Chapter 10 Socket Programming

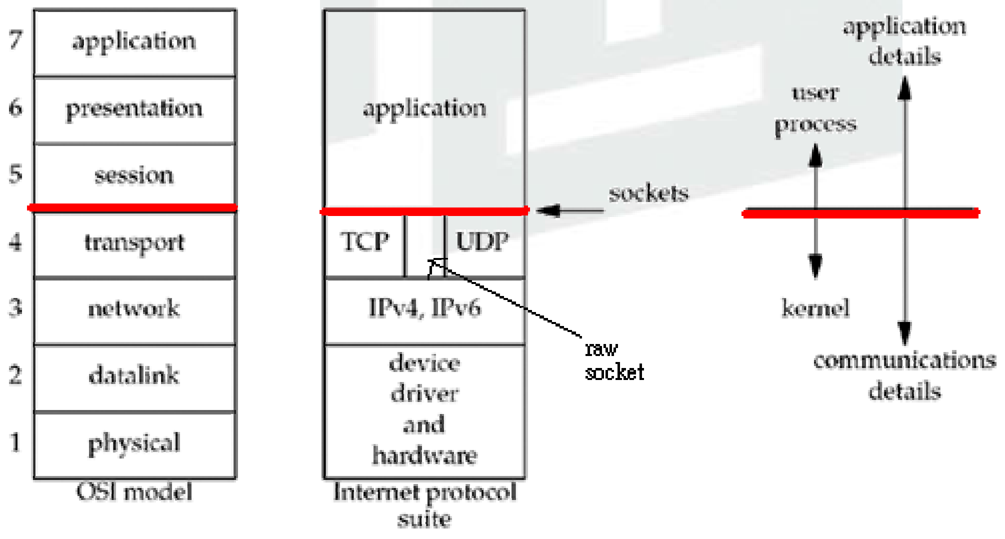
- In computer networking, an **Internet socket** or **network socket** is an endpoint of a bidirectional inter-process communication flow across an Internet Protocol-based computer network, such as the Internet.
- Socket is the application programming interface (API) for accessing a variety of underlying network protocols
- Socket is a network programming interface and not a protocol.
- Berkeley (BSD) Sockets implementation on UNIX platforms, which is capable of accessing multiple networking protocols.
- The Winsock interface inherits a great deal from BSD socket.

Network reference models and layering.

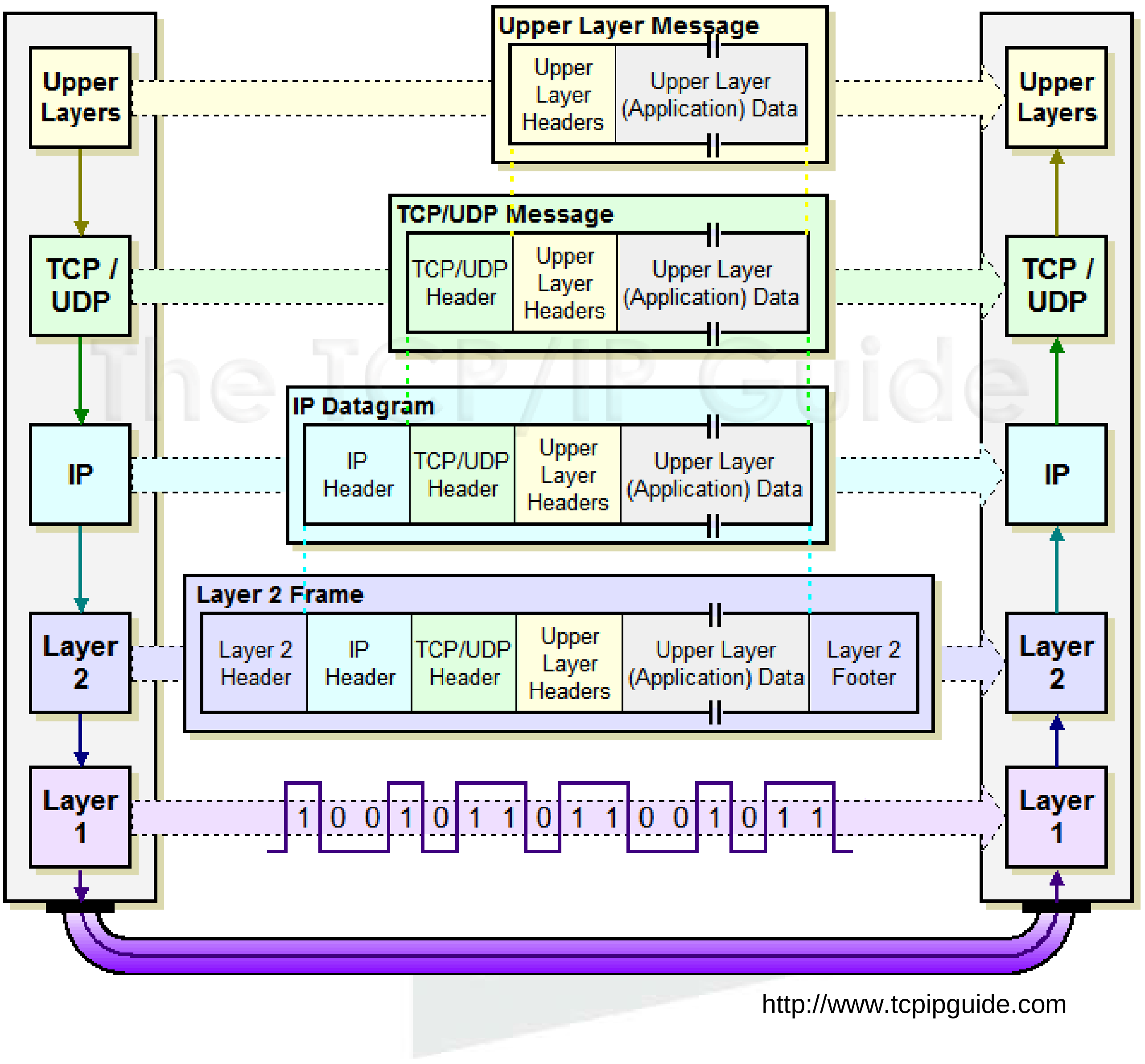
A **socket address** is the combination of an IP address (the location of the computer) and a port

- The socket **161.25.19.8:9734** refers to port **9734** on host **161.25.19.8**

Layers in OSI model and Internet protocol suite

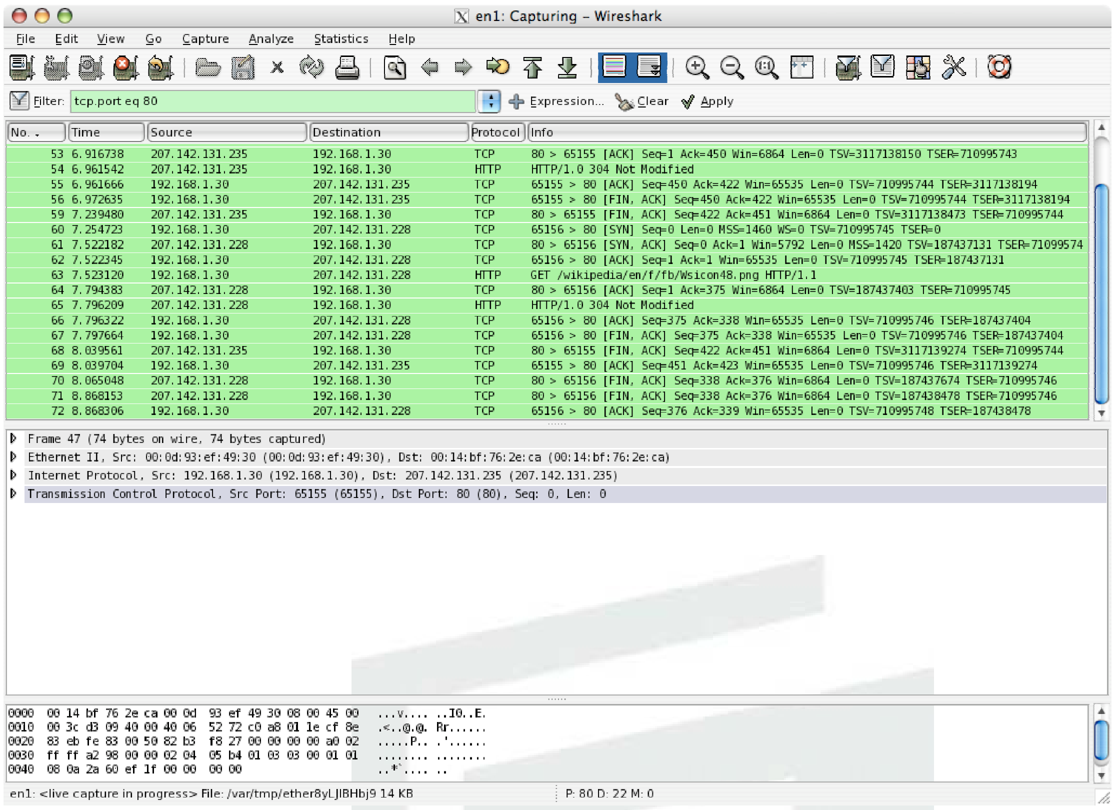


Client and server on the same Ethernet communicating using TCP



Wireshark - packet sniffer

Wireshark is a free packet sniffer computer application. It is used for network troubleshooting, analysis, software and communications protocol development, and education


en1: Capturing – Wireshark
File Edit View Go Capture Analyze Statistics Help
Expression... Clear Apply

Filter: tcp.port eq 80

```
No. .  Time      Source           Destination      Protocol Info
    53 6.916738  207.142.131.235  192.168.1.30     TCP      80 > 65155 [ACK] Seq=1 Ack=450 Win=6864 Len=0 TSV=3117138150 TSER=710995743
    54 6.961542  207.142.131.235  192.168.1.30     HTTP     HTTP/1.0 304 Not Modified
    55 6.961666  192.168.1.30     207.142.131.235  TCP      65155 > 80 [ACK] Seq=450 Ack=422 Win=65535 Len=0 TSV=710995744 TSER=3117138194
    56 6.972635  192.168.1.30     207.142.131.235  TCP      65155 > 80 [FIN, ACK] Seq=450 Ack=422 Win=65535 Len=0 TSV=710995744 TSER=3117138194
    59 7.239480  207.142.131.235  192.168.1.30     TCP      80 > 65155 [FIN, ACK] Seq=422 Ack=451 Win=6864 Len=0 TSV=3117138473 TSER=710995744
    60 7.254723  192.168.1.30     207.142.131.228  TCP      65156 > 80 [SYN] Seq=0 Len=0 MSS=1460 WS=0 TSV=710995745 TSER=0
    61 7.522182  207.142.131.228  192.168.1.30     TCP      80 > 65156 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1420 TSV=187437131 TSER=71099574
    62 7.522345  192.168.1.30     207.142.131.228  TCP      65156 > 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0 TSV=710995745 TSER=187437131
    63 7.523120  192.168.1.30     207.142.131.228  HTTP     GET /wikipedia/en/f/fb/Wsicon48.png HTTP/1.1
    64 7.794383  207.142.131.228  192.168.1.30     TCP      80 > 65156 [ACK] Seq=1 Ack=375 Win=6864 Len=0 TSV=187437403 TSER=710995745
    65 7.796209  207.142.131.228  192.168.1.30     HTTP     HTTP/1.0 304 Not Modified
    66 7.796322  192.168.1.30     207.142.131.228  TCP      65156 > 80 [ACK] Seq=375 Ack=338 Win=65535 Len=0 TSV=710995746 TSER=187437404
    67 7.797664  192.168.1.30     207.142.131.228  TCP      65156 > 80 [FIN, ACK] Seq=375 Ack=338 Win=65535 Len=0 TSV=710995746 TSER=187437404
    68 8.039561  207.142.131.235  192.168.1.30     TCP      80 > 65155 [FIN, ACK] Seq=422 Ack=451 Win=6864 Len=0 TSV=3117139274 TSER=710995744
    69 8.039704  192.168.1.30     207.142.131.235  TCP      65155 > 80 [ACK] Seq=451 Ack=423 Win=65535 Len=0 TSV=710995746 TSER=3117139274
    70 8.065048  207.142.131.228  192.168.1.30     TCP      80 > 65156 [FIN, ACK] Seq=338 Ack=376 Win=6864 Len=0 TSV=187437674 TSER=710995746
    71 8.868153  207.142.131.228  192.168.1.30     TCP      80 > 65156 [FIN, ACK] Seq=338 Ack=376 Win=6864 Len=0 TSV=187438478 TSER=710995746
    72 8.868306  192.168.1.30     207.142.131.228  TCP      65156 > 80 [ACK] Seq=376 Ack=339 Win=65535 Len=0 TSV=710995748 TSER=187438478
```

```
▷ Frame 47 (74 bytes on wire, 74 bytes captured)
▷ Ethernet II, Src: 00:0d:93:ef:49:30 (00:0d:93:ef:49:30), Dst: 00:14:bf:76:2e:ca (00:14:bf:76:2e:ca)
▷ Internet Protocol, Src: 192.168.1.30 (192.168.1.30), Dst: 207.142.131.235 (207.142.131.235)
▷ Transmission Control Protocol, Src Port: 65155 (65155), Dst Port: 80 (80), Seq: 0, Len: 0
```

```
0000  00 14 bf 76 2e ca 00 0d  93 ef 49 30 08 00 45 00   ...v.... ..I0..E.
0010  00 3c d3 09 40 00 40 06  52 72 c0 a8 01 1e cf 8e   .<..@.@. Rr......
0020  83 eb fe 83 00 50 82 b3  f8 27 00 00 00 00 a0 02   .....P.. .'......
0030  ff ff a2 98 00 00 02 04  05 b4 01 03 03 00 01 01   ........ ........
0040  08 0a 2a 60 ef 1f 00 00  00 00                     ..*`.... ..
```

en1: <live capture in progress> File: /var/tmp/ether8yLJlBHbj9 14 KB   P: 80 D: 22 M: 0

INSTALLING WIRESHARK

Fedora

```
yum install wireshark-gnome
```

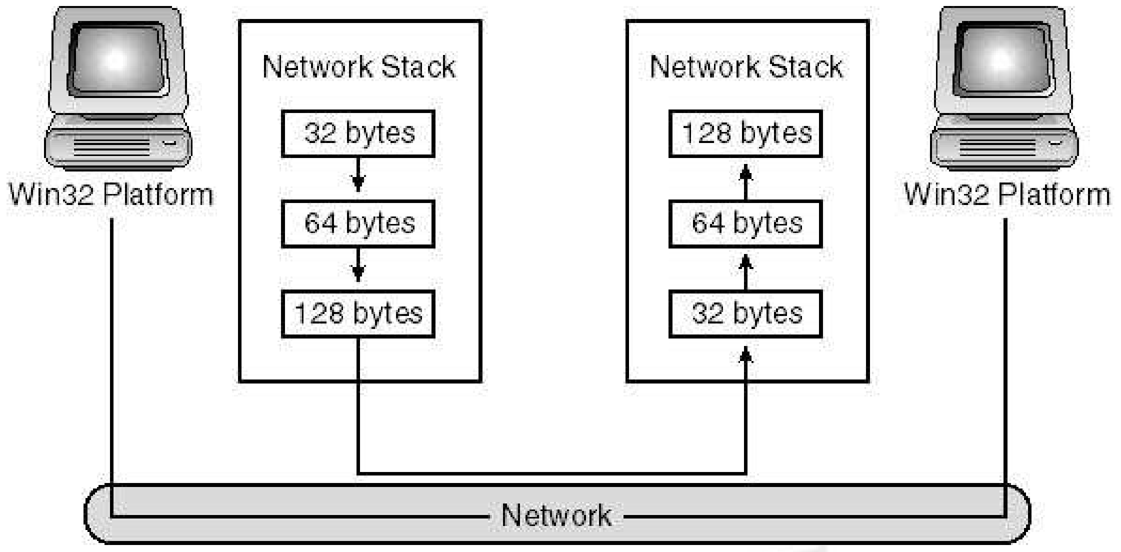
Ubuntu:

```
sudo apt-get install wireshark
```

Transport Protocol Characteristics

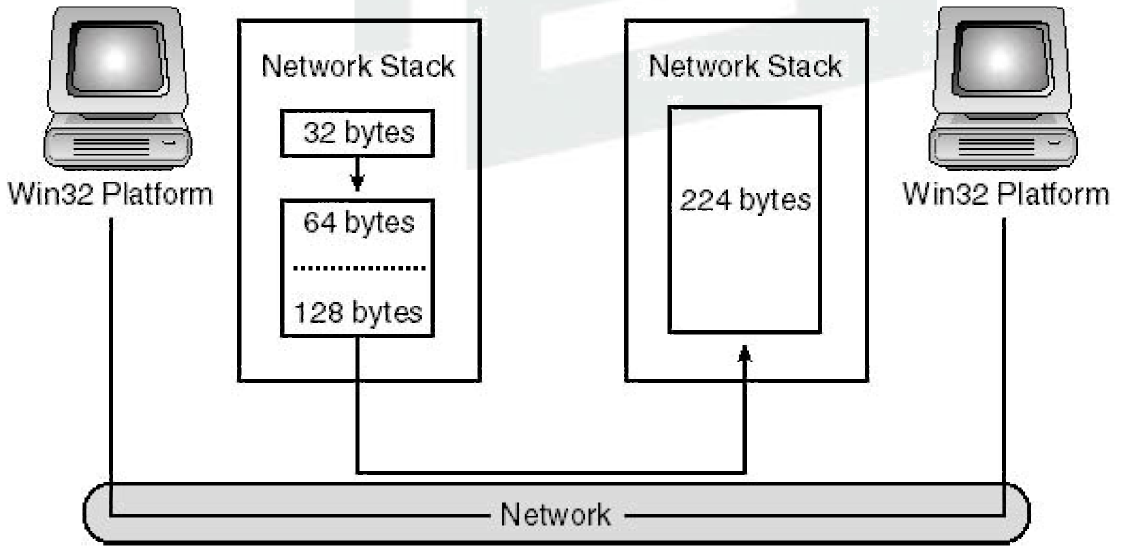
Message-based protocol (datagram)

- For each discrete write command it transmits those and only those bytes as a single message on the network.
- This also means that when the receiver requests data, the data returned is a discrete message written by the sender. The receiver will not get more than one message



Stream-based protocol

- Stream service is defined as the transmitting of data in a continual process: the receiver reads as much data as is available with no respect to message boundaries.
- For the sender, this means that the system is allowed to break up the original message into pieces or lump several messages together to form a bigger packet of data.
- On the receiving end, the network stack reads data as it arrives and buffers it for the process. When the process requests an amount of data, the system returns as much data as possible without overflowing the buffer supplied by the client call

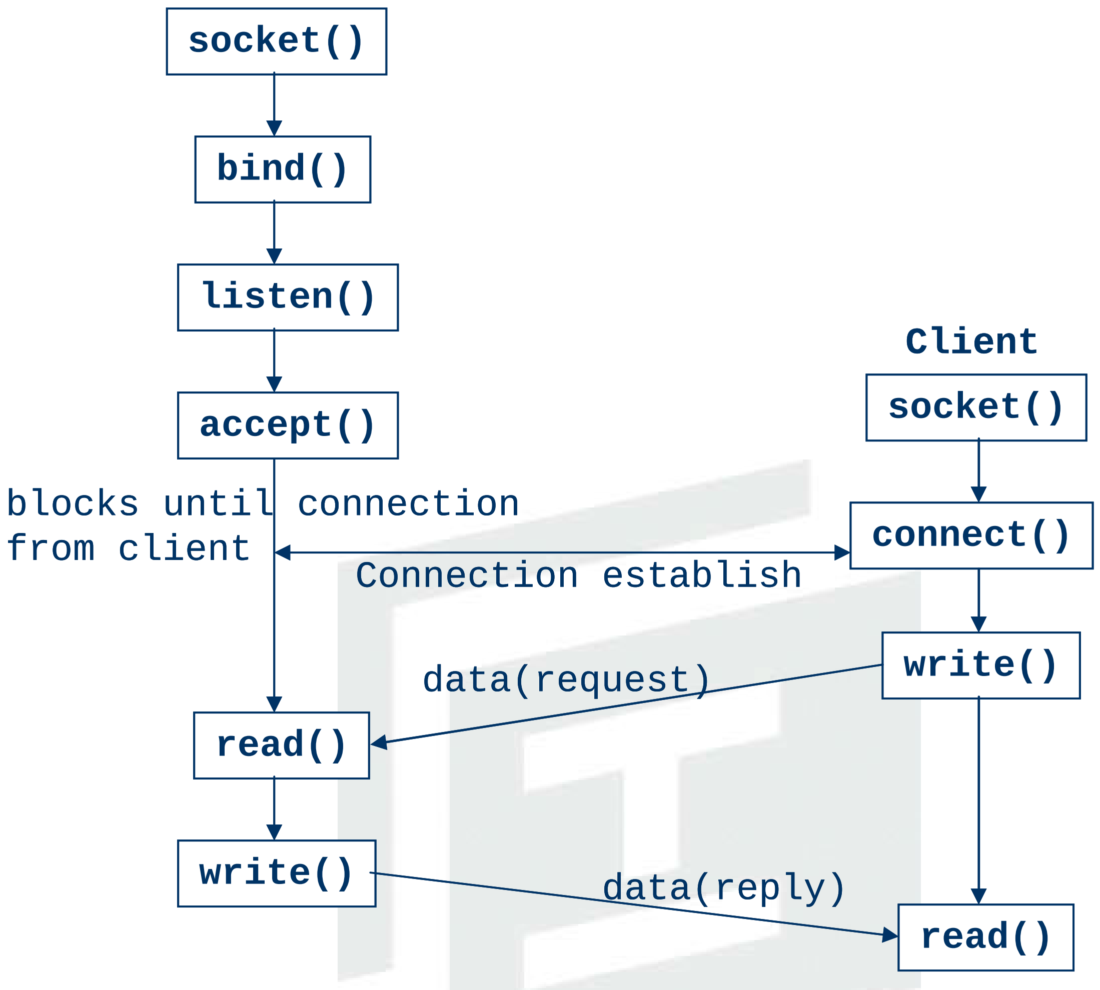


API QuickView

These are examples of functions or methods typically provided by the API library:

- **`socket()`** creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- **`bind()`** is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- **`listen()`** is used on the server side, and causes a bound TCP socket to enter listening state.
- **`connect()`** is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
- **`accept()`** is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- **`send()`** and **`recv()`**, or **`write()`** and **`read()`**, or **`recvfrom()`** and **`sendto()`**, are used for sending and receiving data to/from a remote socket.
- **`close()`** causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
- **`gethostbyname()`** and **`gethostbyaddr()`** are used to resolve host names and addresses.
- **`select()`** is used to prune a provided list of sockets for those that are ready to read, ready to write or have errors
- **`poll()`** is used to check on the state of a socket. The socket can be tested to see if it can be written to, read from or has errors

Socket System Calls for Connection-Oriented Protocols

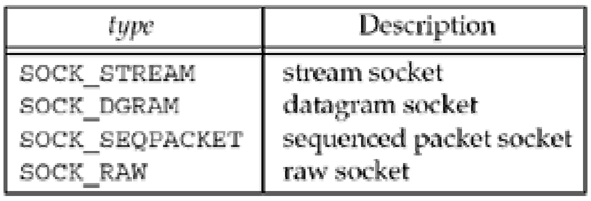


Socket Attributes

Family: specifies the protocol family , which is often referred to as *domain* instead of family.

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

socket type



type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

Stream Sockets

- Stream sockets (in some ways similar to standard input/output streams) provide a connection that is a sequenced and *reliable two-way byte stream.*
- Thus, data sent is guaranteed not to be lost, *duplicated*, or *reordered* without an indication that an error has occurred.
- Large messages are fragmented, transmitted, and reassembled. This is like a file stream, as it accepts large amounts of data and writes it to the low-level disk in smaller blocks.
- Stream sockets, specified by the type `SOCK_STREAM`, are implemented in the `AF_INET` domain by TCP/IP connections. They are also the usual type in the `AF_UNIX` domain.

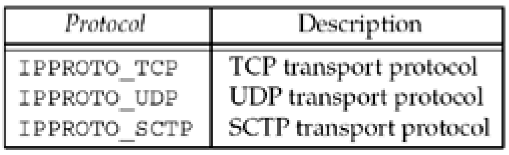
Datagram Sockets

- It doesn't establish and maintain a connection.
- There is also a limit on the size of a datagram that can be sent. It's transmitted as a single network message that may get lost, duplicated, or arrives out of sequence—ahead of datagrams sent after it.
- Datagram sockets are implemented in the `AF_INET` domain by UDP/IP connections and provide an unsequenced, unreliable service.
- However, they are relatively inexpensive in terms of resources, since network connections need not be maintained. They're fast because there is no associated connection setup time. UDP stands for User Datagram Protocol.

Raw Sockets

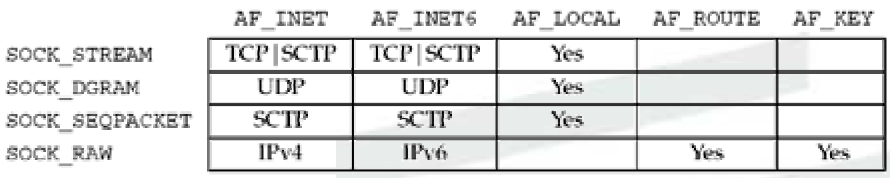
- A raw socket is a socket that allows access to the underlying transport protocol
- For Example , ICMP packet, which does not provide any data transfer facilities,
- Using raw sockets requires substantial knowledge of the underlying protocol structure

`protocol` : the specific protocol type found in below, or 0 to select the system's default for the given combination of family and type.



Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Not all combinations of socket family and type are valid. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.



	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP|SCTP	TCP|SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Socket Address Structures

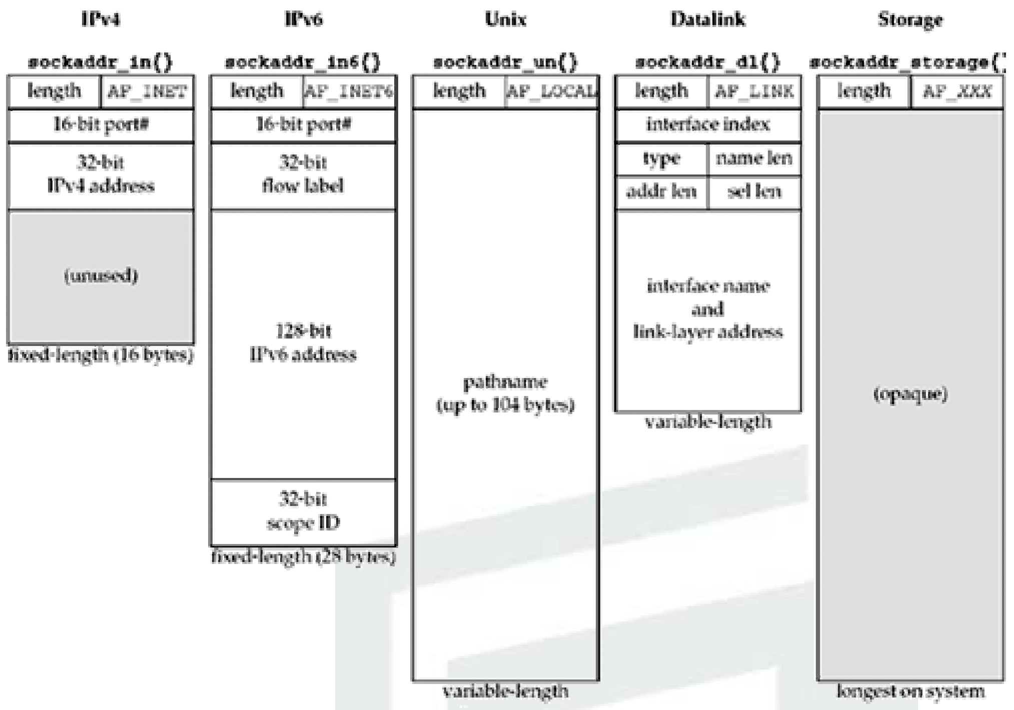
The Internet (IPv4) socket address AF_INET structure: sockaddr_in.

```
struct in_addr {
   in_addr_t   s_addr;          /* 32-bit IPv4 address */
                                /* network byte ordered */
};

struct sockaddr_in {
   uint8_t        sin_len;  /* length of structure (16) */
   sa_family_t    sin_family; /* AF_INET */
   in_port_t       sin_port; /* 16-bit TCP or UDP port */
                                /* network byte ordered */
   struct in_addr  sin_addr;   /* 32-bit IPv4 address */
                                /* network byte ordered */
   char           sin_zero[8];  /* unused */
};
```

The generic socket address structure: sockaddr

```
   struct sockaddr {
     uint8_t       sa_len;
     sa_family_t   sa_family; /*AF_xxx value */
     char  sa_data[14];  /* protocol-specific address */
   };
```

Comparison of various socket address structures



Creating a Socket

The `socket` system call creates a socket and returns a descriptor that can be used for accessing the socket.

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

- The protocol used for communication is usually determined by the socket type and domain. There is normally no choice. The `protocol` parameter is used where there is a choice. `0` selects the default protocol,
- The `socket` system call returns a descriptor that is in many ways similar to a low-level file descriptor. When the socket has been connected to another end-point socket, you may use the `read` and `write` system calls with the descriptor to send and receive data on the socket.

- The `close` system call is used to end a socket connection.

Naming a Socket

To make a socket (as created by a call to `socket`) available for use by other processes, a server program needs to give the socket a name.

- `AF_UNIX` sockets are associated with a file system pathname
- `AF_INET` sockets are associated with an IP port number.

```
#include <sys/socket.h>
int bind(int socket, const struct sockaddr *address,
          size_t address_len);
```

- The `bind` system call assigns the address specified in the parameter, `address`, to the unnamed socket associated with the file descriptor `socket`.
- The length of the address structure is passed as `address_len`.
- The length and format of the address depend on the address family. A particular address structure pointer will need to be cast to the generic address type `(struct sockaddr *)` in the call to `bind`.
- On successful completion, `bind` returns `0`. If it fails, it returns `-1` and sets `errno` to one of the following.

Creating a Socket Queue

To accept incoming connections on a socket, a server program must create a queue to

store pending requests. It does this using the `listen` system call.

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

- A Linux system may limit the maximum number of pending cossnnections that may be held in a queue. Subject to this maximum, `listen` sets the queue length to `backlog`.

- Incoming connections up to this queue length are held pending on the socket; further connections will be refused and the client's connection will fail. This mechanism is provided by `listen` to allow incoming connections to be held pending while a server program is busy dealing with a previous client. A value of `5` for `backlog` is very common.
- The `listen` function will return `0` on success or `-1` on error. Errors include `EBADF`, `EINVAL`, and `ENOTSOCK`, as for the `bind` system call.

Accepting Connections

Once a server program has created and named a socket, it can wait for connections to be made to the socket by using the `accept` system call.

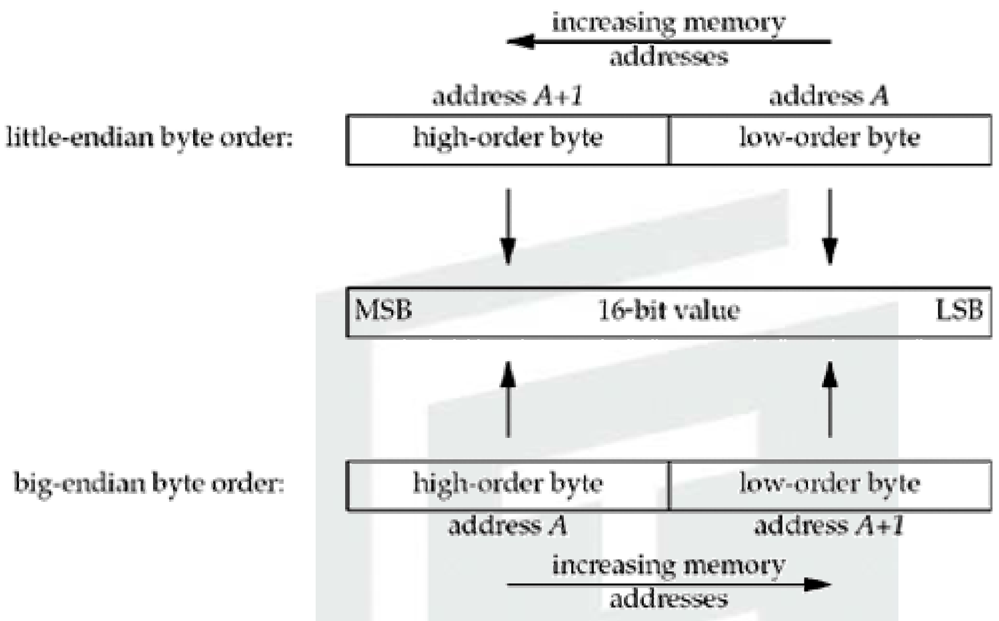
```
#include <sys/socket.h>
int accept(int socket, struct sockaddr *address,
         size_t *address_len);
```

- The `accept` system call returns when a client program attempts to connect to the socket specified by the parameter `socket`. The client is the first pending connection from that socket's queue.
- The `accept` function creates a new socket to communicate with the client and returns its descriptor. The new socket will have the same type as the server listen socket.
- The socket must have previously been named by a call to `bind` and had a connection queue allocated by `listen`.
- The address of the calling client will be placed in the `sockaddr` structure pointed to by `address`. A null pointer may be used here if the client address isn't of interest.
- The `address_len` parameter specifies the length of the client structure. If the client address is longer than this value, it will be truncated.
- Before calling `accept`, `address_len` must be set to the expected address length On return, `address_len` will be set to the actual length of the calling client's address structure.
- If there are no connections pending on the socket's queue, `accept` will block (so that the program won't continue) until a client makes a connection.

Byte ordering

Different computer processors represent numbers in *big-endian* and *little-endian* form, depending on how they are designed

Little-endian byte order and big-endian byte order for a 16-bit integer.



```
#include<stdio.h
int main(int argc, char **argv)
 {
     union {
        short   s;
        char    c[sizeof(short)];
     } un;

     un.s = 0x0102;
    printf("%s: ", CPU_VENDOR_OS);
    if (sizeof(short) == 2) {
       if (un.c[0] == 1 && un.c[1] == 2)
          printf("big-endian\n");
       else if (un.c[0] == 2 && un.c[1] == 1)
          printf("little-endian\n");
       else
          printf("unknown\n");
    } else
       printf("sizeof(short) = %d\n", sizeof(short));

    exit(0);
}
```

Internet networking standards specify that multibyte values must be represented in big-endian form (most significant byte to least significant byte), normally referred to as *network-byte* order.

Utility routines

Byte ordering routines

```
#include <sys/types.h>
#include <netinet/in.h>
 /* host-to-network, long integer */
u_long htonl(u_long hostlong);
/* host-to-network, short integer */
u_short htons(u_short hostshort);
/* network-to-host, long integer */
u_long ntohl(u_long netlong);
/* network-to-host, short integer */
u_short ntohs(u_short netshort);
```

Address conversion routines

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

Requesting Connections

```
int sockfd, newsockfd;
if ( (sockfd = socket ( … )) < 0)
   err_sys (“socket error”);
if (bind(sockfd, … ) < 0)
   err_sys (“bond error”);
if (listen(sockfd, 5) < 0)
   err_sys (“listen error”);
for ( ; ; ) {
   newsocfd = accept (sockfd, … );    /* blocking */
   if (newsockfd < 0)
      err_sys (“accept error”);
   if (fork() == 0) {      /* child */
      close (sockfd);
      new_task(newsockfd);    /* process request */
      exit();
   }
   close (newsockfd);     /* parent */
}
```

Network Information

by consulting network configuration files, such as `/etc/hosts`, or network information services, such as NIS (Network Information Services, formerly known as Yellow Pages) and DNS (Domain Name Service).

```
#include <netdb.h>
struct hostent *gethostbyaddr(const void *addr,
              size_t len, int type);
struct hostent *gethostbyname(const char *name);
struct hostent {
char *h_name; /* name of the host */
char **h_aliases; /* list of aliases (nicknames) */
int h_addrtype; /* address type */
int h_length; /* length in bytes of the address */
char **h_addr_list /* list of address (network order) */
};
```

Similarly, information concerning services and associated port numbers is available through some service information function.

```
#include <netdb.h>
struct servent *getservbyname(const char *name,
            const char *proto);
struct servent *getservbyport(int port, const char *proto);
```

The `proto` parameter specifies the protocol to be used to connect to the service, either `“tcp”` for `SOCK_STREAM` TCP connections or `“udp”` for `SOCK_DGRAM` UDP datagrams.

The structure `servent` contains at least these members:

```
struct servent {
char *s_name; /* name of the service */
char **s_aliases;/* list of aliases (alternative names) */
int s_port; /* The IP port number */
char *s_proto; /* The service type, usually “tcp” or “udp” */
};
```

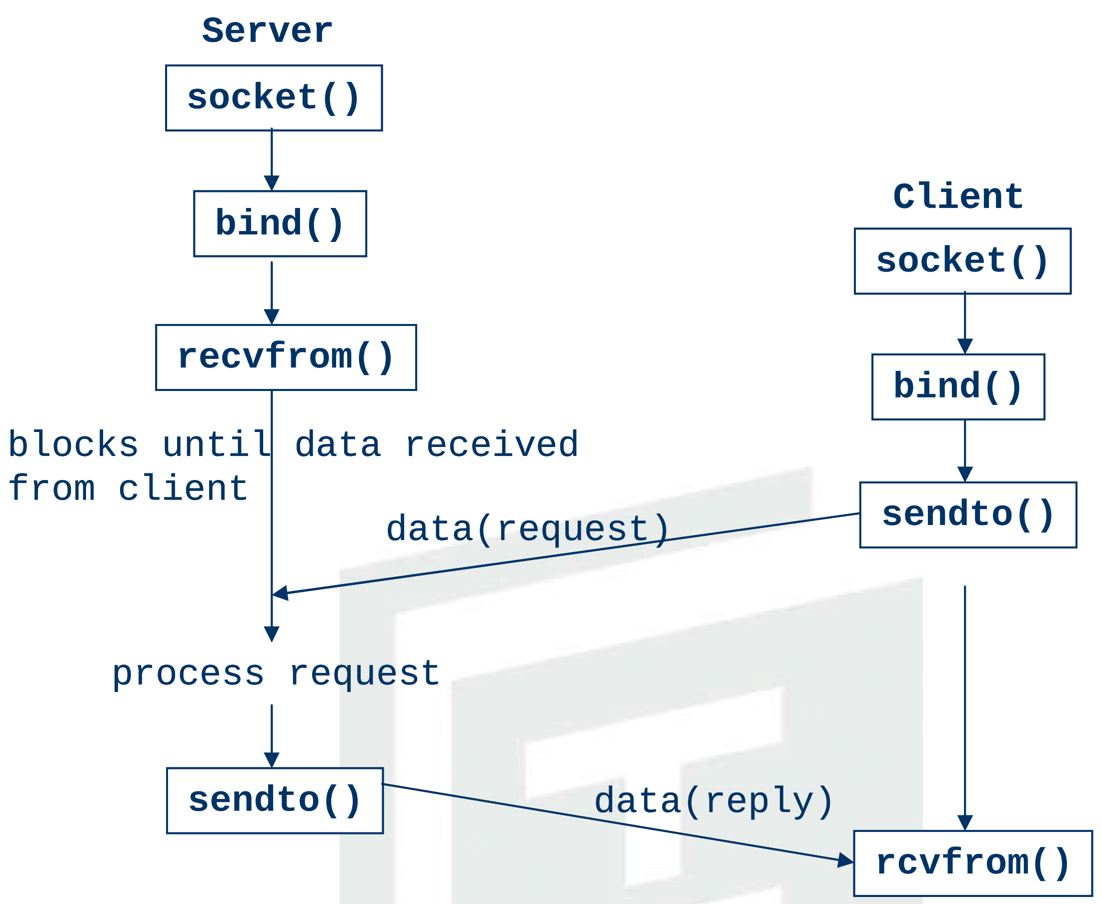
Example: ch7/7-5.c

```
    struct hostent *hostinfo;
    char *host, **names, **addrs;
    hostinfo = gethostbyname(“tw.yahoo.com”);
    if(!hostinfo) {
       fprintf(stderr, "cannot get info for host: %s\n", host);
       exit(1);
    }
/*  Display the hostname and any aliases it may have.  */
    printf("results for host %s:\n", host);
    printf("Name: %s\n", hostinfo -> h_name);
    printf("Aliases:");
    names = hostinfo -> h_aliases;
    while(*names) {
       printf(" %s", *names);
       names++;
    }
    printf("\n");
/*  Warn and exit if the host in question isn't an IP host.  */
    if(hostinfo -> h_addrtype != AF_INET) {
       fprintf(stderr, "not an IP host!\n");
       exit(1);
    }

/*  Otherwise, display the IP address(es).  */
    addrs = hostinfo -> h_addr_list;
    while(*addrs) {
       printf(" %s", inet_ntoa(*(struct in_addr *)*addrs));
       addrs++;
    }
    printf("\n");
    exit(0);
}
```

```
/* Server*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
int main()
{   int server_sockfd, client_sockfd;
    int server_len, client_len;
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;
   int i=1;

/*  Create an unnamed socket for the server.  */
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
/*  Name the socket.  */
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(9734);
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
/*  Create a connection queue and wait for clients.  */
    listen(server_sockfd, 5);
    while(i) {
        char ch;
        printf("server waiting\n");
/* Accept a connection.  */
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);
/*  We can now read/write to client on client_sockfd.  */
        read(client_sockfd, &ch, 1);
        printf("char from client = %c\n", ch);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
       i-=1;
    }
        close(server_sockfd);
```

```
/* Client */
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
int main()
{
    int sockfd;
    int len;
    struct sockaddr_in address;
    int result;
    char ch = 'A';
/*  Create a socket for the client.  */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
/*  Name the socket, as agreed with the server.  */
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("192.168.1.23");
    address.sin_port = htons(9734);
    len = sizeof(address);
/*  Now connect our socket to the server's socket.  */
    result = connect(sockfd, (struct sockaddr *)&address, len);
    if(result == -1) {
        perror("oops: client2");
        exit(1);
    }
/*  We can now read/write via sockfd.  */
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

Socket System Calls for Connectionless Protocols



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

int main(int argc, char *argv[])
{
    char *host;
    int sockfd;
    int len, result;
    struct sockaddr_in address;
    struct hostent *hostinfo;
    struct servent *servinfo;
    char buffer[128];
    if(argc == 1)
        host = "localhost";
    else
        host = argv[1];
/*  Find the host address and report an error if none is found.  */
    hostinfo = gethostbyname(host);
    if(!hostinfo) {
        fprintf(stderr, "no host: %s\n", host);
        exit(1);
    }
/*  Check that the daytime service exists on the host.  */
    servinfo = getservbyname("daytime", "udp");
    if(!servinfo) {
        fprintf(stderr,"no daytime service\n");
        exit(1);
    }
     printf("daytime port is %d\n", ntohs(servinfo -> s_port));

/*  Create a UDP socket.  */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

/*  Construct the address for use with sendto/recvfrom...  */
    address.sin_family = AF_INET;
    address.sin_port = servinfo -> s_port;
    address.sin_addr = *(struct in_addr *)*hostinfo -> h_addr_list;
    len = sizeof(address);
    result = sendto(sockfd, buffer, 1, 0, (struct sockaddr *)&address, len);
    result = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)&address,
             &len);
    buffer[result] = '\0';
    printf("read %d bytes: %s", result, buffer);

    close(sockfd);
    exit(0);
}
```

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int s,
           const void *msg,
           int len, unsigned flags,
           const struct sockaddr *to,
           int tolen
           );
```

1. The first argument s is the socket number. You received this value from the socket(2) function.
2. Argument msg is a pointer to the buffer holding the datagram message that you wish to send.
3. Argument len is the length, in bytes, of the datagram that starts at the pointer given by msg.
4. The flags argument allows you to specify some option bits. In many cases, you will simply supply a value of zero.
5. The argument to is a pointer to a generic socket address that you have established. This is the address of the recipient of the datagram.
6. Argument tolen is the length of the address argument to.

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int s,
             void *buf,
             int len,
             unsigned flags,
             struct sockaddr *from,
             int *fromlen
             );
```

1. The socket s to receive the datagram from.
2. The buffer pointer buf to start receiving the datagram into.
3. The maximum length (len) in bytes of the receiving buffer buf.
4. Option flag bits flags.

5. The pointer to the receiving socket address buffer, which will receive the sender's address (pointer argument from).
6. The pointer to the maximum length (fromlen) in bytes of the receiving socket address buffer from. Note that the integer that this pointer points to must be initialized to the maximum size of the receiving address structure from, prior to calling the function.

Like any normal read() operation, the receiving buffer buf must be large enough to receive the incoming datagram. The maximum length is indicated to the function by the argument len.

The function returns the value -1 if there was an error, and you should consult the value of errno for the cause of the error. Otherwise, the function returns the number of bytes that were received into your receiving buffer buf. This will be the size of your datagram received.

Socket Options and Ioctls

Once a socket has been created, various attributes can be manipulated via socket options and ioctl commands to affect the behavior of the socket.

```
Example 1: IP address
#include <sys/ioctl.h>
{
       int  fd;
       struct ifreq ifr;
        fd = socket(AF_INET, SOCK_DGRAM, 0);
        strcpy(ifr.ifr_name, “eth0”);
       if (ioctl( fd, SIOCGIFADDR, &ifr) < 0)
              perror("ioctl");
        close(fd);
       snprintf(buf,size,"%s", inet_ntoa(((struct
sockaddr_in*)&(ifr.ifr_addr))->sin_addr)); }
```

```
Example 2: Mac address
struct ifreq ifr;
ifr.ifr_addr.sa_family = AF_INET;
 strncpy(ifr.ifr_name, “eth0”, 4);
 ioctl(fd, SIOCGIFHWADDR, &ifr);
```

```
close(fd);

 printf("%.2x%.2x%.2x%.2x%.2x%.2x",
        (unsigned char)ifr.ifr_hwaddr.sa_data[0],
        (unsigned char)ifr.ifr_hwaddr.sa_data[1],
        (unsigned char)ifr.ifr_hwaddr.sa_data[2],
        (unsigned char)ifr.ifr_hwaddr.sa_data[3],
        (unsigned char)ifr.ifr_hwaddr.sa_data[4],
        (unsigned char)ifr.ifr_hwaddr.sa_data[5]);
```

```
#include <net/if.h>
 struct ifreq {
        char ifr_name[IFNAMSIZE];
        union {
         struct sockaddr ifru_addr;
         struct sockaddr ifru_mask;
         struct sockaddr ifru_broadaddr;
         short ifru_flags;
         int ifru_mtu;
         int infu_rbufsize;
         char ifru_linename[10];
         char ifru_TOS;
        } ifr_ifru;
      };
```