"Big Endian" and "Little Endian" Formats for Integers

In base 10, integers are represented with the most significant digit first; e.g., 9,263. A 16-bit (2's complement) representation of this number in base 2 or base 16 (hexadecimal, or hex) is given by

$9263_{10} = 0010\ 0100\ 0010\ 1111_2 = 24\ 2F_{16}$

and its inverse by

$-9263_{10} = 1101\ 1011\ 1101\ 0001_2 = DB\ D1_{16}$.

If the representation is extended to 32-bit 2's complement (represented in hex) the numbers become

$9263_{10} = 00\ 00\ 24\ 2F_{16}$

and

$-9263_{10} = FF\ FF\ DB\ D1_{16}$.

This representation has the advantage of lining up the bytes in the way people are used to writing numbers, which is not necessarily the best way at the level of hardware.

A commonly used practice (e.g., Intel microprocessors employ this) is to change the byte order of the number to have the lowest byte first (the bits within each byte still have highest order bit first); e.g.,

9263_{10} is stored in the order 2F 24 00 00

and

-9263_{10} is stored in the order D1 DB FF FF.

This has been dubbed the "little endian" format. The advantage is that the number starts at the same address in memory, regardless of the precision (16-bit, 32-bit, 64-bit, etc), facilitating construction of extended precision routines for arithmetic regardless of data path width.

A byte order that conforms to the "natural" order has been dubbed "big endian", since the most significant byte comes first. It is not without advantages. The sign of the number is on the first byte. For little endian, you have to know how long the representation is and index to the last byte to find the sign. Also since the order conforms to print

order, converting to print form is simplified. Motorola processors employ a big endian format.

This distinction spills over to non-ASCII file formats, where numeric data is represented in binary form, so you have to know whether to store numbers in big endian or little endian for "standard" programs to process your file. For example, the Windows graphics ".bmp" format is strictly little endian. If a little endian machine is being used, this poses no problem, but if the machine is big endian, the byte order of numbers has to be changed when they are written to file.

Adobe Photoshop uses big endian, "gif" files are little endian, "jpg" files are big endian, and "tiff" files handle either (as data is written, an identifier to tell which endian format is being used has to be included with it).

Unix `od` command

One way to determine the endian format being used is to employ the Unix `od` command:

`od` displays the contents of the file byte by byte (octal dump)
usage: "`od -xc` *<filename>*"
(exhibits the file byte by byte in both hex and ASCII)

The `c` option directs `od` to exhibit the byte order of the file as ASCII characters. The `x` option directs `od` to exhibit the file in 16 bit (2 byte) chunks. On Osprey, integers are in little endian (with `long` 64 bits and `int` 32 bits). If the integer `X'43444546'` (chosen because it also can be interpreted as characters) is written to file `exod`, then

```
od -xc exod
```

returns

```
0000000 4546 4344
          F   E   D   C
0000010
```

which shows (starting from location `0000000` in the file) both hex and character interpretations of what is present. Notice that the actual byte order is used by the character interpretation. In the hex interpretation, each pair of bytes is inverted to big endian for display purposes (presumably for reasons associated with the octal machines for which this command was originally written). The SIC machine uses 24-bit big endian integers, which is an added complication for producing a file of big endian numbers for a SIC/XE program to process under the SIC simulator on Osprey (basically, to produce a SIC integer from an Osprey integer requires writing the last 3 bytes in reverse order byte by byte to the file).

Converting the endian format is straight-forward; e.g.,

```
int swap_endian(int x) {
  char *c;
  c = (char *)&x;
  /* XOR byte pairs 3 times swaps them in place */
  c[0]=c[0]^c[3]; c[3]=c[0]^c[3]; c[0]=c[0]^c[3];
  c[1]=c[1]^c[2]; c[2]=c[1]^c[2]; c[1]=c[1]^c[2];
  return(x);
}
```