Networking API

APPENDIX

B

IN THIS APPENDIX

This appendix groups all the networking library and system calls.

Connecting to the Network

The Sockets API provides tools to help you create sockets and connect to other hosts. This section describes the API relevant to socket creation and connection.

`socket()`

`socket()` creates a bidirectional channel that typically connects with the network. You can use this channel with network-specific system calls or general file I/O.

Prototype

```
#include <resolv.h>
#include <sys/socket.h>
#include <sys/types.h>int socket(int domain, int type, int protocol);
```

Return Value

If successful, the call returns a valid socket descriptor. Otherwise, the result is less than zero. Check `errno` for more information about the error.

Parameters

`domain`	Selects the network protocol for socket (see Appendix A, "Data Tables").
`type`	Selects the network layer (see Appendix A).
`protocol`	Usually zero (see Appendix A).

Possible Errors

`EPROTONOSUPPORT`	The protocol `type` or the specified `protocol` is not supported within this domain.
`ENFILE`	Not enough kernel memory to allocate a new socket structure.
`EMFILE`	Process file table overflow.
`EACCES` and `ENOBUFS`	Permission to create a socket of the specified `type` or `protocol` is denied.
`ENOMEM`	Insufficient memory is available. The socket cannot be created until sufficient resources are freed.
`EINVAL`	Unknown `protocol` or protocol family not available.

Examples

```
/*** Create a TCP socket ***/
int sd;
sd = socket(PF_INET, SOCK_STREAM, 0);
```

```
/*** Create an ICMP socket **/
int sd;
sd = socket(PF_INET, SOCK_RAW, htons(IPPROTO_ICMP));
```

bind()

`bind()` defines a port or name to a socket. If you want a consistent port for an incoming connection, you have to bind your socket to a port. In most instances, the kernel automatically calls `bind()` for the socket if you do not call it explicitly. The kernel-generated port assignment can be different from execution to execution.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int bind(int sockfd, struct sockaddr* addr,
    int addrlen);
```

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in `errno`.

Parameters

`sockfd`	The socket descriptor to bind.
`Addr`	The port assignment or name.
`Addrlen`	The length of `addr` because `addr` can be different sizes.

Possible Errors

`EBADF`	`sockfd` is not a valid descriptor.
`EINVAL`	The socket is already bound to an address. This may change in the future; see `.../linux/unix/socket.c` for details.
`EACCES`	The address is protected, and the user is not the superuser.
`ENOTSOCK`	Argument is a descriptor for a file, not a socket.

Example

```
/*** Bind port #9999 to socket from any Internet Address ***/
int sockfd;
struct sockaddr_in addr;
sockfd = socket(PF_INET, SOCKET_STREAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(9999);  /* or whatever port you want */
/* to bind any network interface */
addr.sin_addr.s_addr = INADDR_ANY;
/* -or- to bind a specific interface, use this: */
/* inet_aton("128.1.1.1", &addr.sin_addr); */
if ( bind(sockfd, &addr, sizeof(addr)) != 0 )
    perror("bind");
```

listen()

listen() converts the socket into a listening socket. This option is available only to SOCK_STREAM protocols. The call creates a queue of incoming connections.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int listen(int sockfd, int queue_len);
```

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in errno.

Parameters

sockfd	A SOCK_STREAM socket that has been bound to a port
queue_len	The maximum number of pending connections

Possible Errors

EBADF	The argument sockfd is not a valid descriptor.
ENOTSOCK	The argument sockfd is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the listen() operation. If you supply any socket that is not SOCK_STREAM, you get this error.

Example

```
/*** Convert a socket to a listening socket with 10 slots ***/
int sockfd;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
/*---set up address with bind()---*/
listen(sockfd, 10);  /* create a 10-pending queue */
```

accept()

Wait for a connection. When a connection arrives, return a new socket descriptor (independent of sockfd) for that specific connection. This call is available only to SOCK_STREAM sockets.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int accept(int sockfd, struct sockaddr *addr, int *addr_len);
```

Return value

If >= 0	A new socket descriptor.
If < 0	Error; errno has details.

Parameters

`sockfd`	The bound and listening socket descriptor.
`addr`	If nonzero, the call places the address definition in this region. While it should match the `sockfd`'s family (`AF_INET`), do not assume it.
`addr_len`	You pass the length by reference so that the call can tell you exactly how much of `addr`'s data block it used. This means that you need to reset the value for each call.

Possible Errors

`EBADF`	The socket descriptor is invalid.
`ENOTSOCK`	The descriptor references a file, not a socket.
`EOPNOTSUPP`	The referenced socket is not of type `SOCK_STREAM`.
`EFAULT`	The *`addr`* parameter is not in a writeable part of the user address space.
`EAGAIN`	The socket is marked non-blocking and no connections are present to be accepted.
`EPERM`	Firewall rules forbid connection.
`ENOBUFS, ENOMEM`	Not enough free memory.

Examples

```
/*** Accept a connection, ignoring the origin ***/
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
/*---bind address to socket with bind()---*/
/*---convert it to a listening socket with listen()---*/
for (;;)
{   int client;
    client = accept(sockfd, 0, 0);
    /*---interact with client---*/
    close(client);
}
```

```
/*** Accept a connection, capturing the origin in a log ***/
int sockfd = socket(PF_INET, SOCK_STREAM, 0);
/*---bind address to socket with bind()---*/
/*---convert it to a listening socket with listen()---*/
for (;;)
{   struct sockaddr_in addr;
    int client, addr_len = addr;
    client = accept(sockfd, &addr, &addr_len);
    printf("Connected: %s:%d\n", inet_ntoa(addr.sin_addr),
        ntohs(addr.sin_port));
    /*---interact with client---*/
    close(client);
}
```

connect()

Connect to a peer or server. You can use this function for either SOCK_DGRAM or SOCK_STREAM protocols. For UDP, it merely remembers the port to which you connected. This allows you to use send() and recv(). For TCP (SOCK_STREAM), this call starts the three-way handshake for stream communications.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int connect(int sockfd, struct sockaddr *addr, int addr_len);
```

Return Value

Zero (0) if everything goes well. If an error occurs, you can find the cause in errno.

Parameters

sockfd	The freshly created socket. You could optionally call bind() before this to assign the local port. If you do not, the kernel assigns the next available port slot.
addr	The address and port of the destination address.
addr_len	The length of addr data block.

Possible Errors

EBADF	Bad descriptor.
EFAULT	The socket structure address is outside the user's address space. This is caused by a bad addr structure reference.
ENOTSOCK	The descriptor is not associated with a socket.
EISCONN	The socket is already connected. You cannot reconnect a connected socket. Instead, you have to close the socket and create a new one.
ECONNREFUSED	Connection refused at server.
ETIMEDOUT	Timeout while attempting connection.
ENETUNREACH	Network is unreachable.
EADDRINUSE	Address is already in use.
EINPROGRESS	The socket is non-blocking, and the connection cannot be completed immediately. It is possible to select() or poll() for completion by selecting the socket for writing. After select() indicates writeability, use getsockopt() to read the SO_ERROR option at level SOL_SOCKET to determine whether the connection was successful (SO_ERROR is zero) or unsuccessful (SO_ERROR is one of the usual error codes previously listed, explaining the reason for the failure).

EALREADY	The socket is non-blocking, and a previous connection attempt has not yet been completed.
EAFNOSUPPORT	The passed address didn't have the correct address family in its sa_family field.
EACCES	The user tried to connect to a broadcast address without having the socket broadcast flag enabled.

Example

```
/*** Connect to a TCP server ***/
int sockfd;
struct sockaddr_in addr;
sockfd = socket(PF_INET, SOCK_STREAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = 13; /* current time */
inet_atoi("127.0.0.1", &addr.sin_addr);
if ( connect(sockfd, &addr, sizeof(addr)) != 0 )
    perror("connect");
```

socketpair()

Creates a pair of sockets that are linked together like a pipe but uses the socket subsystem. It is nearly identical to a pipe() system call but gives you the added functionality of the Socket API. socketpair() supports PF_UNIX or PF_LOCAL sockets only. You do not need to bind() this socket to a file system filename.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int socketpair(int domain, int type, int protocol,
    int sockfds[2]);
```

Return Value

Zero (0) if everything goes well. If an error occurs, you can find the cause in errno.

Parameters

domain	Must be PF_LOCAL or PF_UNIX.
type	SOCK_STREAM; This creates a socket like a pipe(). On some UNIX systems the pipe is bidirectional. Posix 1, however, does not require bidirectionality.
protocol	Must be zero (0).
sockfds[2]	An array of integers into which the call stores the new socket descriptors if successful.

Possible Errors

EMFILE	Too many descriptors are in use by this process.
EAFNOSUPPORT	The specified address family is not supported on this machine.
EPROTONOSUPPORT	The specified protocol is not supported on this machine.
EOPNOSUPPORT	The specified protocol does not support creation of socket pairs.
EFAULT	The address sockfds does not specify a valid part of the process address space.

Example

```
/*** Create a socket pair ***/
int sockfd[2];
struct sockaddr_ux addr;
if ( socketpair(PF_LOCAL, SOCK_STREAM, 0, sockfd) != 0 )
    perror("socketpair");
```

Communicating on a Channel

After establishing the socket and connection, you can use the API for sending and receiving messages. This section defines the calling API for all socket I/O.

send()

Send a message to the connected peer, client, or server. This call is like the write() system call, but send() permits you to define additional options for channel control.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int send(int sockfd, void *buffer, int msg_len,
    int options);
```

Return Value

Like write(), the call returns the number of bytes written. The byte count can be less that msg_len. If the call does not succeed in writing all required bytes, you can use a loop for successive writes. If negative, the call stores the error detail in errno.

Parameters

sockfd	The socket channel. This can be a connected SOCK_DGRAM or SOCK_STREAM socket.
buffer	The data to send.

`msg_len`	The number of bytes to send.
`options`	A set of flags to enable special message handling: • `MSG_OOB` Send message out-of-band (urgent). • `MSG_DONTROUTE` Send the message bypassing all routers. If unsuccessful, the network sends an error back. • `MSG_DONTWAIT` Don't allow blocking. Similar to the `fcntl()` option call, but only applies to this call. If the call would block, the call returns with `EWOULDBLOCK` in `errno`. • `MSG_NOSIGNAL` If the peer severs the connection, don't raise a `SIGPIPE` signal locally.

Possible Errors

`EBADF`	An invalid descriptor was specified.
`ENOTSOCK`	The argument `sockfd` is not a socket.
`EFAULT`	An invalid user space address was specified for `buffer`.
`EMSGSIZE`	The call can't complete because the socket requires that the message be sent atomically, and the size of the message to be sent has made this impossible.
`EAGAIN`	The socket is marked non-blocking and the requested operation would block.
`ENOBUFS`	The system was unable to allocate an internal memory block. The operation may succeed when buffers become available.
`EINTR`	A signal occurred.
`ENOMEM`	No memory available.
`EINVAL`	Invalid argument passed.
`EPIPE`	The local end has been shut down on a connected socket. In this case, the process will also receive a `SIGPIPE` unless `MSG_NOSIGNAL` is set.

Example

```
/*** Send a message (TCP, UDP) to a connected destination ***/
int sockfd;
int bytes, bytes_wrote=0;
/*--- Create socket, connect to server/peer ---*/
while ( (bytes = send(sockfd, buffer, msg_len, 0)) > 0 )
    if ( (bytes_wrote += bytes) >= msg_len )
        break;
if ( bytes < 0 )
    perror("send");
```

```
/*** Send an URGENT message (TCP) to a connected destination ***/
int sockfd;
int bytes, bytes_wrote=0;
/*--- Create socket, connect to server ---*/
if ( send(sockfd, buffer, 1, MSG_OOB) != 1 )
    perror("Urgent message");
```

sendto()

Send a message to a specific destination without connecting. Typically, you use this system call for UDP and raw sockets. Using this call for TCP sockets is Transaction TCP (T/TCP). (Linux, however, does not yet support T/TCP.)

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int sendto(int sockfd, void* msg, int len, int options,
    struct sockaddr *addr, int addr_len);
```

Return Value

Returns the number of bytes sent or `-1` if an error occurred.

Parameters

`sockfd`	The socket descriptor
`msg`	The data to send
`len`	The number of bytes to send
`options`	Message-controlling flags (same as `send()`)
`addr`	The address of the destination
`addr_len`	The size of the destination data body

Possible Errors

(Same as `send()`)

Example

```
/*** Send a message (TCP, UDP) to an UNconnected destination ***/
int sockfd;
struct sockaddr_in addr;
if ( (sd = socket(PF_INET, SOCK_DGRAM, 0)) < 0 )
    perror("socket");
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(DEST_PORT);
inet_aton(DEST_ADDR, &addr.sin_addr);
if ( send(sockfd, buffer, msg_len, 0, &addr, sizeof(addr)) < 0 )
    perror("sendto");
```

sendmsg()

Assemble a message from several blocks of data. This routine takes the data the `iovec` structure and creates a single message. If the `msg_name` points to a real `sockaddr`, the routine sends the message connectionless. If `NULL`, the routine assumes that the socket is connected.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
#include <sys/uio.h>
int sendmsg(int sockfd, const struct msghdr *msg,
    unsigned int options);
```

Return Value

This call returns the total number of bytes sent or `-1` if error (check `errno` for more information).

Parameters

`sockfd` — The open socket descriptor.

`msg` — This is a reference to a `msghdr` structure that holds destination, flags, and messages. The structure definitions are as follows:

```
struct iovec
{
   void *iov_base;        /* Buffer start */
 __kernel_size_t iov_len; /* Buffer length */
};

struct msghdr
{
   __ptr_t msg_name;      /* Dest address */
   socklen_t msg_namelen; /* Address length */
   struct iovec *msg_iov; /* Buffers vector */
   size_t msg_iovlen;     /* Vector length */
   __ptr_t msg_control;   /* Ancillary data */
   size_t msg_controllen; /* Ancillary data len */
   int msg_flags;         /* Received msg flags */
};
```

The ancillary data allows the program to transfer special data like file descriptors.

`options` — Message-controlling flags (same as `send()`).

Possible Errors

(Same as `send()`)

Example

```
int i, sd, len, bytes;
char buffer[MSGS][100];
struct iovec io[MSGS];
struct msghdr msg;
struct sockaddr_in addr;

sd = socket(PF_INET, SOCK_DGRAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
inet_aton(&addr.sin_addr, "127.0.0.1");
bzero(&msg, sizeof(msf));
msg.msg_name = &addr;
msg.msg_namelen = sizeof(addr);
for ( i = 0; i < MSGS; i++ )
{
    io[i].iov_base = buffer[i];
    sprintf(buffer[i], "Buffer #%d: this is a test\n", i);
    io[i].iov_len = strlen(buffer[i]);
}
msg.msg_iov = io;
msg.msg_iovlen = MSGS;
if ( (bytes = sendmsg(sd, &msg, 0)) < 0 )
    perror("sendmsg");
```

`sendfile()`

A fast way to transmit a file through a socket. The call reads the data from `in_fd` and writes it out to `out_fd`. The call does not change the file pointer in `in_fd`, but `out_fd`'s file pointer is changed. The call begins reading from `*offset` for count bytes. When done, `*offset` points to the byte after the last byte read. If you need to add header information, refer to the `TCP_CORK` option in TCP(4) to improve performance.

Prototype

```
#include <unistd>
int sendfile(int out_fd, int in_fd, off_t *offset,
     size_t count);
```

Return Value

If successful, the call returns the total number of bytes copied. If error, the call returns `-1` and sets `errno` to the error code.

Parameters

`out_fd`	The destination descriptor (file pointer modified)
`in_fd`	The source descriptor (file pointer not modified)
`offset`	A pointer to a variable that holds the starting offset
`count`	The number of bytes to send

Possible Errors

`EBADF`	Input file was not opened for reading or output file was not opened for writing.
`EINVAL`	Descriptor is not valid or locked.
`ENOMEM`	Insufficient memory for reading from `in_fd`.
`EIO`	Unspecified error while reading from `in_fd`.

Example

```
#include <unistd.h>
...
struct stat fdstat;
int client = accept(sd, 0, 0);
int fd = open("filename.gif", O_RDONLY);
fstat(fd, &fdstat);
sendfile(client, fd, 0, fdstat.st_size);
close(fd);
close(client);
```

`recv()`

Wait for and accept a message from a connected peer, client, or server. The system call behaves similarly to `read()` but adds control flags. UDP can use this call if connected to a peer.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int recv(int sockfd, void* buf, int maxbuf,
    int options);
```

Return Value

The number of bytes read or `-1` if an error occurred.

Parameters

`sockfd`	The open socket descriptor
`buf`	The byte array to accept the incoming message

`maxbuf`	The size of the array
`options`	A set of flags that can be arithmetically ORed together: • `MSG_OOB` This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and so this flag cannot be used with such protocols. • `MSG_PEEK` This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. • `MSG_WAITALL` This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned. • `MSG_ERRQUEUE` Receive packet from the error queue. • `MSG_NOSIGNAL` This flag turns off raising of `SIGPIPE` on stream sockets when the other end disappears. • `MSG_ERRQUEUE` This flag specifies that queued errors should be received from the socket error queue. The error is passed in an ancillary message with a type dependent on the protocol (for IP, `IP_RECVERR`). The error is supplied in a `sock_extended_error` structure.

Possible Errors

`EBADF`	The argument `s` is an invalid descriptor.
`ENOTCONN`	The socket is associated with a connection-oriented protocol and has not been connected (see `connect()` and `accept()`).
`ENOTSOCK`	The argument `sockfd` does not refer to a socket.
`EAGAIN`	The socket is marked as non-blocking, and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.
`EINTR`	A signal interrupted the receive operation before any data was available.
`EFAULT`	The receive buffer pointer points outside the process's address space.
`EINVAL`	An invalid argument was passed.

Example

```
/*** Recv a message (TCP, UDP) from a connected destination ***/
int sockfd;
int bytes, bytes_wrote=0;
/*--- Create socket, connect to server/peer ---*/
if ( (bytes = recv(sockfd, buffer, msg_len, 0)) < 0 )
    perror("send");

/*** Recv an URGENT message (TCP) from a connected destination ***/
/*** This code is typically in a SIGURG signal handler ***/
int sockfd;
int bytes, bytes_wrote=0;
/*--- Create socket, connect to server ---*/
if ( (bytes = recv(sockfd, buffer, msg_len, 0)) < 0 )
    perror("Urgent message");
```

`recvfrom()`

Wait for and receive a message from an unconnected peer (UDP and raw sockets). Please note that T/TCP never uses this call. Instead, the receiver uses `accept()`.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
int recvfrom(int sockfd, void *buf, int buf_len, int options,
    struct sockaddr *addr, int *addr_len);
```

Return Value

If successful, the number of bytes read. If an error occurred, the return value is `-1`, and `errno` holds the error code.

Parameters

`sockfd`	The open socket descriptor.
`buf`	The byte array to receive the data.
`buf_len`	The maximum size of the buffer (message truncated and dropped if buffer too small).
`options`	Channel controlling options (same as `recv()`).
`addr`	The sending peer's address and port.
`addr_len`	The maximum size of `addr` (address truncated if too small). Upon return, this value changes to match the number of bytes used.

Possible Errors

(Same as `recv()`)

Example

```
struct sockaddr_in addr;
int addr_len=sizeof(addr), bytes_read;
char buf[1024];
int sockfd = socket(PF_INET, SOCK_DGRAM, 0);
/***Bind socket to specific port***/
bytes_read = recvfrom(sockfd, buf, sizeof(buf), 0, &addr, &add_len);
if ( bytes_read < 0 )
    perror("recvfrom failed");
```

recvmsg()

Receive several messages at once from the same source. This call is customarily used with SOCK_DGRAM sockets (same reasoning as sendmsg()). The operation does not have the flexibility to accept from different sources.

Prototype

```
#include <sys/socket.h>
#include <resolv.h>
#include <sys/uio.h>
int recvmsg(int sockfd, struct msghdr *msg,
    unsigned int options);
```

Return Value

Total number of bytes received if no error occurred; otherwise, returns -1.

Parameters

sockfd	The socket descriptor waiting for a message
msg	The data received
options	Channel controlling options (same as recv())

Possible Errors

(Same as recv())

Example

```
char buffer[MSGS][1000];
struct sockaddr_in addr;
struct iovec io[MSGS];
struct msghdr msg;
...
bzero(&addr, sizeof(addr));
msg.msg_name = &addr;
msg.msg_namelen = sizeof(addr);
for ( i = 0; i < MSGS; i++ )
{
```

```
    io[i].iov_base = buffer[i];
    io[i].iov_len = sizeof(buffer[i]);
}
msg.msg_iov = io;
msg.msg_iovlen = MSGS;
if ( (bytes = recvmsg(sd, &msg, 0)) < 0 )
    perror("recvmsg");
```

Terminating Connections

The last step a solid program performs after communicating with the external host is to close the connection. This section describes the API for socket termination.

`shutdown()`

Closes specific paths or directions of data flow. Socket connections, by default, are bidirectional. If you want to limit the flow to be read-only or write-only, `shutdown` closes the other half of the channel.

Prototype

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

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in `errno`.

Parameters

`sockfd`	The open socket descriptor.
`how`	A flag indicating which half (or both) of channel to close:
	`SHUT_RD` (`0`)—Make the channel output only.
	`SHUT_WR` (`1`)—Make the channel input only.
	`SHUT_RDWR` (`2`)—Close both halves, functionally same as `close()`.
	This operates only on connected sockets.

Possible Errors

`EBADF`	`sockfd` is not a valid descriptor.
`ENOTSOCK`	`sockfd` is a file, not a socket.
`ENOTCONN`	The specified socket is not connected.

Example

```
int sockfd;
struct sockaddr_in addr;
```

```
sockfd = socket(PF_INET, SOCK_STREAM, 0);
bzero(&addr, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(DEST_PORT);
inet_aton(DEST_ADDR, &addr.sin_addr);
connect(sockfd, &addr, sizeof(addr));
if ( shutdown(sockfd, SHUT_WR) != 0 )
    PANIC("Can't make socket input-only");
```

Network Data Conversions

While working with data on the network, you must consider the data byte-ordering, converting addresses, and so on. The Socket API includes a sizeable list of tools to help you get the information you need. This section describes the tools that the book has used (and a few others).

`htons()` or `htonl()`

Convert host-byte order binary data to network-byte order. On a little-endian processor, the call swaps the bytes around. A big-endian host does nothing but return the value.

Prototype

```
#include <netinet/in.h>
unsigned short int htons(unsigned short int host_short);
unsigned long int htonl(unsigned long int host_lonf);
```

Return Value

(none)

Parameters

`host_short`	The 16-bit host value
`host_long`	The 32-bit host value

Possible Errors

(none)

Example

```
/*** Assign #1023 to socket ***/
struct sockaddr_in addr;
addr.sin_port = htons(1023);

/*** Assign 128.1.32.10 to the destination address ***/
struct sockaddr_in addr;
addr.sin_addr.s_addr = hton(0x8001200A);
```

`ntohs()` or `ntohl()`

Converts the network byte order to the host's byte order.

Prototype

```
#include <netinet/in.h>
unsigned short int ntohs(unsigned short int network_short);
unsigned long int ntohl(unsigned long int network_lonf);
```

Return Value

The converted value (16 bits or 32 bits).

Parameters

`network_short`	The 16-bit value to convert
`network_long`	The 32-bit value to convert

Possible Errors

(none)

Example

```
struct sockaddr_in addr;
int client, addrlen=sizeof(addr);
client = accept(sockfd, &addr, &addrlen);
if ( client > 0 )
    printf("Connected %lX:%d\n", ntohl(addr.sin_addr),
        ntohs(addr.sin_port));
```

`inet_addr()`

A deprecated conversion tool for converting numeric, dot-notation addresses into the network-byte-order binary form. If it fails, the return value (`-1` or `255.255.255.255`) is still a legal address. The `inet_aton()` tool has better error handling.

Prototype

```
#include <netinet/in.h>
unsigned long int inet_addr(const char *ip_address);
```

Return Value

Non-zero	If everything goes well, the value is the converted IP address.
`INADDR_NONE(-1)`	The parameter is not valid. (This is the call's defect—there is no negative value, and `255.255.255.255` is the general broadcast address.)

Parameters

`ip_address`	The human-readable, dot-notation form (for example, `128.187.34.2`)

Possible Errors

(`errno` not set)

Example

```
if ( (addr.sin_addr.s_addr = inet_addr("128.187.34.2")) == -1 )
    perror("Couldn't convert address");
```

inet_aton()

Converts a human-readable IP address from the dot-notation to the binary, network-byte ordered form. This call replaces `inet_addr()`.

Prototype

```
#include <netinet/in.h>
int inet_aton(const char* ip_addr, struct in_addr *addr);
```

Return Value

Non-zero if everything goes well. If an error occurs, the call returns a zero.

Parameters

`ip_addr`	The ASCII string of the IP address (for example, `187.34.2.1`).
`addr`	The destination. Typically, you would populate the `sin_addr` field from the `sockaddr_in` structure.

Possible Errors

(`errno` not set)

Example

```
Struct sockaddr_in addr;
if ( inet_aton("187.43.32.1", &addr.sin_addr) == 0 )
    perror("inet_aton() failed");
```

inet_ntoa()

This call converts the network-byte order binary to a human-readable form. Note that this call uses a static memory region; subsequent calls overwrite the older results.

Prototype

```
#include <netinet/in.h>
const char* inet_ntoa(struct in_addr *addr);
```

Return Value

The address is in the returned string.

Parameters

`addr`	The binary address (typically the address field of `struct sockaddr_in`)

Possible Errors
(errno not set)

Example
```
Clientfd = accept(serverfd, &addr, &addr_size);
if ( clientfd > 0 )
    printf("Connected %s:%d\n",
        inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
```

inet_pton()
Converts a human-readable IPv4 or IPv6 address from the dot/colon-notation to the binary, network-byte ordered form.

Prototype
```
#include <netinet/in.h>
int inet_pton(int domain, const char* prsnt, void *addr);
```

Return Value
Non-zero if everything goes well. If an error occurs, the call returns zero.

Parameters
domain	The network type (AF_INET or AF_INET6).
prsnt	The ASCII string of the IP address (for example, 187.34.2.1 or FFFF::8090:A03:3245).
addr	The destination. Typically, you would populate the sin_addr field from the sockaddr_in structure or sin6_addr field from the sockaddr_in6 structure.

Possible Errors
(errno not set)

Example
```
struct sockaddr_in addr;
if ( inet_pton(AF_INET6, "187.43.32.1", &addr.sin6_addr) == 0 )
    perror("inet_pton() failed");
```

inet_ntop()
This call converts the network-byte order binary to a human-readable form. Unlike the inet_ntoa, you need to supply it a scratchpad string. This function supports both AF_INET and AF_INET6.

Prototype
```
#include <arpa/inet.h>
char* inet_ntop(int domain, struct in_addr *addr, char* str, int len);
```

Return value

The call returns the parameter `str`.

Parameters

`domain`	The network type (`AF_INET` or `AF_INET6`)
`addr`	The binary address (typically the address field of `struct sockaddr_in`)
`str`	The string buffer
`len`	The number of bytes available in `str`

Possible Errors

(`errno` not set)

Example

```
char str[100];
clientfd = accept(serverfd, &addr, &addr_size);
if ( clientfd > 0 )
    printf("Connected %s:%d\n",
        inet_ntop(AF_INET, addr.sin_addr, str, sizeof(str)),
        ntohs(addr.sin_port));
```

Network Addressing Tools

Like the data and addressing tools, the API gives you access to the naming services. These services include domain name services (DNS), the protocols, and so on. This section describes the calls for converting human-readable names.

`getpeername()`

This routine gets the bound address or name of connected peer at the other end of the `sockfd` channel. The call places the results in `buf`. The `buf_len` is the number of bytes available in `buf`. If too small, the information gets truncated. This is the same information you get with the `accept()` call.

Prototype

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr,
    socklen_t *addr_len);
```

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in `errno`.

Parameters

`sockfd`	The connected socket channel.
`addr`	A buffer that holds the address structure.
`addr_len`	The number of bytes available in `addr`. This is passed by reference (the call changes this field).

Possible Errors

`EBADF`	The argument `sockfd` is not a valid descriptor.
`ENOTSOCK`	The argument `sockfd` is a file, not a socket.
`ENOTCONN`	The socket is not connected.
`ENOBUFS`	Insufficient resources were available in the system to perform the operation.
`EFAULT`	The `addr` parameter points to memory that is not in a valid part of the process address space.

Example

```
struct sockaddr_in addr;
int add_len = sizeof(addr);
if ( getpeername(client, &addr, &addr_len) != 0 )
    perror("getpeername() failed");
printf("Peer: %s:%d\n", inet_ntoa(addr.sin_addr),
    ntohs(addr.sin_port));
```

`gethostname()`

Gets the localhost's name. The call places the result in the name parameter up to `len` bytes.

Prototype

```
#include <unistd.h>
int gethostname(char *name, size_t len);
```

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in `errno`.

Parameters

`name`	The buffer to accept the name of the localhost
`len`	The number of bytes available in `name`

Possible Errors

`EINVAL`	`len` is negative or, for `gethostname()` on Linux/i386, `len` is smaller than the actual size.
`EFAULT`	`name` is an invalid address.

Example

```
char name[50];
if ( getpeername(name, sizeof(name)) != 0 )
    perror("getpeername() failed");
printf("My host is: %s\n", name);
```

gethostbyname()

Search for and translate the hostname to an IP address. The name can be a hostname or an address. If it is an address, the call does no searches; instead, it returns the address in the `h_name` and `h_addr_list[0]` fields of the `hostent` structure.

If the hostname ends with a period, the call treats the name as an absolute name with no abbreviation. Otherwise, the call searches the local subnetwork names. If you have `HOSTALIASES` defined in your environment, the call searches the file to which `HOSTALIASES` points.

Prototype

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

Return Value

The call returns a pointer `struct hostent`; if it fails, the return value is `NULL`. The structure lists all the names and addresses the host owns. The macro `h_addr` provides backward compatibility.

```
#define h_addr h_addr_list[0]
struct hostent {
    char *h_name;       /* official name of host */
    char **h_aliases;   /* alias list */
    int h_addrtype;     /* host address type */
    int h_length;       /* length of address */
    char **h_addr_list; /* list of addresses; 0th is the primary */
};
```

Parameters

`name`	The hostname to search or the IP address

Possible Errors

`ENOTFOUND`	The specified host is unknown.
`NO_ADDRESS, NO_DATA`	The requested name is valid but does not have an IP address.
`NO_RECOVERY`	A non-recoverable name server error occurred.
`EAGAIN`	A temporary error occurred on an authoritative name server. Try again later.

Example

```
int i;
struct hostent *host;
host = gethostbyname("sunsite.unc.edu");
if ( host != NULL )
{
    printf("Official name: %s\n", host->h_name);
    for ( i = 0; host->h_aliases[i] != 0; i++ )
        printf("   alias[%d]: %s\n", i+1,
            host->h_aliases[i]);
    printf("Address type=%d\n", host->h_addrtype);
    for ( i = 0; i < host->h_length; i++ )
        printf("Addr[%d]: %s\n", i+1,
            inet_ntoa(host->h_addr_list[i]));
}
else
    perror("sunsite.unc.edu");
```

getprotobyname()

This function reads the `/etc/protocol` file to get the protocol that matches `pname`. You use this call to translate names such as HTTP, FTP, and Telnet into their default port numbers.

Prototype

```
#include <netdb.h>
struct protoent *getprotobyname(const char *pname);
```

Return Value

The call returns a pointer to `protoent` (defined later in this appendix) if successful. Otherwise, it returns `NULL`.

```
struct protoent {
    char    *p_name;        /* official protocol name */
    char    **p_aliases;    /* alias list */
    int     p_proto;        /* protocol number */
};
```

The field `p_proto` is the port number.

Parameters

`pname`	Protocol name. This can be any of the recognized protocol names or aliases.

Possible Errors

(`errno` not set)

Example

```
#include <netdb.h>
...
int i;
struct protoent *proto = getprotobyname("http");
if ( proto != NULL )
{
    printf("Official name: %s\n", proto->name);
    printf("Port#: %d\n", proto->p_proto);
    for ( i = 0; proto->p_aliases[i] != 0; i++ )
        printf("Alias[%d]: %s\n", i+1,
            proto->p_aliases[i]);
}
else
    perror("http");
```

Socket Controls

While the socket is open, you can configure it to behave in various ways. This section describes the API calls.

`setsockopt()`

Change the behavior of socket `sd`. Every option has a value (some options are read- or write-only). You can set each option through `optval` and `optlen`. For a complete list of options, see Appendix A.

Prototype

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sd, int level, int optname,
    const void *optval, socklen_t optlen);
```

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in `errno`.

Parameters

`sd`	The socket to modify
`level`	The feature level (`SOL_SOCKET`, `SOL_IP`, `SOL_TCP`, `SOL_IPV6`)
`optname`	The option to revise
`optval`	A pointer to the new value
`optlen`	The length of value in bytes

Possible Errors

EBADF	The argument sd is not a valid descriptor.
ENOTSOCK	The argument sd is a file, not a socket.
ENOPROTOOPT	The option is unknown at the level indicated.
EFAULT	The address pointed to by optval is not in a valid part of the process address space.

Example

```
const int TTL=128;
/*---Change the time-to-live to 128 hops---*/
if ( setsockopt(sd, SOL_IP, SO_TTL, &TTL, sizeof(TTL)) != 0 )
    perror("setsockopt() failed");
```

getsockopt()

Get the socket configuration.

Prototype

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int sd, int level, int optname, void *optval,
    socklen_t *optlen);
```

Return Value

Zero if everything goes well. If an error occurs, you can find the cause in errno.

Parameters

sd	The socket to read.
level	The feature level (SOL_SOCKET, SOL_IP, SOL_TCP, SOL_IPV6).
optname	The option to revise.
optval	Place for value.
optlen	The length of value in bytes. This field is passed by reference.

Possible Errors

EBADF	The argument sd is not a valid descriptor.
ENOTSOCK	The argument sd is a file, not a socket.
ENOPROTOOPT	The option is unknown at the level indicated.
EFAULT	The address pointed to by optval or optlen is not in a valid part of the process address space.

Example

```
int error, size=sizeof(error);
if ( getsockopt(sd, SOL_SOCKET, SO_ERROR, &error,
        &size) != 0 )
    perror("getsockopt() failed");
printf("socket error=%d\n", error);
```