# 9. Man Pages

In the Unix world, there are a lot of manuals. They have little sections that describe individual functions that you have at your disposal.

Of course, **manual** would be too much of a thing to type. I mean, no one in the Unix world, including myself, likes to type that much. Indeed I could go on and on at great length about how much I prefer to be terse but instead I shall be brief and not bore you with long-winded diatribes about how utterly amazingly brief I prefer to be in virtually all circumstances in their entirety.

*[Applause]*

Thank you. What I am getting at is that these pages are called "man pages" in the Unix world, and I have included my own personal truncated variant here for your reading enjoyment. The thing is, many of these functions are way more general purpose than I'm letting on, but I'm only going to present the parts that are relevant for Internet Sockets Programming.

But wait! That's not all that's wrong with my man pages:

- They are incomplete and only show the basics from the guide.
- There are many more man pages than this in the real world.
- They are different than the ones on your system.
- The header files might be different for certain functions on your system.
- The function parameters might be different for certain functions on your system.

If you want the real information, check your local Unix man pages by typing **man whatever**, where "whatever" is something that you're incredibly interested in, such as "`accept`". (I'm sure Microsoft Visual Studio has something similar in their help section. But "man" is better because it is one byte more concise than "help". Unix wins again!)

So, if these are so flawed, why even include them at all in the Guide? Well, there are a few reasons, but the best are that (a) these versions are geared specifically toward network programming and are easier to digest than the real ones, and (b) these versions contain examples!

Oh! And speaking of the examples, I don't tend to put in all the error checking because it really increases the length of the code. But you should absolutely do error checking pretty much any time you make any of the system calls unless you're totally 100% sure it's not going to fail, and you should probably do it even then!

## 9.1. `accept()`

Accept an incoming connection on a listening socket

### Prototypes

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

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

### Description

Once you've gone through the trouble of getting a `SOCK_STREAM` socket and setting it up for incoming connections with **`listen()`**, then you call **`accept()`** to actually get yourself a new socket descriptor to use for subsequent communication with the newly connected client.

The old socket that you are using for listening is still there, and will be used for further **`accept()`** calls as they come in.

| | |
|---|---|
| *`s`* | The **`listen()`**ing socket descriptor. |
| *`addr`* | This is filled in with the address of the site that's connecting to you. |
| *`addrlen`* | This is filled in with the **`sizeof()`** the structure returned in the *`addr`* parameter. You can safely ignore it if you assume you're getting a `struct sockaddr_in` back, which you know you are, because that's the type you passed in for *`addr`*. |

**`accept()`** will normally block, and you can use **`select()`** to peek on the listening socket descriptor ahead of time to see if it's “ready to read”. If so, then there's a new connection waiting to be **`accept()`**ed! Yay! Alternatively, you could set the `O_NONBLOCK` flag on the listening socket using **`fcntl()`**, and then it will never block, choosing instead to return `-1` with *`errno`* set to `EWOULDBLOCK`.

The socket descriptor returned by **`accept()`** is a bona fide socket descriptor, open and connected to the remote host. You have to **`close()`** it when you're done with it.

### Return Value

**`accept()`** returns the newly connected socket descriptor, or `-1` on error, with *`errno`* set appropriately.

### Example

```
struct sockaddr_storage their_addr;
socklen_t addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, MYPORT, &hints, &res);

// make a socket, bind it, and listen on it:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);

// now accept an incoming connection:

addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
```

```
// ready to communicate on socket descriptor new_fd!
```

### See Also

**`socket()`**, **`getaddrinfo()`**, **`listen()`**, `struct sockaddr_in`

## 9.2. bind()

Associate a socket with an IP address and port number

### Prototypes

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

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

### Description

When a remote machine wants to connect to your server program, it needs two pieces of information: the IP address and the port number. The **bind()** call allows you to do just that.

First, you call **getaddrinfo()** to load up a **struct sockaddr** with the destination address and port information. Then you call **socket()** to get a socket descriptor, and then you pass the socket and address into **bind()**, and the IP address and port are magically (using actual magic) bound to the socket!

If you don't know your IP address, or you know you only have one IP address on the machine, or you don't care which of the machine's IP addresses is used, you can simply pass the `AI_PASSIVE` flag in the *hints* parameter to **getaddrinfo()**. What this does is fill in the IP address part of the **struct sockaddr** with a special value that tells **bind()** that it should automatically fill in this host's IP address.

What what? What special value is loaded into the **struct sockaddr**'s IP address to cause it to auto-fill the address with the current host? I'll tell you, but keep in mind this is only if you're filling out the `struct sockaddr` by hand; if not, use the results from **getaddrinfo()**, as per above. In IPv4, the *sin_addr.s_addr* field of the `struct sockaddr_in` structure is set to `INADDR_ANY`. In IPv6, the *sin6_addr* field of the `struct sockaddr_in6` structure is assigned into from the global variable *in6addr_any*. Or, if you're declaring a new `struct in6_addr`, you can initialize it to `IN6ADDR_ANY_INIT`.

Lastly, the *addrlen* parameter should be set to `sizeof my_addr`.

### Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

### Example

```
// modern way of doing things with getaddrinfo()

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:
// (you should actually walk the "res" linked list and error-check!)

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

```
// example of packing a struct by hand, IPv4

struct sockaddr_in myaddr;
int s;
```

```
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// you can specify an IP address:
inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr));

// or you can let it automatically select one:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

### See Also

**`getaddrinfo()`**, **`socket()`**, `struct sockaddr_in`, `struct in_addr`

## 9.3. connect()

Connect a socket to a server

### Prototypes

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

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

### Description

Once you've built a socket descriptor with the **socket()** call, you can **connect()** that socket to a remote server using the well-named **connect()** system call. All you need to do is pass it the socket descriptor and the address of the server you're interested in getting to know better. (Oh, and the length of the address, which is commonly passed to functions like this.)

Usually this information comes along as the result of a call to **getaddrinfo()**, but you can fill out your own `struct sockaddr` if you want to.

If you haven't yet called **bind()** on the socket descriptor, it is automatically bound to your IP address and a random local port. This is usually just fine with you if you're not a server, since you really don't care what your local port is; you only care what the remote port is so you can put it in the *serv_addr* parameter. You *can* call **bind()** if you really want your client socket to be on a specific IP address and port, but this is pretty rare.

Once the socket is **connect()**ed, you're free to **send()** and **recv()** data on it to your heart's content.

Special note: if you **connect()** a `SOCK_DGRAM` UDP socket to a remote host, you can use **send()** and **recv()** as well as **sendto()** and **recvfrom()**. If you want.

### Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

### Example

```
// connect to www.example.com port 80 (http)

struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;

// we could put "80" instead on "http" on the next line:
getaddrinfo("www.example.com", "http", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect it to the address and port we passed in to getaddrinfo():

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

### See Also

**socket()**, **bind()**

## 9.4. `close()`

Close a socket descriptor

### Prototypes

```
#include <unistd.h>

int close(int s);
```

### Description

After you've finished using the socket for whatever demented scheme you have concocted and you don't want to **`send()`** or **`recv()`** or, indeed, do *anything else* at all with the socket, you can **`close()`** it, and it'll be freed up, never to be used again.

The remote side can tell if this happens one of two ways. One: if the remote side calls **`recv()`**, it will return `0`. Two: if the remote side calls **`send()`**, it'll receive a signal `SIGPIPE` and send() will return `-1` and *`errno`* will be set to `EPIPE`.

**Windows users**: the function you need to use is called **`closesocket()`**, not **`close()`**. If you try to use **`close()`** on a socket descriptor, it's possible Windows will get angry... And you wouldn't like it when it's angry.

### Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

### Example

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// a whole lotta stuff...*BRRRONNNN!*
.
.
.
close(s);  // not much to it, really.
```

### See Also

**`socket()`**, **`shutdown()`**

## 9.5. `getaddrinfo()`, `freeaddrinfo()`, `gai_strerror()`

Get information about a host name and/or service and load up a `struct sockaddr` with the result.

### Prototypes

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

int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *ai);

const char *gai_strerror(int ecode);

struct addrinfo {
  int     ai_flags;          // AI_PASSIVE, AI_CANONNAME, ...
  int     ai_family;         // AF_xxx
  int     ai_socktype;       // SOCK_xxx
  int     ai_protocol;       // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP

  socklen_t  ai_addrlen;     // length of ai_addr
  char   *ai_canonname;      // canonical name for nodename
  struct sockaddr  *ai_addr; // binary address
  struct addrinfo  *ai_next; // next structure in linked list
};
```

### Description

**`getaddrinfo()`** is an excellent function that will return information on a particular host name (such as its IP address) and load up a `struct sockaddr` for you, taking care of the gritty details (like if it's IPv4 or IPv6.) It replaces the old functions **`gethostbyname()`** and **`getservbyname()`**.The description, below, contains a lot of information that might be a little daunting, but actual usage is pretty simple. It might be worth it to check out the examples first.

The host name that you're interested in goes in the *`nodename`* parameter. The address can be either a host name, like “www.example.com”, or an IPv4 or IPv6 address (passed as a string). This parameter can also be `NULL` if you're using the `AI_PASSIVE` flag (see below.)

The *`servname`* parameter is basically the port number. It can be a port number (passed as a string, like “80”), or it can be a service name, like “http” or “tftp” or “smtp” or “pop”, etc. Well-known service names can be found in the IANA Port List[42] or in your *`/etc/services`* file.

Lastly, for input parameters, we have *`hints`*. This is really where you get to define what the **`getaddinfo()`** function is going to do. Zero the whole structure before use with **`memset()`**. Let's take a look at the fields you need to set up before use.

The *`ai_flags`* can be set to a variety of things, but here are a couple important ones. (Multiple flags can be specified by bitwise-ORing them together with the **`|`** operator.) Check your man page for the complete list of flags.

`AI_CANONNAME` causes the *`ai_canonname`* of the result to the filled out with the host's canonical (real) name. `AI_PASSIVE` causes the result's IP address to be filled out with `INADDR_ANY` (IPv4)or *`in6addr_any`* (IPv6); this causes a subsequent call to **`bind()`** to auto-fill the IP address of the `struct sockaddr` with the address of the current host. That's excellent for setting up a server when you don't want to hardcode the address.

If you do use the `AI_PASSIVE`, flag, then you can pass `NULL` in the *`nodename`* (since **`bind()`** will fill it in for you later.)

42. `http://www.iana.org/assignments/port-numbers`

Continuing on with the input paramters, you'll likely want to set *ai_family* to AF_UNSPEC which tells **getaddrinfo()** to look for both IPv4 and IPv6 addresses. You can also restrict yourself to one or the other with AF_INET or AF_INET6.

Next, the *socktype* field should be set to SOCK_STREAM or SOCK_DGRAM, depending on which type of socket you want.

Finally, just leave *ai_protocol* at 0 to automatically choose your protocol type.

Now, after you get all that stuff in there, you can *finally* make the call to **getaddrinfo()**!

Of course, this is where the fun begins. The *res* will now point to a linked list of struct addrinfos, and you can go through this list to get all the addresses that match what you passed in with the hints.

Now, it's possible to get some addresses that don't work for one reason or another, so what the Linux man page does is loops through the list doing a call to **socket()** and **connect()** (or **bind()** if you're setting up a server with the AI_PASSIVE flag) until it succeeds.

Finally, when you're done with the linked list, you need to call **freeaddrinfo()** to free up the memory (or it will be leaked, and Some People will get upset.)

## Return Value

Returns zero on success, or nonzero on error. If it returns nonzero, you can use the function **gai_strerror()** to get a printable version of the error code in the return value.

## Example

```
// code for a client connecting to a server
// namely a stream socket to www.example.com on port 80 (http)
// either IPv4 or IPv6

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("connect");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no connection
    fprintf(stderr, "failed to connect\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

```
// code for a server waiting for connections
// namely a stream socket on port 3490, on this host's IP
// either IPv4 or IPv6.

int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; // use my IP address

if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    exit(1);
}

// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("socket");
        continue;
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("bind");
        continue;
    }

    break; // if we get here, we must have connected successfully
}

if (p == NULL) {
    // looped off the end of the list with no successful bind
    fprintf(stderr, "failed to bind socket\n");
    exit(2);
}

freeaddrinfo(servinfo); // all done with this structure
```

### See Also

**`gethostbyname()`**, **`getnameinfo()`**

## 9.6. gethostname()

Returns the name of the system

### Prototypes

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

int gethostname(char *name, size_t len);
```

### Description

Your system has a name. They all do. This is a slightly more Unixy thing than the rest of the networky stuff we've been talking about, but it still has its uses.

For instance, you can get your host name, and then call **`gethostbyname()`** to find out your IP address.

The parameter *`name`* should point to a buffer that will hold the host name, and *`len`* is the size of that buffer in bytes. **`gethostname()`** won't overwrite the end of the buffer (it might return an error, or it might just stop writing), and it will `NUL`-terminate the string if there's room for it in the buffer.

### Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

### Example

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("My hostname: %s\n", hostname);
```

### See Also

**`gethostbyname()`**

## 9.7. `gethostbyname()`, `gethostbyaddr()`

Get an IP address for a hostname, or vice-versa

### Prototypes

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname(const char *name); // DEPRECATED!
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

### Description

*PLEASE NOTE: these two functions are superseded by **`getaddrinfo()`** and **`getnameinfo()`**!* In particular, **`gethostbyname()`** doesn't work well with IPv6.

These functions map back and forth between host names and IP addresses. For instance, if you have “www.example.com”, you can use **`gethostbyname()`** to get its IP address and store it in a `struct in_addr`.

Conversely, if you have a `struct in_addr` or a `struct in6_addr`, you can use **`gethostbyaddr()`** to get the hostname back. **`gethostbyaddr()`** *is* IPv6 compatible, but you should use the newer shinier **`getnameinfo()`** instead.

(If you have a string containing an IP address in dots-and-numbers format that you want to look up the hostname of, you'd be better off using **`getaddrinfo()`** with the `AI_CANONNAME` flag.)

**`gethostbyname()`** takes a string like “www.yahoo.com”, and returns a `struct hostent` which contains tons of information, including the IP address. (Other information is the official host name, a list of aliases, the address type, the length of the addresses, and the list of addresses—it's a general-purpose structure that's pretty easy to use for our specific purposes once you see how.)

**`gethostbyaddr()`** takes a `struct in_addr` or `struct in6_addr` and brings you up a corresponding host name (if there is one), so it's sort of the reverse of **`gethostbyname()`**. As for parameters, even though *`addr`* is a `char*`, you actually want to pass in a pointer to a `struct in_addr`. *`len`* should be `sizeof(struct in_addr)`, and *`type`* should be `AF_INET`.

So what is this `struct hostent` that gets returned? It has a number of fields that contain information about the host in question.

| | |
|---|---|
| *`char *h_name`* | The real canonical host name. |
| *`char **h_aliases`* | A list of aliases that can be accessed with arrays—the last element is `NULL` |
| *`int h_addrtype`* | The result's address type, which really should be `AF_INET` for our purposes. |
| *`int length`* | The length of the addresses in bytes, which is 4 for IP (version 4) addresses. |
| *`char **h_addr_list`* | A list of IP addresses for this host. Although this is a `char**`, it's really an array of `struct in_addr*`s in disguise. The last array element is `NULL`. |
| *`h_addr`* | A commonly defined alias for *`h_addr_list[0]`*. If you just want any old IP address for this host (yeah, they can have more than one) just use this field. |

### Return Value

Returns a pointer to a resultant `struct hostent` or success, or `NULL` on error.

Instead of the normal **`perror()`** and all that stuff you'd normally use for error reporting, these functions have parallel results in the variable *`h_errno`*, which can be printed using the functions **`herror()`** or **`hstrerror()`**. These work just like the classic *`errno`*, **`perror()`**, and **`strerror()`** functions you're used to.

### Example

```
// THIS IS A DEPRECATED METHOD OF GETTING HOST NAMES
// use getaddrinfo() instead!

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int i;
    struct hostent *he;
    struct in_addr **addr_list;

    if (argc != 2) {
        fprintf(stderr,"usage: ghbn hostname\n");
        return 1;
    }

    if ((he = gethostbyname(argv[1])) == NULL) {  // get the host info
        herror("gethostbyname");
        return 2;
    }

    // print information about this host:
    printf("Official name is: %s\n", he->h_name);
    printf("    IP addresses: ");
    addr_list = (struct in_addr **)he->h_addr_list;
    for(i = 0; addr_list[i] != NULL; i++) {
        printf("%s ", inet_ntoa(*addr_list[i]));
    }
    printf("\n");

    return 0;
}
```

```
// THIS HAS BEEN SUPERCEDED
// use getnameinfo() instead!

struct hostent *he;
struct in_addr ipv4addr;
struct in6_addr ipv6addr;

inet_pton(AF_INET, "192.0.2.34", &ipv4addr);
he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET);
printf("Host name: %s\n", he->h_name);

inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr);
he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6);
printf("Host name: %s\n", he->h_name);
```

### See Also

**getaddrinfo()**, **getnameinfo()**, **gethostname()**, *errno*, **perror()**, **strerror()**, struct in_addr

## 9.8. getnameinfo()

Look up the host name and service name information for a given `struct sockaddr`.

### Prototypes

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

### Description

This function is the opposite of **`getaddrinfo()`**, that is, this function takes an already loaded `struct sockaddr` and does a name and service name lookup on it. It replaces the old **`gethostbyaddr()`** and **`getservbyport()`** functions.

You have to pass in a pointer to a `struct sockaddr` (which in actuality is probably a `struct sockaddr_in` or `struct sockaddr_in6` that you've cast) in the *`sa`* parameter, and the length of that `struct` in the *`salen`*.

The resultant host name and service name will be written to the area pointed to by the *`host`* and *`serv`* parameters. Of course, you have to specify the max lengths of these buffers in *`hostlen`* and *`servlen`*.

Finally, there are several flags you can pass, but here a a couple good ones. `NI_NOFQDN` will cause the *`host`* to only contain the host name, not the whole domain name. `NI_NAMEREQD` will cause the function to fail if the name cannot be found with a DNS lookup (if you don't specify this flag and the name can't be found, **`getnameinfo()`** will put a string version of the IP address in *`host`* instead.)

As always, check your local man pages for the full scoop.

### Return Value

Returns zero on success, or non-zero on error. If the return value is non-zero, it can be passed to **`gai_strerror()`** to get a human-readable string. See **`getaddrinfo`** for more information.

### Example

```
struct sockaddr_in6 sa; // could be IPv4 if you want
char host[1024];
char service[20];

// pretend sa is full of good information about the host and port...

getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0);

printf("   host: %s\n", host);    // e.g. "www.example.com"
printf("service: %s\n", service); // e.g. "http"
```

### See Also

**`getaddrinfo()`**, **`gethostbyaddr()`**

## 9.9. getpeername()

Return address info about the remote side of the connection

### Prototypes

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

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

### Description

Once you have either **accept()**ed a remote connection, or **connect()**ed to a server, you now have what is known as a *peer*. Your peer is simply the computer you're connected to, identified by an IP address and a port. So...

**getpeername()** simply returns a `struct sockaddr_in` filled with information about the machine you're connected to.

Why is it called a “name”? Well, there are a lot of different kinds of sockets, not just Internet Sockets like we're using in this guide, and so “name” was a nice generic term that covered all cases. In our case, though, the peer's “name” is it's IP address and port.

Although the function returns the size of the resultant address in *`len`*, you must preload *`len`* with the size of *`addr`*.

### Return Value

Returns zero on success, or `-1` on error (and **errno** will be set accordingly.)

### Example

```
// assume s is a connected socket

socklen_t len;
struct sockaddr_storage addr;
char ipstr[INET6_ADDRSTRLEN];
int port;

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);

// deal with both IPv4 and IPv6:
if (addr.ss_family == AF_INET) {
    struct sockaddr_in *s = (struct sockaddr_in *)&addr;
    port = ntohs(s->sin_port);
    inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr);
} else { // AF_INET6
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr;
    port = ntohs(s->sin6_port);
    inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr);
}

printf("Peer IP address: %s\n", ipstr);
printf("Peer port      : %d\n", port);
```

### See Also

**gethostname()**, **gethostbyname()**, **gethostbyaddr()**

## 9.10. *errno*

Holds the error code for the last system call

### Prototypes

```
#include <errno.h>

int errno;
```

### Description

This is the variable that holds error information for a lot of system calls. If you'll recall, things like **`socket()`** and **`listen()`** return `-1` on error, and they set the exact value of *`errno`* to let you know specifically which error occurred.

The header file *`errno.h`* lists a bunch of constant symbolic names for errors, such as `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, etc. Your local man pages will tell you what codes can be returned as an error, and you can use these at run time to handle different errors in different ways.

Or, more commonly, you can call **`perror()`** or **`strerror()`** to get a human-readable version of the error.

One thing to note, for you multithreading enthusiasts, is that on most systems *`errno`* is defined in a threadsafe manner. (That is, it's not actually a global variable, but it behaves just like a global variable would in a single-threaded environment.)

### Return Value

The value of the variable is the latest error to have transpired, which might be the code for "success" if the last action succeeded.

### Example

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // or use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // an error has occurred!!

    // if we were only interrupted, just restart the select() call:
    if (errno == EINTR) goto tryagain;  // AAAA!  goto!!!

    // otherwise it's a more serious error:
    perror("select");
    exit(1);
}
```

### See Also

**`perror()`**, **`strerror()`**

## 9.11. `fcntl()`

Control socket descriptors

### Prototypes

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

### Description

This function is typically used to do file locking and other file-oriented stuff, but it also has a couple socket-related functions that you might see or use from time to time.

Parameter *`s`* is the socket descriptor you wish to operate on, *`cmd`* should be set to `F_SETFL`, and *`arg`* can be one of the following commands. (Like I said, there's more to **`fcntl()`** than I'm letting on here, but I'm trying to stay socket-oriented.)

| | |
|---|---|
| `O_NONBLOCK` | Set the socket to be non-blocking. See the section on blocking for more details. |
| `O_ASYNC` | Set the socket to do asynchronous I/O. When data is ready to be **`recv()`**'d on the socket, the signal `SIGIO` will be raised. This is rare to see, and beyond the scope of the guide. And I think it's only available on certain systems. |

### Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

Different uses of the **`fcntl()`** system call actually have different return values, but I haven't covered them here because they're not socket-related. See your local **`fcntl()`** man page for more information.

### Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // set to non-blocking
fcntl(s, F_SETFL, O_ASYNC);     // set to asynchronous I/O
```

### See Also

Blocking, **`send()`**

## 9.12. `htons()`, `htonl()`, `ntohs()`, `ntohl()`

Convert multi-byte integer types from host byte order to network byte order

### Prototypes

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

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

### Description

Just to make you really unhappy, different computers use different byte orderings internally for their multibyte integers (i.e. any integer that's larger than a `char`.) The upshot of this is that if you **`send()`** a two-byte `short int` from an Intel box to a Mac (before they became Intel boxes, too, I mean), what one computer thinks is the number `1`, the other will think is the number `256`, and vice-versa.

The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to “big-endian” before sending them out. Since Intel is a “little-endian” machine, it's far more politically correct to call our preferred byte ordering “Network Byte Order”. So these functions convert from your native byte order to network byte order and back again.

(This means on Intel these functions swap all the bytes around, and on PowerPC they do nothing because the bytes are already in Network Byte Order. But you should always use them in your code anyway, since someone might want to build it on an Intel machine and still have things work properly.)

Note that the types involved are 32-bit (4 byte, probably `int`) and 16-bit (2 byte, very likely `short`) numbers. 64-bit machines might have a **`htonll()`** for 64-bit `int`s, but I've not seen it. You'll just have to write your own.

Anyway, the way these functions work is that you first decide if you're converting *from* host (your machine's) byte order or from network byte order. If “host”, the the first letter of the function you're going to call is “h”. Otherwise it's “n” for “network”. The middle of the function name is always “to” because you're converting from one “to” another, and the penultimate letter shows what you're converting *to*. The last letter is the size of the data, “s” for short, or “l” for long. Thus:

| | |
|---|---|
| **`htons()`** | **h**ost **`to`** **n**etwork **s**hort |
| **`htonl()`** | **h**ost **`to`** **n**etwork **`l`**ong |
| **`ntohs()`** | **n**etwork **`to`** **h**ost **s**hort |
| **`ntohl()`** | **n**etwork **`to`** **h**ost **`l`**ong |

### Return Value

Each function returns the converted value.

### Example

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// convert and send
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // this expression is true
```

## 9.13. `inet_ntoa()`, `inet_aton()`, `inet_addr`

Convert IP addresses from a dots-and-number string to a `struct in_addr` and back

### Prototypes

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

// ALL THESE ARE DEPRECATED!  Use inet_pton()  or inet_ntop() instead!!

char *inet_ntoa(struct in_addr in);
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
```

### Description

*These functions are deprecated because they don't handle IPv6! Use* ***`inet_ntop()`*** *or* ***`inet_pton()`*** *instead! They are included here because they can still be found in the wild.*

All of these functions convert from a `struct in_addr` (part of your `struct sockaddr_in`, most likely) to a string in dots-and-numbers format (e.g. “192.168.5.10”) and vice-versa. If you have an IP address passed on the command line or something, this is the easiest way to get a `struct in_addr` to **`connect()`** to, or whatever. If you need more power, try some of the DNS functions like **`gethostbyname()`** or attempt a *coup d'État* in your local country.

The function **`inet_ntoa()`** converts a network address in a `struct in_addr` to a dots-and-numbers format string. The “n” in “ntoa” stands for network, and the “a” stands for ASCII for historical reasons (so it's “Network To ASCII”—the “toa” suffix has an analogous friend in the C library called **`atoi()`** which converts an ASCII string to an integer.)

The function **`inet_aton()`** is the opposite, converting from a dots-and-numbers string into a `in_addr_t` (which is the type of the field `s_addr` in your `struct in_addr`.)

Finally, the function **`inet_addr()`** is an older function that does basically the same thing as **`inet_aton()`**. It's theoretically deprecated, but you'll see it a lot and the police won't come get you if you use it.

### Return Value

**`inet_aton()`** returns non-zero if the address is a valid one, and it returns zero if the address is invalid.

**`inet_ntoa()`** returns the dots-and-numbers string in a static buffer that is overwritten with each call to the function.

**`inet_addr()`** returns the address as an `in_addr_t`, or `-1` if there's an error. (That is the same result as if you tried to convert the string “255.255.255.255”, which is a valid IP address. This is why **`inet_aton()`** is better.)

### Example

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope

some_addr = inet_ntoa(antelope.sin_addr); // return the IP
printf("%s\n", some_addr); // prints "10.0.0.1"

// and this call is the same as the inet_aton() call, above:
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

### See Also

**`inet_ntop()`**, **`inet_pton()`**, **`gethostbyname()`**, **`gethostbyaddr()`**

## 9.14. `inet_ntop()`, `inet_pton()`

Convert IP addresses to human-readable form and back.

### Prototypes

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);

int inet_pton(int af, const char *src, void *dst);
```

### Description

These functions are for dealing with human-readable IP addresses and converting them to their binary representation for use with various functions and system calls. The “n” stands for “network”, and “p” for “presentation”. Or “text presentation”. But you can think of it as “printable”. “ntop” is “network to printable”. See?

Sometimes you don't want to look at a pile of binary numbers when looking at an IP address. You want it in a nice printable form, like `192.0.2.180`, or `2001:db8:8714:3a90::12`. In that case, **`inet_ntop()`** is for you.

**`inet_ntop()`** takes the address family in the *`af`* parameter (either `AF_INET` or `AF_INET6`). The *`src`* parameter should be a pointer to either a `struct in_addr` or `struct in6_addr` containing the address you wish to convert to a string. Finally *`dst`* and *`size`* are the pointer to the destination string and the maximum length of that string.

What should the maximum length of the *`dst`* string be? What is the maximum length for IPv4 and IPv6 addresses? Fortunately there are a couple of macros to help you out. The maximum lengths are: `INET_ADDRSTRLEN` and `INET6_ADDRSTRLEN`.

Other times, you might have a string containing an IP address in readable form, and you want to pack it into a `struct sockaddr_in` or a `struct sockaddr_in6`. In that case, the opposite funcion **`inet_pton()`** is what you're after.

**`inet_pton()`** also takes an address family (either `AF_INET` or `AF_INET6`) in the *`af`* parameter. The *`src`* parameter is a pointer to a string containing the IP address in printable form. Lastly the *`dst`* parameter points to where the result should be stored, which is probably a `struct in_addr` or `struct in6_addr`.

These functions don't do DNS lookups—you'll need **`getaddinfo()`** for that.

### Return Value

**`inet_ntop()`** returns the *`dst`* parameter on success, or `NULL` on failure (and *`errno`* is set).

**`inet_pton()`** returns `1` on success. It returns `-1` if there was an error (*`errno`* is set), or `0` if the input isn't a valid IP address.

### Example

```
// IPv4 demo of inet_ntop() and inet_pton()

struct sockaddr_in sa;
char str[INET_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr));

// now get it back and print it
inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN);

printf("%s\n", str); // prints "192.0.2.33"
```

```
// IPv6 demo of inet_ntop() and inet_pton()
// (basically the same except with a bunch of 6s thrown around)

struct sockaddr_in6 sa;
```

```
char str[INET6_ADDRSTRLEN];

// store this IP address in sa:
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr));

// now get it back and print it
inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN);

printf("%s\n", str); // prints "2001:db8:8714:3a90::12"
```

```
// Helper function you can use:

//Convert a struct sockaddr address to a string, IPv4 and IPv6:

char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen)
{
    switch(sa->sa_family) {
        case AF_INET:
            inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr),
                    s, maxlen);
            break;

        case AF_INET6:
            inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr),
                    s, maxlen);
            break;

        default:
            strncpy(s, "Unknown AF", maxlen);
            return NULL;
    }

    return s;
}
```

### See Also

**`getaddrinfo()`**

## 9.15. `listen()`

Tell a socket to listen for incoming connections

### Prototypes

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

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

### Description

You can take your socket descriptor (made with the **`socket()`** system call) and tell it to listen for incoming connections. This is what differentiates the servers from the clients, guys.

The *`backlog`* parameter can mean a couple different things depending on the system you on, but loosely it is how many pending connections you can have before the kernel starts rejecting new ones. So as the new connections come in, you should be quick to **`accept()`** them so that the backlog doesn't fill. Try setting it to 10 or so, and if your clients start getting “Connection refused” under heavy load, set it higher.

Before calling **`listen()`**, your server should call **`bind()`** to attach itself to a specific port number. That port number (on the server's IP address) will be the one that clients connect to.

### Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

### Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);

listen(sockfd, 10); // set s up to be a server (listening) socket

// then have an accept() loop down here somewhere
```

### See Also

**`accept()`**, **`bind()`**, **`socket()`**

## 9.16. perror(), strerror()

Print an error as a human-readable string

### Prototypes

```
#include <stdio.h>
#include <string.h>   // for strerror()

void perror(const char *s);
char *strerror(int errnum);
```

### Description

Since so many functions return `-1` on error and set the value of the variable *errno* to be some number, it would sure be nice if you could easily print that in a form that made sense to you.

Mercifully, **perror()** does that. If you want more description to be printed before the error, you can point the parameter *s* to it (or you can leave *s* as `NULL` and nothing additional will be printed.)

In a nutshell, this function takes *errno* values, like `ECONNRESET`, and prints them nicely, like “Connection reset by peer.”

The function **strerror()** is very similar to **perror()**, except it returns a pointer to the error message string for a given value (you usually pass in the variable *errno*.)

### Return Value

**strerror()** returns a pointer to the error message string.

### Example

```
int s;

s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // some error has occurred
    // prints "socket error: " + the error message:
    perror("socket error");
}

// similarly:
if (listen(s, 10) == -1) {
    // this prints "an error: " + the error message from errno:
    printf("an error: %s\n", strerror(errno));
}
```

### See Also

*errno*

## 9.17. poll()

Test for events on multiple sockets simultaneously

### Prototypes

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

int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

### Description

This function is very similar to **select()** in that they both watch sets of file descriptors for events, such as incoming data ready to **recv()**, socket ready to **send()** data to, out-of-band data ready to **recv()**, errors, etc.

The basic idea is that you pass an array of *nfds* `struct pollfd`s in *ufds*, along with a timeout in milliseconds (1000 milliseconds in a second.) The *timeout* can be negative if you want to wait forever. If no event happens on any of the socket descriptors by the timeout, **poll()** will return.

Each element in the array of `struct pollfd`s represents one socket descriptor, and contains the following fields:

```
struct pollfd {
    int fd;         // the socket descriptor
    short events;   // bitmap of events we're interested in
    short revents;  // when poll() returns, bitmap of events that occurred
};
```

Before calling **poll()**, load *fd* with the socket descriptor (if you set *fd* to a negative number, this `struct pollfd` is ignored and its *revents* field is set to zero) and then construct the *events* field by bitwise-ORing the following macros:

| | |
|---|---|
| `POLLIN` | Alert me when data is ready to **recv()** on this socket. |
| `POLLOUT` | Alert me when I can **send()** data to this socket without blocking. |
| `POLLPRI` | Alert me when out-of-band data is ready to **recv()** on this socket. |

Once the **poll()** call returns, the *revents* field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present:

| | |
|---|---|
| `POLLERR` | An error has occurred on this socket. |
| `POLLHUP` | The remote side of the connection hung up. |
| `POLLNVAL` | Something was wrong with the socket descriptor *fd*—maybe it's uninitialized? |

### Return Value

Returns the number of elements in the *ufds* array that have had event occur on them; this can be zero if the timeout occurred. Also returns `-1` on error (and **errno** will be set accordingly.)

### Example

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// pretend we've connected both to a server at this point
//connect(s1, ...)...
//connect(s2, ...)...
```

```
// set up the array of file descriptors.
//
// in this example, we want to know when there's normal or out-of-band
// data ready to be recv()'d...

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band

ufds[1] = s2;
ufds[1].events = POLLIN; // check for just normal data

// wait for events on the sockets, 3.5 second timeout
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // error occurred in poll()
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 3.5 seconds.\n");
} else {
    // check for events on s1:
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0); // receive normal data
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band data
    }

    // check for events on s2:
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

### See Also

**`select()`**

## 9.18. `recv()`, `recvfrom()`

Receive data on a socket

### Prototypes

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

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, socklen_t *fromlen);
```

### Description

Once you have a socket up and connected, you can read incoming data from the remote side using the **`recv()`** (for TCP `SOCK_STREAM` sockets) and **`recvfrom()`** (for UDP `SOCK_DGRAM` sockets).

Both functions take the socket descriptor *`s`*, a pointer to the buffer *`buf`*, the size (in bytes) of the buffer *`len`*, and a set of *`flags`* that control how the functions work.

Additionally, the **`recvfrom()`** takes a `struct sockaddr*`, *`from`* that will tell you where the data came from, and will fill in *`fromlen`* with the size of `struct sockaddr`. (You must also initialize *`fromlen`* to be the size of *`from`* or `struct sockaddr`.)

So what wondrous flags can you pass into this function? Here are some of them, but you should check your local man pages for more information and what is actually supported on your system. You bitwise-or these together, or just set *`flags`* to `0` if you want it to be a regular vanilla **`recv()`**.

| | |
|---|---|
| `MSG_OOB` | Receive Out of Band data. This is how to get data that has been sent to you with the `MSG_OOB` flag in **`send()`**. As the receiving side, you will have had signal `SIGURG` raised telling you there is urgent data. In your handler for that signal, you could call **`recv()`** with this `MSG_OOB` flag. |
| `MSG_PEEK` | If you want to call **`recv()`** “just for pretend”, you can call it with this flag. This will tell you what's waiting in the buffer for when you call **`recv()`** “for real” (i.e. *without* the `MSG_PEEK` flag. It's like a sneak preview into the next **`recv()`** call. |
| `MSG_WAITALL` | Tell **`recv()`** to not return until all the data you specified in the *`len`* parameter. It will ignore your wishes in extreme circumstances, however, like if a signal interrupts the call or if some error occurs or if the remote side closes the connection, etc. Don't be mad with it. |

When you call **`recv()`**, it will block until there is some data to read. If you want to not block, set the socket to non-blocking or check with **`select()`** or **`poll()`** to see if there is incoming data before calling **`recv()`** or **`recvfrom()`**.

### Return Value

Returns the number of bytes actually received (which might be less than you requested in the *`len`* parameter), or `-1` on error (and **`errno`** will be set accordingly.)

If the remote side has closed the connection, **`recv()`** will return `0`. This is the normal method for determining if the remote side has closed the connection. Normality is good, rebel!

### Example

```
// stream sockets and recv()

struct addrinfo hints, *res;
int sockfd;
char buf[512];
int byte_count;

// get host info, make socket, and connect it
memset(&hints, 0, sizeof hints);
```

```
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.example.com", "3490", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
connect(sockfd, res->ai_addr, res->ai_addrlen);

// all right!  now that we're connected, we can receive some data!
byte_count = recv(sockfd, buf, sizeof buf, 0);
printf("recv()'d %d bytes of data in buf\n", byte_count);
```

```
// datagram sockets and recvfrom()

struct addrinfo hints, *res;
int sockfd;
int byte_count;
socklen_t fromlen;
struct sockaddr_storage addr;
char buf[512];
char ipstr[INET6_ADDRSTRLEN];

// get host info, make socket, bind it to port 4950
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, "4950", &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);

// no need to accept(), just recvfrom():

fromlen = sizeof addr;
byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen);

printf("recv()'d %d bytes of data in buf\n", byte_count);
printf("from IP address %s\n",
    inet_ntop(addr.ss_family,
        addr.ss_family == AF_INET?
            ((struct sockadd_in *)&addr)->sin_addr:
            ((struct sockadd_in6 *)&addr)->sin6_addr,
        ipstr, sizeof ipstr);
```

## See Also

**`send()`**, **`sendto()`**, **`select()`**, **`poll()`**, Blocking

## 9.19. select()

Check if sockets descriptors are ready to read/write

### Prototypes

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

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

### Description

The **select()** function gives you a way to simultaneously check multiple sockets to see if they have data waiting to be **recv()**d, or if you can **send()** data to them without blocking, or if some exception has occurred.

You populate your sets of socket descriptors using the macros, like **FD_SET()**, above. Once you have the set, you pass it into the function as one of the following parameters: *readfds* if you want to know when any of the sockets in the set is ready to **recv()** data, *writefds* if any of the sockets is ready to **send()** data to, and/or *exceptfds* if you need to know when an exception (error) occurs on any of the sockets. Any or all of these parameters can be NULL if you're not interested in those types of events. After **select()** returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions.

The first parameter, *n* is the highest-numbered socket descriptor (they're just ints, remember?) plus one.

Lastly, the struct timeval, *timeout*, at the end—this lets you tell **select()** how long to check these sets for. It'll return after the timeout, or when an event occurs, whichever is first. The struct timeval has two fields: *tv_sec* is the number of seconds, to which is added *tv_usec*, the number of microseconds (1,000,000 microseconds in a second.)

The helper macros do the following:

| | |
|---|---|
| **FD_SET(int fd, fd_set *set);** | Add *fd* to the *set*. |
| **FD_CLR(int fd, fd_set *set);** | Remove *fd* from the *set*. |
| **FD_ISSET(int fd, fd_set *set);** | Return true if *fd* is in the *set*. |
| **FD_ZERO(fd_set *set);** | Clear all entries from the *set*. |

### Return Value

Returns the number of descriptors in the set on success, 0 if the timeout was reached, or -1 on error (and **errno** will be set accordingly.) Also, the sets are modified to show which sockets are ready.

### Example

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

// pretend we've connected both to a server at this point
//s1 = socket(...);
//s2 = socket(...);
//connect(s1, ...)...
//connect(s2, ...)...

// clear the set ahead of time
FD_ZERO(&readfds);
```

```
// add our descriptors to the set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// since we got s2 second, it's the "greater", so we use that for
// the n param in select()
n = s2 + 1;

// wait until either socket has data ready to be recv()d (timeout 10.5 secs)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // error occurred in select()
} else if (rv == 0) {
    printf("Timeout occurred!  No data after 10.5 seconds.\n");
} else {
    // one or both of the descriptors have data
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

### See Also

**`poll()`**

## 9.20. `setsockopt()`, `getsockopt()`

Set various options for a socket

### Prototypes

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

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
```

### Description

Sockets are fairly configurable beasts. In fact, they are so configurable, I'm not even going to cover it all here. It's probably system-dependent anyway. But I will talk about the basics.

Obviously, these functions get and set certain options on a socket. On a Linux box, all the socket information is in the man page for socket in section 7. (Type: "**man 7 socket**" to get all these goodies.)

As for parameters, *`s`* is the socket you're talking about, level should be set to `SOL_SOCKET`. Then you set the *`optname`* to the name you're interested in. Again, see your man page for all the options, but here are some of the most fun ones:

| | |
|---|---|
| `SO_BINDTODEVICE` | Bind this socket to a symbolic device name like `eth0` instead of using **`bind()`** to bind it to an IP address. Type the command **ifconfig** under Unix to see the device names. |
| `SO_REUSEADDR` | Allows other sockets to **`bind()`** to this port, unless there is an active listening socket bound to the port already. This enables you to get around those "Address already in use" error messages when you try to restart your server after a crash. |
| `SO_BROADCAST` | Allows UDP datagram (`SOCK_DGRAM`) sockets to send and receive packets sent to and from the broadcast address. Does nothing —*NOTHING!!*—to TCP stream sockets! Hahaha! |

As for the parameter *`optval`*, it's usually a pointer to an `int` indicating the value in question. For booleans, zero is false, and non-zero is true. And that's an absolute fact, unless it's different on your system. If there is no parameter to be passed, *`optval`* can be `NULL`.

The final parameter, *`optlen`*, is filled out for you by **`getsockopt()`** and you have to specify it for **`setsockopt()`**, where it will probably be `sizeof(int)`.

**Warning**: on some systems (notably Sun and Windows), the option can be a `char` instead of an `int`, and is set to, for example, a character value of `'1'` instead of an `int` value of `1`. Again, check your own man pages for more info with "**man setsockopt**" and "**man 7 socket**"!

### Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

### Example

```
int optval;
int optlen;
char *optval2;

// set SO_REUSEADDR on a socket to true (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// bind a socket to a device name (might not work on all systems):
optval2 = "eth1"; // 4 bytes long, so 4, below:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);
```

```
// see if the SO_BROADCAST flag is set:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST enabled on s3!\n");
}
```

## See Also

**`fcntl()`**

## 9.21. `send()`, `sendto()`

Send data out over a socket

### Prototypes

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

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
               int flags, const struct sockaddr *to,
               socklen_t tolen);
```

### Description

These functions send data to a socket. Generally speaking, **`send()`** is used for TCP `SOCK_STREAM` connected sockets, and **`sendto()`** is used for UDP `SOCK_DGRAM` unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of **`sendto()`** define where the packet is going.

With both **`send()`** and **`sendto()`**, the parameter *`s`* is the socket, *`buf`* is a pointer to the data you want to send, *`len`* is the number of bytes you want to send, and *`flags`* allows you to specify more information about how the data is to be sent. Set *`flags`* to zero if you want it to be "normal" data. Here are some of the commonly used flags, but check your local **`send()`** man pages for more details:

| | |
|---|---|
| `MSG_OOB` | Send as "out of band" data. TCP supports this, and it's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal `SIGURG` and it can then receive this data without first receiving all the rest of the normal data in the queue. |
| `MSG_DONTROUTE` | Don't send this data over a router, just keep it local. |
| `MSG_DONTWAIT` | If **`send()`** would block because outbound traffic is clogged, have it return `EAGAIN`. This is like a "enable non-blocking just for this send." See the section on blocking for more details. |
| `MSG_NOSIGNAL` | If you **`send()`** to a remote host which is no longer **`recv()`**ing, you'll typically get the signal `SIGPIPE`. Adding this flag prevents that signal from being raised. |

### Return Value

Returns the number of bytes actually sent, or `-1` on error (and **`errno`** will be set accordingly.) Note that the number of bytes actually sent might be less than the number you asked it to send! See the section on handling partial **`send()`**s for a helper function to get around this.

Also, if the socket has been closed by either side, the process calling **`send()`** will get the signal `SIGPIPE`. (Unless **`send()`** was called with the `MSG_NOSIGNAL` flag.)

### Example

```
int spatula_count = 3490;
char *secret_message = "The Cheese is in The Toaster";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// first with TCP stream sockets:

// assume sockets are made and connected
//stream_socket = socket(...
//connect(stream_socket, ...
// convert to network byte order
temp = htonl(spatula_count);
// send data normally:
send(stream_socket, &temp, sizeof temp, 0);

// send secret message out of band:
send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB);

// now with UDP datagram sockets:
//getaddrinfo(...
//dest = ...  // assume "dest" holds the address of the destination
//dgram_socket = socket(...

// send secret message normally:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
       (struct sockaddr*)&dest, sizeof dest);
```

### See Also

**`recv()`**, **`recvfrom()`**

## 9.22. `shutdown()`

Stop further sends and receives on a socket

### Prototypes

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

int shutdown(int s, int how);
```

### Description

That's it! I've had it! No more **`send()`**s are allowed on this socket, but I still want to **`recv()`** data on it! Or vice-versa! How can I do this?

When you **`close()`** a socket descriptor, it closes both sides of the socket for reading and writing, and frees the socket descriptor. If you just want to close one side or the other, you can use this **`shutdown()`** call.

As for parameters, *`s`* is obviously the socket you want to perform this action on, and what action that is can be specified with the *`how`* parameter. How can be `SHUT_RD` to prevent further **`recv()`**s, `SHUT_WR` to prohibit further **`send()`**s, or `SHUT_RDWR` to do both.

Note that **`shutdown()`** doesn't free up the socket descriptor, so you still have to eventually **`close()`** the socket even if it has been fully shut down.

This is a rarely used system call.

### Return Value

Returns zero on success, or `-1` on error (and **`errno`** will be set accordingly.)

### Example

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...do some send()s and stuff in here...

// and now that we're done, don't allow any more sends()s:
shutdown(s, SHUT_WR);
```

### See Also

**`close()`**

## 9.23. `socket()`

Allocate a socket descriptor

### Prototypes

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

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

### Description

Returns a new socket descriptor that you can use to do sockety things with. This is generally the first call in the whopping process of writing a socket program, and you can use the result for subsequent calls to **`listen()`**, `bind()`, `accept()`, or a variety of other functions.

In usual usage, you get the values for these parameters from a call to **`getaddrinfo()`**, as shown in the example below. But you can fill them in by hand if you really want to.

| | |
|---|---|
| *`domain`* | *`domain`* describes what kind of socket you're interested in. This can, believe me, be a wide variety of things, but since this is a socket guide, it's going to be `PF_INET` for IPv4, and `PF_INET6` for IPv6. |
| *`type`* | Also, the *`type`* parameter can be a number of things, but you'll probably be setting it to either `SOCK_STREAM` for reliable TCP sockets (**`send()`**, **`recv()`**) or `SOCK_DGRAM` for unreliable fast UDP sockets (**`sendto()`**, **`recvfrom()`**.)<br><br>(Another interesting socket type is `SOCK_RAW` which can be used to construct packets by hand. It's pretty cool.) |
| *`protocol`* | Finally, the *`protocol`* parameter tells which protocol to use with a certain socket type. Like I've already said, for instance, `SOCK_STREAM` uses TCP. Fortunately for you, when using `SOCK_STREAM` or `SOCK_DGRAM`, you can just set the protocol to 0, and it'll use the proper protocol automatically. Otherwise, you can use **`getprotobyname()`** to look up the proper protocol number. |

### Return Value

The new socket descriptor to be used in subsequent calls, or `-1` on error (and **`errno`** will be set accordingly.)

### Example

```
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;     // AF_INET, AF_INET6, or AF_UNSPEC
hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM or SOCK_DGRAM

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket using the information gleaned from getaddrinfo():
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

### See Also

**`accept()`**, **`bind()`**, **`getaddrinfo()`**, **`listen()`**

## 9.24. `struct sockaddr` and pals

Structures for handling internet addresses

### Prototypes

```
include <netinet/in.h>

// All pointers to socket address structures are often cast to pointers
// to this type before use in various functions and system calls:

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};


// IPv4 AF_INET sockets:

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};


// IPv6 AF_INET6 sockets:

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // load with inet_pton()
};


// General socket address holding structure, big enough to hold either
// struct sockaddr_in or struct sockaddr_in6 data:

struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

### Description

These are the basic structures for all syscalls and functions that deal with internet addresses. Often you'll use **`getaddinfo()`** to fill these structures out, and then will read them when you have to.

In memory, the `struct sockaddr_in` and `struct sockaddr_in6` share the same beginning structure as `struct sockaddr`, and you can freely cast the pointer of one type to the other without any harm, except the possible end of the universe.

Just kidding on that end-of-the-universe thing...if the universe does end when you cast a `struct sockaddr_in*` to a `struct sockaddr*`, I promise you it's pure coincidence and you shouldn't even worry about it.

So, with that in mind, remember that whenever a function says it takes a `struct sockaddr*` you can cast your `struct sockaddr_in*`, `struct sockaddr_in6*`, or `struct sockadd_storage*` to that type with ease and safety.

`struct sockaddr_in` is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (`AF_INET`), a port in *`sin_port`*, and an IPv4 address in *`sin_addr`*.

There's also this `sin_zero` field in `struct sockaddr_in` which some people claim must be set to zero. Other people don't claim anything about it (the Linux documentation doesn't even mention it at all), and setting it to zero doesn't seem to be actually necessary. So, if you feel like it, set it to zero using **`memset()`**.

Now, that `struct in_addr` is a weird beast on different systems. Sometimes it's a crazy `union` with all kinds of `#define`s and other nonsense. But what you should do is only use the *`s_addr`* field in this structure, because many systems only implement that one.

`struct sockadd_in6` and `struct in6_addr` are very similar, except they're used for IPv6.

`struct sockaddr_storage` is a struct you can pass to **`accept()`** or **`recvfrom()`** when you're trying to write IP version-agnostic code and you don't know if the new address is going to be IPv4 or IPv6. The `struct sockaddr_storage` structure is large enough to hold both types, unlike the original small `struct sockaddr`.

### Example

```
// IPv4:

struct sockaddr_in ip4addr;
int s;

ip4addr.sin_family = AF_INET;
ip4addr.sin_port = htons(3490);
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr);
```

```
// IPv6:

struct sockaddr_in6 ip6addr;
int s;

ip6addr.sin6_family = AF_INET6;
ip6addr.sin6_port = htons(4950);
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);

s = socket(PF_INET6, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr);
```

### See Also

**`accept()`**, **`bind()`**, **`connect()`**, **`inet_aton()`**, **`inet_ntoa()`**