16 Sockets

This chapter describes the GNU facilities for interprocess communication using sockets.

A *socket* is a generalized interprocess communication channel. Like a pipe, a socket is represented as a file descriptor. Unlike pipes sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network. Sockets are the primary means of communicating with other machines; `telnet`, `rlogin`, `ftp`, `talk` and the other familiar network programs use sockets.

Not all operating systems support sockets. In the GNU C Library, the header file `sys/socket.h` exists regardless of the operating system, and the socket functions always exist, but if the system does not really support sockets these functions always fail.

Incomplete: We do not currently document the facilities for broadcast messages or for configuring Internet interfaces. The reentrant functions and some newer functions that are related to IPv6 aren't documented either so far.

16.1 Socket Concepts

When you create a socket, you must specify the style of communication you want to use and the type of protocol that should implement it. The *communication style* of a socket defines the user-level semantics of sending and receiving data on the socket. Choosing a communication style specifies the answers to questions such as these:

- **What are the units of data transmission?** Some communication styles regard the data as a sequence of bytes with no larger structure; others group the bytes into records (which are known in this context as *packets*).
- **Can data be lost during normal operation?** Some communication styles guarantee that all the data sent arrives in the order it was sent (barring system or network crashes); other styles occasionally lose data as a normal part of operation, and may sometimes deliver packets more than once or in the wrong order.

  Designing a program to use unreliable communication styles usually involves taking precautions to detect lost or misordered packets and to retransmit data as needed.
- **Is communication entirely with one partner?** Some communication styles are like a telephone call—you make a *connection* with one remote socket and then exchange data freely. Other styles are like mailing letters—you specify a destination address for each message you send.

You must also choose a *namespace* for naming the socket. A socket name ("address") is meaningful only in the context of a particular namespace. In fact, even the data type to use for a socket name may depend on the namespace. Namespaces are also called "domains", but we avoid that word as it can be confused with other usage of the same term. Each namespace has a symbolic name that starts with '`PF_`'. A corresponding symbolic name starting with '`AF_`' designates the address format for that namespace.

Finally you must choose the *protocol* to carry out the communication. The protocol determines what low-level mechanism is used to transmit and receive data. Each protocol is valid for a particular namespace and communication style; a namespace is sometimes called a *protocol family* because of this, which is why the namespace names start with '`PF_`'.

The rules of a protocol apply to the data passing between two programs, perhaps on different computers; most of these rules are handled by the operating system and you need not know about them. What you do need to know about protocols is this:

- In order to have communication between two sockets, they must specify the *same* protocol.
- Each protocol is meaningful with particular style/namespace combinations and cannot be used with inappropriate combinations. For example, the TCP protocol fits only the byte stream style of communication and the Internet namespace.
- For each combination of style and namespace there is a *default protocol*, which you can request by specifying 0 as the protocol number. And that's what you should normally do—use the default.

Throughout the following description at various places variables/parameters to denote sizes are required. And here the trouble starts. In the first implementations the type of these variables was simply `int`. On most machines at that time an `int` was 32 bits wide, which created a *de facto* standard requiring 32-bit variables. This is important since references to variables of this type are passed to the kernel.

Then the POSIX people came and unified the interface with the words "all size values are of type `size_t`". On 64-bit machines `size_t` is 64 bits wide, so pointers to variables were no longer possible.

The Unix98 specification provides a solution by introducing a type `socklen_t`. This type is used in all of the cases that POSIX changed to use `size_t`. The only requirement of this type is that it be an unsigned type of at least 32 bits. Therefore, implementations which require that references to 32-bit variables be passed can be as happy as implementations which use 64-bit values.

16.2 Communication Styles

The GNU C Library includes support for several different kinds of sockets, each with different characteristics. This section describes the supported socket types. The symbolic constants listed here are defined in `sys/socket.h`.

`int SOCK_STREAM` [Macro]

The `SOCK_STREAM` style is like a pipe (see Chapter 15 [Pipes and FIFOs], page 422). It operates over a connection with a particular remote socket and transmits data reliably as a stream of bytes.

Use of this style is covered in detail in Section 16.9 [Using Sockets with Connections], page 453.

`int SOCK_DGRAM` [Macro]

The `SOCK_DGRAM` style is used for sending individually-addressed packets unreliably. It is the diametrical opposite of `SOCK_STREAM`.

Each time you write data to a socket of this kind, that data becomes one packet. Since `SOCK_DGRAM` sockets do not have connections, you must specify the recipient address with each packet.

The only guarantee that the system makes about your requests to transmit data is that it will try its best to deliver each packet you send. It may succeed with the sixth

packet after failing with the fourth and fifth packets; the seventh packet may arrive before the sixth, and may arrive a second time after the sixth.

The typical use for `SOCK_DGRAM` is in situations where it is acceptable to simply re-send a packet if no response is seen in a reasonable amount of time.

See Section 16.10 [Datagram Socket Operations], page 465, for detailed information about how to use datagram sockets.

`int SOCK_RAW` [Macro]

This style provides access to low-level network protocols and interfaces. Ordinary user programs usually have no need to use this style.

16.3 Socket Addresses

The name of a socket is normally called an *address*. The functions and symbols for dealing with socket addresses were named inconsistently, sometimes using the term "name" and sometimes using "address". You can regard these terms as synonymous where sockets are concerned.

A socket newly created with the `socket` function has no address. Other processes can find it for communication only if you give it an address. We call this *binding* the address to the socket, and the way to do it is with the `bind` function.

You need be concerned with the address of a socket if other processes are to find it and start communicating with it. You can specify an address for other sockets, but this is usually pointless; the first time you send data from a socket, or use it to initiate a connection, the system assigns an address automatically if you have not specified one.

Occasionally a client needs to specify an address because the server discriminates based on address; for example, the rsh and rlogin protocols look at the client's socket address and only bypass password checking if it is less than `IPPORT_RESERVED` (see Section 16.6.3 [Internet Ports], page 445).

The details of socket addresses vary depending on what namespace you are using. See Section 16.5 [The Local Namespace], page 433, or Section 16.6 [The Internet Namespace], page 435, for specific information.

Regardless of the namespace, you use the same functions `bind` and `getsockname` to set and examine a socket's address. These functions use a phony data type, `struct sockaddr *`, to accept the address. In practice, the address lives in a structure of some other data type appropriate to the address format you are using, but you cast its address to `struct sockaddr *` when you pass it to `bind`.

16.3.1 Address Formats

The functions `bind` and `getsockname` use the generic data type `struct sockaddr *` to represent a pointer to a socket address. You can't use this data type effectively to interpret an address or construct one; for that, you must use the proper data type for the socket's namespace.

Thus, the usual practice is to construct an address of the proper namespace-specific type, then cast a pointer to `struct sockaddr *` when you call `bind` or `getsockname`.

The one piece of information that you can get from the `struct sockaddr` data type is the *address format designator*. This tells you which data type to use to understand the address fully.

The symbols in this section are defined in the header file `sys/socket.h`.

`struct sockaddr` [Data Type]

The `struct sockaddr` type itself has the following members:

`short int sa_family`
: This is the code for the address format of this address. It identifies the format of the data which follows.

`char sa_data[14]`
: This is the actual socket address data, which is format-dependent. Its length also depends on the format, and may well be more than 14. The length 14 of `sa_data` is essentially arbitrary.

Each address format has a symbolic name which starts with '`AF_`'. Each of them corresponds to a '`PF_`' symbol which designates the corresponding namespace. Here is a list of address format names:

`AF_LOCAL` This designates the address format that goes with the local namespace. (`PF_LOCAL` is the name of that namespace.) See Section 16.5.2 [Details of Local Namespace], page 433, for information about this address format.

`AF_UNIX` This is a synonym for `AF_LOCAL`. Although `AF_LOCAL` is mandated by POSIX.1g, `AF_UNIX` is portable to more systems. `AF_UNIX` was the traditional name stemming from BSD, so even most POSIX systems support it. It is also the name of choice in the Unix98 specification. (The same is true for `PF_UNIX` vs. `PF_LOCAL`).

`AF_FILE` This is another synonym for `AF_LOCAL`, for compatibility. (`PF_FILE` is likewise a synonym for `PF_LOCAL`.)

`AF_INET` This designates the address format that goes with the Internet namespace. (`PF_INET` is the name of that namespace.) See Section 16.6.1 [Internet Socket Address Formats], page 436.

`AF_INET6` This is similar to `AF_INET`, but refers to the IPv6 protocol. (`PF_INET6` is the name of the corresponding namespace.)

`AF_UNSPEC`
: This designates no particular address format. It is used only in rare cases, such as to clear out the default destination address of a "connected" datagram socket. See Section 16.10.1 [Sending Datagrams], page 465.

  The corresponding namespace designator symbol `PF_UNSPEC` exists for completeness, but there is no reason to use it in a program.

`sys/socket.h` defines symbols starting with '`AF_`' for many different kinds of networks, most or all of which are not actually implemented. We will document those that really work as we receive information about how to use them.

16.3.2 Setting the Address of a Socket

Use the `bind` function to assign an address to a socket. The prototype for `bind` is in the header file `sys/socket.h`. For examples of use, see Section 16.5.3 [Example of Local-Namespace Sockets], page 434, or see Section 16.6.7 [Internet Socket Example], page 450.

`int bind` (*int* ***socket***, *struct sockaddr* ***addr***, *socklen_t* ***length***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `bind` function assigns an address to the socket *socket*. The *addr* and *length* arguments specify the address; the detailed format of the address depends on the namespace. The first part of the address is always the format designator, which specifies a namespace, and says that the address is in the format of that namespace.

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file descriptor.

`ENOTSOCK` The descriptor *socket* is not a socket.

`EADDRNOTAVAIL`
: The specified address is not available on this machine.

`EADDRINUSE`
: Some other socket is already using the specified address.

`EINVAL` The socket *socket* already has an address.

`EACCES` You do not have permission to access the requested address. (In the Internet domain, only the super-user is allowed to specify a port number in the range 0 through `IPPORT_RESERVED` minus one; see Section 16.6.3 [Internet Ports], page 445.)

Additional conditions may be possible depending on the particular namespace of the socket.

16.3.3 Reading the Address of a Socket

Use the function `getsockname` to examine the address of an Internet socket. The prototype for this function is in the header file `sys/socket.h`.

`int getsockname` (*int* ***socket***, *struct sockaddr* ***addr***, *socklen_t* ***length-ptr***) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe mem/hurd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getsockname` function returns information about the address of the socket *socket* in the locations specified by the *addr* and *length-ptr* arguments. Note that the *length-ptr* is a pointer; you should initialize it to be the allocation size of *addr*, and on return it contains the actual size of the address data.

The format of the address data depends on the socket namespace. The length of the information is usually fixed for a given namespace, so normally you can know exactly how much space is needed and can provide that much. The usual practice is

to allocate a place for the value using the proper data type for the socket's namespace, then cast its address to `struct sockaddr *` to pass it to `getsockname`.

The return value is `0` on success and `-1` on error. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file descriptor.

`ENOTSOCK` The descriptor *socket* is not a socket.

`ENOBUFS` There are not enough internal buffers available for the operation.

You can't read the address of a socket in the file namespace. This is consistent with the rest of the system; in general, there's no way to find a file's name from a descriptor for that file.

16.4 Interface Naming

Each network interface has a name. This usually consists of a few letters that relate to the type of interface, which may be followed by a number if there is more than one interface of that type. Examples might be `lo` (the loopback interface) and `eth0` (the first Ethernet interface).

Although such names are convenient for humans, it would be clumsy to have to use them whenever a program needs to refer to an interface. In such situations an interface is referred to by its *index*, which is an arbitrarily-assigned small positive integer.

The following functions, constants and data types are declared in the header file `net/if.h`.

`size_t IFNAMSIZ` [Constant]

This constant defines the maximum buffer size needed to hold an interface name, including its terminating zero byte.

`unsigned int if_nametoindex` (*const char *ifname*) [Function]

Preliminary: | MT-Safe | AS-Unsafe lock | AC-Unsafe lock fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function yields the interface index corresponding to a particular name. If no interface exists with the name given, it returns 0.

`char * if_indextoname` (*unsigned int ifindex, char *ifname*) [Function]

Preliminary: | MT-Safe | AS-Unsafe lock | AC-Unsafe lock fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function maps an interface index to its corresponding name. The returned name is placed in the buffer pointed to by `ifname`, which must be at least `IFNAMSIZ` bytes in length. If the index was invalid, the function's return value is a null pointer, otherwise it is `ifname`.

`struct if_nameindex` [Data Type]

This data type is used to hold the information about a single interface. It has the following members:

`unsigned int if_index;`

This is the interface index.

`char *if_name`
: This is the null-terminated index name.

`struct if_nameindex * if_nameindex` (*void*) [Function]
: Preliminary: | MT-Safe | AS-Unsafe heap lock/hurd | AC-Unsafe lock/hurd fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function returns an array of `if_nameindex` structures, one for every interface that is present. The end of the list is indicated by a structure with an interface of 0 and a null name pointer. If an error occurs, this function returns a null pointer.

The returned structure must be freed with `if_freenameindex` after use.

`void if_freenameindex` (*struct if_nameindex *ptr*) [Function]
: Preliminary: | MT-Safe | AS-Unsafe heap | AC-Unsafe mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function frees the structure returned by an earlier call to `if_nameindex`.

16.5 The Local Namespace

This section describes the details of the local namespace, whose symbolic name (required when you create a socket) is `PF_LOCAL`. The local namespace is also known as "Unix domain sockets". Another name is file namespace since socket addresses are normally implemented as file names.

16.5.1 Local Namespace Concepts

In the local namespace socket addresses are file names. You can specify any file name you want as the address of the socket, but you must have write permission on the directory containing it. It's common to put these files in the `/tmp` directory.

One peculiarity of the local namespace is that the name is only used when opening the connection; once open the address is not meaningful and may not exist.

Another peculiarity is that you cannot connect to such a socket from another machine–not even if the other machine shares the file system which contains the name of the socket. You can see the socket in a directory listing, but connecting to it never succeeds. Some programs take advantage of this, such as by asking the client to send its own process ID, and using the process IDs to distinguish between clients. However, we recommend you not use this method in protocols you design, as we might someday permit connections from other machines that mount the same file systems. Instead, send each new client an identifying number if you want it to have one.

After you close a socket in the local namespace, you should delete the file name from the file system. Use `unlink` or `remove` to do this; see Section 14.6 [Deleting Files], page 395.

The local namespace supports just one protocol for any communication style; it is protocol number `0`.

16.5.2 Details of Local Namespace

To create a socket in the local namespace, use the constant `PF_LOCAL` as the *namespace* argument to `socket` or `socketpair`. This constant is defined in `sys/socket.h`.

`int PF_LOCAL` [Macro]
This designates the local namespace, in which socket addresses are local names, and its associated family of protocols. `PF_Local` is the macro used by Posix.1g.

`int PF_UNIX` [Macro]
This is a synonym for `PF_LOCAL`, for compatibility's sake.

`int PF_FILE` [Macro]
This is a synonym for `PF_LOCAL`, for compatibility's sake.

The structure for specifying socket names in the local namespace is defined in the header file `sys/un.h`:

`struct sockaddr_un` [Data Type]
This structure is used to specify local namespace socket addresses. It has the following members:

`short int sun_family`
This identifies the address family or format of the socket address. You should store the value `AF_LOCAL` to designate the local namespace. See Section 16.3 [Socket Addresses], page 429.

`char sun_path[108]`
This is the file name to use.

Incomplete: Why is 108 a magic number? RMS suggests making this a zero-length array and tweaking the following example to use `alloca` to allocate an appropriate amount of storage based on the length of the filename.

You should compute the *length* parameter for a socket address in the local namespace as the sum of the size of the `sun_family` component and the string length (*not* the allocation size!) of the file name string. This can be done using the macro `SUN_LEN`:

`int SUN_LEN` (*struct sockaddr_un * ptr*) [Macro]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The macro computes the length of socket address in the local namespace.

16.5.3 Example of Local-Namespace Sockets

Here is an example showing how to create and name a socket in the local namespace.

```
#include <stddef.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>

int
make_named_socket (const char *filename)
```

```
{
  struct sockaddr_un name;
  int sock;
  size_t size;

  /* Create the socket. */
  sock = socket (PF_LOCAL, SOCK_DGRAM, 0);
  if (sock < 0)
    {
      perror ("socket");
      exit (EXIT_FAILURE);
    }

  /* Bind a name to the socket. */
  name.sun_family = AF_LOCAL;
  strncpy (name.sun_path, filename, sizeof (name.sun_path));
  name.sun_path[sizeof (name.sun_path) - 1] = '\0';

  /* The size of the address is
     the offset of the start of the filename,
     plus its length (not including the terminating null byte).
     Alternatively you can just do:
     size = SUN_LEN (&name);
 */
  size = (offsetof (struct sockaddr_un, sun_path)
          + strlen (name.sun_path));

  if (bind (sock, (struct sockaddr *) &name, size) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return sock;
}
```

16.6 The Internet Namespace

This section describes the details of the protocols and socket naming conventions used in the Internet namespace.

Originally the Internet namespace used only IP version 4 (IPv4). With the growing number of hosts on the Internet, a new protocol with a larger address space was necessary: IP version 6 (IPv6). IPv6 introduces 128-bit addresses (IPv4 has 32-bit addresses) and other features, and will eventually replace IPv4.

To create a socket in the IPv4 Internet namespace, use the symbolic name `PF_INET` of this namespace as the *namespace* argument to `socket` or `socketpair`. For IPv6 addresses you need the macro `PF_INET6`. These macros are defined in `sys/socket.h`.

`int PF_INET` [Macro]
: This designates the IPv4 Internet namespace and associated family of protocols.

`int PF_INET6` [Macro]
: This designates the IPv6 Internet namespace and associated family of protocols.

A socket address for the Internet namespace includes the following components:

- The address of the machine you want to connect to. Internet addresses can be specified in several ways; these are discussed in Section 16.6.1 [Internet Socket Address Formats], page 436, Section 16.6.2 [Host Addresses], page 437 and Section 16.6.2.4 [Host Names], page 441.
- A port number for that machine. See Section 16.6.3 [Internet Ports], page 445.

You must ensure that the address and port number are represented in a canonical format called *network byte order*. See Section 16.6.5 [Byte Order Conversion], page 447, for information about this.

16.6.1 Internet Socket Address Formats

In the Internet namespace, for both IPv4 (`AF_INET`) and IPv6 (`AF_INET6`), a socket address consists of a host address and a port on that host. In addition, the protocol you choose serves effectively as a part of the address because local port numbers are meaningful only within a particular protocol.

The data types for representing socket addresses in the Internet namespace are defined in the header file `netinet/in.h`.

`struct sockaddr_in` [Data Type]

This is the data type used to represent socket addresses in the Internet namespace. It has the following members:

`sa_family_t sin_family`

This identifies the address family or format of the socket address. You should store the value `AF_INET` in this member. See Section 16.3 [Socket Addresses], page 429.

`struct in_addr sin_addr`

This is the Internet address of the host machine. See Section 16.6.2 [Host Addresses], page 437, and Section 16.6.2.4 [Host Names], page 441, for how to get a value to store here.

`unsigned short int sin_port`

This is the port number. See Section 16.6.3 [Internet Ports], page 445.

When you call `bind` or `getsockname`, you should specify `sizeof (struct sockaddr_in)` as the *length* parameter if you are using an IPv4 Internet namespace socket address.

`struct sockaddr_in6` [Data Type]

This is the data type used to represent socket addresses in the IPv6 namespace. It has the following members:

`sa_family_t sin6_family`

This identifies the address family or format of the socket address. You should store the value of `AF_INET6` in this member. See Section 16.3 [Socket Addresses], page 429.

`struct in6_addr sin6_addr`

This is the IPv6 address of the host machine. See Section 16.6.2 [Host Addresses], page 437, and Section 16.6.2.4 [Host Names], page 441, for how to get a value to store here.

`uint32_t sin6_flowinfo`
: This is a currently unimplemented field.

`uint16_t sin6_port`
: This is the port number. See Section 16.6.3 [Internet Ports], page 445.

16.6.2 Host Addresses

Each computer on the Internet has one or more *Internet addresses*, numbers which identify that computer among all those on the Internet. Users typically write IPv4 numeric host addresses as sequences of four numbers, separated by periods, as in '`128.52.46.32`', and IPv6 numeric host addresses as sequences of up to eight numbers separated by colons, as in '`5f03:1200:836f:c100::1`'.

Each computer also has one or more *host names*, which are strings of words separated by periods, as in '`www.gnu.org`'.

Programs that let the user specify a host typically accept both numeric addresses and host names. To open a connection a program needs a numeric address, and so must convert a host name to the numeric address it stands for.

16.6.2.1 Internet Host Addresses

An IPv4 Internet host address is a number containing four bytes of data. Historically these are divided into two parts, a *network number* and a *local network address number* within that network. In the mid-1990s classless addresses were introduced which changed this behavior. Since some functions implicitly expect the old definitions, we first describe the class-based network and will then describe classless addresses. IPv6 uses only classless addresses and therefore the following paragraphs don't apply.

The class-based IPv4 network number consists of the first one, two or three bytes; the rest of the bytes are the local address.

IPv4 network numbers are registered with the Network Information Center (NIC), and are divided into three classes—A, B and C. The local network address numbers of individual machines are registered with the administrator of the particular network.

Class A networks have single-byte numbers in the range 0 to 127. There are only a small number of Class A networks, but they can each support a very large number of hosts. Medium-sized Class B networks have two-byte network numbers, with the first byte in the range 128 to 191. Class C networks are the smallest; they have three-byte network numbers, with the first byte in the range 192-255. Thus, the first 1, 2, or 3 bytes of an Internet address specify a network. The remaining bytes of the Internet address specify the address within that network.

The Class A network 0 is reserved for broadcast to all networks. In addition, the host number 0 within each network is reserved for broadcast to all hosts in that network. These uses are obsolete now but for compatibility reasons you shouldn't use network 0 and host number 0.

The Class A network 127 is reserved for loopback; you can always use the Internet address '`127.0.0.1`' to refer to the host machine.

Since a single machine can be a member of multiple networks, it can have multiple Internet host addresses. However, there is never supposed to be more than one machine with the same host address.

There are four forms of the *standard numbers-and-dots notation* for Internet addresses:

`a.b.c.d` This specifies all four bytes of the address individually and is the commonly used representation.

`a.b.c` The last part of the address, *c*, is interpreted as a 2-byte quantity. This is useful for specifying host addresses in a Class B network with network address number `a.b`.

`a.b` The last part of the address, *b*, is interpreted as a 3-byte quantity. This is useful for specifying host addresses in a Class A network with network address number *a*.

`a` If only one part is given, this corresponds directly to the host address number.

Within each part of the address, the usual C conventions for specifying the radix apply. In other words, a leading '`0x`' or '`0X`' implies hexadecimal radix; a leading '`0`' implies octal; and otherwise decimal radix is assumed.

Classless Addresses

IPv4 addresses (and IPv6 addresses also) are now considered classless; the distinction between classes A, B and C can be ignored. Instead an IPv4 host address consists of a 32-bit address and a 32-bit mask. The mask contains set bits for the network part and cleared bits for the host part. The network part is contiguous from the left, with the remaining bits representing the host. As a consequence, the netmask can simply be specified as the number of set bits. Classes A, B and C are just special cases of this general rule. For example, class A addresses have a netmask of '`255.0.0.0`' or a prefix length of 8.

Classless IPv4 network addresses are written in numbers-and-dots notation with the prefix length appended and a slash as separator. For example the class A network 10 is written as '`10.0.0.0/8`'.

IPv6 Addresses

IPv6 addresses contain 128 bits (IPv4 has 32 bits) of data. A host address is usually written as eight 16-bit hexadecimal numbers that are separated by colons. Two colons are used to abbreviate strings of consecutive zeros. For example, the IPv6 loopback address '`0:0:0:0:0:0:0:1`' can just be written as '`::1`'.

16.6.2.2 Host Address Data Type

IPv4 Internet host addresses are represented in some contexts as integers (type `uint32_t`). In other contexts, the integer is packaged inside a structure of type `struct in_addr`. It would be better if the usage were made consistent, but it is not hard to extract the integer from the structure or put the integer into a structure.

You will find older code that uses `unsigned long int` for IPv4 Internet host addresses instead of `uint32_t` or `struct in_addr`. Historically `unsigned long int` was a 32-bit number but with 64-bit machines this has changed. Using `unsigned long int` might break the code if it is used on machines where this type doesn't have 32 bits. `uint32_t` is specified by Unix98 and guaranteed to have 32 bits.

IPv6 Internet host addresses have 128 bits and are packaged inside a structure of type `struct in6_addr`.

The following basic definitions for Internet addresses are declared in the header file `netinet/in.h`:

`struct in_addr` [Data Type]
: This data type is used in certain contexts to contain an IPv4 Internet host address. It has just one field, named `s_addr`, which records the host address number as an `uint32_t`.

`uint32_t INADDR_LOOPBACK` [Macro]
: You can use this constant to stand for "the address of this machine," instead of finding its actual address. It is the IPv4 Internet address '`127.0.0.1`', which is usually called '`localhost`'. This special constant saves you the trouble of looking up the address of your own machine. Also, the system usually implements `INADDR_LOOPBACK` specially, avoiding any network traffic for the case of one machine talking to itself.

`uint32_t INADDR_ANY` [Macro]
: You can use this constant to stand for "any incoming address" when binding to an address. See Section 16.3.2 [Setting the Address of a Socket], page 431. This is the usual address to give in the `sin_addr` member of `struct sockaddr_in` when you want to accept Internet connections.

`uint32_t INADDR_BROADCAST` [Macro]
: This constant is the address you use to send a broadcast message.

`uint32_t INADDR_NONE` [Macro]
: This constant is returned by some functions to indicate an error.

`struct in6_addr` [Data Type]
: This data type is used to store an IPv6 address. It stores 128 bits of data, which can be accessed (via a union) in a variety of ways.

`struct in6_addr in6addr_loopback` [Constant]
: This constant is the IPv6 address '`::1`', the loopback address. See above for a description of what this means. The macro `IN6ADDR_LOOPBACK_INIT` is provided to allow you to initialize your own variables to this value.

`struct in6_addr in6addr_any` [Constant]
: This constant is the IPv6 address '`::`', the unspecified address. See above for a description of what this means. The macro `IN6ADDR_ANY_INIT` is provided to allow you to initialize your own variables to this value.

16.6.2.3 Host Address Functions

These additional functions for manipulating Internet addresses are declared in the header file `arpa/inet.h`. They represent Internet addresses in network byte order, and network numbers and local-address-within-network numbers in host byte order. See Section 16.6.5 [Byte Order Conversion], page 447, for an explanation of network and host byte order.

`int inet_aton` (*const char ***name**, struct in_addr ***addr***) [Function]
: Preliminary: | MT-Safe locale | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the IPv4 Internet host address *name* from the standard numbers-and-dots notation into binary data and stores it in the `struct in_addr` that *addr* points to. `inet_aton` returns nonzero if the address is valid, zero if not.

`uint32_t inet_addr` (*const char *name*) [Function]

Preliminary: | MT-Safe locale | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the IPv4 Internet host address *name* from the standard numbers-and-dots notation into binary data. If the input is not valid, `inet_addr` returns `INADDR_NONE`. This is an obsolete interface to `inet_aton`, described immediately above. It is obsolete because `INADDR_NONE` is a valid address (255.255.255.255), and `inet_aton` provides a cleaner way to indicate error return.

`uint32_t inet_network` (*const char *name*) [Function]

Preliminary: | MT-Safe locale | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function extracts the network number from the address *name*, given in the standard numbers-and-dots notation. The returned address is in host order. If the input is not valid, `inet_network` returns `-1`.

The function works only with traditional IPv4 class A, B and C network types. It doesn't work with classless addresses and shouldn't be used anymore.

`char * inet_ntoa` (*struct in_addr* `addr`) [Function]

Preliminary: | MT-Safe locale | AS-Unsafe race | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the IPv4 Internet host address *addr* to a string in the standard numbers-and-dots notation. The return value is a pointer into a statically-allocated buffer. Subsequent calls will overwrite the same buffer, so you should copy the string if you need to save it.

In multi-threaded programs each thread has an own statically-allocated buffer. But still subsequent calls of `inet_ntoa` in the same thread will overwrite the result of the last call.

Instead of `inet_ntoa` the newer function `inet_ntop` which is described below should be used since it handles both IPv4 and IPv6 addresses.

`struct in_addr inet_makeaddr` (*uint32_t* `net`, *uint32_t* `local`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function makes an IPv4 Internet host address by combining the network number *net* with the local-address-within-network number *local*.

`uint32_t inet_lnaof` (*struct in_addr* `addr`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function returns the local-address-within-network part of the Internet host address *addr*.

The function works only with traditional IPv4 class A, B and C network types. It doesn't work with classless addresses and shouldn't be used anymore.

`uint32_t inet_netof` (*struct in_addr* `addr`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function returns the network number part of the Internet host address *addr*.

The function works only with traditional IPv4 class A, B and C network types. It doesn't work with classless addresses and shouldn't be used anymore.

`int inet_pton` (*int* `af`, *const char* *`cp`, *void* *`buf`) [Function]

Preliminary: | MT-Safe locale | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts an Internet address (either IPv4 or IPv6) from presentation (textual) to network (binary) format. *af* should be either `AF_INET` or `AF_INET6`, as appropriate for the type of address being converted. *cp* is a pointer to the input string, and *buf* is a pointer to a buffer for the result. It is the caller's responsibility to make sure the buffer is large enough.

`const char * inet_ntop` (*int* `af`, *const void* *`cp`, *char* *`buf`, *socklen_t* `len`) [Function]

Preliminary: | MT-Safe locale | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts an Internet address (either IPv4 or IPv6) from network (binary) to presentation (textual) form. *af* should be either `AF_INET` or `AF_INET6`, as appropriate. *cp* is a pointer to the address to be converted. *buf* should be a pointer to a buffer to hold the result, and *len* is the length of this buffer. The return value from the function will be this buffer address.

16.6.2.4 Host Names

Besides the standard numbers-and-dots notation for Internet addresses, you can also refer to a host by a symbolic name. The advantage of a symbolic name is that it is usually easier to remember. For example, the machine with Internet address '`158.121.106.19`' is also known as '`alpha.gnu.org`'; and other machines in the '`gnu.org`' domain can refer to it simply as '`alpha`'.

Internally, the system uses a database to keep track of the mapping between host names and host numbers. This database is usually either the file `/etc/hosts` or an equivalent provided by a name server. The functions and other symbols for accessing this database are declared in `netdb.h`. They are BSD features, defined unconditionally if you include `netdb.h`.

`struct hostent` [Data Type]

This data type is used to represent an entry in the hosts database. It has the following members:

`char *h_name`

This is the "official" name of the host.

`char **h_aliases`

These are alternative names for the host, represented as a null-terminated vector of strings.

`int h_addrtype`
: This is the host address type; in practice, its value is always either `AF_INET` or `AF_INET6`, with the latter being used for IPv6 hosts. In principle other kinds of addresses could be represented in the database as well as Internet addresses; if this were done, you might find a value in this field other than `AF_INET` or `AF_INET6`. See Section 16.3 [Socket Addresses], page 429.

`int h_length`
: This is the length, in bytes, of each address.

`char **h_addr_list`
: This is the vector of addresses for the host. (Recall that the host might be connected to multiple networks and have different addresses on each one.) The vector is terminated by a null pointer.

`char *h_addr`
: This is a synonym for `h_addr_list[0]`; in other words, it is the first host address.

As far as the host database is concerned, each address is just a block of memory `h_length` bytes long. But in other contexts there is an implicit assumption that you can convert IPv4 addresses to a `struct in_addr` or an `uint32_t`. Host addresses in a `struct hostent` structure are always given in network byte order; see Section 16.6.5 [Byte Order Conversion], page 447.

You can use `gethostbyname`, `gethostbyname2` or `gethostbyaddr` to search the hosts database for information about a particular host. The information is returned in a statically-allocated structure; you must copy the information if you need to save it across calls. You can also use `getaddrinfo` and `getnameinfo` to obtain this information.

`struct hostent * gethostbyname` (*const char *name*) [Function]
: Preliminary: | MT-Unsafe race:hostbyname env locale | AS-Unsafe dlopen plugin corrupt heap lock | AC-Unsafe lock corrupt mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

  The `gethostbyname` function returns information about the host named *name*. If the lookup fails, it returns a null pointer.

`struct hostent * gethostbyname2` (*const char *name*, *int af*) [Function]
: Preliminary: | MT-Unsafe race:hostbyname2 env locale | AS-Unsafe dlopen plugin corrupt heap lock | AC-Unsafe lock corrupt mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

  The `gethostbyname2` function is like `gethostbyname`, but allows the caller to specify the desired address family (e.g. `AF_INET` or `AF_INET6`) of the result.

`struct hostent * gethostbyaddr` (*const void *addr*, *socklen_t length*, *int format*) [Function]
: Preliminary: | MT-Unsafe race:hostbyaddr env locale | AS-Unsafe dlopen plugin corrupt heap lock | AC-Unsafe lock corrupt mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `gethostbyaddr` function returns information about the host with Internet address *addr*. The parameter *addr* is not really a pointer to char - it can be a pointer to an IPv4 or an IPv6 address. The *length* argument is the size (in bytes) of the address at *addr*. *format* specifies the address format; for an IPv4 Internet address, specify a value of `AF_INET`; for an IPv6 Internet address, use `AF_INET6`.

If the lookup fails, `gethostbyaddr` returns a null pointer.

If the name lookup by `gethostbyname` or `gethostbyaddr` fails, you can find out the reason by looking at the value of the variable `h_errno`. (It would be cleaner design for these functions to set `errno`, but use of `h_errno` is compatible with other systems.)

Here are the error codes that you may find in `h_errno`:

`HOST_NOT_FOUND`
: No such host is known in the database.

`TRY_AGAIN`
: This condition happens when the name server could not be contacted. If you try again later, you may succeed then.

`NO_RECOVERY`
: A non-recoverable error occurred.

`NO_ADDRESS`
: The host database contains an entry for the name, but it doesn't have an associated Internet address.

The lookup functions above all have one in common: they are not reentrant and therefore unusable in multi-threaded applications. Therefore provides the GNU C Library a new set of functions which can be used in this context.

`int gethostbyname_r` (*const char *restrict* **`name`**, *struct hostent *restrict* **`result_buf`**, *char *restrict* **`buf`**, *size_t* **`buflen`**, *struct hostent **restrict* **`result`**, *int *restrict* **`h_errnop`**) [Function]

Preliminary: | MT-Safe env locale | AS-Unsafe dlopen plugin corrupt heap lock | AC-Unsafe lock corrupt mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `gethostbyname_r` function returns information about the host named *name*. The caller must pass a pointer to an object of type `struct hostent` in the *result_buf* parameter. In addition the function may need extra buffer space and the caller must pass an pointer and the size of the buffer in the *buf* and *buflen* parameters.

A pointer to the buffer, in which the result is stored, is available in `*result` after the function call successfully returned. The buffer passed as the *buf* parameter can be freed only once the caller has finished with the result hostent struct, or has copied it including all the other memory that it points to. If an error occurs or if no entry is found, the pointer `*result` is a null pointer. Success is signalled by a zero return value. If the function failed the return value is an error number. In addition to the errors defined for `gethostbyname` it can also be `ERANGE`. In this case the call should be repeated with a larger buffer. Additional error information is not stored in the global variable `h_errno` but instead in the object pointed to by *h_errnop*.

Here's a small example:

```
struct hostent *
gethostname (char *host)
{
  struct hostent *hostbuf, *hp;
  size_t hstbuflen;
  char *tmphstbuf;
  int res;
  int herr;

  hostbuf = malloc (sizeof (struct hostent));
  hstbuflen = 1024;
  tmphstbuf = malloc (hstbuflen);

  while ((res = gethostbyname_r (host, hostbuf, tmphstbuf, hstbuflen,
                                 &hp, &herr)) == ERANGE)
    {
      /* Enlarge the buffer.  */
      hstbuflen *= 2;
      tmphstbuf = realloc (tmphstbuf, hstbuflen);
    }

  free (tmphstbuf);
  /*  Check for errors.  */
  if (res || hp == NULL)
    return NULL;
  return hp;
}
```

`int gethostbyname2_r` (*const char *__name__, int __af__, struct hostent *restrict __result_buf__, char *restrict __buf__, size_t __buflen__, struct hostent **restrict __result__, int *restrict __h_errnop__*) [Function]

Preliminary: | MT-Safe env locale | AS-Unsafe dlopen plugin corrupt heap lock | AC-Unsafe lock corrupt mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `gethostbyname2_r` function is like `gethostbyname_r`, but allows the caller to specify the desired address family (e.g. `AF_INET` or `AF_INET6`) for the result.

`int gethostbyaddr_r` (*const void *__addr__, socklen_t __length__, int __format__, struct hostent *restrict __result_buf__, char *restrict __buf__, size_t __buflen__, struct hostent **restrict __result__, int *restrict __h_errnop__*) [Function]

Preliminary: | MT-Safe env locale | AS-Unsafe dlopen plugin corrupt heap lock | AC-Unsafe lock corrupt mem fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `gethostbyaddr_r` function returns information about the host with Internet address *addr*. The parameter *addr* is not really a pointer to char - it can be a pointer to an IPv4 or an IPv6 address. The *length* argument is the size (in bytes) of the address at *addr*. *format* specifies the address format; for an IPv4 Internet address, specify a value of `AF_INET`; for an IPv6 Internet address, use `AF_INET6`.

Similar to the `gethostbyname_r` function, the caller must provide buffers for the result and memory used internally. In case of success the function returns zero. Otherwise the value is an error number where `ERANGE` has the special meaning that the caller-provided buffer is too small.

You can also scan the entire hosts database one entry at a time using `sethostent`, `gethostent` and `endhostent`. Be careful when using these functions because they are not reentrant.

`void sethostent` (*int stayopen*) [Function]

Preliminary: | MT-Unsafe race:hostent env locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function opens the hosts database to begin scanning it. You can then call `gethostent` to read the entries.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `gethostbyname` or `gethostbyaddr` will not close the database (as they usually would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct hostent * gethostent` (*void*) [Function]

Preliminary: | MT-Unsafe race:hostent race:hostentbuf env locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function returns the next entry in the hosts database. It returns a null pointer if there are no more entries.

`void endhostent` (*void*) [Function]

Preliminary: | MT-Unsafe race:hostent env locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function closes the hosts database.

16.6.3 Internet Ports

A socket address in the Internet namespace consists of a machine's Internet address plus a *port number* which distinguishes the sockets on a given machine (for a given protocol). Port numbers range from 0 to 65,535.

Port numbers less than `IPPORT_RESERVED` are reserved for standard servers, such as `finger` and `telnet`. There is a database that keeps track of these, and you can use the `getservbyname` function to map a service name onto a port number; see Section 16.6.4 [The Services Database], page 446.

If you write a server that is not one of the standard ones defined in the database, you must choose a port number for it. Use a number greater than `IPPORT_USERRESERVED`; such numbers are reserved for servers and won't ever be generated automatically by the system. Avoiding conflicts with servers being run by other users is up to you.

When you use a socket without specifying its address, the system generates a port number for it. This number is between `IPPORT_RESERVED` and `IPPORT_USERRESERVED`.

On the Internet, it is actually legitimate to have two different sockets with the same port number, as long as they never both try to communicate with the same socket address (host address plus port number). You shouldn't duplicate a port number except in special circumstances where a higher-level protocol requires it. Normally, the system won't let you

do it; `bind` normally insists on distinct port numbers. To reuse a port number, you must set the socket option `SO_REUSEADDR`. See Section 16.12.2 [Socket-Level Options], page 471.

These macros are defined in the header file `netinet/in.h`.

`int IPPORT_RESERVED` [Macro]
Port numbers less than `IPPORT_RESERVED` are reserved for superuser use.

`int IPPORT_USERRESERVED` [Macro]
Port numbers greater than or equal to `IPPORT_USERRESERVED` are reserved for explicit use; they will never be allocated automatically.

16.6.4 The Services Database

The database that keeps track of "well-known" services is usually either the file `/etc/services` or an equivalent from a name server. You can use these utilities, declared in `netdb.h`, to access the services database.

`struct servent` [Data Type]
This data type holds information about entries from the services database. It has the following members:

`char *s_name`
This is the "official" name of the service.

`char **s_aliases`
These are alternate names for the service, represented as an array of strings. A null pointer terminates the array.

`int s_port`
This is the port number for the service. Port numbers are given in network byte order; see Section 16.6.5 [Byte Order Conversion], page 447.

`char *s_proto`
This is the name of the protocol to use with this service. See Section 16.6.6 [Protocols Database], page 448.

To get information about a particular service, use the `getservbyname` or `getservbyport` functions. The information is returned in a statically-allocated structure; you must copy the information if you need to save it across calls.

`struct servent * getservbyname (const char *name, const char *proto)` [Function]
Preliminary: | MT-Unsafe race:servbyname locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getservbyname` function returns information about the service named *name* using protocol *proto*. If it can't find such a service, it returns a null pointer.

This function is useful for servers as well as for clients; servers use it to determine which port they should listen on (see Section 16.9.2 [Listening for Connections], page 455).

`struct servent * getservbyport` (*int* `port`, *const char* `*proto`) [Function]
Preliminary: | MT-Unsafe race:servbyport locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getservbyport` function returns information about the service at port *port* using protocol *proto*. If it can't find such a service, it returns a null pointer.

You can also scan the services database using `setservent`, `getservent` and `endservent`. Be careful when using these functions because they are not reentrant.

`void setservent` (*int* `stayopen`) [Function]
Preliminary: | MT-Unsafe race:servent locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function opens the services database to begin scanning it.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `getservbyname` or `getservbyport` will not close the database (as they usually would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct servent * getservent` (*void*) [Function]
Preliminary: | MT-Unsafe race:servent race:serventbuf locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function returns the next entry in the services database. If there are no more entries, it returns a null pointer.

`void endservent` (*void*) [Function]
Preliminary: | MT-Unsafe race:servent locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function closes the services database.

16.6.5 Byte Order Conversion

Different kinds of computers use different conventions for the ordering of bytes within a word. Some computers put the most significant byte within a word first (this is called "big-endian" order), and others put it last ("little-endian" order).

So that machines with different byte order conventions can communicate, the Internet protocols specify a canonical byte order convention for data transmitted over the network. This is known as *network byte order*.

When establishing an Internet socket connection, you must make sure that the data in the `sin_port` and `sin_addr` members of the `sockaddr_in` structure are represented in network byte order. If you are encoding integer data in the messages sent through the socket, you should convert this to network byte order too. If you don't do this, your program may fail when running on or talking to other kinds of machines.

If you use `getservbyname` and `gethostbyname` or `inet_addr` to get the port number and host address, the values are already in network byte order, and you can copy them directly into the `sockaddr_in` structure.

Otherwise, you have to convert the values explicitly. Use `htons` and `ntohs` to convert values for the `sin_port` member. Use `htonl` and `ntohl` to convert IPv4 addresses for the `sin_addr` member. (Remember, `struct in_addr` is equivalent to `uint32_t`.) These functions are declared in `netinet/in.h`.

`uint16_t htons` (*uint16_t* ***hostshort***) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the `uint16_t` integer *hostshort* from host byte order to network byte order.

`uint16_t ntohs` (*uint16_t* ***netshort***) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the `uint16_t` integer *netshort* from network byte order to host byte order.

`uint32_t htonl` (*uint32_t* ***hostlong***) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the `uint32_t` integer *hostlong* from host byte order to network byte order.

This is used for IPv4 Internet addresses.

`uint32_t ntohl` (*uint32_t* ***netlong***) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function converts the `uint32_t` integer *netlong* from network byte order to host byte order.

This is used for IPv4 Internet addresses.

16.6.6 Protocols Database

The communications protocol used with a socket controls low-level details of how data are exchanged. For example, the protocol implements things like checksums to detect errors in transmissions, and routing instructions for messages. Normal user programs have little reason to mess with these details directly.

The default communications protocol for the Internet namespace depends on the communication style. For stream communication, the default is TCP ("transmission control protocol"). For datagram communication, the default is UDP ("user datagram protocol"). For reliable datagram communication, the default is RDP ("reliable datagram protocol"). You should nearly always use the default.

Internet protocols are generally specified by a name instead of a number. The network protocols that a host knows about are stored in a database. This is usually either derived

from the file `/etc/protocols`, or it may be an equivalent provided by a name server. You look up the protocol number associated with a named protocol in the database using the `getprotobyname` function.

Here are detailed descriptions of the utilities for accessing the protocols database. These are declared in `netdb.h`.

`struct protoent` [Data Type]

This data type is used to represent entries in the network protocols database. It has the following members:

`char *p_name`
: This is the official name of the protocol.

`char **p_aliases`
: These are alternate names for the protocol, specified as an array of strings. The last element of the array is a null pointer.

`int p_proto`
: This is the protocol number (in host byte order); use this member as the *protocol* argument to `socket`.

You can use `getprotobyname` and `getprotobynumber` to search the protocols database for a specific protocol. The information is returned in a statically-allocated structure; you must copy the information if you need to save it across calls.

`struct protoent * getprotobyname` (*const char *name*) [Function]

Preliminary: | MT-Unsafe race:protobyname locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getprotobyname` function returns information about the network protocol named *name*. If there is no such protocol, it returns a null pointer.

`struct protoent * getprotobynumber` (*int protocol*) [Function]

Preliminary: | MT-Unsafe race:protobynumber locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getprotobynumber` function returns information about the network protocol with number *protocol*. If there is no such protocol, it returns a null pointer.

You can also scan the whole protocols database one protocol at a time by using `setprotoent`, `getprotoent` and `endprotoent`. Be careful when using these functions because they are not reentrant.

`void setprotoent` (*int stayopen*) [Function]

Preliminary: | MT-Unsafe race:protoent locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function opens the protocols database to begin scanning it.

If the *stayopen* argument is nonzero, this sets a flag so that subsequent calls to `getprotobyname` or `getprotobynumber` will not close the database (as they usually

would). This makes for more efficiency if you call those functions several times, by avoiding reopening the database for each call.

`struct protoent * getprotoent` (*void*) [Function]

Preliminary: | MT-Unsafe race:protoent race:protoentbuf locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function returns the next entry in the protocols database. It returns a null pointer if there are no more entries.

`void endprotoent` (*void*) [Function]

Preliminary: | MT-Unsafe race:protoent locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function closes the protocols database.

16.6.7 Internet Socket Example

Here is an example showing how to create and name a socket in the Internet namespace. The newly created socket exists on the machine that the program is running on. Rather than finding and using the machine's Internet address, this example specifies `INADDR_ANY` as the host address; the system replaces that with the machine's actual address.

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

int
make_socket (uint16_t port)
{
  int sock;
  struct sockaddr_in name;

  /* Create the socket. */
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
      perror ("socket");
      exit (EXIT_FAILURE);
    }

  /* Give the socket a name. */
  name.sin_family = AF_INET;
  name.sin_port = htons (port);
  name.sin_addr.s_addr = htonl (INADDR_ANY);
  if (bind (sock, (struct sockaddr *) &name, sizeof (name)) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return sock;
}
```

Here is another example, showing how you can fill in a `sockaddr_in` structure, given a host name string and a port number:

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

void
init_sockaddr (struct sockaddr_in *name,
               const char *hostname,
               uint16_t port)
{
  struct hostent *hostinfo;

  name->sin_family = AF_INET;
  name->sin_port = htons (port);
  hostinfo = gethostbyname (hostname);
  if (hostinfo == NULL)
    {
      fprintf (stderr, "Unknown host %s.\n", hostname);
      exit (EXIT_FAILURE);
    }
  name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
}
```

16.7 Other Namespaces

Certain other namespaces and associated protocol families are supported but not documented yet because they are not often used. `PF_NS` refers to the Xerox Network Software protocols. `PF_ISO` stands for Open Systems Interconnect. `PF_CCITT` refers to protocols from CCITT. `socket.h` defines these symbols and others naming protocols not actually implemented.

`PF_IMPLINK` is used for communicating between hosts and Internet Message Processors. For information on this and `PF_ROUTE`, an occasionally-used local area routing protocol, see the GNU Hurd Manual (to appear in the future).

16.8 Opening and Closing Sockets

This section describes the actual library functions for opening and closing sockets. The same functions work for all namespaces and connection styles.

16.8.1 Creating a Socket

The primitive for creating a socket is the `socket` function, declared in `sys/socket.h`.

`int socket` (*int* `namespace`, *int* `style`, *int* `protocol`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function creates a socket and specifies communication style *style*, which should be one of the socket styles listed in Section 16.2 [Communication Styles], page 428. The *namespace* argument specifies the namespace; it must be `PF_LOCAL` (see Section 16.5

[The Local Namespace], page 433) or `PF_INET` (see Section 16.6 [The Internet Namespace], page 435). *protocol* designates the specific protocol (see Section 16.1 [Socket Concepts], page 427); zero is usually right for *protocol*.

The return value from `socket` is the file descriptor for the new socket, or `-1` in case of error. The following `errno` error conditions are defined for this function:

`EPROTONOSUPPORT`
: The *protocol* or *style* is not supported by the *namespace* specified.

`EMFILE`
: The process already has too many file descriptors open.

`ENFILE`
: The system already has too many file descriptors open.

`EACCES`
: The process does not have the privilege to create a socket of the specified *style* or *protocol*.

`ENOBUFS`
: The system ran out of internal buffer space.

The file descriptor returned by the `socket` function supports both read and write operations. However, like pipes, sockets do not support file positioning operations.

For examples of how to call the `socket` function, see Section 16.5.3 [Example of Local-Namespace Sockets], page 434, or Section 16.6.7 [Internet Socket Example], page 450.

16.8.2 Closing a Socket

When you have finished using a socket, you can simply close its file descriptor with `close`; see Section 13.1 [Opening and Closing Files], page 322. If there is still data waiting to be transmitted over the connection, normally `close` tries to complete this transmission. You can control this behavior using the `SO_LINGER` socket option to specify a timeout period; see Section 16.12 [Socket Options], page 470.

You can also shut down only reception or transmission on a connection by calling `shutdown`, which is declared in `sys/socket.h`.

`int shutdown` (*int* `socket`, *int* `how`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `shutdown` function shuts down the connection of socket *socket*. The argument *how* specifies what action to perform:

`0`
: Stop receiving data for this socket. If further data arrives, reject it.

`1`
: Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don't retransmit it if it is lost.

`2`
: Stop both reception and transmission.

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF`
: *socket* is not a valid file descriptor.

`ENOTSOCK`
: *socket* is not a socket.

`ENOTCONN`
: *socket* is not connected.

16.8.3 Socket Pairs

A *socket pair* consists of a pair of connected (but unnamed) sockets. It is very similar to a pipe and is used in much the same way. Socket pairs are created with the `socketpair` function, declared in `sys/socket.h`. A socket pair is much like a pipe; the main difference is that the socket pair is bidirectional, whereas the pipe has one input-only end and one output-only end (see Chapter 15 [Pipes and FIFOs], page 422).

`int socketpair (`*int* `namespace`*, int* `style`*, int* `protocol`*, int* `filedes`[2]`)` [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function creates a socket pair, returning the file descriptors in `filedes[0]` and `filedes[1]`. The socket pair is a full-duplex communications channel, so that both reading and writing may be performed at either end.

The *namespace*, *style* and *protocol* arguments are interpreted as for the `socket` function. *style* should be one of the communication styles listed in Section 16.2 [Communication Styles], page 428. The *namespace* argument specifies the namespace, which must be `AF_LOCAL` (see Section 16.5 [The Local Namespace], page 433); *protocol* specifies the communications protocol, but zero is the only meaningful value.

If *style* specifies a connectionless communication style, then the two sockets you get are not *connected*, strictly speaking, but each of them knows the other as the default destination address, so they can send packets to each other.

The `socketpair` function returns `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EMFILE` The process has too many file descriptors open.

`EAFNOSUPPORT`
: The specified namespace is not supported.

`EPROTONOSUPPORT`
: The specified protocol is not supported.

`EOPNOTSUPP`
: The specified protocol does not support the creation of socket pairs.

16.9 Using Sockets with Connections

The most common communication styles involve making a connection to a particular other socket, and then exchanging data with that socket over and over. Making a connection is asymmetric; one side (the *client*) acts to request a connection, while the other side (the *server*) makes a socket and waits for the connection request.

- Section 16.9.1 [Making a Connection], page 454, describes what the client program must do to initiate a connection with a server.
- Section 16.9.2 [Listening for Connections], page 455 and Section 16.9.3 [Accepting Connections], page 455 describe what the server program must do to wait for and act upon connection requests from clients.
- Section 16.9.5 [Transferring Data], page 457, describes how data are transferred through the connected socket.

16.9.1 Making a Connection

In making a connection, the client makes a connection while the server waits for and accepts the connection. Here we discuss what the client program must do with the `connect` function, which is declared in `sys/socket.h`.

`int connect` (*int* `socket`, *struct sockaddr* *`addr`, *socklen_t* `length`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `connect` function initiates a connection from the socket with file descriptor *socket* to the socket whose address is specified by the *addr* and *length* arguments. (This socket is typically on another machine, and it must be already set up as a server.) See Section 16.3 [Socket Addresses], page 429, for information about how these arguments are interpreted.

Normally, `connect` waits until the server responds to the request before it returns. You can set nonblocking mode on the socket *socket* to make `connect` return immediately without waiting for the response. See Section 13.14 [File Status Flags], page 362, for information about nonblocking mode.

The normal return value from `connect` is `0`. If an error occurs, `connect` returns `-1`. The following `errno` error conditions are defined for this function:

`EBADF` The socket *socket* is not a valid file descriptor.

`ENOTSOCK` File descriptor *socket* is not a socket.

`EADDRNOTAVAIL`
: The specified address is not available on the remote machine.

`EAFNOSUPPORT`
: The namespace of the *addr* is not supported by this socket.

`EISCONN` The socket *socket* is already connected.

`ETIMEDOUT`
: The attempt to establish the connection timed out.

`ECONNREFUSED`
: The server has actively refused to establish the connection.

`ENETUNREACH`
: The network of the given *addr* isn't reachable from this host.

`EADDRINUSE`
: The socket address of the given *addr* is already in use.

`EINPROGRESS`
: The socket *socket* is non-blocking and the connection could not be established immediately. You can determine when the connection is completely established with `select`; see Section 13.8 [Waiting for Input or Output], page 341. Another `connect` call on the same socket, before the connection is completely established, will fail with `EALREADY`.

`EALREADY` The socket *socket* is non-blocking and already has a pending connection in progress (see `EINPROGRESS` above).

This function is defined as a cancellation point in multi-threaded programs, so one has to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores or whatever) are freed even if the thread is canceled.

16.9.2 Listening for Connections

Now let us consider what the server process must do to accept connections on a socket. First it must use the `listen` function to enable connection requests on the socket, and then accept each incoming connection with a call to `accept` (see Section 16.9.3 [Accepting Connections], page 455). Once connection requests are enabled on a server socket, the `select` function reports when the socket has a connection ready to be accepted (see Section 13.8 [Waiting for Input or Output], page 341).

The `listen` function is not allowed for sockets using connectionless communication styles.

You can write a network server that does not even start running until a connection to it is requested. See Section 16.11.1 [`inetd` Servers], page 469.

In the Internet namespace, there are no special protection mechanisms for controlling access to a port; any process on any machine can make a connection to your server. If you want to restrict access to your server, make it examine the addresses associated with connection requests or implement some other handshaking or identification protocol.

In the local namespace, the ordinary file protection bits control who has access to connect to the socket.

`int listen` (*int* `socket`, *int* `n`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `listen` function enables the socket *socket* to accept connections, thus making it a server socket.

The argument *n* specifies the length of the queue for pending connections. When the queue fills, new clients attempting to connect fail with `ECONNREFUSED` until the server calls `accept` to accept a connection from the queue.

The `listen` function returns `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF` The argument *socket* is not a valid file descriptor.

`ENOTSOCK` The argument *socket* is not a socket.

`EOPNOTSUPP`
The socket *socket* does not support this operation.

16.9.3 Accepting Connections

When a server receives a connection request, it can complete the connection by accepting the request. Use the function `accept` to do this.

A socket that has been established as a server can accept connection requests from multiple clients. The server's original socket *does not become part of the connection*; instead, `accept` makes a new socket which participates in the connection. `accept` returns the

descriptor for this socket. The server's original socket remains available for listening for further connection requests.

The number of pending connection requests on a server socket is finite. If connection requests arrive from clients faster than the server can act upon them, the queue can fill up and additional requests are refused with an `ECONNREFUSED` error. You can specify the maximum length of this queue as an argument to the `listen` function, although the system may also impose its own internal limit on the length of this queue.

`int accept` (*int* `socket`, *struct sockaddr* *`addr`, *socklen_t* *`length_ptr`) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is used to accept a connection request on the server socket *socket*.

The `accept` function waits if there are no connections pending, unless the socket *socket* has nonblocking mode set. (You can use `select` to wait for a pending connection, with a nonblocking socket.) See Section 13.14 [File Status Flags], page 362, for information about nonblocking mode.

The *addr* and *length-ptr* arguments are used to return information about the name of the client socket that initiated the connection. See Section 16.3 [Socket Addresses], page 429, for information about the format of the information.

Accepting a connection does not make *socket* part of the connection. Instead, it creates a new socket which becomes connected. The normal return value of `accept` is the file descriptor for the new socket.

After `accept`, the original socket *socket* remains open and unconnected, and continues listening until you close it. You can accept further connections with *socket* by calling `accept` again.

If an error occurs, `accept` returns `-1`. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file descriptor.

`ENOTSOCK` The descriptor *socket* argument is not a socket.

`EOPNOTSUPP`
The descriptor *socket* does not support this operation.

`EWOULDBLOCK`
socket has nonblocking mode set, and there are no pending connections immediately available.

This function is defined as a cancellation point in multi-threaded programs, so one has to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores or whatever) are freed even if the thread is canceled.

The `accept` function is not allowed for sockets using connectionless communication styles.

16.9.4 Who is Connected to Me?

`int getpeername` (*int* `socket`, *struct sockaddr* *`addr`, *socklen_t* *`length-ptr`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getpeername` function returns the address of the socket that *socket* is connected to; it stores the address in the memory space specified by *addr* and *length-ptr*. It stores the length of the address in `*length-ptr`.

See Section 16.3 [Socket Addresses], page 429, for information about the format of the address. In some operating systems, `getpeername` works only for sockets in the Internet domain.

The return value is `0` on success and `-1` on error. The following `errno` error conditions are defined for this function:

`EBADF` The argument *socket* is not a valid file descriptor.

`ENOTSOCK` The descriptor *socket* is not a socket.

`ENOTCONN` The socket *socket* is not connected.

`ENOBUFS` There are not enough internal buffers available.

16.9.5 Transferring Data

Once a socket has been connected to a peer, you can use the ordinary `read` and `write` operations (see Section 13.2 [Input and Output Primitives], page 325) to transfer data. A socket is a two-way communications channel, so read and write operations can be performed at either end.

There are also some I/O modes that are specific to socket operations. In order to specify these modes, you must use the `recv` and `send` functions instead of the more generic `read` and `write` functions. The `recv` and `send` functions take an additional argument which you can use to specify various flags to control special I/O modes. For example, you can specify the `MSG_OOB` flag to read or write out-of-band data, the `MSG_PEEK` flag to peek at input, or the `MSG_DONTROUTE` flag to control inclusion of routing information on output.

16.9.5.1 Sending Data

The `send` function is declared in the header file `sys/socket.h`. If your *flags* argument is zero, you can just as well use `write` instead of `send`; see Section 13.2 [Input and Output Primitives], page 325. If the socket was connected but the connection has broken, you get a `SIGPIPE` signal for any use of `send` or `write` (see Section 24.2.7 [Miscellaneous Signals], page 668).

`ssize_t send` (*int* `socket`, *const void* *`buffer`, *size_t* `size`, *int* `flags`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `send` function is like `write`, but with the additional flags *flags*. The possible values of *flags* are described in Section 16.9.5.3 [Socket Data Options], page 459.

This function returns the number of bytes transmitted, or `-1` on failure. If the socket is nonblocking, then `send` (like `write`) can return after sending just part of the data.

See Section 13.14 [File Status Flags], page 362, for information about nonblocking mode.

Note, however, that a successful return value merely indicates that the message has been sent without error, not necessarily that it has been received without error.

The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file descriptor.

`EINTR` The operation was interrupted by a signal before any data was sent. See Section 24.5 [Primitives Interrupted by Signals], page 685.

`ENOTSOCK` The descriptor *socket* is not a socket.

`EMSGSIZE` The socket type requires that the message be sent atomically, but the message is too large for this to be possible.

`EWOULDBLOCK`
Nonblocking mode has been set on the socket, and the write operation would block. (Normally `send` blocks until the operation can be completed.)

`ENOBUFS` There is not enough internal buffer space available.

`ENOTCONN` You never connected this socket.

`EPIPE` This socket was connected but the connection is now broken. In this case, `send` generates a `SIGPIPE` signal first; if that signal is ignored or blocked, or if its handler returns, then `send` fails with `EPIPE`.

This function is defined as a cancellation point in multi-threaded programs, so one has to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores or whatever) are freed even if the thread is canceled.

16.9.5.2 Receiving Data

The `recv` function is declared in the header file `sys/socket.h`. If your *flags* argument is zero, you can just as well use `read` instead of `recv`; see Section 13.2 [Input and Output Primitives], page 325.

`ssize_t recv` (*int* `socket`, *void* *`buffer`, *size_t* `size`, *int* `flags`) [Function]
Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `recv` function is like `read`, but with the additional flags *flags*. The possible values of *flags* are described in Section 16.9.5.3 [Socket Data Options], page 459.

If nonblocking mode is set for *socket*, and no data are available to be read, `recv` fails immediately rather than waiting. See Section 13.14 [File Status Flags], page 362, for information about nonblocking mode.

This function returns the number of bytes received, or `-1` on failure. The following `errno` error conditions are defined for this function:

`EBADF` The *socket* argument is not a valid file descriptor.

`ENOTSOCK` The descriptor *socket* is not a socket.

`EWOULDBLOCK`
: Nonblocking mode has been set on the socket, and the read operation would block. (Normally, `recv` blocks until there is input available to be read.)

`EINTR`
: The operation was interrupted by a signal before any data was read. See Section 24.5 [Primitives Interrupted by Signals], page 685.

`ENOTCONN`
: You never connected this socket.

This function is defined as a cancellation point in multi-threaded programs, so one has to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores or whatever) are freed even if the thread is canceled.

16.9.5.3 Socket Data Options

The *flags* argument to `send` and `recv` is a bit mask. You can bitwise-OR the values of the following macros together to obtain a value for this argument. All are defined in the header file `sys/socket.h`.

`int MSG_OOB` [Macro]
: Send or receive out-of-band data. See Section 16.9.8 [Out-of-Band Data], page 462.

`int MSG_PEEK` [Macro]
: Look at the data but don't remove it from the input queue. This is only meaningful with input functions such as `recv`, not with `send`.

`int MSG_DONTROUTE` [Macro]
: Don't include routing information in the message. This is only meaningful with output operations, and is usually only of interest for diagnostic or routing programs. We don't try to explain it here.

16.9.6 Byte Stream Socket Example

Here is an example client program that makes a connection for a byte stream socket in the Internet namespace. It doesn't do anything particularly interesting once it has connected to the server; it just sends a text string to the server and exits.

This program uses `init_sockaddr` to set up the socket address; see Section 16.6.7 [Internet Socket Example], page 450.

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

#define PORT            5555
#define MESSAGE         "Yow!!! Are we having fun yet?!?"
#define SERVERHOST      "www.gnu.org"

void
```

```
write_to_server (int filedes)
{
  int nbytes;

  nbytes = write (filedes, MESSAGE, strlen (MESSAGE) + 1);
  if (nbytes < 0)
    {
      perror ("write");
      exit (EXIT_FAILURE);
    }
}


int
main (void)
{
  extern void init_sockaddr (struct sockaddr_in *name,
                             const char *hostname,
                             uint16_t port);
  int sock;
  struct sockaddr_in servername;

  /* Create the socket. */
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
      perror ("socket (client)");
      exit (EXIT_FAILURE);
    }

  /* Connect to the server. */
  init_sockaddr (&servername, SERVERHOST, PORT);
  if (0 > connect (sock,
                   (struct sockaddr *) &servername,
                   sizeof (servername)))
    {
      perror ("connect (client)");
      exit (EXIT_FAILURE);
    }

  /* Send data to the server. */
  write_to_server (sock);
  close (sock);
  exit (EXIT_SUCCESS);
}
```

16.9.7 Byte Stream Connection Server Example

The server end is much more complicated. Since we want to allow multiple clients to be connected to the server at the same time, it would be incorrect to wait for input from a single client by simply calling `read` or `recv`. Instead, the right thing to do is to use `select` (see Section 13.8 [Waiting for Input or Output], page 341) to wait for input on all of the open sockets. This also allows the server to deal with additional connection requests.

This particular server doesn't do anything interesting once it has gotten a message from a client. It does close the socket for that client when it detects an end-of-file condition (resulting from the client shutting down its end of the connection).

This program uses `make_socket` to set up the socket address; see Section 16.6.7 [Internet Socket Example], page 450.

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

#define PORT    5555
#define MAXMSG  512

int
read_from_client (int filedes)
{
  char buffer[MAXMSG];
  int nbytes;

  nbytes = read (filedes, buffer, MAXMSG);
  if (nbytes < 0)
    {
      /* Read error. */
      perror ("read");
      exit (EXIT_FAILURE);
    }
  else if (nbytes == 0)
    /* End-of-file. */
    return -1;
  else
    {
      /* Data read. */
      fprintf (stderr, "Server: got message: `%s'\n", buffer);
      return 0;
    }
}

int
main (void)
{
  extern int make_socket (uint16_t port);
  int sock;
  fd_set active_fd_set, read_fd_set;
  int i;
  struct sockaddr_in clientname;
  size_t size;

  /* Create the socket and set it up to accept connections. */
  sock = make_socket (PORT);
  if (listen (sock, 1) < 0)
    {
      perror ("listen");
      exit (EXIT_FAILURE);
    }

  /* Initialize the set of active sockets. */
```

```
      FD_ZERO (&active_fd_set);
      FD_SET (sock, &active_fd_set);

      while (1)
        {
          /* Block until input arrives on one or more active sockets. */
          read_fd_set = active_fd_set;
          if (select (FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0)
            {
              perror ("select");
              exit (EXIT_FAILURE);
            }

          /* Service all the sockets with input pending. */
          for (i = 0; i < FD_SETSIZE; ++i)
            if (FD_ISSET (i, &read_fd_set))
              {
                if (i == sock)
                  {
                    /* Connection request on original socket. */
                    int new;
                    size = sizeof (clientname);
                    new = accept (sock,
                                  (struct sockaddr *) &clientname,
                                  &size);
                    if (new < 0)
                      {
                        perror ("accept");
                        exit (EXIT_FAILURE);
                      }
                    fprintf (stderr,
                             "Server: connect from host %s, port %hd.\n",
                             inet_ntoa (clientname.sin_addr),
                             ntohs (clientname.sin_port));
                    FD_SET (new, &active_fd_set);
                  }
                else
                  {
                    /* Data arriving on an already-connected socket. */
                    if (read_from_client (i) < 0)
                      {
                        close (i);
                        FD_CLR (i, &active_fd_set);
                      }
                  }
              }
        }
    }
```

16.9.8 Out-of-Band Data

Streams with connections permit *out-of-band* data that is delivered with higher priority than ordinary data. Typically the reason for sending out-of-band data is to send notice of an exceptional condition. To send out-of-band data use `send`, specifying the flag `MSG_OOB` (see Section 16.9.5.1 [Sending Data], page 457).

Out-of-band data are received with higher priority because the receiving process need not read it in sequence; to read the next available out-of-band data, use `recv` with the

`MSG_OOB` flag (see Section 16.9.5.2 [Receiving Data], page 458). Ordinary read operations do not read out-of-band data; they read only ordinary data.

When a socket finds that out-of-band data are on their way, it sends a `SIGURG` signal to the owner process or process group of the socket. You can specify the owner using the `F_SETOWN` command to the `fcntl` function; see Section 13.18 [Interrupt-Driven Input], page 374. You must also establish a handler for this signal, as described in Chapter 24 [Signal Handling], page 659, in order to take appropriate action such as reading the out-of-band data.

Alternatively, you can test for pending out-of-band data, or wait until there is out-of-band data, using the `select` function; it can wait for an exceptional condition on the socket. See Section 13.8 [Waiting for Input or Output], page 341, for more information about `select`.

Notification of out-of-band data (whether with `SIGURG` or with `select`) indicates that out-of-band data are on the way; the data may not actually arrive until later. If you try to read the out-of-band data before it arrives, `recv` fails with an `EWOULDBLOCK` error.

Sending out-of-band data automatically places a "mark" in the stream of ordinary data, showing where in the sequence the out-of-band data "would have been". This is useful when the meaning of out-of-band data is "cancel everything sent so far". Here is how you can test, in the receiving process, whether any ordinary data was sent before the mark:

```
success = ioctl (socket, SIOCATMARK, &atmark);
```

The `integer` variable *atmark* is set to a nonzero value if the socket's read pointer has reached the "mark".

Here's a function to discard any ordinary data preceding the out-of-band mark:

```
int
discard_until_mark (int socket)
{
  while (1)
    {
      /* This is not an arbitrary limit; any size will do.  */
      char buffer[1024];
      int atmark, success;

      /* If we have reached the mark, return.  */
      success = ioctl (socket, SIOCATMARK, &atmark);
      if (success < 0)
        perror ("ioctl");
      if (result)
        return;

      /* Otherwise, read a bunch of ordinary data and discard it.
         This is guaranteed not to read past the mark
         if it starts before the mark.  */
      success = read (socket, buffer, sizeof buffer);
      if (success < 0)
        perror ("read");
    }
}
```

If you don't want to discard the ordinary data preceding the mark, you may need to read some of it anyway, to make room in internal system buffers for the out-of-band data. If you try to read out-of-band data and get an `EWOULDBLOCK` error, try reading some ordinary

data (saving it so that you can use it when you want it) and see if that makes room. Here is an example:

```
struct buffer
{
  char *buf;
  int size;
  struct buffer *next;
};

/* Read the out-of-band data from SOCKET and return it
   as a 'struct buffer', which records the address of the data
   and its size.

   It may be necessary to read some ordinary data
   in order to make room for the out-of-band data.
   If so, the ordinary data are saved as a chain of buffers
   found in the 'next' field of the value.  */

struct buffer *
read_oob (int socket)
{
  struct buffer *tail = 0;
  struct buffer *list = 0;

  while (1)
    {
      /* This is an arbitrary limit.
         Does anyone know how to do this without a limit?  */
#define BUF_SZ 1024
      char *buf = (char *) xmalloc (BUF_SZ);
      int success;
      int atmark;

      /* Try again to read the out-of-band data.  */
      success = recv (socket, buf, BUF_SZ, MSG_OOB);
      if (success >= 0)
        {
          /* We got it, so return it.  */
          struct buffer *link
            = (struct buffer *) xmalloc (sizeof (struct buffer));
          link->buf = buf;
          link->size = success;
          link->next = list;
          return link;
        }

      /* If we fail, see if we are at the mark.  */
      success = ioctl (socket, SIOCATMARK, &atmark);
      if (success < 0)
        perror ("ioctl");
      if (atmark)
        {
          /* At the mark; skipping past more ordinary data cannot help.
             So just wait a while.  */
          sleep (1);
          continue;
        }
```

```
          /* Otherwise, read a bunch of ordinary data and save it.
             This is guaranteed not to read past the mark
             if it starts before the mark.  */
          success = read (socket, buf, BUF_SZ);
          if (success < 0)
            perror ("read");

          /* Save this data in the buffer list.  */
          {
            struct buffer *link
              = (struct buffer *) xmalloc (sizeof (struct buffer));
            link->buf = buf;
            link->size = success;

            /* Add the new link to the end of the list.  */
            if (tail)
              tail->next = link;
            else
              list = link;
            tail = link;
          }
        }
    }
```

16.10 Datagram Socket Operations

This section describes how to use communication styles that don't use connections (styles `SOCK_DGRAM` and `SOCK_RDM`). Using these styles, you group data into packets and each packet is an independent communication. You specify the destination for each packet individually.

Datagram packets are like letters: you send each one independently with its own destination address, and they may arrive in the wrong order or not at all.

The `listen` and `accept` functions are not allowed for sockets using connectionless communication styles.

16.10.1 Sending Datagrams

The normal way of sending data on a datagram socket is by using the `sendto` function, declared in `sys/socket.h`.

You can call `connect` on a datagram socket, but this only specifies a default destination for further data transmission on the socket. When a socket has a default destination you can use `send` (see Section 16.9.5.1 [Sending Data], page 457) or even `write` (see Section 13.2 [Input and Output Primitives], page 325) to send a packet there. You can cancel the default destination by calling `connect` using an address format of `AF_UNSPEC` in the *addr* argument. See Section 16.9.1 [Making a Connection], page 454, for more information about the `connect` function.

`ssize_t sendto` (*int* **`socket`**, *const void* ***`buffer`**, *size_t* **`size`**, *int* **`flags`**, *struct sockaddr* ***`addr`**, *socklen_t* **`length`**) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `sendto` function transmits the data in the *buffer* through the socket *socket* to the destination address specified by the *addr* and *length* arguments. The *size* argument specifies the number of bytes to be transmitted.

The *flags* are interpreted the same way as for `send`; see Section 16.9.5.3 [Socket Data Options], page 459.

The return value and error conditions are also the same as for `send`, but you cannot rely on the system to detect errors and report them; the most common error is that the packet is lost or there is no-one at the specified address to receive it, and the operating system on your machine usually does not know this.

It is also possible for one call to `sendto` to report an error owing to a problem related to a previous call.

This function is defined as a cancellation point in multi-threaded programs, so one has to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores or whatever) are freed even if the thread is canceled.

16.10.2 Receiving Datagrams

The `recvfrom` function reads a packet from a datagram socket and also tells you where it was sent from. This function is declared in `sys/socket.h`.

`ssize_t recvfrom` (*int* **`socket`**, *void* ***`buffer`**, *size_t* **`size`**, *int* **`flags`**, *struct sockaddr* ***`addr`**, *socklen_t* ***`length-ptr`**) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `recvfrom` function reads one packet from the socket *socket* into the buffer *buffer*. The *size* argument specifies the maximum number of bytes to be read.

If the packet is longer than *size* bytes, then you get the first *size* bytes of the packet and the rest of the packet is lost. There's no way to read the rest of the packet. Thus, when you use a packet protocol, you must always know how long a packet to expect.

The *addr* and *length-ptr* arguments are used to return the address where the packet came from. See Section 16.3 [Socket Addresses], page 429. For a socket in the local domain the address information won't be meaningful, since you can't read the address of such a socket (see Section 16.5 [The Local Namespace], page 433). You can specify a null pointer as the *addr* argument if you are not interested in this information.

The *flags* are interpreted the same way as for `recv` (see Section 16.9.5.3 [Socket Data Options], page 459). The return value and error conditions are also the same as for `recv`.

This function is defined as a cancellation point in multi-threaded programs, so one has to be prepared for this and make sure that allocated resources (like memory, files descriptors, semaphores or whatever) are freed even if the thread is canceled.

You can use plain `recv` (see Section 16.9.5.2 [Receiving Data], page 458) instead of `recvfrom` if you don't need to find out who sent the packet (either because you know where it should come from or because you treat all possible senders alike). Even `read` can be used if you don't want to specify *flags* (see Section 13.2 [Input and Output Primitives], page 325).

16.10.3 Datagram Socket Example

Here is a set of example programs that send messages over a datagram stream in the local namespace. Both the client and server programs use the `make_named_socket` function that was presented in Section 16.5.3 [Example of Local-Namespace Sockets], page 434, to create and name their sockets.

First, here is the server program. It sits in a loop waiting for messages to arrive, bouncing each message back to the sender. Obviously this isn't a particularly useful program, but it does show the general ideas involved.

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER  "/tmp/serversocket"
#define MAXMSG  512

int
main (void)
{
  int sock;
  char message[MAXMSG];
  struct sockaddr_un name;
  size_t size;
  int nbytes;

  /* Remove the filename first, it's ok if the call fails */
  unlink (SERVER);

  /* Make the socket, then loop endlessly. */
  sock = make_named_socket (SERVER);
  while (1)
    {
      /* Wait for a datagram. */
      size = sizeof (name);
      nbytes = recvfrom (sock, message, MAXMSG, 0,
                         (struct sockaddr *) & name, &size);
      if (nbytes < 0)
        {
          perror ("recfrom (server)");
          exit (EXIT_FAILURE);
        }

      /* Give a diagnostic message. */
      fprintf (stderr, "Server: got message: %s\n", message);

      /* Bounce the message back to the sender. */
      nbytes = sendto (sock, message, nbytes, 0,
                       (struct sockaddr *) & name, size);
      if (nbytes < 0)
        {
          perror ("sendto (server)");
          exit (EXIT_FAILURE);
        }
    }
```

```
}
```

16.10.4 Example of Reading Datagrams

Here is the client program corresponding to the server above.

It sends a datagram to the server and then waits for a reply. Notice that the socket for the client (as well as for the server) in this example has to be given a name. This is so that the server can direct a message back to the client. Since the socket has no associated connection state, the only way the server can do this is by referencing the name of the client.

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER  "/tmp/serversocket"
#define CLIENT  "/tmp/mysocket"
#define MAXMSG  512
#define MESSAGE "Yow!!! Are we having fun yet?!?"

int
main (void)
{
  extern int make_named_socket (const char *name);
  int sock;
  char message[MAXMSG];
  struct sockaddr_un name;
  size_t size;
  int nbytes;

  /* Make the socket. */
  sock = make_named_socket (CLIENT);

  /* Initialize the server socket address. */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, SERVER);
  size = strlen (name.sun_path) + sizeof (name.sun_family);

  /* Send the datagram. */
  nbytes = sendto (sock, MESSAGE, strlen (MESSAGE) + 1, 0,
                   (struct sockaddr *) & name, size);
  if (nbytes < 0)
    {
      perror ("sendto (client)");
      exit (EXIT_FAILURE);
    }

  /* Wait for a reply. */
  nbytes = recvfrom (sock, message, MAXMSG, 0, NULL, 0);
  if (nbytes < 0)
    {
      perror ("recfrom (client)");
      exit (EXIT_FAILURE);
    }
```

```
  /* Print a diagnostic message. */
  fprintf (stderr, "Client: got message: %s\n", message);

  /* Clean up. */
  remove (CLIENT);
  close (sock);
}
```

Keep in mind that datagram socket communications are unreliable. In this example, the client program waits indefinitely if the message never reaches the server or if the server's response never comes back. It's up to the user running the program to kill and restart it if desired. A more automatic solution could be to use `select` (see Section 13.8 [Waiting for Input or Output], page 341) to establish a timeout period for the reply, and in case of timeout either re-send the message or shut down the socket and exit.

16.11 The `inetd` Daemon

We've explained above how to write a server program that does its own listening. Such a server must already be running in order for anyone to connect to it.

Another way to provide a service on an Internet port is to let the daemon program `inetd` do the listening. `inetd` is a program that runs all the time and waits (using `select`) for messages on a specified set of ports. When it receives a message, it accepts the connection (if the socket style calls for connections) and then forks a child process to run the corresponding server program. You specify the ports and their programs in the file `/etc/inetd.conf`.

16.11.1 `inetd` Servers

Writing a server program to be run by `inetd` is very simple. Each time someone requests a connection to the appropriate port, a new server process starts. The connection already exists at this time; the socket is available as the standard input descriptor and as the standard output descriptor (descriptors 0 and 1) in the server process. Thus the server program can begin reading and writing data right away. Often the program needs only the ordinary I/O facilities; in fact, a general-purpose filter program that knows nothing about sockets can work as a byte stream server run by `inetd`.

You can also use `inetd` for servers that use connectionless communication styles. For these servers, `inetd` does not try to accept a connection since no connection is possible. It just starts the server program, which can read the incoming datagram packet from descriptor 0. The server program can handle one request and then exit, or you can choose to write it to keep reading more requests until no more arrive, and then exit. You must specify which of these two techniques the server uses when you configure `inetd`.

16.11.2 Configuring `inetd`

The file `/etc/inetd.conf` tells `inetd` which ports to listen to and what server programs to run for them. Normally each entry in the file is one line, but you can split it onto multiple lines provided all but the first line of the entry start with whitespace. Lines that start with '`#`' are comments.

Here are two standard entries in `/etc/inetd.conf`:

```
ftp stream tcp nowait root /libexec/ftpd ftpd
talk dgram udp wait root /libexec/talkd talkd
```

An entry has this format:

```
service style protocol wait username program arguments
```

The *service* field says which service this program provides. It should be the name of a service defined in `/etc/services`. `inetd` uses *service* to decide which port to listen on for this entry.

The fields *style* and *protocol* specify the communication style and the protocol to use for the listening socket. The style should be the name of a communication style, converted to lower case and with '`SOCK_`' deleted—for example, '`stream`' or '`dgram`'. *protocol* should be one of the protocols listed in `/etc/protocols`. The typical protocol names are '`tcp`' for byte stream connections and '`udp`' for unreliable datagrams.

The *wait* field should be either '`wait`' or '`nowait`'. Use '`wait`' if *style* is a connectionless style and the server, once started, handles multiple requests as they come in. Use '`nowait`' if `inetd` should start a new process for each message or request that comes in. If *style* uses connections, then *wait* **must** be '`nowait`'.

user is the user name that the server should run as. `inetd` runs as root, so it can set the user ID of its children arbitrarily. It's best to avoid using '`root`' for *user* if you can; but some servers, such as Telnet and FTP, read a username and password themselves. These servers need to be root initially so they can log in as commanded by the data coming over the network.

program together with *arguments* specifies the command to run to start the server. *program* should be an absolute file name specifying the executable file to run. *arguments* consists of any number of whitespace-separated words, which become the command-line arguments of *program*. The first word in *arguments* is argument zero, which should by convention be the program name itself (sans directories).

If you edit `/etc/inetd.conf`, you can tell `inetd` to reread the file and obey its new contents by sending the `inetd` process the `SIGHUP` signal. You'll have to use `ps` to determine the process ID of the `inetd` process as it is not fixed.

16.12 Socket Options

This section describes how to read or set various options that modify the behavior of sockets and their underlying communications protocols.

When you are manipulating a socket option, you must specify which *level* the option pertains to. This describes whether the option applies to the socket interface, or to a lower-level communications protocol interface.

16.12.1 Socket Option Functions

Here are the functions for examining and modifying socket options. They are declared in `sys/socket.h`.

`int getsockopt` (*int* `socket`, *int* `level`, *int* `optname`, *void* *`optval`, *socklen_t* *`optlen-ptr`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getsockopt` function gets information about the value of option *optname* at level *level* for socket *socket*.

The option value is stored in a buffer that *optval* points to. Before the call, you should supply in `*optlen-ptr` the size of this buffer; on return, it contains the number of bytes of information actually stored in the buffer.

Most options interpret the *optval* buffer as a single `int` value.

The actual return value of `getsockopt` is `0` on success and `-1` on failure. The following `errno` error conditions are defined:

`EBADF` The *socket* argument is not a valid file descriptor.

`ENOTSOCK` The descriptor *socket* is not a socket.

`ENOPROTOOPT`
The *optname* doesn't make sense for the given *level*.

`int setsockopt` (*int* `socket`, *int* `level`, *int* `optname`, *const void* `*optval`, *socklen_t* `optlen`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

This function is used to set the socket option *optname* at level *level* for socket *socket*. The value of the option is passed in the buffer *optval* of size *optlen*.

The return value and error codes for `setsockopt` are the same as for `getsockopt`.

16.12.2 Socket-Level Options

`int SOL_SOCKET` [Constant]

Use this constant as the *level* argument to `getsockopt` or `setsockopt` to manipulate the socket-level options described in this section.

Here is a table of socket-level option names; all are defined in the header file `sys/socket.h`.

`SO_DEBUG`

This option toggles recording of debugging information in the underlying protocol modules. The value has type `int`; a nonzero value means "yes".

`SO_REUSEADDR`

This option controls whether `bind` (see Section 16.3.2 [Setting the Address of a Socket], page 431) should permit reuse of local addresses for this socket. If you enable this option, you can actually have two sockets with the same Internet port number; but the system won't allow you to use the two identically-named sockets in a way that would confuse the Internet. The reason for this option is that some higher-level Internet protocols, including FTP, require you to keep reusing the same port number.

The value has type `int`; a nonzero value means "yes".

`SO_KEEPALIVE`

This option controls whether the underlying protocol should periodically transmit messages on a connected socket. If the peer fails to respond to these messages, the connection is considered broken. The value has type `int`; a nonzero value means "yes".

`SO_DONTROUTE`
: This option controls whether outgoing messages bypass the normal message routing facilities. If set, messages are sent directly to the network interface instead. The value has type `int`; a nonzero value means "yes".

`SO_LINGER`
: This option specifies what should happen when the socket of a type that promises reliable delivery still has untransmitted messages when it is closed; see Section 16.8.2 [Closing a Socket], page 452. The value has type `struct linger`.

  `struct linger` [Data Type]
  : This structure type has the following members:

    `int l_onoff`
    : This field is interpreted as a boolean. If nonzero, `close` blocks until the data are transmitted or the timeout period has expired.

    `int l_linger`
    : This specifies the timeout period, in seconds.

`SO_BROADCAST`
: This option controls whether datagrams may be broadcast from the socket. The value has type `int`; a nonzero value means "yes".

`SO_OOBINLINE`
: If this option is set, out-of-band data received on the socket is placed in the normal input queue. This permits it to be read using `read` or `recv` without specifying the `MSG_OOB` flag. See Section 16.9.8 [Out-of-Band Data], page 462. The value has type `int`; a nonzero value means "yes".

`SO_SNDBUF`
: This option gets or sets the size of the output buffer. The value is a `size_t`, which is the size in bytes.

`SO_RCVBUF`
: This option gets or sets the size of the input buffer. The value is a `size_t`, which is the size in bytes.

`SO_STYLE`
`SO_TYPE`
: This option can be used with `getsockopt` only. It is used to get the socket's communication style. `SO_TYPE` is the historical name, and `SO_STYLE` is the preferred name in GNU. The value has type `int` and its value designates a communication style; see Section 16.2 [Communication Styles], page 428.

`SO_ERROR`
: This option can be used with `getsockopt` only. It is used to reset the error status of the socket. The value is an `int`, which represents the previous error status.

16.13 Networks Database

Many systems come with a database that records a list of networks known to the system developer. This is usually kept either in the file `/etc/networks` or in an equivalent from a name server. This data base is useful for routing programs such as `route`, but it is not useful for programs that simply communicate over the network. We provide functions to access this database, which are declared in `netdb.h`.

`struct netent` [Data Type]

This data type is used to represent information about entries in the networks database. It has the following members:

`char *n_name`
: This is the "official" name of the network.

`char **n_aliases`
: These are alternative names for the network, represented as a vector of strings. A null pointer terminates the array.

`int n_addrtype`
: This is the type of the network number; this is always equal to `AF_INET` for Internet networks.

`unsigned long int n_net`
: This is the network number. Network numbers are returned in host byte order; see Section 16.6.5 [Byte Order Conversion], page 447.

Use the `getnetbyname` or `getnetbyaddr` functions to search the networks database for information about a specific network. The information is returned in a statically-allocated structure; you must copy the information if you need to save it.

`struct netent * getnetbyname` (*const char *name*) [Function]

Preliminary: | MT-Unsafe race:netbyname env locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getnetbyname` function returns information about the network named *name*. It returns a null pointer if there is no such network.

`struct netent * getnetbyaddr` (*uint32_t net*, *int type*) [Function]

Preliminary: | MT-Unsafe race:netbyaddr locale | AS-Unsafe dlopen plugin heap lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `getnetbyaddr` function returns information about the network of type *type* with number *net*. You should specify a value of `AF_INET` for the *type* argument for Internet networks.

`getnetbyaddr` returns a null pointer if there is no such network.

You can also scan the networks database using `setnetent`, `getnetent` and `endnetent`. Be careful when using these functions because they are not reentrant.