# Big and Little Endian

## Basic Memory Concepts

In order to understand the concept of big and little endian, you need to understand memory. Fortunately, we only need a very high level abstraction for memory. You don't need to know all the little details of how memory works.

All you need to know about memory is that it's one large array. But one large array containing what? The array contains bytes. In computer organization, people don't use the term "index" to refer to the array locations. Instead, we use the term "address". "address" and "index" mean the same, so if you're getting confused, just think of "address" as "index".

Each address stores one element of the memory "array". Each element is typically one byte. There are some memory configurations where each address stores something besides a byte. For example, you might store a nybble or a bit. However, those are exceedingly rare, so for now, we make the broad assumption that all memory addresses store bytes.

I will sometimes say that memory is byte-addresseable. This is just a fancy way of saying that each address stores one byte. If I say memory is nybble-addressable, that means each memory address stores one nybble.

## Storing Words in Memory

We've defined a word to mean 32 bits. This is the same as 4 bytes. Integers, single-precision floating point numbers, and MIPS instructions are all 32 bits long. How can we store these values into memory? After all, each memory address can store a single byte, not 4 bytes.

The answer is simple. We split the 32 bit quantity into 4 bytes. For example, suppose we have a 32 bit quantity, written as $90AB12CD_{16}$, which is hexadecimal. Since each hex digit is 4 bits, we need 8 hex digits to represent the 32 bit value.

So, the 4 bytes are: 90, AB, 12, CD where each byte requires 2 hex digits.

It turns out there are two ways to store this in memory.

### Big Endian

In big endian, you store the most significant byte in the smallest address. Here's how it would look:

| Address | Value |
|---|---|
| 1000 | 90 |
| 1001 | AB |
| 1002 | 12 |
| | |

| | |
|---|---|
| 1003 | CD |

## Little Endian

In little endian, you store the least significant byte in the smallest address. Here's how it would look:

| Address | Value |
|---|---|
| 1000 | CD |
| 1001 | 12 |
| 1002 | AB |
| 1003 | 90 |

Notice that this is in the reverse order compared to big endian. To remember which is which, recall whether the least significant byte is stored first (thus, little endian) or the most significant byte is stored first (thus, big endian).

Notice I used "byte" instead of "bit" in least significant bit. I sometimes abbreciated this as LSB and MSB, with the 'B' capitalized to refer to byte and use the lowercase 'b' to represent bit. I only refer to most and least significant byte when it comes to endianness.

## Which Way Makes Sense?

Different ISAs use different endianness. While one way may seem more natural to you (most people think big-endian is more natural), there is justification for either one.

For example, DEC and IBMs(?) are little endian, while Motorolas and Suns are big endian. MIPS processors allowed you to select a configuration where it would be big or little endian.

Why is endianness so important? Suppose you are storing int values to a file, then you send the file to a machine which uses the opposite endianness and read in the value. You'll run into problems because of endianness. You'll read in reversed values that won't make sense.

Endianness is also a big issue when sending numbers over the network. Again, if you send a value from a machine of one endianness to a machine of the opposite endianness, you'll have problems. This is even worse over the network, because you might not be able to determine the endianness of the machine that sent you the data.

The solution is to send 4 byte quantities using network byte order which is arbitrarily picked to be one of the endianness (not sure if it's big or little, but it's one of them). If your machine has the same endianness as network byte order, then great, no change is needed. If not, then you must reverse the bytes.

## History of Endian-ness

Where does this term "endian" come from? Jonathan Swift was a satirist (he poked

fun at society through his writings). His most famous book is "Gulliver's Travels", and he talks about how certain people prefer to eat their hard boiled eggs from the little end first (thus, little endian), while others prefer to eat from the big end (thus, big endians) and how this lead to various wars.

Of course, the point was to say that it was a silly thing to debate over, and yet, people argue over such trivialities all the time (for example, should braces line in parallel or not? vi or emacs? UNIX or Windows).

## Misconceptions

Endianness only makes sense when you want to break a large value (such as a word) into several small ones. You must decide on an order to place it in memory.

However, if you have a 32 bit register storing a 32 bit value, it makes no sense to talk about endianness. The register is neither big endian nor little endian. It's just a register holding a 32 bit value. The rightmost bit is the least significant bit, and the leftmost bit is the most significant bit.

There's no reason to rearrange the bytes in a register in some other way.

Endianness only makes sense when you are breaking up a multi-byte quantity, and attempting to store the bytes at consecutive memory locations. In a register, it doesn't make sense. A register is simply a 32 bit quantity, $b_{31}....b_0$, and endianness does not apply to it.

With regard to endianness, You may argue there's a very natural way to store 4 bytes in 4 consecutive addresses, and that the other way looks strange. In particular, it looks "backwards". However, what's natural to you may not be natural to someone else. The fact of the matter is that the word is split in 4 bytes, and most people would agree that you need some order to place it in memory.

### C-style strings

Once you start thinking about endianness, you begin to think it applies to everything. Before you see big or little endian, you may have had no idea it even existed. That's because it's reasonably well-hidden from you.

If you do bitwise/bitshift operations on an int, you don't notice the endianness. The machine arranges the multiple bytes so the least significant byte is still the least significant byte (e.g., $b_{7-0}$) and the most significant byte is still the most significant byte (e.g., $b_{31-24}$).

So, it's natural to think whether strings might be saved in some sort of strange order, depending on the machine.

This is where it's useful to think about all the facts you know about arrays. A C-style string, after all, is still an array of characters.

Here are some facts you should know about C-style strings and arrays.

- C-style strings are stored in arrays of characters.
- Each character requires one byte of memory, since characters are represented in ASCII (in the future, this could change, as Unicode becomes more popular).

- In an array, the address of consecutive array elements increases. Thus, & arr[ i ] is less than & arr[ i + 1 ].
- What's not as obvious is that if something is stored in increasing addresses in memory, it's going to be stored in increasing "addresses" in a file. When you write to a file, you usually specify an address in memory, and the number of bytes you wish to write to the file starting at that address.

So, let's imagine some C-style string in memory. You have the word "cat". Let's pretend 'c' is stored at address 1000. Then 'a' is stored at 1001. 't' is at 1002. The null character '\0' is at 1003.

Since C-style strings are arrays of characters, they follow the rules of characters. Unlike int or long, you can easily see the individual bytes of a C-style string, one byte at a time. You use array indexing to access the bytes (i.e., characters) of a string. You can't easily index the bytes of an int or long, without playing some pointer tricks (using reinterpret cast, for example, in C++). The individual bytes of an int are more or less hidden from you.

Now imagine writing out this string to a file using some sort of write() method. You specify a pointer to 'c', and the number of bytes you wish to print (in this case 4). The write() method proceeds byte by byte in the character string and writes it to the file, starting with 'c' and working to the null character.

Given that explanation, is it clear whether endianness matters with C-style strings? Hopefully, it is clear.

As an aside, since C++ strings are objects, it may have complicated inner structures, and so it's less obvious what a C++ string would look like when print out to a file. It's well-known what a C-style string looks like (a sequence of characters ending in a null character), which is why I've been careful to call them C-style strings.