A note on Endiansim

Foundation of High Perfomance Computing @ UniTS, 2020-2021

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- adresseses 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 addressed 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 its 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 (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 little-endian representation looks much more natural in a computer system as it follows the “natural” power-of-two positional sense (the highest the position, the highest the power) and is nowadays the common choice in most systems because of this clear advantage.
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 you to look ad a little-endian memory dump, that you must reverse the order of byte interpretation).

Funny enough, historically in the geek digital culture the confrontation between big- and little-endianers assumed the spirit of another famous *querelle*, from which it also grabbed the named: obviously, that between the Swift’s Lilliputians. At least, no one could say that the geeks lack auto-irony next to some ego inflation.

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 be lies at the lowest “addresseses”.

As for many other things, then, there is much more than meets the eyes.

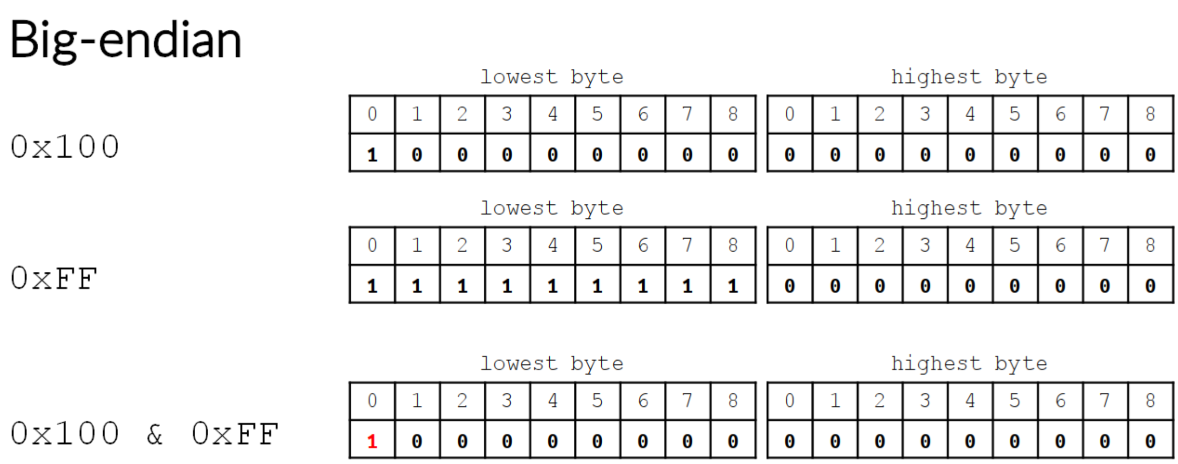
Practical consideration

You encountered a manifestation of this facts in the `pgm` image format, in that it, being big-endian, requires that the 2 bytes used in the `short int` case are written in big endian (i.e. the most significant byte comes first) while you are normally using a little-endian machine.

Let’s then examine the code that I gave to you to handle this matter, in the `swpa_image()` routine.

First things first: how to recognize your architecture

Provided that the C language provides you a macro (the Boolean `LITTLE_ENDIAN`) that informs you about that, the trick that I normally use is quite simple: assigning to a multiple-byte basic-type (a `short int` is sufficient) a value that sets the bits in the highest byte and no bits in the lowest byte and then checking which comes first in memory.
For istance, `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.
The logical `and` between the two results in the two following sequences.



Little-endian

	lowest byte									highest byte								
	0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
`0x100`	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
`0xFF`	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
`0x100 & 0xFF`	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 endianism of the the architecture you're running on. In the actual code that I gave to you, the value `0xF` 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 equavalent to the case depicted above.

How to adapt your bytes to an alien endianism

Swtching between big- and little- endianism is faitly simple. It amounts to swap bytes. When the bytes are only two, as in the `pgm` image format, it is the simplest case.
Using the bit-shift and bit-wise logical operators the task is accomplished (`MEM` in the following is the `short int` value we want to swp):

1. `MEM & (short int)0xff00` builds a `short int` value that has only the most significant byte of the original `MEM`. The most significant byte is still in its original position.
2. `(MEM & (short int)0xff00) >> 8` builds the value and shifts it by 1 byte *to the right*, i.e. the shift moves the most significant byte in the least significant position. It is the same than dividing the original value by 256; actually that first operation could be simplified in `(MEM >> 8)`.
3. `(MEM & (short int)0x00FF) << 8` selects theleast significant byte and shifts it to the most significant position. Also this operation could be written as `(MEM << 8)`.
4. summing the 2 previous results, which could also be written as a bit-wise or `|`, gives you back your swapped 2-bytes values.

The simplifications `MEM >> 8` and `MEM << 8` have not been adopted for eductaional purposes. They can be used only in the 2-bytes case, while if you have 4, 8 or even more byte you have to select individually each byte by using suited bit masks.
Are you willing to write your own general routine to swap endianism?
At the end of the course I could provide you my own for testing purposes.