

accept(2)

accept(2)

**NAME**  
accept – accept a connection on a socket

**SYNOPSIS**  
`#include <sys/types.h>`  
`#include <sys/socket.h>`

`int accept(int s, struct sockaddr *addr, int *addrlen);`

#### DESCRIPTION

The argument *s* is a socket that has been created with `socket(3N)` and bound to an address with `bind(3N)`, and that is listening for connections after a call to `listen(3N)`. The `accept()` function extracts the first connection on the queue of pending connections, creates a new socket with the properties of *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, `accept()` blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, `accept()` returns an error as described below. The `accept()` function uses the `netconfig(4)` file to determine the STREAMS device file name associated with *s*. This is the device on which the connect indication will be accepted. The accepted socket, *ns*, is used to read and write data to and from the socket that connected to *s*; it is not used to accept more connections. The original socket (*s*) remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity as it is known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication occurs.

The argument *addrlen* is a value-result parameter. Initially, it contains the amount of space pointed to by *addr*; on return it contains the length in bytes of the address returned.

The `accept()` function is used with connection-based socket types, currently with `SOCK_STREAM`.

It is possible to `select(3C)` or `poll(2)` a socket for the purpose of an `accept()` by selecting or polling it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call `accept()`.

#### RETURN VALUES

The `accept()` function returns `-1` on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

#### ERRORS

`accept()` will fail if:

**EBADF** The descriptor is invalid.

**EINTR** The accept attempt was interrupted by the delivery of a signal.

**EMFILE** The per-process descriptor table is full.

**ENODEV** The protocol family and type corresponding to *s* could not be found in the `netconfig` file file.

**ENOMEM** There was insufficient user memory available to complete the operation.

**EPROTO** A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized or the connection has already been released.

**EWOLDBLOCK** The socket is marked as non-blocking and no connections are present to be accepted.

#### SEE ALSO

`poll(2)`, `bind(3N)`, `connect(3N)`, `listen(3N)`, `select(3C)`, `socket(3N)`, `netconfig(4)`, `attributes(5)`, `socket(5)`

bind(2)

bind(2)

**NAME**  
bind – bind a name to a socket

**SYNOPSIS**  
`#include <sys/types.h>`  
`#include <sys/socket.h>`

`int bind(int s, const struct sockaddr *name, int namelen);`

#### DESCRIPTION

`bind()` assigns a name to an unnamed socket. When a socket is created with `socket(3N)`, it exists in a name space (address family) but has no name assigned. `bind()` requests that the name pointed to by *name* be assigned to the socket.

#### RETURN VALUES

If the bind is successful, `0` is returned. A return value of `-1` indicates an error, which is further specified in the global `errno`.

#### ERRORS

The `bind()` call will fail if:

**EACCES** The requested address is protected and the current user has inadequate permission to access it.

**EADDRINUSE** The specified address is already in use.

**EADDRNOTAVAIL** The specified address is not available on the local machine.

**EBADF** *s* is not a valid descriptor.

**EINVAL** *namelen* is not the size of a valid address for the specified address family.

**EINVAL** The socket is already bound to an address.

**ENOSR** There were insufficient STREAMS resources for the operation to complete.

**ENOTSOCK** *s* is a descriptor for a file, not a socket.

The following errors are specific to binding names in the UNIX domain:

**EACCES** Search permission is denied for a component of the path prefix of the pathname in *name*.

**EIO** An I/O error occurred while making the directory entry or allocating the inode.

**EISDIR** A null pathname was specified.

**ELOOP** Too many symbolic links were encountered in translating the pathname in *name*.

**ENOENT** A component of the path prefix of the pathname in *name* does not exist.

**ENOTDIR** A component of the path prefix of the pathname in *name* is not a directory.

**EROFS** The inode would reside on a read-only file system.

#### SEE ALSO

`unlink(2)`, `socket(3N)`, `attributes(5)`, `socket(5)`

#### NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using `unlink(2)`).

The rules used in name binding vary between communication domains.

chdir(2)

chdir(2)

**NAME**

chdir, fchdir – change working directory

**SYNOPSIS**

```
#include <unistd.h>
```

```
int chdir(const char *path);
```

```
int fchdir(int fd);
```

**DESCRIPTION**

chdir() changes the current working directory of the calling process to the directory specified in *path*.

fchdir() is identical to chdir(); the only difference is that the directory is given as an open file descriptor.

**RETURN VALUE**

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

**ERRORS**

Depending on the file system, other errors can be returned. The more general errors for chdir() are listed below:

**EACCES**

Search permission is denied for one of the components of *path*. (See also [path\\_resolution\(7\)](#))

**EFAULT**

*path* points outside your accessible address space.

**EIO**

An I/O error occurred.

**EINVAL**

Too many symbolic links were encountered in resolving *path*.

**ENAMETOOLONG**

*path* is too long.

**ENOENT**

The file does not exist.

**ENOMEM**

Insufficient kernel memory was available.

**ENOTDIR**

A component of *path* is not a directory.

The general errors for fchdir() are listed below:

**EACCES**

Search permission was denied on the directory open on *fd*.

**EBADF**

*fd* is not a valid file descriptor.

**SEE ALSO**

[chroot\(2\)](#), [getcwd\(3\)](#), [path\\_resolution\(7\)](#)

opendir/readdir(3)

opendir/readdir(3)

**NAME**

opendir – open a directory / readdir – read a directory

**SYNOPSIS**

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

```
#include <dirent.h>
```

```
DIR *opendir(const char *name);
```

```
struct dirent *readdir(DIR *dir);
```

```
int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

**DESCRIPTION**

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 or NULL, if an error occurred.

**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.

**DESCRIPTION readdir\_r**

The readdir\_r() function initializes the structure referenced by *entry*, and stores a pointer to this structure in *result*. On successful return, the pointer returned at *result* will have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value NULL.

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 */
    off_t     d_off;         /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char  d_type;   /* type of file */
    char         d_name[256]; /* filename */
};
```

**RETURN VALUE**

The readdir() function returns a pointer to a dirent structure, or NULL, if an error occurs or end-of-file is reached.

readdir\_r() returns 0 if successful or an error number to indicate failure.

**ERRORS**

**EACCES** Permission denied.

**ENOENT**

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

**ENOTDIR**

*name* is not a directory.

dup(2)

dup(2)

**NAME**

dup, dup2 – duplicate a file descriptor

**SYNOPSIS**

#include <unistd.h>

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

**DESCRIPTION**

dup() and dup2() create a copy of the file descriptor *oldfd*.

dup() uses the lowest-numbered unused descriptor for the new descriptor.

dup2() makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary, but note the following:

- \* If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
- \* If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then dup2() does nothing, and returns *newfd*.

After a successful return from dup() or dup2(), the old and new file descriptors may be used interchangeably. They refer to the same open file description (see open(2)) and thus share file offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of the descriptors, the offset is also changed for the other.

The two descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (FD\_CLOEXEC; seefcntl(2)) for the duplicate descriptor is off.

**RETURN VALUE**

dup() and dup2() return the new descriptor, or -1 if an error occurred (in which case, *errno* is set appropriately).

**ERRORS**

**EBADF**

*oldfd* isn't an open file descriptor, or *newfd* is out of the allowed range for file descriptors.

**EBUSY**

(Linux only) This may be returned by dup2() during a race condition with open(2) and dup().

**EINTR**

The dup2() call was interrupted by a signal; see signal(7).

**EMFILE**

The process already has the maximum number of file descriptors open and tried to open a new one.

**NOTES**

The error returned by dup2() is different from that returned by fcntl(..., F\_DUPFD, ...) when *newfd* is out of range. On some systems dup2() also sometimes returns EINVAL like F\_DUPFD.

If *newfd* was open, any errors that would have been reported at close(2) time are lost. A careful programmer will not use dup2() without closing *newfd* first.

**SEE ALSO**

close(2), fcntl(2), open(2)

feof/ferro/fileno(3)

feof/ferro/fileno(3)

**NAME**

clearerr, feof, ferro, fileno – check and reset stream status

**SYNOPSIS**

#include <stdio.h>

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferro(FILE *stream);
int fileno(FILE *stream);
```

**DESCRIPTION**

The function clearerr() clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function feof() tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function clearerr().

The function ferro() tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the clearerr() function.

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

For non-locking counterparts, see unlocked\_stdio(3).

**ERRORS**

These functions should not fail and do not set the external variable *errno*. (However, in case fileno() detects that its argument is not a valid stream, it must return -1 and set *errno* to EBADF.)

**CONFORMING TO**

The functions clearerr(), feof(), and ferro() conform to C89 and C99.

**SEE ALSO**

open(2), filopen(3), stdio(3), unlocked\_stdio(3)

fopen/fdopen/filen0(3)

open/fdopen/filen0(3)

#### NAME

fopen, fdopen, fileno – stream open functions

#### SYNOPSIS

```
#include <stdio.h>
```

```
FILE *fopen(const char *path, const char *mode);  
FILE *fdopenint files, const char *mode);  
int fileno(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 **flopen** function associates a stream with the existing file descriptor, *files*. 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 *files*, 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 **flopen** is closed. The result of applying **flopen** to a shared memory object is undefined.

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

#### ERRORS

##### EINVAL

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

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)**.

#### SEE ALSO

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

getc/fgetc/pgetc/fputs(3)

getc/fgetc/pgetc/fputs(3)

#### NAME

fgetc, fgets, getc, getchar, fputc, fputs, pgetc, 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 pgetc(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').

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

**putchar(c)** is equivalent to **pgetc(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()**, **pgetc()** 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)**, **fgetc(3)**, **fgets(3)**, **fopen(3)**, **fread(3)**, **fseek(3)**, **getline(3)**, **getwchar(3)**, **scanf(3)**, **ungetc(3)**, **write(2)**, **ferror(3)**, **fopen(3)**, **fputc(3)**, **fputs(3)**, **fseek(3)**, **write(3)**, **gets(3)**, **putwchar(3)**, **scanf(3)**, **unlockd\_stdio(3)**

socket(2) / ipv6(7)

socket(2) / ipv6(7)

listen(2)

listen(2)

**NAME**

ipv6, PF\_INET6 – Linux IPv6 protocol implementation

**SYNOPSIS**

```
#include <sys/socket.h>
#include <netinet/in.h>
```

```
tcp6_socket = socket(PF_INET6, SOCK_STREAM, 0);
raw6_socket = socket(PF_INET6, SOCK_RAW, protocol);
udp6_socket = socket(PF_INET6, SOCK_DGRAM, protocol);
```

**DESCRIPTION**

Linux 2.2 optionally implements the Internet Protocol, version 6. This man page contains a description of the IPv6 basic API as implemented by the Linux kernel and glibc 2.1. The interface is based on the BSD sockets interface; see [socket\(7\)](#).

The IPv6 API aims to be mostly compatible with the [ip\(7\)](#) v4 API. Only differences are described in this man page.

To bind an **AF\_INET6** socket to any process the local address should be copied from the *inaddr\_any* variable which has *in6\_addr* type. In static initializations **IN6ADDR\_ANY\_INIT** may also be used, which expands to a constant expression. Both of them are in network order.

The IPv6 loopback address (:::1) is available in the global *inaddr\_loopback* variable. For initializations **IN6ADDR\_LOOPBACK\_INIT** should be used.

IPv4 connections can be handled with the v6 API by using the v4-mapped-on-v6 address type; thus a program only needs only to support this API type to support both protocols. This is handled transparently by the address handling functions in libc.

IPv4 and IPv6 share the local port space. When you get an IPv4 connection or packet to a IPv6 socket its source address will be mapped to v6 and it will be mapped to v6.

**Address Format**

```
struct sockadd_in6 {
    uint16_t    sin6_family; /* AF_INET6 */
    uint16_t    sin6_port; /* port number */
    uint32_t    sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t    sin6_scope_id; /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char s6_addr[16]; /* IPv6 address */
};
```

*sin6\_family* is always set to **AF\_INET6**; *sin6\_port* is the protocol port (see [sin\\_port](#) in [ip\(7\)](#)); *sin6\_flowinfo* is the IPv6 flow identifier; *sin6\_addr* is the 128-bit IPv6 address. *sin6\_scope\_id* is an ID of depending of on the scope of the address. It is new in Linux 2.4. Linux only supports it for link scope addresses, in that case *sin6\_scope\_id* contains the interface index (see [netdevice\(7\)](#))

**NOTES**

The *sockadd\_in6* structure is bigger than the generic *sockaddr*. Programs that assume that all address types can be stored safely in a *struct sockaddr* need to use *struct sockaddr\_storage* for that instead.

**SEE ALSO**

[cmsng\(3\)](#), [ip\(7\)](#)

**NAME**

listen – listen for connections on a socket

**SYNOPSIS**

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
```

```
int listen(int sockfd, int backlog);
```

**DESCRIPTION**

[listen\(\)](#) marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using [accept\(2\)](#).

The *sockfd* argument is a file descriptor that refers to a socket of type **SOCK\_STREAM** or **SOCK\_SEQPACKET**.

The *backlog* argument defines the maximum length to which the queue of pending connections for *sockfd* may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of **ECONNREFUSED** or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

**RETURN VALUE**

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

**ERRORS**

**EADDRINUSE**

Another socket is already listening on the same port.

**EBADF**

The argument *sockfd* is not a valid descriptor.

**ENOTSOCK**

The argument *sockfd* is not a socket.

**NOTES**

To accept connections, the following steps are performed:

1. A socket is created with [socket\(2\)](#).
2. The socket is bound to a local address using [bind\(2\)](#), so that other sockets may be [connect\(2\)](#)ed to it.
3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with [listen\(\)](#).
4. Connections are accepted with [accept\(2\)](#).

If the *backlog* argument is greater than the value in */proc/sys/net/core/somaxconn*, then it is silently truncated to that value; the default value in this file is 128.

**EXAMPLE**

See [bind\(2\)](#).

**SEE ALSO**

[accept\(2\)](#), [bind\(2\)](#), [connect\(2\)](#), [socket\(2\)](#), [socket\(7\)](#)

sigaction(2)

sigaction(2)

**NAME**

sigaction – POSIX signal handling functions.

**SYNOPSIS**

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

**DESCRIPTION**

The `sigaction` system call is used to change the action taken by a process on receipt of a specific signal.

*signum* specifies the signal and can be any valid signal except `SIGKILL` and `SIGSTOP`.

If *act* is non-null, the new action for signal *signum* is installed from *act*. If *oldact* is non-null, the previous action is saved in *oldact*.

The `sigaction` structure is defined as something like

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

On some architectures a union is involved - do not assign to both `sa_handler` and `sa_sigaction`.

The `sa_restorer` element is obsolete and should not be used. POSIX does not specify a `sa_restorer` element.

`sa_handler` specifies the action to be associated with *signum* and may be `SIG_DFL` for the default action, `SIG_IGN` to ignore this signal, or a pointer to a signal handling function.

`sa_mask` gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the `SA_NODEFER` or `SA_NOMASK` flags are used.

`sa_flags` specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

- SA\_NOCLDSTOP** If *signum* is `SIGCHLD`, do not receive notification when child processes stop (i.e., when child processes receive one of `SIGSTOP`, `SIGTSTP`, `SIGTIN` or `SIGTTOU`).
- SA\_RESTART** Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals.

**RETURN VALUES**

`sigaction` returns 0 on success and -1 on error.

**ERRORS**

**EINVAL**

An invalid signal was specified. This will also be generated if an attempt is made to change the action for `SIGKILL` or `SIGSTOP`, which cannot be caught.

**SEE ALSO**

`kill(1)`, `kill(2)`, `killpg(2)`, `pause(2)`, `sigsetops(3)`.

sigsuspend/sigprocmask(2)

sigsuspend/sigprocmask(2)

**NAME**

sigprocmask – change and/or examine caller’s signal mask

sigsuspend – install a signal mask and suspend caller until signal

**SYNOPSIS**

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

```
int sigsuspend(const sigset_t *set);
```

**DESCRIPTION**

`sigprocmask`

The `sigprocmask()` function is used to examine and/or change the caller’s signal mask. If the value is `SIG_BLOCK`, the set pointed to by the argument *set* is added to the current signal mask. If the value is `SIG_UNBLOCK`, the set pointed by the argument *set* is removed from the current signal mask. If the value is `SIG_SETMASK`, the current signal mask is replaced by the set pointed to by the argument *set*. If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the value *how* is not significant and the caller’s signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to `sigprocmask()`, at least one of those signals will be delivered before the call to `sigprocmask()` returns.

It is not possible to block those signals that cannot be ignored this restriction is silently imposed by the system. See `sigaction(2)`.

If `sigprocmask()` fails, the caller’s signal mask is not changed.

**RETURN VALUES**

On success, `sigprocmask()` returns 0. On failure, it returns -1 and sets `errno` to indicate the error.

**ERRORS**

`sigprocmask()` fails if any of the following is true:

**EFAULT** *set* or *oset* points to an illegal address.

**EINVAL** The value of the *how* argument is not equal to one of the defined values.

**DESCRIPTION**

`sigsuspend`

`sigsuspend()` replaces the caller’s signal mask with the set of signals pointed to by the argument *set* and then suspends the caller until delivery of a signal whose action is either to execute a signal catching function or to terminate the process.

If the action is to terminate the process, `sigsuspend()` does not return. If the action is to execute a signal catching function, `sigsuspend()` returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the call to `sigsuspend()`.

It is not possible to block those signals that cannot be ignored (see `signal(5)`); this restriction is silently imposed by the system.

**RETURN VALUES**

Since `sigsuspend()` suspends process execution indefinitely, there is no successful completion return value. On failure, it returns -1 and sets `errno` to indicate the error.

**ERRORS**

`sigsuspend()` fails if either of the following is true:

**EFAULT** *set* points to an illegal address.

**EINTR** A signal is caught by the calling process and control is returned from the signal catching function.

**SEE ALSO**

`sigaction(2)`, `sigsetops(3C)`.

sigsetops(3C)

sigsetops(3C)

printf(3)

printf(3)

**NAME**

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals

**SYNOPSIS**

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
```

**DESCRIPTION**

These functions manipulate *sigset\_t* data types, representing the set of signals supported by the implementation.

**sigemptyset()** initializes the set pointed to by *set* to exclude all signals defined by the system.

**sigfillset()** initializes the set pointed to by *set* to include all signals defined by the system.

**sigaddset()** adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

**sigdelset()** deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

**sigismember()** checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Any object of type *sigset\_t* must be initialized by applying either **sigemptyset()** or **sigfillset()** before applying any other operation.

**RETURN VALUES**

Upon successful completion, the **sigismember()** function returns a value of one if the specified signal is a member of the specified set, or a value of 0 if it is not. Upon successful completion, the other functions return a value of 0. Otherwise a value of -1 is returned and **errno** is set to indicate the error.

**ERRORS**

**sigaddset()**, **sigdelset()**, and **sigismember()** will fail if the following is true:

**EINVAL** The value of the *signo* argument is not a valid signal number.

**sigfillset()** will fail if the following is true:

**EFAULT** The *set* argument specifies an invalid address.

**SEE ALSO**

**sigaction(2)**, **sigpending(2)**, **sigprocmask(2)**, **sigsuspend(2)**, **attributes(5)**, **signal(5)**

**NAME**

printf, fprintf, sprintf, vsprintf, 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 vsprintf(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 functions **printf()** and **vprintf()** write output to *stdout*, the standard output stream; **fprintf()** and **vfprintf()** write output to the given output stream; **sprintf()**, **vsprintf()**, and **vsnprintf()** write to the character string *str*.

The functions **sprintf()** and **vsprintf()** write at most *size* bytes (including the trailing null byte '\0') to *str*.

These eight 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 **sprintf()** and **vsprintf()** 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. (See also below under NOTES.)

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:

**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)**, **wctomb(3)**, **vprintf(3)**, **locale(5)**

stat(2)

stat(2)

stat(2)

stat(2)

**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 */
    blkcnt_t st_blksize; /* blocksize for the system I/O */
    btime_t st_btime; /* 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\(7\)](#)



strchr(3)

strchr(3)

strtok(3)

strtok(3)

**NAME**

strchr, strchr – locate character in string

**SYNOPSIS**

#include <string.h>

char \*strchr(const char \*s, int c);

char \*strchr(const char \*s, int c);

**DESCRIPTION**

The **strchr()** function returns a pointer to the first occurrence of the character *c* in the string *s*.

The **strchr()** function returns a pointer to the last occurrence of the character *c* in the string *s*.

Here "character" means "byte"; these functions do not work with wide or multibyte characters.

**RETURN VALUE**

The **strchr()** and **strchr()** functions return a pointer to the matched character or **NULL** if the character is not found. The terminating null byte is considered part of the string, so that if *c* is specified as '\0', these functions return a pointer to the terminator.

**SEE ALSO**

**index(3), memchr(3), rindex(3), string(3), strlen(3), strpbrk(3), strsep(3), strspn(3), strstr(3), strtok(3), wcschr(3), wstrchr(3)**

**NAME**

strtok, strtok\_r – extract tokens from strings

**SYNOPSIS**

#include <string.h>

char \*strtok(char \*str, const char \*delim);

**DESCRIPTION**

The **strtok()** function parses a string into a sequence of tokens. On the first call to **strtok()** the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* should be **NULL**.

The *delim* argument specifies a set of characters that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

Each call to **strtok()** returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting character. If no more tokens are found, **strtok()** returns **NULL**.

A sequence of two or more contiguous delimiter characters in the parsed string is considered to be a single delimiter. Delimiter characters at the start or end of the string are ignored. Put another way: the tokens returned by **strtok()** are always non-empty strings.

**RETURN VALUE**

The **strtok()** function returns a pointer to the next token, or **NULL** if there are no more tokens.

**BUGS**

Avoid using these functions. If you do use them, note that:

These functions modify their first argument.

These functions cannot be used on constant strings.

The identity of the delimiting character is lost.

The **strtok()** function uses a static buffer while parsing, so it's not thread safe. Use **strtok\_r()** if this matters to you.

**SEE ALSO**

**index(3), memchr(3), rindex(3), strchr(3), strpbrk(3), strsep(3), strspn(3), strstr(3), wstrtok(3)**

waitpid(2)

waitpid(2)

**NAME**  
waitpid – wait for child process to change state

**SYNOPSIS**  
`#include <sys/types.h>`  
`#include <sys/wait.h>`

`pid_t waitpid(pid_t pid, int *stat_loc, int options);`

**DESCRIPTION**

`waitpid()` suspends the calling process until one of its children changes state; if a child process changed state prior to the call to `waitpid()`, return is immediate. `pid` specifies a set of child processes for which status is requested.

If `pid` is equal to `(pid_0-1)`, status is requested for any child process.

If `pid` is greater than `(pid_0)`, it specifies the process ID of the child process for which status is requested.

If `pid` is equal to `(pid_0)` status is requested for any child process whose process group ID is equal to that of the calling process.

If `pid` is less than `(pid_0-1)`, status is requested for any child process whose process group ID is equal to the absolute value of `pid`.

If `waitpid()` returns because the status of a child process is available, then that status may be evaluated with the macros defined by `wstat(5)`. If the calling process had specified a non-zero value of `stat_loc`, the status of the child process will be stored in the location pointed to by `stat_loc`.

The `options` argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header `<sys/wait.h>`:

**WCONTINUED**

The status of any continued child process specified by `pid`, whose status has not been reported since it continued, is also reported to the calling process.

**WNOHANG**

`waitpid()` will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by `pid`.

**WNOWAIT**

Keep the process whose status is returned in `stat_loc` in a waitable state. The process may be waited for again with identical results.

**RETURN VALUES**

If `waitpid()` returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If `waitpid()` returns due to the delivery of a signal to the calling process, `-1` is returned and `errno` is set to `EINTR`. If this function was invoked with `WNOHANG` set in `options`, it has at least one child process specified by `pid` for which status is not available, and status is not available for any process specified by `pid`, `0` is returned. Otherwise, `-1` is returned, and `errno` is set to indicate the error.

**ERRORS**

`waitpid()` will fail if one or more of the following is true:

**ECHILD**

The process or process group specified by `pid` does not exist or is not a child of the calling process or can never be in the states specified by `options`.

**EINTR**

`waitpid()` was interrupted due to the receipt of a signal sent by the calling process.

**EINVAL**

An invalid value was specified for `options`.

**SEE ALSO**

`exec(2)`, `exit(2)`, `fork(2)`, `sigaction(2)`, `wstat(5)`