

A note on Endiansim

Foundation of High Performance Computing @ UniTS, 2021-2022

Intro

You know that “a basic-type variable” is nothing else but a bunch of n bytes in memory whose bits are read *as if* they form the given basic type (i.e. `int`, `float`, `double`, ...).

However, there is one more detail to be considered when that memory region spans multiple bytes: is the most significant byte stored either in the lowest- or in the highest- addresses in memory ?

To clarify the meaning of that let's consider how *we*, the humans, write numbers in the commonly used arabic notation considering the large number 3,141,592,653.

Since we write from left to right, the lowest “memory” (i.e. the position in the page starting from the top-left corner which has coordinate 0) position is occupied by the first digit `3` which is *the most significant one*:

“Memory” address	0	1	2	3	4	5	6	7	8	9	10
	3	1	4	1	5	9	2	6	5	3	9

Hence, in the arabic notation the most significant digits occupy the lowest memory addresses and as the significance (i.e. the positional value, or in other words the power of 10 to which the digit corresponds) decreases the “address in the page” increases.

In other words, the *big-end* of the number we want to write is stored at the begin of the memory area we used to store it: this notation is referred to as *big-endianism*.

At the opposite, in the case we wrote from right to left (still considering the top-left corner as the origin) we would have that the *least significant* digits would occupy the lowest memory addresses:

“Memory” address	0	1	2	3	4	5	6	7	8	9	10
	9	3	5	6	2	9	5	1	4	1	3

That notation, in which the most significant digits are stored at the end of the memory region, is called *little-endianism*.

So, given that both ways are equally legitimate, how are the number represented in a computer? That is an architectural choice and it amounts to an ab-initio CPU's architectural choice.

It happens that both ways are present in the world: some CPU implement little-endianism (like Intel's and AMD's x86 architecture) and some other implement big-endianism (like IBM's *powerPC* architectures and many historical RISC architectures like the old Motorola 68k). More exotic treatment are also present (ARM's are generally little-endian but, depending on the version, allow switching). In both little- and big- endianism the *bit-order* inside a single byte is always little-endian (i.e. the most significant bit is the the highest address in the byte).

There existed also examples of the so called *middle-endianism* that were big-endian systems which also swapped to big-endian bits-order in each byte.

The big-endian choice is understandable in the light of the frequent necessity in the past to look at memory dumps by human sight, events in which a human-like big-endian order was also a clear advantage (I would dare to say that real *digital natives* are those who understand that by experience; in any case, remember, when looking at a little-endian memory dump, that you must reverse the order of byte interpretation).

As a side consideration, the human big-endian power-of-ten representation that we use as western humans may have been motivated by the necessity of getting immediately the most significant part of a number, which in our left-to-right convention comes to lie at the lowest “addresses”.

How to recognize your architecture

For instance, `0x100` is `1 00000000` in binary and hence it sets the first bit of the second byte and no bits in the first byte; `0xFF` is `11111111` in binary and hence it sets all the bits in the first byte.

Big-endian

[illegible]

Little-endian

0x100

lowest byte									highest byte								
0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

0xFF

lowest byte									highest byte								
0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0

0x100 & 0xFF

lowest byte									highest byte								
0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

And so, clearly, checking the result of the `&` you can easily understand the endianness of the architecture you're running on. In the actual code that I gave to you, the value `0xFF` is used. It sets the first 4 bits of the lowest byte, but since we are checking for zero/non-zero condition that is perfectly equivalent to the case depicted above.