Big Endian versus Little Endian — Define Byte

To help understand the difference between Big and Little Endian let's take a closer look at how data is stored in Flash Program Memory. We will first look at the Define Byte (`.DB`) Assembly Directive and then at the Define Word (`.DW`) Assembly Directive.

```
             //            gfedcba     gfedbca     gfedbca     gfedbca     gfedbca     gfedbca
000036 063f
000037 4f5b
000038 6d66  table: .DB 0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110, 0b01101101
             //         0           1           2           3           4           5
000039 077d
00003a 677f
00003b 7c77         .DB 0b01111101, 0b00000111, 0b01111111, 0b01100111, 0b01110111, 0b01111100
             //         6           7           8           9           A           B
00003c 5e39
00003d 7179         .DB 0b00111001, 0b01011110, 0b01111001, 0b01110001
             //         C           D           E           F
```

Each table entry (`.DB`) contains one byte. If we look at the first table entry we see 0b00111111 which corresponds to 3f in hexadecimal. Comparing this with the corresponding address and data fields on the left ... Wait a minute – where did 06 come from? The the second entry in the table ($0b00000110 = 06_{16}$). The bytes are backwards and here is why.

There are two basic ways information can be saved in memory known as Big Endian and Little endian. For Big Endian the most significant byte (big end) is saved in the lowest order byte, so 0x3f06 woud be saved as bytes 0x3f 0x 0x06. For Little Endian the least sigfificant byte (little end) is saved in the lowest order byte; so 0x3f06 is saved as bytes 0x06 and 0x3f. As you hopefully have guessed by now the AVR processor is designed to work with data words saved as little endian.