3

Sockets Introduction

3.1 Introduction

This chapter begins the description of the sockets API. We begin with socket address structures, which will be found in almost every example in the text. These structures can be passed in two directions: from the process to the kernel, and from the kernel to the process. The latter case is an example of a value-result argument, and we will encounter other examples of these arguments throughout the text.

The address conversion functions convert between a text representation of an address and the binary value that goes into a socket address structure. Most existing IPv4 code uses `inet_addr` and `inet_ntoa`, but two new functions, `inet_pton` and `inet_ntop`, handle both IPv4 and IPv6.

One problem with these address conversion functions is that they are dependent on the type of address being converted: IPv4 or IPv6. We will develop a set of functions whose names begin with `sock_` that work with socket address structures in a protocol-independent fashion. We will use these throughout the text to make our code protocol-independent.

3.2 Socket Address Structures

Most socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure. The names of these structures begin with `sockaddr_` and end with a unique suffix for each protocol suite.

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named `sockaddr_in` and is defined by including the `<netinet/in.h>` header. Figure 3.1 shows the POSIX definition.

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

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

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

There are several points we need to make about socket address structures in general using this example:

- The length member, `sin_len`, was added with 4.3BSD-Reno, when support for the OSI protocols was added (Figure 1.15). Before this release, the first member was `sin_family`, which was historically an `unsigned short`. Not all vendors support a length field for socket address structures and the POSIX specification does not require this member. The datatype that we show, `uint8_t`, is typical, and POSIX-compliant systems provide datatypes of this form (Figure 3.2).

  Having a length field simplifies the handling of variable-length socket address structures.

- Even if the length field is present, we need never set it and need never examine it, unless we are dealing with routing sockets (Chapter 18). It is used within the kernel by the routines that deal with socket address structures from various protocol families (e.g., the routing table code).

  The four socket functions that pass a socket address structure from the process to the kernel, `bind`, `connect`, `sendto`, and `sendmsg`, all go through the `sockargs` function in a Berkeley-derived implementation (p. 452 of TCPv2). This function copies the socket address structure from the process and explicitly sets its `sin_len` member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, `accept`, `recvfrom`, `recvmsg`, `getpeername`, and `getsockname`, all set the `sin_len` member before returning to the process.

  Unfortunately, there is normally no simple compile-time test to determine whether an implementation defines a length field for its socket address structures. In our code, we test our own `HAVE_SOCKADDR_SA_LEN` constant (Figure D.2), but whether to define this constant or not requires trying to compile a simple test program that uses this optional structure member and seeing if the compilation succeeds or not. We will see in Figure 3.4 that IPv6 implementations are required to define `SIN6_LEN` if socket address structures have a length field. Some IPv4 implementations provide the length field of the socket address structure to the application based on a compile-time option (e.g., `_SOCKADDR_LEN`). This feature provides compatibility for older programs.

- The POSIX specification requires only three members in the structure: `sin_family`, `sin_addr`, and `sin_port`. It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the `sin_zero` member so that all socket address structures are at least 16 bytes in size.

- We show the POSIX datatypes for the `s_addr`, `sin_family`, and `sin_port` members. The `in_addr_t` datatype must be an unsigned integer type of at least 32 bits, `in_port_t` must be an unsigned integer type of at least 16 bits, and `sa_family_t` can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported. Figure 3.2 lists these three POSIX-defined datatypes, along with some other POSIX datatypes that we will encounter.

Datatype	Description	Header
`int8_t`	Signed 8-bit integer	`<sys/types.h>`
`uint8_t`	Unsigned 8-bit integer	`<sys/types.h>`
`int16_t`	Signed 16-bit integer	`<sys/types.h>`
`uint16_t`	Unsigned 16-bit integer	`<sys/types.h>`
`int32_t`	Signed 32-bit integer	`<sys/types.h>`
`uint32_t`	Unsigned 32-bit integer	`<sys/types.h>`
`sa_family_t`	Address family of socket address structure	`<sys/socket.h>`
`socklen_t`	Length of socket address structure, normally `uint32_t`	`<sys/socket.h>`
`in_addr_t`	IPv4 address, normally `uint32_t`	`<netinet/in.h>`
`in_port_t`	TCP or UDP port, normally `uint16_t`	`<netinet/in.h>`

Figure 3.2 Datatypes required by the POSIX specification.

- You will also encounter the datatypes `u_char`, `u_short`, `u_int`, and `u_long`, which are all unsigned. The POSIX specification defines these with a note that they are obsolete. They are provided for backward compatibility.

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order. We must be cognizant of this when using these members. We will say more about the difference between host byte order and network byte order in Section 3.4.
- The 32-bit IPv4 address can be accessed in two different ways. For example, if `serv` is defined as an Internet socket address structure, then `serv.sin_addr` references

the 32-bit IPv4 address as an in_addr structure, while serv.sin_addr.s_addr references the same 32-bit IPv4 address as an in_addr_t (typically an unsigned 32-bit integer). We must be certain that we are referencing the IPv4 address correctly, especially when it is used as an argument to a function, because compilers often pass structures differently from integers.

The reason the sin_addr member is a structure, and not just an in_addr_t, is historical. Earlier releases (4.2BSD) defined the in_addr structure as a union of various structures, to allow access to each of the 4 bytes and to both of the 16-bit values contained within the 32-bit IPv4 address. This was used with class A, B, and C addresses to fetch the appropriate bytes of the address. But with the advent of subnetting and then the disappearance of the various address classes with classless addressing (Section A.4), the need for the union disappeared. Most systems today have done away with the union and just define in_addr as a structure with a single in_addr_t member.

- The sin_zero member is unused, but we *always* set it to 0 when filling in one of these structures. By convention, we always set the entire structure to 0 before filling it in, not just the sin_zero member.

Although most uses of the structure do not require that this member be 0, when binding a non-wildcard IPv4 address, this member must be 0 (pp. 731–732 of TCPv2).

- Socket address structures are used only on a given host: The structure itself is not communicated between different hosts, although certain fields (e.g., the IP address and port) are used for communication.

Generic Socket Address Structure

A socket address structures is *always* passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from *any* of the supported protocol families.

A problem arises in how to declare the type of pointer that is passed. With ANSI C, the solution is simple: void * is the generic pointer type. But, the socket functions predate ANSI C and the solution chosen in 1982 was to define a *generic* socket address structure in the <sys/socket.h> header, which we show in Figure 3.3.

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

Figure 3.3 The generic socket address structure: sockaddr.

The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the bind function:

```
int bind(int, struct sockaddr *, socklen_t);
```

This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure. For example,

```
struct sockaddr_in   serv;     /* IPv4 socket address structure */

/* fill in serv{} */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

If we omit the cast "(struct sockaddr *)," the C compiler generates a warning of the form "warning: passing arg 2 of 'bind' from incompatible pointer type," assuming the system's headers have an ANSI C prototype for the bind function.

From an application programmer's point of view, the *only* use of these generic socket address structures is to cast pointers to protocol-specific structures.

Recall in Section 1.2 that in our unp.h header, we define SA to be the string "struct sockaddr" just to shorten the code that we must write to cast these pointers.

From the kernel's perspective, another reason for using pointers to generic socket address structures as arguments is that the kernel must take the caller's pointer, cast it to a struct sockaddr *, and then look at the value of sa_family to determine the type of the structure. But from an application programmer's perspective, it would be simpler if the pointer type was void *, omitting the need for the explicit cast.

IPv6 Socket Address Structure

The IPv6 socket address is defined by including the <netinet/in.h> header, and we show it in Figure 3.4.

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (28) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

Figure 3.4 IPv6 socket address structure: sockaddr_in6.

The extensions to the sockets API for IPv6 are defined in RFC 3493 [Gilligan et al. 2003].

Note the following points about Figure 3.4:

- The SIN6_LEN constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is AF_INET6, whereas the IPv4 family is AF_INET.
- The members in this structure are ordered so that if the sockaddr_in6 structure is 64-bit aligned, so is the 128-bit sin6_addr member. On some 64-bit processors, data accesses of 64-bit values are optimized if stored on a 64-bit boundary.
- The sin6_flowinfo member is divided into two fields:
  - The low-order 20 bits are the flow label
  - The high-order 12 bits are reserved

  The flow label field is described with Figure A.2. The use of the flow label field is still a research topic.
- The sin6_scope_id identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address (Section A.5).

New Generic Socket Address Structure

A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct sockaddr. Unlike the struct sockaddr, the new struct sockaddr_storage is large enough to hold any socket address type supported by the system. The sockaddr_storage structure is defined by including the <netinet/in.h> header, which we show in Figure 3.5.

```
struct sockaddr_storage {
  uint8_t      ss_len;       /* length of this struct (implementation dependent) */
  sa_family_t  ss_family;    /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *    all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   *    system supports.
   */
};
```

Figure 3.5 The storage socket address structure: sockaddr_storage.

The sockaddr_storage type provides a generic socket address structure that is different from struct sockaddr in two ways:

a) If any socket address structures that the system supports have alignment requirements, the sockaddr_storage provides the strictest alignment requirement.

b) The `sockaddr_storage` is large enough to contain any socket address structure that the system supports.

Note that the fields of the `sockaddr_storage` structure are opaque to the user, except for `ss_family` and `ss_len` (if present). The `sockaddr_storage` must be cast or copied to the appropriate socket address structure for the address given in `ss_family` to access any other fields.

Comparison of Socket Address Structures

Figure 3.6 shows a comparison of the five socket address structures that we will encounter in this text: IPv4, IPv6, Unix domain (Figure 15.1), datalink (Figure 18.1), and storage. In this figure, we assume that the socket address structures all contain a one-byte length field, that the family field also occupies one byte, and that any field that must be at least some number of bits is exactly that number of bits.

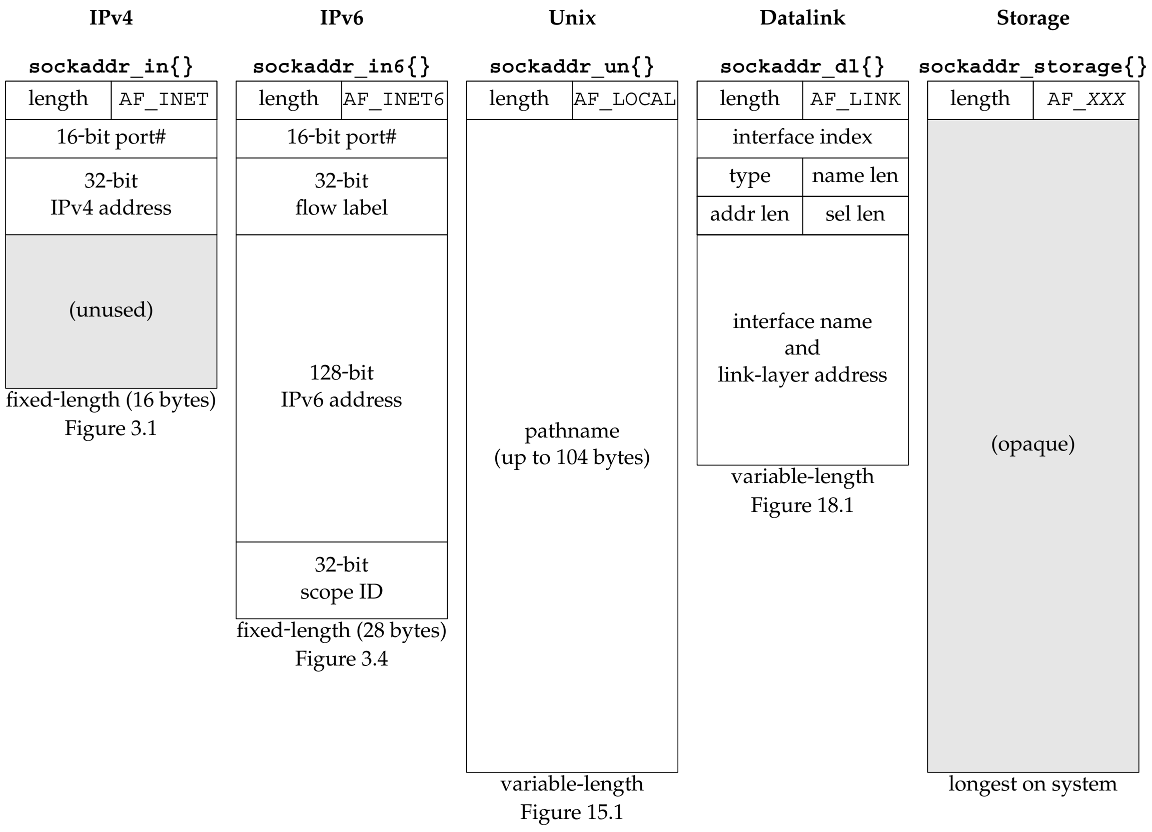

Figure 3.6 Comparison of various socket address structures.

Two of the socket address structures are fixed-length, while the Unix domain structure and the datalink structure are variable-length. To handle variable-length structures, whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument. We show the size in bytes (for the 4.4BSD implementation) of the fixed-length structures beneath each structure.

The `sockaddr_un` structure itself is not variable-length (Figure 15.1), but the amount of information—the pathname within the structure—is variable-length. When passing pointers to these structures, we must be careful how we handle the length field, both the length field in the socket address structure itself (if supported by the implementation) and the length to and from the kernel.

This figure shows the style that we follow throughout the text: structure names are always shown in a bolder font, followed by braces, as in **`sockaddr_in{}`**.

We noted earlier that the length field was added to all the socket address structures with the 4.3BSD Reno release. Had the length field been present with the original release of sockets, there would be no need for the length argument to all the socket functions: the third argument to `bind` and `connect`, for example. Instead, the size of the structure could be contained in the length field of the structure.

3.3 Value-Result Arguments

We mentioned that when a socket address structure is passed to any socket function, it is always passed by reference. That is, a pointer to the structure is passed. The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed: from the process to the kernel, or vice versa.

1. Three functions, `bind`, `connect`, and `sendto`, pass a socket address structure from the process to the kernel. One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in

   ```
   struct sockaddr_in  serv;

   /* fill in serv{} */
   connect(sockfd, (SA *) &serv, sizeof(serv));
   ```

   Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel. Figure 3.7 shows this scenario.

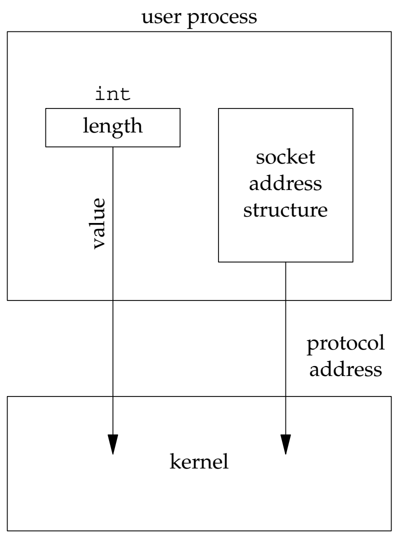

Figure 3.7 Socket address structure passed from process to kernel.

We will see in the next chapter that the datatype for the size of a socket address structure is actually `socklen_t` and not `int`, but the POSIX specification recommends that `socklen_t` be defined as `uint32_t`.

2. Four functions, `accept`, `recvfrom`, `getsockname`, and `getpeername`, pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario. Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure, as in

```
struct sockaddr_un  cli;   /* Unix domain */
socklen_t  len;

len = sizeof(cli);         /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

The reason that the size changes from an integer to be a pointer to an integer is because the size is both a *value* when the function is called (it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in) and a *result* when the function returns (it tells the process how much information the kernel actually stored in the structure). This type of argument is called a *value-result* argument. Figure 3.8 shows this scenario.

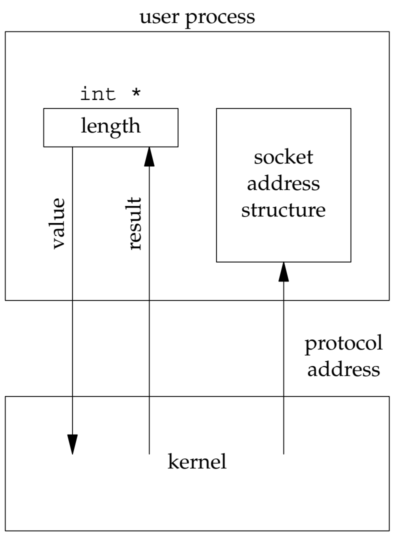

Figure 3.8 Socket address structure passed from kernel to process.

We will see an example of value-result arguments in Figure 4.11.

We have been talking about socket address structures being passed between the process and the kernel. For an implementation such as 4.4BSD, where all the socket functions are system calls within the kernel, this is correct. But in some implementations, notably System V, socket functions are just library functions that execute as part of a normal user process. How these functions interface with the protocol stack in the kernel is an implementation detail that normally does not affect us. Nevertheless, for simplicity, we will continue to talk about these structures as being passed between the process and the kernel by functions such as `bind` and `connect`. (We will see in Section C.1 that System V implementations do indeed pass socket address structures between processes and the kernel, but as part of STREAMS messages.)

Two other functions pass socket address structures: `recvmsg` and `sendmsg` (Section 14.5). But, we will see that the length field is not a function argument but a structure member.

When using value-result arguments for the length of socket address structures, if the socket address structure is fixed-length (Figure 3.6), the value returned by the kernel will always be that fixed size: 16 for an IPv4 `sockaddr_in` and 28 for an IPv6 `sockaddr_in6`, for example. But with a variable-length socket address structure (e.g., a Unix domain `sockaddr_un`), the value returned can be less than the maximum size of the structure (as we will see with Figure 15.2).

With network programming, the most common example of a value-result argument is the length of a returned socket address structure. But, we will encounter other value-result arguments in this text:

- The middle three arguments for the `select` function (Section 6.3)
- The length argument for the `getsockopt` function (Section 7.2)

- The `msg_namelen` and `msg_controllen` members of the `msghdr` structure, when used with `recvmsg` (Section 14.5)
- The `ifc_len` member of the `ifconf` structure (Figure 17.2)
- The first of the two length arguments for the `sysctl` function (Section 18.4)

3.4 Byte Ordering Functions

Consider a 16-bit integer that is made up of 2 bytes. There are two ways to store the two bytes in memory: with the low-order byte at the starting address, known as *little-endian* byte order, or with the high-order byte at the starting address, known as *big-endian* byte order. We show these two formats in Figure 3.9.

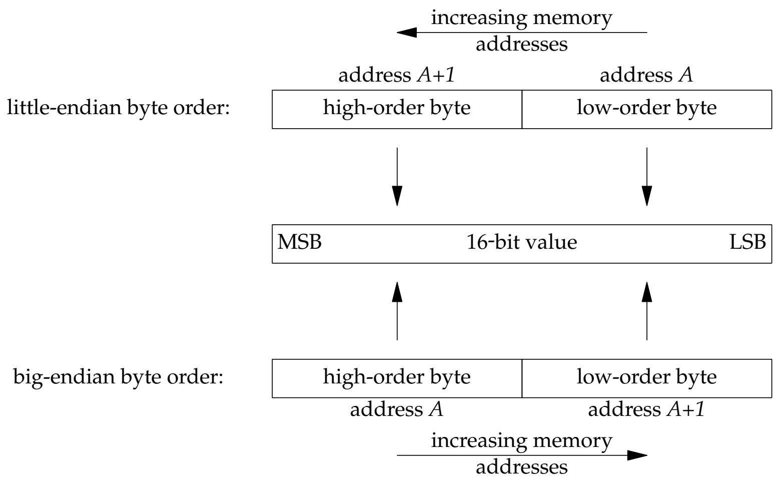

Figure 3.9 Little-endian byte order and big-endian byte order for a 16-bit integer.

In this figure, we show increasing memory addresses going from right to left in the top, and from left to right in the bottom. We also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.

The terms "little-endian" and "big-endian" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

Unfortunately, there is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the *host byte order*. The program shown in Figure 3.10 prints the host byte order.

intro/byteorder.c

```
#include    "unp.h"

int
main(int argc, char **argv)
{
    union {
        short   s;
        char    c[sizeof(short)];
    } un;

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

    exit(0);
}
```

intro/byteorder.c

Figure 3.10 Program to determine host byte order.

We store the two-byte value `0x0102` in the short integer and then look at the two consecutive bytes, `c[0]` (the address *A* in Figure 3.9) and `c[1]` (the address *A+1* in Figure 3.9), to determine the byte order.

The string `CPU_VENDOR_OS` is determined by the GNU `autoconf` program when the software in this book is configured, and it identifies the CPU type, vendor, and OS release. We show some examples here in the output from this program when run on the various systems in Figure 1.16.

```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian

macosx % byteorder
powerpc-apple-darwin6.6: big-endian

freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian

aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian

hpux % byteorder
hppa1.1-hp-hpux11.11: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.9: big-endian
```

We have talked about the byte ordering of a 16-bit integer; obviously, the same discussion applies to a 32-bit integer.

There are currently a variety of systems that can change between little-endian and big-endian byte ordering, sometimes at system reset, sometimes at run-time.

We must deal with these byte ordering differences as network programmers because networking protocols must specify a *network byte order*. For example, in a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multibyte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multibyte integers.

In theory, an implementation could store the fields in a socket address structure in host byte order and then convert to and from the network byte order when moving the fields to and from the protocol headers, saving us from having to worry about this detail. But, both history and the POSIX specification say that certain fields in the socket address structures must be maintained in network byte order. Our concern is therefore converting between host byte order and network byte order. We use the following four functions to convert between these two byte orders.

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

uint16_t htons(uint16_t host16bitvalue);

uint32_t htonl(uint32_t host32bitvalue);

                                Both return: value in network byte order

uint16_t ntohs(uint16_t net16bitvalue);

uint32_t ntohl(uint32_t net32bitvalue);

                                   Both return: value in host byte order
```

In the names of these functions, `h` stands for *host*, `n` stands for *network*, `s` stands for *short*, and `l` stands for *long*. The terms "short" and "long" are historical artifacts from the Digital VAX implementation of 4.2BSD. We should instead think of `s` as a 16-bit value (such as a TCP or UDP port number) and `l` as a 32-bit value (such as an IPv4 address). Indeed, on the 64-bit Digital Alpha, a long integer occupies 64 bits, yet the `htonl` and `ntohl` functions operate on 32-bit values.

When using these functions, we do not care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. What we must do is call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big-endian), these four functions are usually defined as null macros.

We will talk more about the byte ordering problem, with respect to the data contained in a network packet as opposed to the fields in the protocol headers, in Section 5.18 and Exercise 5.8.

We have not yet defined the term "byte." We use the term to mean an 8-bit quantity since almost all current computer systems use 8-bit bytes. Most Internet standards use the term *octet* instead of byte to mean an 8-bit quantity. This started in the early days of TCP/IP because much of the early work was done on systems such as the DEC-10, which did not use 8-bit bytes.

Another important convention in Internet standards is bit ordering. In many Internet standards, you will see "pictures" of packets that look similar to the following (this is the first 32 bits of the IPv4 header from RFC 791):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This represents four bytes in the order in which they appear on the wire; the leftmost bit is the most significant. However, the numbering starts with zero assigned to the most significant bit. This is a notation that you should become familiar with to make it easier to read protocol definitions in RFCs.

A common network programming error in the 1980s was to develop code on Sun workstations (big-endian Motorola 68000s) and forget to call any of these four functions. The code worked fine on these workstations, but would not work when ported to little-endian machines (such as VAXes).

3.5 Byte Manipulation Functions

There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string. We need these types of functions when dealing with socket address structures because we need to manipulate fields such as IP addresses, which can contain bytes of 0, but are not C character strings. The functions beginning with `str` (for string), defined by including the `<string.h>` header, deal with null-terminated C character strings.

The first group of functions, whose names begin with `b` (for byte), are from 4.2BSD and are still provided by almost any system that supports the socket functions. The second group of functions, whose names begin with `mem` (for memory), are from the ANSI C standard and are provided with any system that supports an ANSI C library.

We first show the Berkeley-derived functions, although the only one we use in this text is `bzero`. (We use it because it has only two arguments and is easier to remember than the three-argument `memset` function, as explained on p. 8.) You may encounter the other two functions, `bcopy` and `bcmp`, in existing applications.

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                                    Returns: 0 if equal, nonzero if unequal
```

This is our first encounter with the ANSI C `const` qualifier. In the three uses here, it indicates that what is pointed to by the pointer with this qualification, *src*, *ptr1*, and *ptr2*, is not modified by the function. Worded another way, the memory pointed to by the `const` pointer is read but not modified by the function.

`bzero` sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0. `bcopy` moves the specified number of bytes from the source to the destination. `bcmp` compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

The following functions are the ANSI C functions:

```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

                        Returns: 0 if equal, <0 or >0 if unequal (see text)
```

`memset` sets the specified number of bytes to the value *c* in the destination. `memcpy` is similar to `bcopy`, but the order of the two pointer arguments is swapped. `bcopy` correctly handles overlapping fields, while the behavior of `memcpy` is undefined if the source and destination overlap. The ANSI C `memmove` function must be used when the fields overlap.

One way to remember the order of the two pointers for `memcpy` is to remember that they are written in the same left-to-right order as an assignment statement in C:

```
dest = src;
```

One way to remember the order of the final two arguments to `memset` is to realize that all of the ANSI C `mem`*XXX* functions require a length argument, and it is always the final argument.

`memcmp` compares two arbitrary byte strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending on whether the first unequal byte pointed to by *ptr1* is greater than or less than the corresponding byte pointed to by *ptr2*. The comparison is done assuming the two unequal bytes are `unsigned chars`.

3.6 `inet_aton`, `inet_addr`, and `inet_ntoa` Functions

We will describe two groups of address conversion functions in this section and the next. They convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

1. `inet_aton`, `inet_ntoa`, and `inet_addr` convert an IPv4 address from a dotted-decimal string (e.g., `"206.168.112.96"`) to its 32-bit network byte ordered binary value. You will probably encounter these functions in lots of existing code.
2. The newer functions, `inet_pton` and `inet_ntop`, handle both IPv4 and IPv6 addresses. We describe these two functions in the next section and use them throughout the text.

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

int inet_aton(const char *strptr, struct in_addr *addrptr);

                                  Returns: 1 if string was valid, 0 on error

in_addr_t inet_addr(const char *strptr);

          Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error

char *inet_ntoa(struct in_addr inaddr);

                                  Returns: pointer to dotted-decimal string
```

The first of these, `inet_aton`, converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*. If successful, 1 is returned; otherwise, 0 is returned.

An undocumented feature of `inet_aton` is that if *addrptr* is a null pointer, the function still performs its validation of the input string but does not store any result.

`inet_addr` does the same conversion, returning the 32-bit binary network byte ordered value as the return value. The problem with this function is that all 2^{32} possible binary values are valid IP addresses (0.0.0.0 through 255.255.255.255), but the function returns the constant `INADDR_NONE` (typically 32 one-bits) on an error. This means the dotted-decimal string 255.255.255.255 (the IPv4 limited broadcast address, Section 20.2) cannot be handled by this function since its binary value appears to indicate failure of the function.

A potential problem with `inet_addr` is that some man pages state that it returns –1 on an error, instead of `INADDR_NONE`. This can lead to problems, depending on the C compiler, when comparing the return value of the function (an unsigned value) to a negative constant.

Today, `inet_addr` is deprecated and any new code should use `inet_aton` instead. Better still is to use the newer functions described in the next section, which handle both IPv4 and IPv6.

The `inet_ntoa` function converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string. The string pointed to by the return value of the function resides in static memory. This means the function is not reentrant, which we will discuss in Section 11.18. Finally, notice that this function takes a structure as its argument, not a pointer to a structure.

> Functions that take actual structures as arguments are rare. It is more common to pass a pointer to the structure.

3.7 `inet_pton` and `inet_ntop` Functions

These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses. We use these two functions throughout the text. The letters "p" and "n" stand for *presentation* and *numeric*. The presentation format for an address is often an ASCII string and the numeric format is the binary value that goes into a socket address structure.

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

int inet_pton(int family, const char *strptr, void *addrptr);

                         Returns: 1 if OK, 0 if input not a valid presentation format, –1 on error

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);

                                                Returns: pointer to result if OK, NULL on error
```

The *family* argument for both functions is either `AF_INET` or `AF_INET6`. If *family* is not supported, both functions return an error with `errno` set to `EAFNOSUPPORT`.

The first function tries to convert the string pointed to by *strptr*, storing the binary result through the pointer *addrptr*. If successful, the return value is 1. If the input string is not a valid presentation format for the specified *family*, 0 is returned.

`inet_ntop` does the reverse conversion, from numeric (*addrptr*) to presentation (*strptr*). The *len* argument is the size of the destination, to prevent the function from overflowing the caller's buffer. To help specify this size, the following two definitions are defined by including the `<netinet/in.h>` header:

```
#define  INET_ADDRSTRLEN    16    /* for IPv4 dotted-decimal */
#define  INET6_ADDRSTRLEN   46    /* for IPv6 hex string */
```

If *len* is too small to hold the resulting presentation format, including the terminating null, a null pointer is returned and `errno` is set to `ENOSPC`.

The *strptr* argument to `inet_ntop` cannot be a null pointer. The caller must allocate memory for the destination and specify its size. On success, this pointer is the return value of the function.

Figure 3.11 summarizes the five functions that we have described in this section and the previous section.

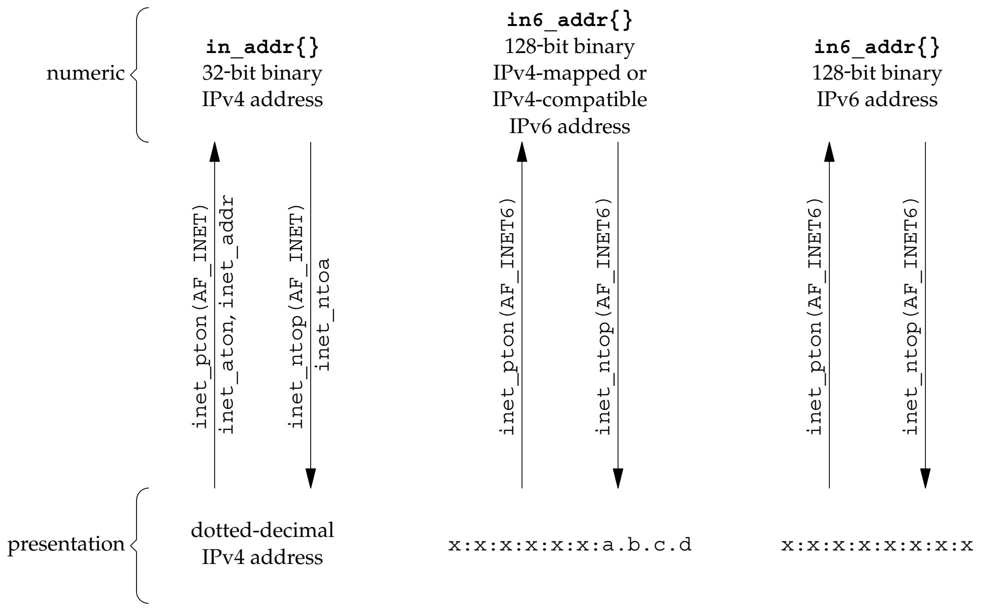

Figure 3.11 Summary of address conversion functions.

Example

Even if your system does not yet include support for IPv6, you can start using these newer functions by replacing calls of the form

```
foo.sin_addr.s_addr = inet_addr(cp);
```

with

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

and replacing calls of the form

```
ptr = inet_ntoa(foo.sin_addr);
```

with

```
char  str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

Figure 3.12 shows a simple definition of `inet_pton` that supports only IPv4. Similarly, Figure 3.13 shows a simple version of `inet_ntop` that supports only IPv4.

libfree/inet_pton_ipv4.c

```
int
inet_pton(int family, const char *strptr, void *addrptr)
{
    if (family == AF_INET) {
        struct in_addr in_val;

        if (inet_aton(strptr, &in_val)) {
            memcpy(addrptr, &in_val, sizeof(struct in_addr));
            return (1);
        }
        return (0);
    }
    errno = EAFNOSUPPORT;
    return (-1);
}
```

libfree/inet_pton_ipv4.c

Figure 3.12 Simple version of `inet_pton` that supports only IPv4.

libfree/inet_ntop_ipv4.c

```
const char *
inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
{
    const u_char *p = (const u_char *) addrptr;

    if (family == AF_INET) {
        char    temp[INET_ADDRSTRLEN];

        snprintf(temp, sizeof(temp), "%d.%d.%d.%d", p[0], p[1], p[2], p[3]);
        if (strlen(temp) >= len) {
            errno = ENOSPC;
            return (NULL);
        }
        strcpy(strptr, temp);
        return (strptr);
    }
    errno = EAFNOSUPPORT;
    return (NULL);
}
```

libfree/inet_ntop_ipv4.c

Figure 3.13 Simple version of `inet_ntop` that supports only IPv4.

3.8 `sock_ntop` and Related Functions

A basic problem with `inet_ntop` is that it requires the caller to pass a pointer to a binary address. This address is normally contained in a socket address structure, requiring the caller to know the format of the structure and the address family. That is, to use it, we must write code of the form

```
struct sockaddr_in   addr;

inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));
```

for IPv4, or

```
struct sockaddr_in6   addr6;

inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

for IPv6. This makes our code protocol-dependent.

To solve this, we will write our own function named `sock_ntop` that takes a pointer to a socket address structure, looks inside the structure, and calls the appropriate function to return the presentation format of the address.

```
#include "unp.h"

char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);

                                    Returns: non-null pointer if OK, NULL on error
```

This is the notation we use for functions of our own (nonstandard system functions) that we use throughout the book: the box around the function prototype and return value is dashed. The header is included at the beginning is usually our `unp.h` header.

sockaddr points to a socket address structure whose length is *addrlen*. The function uses its own static buffer to hold the result and a pointer to this buffer is the return value.

Notice that using static storage for the result prevents the function from being *re-entrant* or *thread-safe*. We will talk more about this in Section 11.18. We made this design decision for this function to allow us to easily call it from the simple examples in the book.

The presentation format is the dotted-decimal form of an IPv4 address or the hex string form of an IPv6 address surrounded by brackets, followed by a terminator (we use a colon, similar to URL syntax), followed by the decimal port number, followed by a null character. Hence, the buffer size must be at least `INET_ADDRSTRLEN` plus 6 bytes for IPv4 (16 + 6 = 22), or `INET6_ADDRSTRLEN` plus 8 bytes for IPv6 (46 + 8 = 54).

We show the source code for only the `AF_INET case` in Figure 3.14.

lib/sock_ntop.c

```
char *
sock_ntop(const struct sockaddr *sa, socklen_t salen)
{
    char    portstr[8];
    static char str[128];       /* Unix domain is largest */

    switch (sa->sa_family) {
    case AF_INET:{
            struct sockaddr_in *sin = (struct sockaddr_in *) sa;

            if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
                return (NULL);
            if (ntohs(sin->sin_port) != 0) {
                snprintf(portstr, sizeof(portstr), ":%d",
                         ntohs(sin->sin_port));
                strcat(str, portstr);
            }
            return (str);
        }
```

lib/sock_ntop.c

Figure 3.14 Our sock_ntop function.

There are a few other functions that we define to operate on socket address structures, and these will simplify the portability of our code between IPv4 and IPv6.

```
#include "unp.h"

int sock_bind_wild(int sockfd, int family);
```

Returns: 0 if OK, –1 on error

```
int sock_cmp_addr(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Returns: 0 if addresses are of the same family and equal, else nonzero

```
int sock_cmp_port(const struct sockaddr *sockaddr1,
                  const struct sockaddr *sockaddr2, socklen_t addrlen);
```

Returns: 0 if addresses are of the same family and ports are equal, else nonzero

```
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Returns: non-negative port number for IPv4 or IPv6 address, else –1

```
char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);
```

Returns: non-null pointer if OK, NULL on error

```
void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen, void *ptr);

void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen, int port);

void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

sock_bind_wild binds the wildcard address and an ephemeral port to a socket. sock_cmp_addr compares the address portion of two socket address structures, and sock_cmp_port compares the port number of two socket address structures. sock_get_port returns just the port number, and sock_ntop_host converts just the host portion of a socket address structure to presentation format (not the port number). sock_set_addr sets just the address portion of a socket address structure to the value pointed to by *ptr*, and sock_set_port sets just the port number of a socket address structure. sock_set_wild sets the address portion of a socket address structure to the wildcard. As with all the functions in the text, we provide a wrapper function whose name begins with "S" for all of these functions that return values other than void and normally call the wrapper function from our programs. We do not show the source code for all these functions, but it is freely available (see the Preface).

3.9 readn, writen, and readline Functions

Stream sockets (e.g., TCP sockets) exhibit a behavior with the read and write functions that differs from normal file I/O. A read or write on a stream socket might input or output fewer bytes than requested, but this is not an error condition. The reason is that buffer limits might be reached for the socket in the kernel. All that is required to input or output the remaining bytes is for the caller to invoke the read or write function again. Some versions of Unix also exhibit this behavior when writing more than 4,096 bytes to a pipe. This scenario is always a possibility on a stream socket with read, but is normally seen with write only if the socket is nonblocking. Nevertheless, we always call our writen function instead of write, in case the implementation returns a short count.

We provide the following three functions that we use whenever we read from or write to a stream socket:

```
#include "unp.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t writen(int filedes, const void *buff, size_t nbytes);

ssize_t readline(int filedes, void *buff, size_t maxlen);

                    All return: number of bytes read or written, –1 on error
```

Figure 3.15 shows the readn function, Figure 3.16 shows the writen function, and Figure 3.17 shows the readline function.

lib/readn.c

```
#include    "unp.h"

ssize_t                         /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char   *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;      /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

lib/readn.c

Figure 3.15 readn function: Read *n* bytes from a descriptor.

lib/writen.c

```
#include    "unp.h"

ssize_t                         /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;   /* and call write() again */
            else
                return (-1);    /* error */
        }

        nleft -= nwritten;
        ptr += nwritten;
    }
    return (n);
}
```

lib/writen.c

Figure 3.16 writen function: Write *n* bytes to a descriptor.

test/readline1.c

```
#include    "unp.h"

/* PAINFULLY SLOW VERSION -- example only */
ssize_t
readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char    c, *ptr;

    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
      again:
        if ( (rc = read(fd, &c, 1)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;          /* newline is stored, like fgets() */
        } else if (rc == 0) {
            *ptr = 0;
            return (n - 1);     /* EOF, n - 1 bytes were read */
        } else {
            if (errno == EINTR)
                goto again;
            return (-1);        /* error, errno set by read() */
        }
    }

    *ptr = 0;                   /* null terminate like fgets() */
    return (n);
}
```

test/readline1.c

Figure 3.17 `readline` function: Read a text line from a descriptor, one byte at a time.

Our three functions look for the error `EINTR` (the system call was interrupted by a caught signal, which we will discuss in more detail in Section 5.9) and continue reading or writing if the error occurs. We handle the error here, instead of forcing the caller to call `readn` or `writen` again, since the purpose of these three functions is to prevent the caller from having to handle a short count.

In Section 14.3, we will mention that the `MSG_WAITALL` flag can be used with the `recv` function to replace the need for a separate `readn` function.

Note that our `readline` function calls the system's `read` function once for every byte of data. This is very inefficient, and why we've commented the code to state it is "PAINFULLY SLOW." When faced with the desire to read lines from a socket, it is quite tempting to turn to the standard I/O library (referred to as "stdio"). We will discuss this approach at length in Section 14.8, but it can be a dangerous path. The same stdio buffering that solves this performance problem creates numerous logistical problems that can lead to well-hidden bugs in your application. The reason is that the state of the stdio buffers is not exposed. To explain this further, consider a line-based protocol between a client and a server, where several clients and servers using that protocol may be implemented over time (really quite common; for example, there are many Web browsers and Web servers independently written to the HTTP specification). Good "defensive programming" techniques require these programs to not only expect their counterparts to follow the network protocol, but to check for unexpected network traffic as well. Such protocol violations should be reported as errors so that bugs are noticed and fixed (and malicious attempts are detected as well), and also so that network applications can recover from problem traffic and continue working if possible. Using stdio to buffer data for performance flies in the face of these goals since the application has no way to tell if unexpected data is being held in the stdio buffers at any given time.

There are many line-based network protocols such as SMTP, HTTP, the FTP control connection protocol, and finger. So, the desire to operate on lines comes up again and again. But our advice is to think in terms of buffers and not lines. Write your code to read buffers of data, and if a line is expected, check the buffer to see if it contains that line.

Figure 3.18 shows a faster version of the `readline` function, which uses its own buffering rather than stdio buffering. Most importantly, the state of `readline`'s internal buffer is exposed, so callers have visibility into exactly what has been received. Even with this feature, `readline` can be problematic, as we'll see in Section 6.3. System functions like `select` still won't know about `readline`'s internal buffer, so a carelessly written program could easily find itself waiting in `select` for data already received and stored in `readline`'s buffers. For that matter, mixing `readn` and `readline` calls will not work as expected unless `readn` is modified to check the internal buffer as well.

lib/readline.c

```
#include    "unp.h"

static int read_cnt;
static char *read_ptr;
static char read_buf[MAXLINE];

static ssize_t
my_read(int fd, char *ptr)
{

    if (read_cnt <= 0) {
      again:
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return (-1);
        } else if (read_cnt == 0)
            return (0);
        read_ptr = read_buf;
    }

    read_cnt--;
    *ptr = *read_ptr++;
    return (1);
}
```

```
22 ssize_t
23 readline(int fd, void *vptr, size_t maxlen)
24 {
25     ssize_t n, rc;
26     char    c, *ptr;

27     ptr = vptr;
28     for (n = 1; n < maxlen; n++) {
29         if ( (rc = my_read(fd, &c)) == 1) {
30             *ptr++ = c;
31             if (c == '\n')
32                 break;          /* newline is stored, like fgets() */
33         } else if (rc == 0) {
34             *ptr = 0;
35             return (n - 1);     /* EOF, n - 1 bytes were read */
36         } else
37             return (-1);        /* error, errno set by read() */
38     }

39     *ptr = 0;                   /* null terminate like fgets() */
40     return (n);
41 }

42 ssize_t
43 readlinebuf(void **vptrptr)
44 {
45     if (read_cnt)
46         *vptrptr = read_ptr;
47     return (read_cnt);
48 }
```
lib/readline.c

Figure 3.18 Better version of `readline` function.

2-21 The internal function `my_read` reads up to `MAXLINE` characters at a time and then returns them, one at a time.

29 The only change to the `readline` function itself is to call `my_read` instead of `read`.

42-48 A new function, `readlinebuf`, exposes the internal buffer state so that callers can check and see if more data was received beyond a single line.

Unfortunately, by using `static` variables in `readline.c` to maintain the state information across successive calls, the functions are not *re-entrant* or *thread-safe*. We will discuss this in Sections 11.18 and 26.5. We will develop a thread-safe version using thread-specific data in Figure 26.11.

3.10 Summary

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents. We always pass these structures by reference (that is, we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument. When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Socket address structures are self-defining because they always begin with a field (the "family") that identifies the address family contained in the structure. Newer implementations that support variable-length socket address structures also contain a length field at the beginning, which contains the length of the entire structure.

The two functions that convert IP addresses between presentation format (what we write, such as ASCII characters) and numeric format (what goes into a socket address structure) are `inet_pton` and `inet_ntop`. Although we will use these two functions in the coming chapters, they are protocol-dependent. A better technique is to manipulate socket address structures as opaque objects, knowing just the pointer to the structure and its size. We used this method to develop a set of `sock_` functions that helped to make our programs protocol-independent. We will complete the development of our protocol-independent tools in Chapter 11 with the `getaddrinfo` and `getnameinfo` functions.

TCP sockets provide a byte stream to an application: There are no record markers. The return value from a `read` can be less than what we asked for, but this does not indicate an error. To help read and write a byte stream, we developed three functions, `readn`, `writen`, and `readline`, which we will use throughout the text. However, network programs should be written to act on buffers rather than lines.

Exercises

3.1 Why must value-result arguments such as the length of a socket address structure be passed by reference?

3.2 Why do both the `readn` and `writen` functions copy the `void*` pointer into a `char*` pointer?

3.3 The `inet_aton` and `inet_addr` functions have traditionally been liberal in what they accept as a dotted-decimal IPv4 address string: allowing from one to four numbers separated by decimal points, and allowing a leading `0x` to specify a hexadecimal number, or a leading 0 to specify an octal number. (Try `telnet 0xe` to see this behavior.) `inet_pton` is much stricter with IPv4 address and requires exactly four numbers separated by three decimal points, with each number being a decimal number between 0 and 255. `inet_pton` does not allow a dotted-decimal number to be specified when the address family is `AF_INET6`, although one could argue that these should be allowed and the return value should be the IPv4-mapped IPv6 address for the dotted-decimal string (Figure A.10).

Write a new function named `inet_pton_loose` that handles these scenarios: If the address family is `AF_INET` and `inet_pton` returns 0, call `inet_aton` and see if it succeeds. Similarly, if the address family is `AF_INET6` and `inet_pton` returns 0, call `inet_aton` and if it succeeds, return the IPv4-mapped IPv6 address.