Prev Contents Next

3. `structs` and Data Handling

Well, we're finally here. It's time to talk about programming. In this section, I'll cover various data types used by the sockets interface, since some of them are a real bear to figure out.

First the easy one: a socket descriptor. A socket descriptor is the following type:

```
int
```

Just a regular `int`.

Things get weird from here, so just read through and bear with me. Know this: there are two byte orderings: most significant byte (sometimes called an "octet") first, or least significant byte first. The former is called "Network Byte Order". Some machines store their numbers internally in Network Byte Order, some don't. When I say something has to be in Network Byte Order, you have to call a function (such as `htons()`) to change it from "Host Byte Order". If I don't say "Network Byte Order", then you must leave the value in Host Byte Order.

(For the curious, "Network Byte Order" is also known as "Big-Endian Byte Order".)

My First Struct™--`struct sockaddr`. This structure holds socket address information for many types of sockets:

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

`sa_family` can be a variety of things, but it'll be **`AF_INET`** for everything we do in this document. *`sa_data`* contains a destination address and port number for the socket. This is rather unwieldy since you don't want to tediously pack the address in the *`sa_data`* by hand.

To deal with `struct sockaddr`, programmers created a parallel structure: `struct sockaddr_in` ("in" for "Internet".)

```
struct sockaddr_in {
    short int          sin_family;  // Address family
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that *`sin_zero`* (which is included to pad the structure to the length of a `struct sockaddr`) should be set to all zeros with the function `memset()`. Also, and this is the *important* bit, a pointer to a `struct sockaddr_in` can be cast to a pointer to a `struct sockaddr` and vice-versa. So even though `connect()` wants a `struct sockaddr*`, you can still use a `struct sockaddr_in` and cast it at the last minute! Also, notice that *`sin_family`* corresponds to *`sa_family`* in a `struct sockaddr` and should be set to "**`AF_INET`**". Finally, the *`sin_port`* and *`sin_addr`* must be in *Network Byte Order*!

"But," you object, "how can the entire structure, `struct in_addr sin_addr`, be in Network Byte Order?" This question requires careful examination of the structure `struct in_addr`, one of the worst unions alive:

```
// Internet address (a structure for historical reasons)
struct in_addr {
    unsigned long s_addr; // that's a 32-bit long, or 4 bytes
};
```

Well, it *used* to be a union, but now those days seem to be gone. Good riddance. So if you have declared *ina* to be of type `struct sockaddr_in`, then *ina.sin_addr.s_addr* references the 4-byte IP address (in Network Byte Order). Note that even if your system still uses the God-awful union for `struct in_addr`, you can still reference the 4-byte IP address in exactly the same way as I did above (this due to `#defines`.)

3.1. Convert the Natives!

We've now been lead right into the next section. There's been too much talk about this Network to Host Byte Order conversion--now is the time for action!

All righty. There are two types that you can convert: `short` (two bytes) and `long` (four bytes). These functions work for the `unsigned` variations as well. Say you want to convert a `short` from Host Byte Order to Network Byte Order. Start with "h" for "host", follow it with "to", then "n" for "network", and "s" for "short": h-to-n-s, or `htons()` (read: "Host to Network Short").

It's almost too easy...

You can use every combination of "n", "h", "s", and "l" you want, not counting the really stupid ones. For example, there is NOT a `stolh()` ("Short to Long Host") function--not at this party, anyway. But there are:

- `htons()` -- "Host to Network Short"
- `htonl()` -- "Host to Network Long"
- `ntohs()` -- "Network to Host Short"
- `ntohl()` -- "Network to Host Long"

Now, you may think you're wising up to this. You might think, "What do I do if I have to change byte order on a `char`?" Then you might think, "Uh, never mind." You might also think that since your 68000 machine already uses network byte order, you don't have to call `htonl()` on your IP addresses. You would be right, *BUT* if you try to port to a machine that has reverse network byte order, your program will fail. Be portable! This is a Unix world! (As much as Bill Gates would like to think otherwise.) Remember: put your bytes in Network Byte Order before you put them on the network.

A final point: why do *sin_addr* and *sin_port* need to be in Network Byte Order in a `struct sockaddr_in`, but *sin_family* does not? The answer: *sin_addr* and *sin_port* get encapsulated in the packet at the IP and UDP layers, respectively. Thus, they must be in Network Byte Order. However, the *sin_family* field is only used by the kernel to determine what type of address the structure contains, so it must be in Host Byte Order. Also, since *sin_family* does *not* get sent out on the network, it can be in Host Byte Order.

3.2. IP Addresses and How to Deal With Them

Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a `long` with the << operator.

First, let's say you have a `struct sockaddr_in ina`, and you have an IP address "`10.12.110.57`" that you want to store into it. The function you want to use, `inet_addr()`, converts an IP address in numbers-and-dots notation into an unsigned long. The assignment can be made as follows:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

Notice that `inet_addr()` returns the address in Network Byte Order already--you don't have to call `htonl()`. Swell!

Now, the above code snippet isn't very robust because there is no error checking. See, `inet_addr()` returns **`-1`** on error. Remember binary numbers? **`(unsigned)-1`** just happens to correspond to the IP address `255.255.255.255`! That's the broadcast address! Wrongo. Remember to do your error checking properly.

Actually, there's a cleaner interface you can use instead of `inet_addr()`: it's called `inet_aton()` ("aton" means "ascii to network"):

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

int inet_aton(const char *cp, struct in_addr *inp);
```

And here's a sample usage, while packing a `struct sockaddr_in` (this example will make more sense to you when you get to the sections on `bind()` and `connect()`.)

```
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET;         // host byte order
my_addr.sin_port = htons(MYPORT);     // short, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
```

`inet_aton()`, *unlike practically every other socket-related function*, returns non-zero on success, and zero on failure. And the address is passed back in *`inp`*.

Unfortunately, not all platforms implement `inet_aton()` so, although its use is preferred, the older more common `inet_addr()` is used in this guide.

All right, now you can convert string IP addresses to their binary representations. What about the other way around? What if you have a `struct in_addr` and you want to print it in numbers-and-dots notation? In this case, you'll want to use the function `inet_ntoa()` ("ntoa" means "network to ascii") like this:

```
printf("%s", inet_ntoa(ina.sin_addr));
```

That will print the IP address. Note that `inet_ntoa()` takes a `struct in_addr` as an argument, not a `long`. Also notice that it returns a pointer to a char. This points to a statically stored char array within `inet_ntoa()` so that each time you call `inet_ntoa()` it will overwrite the last IP address you asked for. For example:

```
char *a1, *a2;

a1 = inet_ntoa(ina1.sin_addr);  // this is 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr);  // this is 10.12.110.57
```

```
printf("address 1: %s\n",a1);
printf("address 2: %s\n",a2);
```

will print:

```
address 1: 10.12.110.57
address 2: 10.12.110.57
```

If you need to save the address, `strcpy()` it to your own character array.

That's all on this topic for now. Later, you'll learn to convert a string like "whitehouse.gov" into its corresponding IP address (see DNS, below.)

3.2.1. Private (Or Disconnected) Networks

Lots of places have a firewall that hides the network from the rest of the world for their own protection. And often times, the firewall translates "internal" IP addresses to "external" (that everyone else in the world knows) IP addresses using a process called *Network Address Translation*, or NAT.

Are you getting nervous yet? "Where's he going with all this weird stuff?"

Well, relax and buy yourself a drink, because as a beginner, you don't even have to worry about NAT, since it's done for you transparently. But I wanted to talk about the network behind the firewall in case you started getting confused by the network numbers you were seeing.

For instance, I have a firewall at home. I have two static IP addresses allocated to me by the DSL company, and yet I have seven computers on the network. How is this possible? Two computers can't share the same IP address, or else the data wouldn't know which one to go to!

The answer is: they don't share the same IP addresses. They are on a private network with 24 million IP addresses allocated to it. They are all just for me. Well, all for me as far as anyone else is concerned. Here's what's happening:

If I log into a remote computer, it tells me I'm logged in from 64.81.52.10 (not my real IP). But if I ask my local computer what it's IP address is, it says 10.0.0.5. Who is translating the IP address from one to the other? That's right, the firewall! It's doing NAT!

10.*x*.*x*.*x* is one of a few reserved networks that are only to be used either on fully disconnected networks, or on networks that are behind firewalls. The details of which private network numbers are available for you to use are outlined in RFC 1918, but some common ones you'll see are 10.*x*.*x*.*x* and 192.168.*x*.*x*, where *x* is 0-255, generally. Less common is 172.*y*.*x*.*x*, where *y* goes between 16 and 31.

Networks behind a NATing firewall don't *need* to be on one of these reserved networks, but they commonly are.
