Index

getprotobyname

    struct protoent *getprotobyname(const char *name);

getprotobyname_r

    int getprotobyname_r(const char *name,   struct protoent *result,   struct protoent_data *buffer);

setprotoent

    int setprotoent(int stayopen);

setprotoent_r

    int setprotoent_r(int stayopen, struct protoent_data *buffer);

endprotoent

    int endprotoent(void);

endprotoent_r

    int endprotoent_r(struct protoent_data *buffer);

Getpeername   get address of connected peer

    int getpeername(int s, void *addr, int *addrlen);

Perror   system error messages

    char *strerror(int errnum);

Select   synchronous I/O multiplexing

    int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set   *errorfds, struct timeval *timeout);

NAME

socket() - create an endpoint for communication

SYNOPSIS

#include <sys/socket.h>

AF_CCITT Only

#include <x25/x25ccittproto.h>

int socket(int af, int type, int protocol);

DESCRIPTION

The socket() system call creates an endpoint for communication and returns a descriptor. The socket descriptor returned is used in all subsequent socket-related system calls.

The *af* parameter specifies an address family to be used to interpret addresses in later operations that specify the socket. These address families are defined in the include files <sys/socket.h> and <x25/ccittproto.h>. The only currently supported address families are:

AF_INET	(DARPA Internet addresses)
AF_UNIX	(path names on a local node)
AF_CCITT	(CCITT X.25 addresses)

The *type* specifies the semantics of communication for the socket.Currently defined types are:

SOCK_STREAM	Sequenced, reliable, two-way-connection-based byte streams.
SOCK_DGRAM	Datagrams (connectionless, unreliable messages of a fixed, typically small, maximum length; for AF_INET only).

protocol specifies a particular protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address family. However, many protocols may exist, in which case a particular protocol must be specified. The protocol number to use depends on the communication domain in which communication is to take place (see *services*(4) and *protocols*(4)).

protocol can be specified as zero, which causes the system to choose a protocol type to use.

Sockets of type SOCK_STREAM are byte streams similar to pipes, except that they are full-duplex instead of half-duplex. A stream socket must be in a *connected* state before any data can be sent or received on it. A connection to another socket is created with a connect() or accept() call. Once connected, data can be transferred using some variant of the send() and recv() or the read() and write() calls.

When a session is complete, use close() or shutdown() calls to terminate the connection.

TCP, the communications protocol used to implement SOCK_STREAM for AF_INET sockets, ensures that data is not lost or duplicated. If a peer has buffer space for data and the data cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and the next recv() call indicates an error with errno set to [ETIMEDOUT]. If SO_KEEPALIVE is set and the connection has been idle for two hours, the TCP protocol sends "keepalive" packets every 75 seconds to determine whether the connection is active. These transmissions are not visible to users and cannot be read by a recv() call. If the remote system does not respond within 10 minutes (i.e., after 8 "keepalive" packets have been sent), the next socket call (e.g., recv()) returns an error with errno set to [ETIMEDOUT]. A SIGPIPE signal is raised if a process sends on a broken stream. This causes naive processes that do not handle the signal to exit. An end-of-file condition (zero bytes read) is returned if a process tries to read on a broken stream.

SOCK_DGRAM sockets allow sending of messages to correspondents named in send() calls. It is also possible to receive messages at such a socket with recv().

The operation of sockets is controlled by socket level options set by the setsockopt() system call described by the *getsockopt*(2) manual entry. These options are defined in the file <sys/socket.h> and explained in the *getsockopt*(2) manual entry.

X.25 Only

Socket endpoints for communication over an X.25/9000 link can be in either address family, AF_INET or AF_CCITT. If the socket is in the AF_INET family, the connection behaves as described above. TCP is used if the socket type is SOCK_STREAM. UDP is used if the socket type is SOCK_DGRAM. In both cases, Internet protocol (IP) and the X.25-to-IP interface module are used.

If the socket is in the AF_CCITT address family, only the SOCK_STREAM socket type is supported. Refer to the topic "Comparing X.25 Level 3 Access to IP" in the *X.25 Programmer's Guide* for more details on the difference between programmatic access to X.25 via IP and X.25 Level 3.

If the socket is in the AF_CCITT family, the connection and all other operations pass data directly from the application to the X.25 Packet Level (level 3) without passing through a TCP or UDP protocol. Connections of the AF_CCITT family cannot use most of the socket level options described in *getsockopt*(2). However, AF_CCITT connections can use many X.25-specific ioctl() calls, described in *socketx25*(7).

DEPENDENCIES

AF_CCITT

Only the SOCK_STREAM type is supported.

RETURN VALUE

socket() returns the following values:

n Successful completion. *n* is a valid file descriptor referring to the socket.

-1 Failure. errno is set to indicate the error.

ERRORS

If socket() fails, errno is set to one of the following values.

[EAFNOSUPPORT] The specified address family is not supported in this version of the system.

[EHOSTDOWN] The networking subsystem is not up.

[EINVAL] SOCK_DGRAM sockets are currently not supported for the AF_UNIX address family.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system's table of open files is temporarily full and no more socket()calls can be accepted.

[ENOBUFS] No buffer space is available. The socket cannot be created.

[EPROTONOSUPPORT] The specified protocol is not supported.

[EPROTOTYPE] The type of socket and protocol do not match.

[ESOCKTNOSUPPORT] The specified socket type is not supported in this address family.

[ETIMEDOUT] Connection timed out.

AUTHOR

socket() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), af_ccitt(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

socket(): XPG4

NAME

getsockname - get socket address

SYNOPSIS

#include <sys/socket.h>

AF_CCITT only:

#include <x25/x25addrstr.h>

int getsockname(int s, void *addr, int *addrlen);

_XOPEN_SOURCE_EXTENDED only

int getsockname(int s, struct sockaddr *addr, size_t *addrlen);

DESCRIPTION

getsockname() returns the local address of the socket indicated by *s*, where *s* is a socket descriptor. *addr* points to a socket address structure in which this address is returned. *addrlen* points to an int which should be initialized to indicate the size of the address structure. On return it contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the socket, only the first *addrlen* bytes of the address are returned.

AF_CCITT only:

The x25_host[] field of the *addr* struct returns the X.25 addressing information of the local socket *s*. The x25ifname[] field of the *addr* struct contains the name of the local X.25 interface through which the call arrived.

RETURN VALUE

Upon successful completion, getsockname() returns 0; otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

getsockname() fails if any of the following conditions are encountered:

[EBADF] *s* is not a valid file descriptor.

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[ENOBUFS] No buffer space is available to perform the operation.

[EFAULT] *addr* or *addrlen* are not valid pointers.

[EINVAL] The socket has been shut down.

[EOPNOTSUPP] Operation not supported for AF_UNIX sockets.

AUTHOR

getsockname() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

bind(2), socket(2), getpeername(2), inet(7F), af_ccitt(7F), xopen_networking(7).

STANDARDS CONFORMANCE

getsockname(): XPG4

NAME

bind - bind an address to a socket

SYNOPSIS

```
#include <sys/socket.h>
AF_CCITT only
        #include <x25/x25addrstr.h>
AF_INET only
        #include <netinet/in.h>
AF_UNIX only
        #include <sys/un.h>
        int bind(int s, const void *addr, int addrlen);
_XOPEN_SOURCE_EXTENDED only
        int bind(int s, const struct sockaddr *addr, size_t addrlen);
```

DESCRIPTION

The bind() system call assigns an address to an unbound socket. When a socket is created with socket(), it exists in an address space (address family) but has no address assigned. bind() causes the socket whose descriptor is *s* to become bound to the address specified in the socket address structure pointed to by *addr*.

addrlen must specify the size of the address structure. Since the size of the socket address structure varies between socket address families, the correct socket address structure should be used with each address family (for example, struct sockaddr_in for AF_INET, and struct sockaddr_un for AF_UNIX). Typically, the sizeof() function is used to pass this value in the bind() call (for example, sizeof(struct sockaddr_in)).

The rules used in address binding vary between communication domains. For example, when binding an AF_UNIX socket to a path name (such as /tmp/mysocket), an open file having that name is created in the file system. When the bound socket is closed, that file still exists unless it is removed or unlinked. When binding an AF_INET socket, *sin_port* can be a port number or it can be zero. If *sin_port* is zero, the system assigns an unused port number automatically.

RETURN VALUE

bind() returns the following values:

0	Successful completion.
-1	Failure. errno is set to indicate the error.

ERRORS

If bind() fails, errno is set to one of the following values.

[EACCES]	The requested address is protected, and the current user has inadequate permission to access it. (This error can be returned by AF_INET only.)
[EADDRINUSE]	The specified address is already in use.
[EADDRNOTAVAIL]	The specified address is invalid or not available from the local machine, or for AF_CCITT sockets which use "wild card" addressing, the specified address space overlays the address space of an existing bind.
[EAFNOSUPPORT]	The specified address is not a valid address for the address family

of this socket.

[EBADF] *s* is not a valid file descriptor.

[EDESTADDRREQ] No *addr* parameter was specified.

[EFAULT] *addr* is not a valid pointer.

[EINVAL] The socket is already bound to an address, the socket has been shut down, *addrlen* is a bad value, or an attempt was made to bind() an AF_UNIX socket to an NFS-mounted (remote) name.

AF_CCITT: The protocol-ID length is negative or greater than 8, the X.121 address string contains an illegal character, or the X.121 address string is greater than 15 digits long.

[ENETDOWN] The *x25ifname* field name specifies an interface that was shut down, or never initialized, or whose Level 2 protocol indicates that the link is not working: Wires might be broken, the interface hoods on the modem are broken, the modem failed, the phone connection failed (this error can be returned by AF_CCITT only), noise interfered with the line for a long period of time.

[ENETUNREACH] The X.25 Level 2 protocol is down. The X.25 link is not working: Wires might be broken, or connections are loose on the interface hoods at the modem, the modem failed, or noise interfered with the line for an extremely long period of time.

[ENOBUFS] No buffer space is available. The bind() cannot complete.

[ENODEV] The *x25ifname* field name specifies a nonexistent interface. (This error can be returned by AF_CCITT only.)

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[EOPNOTSUPP] The socket referenced by *s* does not support address binding.

AUTHOR

bind() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

connect(2), getsockname(2), listen(2), socket(2), af_ccitt(7F), inet(7F), socketx25(7), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

bind(): XPG4

NAME

connect - initiate a connection on a socket

SYNOPSIS

```
#include <sys/socket.h>
AF_CCITT only
     #include <x25/x25addrstr.h>
AF_INET only
     #include <netinet/in.h>
AF_UNIX only
     #include <sys/un.h>
     int connect(int s, const void *addr, int addrlen);
_XOPEN_SOURCE_EXTENDED only
     int connect(int s, const struct sockaddr *addr, size_t addrlen);
```

DESCRIPTION

The connect() function initiates a connection on a socket. *s* is a socket descriptor.

addr is a pointer to a socket address structure containing the address of a remote socket to which a connection is to be established.

addrlen is the size of this address structure. Since the size of the socket address structure varies among socket address families, the correct socket address structure should be used with each address family (for example, struct sockaddr_in for AF_INET and struct sockaddr_un for AF_UNIX). Typically, the sizeof() function is used to pass this value (for example, sizeof(struct sockaddr_in)).

If the socket is of type SOCK_DGRAM, connect() specifies the peer address to which messages are to be sent, and the call returns immediately. Furthermore, this socket can only receive messages sent from this address.

If the socket is of type SOCK_STREAM, connect() attempts to contact the remote host to make a connection between the remote socket (peer) and the local socket specified by *s*. The call normally blocks until the connection completes. If nonblocking mode has been enabled with the O_NONBLOCK or O_NDELAY fcntl() flags or the FIOSNBIO ioctl() request and the connection cannot be completed immediately, connect() returns an error as described below. In these cases, select() can be used on this socket to determine when the connection has completed by selecting it for writing.

The connect() system call will complete if remote program has a pending listen() even though remote program had not yet issued an accept() system call.

O_NONBLOCK and O_NDELAY are defined in <sys/fcntl.h> and explained in *fcntl*(2), *fcntl*(5), and *socket*(7). FIOSNBIO is defined in <sys/ioctl.h> and explained in *ioctl*(2), *ioctl*(5), and *socket*(7).

If *s* is a SOCK_STREAM socket that is bound to the same local address as another SOCK_STREAM socket, connect() returns [EADDRINUSE] if *addr* is the same as the peer address of that other socket. This situation can only happen if the SO_REUSEADDR option has been set on *s*, which is an AF_INET socket (see *getsockopt*(2)).

If the AF_INET socket does not already have a local address bound to it (see *bind*(2)), connect() also binds the socket to a local address chosen by the system.

Generally, stream sockets may successfully connect only once; datagram sockets may use connect() multiple times to change the peer address. For datagram sockets, a side effect of attempting to connect to some invalid address (see ERRORS below) is that the peer address is no longer maintained by the system. An example of an invalid address for a datagram socket is *addrlen* set to 0 and *addr* set to any value.

AF_CCITT Only

Use the x25addrstr struct for the address structure. The caller must know the X.121 address of the DTE to which the connection is to be established, including any subaddresses or protocol IDs that may be needed. Refer to *af_ccitt*(7F) for a detailed description of the x25addrstr address structure. If address-matching by protocol ID, specify the protocol ID with the X25_WR_USER_DATA ioctl() call before issuing the connect() call. The X25_WR_USER_DATA ioctl() call is described in *socketx25*(7).

DEPENDENCIES

AF_CCITT

The SO_REUSEADDR option to setsockopt() is not supported for sockets in the AF_CCITT address family.

RETURN VALUE

connect() returns the following values:

0 Successful completion.

-1 Failure. errno is set to indicate the error.

ERRORS

If connect() fails, errno is set to one of the following values.

[EADDRINUSE] The specified address is already in use. For datagram sockets, the peer address is no longer maintained by the system.

[EADDRNOTAVAIL] The specified address is not available on this machine, or the socket is a TCP/UDP socket and the zero port number is specified.

For datagram sockets, the peer address is no longer maintained by the system.

[EAFNOSUPPORT] The specified address is not a valid address for the address family of this socket.

For datagram sockets, the peer address is no longer maintained by the system.

[EALREADY] Nonblocking I/O is enabled with O_NONBLOCK, O_NDELAY, or FIOSNBIO, and a previous connection attempt has not yet completed.

[EBADF] *s* is not a valid file descriptor.

[ECONNREFUSED] The attempt to connect was forcefully rejected.

[EFAULT] *addr* is not a valid pointer.

[EINPROGRESS] Nonblocking I/O is enabled using O_NONBLOCK, O_NDELAY, or FIOSNBIO, and the connection cannot be completed immediately. This is not a failure. Make the connect() call again a few seconds later. Alternatively, wait for completion by calling select() and selecting for write.

[EINTR] The connect was interrupted by a signal before the connect sequence was complete. The building of the connection still takes place, even though the user is not blocked on the connect() call.

[EINVAL] The socket has already been shut down or has a listen() active on it;

addrlen is a bad value; an attempt was made to connect() an AF_UNIX socket to an NFS- mounted (remote) name; the X.121 address length is zero, negative, or greater than 15 digits.

For datagram sockets, if *addrlen* is a bad value, the peer address is no longer maintained by the system.

[EISCONN] The socket is already connected.

[ENETDOWN] The X.25 interface specified in the *addr* struct was found but was not in the initialized state. *x25ifname* field name is an interface which has been shut down or never initialized or suffered a power failure which erased its state information.

[ENETUNREACH] The network is not reachable from this host. For AF_CCITT only: X.25 Level 2 is down. The X.25 link is not working: wires might be broken, connections are loose on the interface hoods at the modem, the modem failed, or noise interfered with the line for an extremely long period of time.

[ENOBUFS] No buffer space is available. The connect() has failed.

[ENODEV] The *x25ifname* field refers to a nonexistent interface.

[ENOSPC] All available virtual circuits are in use.

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket. [EOPNOTSUPP]

The socket referenced by *s* does not support connect(). With X.25 an attempt was made to issue a connect() call on a listen() socket.

[ETIMEDOUT] Connection establishment timed out without establishing a connection. One reason could be that the connection requests queue at the remote socket may be full (see listen(2)).

AUTHOR

connect() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

accept(2), getsockname(2), select(2), socket(2), af_ccitt(7F), socket(7), socketx25(7), xopen_networking(7).

NAME

accept - accept a connection on a socket

SYNOPSIS

#include <sys/socket.h>

AF_CCITT only

#include <x25/x25addrstr.h>

int accept(int s, void *addr, int *addrlen);

_XOPEN_SOURCE_EXTENDED only

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

DESCRIPTION

The accept() system call is used with connection-based socket types, such as SOCK_STREAM. The argument, *s*, is a socket descriptor created with socket(), bound to a local address by bind(), and listening for connections after a listen(). accept() extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *s*, and returns a new file descriptor, *ns*, for the socket.

If no pending connections are present on the queue and nonblocking mode has not been enabled with the fcntl() O_NONBLOCK or O_NDELAY flags or the ioctl() FIOSNBIO request, accept() blocks the caller until a connection is present. O_NONBLOCK and O_NDELAY are defined in <sys/fcntl.h> (see *fcntl*(2) *fcntl*(5), and *socket*(7)). FIOSNBIO and the equivalent request FIONBIO are defined in <sys/ioctl.h>, although use of FIONBIO is not recommended (see *ioctl*(2), *ioctl*(5), and *socket*(7)).

If the socket has nonblocking mode enabled and no pending connections are present on the queue, accept() returns an error as described below. The accepted socket, *ns*, cannot be used to accept more connections. The original socket *s* remains open for incoming connection requests. To determine whether a listening socket has pending connection requests ready for an accept() call, use select() for reading.

The argument *addr* should point to a socket address structure. The accept() call fills in this structure with the address of the connecting entity, as known to the underlying protocol. In the case of AF_UNIX sockets, the peer's address is filled in only if the peer had done an explicit bind() before doing a connect(). Therefore, for AF_UNIX sockets, in the common case, when the peer had not done an explicit bind() before doing a connect(), the structure is filled with a string of nulls for the address. The format of the address depends upon the protocol and the address-family of the socket *s*. *addrlen* is a pointer to an int; it should initially contain the size of the structure pointed to by *addr*. On return, it contains the actual length (in bytes) of the address returned. If the memory pointed to by *addr* is not large enough to contain the entire address, only the first *addrlen* bytes of the address are returned. If *addr* is NULL or *addrlen* contains 0, then the connecting entity's address will not be returned.

The fcntl() O_NONBLOCK and O_NDELAY flags and ioctl() FIOSNBIO request are all supported. These features interact as follows:

- If the O_NONBLOCK or O_NDELAY flag has been set, accept() requests behave accordingly, regardless of any FIOSNBIO requests.
- If neither the O_NONBLOCK flag nor the O_NDELAY flag has been set, FIOSNBIO requests control the behavior of accept().

AF_CCITT only

The *addr* parameter to accept() returns addressing information for the connecting entity, except for the x25ifname[] field of *addr* which contains the name of the local X.25 interface through which the connection request arrived. Call-acceptance can be controlled with the ioctl() X25_CALL_ACPT_APPROVAL request (see *socketx25*(7)).

RETURN VALUE

Upon successful completion, accept() returns a nonnegative integer which is a descriptor for the accepted socket.

If an error occurs, accept() returns -1 and sets errno to indicate the cause.

ERRORS

If accept() fails, errno is set to one of the following values:

[EAGAIN] Nonblocking I/O is enabled using O_NONBLOCK and no connections are present to be accepted.

[EBADF] The argument, *s*, is not a valid file descriptor.

[EFAULT] The *addr* parameter is not a valid pointer.

[EINTR] The call was interrupted by a signal before a valid connection arrived.

[EINVAL] The socket referenced by *s* is not currently a listen socket or has been shut down with shut down(). A listen() must be done before an accept() is allowed.

[EMFILE] The maximum number of file descriptors for this process are currently open.

[ENFILE] The system's table of open files is full and no more accept() calls can be processed at this time.

[ENOBUFS] No buffer space is available. The accept() cannot complete. The queued socket connect request is aborted.

[ENOTSOCK] The argument, *s*, is a valid file descriptor, but it is not a socket.

[EOPNOTSUPP] The socket referenced by *s* does not support accept().

[EWOULDBLOCK]Nonblocking I/O is enabled using O_NDELAY or FIOSNBIO and no connections are present to be accepted.

AUTHOR

accept() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Openspecification(see *xopen_networking*(*7*)).

SEE ALSO

bind(2),connect(2),listen(2),select(2),socket(2)socketx25(7),xopen_networking(7).

STANDARDS CONFORMANCE

accept(): XPG

NAME

listen - listen for connections on a socket

SYNOPSIS

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

DESCRIPTION

To accept connections, a socket is first created using socket(), a queue for incoming connections is activated using listen(), and then connections are accepted using accept(). listen() applies only to unconnected sockets of type SOCK_STREAM. If the socket has not been bound to a local port before listen() is invoked, the system automatically binds a local port for the socket to listen on (see *inet*(7F)). For sockets in the address family AF_CCITT, the socket *must* be bound to an address by using bind() before connection establishment can continue, otherwise an EADDREQUIRED error is returned.

A listen queue is established for the socket specified by the *s* parameter, which is a socket descriptor. *backlog* defines the desirable queue length for pending connections. The actual queue length may be greater than the specified *backlog* . If a connection request arrives when the queue is full, the client will receive an ETIMEDOUT error. *backlog* is limited to the range of 0 to SOMAXCONN, which is defined in <sys/socket.h>. SOMAXCONN is currently set to 20. If any other value is specified, the system automatically assigns the closest value within the range. A *backlog* of 0 specifies only 1 pending connection is allowed at any given time.

DEPENDENCIES

AF_CCITT:

Call-acceptance can be controlled by the X25_CALL_ACPT_APPROVAL ioctl() call described in RETURN VALUE . Upon successful completion, listen() returns 0; otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

listen() fails if any of the following conditions are encountered:

[EBADF] *s* is not a valid file descriptor.

[EDESTADDRREQ] The socket *s* has not been bound to an address by using bind() .

[ENOTSOCK] *s* is a valid file descriptor but it is not a socket.

[EOPNOTSUPP] The socket referenced by *s* does not support listen() .

[EINVAL] he socket has been shut down or is already connected (see *socketx25*(7)).

AUTHOR

listen() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

accept(2),connect(2),socket(2),socketx25(7),af_ccitt(7F),inet(7F), xopen_networking(7).

STANDARDS CONFORMANCE

listen(): XPG4

NAME

send(), sendmsg(), sendto() - send a message from a socket

SYNOPSIS

```
#include <sys/socket.h>
    int send(int s, const void *msg, int len, int flags);
    int sendto(
        int         s,
        const void *msg,
        int         len,
        int         flags,
        const void *to,
        int         tolen
    );
    int sendmsg(int s, const struct msghdr msg[], int flags);
    _XOPEN_SOURCE_EXTENDED only
    ssize_t send(int s, const void *msg, size_t len, int flags);
    ssize_t sendto(
    int                 s,
        const void      *msg,
        size_t              len,
        int                flags,
        const struct sockaddr *to,
        size_t              tolen
    );
    ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

DESCRIPTION

The send(), sendmsg(), and sendto() system calls transmit a message to another socket. send() can be used only when the socket is in a connected state, whereas sendmsg() and sendto() can be used at any time. sendmsg() allows the send data to be gathered from several buffers specified in the msghdr structure. See *recv*(2) for a description of the msghdr structure. *s* is a socket descriptor that specifies the socket on which the message will be sent.

msg points to the buffer containing the message.

If the socket uses connection-based communications, such as a SOCK_STREAM socket, these calls can only be used after the connection has been established (see *connect*(2)). In this case, any destination specified by *to* is ignored. For connectionless sockets, such as SOCK_DGRAM, sendto() must be used unless the destination address has already been specified by connect(). If the destination address has been specified and sendto() is used, an error results if any address is specified by *to*.

The address of the target socket is contained in a socket address structure pointed to by *to* with *tolen* specifying the size of the structure.

If a sendto() is attempted on a SOCK_DGRAM socket before any loca address has been bound to it, the system automatically selects a local address to be used for the message. In this case, there is no guarantee that the same local address will be used for successive sendto() requests on the same socket.

The length of the message is given by *len* in bytes. The length of data actually sent is returned. If the message is too long to pass atomically through the underlying protocol, the message is not transmitted, -1 is returned, and errno is set to [EMSGSIZE]. For SOCK_DGRAM sockets, this size is fixed by the implementation (see the DEPENDENCIES section). Otherwise there is no size limit.

When send() or sendto() returns a positive value, it only indicates this number of bytes have been sent to the local transport provider. It does not mean this number of bytes have been delivered to the peer socket application. A SOCK_DGRAM socket does not guarantee end-to-end delivery. A SOCK_STREAM socket guarantees eventual end-to-end delivery, however its underlying transport provider may later detect an irrecoverable error and returns a value of -1 at another socket function call.

When send() or sendto() returns a value of -1 , it indicates a locally detected error. errno is set to indicate the error.

If no buffer space is available to hold the data to be transmitted, send() blocks unless nonblocking mode is enabled. The three ways to enable nonblocking mode are:

- with the FIOSNBIO ioctl() request,
- with the O_NONBLOCK flag, and
- with the O_NDELAY fcntl() flag.

If nonblocking I/O is enabled using FIOSNBIO or the equivalent FIONBIO request (defined in <sys/ioctl.h> and explained in *ioctl*(2), *ioctl*(5), and *socket*(7)), although the use of FIONBIO is not recommended, the send() request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, send() completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, send() completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, send() fails, having written no data, and errno is set to [EWOULDBLOCK].

If nonblocking I/O is disabled using FIOSNBIO, send() always executes completely (blocking as necessary) and returns the number of bytes written.

If the O_NONBLOCK flag is set using fcntl() (defined in <sys/fcntl.h> and explained in *fcntl*(2) and *fcntl*(5)), POSIX-style nonblocking I/O is enabled. In this case, the send() request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, send() completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, send() completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, send() completes, having written no data, and returns -1, with errno set to [EAGAIN].

If the O_NDELAY flag is set using fcntl() (defined in <sys/fcntl.h> and explained in *fcntl*(2) and *fcntl*(5)), nonblocking I/O is enabled. In this case, the send() request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, send() completes successfully, having written out all of the data, and returns the number of bytes written.

- If there is not enough space in the buffer to write out the entire request, send() completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, send() completes successfully, having written no data, and returns 0.

If the O_NDELAY flag is cleared using fcntl(), nonblocking I/O is disabled. In this case, the send() always executes completely (blocking as necessary) and returns the number of bytes written.

Since the fcntl() O_NONBLOCK and O_NDELAY flags and ioctl() FIOSNBIO requests are supported, the following clarifies on how these features interact. If the O_NONBLOCK or O_NDELAY flag has been set, send() requests behave accordingly, regardless of any FIOSNBIO requests. If neither the O_NONBLOCK flag nor the O_NDELAY flag has been set, FIOSNBIO requests control the behavior of send().

By default nonblocking I/O is disabled.

The supported values for *flags* are zero or MSG_OOB (to send out-of- band data). A write() call made to a socket behaves in exactly the same way as send() with *flags* set to zero. MSG_OOB is not supported for AF_UNIX sockets.

select(2) can be used to determine when it is possible to send more data.

AF_CCITT Only

Sockets of the address family AF_CCITT operate in message mode.

Although they are specified as connection-based (SOCK_STREAM) sockets, the X.25 subsystem communicates via messages. They require that a connection be established with the connect() or accept() calls.

The O_NDELAY flag is not supported. Use FIOSNBIO requests to control nonblocking I/O. If the available buffer space is not large enough for the entire message and the socket is in nonblocking mode, errno is set to [EWOULDBLOCK]. If the amount of data in the send() exceeds the maximum outbound message size, errno is set to [EMSGSIZE].

The sendto() call is not supported.

Each call sends either a complete or a partial X.25 message. This is controlled by the setting of the More-Data-To-Follow (MDTF) bit. If the user wants to send a partial message, MDTF should be set to 1 before the send() call. The MDTF bit should be cleared to 0 before sending the final message fragment.

Message fragment length may range from 0 bytes up to the size of the socket's send buffer (see *af_ccitt*(7F)). The MDTF bit and multiple send() calls can be combined to transmit complete X.25 packet sequences (i.e., zero or more DATA packets in which the More Data bit is set, followed by one DATA packet in which the More Data bit is clear) of arbitrary length. Note that a 0-byte message is not actually sent, but may be necessary to flush a complete X.25 message if the user is controlling the MDTF bit.

Sockets of the AF_CCITT address family can send 1 byte of out-of-band data (known as an INTERRUPT data packet in X.25 terminology), or up to 32 bytes if the X.25 interface is configured for 1984 CCITT X.25 recommendations. INTERRUPT data packets sent in blocking mode cause the process to block until confirmation is received. INTERRUPT data packets sent with the socket in nonblocking mode do not cause the process to block; instead, an out-of-band message is queued to the socket when the INTERRUPT confirmation packet is received (see *recv*(2)).

_XOPEN_SOURCE_EXTENDED only X/Open Sockets msghdr has the following form :

```
struct msghdr {
        void        *msg_name;          /* optional address */
        size_t      msg_namelen;        /* size of address   */
        struct      iovec *msg_iov;     /* scatter array for data */
        int         msg_iovlen;         /* # of elements in msg_iov */
        void        *msg_control;       /* ancillary data, see below */
        size_t      msg_controllen;     /* ancillary data buffer len */
        int         msg_flags;          /* flags on received message */
    }
```

msg_control specifies a buffer of ancillary data to send along with the message. Ancillary data consists of a sequence of pairs, each consisting of a *cmsghdr* structure followed by a data array. The data array contains the ancillary data message, and the *cmsghdr* structure contains descriptive information that allows an application to correctly parse the data. *cmsghdr* has the following structure:

```
struct cmsghdr {
        size_t   cmsg_len;          /* data byte count, including hdr*/
        int      cmsg_level;        /* originating protocol */
        int      cmsg_type;         /* protocol-specific type */
    }
```

The supported value for cmsg_level is SOL_SOCKET. and the supported value for cmsg_type is SCM_RIGHTS. Together they indicate the data array contains the access rights to be sent. Access rights are supported only for AF_UNIX. Access rights are limited to file descriptors of size *int*. If ancillary data are not being transferred, set the *msg_control* field to NULL and set the *msg_controllen* field to 0.

The *msg_flags* member is ignored.

RETURN VALUE

send(), sendmsg(), and sendto() return the following values:

n Successful completion. *n* is the number of bytes sent.

-1 Failure. errno is set to indicate the error.

ERRORS

If send(), sendmsg(), or sendto() fails, errno is set to one of the following values.

[EACCES] Process doing a send() of a broadcast packet does not have broadcast capability enabled for the socket. Use setsockopt() to enable broadcast capability.

[EAFNOSUPPORT] The specified address is not a valid address for the address family of this socket.

[EAGAIN] Nonblocking I/O is enabled using the O_NONBLOCK flag with fcntl(), and the requested operation would block, or the socket has an error that was set asynchronously. An asynchronous error can be caused by a gateway failing to forward a datagram from this socket because the datagram exceeds the MTU of the next-hop network and the "Don't Fragment" (DF) bit in the datagram is set. (See SO_PMTU in *getsockopt*(2)).

[EBADF] *s* is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EDESTADDRREQ] The *to* parameter needs to specify a destination address for the message. This is also given if the specified address contains unspecified fields (see *inet*(7F)).

[EFAULT] An invalid pointer was specified in the *msg* or *to* parameter, or in the msghdr structure.

[EINTR] The operation was interrupted by a signal before any data was sent. (If some data was sent, send() returns the number of bytes sent before the signal, and [EINTR] is not set).

[EINVAL] The *len* or *tolen* parameter, or a length in the msghdr structure is invalid. A sendto() system call was issued on an X.25 socket, or the connection is in its reset sequence and cannot accept data.

[EIO] A timeout occurred.

[EISCONN] An address was specified by *to* for a SOCK_DGRAM socket which is already connected.

[EMSGSIZE] A length in the msghdr structure is invalid. The socket requires that messages be sent atomically, and the size of the message to be sent made this impossible.

SOCK_DGRAM/AF_INET or SOCK_STREAM/AF_CCITT:The message size exceeded the outbound buffer size.

[ENETDOWN] The interface used for the specified address is "down" (see *ifconfig*(1M)), no interface for the specified address can be found (SO_DONTROUTE socket option in use), or the X.25 Level 2 is down.

[EHOSTUNREACH] The destination host is not reachable.

[ENETUNREACH] The destination network is not reachable. Some of the possible causes for this error are:(LAN) Allencapsulations (e.g., ether, ieee) have been turned off (see also *lanconfig*(1M), and *ifconfig*(1M)).

(X.25) The X.25 Level 2 is down. The X.25 link layer is not working (wires might be broken, connections are loose on the interface hoods at the modem, the modem failed, the packet switch at the remote end lost power or failed for some reason, or electrical noise interfered with the line for an extremely long period of time).

[ENOBUFS] No buffer space is available in the system toperform the operation.

[ENOTCONN] A send() on a socket that is not connected, or a send() on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[EOPNOTSUPP] The MSG_OOB flag was specified; it is not supported for AF_UNIX sockets.

[EPIPE] and SIGPIPE signal

An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. Note that the default action for SIGPIPE, unless the process has established a signal handler for this signal, is to terminate the process.

[EWOULDBLOCK] Nonblocking I/O is enabled using ioctl() FIOSNBIO request and the requested operation would block.

DEPENDENCIES

UDP messages are fragmented at the IP level into Maximum Transmission

Unit (MTU) sized pieces; MTU varies for different link types. These pieces, called IP fragments, can be transmitted, but IP does not guarantee delivery. Sending large messages may cause too many fragments and overrun a receiver's ability to receive them. If this happens the complete message cannot be reassembled. This affects the apparent reliability and throughput of the network as viewed by the end user.

The default and maximum buffer sizes are protocol-specific. Refer to the appropriate entries in Sections 7F and 7P for details. The buffer size can be set by calling setsockopt() with SO_SNDBUF.

AF_CCITT

If the receiving process is on a Series 700/800 HP-UX system and the connection has been set up to use the D-bit, data sent with the D-bit set is acknowledged when the receiving process has read the data. Otherwise, the acknowledgement is sent when the firmware receives it.

AUTHOR

send() was developed at the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD

Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

ifconfig(1M), lanconfig(1M), getsockopt(2), recv(2), select(2), setsockopt(2), socket(2), socket(7), socketx25(7), af_ccitt(7F), inet(7F), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

send(): XPG4

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

```
#include <sys/socket.h>
    int recv(int s, void *buf, int len, int flags);
    int recvfrom(
        int       s,
        void      *buf,
        int       len,
        int       flags,
        void      *from,
        int       *fromlen
        );
    int recvmsg(int s, struct msghdr msg[], int flags);
    _XOPEN_SOURCE_EXTENDED only
    ssize_t recv(int s, void *buf, size_t len, int flags);
    ssize_t recvfrom(
    int             s,
        void        *buf,
        size_t          len,
        int             flags,
        struct sockaddr *from,
        size_t            *fromlen
        );
    ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

DESCRIPTION

The recv(), recvfrom(), and recvmsg() system calls are used to receive messages from a socket.

s is a socket descriptor from which messages are received.

buf is a pointer to the buffer into which the messages are placed.

len is the maximum number of bytes that can fit in the buffer referenced by *buf*.

If the socket uses connection-based communications, such as aSOCK_STREAM socket, these calls can only be used after the connectionhas been established (see *connect*(2)). For connectionless socketssuch as SOCK_DGRAM, these calls can be used whether a connection hasbeen specified or not.

recvfrom() operates in the same manner as recv() except that it is able to return the address of the socket from which the message was sent. For connected datagram sockets, recvfrom() simply returns the same address as getpeername() (see *getpeername*(2)). For stream sockets, recvfrom() retrieves data in the same manner as recv(), but does not return the socket address of the sender. If *from* is nonzero, the source address of the message is placed in the socket address structure pointed to by *from*. *fromlen* is a value-result parameter, initialized to the size of the structure associated with *from*, and modified on return to indicate the actual size of the address stored there. If the memory pointed to by *from* is not large enough to contain the entire address, only the first *fromlen* bytes of the address are returned.

For message-based sockets such as SOCK_DGRAM, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffer, the excess bytes are discarded. For stream-based sockets such as SOCK_STREAM, there is no concept of message boundaries. In this case, data is returned to the user as soon as it becomes available, and no data is discarded. See the AF_CCITT Only subsection below for a list of the exceptions to this behavior for connections in the address family AF_CCITT.

recvmsg() performs the same action as recv(), but scatters the read data into the buffers specified in the msghdr structure (see _XOPEN_SOURCE_EXTENDED only below). This structure is defined in <sys/socket.h>, and has the following form :

HP-UX BSD Sockets only

```
struct msghdr {
        caddr_t    msg_name;            /* optional address */
        int        msg_namelen;         /* size of address  */
        struct     iovec *msg_iov;      /* scatter array for data */
        int        msg_iovlen;          /* # of elements in msg_iov */
        caddr_t    msg_accrights;       /* access rights */
        int        msg_accrightslen;    /* size of msg_accrights */
        }
```

msg_name points to a sockaddr structure in which the address of the sending socket is to be stored, if the socket is connectionless; *sg_name* may be a null pointer if no name is specified. *msg_iov* specifies the locations of the character arrays for storing the incoming data. *msg_accrights* specifies a buffer to receive any access rights sent along with the message. Access rights are limited to file descriptors of size *int*. If access rights are not being transferred, set the *msg_accrights* field to NULL. Access rights are supported only for AF_UNIX.

If no data is available to be received, recv() waits for a message to arrive unless nonblocking mode is enabled. There are three ways to enable nonblocking mode:

- With the FIOSNBIO ioctl() request
- With the O_NONBLOCK fcntl() flag
- With the O_NDELAY fcntl() flag

Although the use of FIONBIO is not recommended, if nonblocking I/O is enabled using FIOSNBIO or the equivalent FIONBIO request (defined in <sys/ioctl.h> and explained in *ioctl*(2), *ioctl*(5) and *socket*(7)), the recv() request completes in one of three ways:

- If there is enough data available to satisfy the entire request, recv() completes successfully, having read all of the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, recv() complete successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, recv() fails and errno is set to [EWOULDBLOCK].

If nonblocking I/O is disabled using FIOSNBIO, recv() always executes completely (blocking as necessary) and returns the number of bytes read.

If the O_NONBLOCK flag is set using fcntl() (defined in <sys/fcntl.h> and explained in *fcntl*(2) and *fcntl*(5)), POSIX-style nonblocking I/O is enabled. In this case, the recv() request completes in one of three ways

- If there is enough data available to satisfy the entire request,recv() completes successfully, having read all the data, and returns the number of bytes read.

- If there is not enough data available to satisfy the entire request, recv() completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, recv() completes, having read no data, and returns -1 with errno set to [EAGAIN].

If the O_NDELAY flag is set using fcntl() (defined in <sys/fcntl.h> and explained in *fcntl*(2) and *fcntl*(5)), nonblocking I/O is enabled. In this case, the recv() request completes in one of three ways:

- If there is enough data available to satisfy the entire request, recv() completes successfully, having read all the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, recv() completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, recv() completes successfully, having read no data, and returns 0.

If the O_NONBLOCK or O_NDELAY flag is cleared using fcntl(), the corresponding style of nonblocking I/O, if previously enabled, is disabled. In this case, recv() always executes completely (blocking as necessary) and returns the number of bytes read.

Since both the fcntl() O_NONBLOCK and O_NDELAY flags and ioctl()

FIOSNBIO request are supported, some clarification on how these features interact is necessary. If the O_NONBLOCK or O_NDELAY flag has been set, recv() requests behave accordingly, regardless of any FIOSNBIO requests. If neither the O_NONBLOCK flag nor the O_NDELAY flag has been set, FIOSNBIO requests control the the behavior of recv() .

By default nonblocking I/O is disabled.

select() can be used to determine when more data arrives by selecting the socket for reading.

The *flags* parameter can be set to MSG_PEEK, MSG_OOB, both, or zero.

If it is set to MSG_PEEK, any data returned to the user still is treated as if it had not been read. The next recv() rereads the same data. The MSG_OOB flag is used to receive out-of-band data. For TCP SOCK_STREAM sockets, both the MSG_PEEK and MSG_OOB flags can be set at the same time. The MSG_OOB flag value is supported for TCP SOCK_STREAM sockets only. MSG_OOB is not supported for AF_UNIX sockets.

A read() call made to a socket behaves in exactly the same way as a recv() with *flags* set to zero.

AF_CCITT Only Connections in the address family AF_CCITT support message-based sockets only. Although the user specifies connection-based communications (SOCK_STREAM), the X.25 subsystem communicates via messages. This address family does not support SOCK_DGRAM socket types.

Normally, each recv() returns one complete X.25 message. If the socket is in nonblocking mode, recv() behaves as described above. Note that if the user specifies *len* less than the actual X.25 message size, the excess data is discarded and no error indication is returned. The size of the next available message as well as the state of MDTF, D, and Q bits can be obtained with ioctl(X25_NEXT_MSG_STAT).

Connections of the address family AF_CCITT receive data in the same way as message-based connections described above, with the following additions and exceptions:

- recvfrom() is supported; however, the *from* and *fromlen* parameters are ignored (that is, it works in the same manner as recv()).

- To receive a message in fragments of the complete X.25 message, use ioctl(X25_SET_FRAGMENT_SIZE). The state of the MDTF bit is 1 for all except the last fragment of the message.
- The MSG_OOB flag is supported.
- The MSG_PEEK flag is supported; the two flags can be combined.
- If a message is received that is larger than the user-controlled maximum message size (see *af_ccitt*(7F)), the X.25 subsystem RESETs the circuit, discards the data, and sends the out-of-band event OOB_VC_MESSAGE_TOO_BIG to the socket.

DEPENDENCIES

AF_CCITT

recvfrom() is supported; however, the *from* and *fromlen* parameters are ignored (i.e., it works in the same manner as recv()).

The O_NDELAY fcntl() call is not supported over X.25 links. Use the FIOSNBIO ioctl() call instead to enable nonblocking I/0.

_XOPEN_SOURCE_EXTENDED only X/Open Sockets msghdr has the following form :

```
struct msghdr {
        void        *msg_name;          /* optional address */
        size_t      msg_namelen;            /* size of address   */
        struct      iovec *msg_iov;         /* scatter array for data */
        int         msg_iovlen;             /* # of elements in msg_iov */
        void        *msg_control;           /* ancillary data, see below */
        size_t      msg_controllen;         /* ancillary data buffer len */
        int         msg_flags;          /* flags on received message */
        }
```

msg_control specifies a buffer to receive any ancillary data sent along with the message. Ancillary data consists of a sequence of pairs, each consisting of a *cmsghdr* structure followed by a data array. The data array contains the ancillary data message, and the *cmsghdr* structure contains descriptive information that allows an application to correctly parse the data. *cmsghdr* has the following structure:

```
struct cmsghdr {
        size_t  cmsg_len;           /* data byte count, including hdr*/
        int     cmsg_level;         /* originating protocol */
        int     cmsg_type;          /* protocol-specific type */
        }
```

The supported value for cmsg_level is SOL_SOCKET, and the supported value for cmsg_type is SCM_RIGHTS. Together they indicate that the data array contains the access rights to be received. Access rights are supported only for AF_UNIX. Access rights are limited to file descriptors of size *int*. If ancillary data are not being transferred, set the *msg_control* field to NULL and set the *msg_controllen* field to 0.

The *flags* parameter accepts a new value, MSG_WAITALL, which requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, or an error is pending for the socket.

On successful completion of recvmsg(), the *msg_flags* member of the message header is the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message.

MSG_EOR End of record was received(if supported by the protocol).

MSG_OOB Out-of-band data was received.

MSG_TRUNC Normal data was truncated.

MSG_CTRUNC Control data was truncated.

RETURN VALUE

recv(), recvfrom(), and recvmsg() returns the following values:

n Successful completion. *n* is the number of bytes received.

0 The socket is blocking and the transport connection to the remote node failed.

-1 Failure. errno is set to indicate the error.

ERRORS

If recv(), recvfrom(), or recvmsg() fails, errno is set to one of the following values.

[EAGAIN] Non-blocking I/O is enabled using O_NONBLOCK flag with fcntl() and the receive operation would block, or the socket has an error that was set asynchronously. An asynchronous error can be caused by a gateway failing to forward a datagram because the datagram exceeds the MTU of the next-hop network and the "Don't Fragment" (DF) bit in the datagram is set. (See SO_PMTU in *getsockopt*(2).)

[EBADF] The argument *s* is an invalid descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] An invalid pointer was specified in the *buf* , *from* , or *fromlen* parameter, or in the msghdr structure.

[EINTR] The receive was interrupted by delivery of a signal

before any data was available for the receive.

[EINVAL] The *len* parameter or a length in the msghdr structure is invalid; or no data is available on receive of out of band data.

[EMSGSIZE] A length in the msghdr structure is invalid.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOTCONN] Receive on a SOCK_STREAM socket that is not yet connected.

[ENOTSOCK] The argument *s* is a valid file descriptor, but it is not a socket.

[EOPNOTSUPP] The MSG_OOB flag was set for a UDP SOCK_DGRAM message-based socket, or MSG_OOB or MSG_PEEK was set for any AF_UNIX socket. The MSG_OOB flag is supported only for stream-based TCP SOCK_STREAM sockets. Neither MSG_PEEK nor MSG_OOB is supported for AF_UNIX sockets. AF_CCITT only: recv() was issued on a listen() socket.

[ETIMEDOUT] The connection timed out during connection establishment, or due to a transmission timeout on active connection.

[EWOULDBLOCK] Non-blocking I/O is enabled using ioctl() FIOSNBIO request, and the requested operation would block.

AUTHOR

recv() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

getsockopt(2), read(2), select(2), send(2), socket(2), af_ccitt(7F), inet(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P), xopen_networking(7).

STANDARDS CONFORMANCE

recv(): XPG4

NAME

close - close a file descriptor

SYNOPSIS

```
#include <unistd.h>
        int close(int fildes);
```

DESCRIPTION

close() closes the file descriptor indicated by *fildes*. *fildes* is a file descriptor obtained from a creat(), open(), dup(), fcntl(), or pipe() system call. All associated file segments which have been locked by this process with the lockf() function are released (i.e., unlocked).

RETURN VALUE

Upon successful completion, close() returns a value of 0; otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

close() fails if the any of following conditions are encountered:

[EBADF] *fildes* is not a valid open file descriptor.

[EINTR] An attempt to close a slow device or connection was interrupted by a signal. The file descriptor still points to an open device or connection.

[ENOSPC] Not enough space on the file system. This error can occur when closing a file on an NFS file system. [When a write() system call is executed on a local file system and if a new buffer needs to be allocated to hold the data, the buffer is mapped onto the disk at that time. A full disk is detected at this time and write() returns an error. When the write() system call is executed on an NFS file system, the new buffer is allocated without communicating with the NFS server to see if there is space for the buffer (to improve NFS performance). It is only when the buffer is written to the server (at file close or the buffer is full) that the disk-full condition is detected.]

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), lockf(2), open(2), pipe(2).

STANDARDS CONFORMANCE

close(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

shutdown - shut down a socket

SYNOPSIS

```
#include <sys/socket.h>
        int shutdown(int s, int how);
```

DESCRIPTION

The shutdown() system call is used to shut down a socket. In the case of a full-duplex connection, shutdown() can be used to either partially or fully shut down the socket, depending upon the value of *how*.

How	Interpretation
SHUT_RD or 0	Further receives are disallowed
SHUT_WR or 1	Further sends are disallowed
SHUT_RDWR or 2	Further sends and receives are disallowed

The *s* parameter is a socket descriptor for the socket to be shut down.

Once the socket has been shut down for receives, all further recv() calls return an end-of-file condition. A socket that has been shut down for sending causes further send() calls to return an EPIPE error and send the SIGPIPE signal. After a socket has been fully shut down, operations other than recv() and send() return appropriate errors, and the only other thing that can be done to the socket is a close().

Multiple shutdowns on a connected socket and shutdowns on a socket that is not connected may not return errors.

A shutdown() on a connectionless socket, such as SOCK_DGRAM , only marks the socket as unable to do further send() or recv() calls, depending upon the value of *how*. Once this type of socket has been disabled for both sending and receiving data, it becomes fully shut down. For SOCK_STREAM sockets, if *how* is 1 or 2, the connection begins to be closed gracefully in addition to the normal actions. However, the shutdown() call does not wait for the completion of the graceful disconnection. The disconnection is complete when both sides of the connection have done a shutdown() with *how* equal to 1 or 2. Once the connection has been completely terminated, the socket becomes fully shut down. The SO_LINGER option (see *socket*(2)) does not have any meaning for the shutdown() call, but does for the close() call. For more information on how the close() call interacts with sockets, see *socket*(2).

If a shutdown() is performed on a SOCK_STREAM socket that has a listen() pending on it, that socket becomes fully shut down when *how* = 1.

AF_CCITT only:

The *how* parameter behaves differently if the socket is of the the AF_CCITT address family. If *how* is set to 0 the specified socket can no longer receive data. The SVC is not cleared and remains intact. However, if data is subsequently received on the SVC, it is cleared. The connection is not completely down until either side executes a close() or shutdown() with *how* set to 1 or 2.

If *how* is set to 1 or 2, the SVC can no longer send or receive data and the SVC is cleared. The socket's resources are maintained so that data arriving prior to the shutdown() call can still be read.

RETURN VALUE

Upon successful completion, shutdown() returns 0; otherwise it returns -1 and errno is set to indicate the error.

ERRORS

shutdown() fails if any of the following conditions are encountered:

[EBADF] *s* is not a valid file descriptor.

[ENOTSOCK] *s* is a valid file descriptor, but it is not a socket.

[EINVAL] HP-UX BSD Sockets only. The specified socket is not connected.

[ENOTCONN] _XOPEN_SOURCE_EXTENDED only. The specified socket is not connected.

[EINVAL] _XOPEN_SOURCE_EXTENDED only. The *how* argument is invalid.

AUTHOR

shutdown() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

close(2), connect(2), socket(2), xopen_networking(7).

STANDARDS CONFORMANCE

shutdown(): XPG4

NAME

fcntl - file control

SYNOPSIS

#include <fcntl.h>

int fcntl(int fildes, int cmd, ... /* arg */);

Remarks The ANSI C ", ... " construct denotes a variable length argument list whose optional [or required] members are given in the associated comment (/* */).

DESCRIPTION

fcntl() provides for control over open files. *fildes* is an open file descriptor.

The following are possible values for the *cmd* argument:

F_DUPFD Return a new file descriptor having the following characteristics:

- Lowest numbered available file descriptor greater than or equal to *arg*.val.
- Same open file (or pipe) as the original file.
- Same file pointer as the original file (that is, both file descriptors share one file pointer).
- Same access mode (read, write or read/write).
- Same file status flags (that is, both file descriptors share the same file status flags).
- The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 the file will remain open across *exec*(2), otherwise the file will be closed upon execution of *exec*(2).

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg*.val (see F_GETFD).

F_GETFL Get file status flags and access modes; see *fcntl*(5).

F_SETFL Set file status flags to *arg.val*. Only certain flags can be set; see *fcntl*(5). It is not possible to set both O_NDELAY and O_NONBLOCK.

F_GETLK Get the first lock that blocks the lock described by the variable of type struct flock pointed to by *arg*. The information retrieved overwrites the information passed to fcntl() in the flock structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged, except that the lock type is set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the variable of type struct flock pointed to by *arg.lockdes* (see *fcntl*(5)). The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as to remove either type of lock (F_UNLCK). If a read or write lock cannot be set, fcntl() returns immediately with an error value of -1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

F_GETOWN — If *fildes* refers to a socket, fcntl() returns the process or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values indicate a process ID; negative values, other than -1, indicate a process group ID.

F_SETOWN — If *fildes* refers to a socket, fcntl() sets the process or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, arg, taken as type int. Positive values indicate a process ID; negative values, other than -1, indicate a process group ID.

F_GETLK64 — Same as F_GETLK, except *arg* is a pointer to struct flock64 instead of struct flock.

F_SETLK64 — Same as F_SETLK, except *arg* is a pointer to struct flock64 instead of struct flock.

F_SETLKW64 — Same as F_SETLKW, except *arg* is a pointer to struct flock64 instead of struct flock.

Turning the O_LARGEFILE flag on and off can be done with F_SETFL.

A read lock prevents any other process from write-locking the protected area. More than one read lock can exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from read-locking or write-locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure flock describes the type (l_type), starting offset (l_whence), relative offset (l_start), size (l_len), and process ID (l_pid) of the segment of the file to be affected. The process ID field is only used with the F_GETLK *cmd* to return the value of a block in lock. Locks can start and extend beyond the current end of a file, but cannot be negative relative to the beginning of the file. A lock can be set to always extend to the end of file by setting l_len to zero (0). If such a lock also has l_start set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When enforcement-mode file and record locking is activated on a file (see *chmod*(2)), future read() and write() system calls on the file are affected by the record locks in effect.

NETWORKING FEATURES

NFS The advisory record-locking capabilities of *fcntl*(*2*) are implemented throughout the network by the ``network lock daemon'' (see *lockd*(*1M*)). If the file server crashes and is rebooted, the lock daemon attempts to recover all locks associated with the crashed server. If a lock cannot be reclaimed, the process that held the lock is issued a SIGLOST signal.

Record locking, as implemented for NFS files, is only advisory.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD A new file descriptor.

F_GETFD Value of close-on-exec flag (only the low-order bit is defined).

F_SETFD Value other than -1.

F_GETFL Value of file status flags and access modes.

F_SETFL Value other than -1.

F_GETLK Value other than -1.

F_SETLK Value other than -1.

F_SETLKW Value other than -1.

F_GETOWN Value of process or process group ID specified to receive SIGURG signals when out-of-band data is available.

F_SETOWN Value other than -1.

F_GETLK64 Value other than -1.

F_SETLK64 Value other than -1.

F_SETLKW64 Value other than -1.

Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

fcntl() fails if any of the following conditions occur:

[EBADF] *fildes* is not a valid open file descriptor, or was not opened for reading when setting a read lock or for writing when setting a write lock.

[EMFILE] *cmd* is F_DUPFD and the maximum number of file descriptors is currently open.

[EMFILE] *cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and no more file-locking headers are available (too many files have segments locked).

[EMFILE] *cmd* is F_DUPFD and *arg.val* is greater than or equal to the maximum number of file descriptors.

[EMFILE] *cmd* is F_DUPFD and *arg.val* is negative.

[EINVAL] *cmd* is F_GETLK, F_SETLK, or F_SETLKW, and *arg.lockdes* or the data it points to is not valid, or *fildes* refers to a file that does not support locking.

[EINVAL] *cmd* is not a valid command.

[EINVAL] *cmd* is F_SETFL and both O_NONBLOCK and O_NDELAY are specified.

[EINTR] *cmd* is F_SETLKW and the call was interrupted by a signal.

[EACCES] *cmd* is F_SETLK, the type of lock (l_type) is a read lock (F_RDLCK) or write lock (F_WRLCK) and the segment of a file to be locked is already write-locked by another process, or the type is a write lock (F_WRLCK) and the segment of a file to be locked is already read- or write-locked by another process.

[ENOLCK] *cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and no more file-locking headers are available (too many files have segments locked), or no more record locks are available (too many file segments locked).

[ENOLCK] *cmd* is F_SETLK or F_SETLKW, the type of lock (l_type) is a read lock (F_RDLCK) or write lock (F_WRLCK) and the file is an NFS file with access bits set for enforcement mode.

[ENOLCK *cmd* is F_GETLK, F_SETLK, or F_SETLKW, the file is an NFS file, and a system error occurred on the remote node.

[EOVERFLOW] *cmd* is F_GETLK and the blocking lock's starting offset or length would not fit in the caller's structure.

[EDEADLK] *cmd* is F_SETLKW, when the lock is blocked by a lock from another process and sleeping (waiting) for that lock to become free. This causes a deadlock situation.

[EAGAIN] *cmd* is F_SETLK or F_SETLKW, and the file is mapped in to virtual memory via the mmap() system call (see *mmap*(2)).

[EFAULT] *cmd* is either F_GETLK, F_SETLK, or F_SETLKW, and *arg* points to an illegal address.

[ENOTSOCK] *cmd* is F_GETOWN or F_SETOWN, and *fildes* does not refer to a socket.

AUTHOR

fcntl() was developed by HP, AT&T and the University of California, Berkeley.

APPLICATION USAGE

Because in the future the external variable errno will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value, for example:

```
flk->l_type = F_RDLCK;
    if (fcntl(fd, F_SETLK, flk) == -1)
        if ((errno == EACCES) || (errno == EAGAIN))
              /*
                 * section locked by another process,
                 * check for either EAGAIN or EACCES
                 * due to different implementations
           */
             else if ...
                 /*
                 * check for other errors
                 */
```

SEE ALSO

lockd(1M), statd(1M), chmod(2), close(2), exec(2), lockf(2), lockf64(), open(2), read(2), write(2), fcntl(5).

FUTURE DIRECTIONS

The error condition which currently sets errno to EACCES will instead set errno to EAGAIN (see also APPLICATION USAGE above).

STANDARDS CONFORMANCE

fcntl(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

gethostent(), gethostent_r(), gethostbyaddr(), gethostbyaddr_r(), gethostbyname(), gethostbyname_r(), sethostent(), sethostent_r(), endhostent(), endhostent_r() - get network host entry

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
extern int h_errno;
struct hostent *gethostent(void);
int gethostent_r(struct hostent *result,struct hostent_data *buffer);
struct hostent *gethostbyname(const char *name);
int gethostbyname_r(const char *name, struct hostent *result, struct hostent_data *buffer);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
_XOPEN_SOURCE_EXTENDED only
struct hostent *gethostbyaddr(const void *addr, size_t len int type);
int gethostbyaddr_r(const char *addr, int len, int type, struct hostent *result, struct hostent_data *buffer);
int sethostent(int stayopen);
int sethostent_r(int stayopen, struct hostent_data *buffer);
int endhostent(void);
int endhostent_r(struct hostent_data *buffer);
_XOPEN_SOURCE_EXTENDED only void sethostent(int stayopen); void endhostent(void);
```

DESCRIPTION

The gethostent(), gethostbyname(), and gethostbyaddr() functions each return a pointer to a structure of type hostent, defined as follows in <netdb.h>:

```
struct hostent {
    char     *h_name;
    char     **h_aliases;
    int      h_addrtype;
    int      h_length;
    char     **h_addr_list;
};
#define h_addr   h_addr_list[0]
```

The members of this structure are:

h_name	The official name of the host.
h_aliases	A null-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A null-terminated array of network addresses for the host.
h_addr	The first address in h_addr_list; this is for compatibility with previous HP-UX implementations where a struct hostent contains only one network address per host.

Reentrant Interfaces

gethostent_r(), gethostbyname_r(), and gethostbyaddr_r() expect to be passed the address of a struct hostent and will store the result at the supplied location. An additional parameter, a pointer to a struct hostent_data, must also be supplied. This structure is used to store data, to which fields in the struct hostent will point, as well as state information such as open file descriptors. The struct hostent_data is defined in the header file <netdb.h>.

sethostent_r() and endhostent_r() are to be used only in conjunction with gethostent_r() and take the same pointer to a struct hostent_data as a parameter. If the Network Information Service is being used, sethostent_r() initializes an internal database key. If the /etc/hosts file is being used, sethostent_r() opens or rewinds the file. If the named name server (see *named*(1M)) is being used, then sethostent_r() has no effect. endhostent_r() should always be called to ensure that files are closed and internally allocated data structures are released.

The *stayopen* parameter to sethostent_r() currently has no effect. However, sethostent() can still be used to keep the /etc/hosts file open, or to use connected stream sockets to the name server, when making calls to gethostbyaddr_r() and gethostbyname_r().

The *hostf* field in the struct hostent_data must be initialized to NULL before it is passed to either gethostent_r() or sethostent_r() for the first time. The *current* field in the struct hostent_data must be initialized to NULL before it is passed to gethostbyname_r() or gethostbyaddr_r() for the first time. Thereafter, these fields should not be modified in any way. These are the only hostent_data fields that should ever be explicitly accessed.

Name Service Switch-Based Operation

These host entry library routines internally call the name service switch to access the "hosts" database lookup policy configured in the /etc/nsswitch.conf file (see *switch*(*4*)). The lookup policy defines the order and the criteria of the supported name services used to resolve host names and Internet addresses. The operations of the three name services: Domain Name Server, NIS, and nonserver mode (e.g., files) are listed below.

Domain Name Server Operation

If the local system is configured to use the named name server (see *named*(*1M*) and *resolver*(*4*)) for name or address resolution, then the function:

gethostent() Always returns a NULL pointer.

sethostent() Requests the use of a connected stream socket for queries to the name server if the *stayopen* flag is non-zero. The connection is retained after each call to gethostbyname() or gethostbyaddr().

endhostent() Closes the stream socket connection.

gethostbyname()
gethostbyaddr() Each retrieves host information from the name server. Names are matched without respect to uppercase or lowercase. For example, berkeley.edu, Berkeley.EDU, and BERKELEY.EDU all match the entry for berkeley.edu.

NIS Server Operation

If ypserv, the server for the Network Information Service (see *ypserv*(1M)), is used for name or address resolution, then the function:

gethostent() Returns the next entry in the NIS database.

sethostent() Initializes an internal key for the NIS database. If the *stayopen* flag is non-zero, the internal key is not cleared after calls to endhostent().

endhostent() Clears the internal NIS database key.

gethostbyname()

gethostbyaddr() Each retrieves host information from the NIS database. Names are matched without respect to uppercase or lowercase. For example, berkeley.edu, Berkeley.EDU, and BERKELEY.EDU all match the entry for berkeley.edu.

Nonserver Operation

If the /etc/hosts file is used for name or address resolution, then the function:

gethostent() Reads the next line of /etc/hosts, opening the file if necessary.

sethostent() opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base is not closed after each call to gethostent() (either directly or indirectly through one of the other gethost calls).

endhostent() Closes the file.

gethostbyname() Sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matching its *name* parameter is found, or until EOF is encountered. Names are matched without respect to uppercase or lowercase, as described above in the name server case.

gethostbyaddr() Sequentially searches from the beginning of the file until an Internet address matching its *addr* parameter is found, or until EOF is encountered.

Arguments

Currently, only the Internet address format is understood. In calls to gethostbyaddr(), the parameter *addr* must be a pointer to an *in_addr* structure, an Internet address in network order (see *byteorder*(3N)) and the header file <netinet/in.h>). The parameter *len* must be the number of bytes in an Internet address; that is, sizeof (struct in_addr). The parameter *type* must be the constant AF_INET.

RETURN VALUE

If successful, gethostbyname(), gethostbyaddr(), and gethostent() return a pointer to the requested hostent structure.

gethostbyname() and gethostbyaddr() return NULL if their *host* or *addr* parameters, respectively, cannot be found in the database. If /etc/hosts is being used, they also return NULL if they are unable to open /etc/hosts.

gethostbyaddr() also returns NULL if either its *addr* or *len* parameter is invalid.

gethostent() always returns NULL if the name server is being used.

For the reentrant (_r) versions of these routines, -1 is returned if the operation is unsuccessful or, in the case of gethostent_r(), if the end of the hosts list has been reached. 0 is returned otherwise.

ERRORS

If the name server is being used and gethostbyname() or gethostbyaddr() returns a NULL pointer, the external integer h_errno contains one of the following values:

HOST_NOT_FOUND No such host is known.

TRY_AGAIN This is usually a temporary error. The local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY This is a non-recoverable error.

NO_ADDRESS The requested name is valid but does not have an IP address; this is not a temporary error. This means another type of request to the name server will result in an answer.

If the name server is not being used, the value of h_errno may not be meaningful.

EXAMPLES

The following code excerpt counts the number of host entries:

```
int count = 0;
struct hostent htbuf;
struct hostent_data hdbuf;
hdbuf.hostf = NULL;
(void) sethostent_r(0, &hdbuf);
while (gethostent_r(&htbuf, &hdbuf) != -1)
    count++;
(void) endhostent_r(&hdbuf);
```

WARNINGS

For the non-reentrant versions of these routines, all information is contained in a static area so it must be copied if it is to be saved.

gethostent(), gethostbyaddr(), gethostbyname(), sethostent(), and endhostent() are unsafe in multi-thread applications. gethostent_r(), gethostbyaddr_r(), gethostbyname_r(), sethostent_r(), and endhostent_r() are MT-Safe and should be used instead.

AUTHOR

gethostent() was developed by the University of California, Berkeley.

FILES

/etc/hosts

SEE ALSO

named(1M), ypserv(1M), resolver(3N), ypclnt(3C), hosts(4), switch(4), ypfiles(4).

STANDARDS CONFORMANCE

gethostent(): XPG4

NAME

gethostent(), gethostent_r(), gethostbyaddr(), gethostbyaddr_r(), gethostbyname(), gethostbyname_r(), sethostent(), sethostent_r(), endhostent(), endhostent_r() - get network host entry

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
extern int h_errno;
struct hostent *gethostent(void);
int gethostent_r(struct hostent *result, struct hostent_data *buffer);
struct hostent *gethostbyname(const char *name);
int gethostbyname_r(const char *name, struct hostent *result, struct hostent_data *buffer);
struct hostent *gethostbyaddr(const char *addr, int len,   int type);
_XOPEN_SOURCE_EXTENDED only struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
int gethostbyaddr_r(const char *addr, int len, int type, struct hostent *result, struct hostent_data *buffer);
int sethostent(int stayopen);
int sethostent_r(int stayopen, struct hostent_data *buffer);
int endhostent(void);
int endhostent_r(struct hostent_data *buffer);
_XOPEN_SOURCE_EXTENDED only void sethostent(int stayopen); void endhostent(void);
```

DESCRIPTION

The gethostent(), gethostbyname(), and gethostbyaddr() functions each return a pointer to a structure of type hostent, defined as follows in <netdb.h>:

```
struct hostent {
    char     *h_name;
    char     **h_aliases;
    int      h_addrtype;
    int      h_length;
    char     **h_addr_list;
};
#define h_addr   h_addr_list[0]
```

The members of this structure are:

h_name	The official name of the host.
h_aliases	A null-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A null-terminated array of network addresses for the host.
h_addr	The first address in h_addr_list; this is for compatibility with previous HP-UX implementations where a struct hostent contains only one network address per host.

Reentrant Interfaces

gethostent_r(), gethostbyname_r(), and gethostbyaddr_r() expect to be passed the address of a struct hostent and will store the result at the supplied location. An additional parameter, a pointer to a struct hostent_data, must also be supplied. This structure is used to store data, to which fields in the struct hostent will point, as well as state information such as open file descriptors. The struct hostent_data is defined in the header file <netdb.h>.

sethostent_r() and endhostent_r() are to be used only in conjunction with gethostent_r() and take the same pointer to a struct hostent_data as a parameter. If the Network Information Service is being used, sethostent_r() initializes an internal database key. If the /etc/hosts file is being used, sethostent_r() opens or rewinds the file. If the named name server (see *named*(1M)) is being used, then sethostent_r() has no effect. endhostent_r() should always be called to ensure that files are closed and internally allocated data structures are released.

The *stayopen* parameter to sethostent_r() currently has no effect. However, sethostent() can still be used to keep the /etc/hosts file open, or to use connected stream sockets to the name server, when making calls to gethostbyaddr_r() and gethostbyname_r().

The *hostf* field in the struct hostent_data must be initialized to NULL before it is passed to either gethostent_r() or sethostent_r() for the first time. The *current* field in the struct hostent_data must be initialized to NULL before it is passed to gethostbyname_r() or gethostbyaddr_r() for the first time. Thereafter, these fields should not be modified in any way. These are the only hostent_data fields that should ever be explicitly accessed.

Name Service Switch-Based Operation These host entry library routines internally call the name service switch to access the "hosts" database lookup policy configured in the /etc/nsswitch.conf file (see *switch*(*4*)). The lookup policy defines the order and the criteria of the supported name services used to resolve host names and Internet addresses. The operations of the three name services: Domain Name Server, NIS, and nonserver mode (e.g., files) are listed below.

Domain Name Server Operation If the local system is configured to use the named name server (see *named*(*1M*) and *resolver*(*4*)) for name or address resolution, then the function:

gethostent() Always returns a NULL pointer.

sethostent(), Requests the use of a connected stream socket for queries to the name server I the *stayopen* flag is non-zero. The connection is retained after each call to gethostbyname() or gethostbyaddr().

endhostent() Closes the stream socket connection.

gethostbyname()

gethostbyaddr() Each retrieves host information from the name server. Names are matched without respect to uppercase or lowercase. For example, berkeley.edu, Berkeley.EDU, and BERKELEY.EDU all match the entry for berkeley.edu.

NIS Server Operation

If ypserv, the server for the Network Information Service (see *ypserv*(1M)), is used for name or address resolution, then the function:

gethostent() Returns the next entry in the NIS database.

sethostent() Initializes an internal key for the NIS database. If the *stayopen* flag is non- zero, the internal key is not cleared after calls to endhostent().

endhostent() Clears the internal NIS database key.

gethostbyname()

gethostbyaddr() Each retrieves host information from the NIS database. Names are matched without respect to uppercase or lowercase. For example, berkeley.edu, Berkeley.EDU, and BERKELEY.EDU all match the entry for berkeley.edu.

Nonserver Operation

If the /etc/hosts file is used for name or address resolution, then the function:

gethostent() Reads the next line of /etc/hosts, opening the file if necessary.

sethostent() opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base is not closed after each call to gethostent() (either directly or indirectly through one of the other gethost calls).

endhostent() Closes the file.

gethostbyname() Sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matching its *name* parameter is found, or until EOF is encountered. Names are matched without respect to uppercase or lowercase, as described above in the name server case.

gethostbyaddr() Sequentially searches from the beginning of the file until an Internet address matching its *addr* parameter is found, or until EOF is encountered.

Arguments

Currently, only the Internet address format is understood. In calls to gethostbyaddr(), the parameter *addr* must be a pointer to an *in_addr* structure, an Internet address in network order (see *byteorder*(3N)) and the header file <netinet/in.h>). The parameter *len* must be the number of bytes in an Internet address; that is, sizeof (struct in_addr). The parameter *type* must be the constant AF_INET.

RETURN VALUE

If successful, gethostbyname(), gethostbyaddr(), and gethostent() return a pointer to the requested hostent structure. gethostbyname() and gethostbyaddr() return NULL if their *host* or *addr* parameters, respectively, cannot be found in the database. If /etc/hosts is being used, they also return NULL if they are unable to open /etc/hosts.

gethostbyaddr() also returns NULL if either its *addr* or *len* parameter is invalid.

gethostent() always returns NULL if the name server is being used.

For the reentrant (_r) versions of these routines, -1 is returned if the operation is unsuccessful or, in the case of gethostent_r(), if the end of the hosts list has been reached. 0 is returned otherwise.

ERRORS

If the name server is being used and gethostbyname() or gethostbyaddr() returns a NULL pointer, the external integer h_errno contains one of the following values:

HOST_NOT_FOUND No such host is known.

TRY_AGAIN This is usually a temporary error. The local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY This is a non-recoverable error.

NO_ADDRESS The requested name is valid but does not have an IP address; this is not a temporary error. This means another type of request to the name server will result in an answer.

If the name server is not being used, the value of h_errno may not be meaningful.

EXAMPLES

The following code excerpt counts the number of host entries:

```
int count = 0;
struct hostent htbuf;
struct hostent_data hdbuf;
hdbuf.hostf = NULL;
(void) sethostent_r(0, &hdbuf);
while (gethostent_r(&htbuf, &hdbuf) != -1)
        count++;
(void) endhostent_r(&hdbuf);
```

WARNINGS

For the non-reentrant versions of these routines, all information is contained in a static area so it must be copied if it is to be saved.

gethostent(), gethostbyaddr(), gethostbyname(), sethostent(), and endhostent() are unsafe in multi-thread applications. gethostent_r(), gethostbyaddr_r(), gethostbyname_r(), sethostent_r(), and endhostent_r() are MT-Safe and should be used instead.

AUTHOR

gethostent() was developed by the University of California, Berkeley.

FILES

/etc/hosts

SEE ALSO

named(1M), ypserv(1M), resolver(3N), ypclnt(3C), hosts(4), switch(4), ypfiles(4).

STANDARDS CONFORMANCE

gethostent(): XPG4

NAME

getprotoent(), getprotoent_r(), getprotobynumber(), getprotobynumber_r(), getprotobyname(), getprotobyname_r(), setprotoent(), setprotoent_r(), endprotoent(), endprotoent_r() – get protocol entry

SYNOPSIS

```
#include <netdb.h>
struct protoent *getprotoent(void);
int getprotoent_r(struct protoent *result,    struct protoent_data *buffer);
struct protoent *getprotobyname(const char *name);
int getprotobyname_r(const char *name,    struct protoent *result,    struct protoent_data *buffer);
struct protoent *getprotobynumber(int proto);
int getprotobynumber_r(int proto,    struct protoent *result,    struct protoent_data *buffer);
int setprotoent(int stayopen);
int setprotoent_r(int stayopen, struct protoent_data *buffer);
int endprotoent(void);
int endprotoent_r(struct protoent_data *buffer);
_XOPEN_SOURCE_EXTENDED only
void setprotoent(int stayopen);    void endprotoent(void);
```

DESCRIPTION

The getprotoent(), getprotobyname(), and getprotobynumber() functions each return a pointer to a structure of type *protoent* containing the broken-out fields of a line in the network protocol data base, /etc/protocols.

The members of this structure are:

p_name	The official name of the protocol.
p_aliases	A null-terminated list of alternate names for the protocol.
p_proto	The protocol number.

Functions behave as follows:

getprotoent()	Reads the next line of the file, opening the file if necessary.
setprotoent()	Opens and rewinds the file. If the *stayopen* flag is non-zero, the protocol data base is not closed after each call to getprotoent() (either directly or indirectly through one of the other getproto* calls).
endprotoent()	Closes the file.

getprotobyname()
getprotobynumber()
	Each sequentially searches from the beginning of the file until a matching protocol name (among either the official names or the aliases) or protocol number is found, or until EOF is encountered.

If the system is running the Network Information Service (NIS) services, getprotobyname() and getprotobynumber() get the protocol information from the NIS server (see *ypserv*(1M) and *ypfiles*(4)).

Reentrant Interfaces

getprotoent_r(), getprotobyname_r(), and getprotobynumber_r() expec to be passed the address of a struct protoent and will store the result at the supplied location. An additional parameter, a pointer to a struct protoent_data, must also be supplied. This structure is used to store data, to which fields in the struct protoent will point, as well as state information such as open file descriptors. The struct protoent_data is defined in the file <netdb.h>.

setprotoent_r() and endprotoent_r() are to be used only in conjunction with getprotoent_r() and take the same pointer to a struct protoent_data as a parameter. If the Network Information Service is being used, setprotoent_r() initializes an internal database key. If the /etc/protocols file is being used, setprotoent_r() opens or rewinds the file. endprotoent_r() should always be called to ensure that files are closed and internally allocated data structures are released.

The *stayopen* parameter to setprotoent_r() currently has no effect. However, setprotoent() can still be used to keep the /etc/protocols file open when making calls to getprotobyname_r() and getprotobynumber_r().

The *proto_fp* field in the struct protoent_data must be initialized to NULL before it is passed to either getprotoent_r() or setprotoent_r() for the first time. Thereafter it should not be modified in any way. This is the only protoent_data field that should ever be explicitly accessed.

Name Service Switch-Based Operation

The library routines, getprotobyname(), getprotobynumber(), getprotoent(), and their reentrant counterparts, internally call the name service switch to access the "protocols" database lookup policy configured in the /etc/nsswitch.conf file (see *switch*(*4*)). The lookup policy defines the order and the criteria of the supported name services used to resolve protocol names and numbers.

RETURN VALUE

getprotoent(), getprotobyname(), and getprotobynumber() return a null pointer (0) on EOF or when they are unable to open /etc/protocols.

For the reentrant (_r) versions of these routines, -1 will be returned if the operation is unsuccessful or, in the case of getprotoent_r(), if the end of the protocols list has been reached. 0 is returned otherwise.

EXAMPLES

The following code excerpt counts the number of protocols entries:

```
int count = 0;
struct protoent protobuf;
struct protoent_data pdbuf;

pdbuf.proto_fp = NULL;
(void) setprotoent_r(0, &pdbuf);
while (getprotoent_r(&protobuf, &pdbuf) != -1)
    count++;
(void) endprotoent_r(&pdbuf);
```

WARNINGS

In the non-reentrant versions of these routines, all information is contained in a static area so it must be copied if it is to be saved.

getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(), and endprotoent() are unsafe in multi-thread applications. getprotoent_r(), getprotobynumber_r(), getprotobyname_r(), setprotoent_r(), and endprotoent_r() are MT-Safe and should be used instead.

AUTHOR

getprotoent() was developed by the University of California, Berkeley.

FILES

/etc/protocols

SEE ALSO

ypserv(1M), protocols(4), ypfiles(4).

STANDARDS CONFORMANCE

getprotoent(): XPG4

NAME

getpeername - get address of connected peer

SYNOPSIS

#include <sys/socket.h>

AF_CCITT only:

#include <x25/x25addrstr.h>

int getpeername(int s, void *addr, int *addrlen);

_XOPEN_SOURCE_EXTENDED only int getpeername(int s, struct sockaddr *addr, size_t *addrlen);

DESCRIPTION

getpeername() returns the address of the peer socket connected to the socket indicated by *s*, where *s* is a socket descriptor. *addr* points to a socket address structure in which this address is returned. *Addrlen* points to an object of type int, which should be initialized to indicate the size of the address structure. On return, it contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the peer, only the first *addrlen* bytes of the address are returned.

AF_CCITT only:

The *addr* struct contains the X.25 addressing information of the *remote* peer socket connected to socket *s*. However, the x25ifname[] field of the *addr* struct contains the name of the *local* X.25 interface through which the call arrived.

RETURN VALUE

Upon successful completion, getpeername() returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

getpeername() fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid file descriptor.
[ENOTSOCK]	*s* is a valid file descriptor, but it is not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	No buffer space is available to perform theoperation.
[EFAULT]	*addr* or *addrlen* are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOTSUPP]	Operation not supported for AF_UNIX sockets.

AUTHOR

getpeername() was developed by the University of California, Berkeley.

FUTURE DIRECTION

The default behavior in this release is still the classic HP-UX BSD Sockets, however it will be changed to X/Open Sockets in some future release. At that time, any HP-UX BSD Sockets behavior which is incompatible with X/Open Sockets may be obsoleted. HP customers are advised to migrate their applications to conform to X/Open specification(see *xopen_networking*(*7*)).

SEE ALSO

bind(2), socket(2), getsockname(2), inet(7F), af_ccitt(7F), xopen_networking(7).

NAME

perror(), strerror(), strerror_r(), errno, sys_errlist, sys_nerr -system error messages

SYNOPSIS

```
#include <errno.h>
void perror(const char *s);
char *strerror(int errnum);
int strerror_r(int errnum, char *buffer, int buflen);
extern int errno;
extern char *sys_errlist[];
extern int sys_nerr;
```

DESCRIPTION

perror() writes a language-dependent message to the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, followed by a colon, a blank, the message, and a new-line. To be most useful, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable errno, which is set when errors occur but not cleared when non-erroneous calls are made. The contents of the message is identical to those returned by the strerror() function with errno as the argument. If given a NULL string, the perror() function prints only the message and a new-line.

To simplify variant formatting of messages, the strerror() function and the sys_errlist array of message strings are provided. The strerror() function maps the error number in *errnum* to a language-dependent error message string and returns a pointer to the string. The message string is returned without a new-line. errno can be used as an index into sys_errlist to get an untranslated message string without the new-line. sys_nerr is the largest message number provided for in the table; it should be checked because new error codes might be added to the system before they are added to the table. strerror() must be used to retrieve messages when translations are desired.

strerror_r() is identical to strerror(), except that the result string is passed back in the supplied buffer. A buffer length of 80 is recommended. If an error is detected or the buffer is of insufficient length, -1 is returned. If the operation is successful, 0 is returned.

EXTERNAL INFLUENCES

Environment Variables

The language of the message returned by strerror() and printed by perror() is specified by the LANG environment variable. If the language-dependent message is not available, or if LANG is not set or is set to the empty string, the default version of the message associated with the "C" language (see *lang*(5)) is used.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

perror() returns no value.

If the *errnum* message number is valid, strerror() returns a pointer to a language-dependent message string. The array pointed to should not be modified by the program, and might be overwritten by a subsequent call to the function. If a valid *errnum* message number does not have a corresponding language-dependent message, strerror() uses *errnum* as an index into sys_errlist to get the message string. If the *errnum* message number is invalid, strerror() returns a pointer to a NULL string.

WARNINGS

The return value for strerror() points to static data whose content is overwritten by each call. strerror() is unsafe for multi-thread applications. strerror_r() is MT-Safe and should be used instead.

SEE ALSO

errno(2), lang(5), environ(5).

STANDARDS CONFORMANCE

perror(): AES, SVID2, SVID3, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strerror(): AES, SVID3, XPG3, XPG4, ANSI C

sys_errlist(): SVID2, SVID3, XPG2

sys_nerr(): SVID2, SVID3, XPG2

NAME

select - synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/time.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set   *errorfds, struct timeval *timeout);
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

DESCRIPTION

The select() function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, select() blocks, up to the specified timeout interval, until the specified condition is true for at least one of the specified file descriptors.

The select() function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs and pipes. The behaviour of select() on file descriptors that refer to other types of file is unspecified.

The *nfds* argument specifies the range of file descriptors to be tested. The select() function tests file descriptors in the range of 0 to *nfds* -1.

If the *readfs* argument is not a null pointer, it points to an object of type *fd_set* that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

If the *writefs* argument is not a null pointer, it points to an object of type *fd_set* that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.

If the *errorfds* argument is not a null pointer, it points to an object of type *fd_set* that *on* input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.

On successful completion, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively. For each file descriptor less than *nfds*, the corresponding bit will be set on successful completion if it was set on input and the associated condition is true for that file descriptor.

If the *timeout* argument is not a null pointer, it points to an object of type *struct timeval* that specifies a maximum interval to wait for the selection to complete. If the *timeout* argument points to an object of type *struct timeval* whose members are 0, select() does not block. If the *timeout* argument is a null pointer, select() blocks until an event causes one of the masks to be returned with a valid (non-zero) value. If the time limit expires before any event occurs that would cause one of the masks to be set to a non-zero value, select() completes successfully and returns 0.

Implementations may place limitations on the maximum timeout interval supported. On all implementations, the maximum timeout interval supported will be at least 31 days. If the timeout argument specifies a timeout interval greater than the implementation- dependent maximum value, the maximum value will be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval will be rounded up to the next supported value.

If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the timeout argument is not a null pointer, select() blocks for the time specified, or until interrupted by a signal. If the *readfs*, *writefs*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, select() blocks until interrupted by a signal.

File descriptors associated with regular files always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments are not modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfs*, *writefs*, and *errorfds* arguments have all bits set to 0.

File descriptor masks of type *fd_set* can be initialised and tested with FD_CLR(), FD_ISSET(), FD_SET(), and FD_ZERO(). It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behaviour is undefined.

FD_CLR(fd, &fdset) Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.

FD_ISSET(fd, &fdset) Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

FD_SET(fd, &fdset) Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.

FD_ZERO(&fdset) Initialises the file descriptor set *fdset* to have zero bits for all file descriptors. The behaviour of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to FD_SETSIZE.

RETURN VALUE

FD_CLR(), FD_SET(), and FD_ZERO() return no value. FD_ISSET() returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

On successful completion, select() returns the total number of bits set in the bit masks. Otherwise, -1 is returned, and errno is set to indicate the error.

ERRORS

Under the following conditions, select() fails and sets errno to:

[EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

[EINTR] The select() function was interrupted before any of the selected events occurred and before the *timeout* interval expired. If SA_RESTART has been set for the interrupting signal, it is implementation-dependent whether select() restarts or returns with EINTR.

[EINVAL] An invalid timeout interval was specified.

[EINVAL] The *nfds* argument is less than 0, or greater than or equal to

FD_SETSIZE.

[EINVAL] One of the specified file descriptors refers o a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

APPLICATION USAGE

The use of a timeout does not affect any pending timers set up by alarm(), ualarm(), or settimer().

On successful completion, the object pointed to by the *timeout* argument may be modified.

SEE ALSO

fcntl(), poll(), read(), write(), <sys/time.h>.

CHANGE HISTORY

First released in Issue 4, Version 2.

HP-UX EXTENSIONS

SYNOPSIS

```
#include <time.h>
int select(
      size_t nfds,
      int *readfds,
      int *writefds,
      int *exceptfds,
      const struct timeval *timeout
 );
```

DESCRIPTION

select() examines the files or devices associated with the file descriptors specified by the bit masks *readfds*, *writefds*, and *exceptfds*. The bits from 0 through *nfds*-1 are examined. File descriptor *f* is represented by the bit 1<<*f* in the masks. More formally, a file descriptor is represented by:

fds[(f / BITS_PER_INT)] & (1 << (f % BITS_PER_INT))

Ttys and sockets are ready for reading or writing, respectively, if a read() or write() would not block for one or more of the following reasons:

- input data is available.
- output data can be accepted.
- an error condition exists, such as a broken pipe, no carrier, or a lost connection.

Sockets select true on reads and/or exceptions if out-of-band data is available.

Pipes are ready for reading if there is any data in the pipe, or if there are no writers left for the pipe. Pipes are ready for writing if there is room for more data in the pipe AND there are one or more readers for the pipe, OR there are no readers left for the pipe. select() returns the same results for a pipe whether a file descriptor associated with the read-only end or the write-only end of the pipe is used, since both file descriptors refer to the same underlying pipe. So a select() of a read-only file descriptor that is associated with a pipe can return ready to write, even though that particular file descriptor cannot be written to.

ERRORS

[EFAULT] One or more of the pointers was invalid. The reliable detection of this error is implementation dependent.

EXAMPLES

The following call to select() checks if any of 4 terminals are ready for reading. select() times out after 5 seconds if no terminals are ready for reading. Note that the code for opening the terminals or reading from the terminals is not shown in this example. Also, note that this example must be modified if the calling process has more than 32 file descriptors open. Following this first example is an example of select with more than 32 file descriptors.

```
#define MASK(f)        (1 << (f))
#define NTTYS 4

int tty[NTTYS];
int ttymask[NTTYS];
int readmask = 0;
int readfds;
int nfound, i;
struct timeval timeout;

    /* First open each terminal for reading and put the
     * file descriptors into array tty[NTTYS].   The code
     * for opening the terminals is not shown here.
     */

for (i=0; i < NTTYS; i++) {
    ttymask[i] = MASK(tty[i]);
    readmask |= ttymask[i];
}

timeout.tv_sec   = 5;
timeout.tv_usec = 0;
readfds = readmask;

/* select on NTTYS+3 file descriptors if stdin, stdout
 * and stderr are also open
 */
if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
    perror ("select failed");
else if (nfound == 0)
    printf ("select timed out \n");
else for (i=0; i < NTTYS; i++)
    if (ttymask[i] & readfds)
        /* Read from tty[i].   The code for reading
         * is not shown here.
         */
    else printf ("tty[%d] is not ready for reading \n",i);
```

The following example is the same as the previous example, except that it works for more than 32 open files. Definitions for howmany, fd_set, and NFDBITS are in <sys/types.h>.

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

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

#define MASK(f)         (1 << (f))
#define NTTYS NOFILE - 3
#define NWORDS    howmany(FD_SETSIZE, NFDBITS)

int tty[NTTYS];
int ttymask[NTTYS];
struct fd_set readmask, readfds;
int nfound, i, j, k;
struct timeval timeout;

/* First open each terminal for reading and put the
        * file descriptors into array tty[NTTYS].    The code
       * for opening the terminals is not shown here.
       */

for (k=0; k < NWORDS; k++)
        readmask.fds_bits[k] = 0;

for (i=0, k=0; i < NTTYS && k < NWORDS; k++)
        for (j=0; j < NFDBITS && i < NTTYS; j++, i++) {
                ttymask[i] = MASK(tty[i]);
                readmask.fds_bits[k] |= ttymask[i];
          }

timeout.tv_sec    = 5;
timeout.tv_usec = 0;
for (k=0; k < NWORDS; k++)
readfds.fds_bits[k] = readmask.fds_bits[k];

/* select on NTTYS+3 file descriptors if stdin, stdout
       * and stderr are also open
       */
     if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
          perror ("select failed");
     else if (nfound == 0)
          printf ("select timed out \n");
     else for (i=0, k=0; i < NTTYS && k < NWORDS; k++)
          for (j=0; j < NFDBITS && i < NTTYS; j++, i++)
               if (ttymask[i] & readfds.fds_bits[k])
                 /* Read from tty[i].    The code for reading
                   * is not shown here.
                */
            else printf ("tty[%d] is not ready for reading \n",i);
```

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support sigvector(). sigvector() can affect the behavior described on this manpage.

The file descriptor masks are always modified on return, even if the call returns as the result of a timeout.

DEPENDENCIES

select() supports the following devices and file types:

- pipes
- fifo special files (named pipes)
- all serial devices
- All ITEs (internal terminal emulators) and HP-HIL input devices
- *hpib*(7) special files
- *lan*(7) special files
- *pty*(7) special files
- sockets

AUTHOR

select() was developed by HP and the University of California, Berkeley.

SEE ALSO

fcntl(2), read(2), write(2).