NAME

fflush – flush a stream

SYNOPSIS

#include <stdio.h>

int fflush(FILE *_stream_);

DESCRIPTION

For output streams, **fflush**() forces a write of all user-space buffered data for the given output or update *stream* via the stream's underlying write function.

For input streams associated with seekable files (e.g., disk files, but not pipes or terminals), **fflush**() discards any buffered data that has been fetched from the underlying file, but has not been consumed by the application.

The open status of the stream is unaffected.

If the *stream* argument is NULL, **fflush**() flushes *all* open output streams.

For a nonlocking counterpart, see **unlocked_stdio**(3).

RETURN VALUE

Upon successful completion 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EBADF
stream is not an open stream, or is not open for writing.

The function **fflush**() may also fail and set *errno* for any of the errors specified for **write**(2).

SEE ALSO

fsync(2), **sync**(2), **write**(2), **fclose**(3), **fileno**(3), **fopen**(3), **setbuf**(3), **unlocked_stdio**(3)

NAME

fnmatch – match filename or pathname

SYNOPSIS

#include <fnmatch.h>

int fnmatch(const char *_pattern_, const char *_string_, int _flags_);

DESCRIPTION

The **fnmatch**() function checks whether the *string* argument matches the *pattern* argument, which is a shell wildcard pattern.

The *flags* argument modifies the behavior; it is the bitwise OR of zero or more of the following flags:

FNM_NOESCAPE
If this flag is set, treat backslash as an ordinary character, instead of an escape character.

FNM_PATHNAME
If this flag is set, match a slash in *string* only with a slash in *pattern* and not by an asterisk (*) or a question mark (?) metacharacter, nor by a bracket expression ([]) containing a slash.

FNM_PERIOD
If this flag is set, a leading period in *string* has to be matched exactly by a period in *pattern*. A period is considered to be leading if it is the first character in *string*, or if both **FNM_PATHNAME** is set and the period immediately follows a slash.

FNM_FILE_NAME
This is a GNU synonym for **FNM_PATHNAME**.

FNM_LEADING_DIR
If this flag (a GNU extension) is set, the pattern is considered to be matched if it matches an initial segment of *string* which is followed by a slash. This flag is mainly for the internal use of glibc and is only implemented in certain cases.

FNM_CASEFOLD
If this flag (a GNU extension) is set, the pattern is matched case-insensitively.

RETURN VALUE

Zero if *string* matches *pattern*, **FNM_NOMATCH** if there is no match or another nonzero value if there is an error.

CONFORMING TO

POSIX.2. The **FNM_FILE_NAME**, **FNM_LEADING_DIR**, and **FNM_CASEFOLD** flags are GNU extensions.

NAME

fopen, fdopen, fileno – stream open functions

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
int fileno(FILE *stream);
int fclose(FILE *stream);
```

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** to a shared memory object is undefined.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** is returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclose**, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL
The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

EBADF
The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for the routine **malloc**(3).

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl**(2).

NAME

fgetc, fgets, getc, getchar, fputc, fputs, putc, putchar – input and output of characters and strings

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

DESCRIPTION

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getc() is equivalent to **fgetc**() except that it may be implemented as a macro which evaluates *stream* more than once.

getchar() is equivalent to **getc**(*stdin*).

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A **'\0'** is stored after the last character in the buffer.

fputc() writes the character *c*, cast to an *unsigned char*, to *stream*.

fputs() writes the string *s* to *stream*, without its terminating null byte ('\0').

putc() is equivalent to **fputc**() except that it may be implemented as a macro which evaluates *stream* more than once.

putchar(*c*); is equivalent to **putc**(*c*, *stdout*).

Calls to the functions described here can be mixed with each other and with calls to other output functions from the *stdio* library for the same output stream.

RETURN VALUE

fgetc(), **getc**() and **getchar**() return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read. **fputc**(), **putc**() and **putchar**() return the character written as an *unsigned char* cast to an *int* or **EOF** on error.

fputs() returns a nonnegative number on success, or **EOF** on error.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **getwchar**(3), **scanf**(3), **ungetwc**(3), **write**(2), **ferror**(3), **fopen**(3), **fputwc**(3), **fputws**(3), **fseek**(3), **fwrite**(3), **gets**(3), **putwchar**(3), **scanf**(3), **unlocked_stdio**(3)

isalnum/isalpha/isascii/isblank/iscntrl/isdigit/isgraph/islower/isprint/ispunct/isspace/isupper/isxdigit

NAME

isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, – character classification functions

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);

int isascii(int c);
int isblank(int c);
```

DESCRIPTION

These functions check whether *c*, which must have the value of an *unsigned char* or **EOF**, falls into a certain character class according to the specified locale. The functions without the "_l" suffix perform the check based on the current locale.

isalnum()
: checks for an alphanumeric character; it is equivalent to (**isalpha**(*c*) **||** **isdigit**(*c*)).

isalpha()
: checks for an alphabetic character; in the standard **"C"** locale, it is equivalent to (**isupper**(*c*) **||** **islower**(*c*)). In some locales, there may be additional characters for which **isalpha**() is true—letters which are neither uppercase nor lowercase.

isascii()
: checks whether *c* is a 7-bit *unsigned char* value that fits into the ASCII character set.

isblank()
: checks for a blank character; that is, a space or a tab.

iscntrl()
: checks for a control character.

isdigit()
: checks for a digit (0 through 9).

isgraph()
: checks for any printable character except space.

islower()
: checks for a lowercase character.

isprint()
: checks for any printable character including space.

ispunct()
: checks for any printable character which is not a space or an alphanumeric character.

isspace()
: checks for white-space characters. In the **"C"** and **"POSIX"** locales, these are: space, form-feed (**'\f'**), newline (**'\n'**), carriage return (**'\r'**), horizontal tab (**'\t'**), and vertical tab (**'\v'**).

isupper()
: checks for an uppercase letter.

isxdigit()
: checks for hexadecimal digits, that is, one of
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

RETURN VALUE

The values returned are nonzero if the character *c* falls into the tested class, and zero if not.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
isalnum(), **isalpha**(), **isascii**(), **isblank**(), **iscntrl**(), **isdigit**(), **isgraph**(), **islower**(), **isprint**(), **ispunct**(), **isspace**(), **isupper**(), **isxdigit**()	Thread safety	MT-Safe

NOTES

The standards require that the argument *c* for these functions is either **EOF** or a value that is representable in the type *unsigned char*. If the argument *c* is of type *char*, it must be cast to *unsigned char*, as in the following example:

```
char c; ... res = toupper((unsigned char) c);
```

This is necessary because *char* may be the equivalent of *signed char*, in which case a byte where the top bit is set would be sign extended when converting to *int*, yielding a value that is outside the range of *unsigned char*.

The details of what characters belong to which class depend on the locale. For example, **isupper**() will not recognize an A-umlaut (Ä) as an uppercase letter in the default **C** locale.

NAME

calloc, malloc, free, realloc – Allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *calloc(size_t *nmemb*, **size_t** *size*);
void *malloc(size_t *size*);
void free(void **ptr*);
void *realloc(void **ptr*, **size_t** *size*);

DESCRIPTION

calloc() allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**() or **realloc**(). Otherwise, or if **free**(*ptr*) has already been called before, undefined behaviour occurs. If *ptr* is **NULL**, no operation is performed.

realloc() changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free**(*ptr*). Unless *ptr* is **NULL**, it must have been returned by an earlier call to **malloc**(), **calloc**() or **realloc**().

RETURN VALUE

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or **NULL** if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to *free*() is returned. If **realloc()** fails the original block is left untouched - it is not freed or moved.

CONFORMING TO

ANSI-C

SEE ALSO

brk(2), **posix_memalign**(3)

NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

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

DIR *opendir(const char ****name*);
int closedir(DIR ****dirp*);
struct dirent *readdir(DIR ****dir*);

DESCRIPTION opendir

The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

DESCRIPTION closedir

The **closedir()** function closes the directory stream associated with *dirp*. A successful call to **closedir()** also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirpP is not available after this call.*

RETURN VALUE

The **closedir()** function returns 0 on success. On error, −1 is returned, and *errno* is set appropriately.

DESCRIPTION readdir

The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred. It is safe to use **readdir()** inside threads if the pointers passed as *dir* are created by distinct calls to **opendir()**.

The data returned by **readdir()** is overwritten by subsequent calls to **readdir()** for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long        d_ino;      /* inode number */
    char        d_name[256]; /* filename */
};
```

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free**(3) it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately. To distinguish end of stream and from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EACCES
Permission denied.

ENOENT
Directory does not exist, or *name* is an empty string.

ENOTDIR
name is not a directory.

NAME

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

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
 ...
```

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The function **printf**() writes output to *stdout*, the standard output stream; **fprintf**() writes output to the given output *stream*; **sprintf**() and **snprintf**(), write to the character string *str*.

The function **snprintf**() writes at most *size* bytes (including the trailing null byte ('\0')) to *str*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg**(3)) are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

The functions **snprintf**() and **vsnprintf**() do not write more than *size* bytes (including the trailing '\0'). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. Thus, a return value of *size* or more means that the output was truncated.

If an output error is encountered, a negative value is returned.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The conversion specifier

A character that specifies the type of conversion to be applied. An example for a conversion specifier is:

o, **u**, **x**, **X**
: The *unsigned int* argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation.

s
: The *const char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

SEE ALSO

printf(1), **asprintf**(3), **dprintf**(3), **scanf**(3), **setlocale**(3), **wcrtomb**(3), **wprintf**(3), **locale**(5)

NAME

pthread_create – create a new thread / pthread_exit – terminate the calling thread

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void *
arg);

void pthread_exit(void *retval);
```

DESCRIPTION

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function *start_routine* passing it *arg* as first argument. The new thread terminates either explicitly, by calling **pthread_exit**(3), or implicitly, by returning from the *start_routine* function. The latter case is equivalent to calling **pthread_exit**(3) with the result returned by *start_routine* as exit code.

The *attr* argument specifies thread attributes to be applied to the new thread. See **pthread_attr_init**(3) for a complete list of thread attributes. The *attr* argument can also be **NULL**, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

pthread_exit terminates the execution of the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push**(3) are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non- **NULL** values associated with them in the calling thread (see **pthread_key_create**(3)). Finally, execution of the calling thread is stopped.

The *retval* argument is the return value of the thread. It can be consulted from another thread using **pthread_join**(3).

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.

The **pthread_exit** function never returns.

ERRORS

EAGAIN
: not enough system resources to create a process for the new thread.

EAGAIN
: more than **PTHREAD_THREADS_MAX** threads are already active.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_join(3), **pthread_detach**(3), **pthread_attr_init**(3).

NAME

pthread_detach – put a running thread in the detached state

SYNOPSIS

#include <pthread.h>

int pthread_detach(pthread_t th);

DESCRIPTION

pthread_detach put the thread *th* in the detached state. This guarantees that the memory resources consumed by *th* will be freed immediately when *th* terminates. However, this prevents other threads from synchronizing on the termination of *th* using **pthread_join**.

A thread can be created initially in the detached state, using the **detachstate** attribute to **pthread_create**(3). In contrast, **pthread_detach** applies to threads created in the joinable state, and which need to be put in the detached state later.

After **pthread_detach** completes, subsequent attempts to perform **pthread_join** on *th* will fail. If another thread is already joining the thread *th* at the time **pthread_detach** is called, **pthread_detach** does nothing and leaves *th* in the joinable state.

RETURN VALUE

On success, 0 is returned. On error, a non-zero error code is returned.

ERRORS

ESRCH
No thread could be found corresponding to that specified by *th*

EINVAL
the thread *th* is already in the detached state

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_create(3), **pthread_join**(3), **pthread_attr_setdetachstate**(3).

NAME

pthread_self – obtain ID of the calling thread

SYNOPSIS

#include <pthread.h>

pthread_t pthread_self(void);

Compile and link with *–pthread.*

DESCRIPTION

The **pthread_self**() function returns the ID of the calling thread. This is the same value that is returned in **thread* in the **pthread_create**(3) call that created this thread.

RETURN VALUE

This function always succeeds, returning the calling thread's ID.

ERRORS

This function always succeeds.

NOTES

POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID; for example, representation using either an arithmetic type or a structure is permitted. Therefore, variables of type *pthread_t* can't portably be compared using the C equality operator (**==**); use **pthread_equal**(3) instead.

Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.

Thread IDs are guaranteed to be unique only within a process. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

The thread ID returned by **pthread_self**() is not the same thing as the kernel thread ID returned by a call to **gettid**(2).

SEE ALSO

pthread_create(3), **pthread_equal**(3), **pthreads**(7)

NAME

qsort – sorts an array

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
      int(*compar)(const void *, const void *));
```

DESCRIPTION

The **qsort**() function sorts an array with *nmemb* elements of size *size*. The *base* argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE

The **qsort**() function returns no value.

SEE ALSO

sort(1), **alphasort**(3), **strcmp**(3), **versionsort**(3)

ATTRIBUTES

Multithreading (see pthreads(7))

The **qsort**() function is thread-safe if the comparison function *compar* does not access any global variables.

NAME

stat, fstat, lstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

lstat(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

These functions return information about a file. No permissions are required on the file itself, but — in the case of **stat**() and **lstat**() — execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number                 */
    mode_t    st_mode;    /* protection                   */
    nlink_t   st_nlink;   /* number of hard links         */
    uid_t     st_uid;     /* user ID of owner             */
    gid_t     st_gid;     /* group ID of owner            */
    dev_t     st_rdev;    /* device ID (if special file)  */
    off_t     st_size;    /* total size, in bytes         */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated   */
    time_t    st_atime;   /* time of last access          */
    time_t    st_mtime;   /* time of last modification    */
    time_t    st_ctime;   /* time of last status change   */
};
```

The *st_dev* field describes the device on which this file resides.

The *st_rdev* field describes the device that this file (inode) represents.

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

The *st_blksize* field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See "noatime" in **mount**(8).)

The field *st_atime* is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following POSIX macros are defined to check the file type using the *st_mode* field:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EACCES
Search permission is denied for one of the directories in the path prefix of *path*. (See also **path_resolution**(7).)

EBADF
fd is bad.

EFAULT
Bad address.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
File name too long.

ENOENT
A component of the path *path* does not exist, or the path is an empty string.

ENOMEM
Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path is not a directory.

SEE ALSO

access(2), **chmod**(2), **chown**(2), **fstatat**(2), **readlink**(2), **utime**(2), **capabilities**(7), **symlink**(7)

NAME

strcat, strchr, strcmp, strcpy, strdup, strlen, strncat, strncmp, strncpy, strstr, strtok − string operations

SYNOPSIS

#include <string.h>

char *strcat(char ****dest*, const char ****src***);**
Append the string *src* to the string *dest*, returning a pointer *dest*.

char *strchr(const char ****s*, int** *c***);**
Return a pointer to the first occurrence of the character *c* in the string *s*.

int strcmp(const char ****s1*, const char ****s2***);**
Compare the strings *s1* with *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strcpy(char ****dest*, const char ****src***);**
Copy the string *src* to *dest*, returning a pointer to the start of *dest*.

char *strdup(const char ****s*);**
Return a duplicate of the string *s* in memory allocated using **malloc**(3).

size_t strlen(const char ****s*);**
Return the length of the string *s*.

char *strncat(char ****dest*, const char ****src***, size_t** *n***);**
Append at most *n* characters from the string *src* to the string *dest*, returning a pointer to *dest*.

int strncmp(const char ****s1*, const char ****s2***, size_t** *n***);**
Compare at most *n* bytes of the strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strncpy(char ****dest*, const char ****src***, size_t** *n***);**
Copy at most *n* bytes from string *src* to *dest*, returning a pointer to the start of *dest*.

char *strstr(const char ****haystack*, const char ****needle***);**
Find the first occurrence of the substring *needle* in the string *haystack*, returning a pointer to the found substring.

char *strtok(char ****s*, const char ****delim***);**
Extract tokens from the string *s* that are delimited by one of the bytes in *delim*.

DESCRIPTION

The string functions perform operations on null-terminated strings.