to **initstate** or **setstate**.

Once a state array has been initialized, it may be restarted at a different point either by calling **initstate** (with the desired seed, the state array, and its size) or by calling both **setstate** (with the state array) and **srandom** (with the desired seed). The advantage of calling both **setstate** and **srandom** is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

AUTHOR

Earl T. Cohen

DIAGNOSTICS

If **initstate** is called with less than 8 bytes of state information, or if **setstate** detects that the state information has been garbled, error messages are printed on the standard error output.

SEE ALSO

rand(3).

NOTES

initstate and **setstate** are not declared in *<stdlib.h>*, programmers must provide their own declarations.

BUGS

About 2/3 the speed of **rand**(3).

NAME

resolver, res_query, res_search, res_mkquery, res_send, res_init, dn_comp, dn_expand − resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <net/gen/in.h>
#include <net/gen/nameser.h>
#include <net/gen/resolv.h>

res_query(dname, class, type, answer, anslen)
char *dname;
int class, type;
u_char *answer;
int anslen;

res_search(dname, class, type, answer, anslen)
char *dname;
int class, type;
u_char *answer;
int anslen;

res_mkquery(op, dname, class, type, data, datalen, newrr, buf, buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;

res_send(msg, msglen, answer, anslen)
char *msg;
int msglen;
char *answer;
int anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn;
int length;
char **dnptrs, **lastdnptr;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, exp_dn;
int length;
```

DESCRIPTION

These routines are used for making, sending and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information that is used by the resolver routines is kept in the structure *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in *resolv.h* and are as follows. Options are stored as a simple bit mask containing the bitwise ‘‘or’’ of the options enabled.

RES_INIT

True if the initial name server address and default domain name are initialized (i.e., *res_init* has been called).

RES_DEBUG
: Print debugging messages.

RES_AAONLY
: Accept authoritative answers only. With this option, *res_send* should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
: Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
: Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
: Unused currently (ignore truncation errors, i.e., don't retry with TCP).

RES_RECURSE
: Set the recursion-desired bit in queries. This is the default. (*res_send* does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
: If set, *res_search* will append the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH
: If this option is set, *res_search* will search for host names in the current domain and in parent domains; see *hostname*(7). This is used by the standard host lookup routine *gethostbyname*(3). This option is enabled by default.

The *res_init* routine reads the configuration file (if any; see *resolver*(5)) to get the default domain name, search list and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable LOCALDOMAIN. Initialization normally occurs on the first call to one of the following routines.

The *res_query* function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname* . The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

The *res_search* routine makes a query and awaits a response like *res_query*, but in addition, it implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply.

The remaining routines are lower-level routines used by *res_query*. The *res_mkquery* function constructs a standard query message and places it in *buf*. It returns the size of the query, or −1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any of the query types defined in <*arpa/nameser.h*>. The domain name for the query is given by *dname*. *Newrr* is currently unused but is intended for making update messages.

The *res_send* routine sends a pre-formatted query and returns an answer. It will call *res_init* if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or −1 if there were errors.

The *dn_comp* function compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or −1 if there were errors. The size of the array pointed to by *comp_dn* is given by *length*. The compression uses an array of pointers *dnptrs* to previously-compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*. A side effect of *dn_comp* is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

The *dn_expand* entry expands the compressed domain name *comp_dn* to a full domain name The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of compressed name is returned or −1 if there was an error.

FILES

/etc/resolv.conf see resolver(5)

SEE ALSO

gethostbyname(3), named(8), resolver(5), hostname(7),
RFC1032, RFC1033, RFC1034, RFC1035, RFC974,
SMM:11 Name Server Operations Guide for BIND

NAME

scanf, fscanf, sscanf, vscanf, vfscanf, vsscanf − formatted input conversion

SYNOPSIS

#include <stdio.h>
#include <stdarg.h>

int scanf(const char ****format* [,** *pointer*] ...**)**
int fscanf(FILE ****stream*, const char ****format* [**,** *pointer*] ...**)**
int sscanf(const char ****s*, const char ****format* [**,** *pointer*] ...**)**
int vscanf(const char ****format*, va_list** *args***)**
int vfscanf(FILE ****stream*, const char ****format***, va_list** *args***)**
int vsscanf(const char ****s*, const char ****format***, va_list** *args***)**

DESCRIPTION

Scanf reads from the standard input stream **stdin**. **Fscanf** reads from the named input *stream*. **Sscanf** reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The **v*scanf** functions can be used to make functions like the first three by using the **stdarg**(3) method to process the argument pointers.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.
2. An ordinary character (not %) which must match the next character of the input stream.
3. Conversion specifications, consisting of the character **%**, an optional assignment suppressing character *****, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *****. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

% a single '%' is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

o an octal integer is expected; the corresponding argument should be a integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try '%1s'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

efg a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a **float**. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until

the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o** and **x** may be capitalized or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f** or **g** may be capitalized or preceded by **l** to indicate a pointer to **double** rather than to **float**. The conversion characters **d**, **o** and **x** may be preceded by **h** to indicate a pointer to **short** rather than to **int**.

The **scanf** functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E−1  thompson
```

will assign to **i** the value 25, **x** the value 5.432, and **name** will contain '**thompson\0**' . Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to **i**, 789.0 to **x**, skip '0123', and place the string '56\0' in **name**. The next call to **getchar** will return 'a'.

SEE ALSO

atof(3), **getc**(3), **printf**(3), **stdarg**(3).

DIAGNOSTICS

The **scanf** functions return **EOF** on end of input, and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf, setvbuf − assign buffering to a stream

SYNOPSIS

#include <stdio.h>

int setbuf(FILE ****stream*, char *****buf***)**
int setvbuf(FILE *****stream*, char *****buf***, int** *type***, size_t** *size***)**

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is encountered or input is read from stdin. **Fflush** (see **fclose**(3)) may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from **malloc**(3) upon the first **getc** or **putc**(3) on the file. If the standard stream **stdout** refers to a terminal it is line buffered. The standard stream **stderr** is always unbuffered.

Setbuf is used after a stream has been opened but before it is read or written. The character array *buf* is used instead of an automatically allocated buffer. If *buf* is the constant pointer **NULL**, input/output will be completely unbuffered. A manifest constant **BUFSIZ** tells how big an array is needed:

```
char buf[BUFSIZ];
```

Setvbuf, an alternate form of **setbuf**, is used after a stream has been opened but before it is read or written. It has three uses, depending on the value of the *type* argument:

setvbuf(*stream***,** *buf***, _IOFBF,** *size***)**
Causes input/output to be fully buffered using the character array *buf* whose size is determined by the *size* argument. If *buf* is the constant pointer **NULL**, then an automatically allocated buffer will be used.

setvbuf(*stream***,** *buf***, _IOLBF,** *size***)**
Like above, except that output will be line buffered, i.e. the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

setvbuf(*stream***,** *buf***, _IONBF,** *size***)**
Causes input/output to be completely unbuffered. *Buf* and *size* are ignored.

A file can be changed between unbuffered, line buffered, or block buffered by using **freopen** (see **fopen**(3)) followed by the appropriate **setvbuf** call.

SEE ALSO

fopen(3), **getc**(3), **putc**(3), **malloc**(3), **fclose**(3), **puts**(3), **printf**(3), **fread**(3).

NAME

sigset, sigaddset, sigdelset, sigemptyset, sigfillset, sigismember − manipulate signal sets

SYNOPSIS

#include <signal.h>

int sigaddset(sigset_t *_set_, int _sig_)
int sigdelset(sigset_t *_set_, int _sig_)
int sigemptyset(sigset_t *_set_)
int sigfillset(sigset_t *_set_)
int sigismember(const sigset_t *_set_, int _sig_)

DESCRIPTION

The system calls that handle signals, such as **sigaction**(2) and **sigprocmask**(2) use sets of signals to keep a process from being interrupted by those signals while executing a signal handler or a critical code segment. These signal sets are manipulated by the following functions:

int sigaddset(sigset_t *_set_, int _sig_)
Add signal *sig* to the signal set referenced by *set*.

int sigdelset(sigset_t *_set_, int _sig_)
Remove signal *sig* from the signal set referenced by *set*.

int sigemptyset(sigset_t *_set_)
Initialize the signal set referenced by *set* to an empty set.

int sigfillset(sigset_t *_set_)
Initialize the signal set referenced by *set* to an full set, i.e. all signals are in the set.

int sigismember(const sigset_t *_set_, int _sig_)
Return **1** if the signal *sig* is present in the set referenced by *set*, **0** otherwise.

SEE ALSO

sigaction(2), **sigpending**(2), **sigprocmask**(2), **sigsuspend**(2).

DIAGNOSTICS

All functions except **sigismember** return **0** on success. **Sigismember** returns **0** or **1** on success. They return **−1** with error code **EINVAL** for an invalid signal number. (They do not use **EFAULT** for a bad *set* address, but will simply cause a segmentation violation.)

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

sleep − suspend execution for interval

SYNOPSIS

#include <unistd.h>

unsigned int sleep(unsigned int *seconds***)**

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument.

The routine is implemented by setting an alarm timer and pausing until it occurs. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous timer, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

SEE ALSO

alarm(2), **pause**(2).

NAME

stdarg − variable argument list

SYNOPSIS

#include <stdarg.h>

void va_start(va_list *ap***,** *argtypeN parmN***)**
type **va_arg(va_list** *ap***,** *type***)**
void va_end(va_list *ap***)**

DESCRIPTION

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists (such as **printf**(3)) that do not use **stdarg** are inherently non-portable, since different machines use different argument passing conventions.

A function that accepts a variable argument list is declared with "..." at the end of its parameter list. It must have at least one normal argument before the "...". For example:

```
int printf(const char *format, ...) { /* code */ }
int fprintf(FILE *stream, const char *format, ...) { /* code */ }
```

va_list is a type which is used for the variable *ap* within the body of a variable argument function which is used to traverse the list.

va_start(*ap*, *parmN*) is called to initialize *ap* to the beginning of the list. The last true parameter of the function, *parmN*, must be supplied to allow **va_start** to compute the address of the first variable parameter.

va_arg(*ap*, *type*) will return the next argument in the list pointed to by *ap*. *Type* is the type to which the expected argument will be converted when passed as an argument.

Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(*ap*) must be used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end,** are possible.

EXAMPLE

```
#include <stdarg.h>

execl(const char *path, ...)
{
    va_list ap;
    char *args[100];
    int argno = 0;

    va_start(ap, path);
    while ((args[argno++] = va_arg(ap, char *)) != NULL) {}
    va_end(ap);
    return execv(path, args);
}
```

NOTES

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, **execl** passes a null pointer to signal the end of the list. **Printf** can tell how many arguments are supposed to be there by the format.

The macros **va_start** and **va_end** may be arbitrarily complex; for example, **va_start** might contain an opening brace, which is closed by a matching brace in **va_end**. Thus, they should only be used where they could be placed within a single complex statement.

BUGS

It is impossible to properly show the macros as C declarations as is done in the synopsis. They can

never be coded as C functions, because all three macros use their arguments by address, and the *type* field is certainly impossible. Just look at them as being part of the C language, like **sizeof**.

NAME

stdio − standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

DESCRIPTION

The functions in the standard I/O library constitute a user-level buffering scheme. The in-line macros **getc** and **putc**(3) handle characters quickly. The higher level routines **gets**, **fgets**, **scanf**, **fscanf**, **fread**, **puts**, **fputs**, **printf**, **fprintf**, **fwrite** all use **getc** and **putc**; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **FILE**. **Fopen**(3) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant 'pointer' **NULL** (0) designates no stream at all.

An integer constant **EOF** (−1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file *<stdio.h>* of pertinent macro definitions. The functions and constants mentioned in the standard I/O manual pages are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: **clearerr**, **getc**, **getchar**, **putc**, **putchar**, **feof**, **ferror**, **fileno**.

SEE ALSO

open(2), **close**(2), **read**(2), **write**(2), **fclose**(3), **ferror**(3), **fopen**(3), **fread**(3), **fseek**(3), **getc**(3), **gets**(3), **printf**(3), **putc**(3), **puts**(3), **scanf**(3), **setbuf**(3), **ungetc**(3).

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **FILE** pointer has not been initialized with **fopen**, input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a **read**(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard i/o routines but use **read**(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to **fflush**(3) the standard output before going off and computing so that the output will appear.

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially **fork** and **abort**.

LIST OF FUNCTIONS

Name	Appears on Page	Description
clearerr	ferror(3)	stream status inquiries
fclose	fclose(3)	close or flush a stream
fdopen	fopen(3)	open a stream

feof	ferror(3)	stream status inquiries
ferror	ferror(3)	stream status inquiries
fflush	fclose(3)	close or flush a stream
fgetc	getc(3)	get character or word from stream
fgets	gets(3)	get a string from a stream
fileno	ferror(3)	stream status inquiries
fopen	fopen(3)	open a stream
fprintf	printf(3)	formatted output conversion
fputc	putc(3)	put character or word on a stream
fputs	puts(3)	put a string on a stream
fread	fread(3)	buffered binary input/output
freopen	fopen(3)	open a stream
fscanf	scanf(3)	formatted input conversion
fseek	fseek(3)	reposition a stream
ftell	fseek(3)	reposition a stream
fwrite	fread(3)	buffered binary input/output
getc	getc(3)	get character or word from stream
getchar	getc(3)	get character or word from stream
gets	gets(3)	get a string from a stream
getw	getc(3)	get character or word from stream
printf	printf(3)	formatted output conversion
putc	putc(3)	put character or word on a stream
putchar	putc(3)	put character or word on a stream
puts	puts(3)	put a string on a stream
putw	putc(3)	put character or word on a stream
rewind	fseek(3)	reposition a stream
scanf	scanf(3)	formatted input conversion
setbuf	setbuf(3)	assign buffering to a stream
setvbuf	setbuf(3)	assign buffering to a stream
snprintf	printf(3)	formatted output conversion
sprintf	printf(3)	formatted output conversion
sscanf	scanf(3)	formatted input conversion
ungetc	ungetc(3)	push character back into input stream
vfprintf	printf(3)	formatted output conversion
vfscanf	scanf(3)	formatted input conversion
vprintf	printf(3)	formatted output conversion
vscanf	scanf(3)	formatted input conversion
vsnprintf	printf(3)	formatted output conversion
vsprintf	printf(3)	formatted output conversion
vsscanf	scanf(3)	formatted input conversion

NAME

string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex − string operations string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strerror, memcmp, memcpy, memmove, memchr, memset, index, rindex − string operations

SYNOPSIS

#include <sys/types.h>
#include <strings.h>

char *strcat(char ****s1*, const char ****s2***)**
char *strncat(char ****s1*, const char ****s2***, size_t** *n***)**
int strcmp(const char ****s1*, const char ****s2***)**
int strncmp(const char ****s1*, const char ****s2***, size_t** *n***)**
char *strcpy(char ****s1*, const char ****s2***)**
char *strncpy(char ****s1*, const char ****s2***, size_t** *n***)**
size_t strlen(const char ****s*)**
char *strchr(const char ****s*, int** *c***)**
char *strrchr(const char ****s*, int** *c***)**
char *strerror(int *errnum***)**
int memcmp(const void ****s1*, const void ****s2***, size_t** *n***)**
void *memcpy(void ****s1*, const void ****s2***, size_t** *n***)**
void *memmove(void ****s1*, const void ****s2***, size_t** *n***)**
void *memchr(const void ****s*, int** *c***, size_t** *n***)**
void *memset(void ****s*, int** *c***, size_t** *n***)**
char *index(const char ****s*, int** *c***)**
char *rindex(const char ****s*, int** *c***)**

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string *s2* to the end of string *s1*. **Strncat** copies at most *n* characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. **Strncmp** makes the same comparison but looks at at most *n* characters.

Strcpy copies string *s2* to *s1*, stopping after the null character has been moved. **Strncpy** copies exactly *n* characters, truncating or null-padding *s2;* the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

Strlen returns the number of non-null characters in *s*.

Strchr (**strrchr**) returns a pointer to the first (last) occurrence of character *c* in string *s*, or null if *c* does not occur in the string.

Strerror returns the error string for the system call error *errnum*. See **intro**(2).

Memcmp is like **strcmp** except that the strings are memory blocks of length *n*. Null characters are treated as ordinary characters.

Memcpy copies *n* bytes from the location pointed to by *s2* to *s1*. **Memmove** is like memcpy, except that it can handle overlap between the two strings. Both functions return *s1*.

Memchr returns a pointer to the first occurrence of character *c* in string *s*, or null if *c* does not occur in the string.

Memset sets *n* bytes to *c* starting at location *s*. It returns *s*.

Index and **rindex** are obsolete versions of **strchr** and **strrchr**. New code should avoid using them.

NOTES

Characters are compared as **unsigned char**, whether **char** itself is signed or not.

NAME

system − issue a shell command

SYNOPSIS

#include <stdlib.h>

int system(const char **string*)

DESCRIPTION

System causes the *string* to be given to **sh**(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

sh(1), **popen**(3), **execve**(2), **wait**(2).

DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

NAME

termcap, tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs − terminal independent operation routines

SYNOPSIS

#include <termcap.h>

int tgetent(char *_bp_, char *_name_)
int tgetflag(char *_id_)
int tgetnum(char *_id_)
char *tgetstr(char *_id_, char **_area_)
char *tgoto(char *_cm_, int _destcol_, int _destline_)
int tputs(char *_cp_, int _affcnt_, void (*_outc_)(int))

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base **termcap**(5). These are low level routines; see **curses**(3) for a higher level package.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to **tgetnum**, **tgetflag**, and **tgetstr**. **Tgetent** returns −1 if it cannot find a termcap file, 0 if the terminal name given does not have an entry, and 1 if all goes well.

Tgetent uses the following recipe to find the termcap file and entry *name*:

```
if $TERMCAP is itself a termcap entry for name
then
        use $TERMCAP
elif $TERMCAP names a file
then
        use entry name found in that file
elif this is Minix-vmd
then
        if $TERMPATH is defined
        then
                search the termcap files named in $TERMPATH for the first occurance of a name
                entry and use that entry
        else
                the path $HOME/.termcap:/etc/termcap:/usr/etc/termcap" is searched for entry
                name
        fi
fi
```

Tgetnum gets the numeric value of capability *id*, returning −1 if is not given for the terminal. **Tgetflag** returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. **Tgetstr** returns the string value of the capability *id*, places it in the buffer at *area*, and advances the *area* pointer. It decodes the abbreviations for this field described in **termcap**(5), except for cursor addressing and padding information. **Tgetstr** returns NULL if the capability was not found.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since **tgoto** may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use CTRL-I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then **tgoto** returns OOPS.

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable **ospeed** should contain the output speed of the terminal as encoded by **stty**(3). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

SEE ALSO

curses(3), **termcap**(5).

AUTHOR

William Joy

NOTES

The Minix implementation does not support any of the external variables, only the functions calls. The Minix-vmd termcap does support it all, although noone in his right mind meddles with those variables.

NAME

termios, tcgetattr, tcsetattr, cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed, tcsendbreak, tcdrain, tcflush, tcflow − change terminal attributes

SYNOPSIS

#include <termios.h>

int tcgetattr(int *fd***, struct termios ****tp***)**
int tcsetattr(int *fd***, int** *action***, const struct termios ****tp***)**

speed_t cfgetispeed(const struct termios ****tp*)**
speed_t cfgetospeed(const struct termios ****tp*)**
int cfsetispeed(struct termios ****tp*, speed_t** *speed***)**
int cfsetospeed(struct termios ****tp*, speed_t** *speed***)**

int tcsendbreak(int *fd***, int** *duration***)**
int tcdrain(int *fd***)**
int tcflush(int *fd***, int** *queue_selector***)**
int tcflow(int *fd***, int** *action***)**

DESCRIPTION

These are the user functions that modify the tty attributes mentioned in **tty**(4). In the following, *fd* refers to an open terminal device file, *tp* is the address of a **struct termios**, and *speed* and values of type **speed_t** are equal to one of the **B0**, **B50**, etc. baud rate symbols. All functions, symbols, and types are declared in **<termios.h>**.

The effects of the tty functions are:

tcgetattr(*fd***,** *tp***)**
Get the current settings of the tty attributes.

tcsetattr(*fd***, TCSANOW,** *tp***)**
Set the terminal attributes. The change occurs immediately.

tcsetattr(*fd***, TCSADRAIN,** *tp***)**
Set the terminal attributes. The change occurs once all the output waiting in the output queues has been transmitted. This should be used when options affecting output are changed.

tcsetattr(*fd***, TCSAFLUSH,** *tp***)**
Set the terminal attributes. But first wait until all the output waiting in the output queues has been transmitted. All input waiting in the input queues is then discarded and the change is made. This should be used when switching from canonical to non-canonical mode or vice-versa. (Oddly enough, this is seldom what you want, because it discards typeahead. An editing shell does the Right Thing if it uses **TCSANOW** instead. POSIX may not guarantee good results, but in practice most systems make the canonical input available in raw mode.)

cfgetispeed(*tp***)**
Return the input baud rate encoded in the termios structure.

cfgetospeed(*tp***)**
Return the output baud rate encoded in the termios structure.

cfsetispeed(*tp***,** *speed***)**
Encode the new input baud rate into the termios structure.

cfsetospeed(*tp***,** *speed***)**
Encode the new output baud rate into the termios structure.

tcsendbreak(*fd***,** *duration***)**
Emit a break condition on a serial line for a time indicated by *duration*. (Always 0.4 seconds under Minix, *duration* is ignored.)

tcdrain(*fd***)**
Wait until all output waiting in the output queues has been transmitted.

tcflush(*fd***, TCIFLUSH)**
Flush the input queue. (I.e. discard it.)

tcflush(*fd***, TCOFLUSH)**
Flush the output queue.

tcflush(*fd***, TCIOFLUSH)**
Flush the input and output queues.

tcflow(*fd***, TCOOFF)**
Suspend output. (Like the effect of **STOP**.)

tcflow(*fd***, TCOON)**
Restart output. (Like the effect of **START**.)

tcflow(*fd***, TCIOFF)**
Transmit a **STOP** character intended to make the remote device stop transmitting data.

tcflow(*fd***, TCION)**
Transmit a **START** character to restart the remote device.

SEE ALSO

stty(1), **tty**(4).

DIAGNOSTICS

All functions return 0 unless otherwise specified, and −1 on error with **errno** set to indicate the type of error. The most notable errors are **ENOTTY** if *fd* does not refer to a terminal device, and **EINTR** if one of the functions waiting for output to drain is interrupted.

NOTES

It may be interesting to know that the functions operating on the tty are directly translated into the following Minix **ioctl** requests: **TCGETS**, **TCSETS** (now), **TCSETSW** (drain), **TCSETSF**, (flush), **TCSBRK**, **TCDRAIN**, **TCFLSH**, and **TCFLOW**. You should only use this knowledge when trying to understand the tty driver code, of course.

BUGS

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ttyname − file descriptor to terminal device name

SYNOPSIS

```
#define _POSIX_SOURCE 1
#include <unistd.h>

char *ttyname(int fd)
```

DESCRIPTION

Ttyname searches through the **/dev** directory for the terminal device file that is associated with file descriptor *fd*. It returns the full path name of the terminal file if found, NULL is returned otherwise.

SEE ALSO

ttyslot(3).

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ttyslot, fttyslot − utmp slot number

SYNOPSIS

```
#define _MINIX_SOURCE 1
#include <unistd.h>

int ttyslot(void)
int fttyslot(int fd)
```

DESCRIPTION

Ttyslot() returns the index of the login terminal in the **utmp** file. It tries **fttyslot()** on file descriptors **0, 1,** and **2** to find the index.

Fttyslot() returns the utmp index of the terminal associated with file descriptor *fd*. First it tries to map *fd* to a terminal name with **ttyname**(3), then it searches the **ttytab**(5) database with the **getttyent**(3) function for this terminal. This means that the utmp slot number is the same as the ttytab entry number counting from 1. The value 0 is returned if no slot number can be found for a file descriptor.

SEE ALSO

ttyname(3), **getttyent**(3), **utmp**(5), **ttytab**(5), **init**(8).

NOTES

Since 0 is used as an error return this means that the first entry in the utmp file is not used.

Ttyslot() is often found in a UNIX implementation, **fttyslot()** is Minix specific.

AUTHOR

Kees J. Bot (kjb@cs.vu.nl)

NAME

ungetc − push character back into input stream

SYNOPSIS

#include <stdio.h>

int ungetc(int *c***, FILE *****stream***)**

DESCRIPTION

Ungetc pushes the character *c* back on an input stream. That character will be returned by the next **getc** call on that stream. **Ungetc** returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

Fseek(3) erases all memory of pushed back characters.

SEE ALSO

getc(3), **setbuf**(3), **fseek**(3).

DIAGNOSTICS

Ungetc returns **EOF** if it can't push a character back.