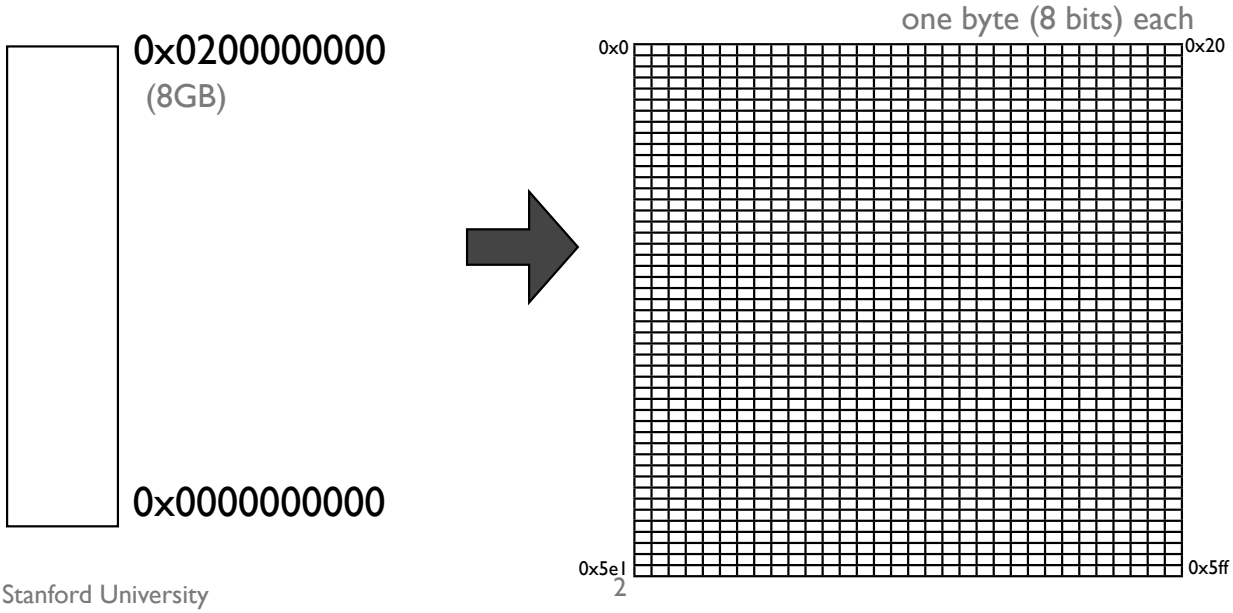
Memory, Byte Order, and Packet Formats

For two parties to communicate, they need to agree on the messages they exchange. If one party assumes messages are in Spanish and the other assumes they are in Cambodian, they will not be able to communicate. For computer communication, this means agreeing on what fields messages have, how they are arranged and formatted, and how they are represented. To generate a message to send, software typically has to create a copy of it in memory, which it then passes to the networking card. Similarly, when a computer receives a message, the networking card puts that message in memory, which software can then access. Understanding how this works and some of the pitfalls you can encounter is important if you want to understand network protocols and write network protocol software.

Computer Memory



So let's start with a simple model of computer memory. In most computers today, memory is organized in terms of bytes: 8 bit chunks of memory. A program has an address space, starting at address zero. Most computers today are 64 bits: this means that memory addresses are 64 bits long, so a computer has up to 2 to the 64 bytes, or 18 sextillion bytes. In practice, computers today do not have this much memory: they have gigabytes, which is 2 to the 30th. In this example, our computer has 8 gigabytes of memory, so its largest address is the hexidecimal value shown. Software can access each byte of this memory, or access bytes in groups, such as loading a 64-bit integer from 8 contiguous byte cells of memory in a single instruction.

Endianness

1,024 = 0x0400 =

?	?

- Multibyte words: how do you arrange the bytes?
- Little endian: least significant byte is at lowest address
  - Makes most sense from an addressing/computational standpoint

0x00	0x04

- Big endian: most significant byte is at lowest address
  - Makes most sense to a human reader

0x04	0x00

But how does a computer represent a multibyte value? Let's say we want to represent the number 1,024, which in hexidecimal is 0x0400, or 4 times 256. This value requires 16 bits, or two bytes. Which byte comes first: 0x00 or 0x04? How you lay out a multibyte value in memory is called endianness, and there are two options. In little endian, the least significant byte is at the lowest address. So the least significant byte comes first in memory. It turns out that from a computational and architectural standpoint, this can make the most sense. The other option is big endian, where the most significant byte is the lowest address. Big endian makes more sense to a human reader, because it's how we write numbers, with the most significant digits first.

Quiz

For each number, mark whether the hexidecimal representation is big endian or little endian. Don't use a calculator or other tool!

Width	Decimal	Bytes	Big Endian	Little Endian
16 bits	53	0x3500		
16 bits	4116	0x1014		
32 bits	5	0x00000005		
32 bits	83,886,080	0x00000005		
32 bits	305,414,945	0x21433412		

Here's a quiz. For each number, mark whether the hexidecimal representation is big endian or little endian. Don't use a calculator or other tool!

53 is represented in little endian. 53 is 3 times 16 plus 5, and 0x35 is in the first byte.

4116 is big endian. 4116 is equal to 4096 plus 20. So the two bytes are 0x10 and 0x14, with 0x10 being the byte representing the more significant bits, those of 4096. Since the hexidecimal is 0x1014, this means the most significant byte comes first and it's big endian.

5 is big endian -- the least significant byte is last and has the highest address.

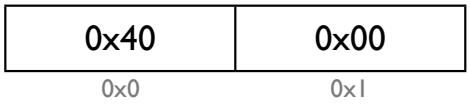
83,886,080 is little endian: it's 5 times 2 to the 24th, so this means that 0x05 is the most significant byte.

Finally, 305,414,945 is little endian. Rather than try to figure out all of the digits on this one, I just looked at the least significant bit. The least significant bit is either part of 0x21 or 0x12. If it's 0x21 and the least significant bit is 1, then the number is odd. If it's 0x12 and the least significant bit is 0, then the number is even. Since 305,414,945 is odd, this means 0x21 is the least significant byte and the number is being stored little-endian.

Network Byte Order

- Different processors have different endianness
  - Little endian: x86, big endian: ARM
- To interoperate, they need to agree how to represent multi-byte fields
- Network byte order is big endian

1,024 = 0x400 =




```
uint16_t val = 0x400;
uint8_t* ptr = (uint8_t*)&val;
```

```
if (ptr[0] == 0x40) {
  printf("big endian\n")
}
else if (ptr[1] == 0x40) {
  printf("little endian\n");
}
else {
  printf("unknown endianness!\n");
}
```


So why does this matter? If two computers are going to communicate, they need to agree on whether they represent numbers using big endian or little endian formats. This is complicated by the fact that different processors use different endianness. For example, x86 processors from Intel and AMD are little endian: the least significant byte comes first. ARM processors, in contrast, such as those in the iPhone, are big endian, where the most significant byte comes first.

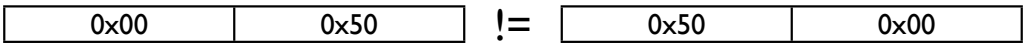
We don't want two computers to care or know whether the other side is big endian or little endian. So protocol specification bodies typically pick one and stick with it. For the Internet, this means big endian. All protocols that are Internet specifications use a big endian format.

Here's an example snippet of C code that will tell you whether your computer is big endian or little endian. It takes a 16-bite value and casts a pointer to it that lets the code look at the bytes individually. If the byte at index 0 is 0x40, the most significant byte comes first and it's big endian. If the byte at index 1 is 0x40, then it's little endian. If it's neither, well, something weird is going on.

Portable Code

- You have to convert network byte order values to your host order
- E.g., packet has a 16-bit port in network byte order, you're using a struct to access it, you want to check on your x86 processor if the port is 80

```
uint16_t http_port = 80; // Host order
if (packet->port == http_port) { ... // Network vs. host order
```



- Helper functions: htons(), ntohs(), htonl(), ntohl()
  - htons: "host to network short", ntohl: "network to host long"
  - #include <arpa/inet.h>

```
uint16_t http_port = 80; // Host order
uint16_t packet_port = ntohs(packet->port);
if (packet_port == http_port) { ... // OK
```


But wait -- this creates a complication. You need a packet to be in a big endian format, but what if your processor is little endian? Let's say, for example, that you want to set the port number of a TCP segment to be 80, the HTTP port. A simple way to do this might be to create a C struct that has a field port at the right offset. But if you use a value 80 to compare with the port field, it will be stored little endian, with 0x50 as the first byte. Big endian needs 0x50 stored in the second byte. So although the port field in the segment is 80, this test will fail.

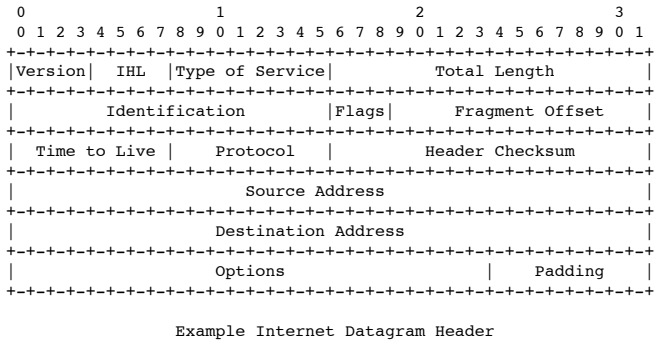
To make this easier, C networking libraries provide utility functions that convert between host order and network order. The function htons() for example, takes a host short, 16-bit, value as a parameter and returns a value in network order. There's also functions for converting a network short to a host short, and functions for longs, 32 bit values. So the right way to test whether the packet port is 80 is to read the port field of the packet structure and call ntohs to convert it from network order to host order. You can then compare it with 80 and get the correct result. In the case of a little endian architecture, ntohs and htons reverse the order of the two bytes. In the case of a big endian architecture, they just return the value unchanged.

Be careful whenever you handle network data!

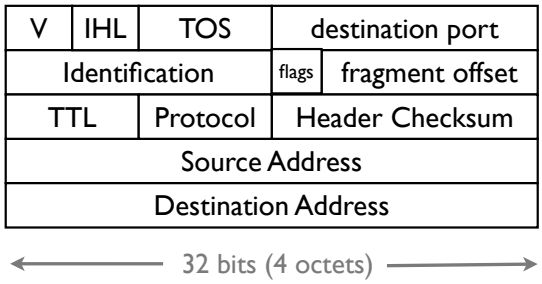
Otherwise you will waste many (avoidable) hours debugging your code due to forgetting to convert or converting twice.

These functions provide you the mechanisms by which you can write networking code that's independent of your processor architecture. But be careful! I can't stress this enough. Be careful whenever you handle network data. If you aren't principled and rigorous about when you translate between host and network order, you'll give yourself a tremendous headache because you've forgotten to convert or have inadvertently converted twice and suddenly your protocol is behaving wrongly or triggering all kinds of weird bugs.

Packet Formats



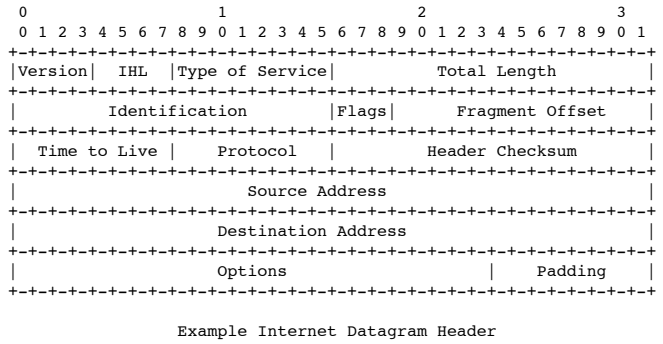
Example Internet Datagram Header



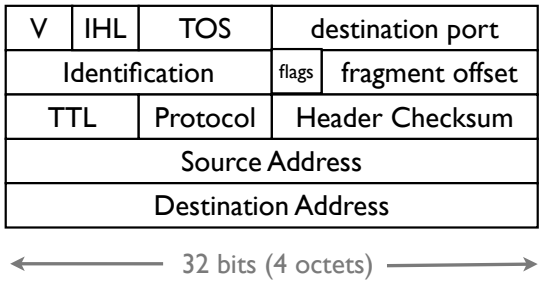
Now that we know how Internet specifications lay out multibyte values in network order, or big endian, we can look at how Internet specifications describe their packet formats. For historical reasons, Internet specifications are written in plain ASCII text. The block of text on the left is taken verbatim from Request for Comments (RFC) 791, which specifies the IP protocol, version 4, or IPv4. The top shows the bits from 0 to 31 -- packets are written 4 bytes wide. Since IPv4 has 5 rows of required fields, this means that an IPv4 header is at least 20 bytes long. Nick and I often use a simpler visual format when we show packets, like the one on the right.

To use this as an example, the total length field of an IPv4 packet is 2 bytes, or 16 bits long. This means an IPv4 packet can't be longer than 65,535 bytes. That field in the packet is stored big endian. A packet of length 1400 bytes is stored as 0x0578. So the third byte of an IP packet of that length is 0x05.

Packet Formats



Example Internet Datagram Header



Let's see this in wireshark. I'm just going to start wireshark and listen for packets. This first packet is for something called TLS, or transport layer security. It's what web browsers use for secure connections (https). TLS hides the data of the packet from us, but we can still see its headers. Using wireshark, we can see that a TLS payload is inside a TCP segment to port 443, the standard TLS port. This TCP segment is inside an IPv4 header. Looking in detail at the IPv4 header, we can see that the packet's total length field is 1230. The hexidecimal for 1230 is 0x04ce: 1024, or 0x04 times 256 plus 106, or 0xce. At the bottom, Wireshark shows us the actual bytes of the packet. And there it is, 04 ce, in big endian, or network order.

You've seen how different processors lay out numbers differently. But since network protocols need to agree, protocol specifications decide how the numbers are laid out, which can differ from your processor. To help with this, C networking libraries provide helper functions that convert between host and network order. But use them carefully! Using them haphazardly can easily lead you to many lost hours of debugging which could be prevented by being careful when you start and deciding on a principled approach to converting in your code.