NAME

connect − initiate a connection on a socket

SYNOPSIS

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

int connect(int *sockfd***, const struct sockaddr ****serv_addr***, socklen_t** *addrlen***);**

DESCRIPTION

The file descriptor *sockfd* must refer to a socket. If the socket is of type **SOCK_DGRAM** then the *serv_addr* address is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type **SOCK_STREAM** or **SOCK_SEQPACKET**, this call attempts to make a connection to another socket. The other socket is specified by *serv_addr*, which is an address (of length *addrlen*) in the communications space of the socket. Each communications space interprets the *serv_addr* parameter in its own way.

Generally, connection-based protocol sockets may successfully **connect** only once; connectionless protocol sockets may use **connect** multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the *sa_family* member of **sockaddr** set to **AF_UNSPEC**.

RETURN VALUE

If the connection or binding succeeds, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

The following are general socket errors only. There may be other domain-specific error codes.

EBADF
The file descriptor is not a valid index in the descriptor table.

EFAULT
The socket structure address is outside the user's address space.

ENOTSOCK
The file descriptor is not associated with a socket.

EISCONN
The socket is already connected.

ECONNREFUSED
No one listening on the remote address.

ENETUNREACH
Network is unreachable.

EADDRINUSE
Local address is already in use.

EAFNOSUPPORT
The passed address didn't have the correct address family in its *sa_family* field.

EACCES, EPERM
The user tried to connect to a broadcast address without having the socket broadcast flag enabled or the connection request failed because of a local firewall rule.

SEE ALSO

accept(2), **bind**(2), **listen**(2), **socket**(2), **getsockname**(2)

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 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 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.

NAME

clearerr, feof, ferror, fileno − check and reset stream status

SYNOPSIS

#include <stdio.h>

void clearerr(FILE ***stream*);**
int feof(FILE ***stream*);**
int ferror(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 **ferror**() 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 **ferror**() conform to C89 and C99.

SEE ALSO

open(2), **fdopen**(3), **stdio**(3), **unlocked_stdio**(3)

NAME

fopen, fdopen − stream open functions

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char ***path*, const char ***mode***);**
FILE *fdopen(int *fildes***, const char ***mode***);**

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.

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)

NAME

fgetc, fgets, getc, getchar, gets, ungetc − input 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);
char *gets(char ****s*);**
int ungetc(int *c***, FILE ****stream***);**

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***)**.

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with **'\0'**. No check for buffer overrun is performed (see **BUGS** below).

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.

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

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

For non-locking counterparts, see **unlocked_stdio**(3).

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.

gets() and **fgets**() return *s* on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns *c* on success, or **EOF** on error.

CONFORMING TO

C89, C99. LSB deprecates **gets**().

BUGS

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

It is not advisable to mix calls to input functions from the *stdio* library with low-level calls to **read**(2) for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **getwchar**(3), **puts**(3), **scanf**(3), **ungetwc**(3), **unlocked_stdio**(3)

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

rename − change the name or location of a file

unlink − remove directory entry

SYNOPSIS

#include <stdio.h>

int rename(const char ***oldpath*, const char ***newpath***);**

int unlink(const char ***path*);**

DESCRIPTION rename

rename() renames a file, moving it between directories if required. Any other hard links to the file (as created using **link**(2)) are unaffected. Open file descriptors for *oldpath* are also unaffected.

RETURN VALUE

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

DESCRIPTION unlink

The **unlink( )** function removes a link to a file. It removes the link named by the pathname pointed to by *path* and decrements the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before **unlink( )** returns, but the removal of the file contents will be postponed until all references to the file are closed.

RETURN VALUES

Upon successful completion, **0** is returned. Otherwise, **−1** is returned and **errno** is set to indicate the error.

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**, **SIGTTIN** 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),

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),

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

socket − create an endpoint for communication

SYNOPSIS

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

int socket(int *domain***, int** *type***, int** *protocol***);**

DESCRIPTION

socket( ) creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. The currently understood formats are:

PF_INET	ARPA Internet protocols

The socket has the indicated *type*, which specifies the communication semantics. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM

A **SOCK_STREAM** type provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism may be supported. A **SOCK_DGRAM** socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).

protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the “communication domain” in which communication is to take place. If a protocol is specified by the caller, then it will be packaged into a socket level option request and sent to the underlying protocol layers.

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a **connect**(3N) call. Once connected, data may be transferred using **read**(2) and **write**(2) calls or some variant of the **send**(3N) and **recv**(3N) calls. When a session has been completed, a **close**(2) may be performed. Out-of-band data may also be transmitted as described on the **send**(3N) manual page and received as described on the **recv**(3N) manual page.

The communications protocols used to implement a **SOCK_STREAM** insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with **ETIMEDOUT** as the specific code in the global variable **errno**. A **SIGPIPE** signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

RETURN VALUES

A **−1** is returned if an error occurs. Otherwise the return value is a descriptor referencing the socket.

ERRORS

The **socket( )** call fails if:

EACCES	Permission to create a socket of the specified type and/or protocol is denied.
EMFILE	The per-process descriptor table is full.
ENOMEM	Insufficient user memory is available.

SEE ALSO

close(2), **read**(2), **write**(2), **accept**(3N), **bind**(3N), **connect**(3N), **listen**(3N),

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_t)−1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)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_t)−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)