

Bit Order

Bit order refers to the direction in which bits are represented in a byte of memory. Just as words of data may be written with the most significant byte or the least significant byte first (in the lowest address of the word), so too can bits within a byte be written with the most significant bit or the least significant bit first (in the lowest position of the byte).

The bit order we see most commonly is the one in which the zeroth, or least significant bit, is the first bit read from the byte. This is referred to as up bit ordering or normal bit direction. When the seventh, or most significant, bit is the first one stored in a byte, we call this down bit ordering, or reverse bit direction.

The terms big-endian and little-endian are sometimes erroneously applied to bit order. These terms were specifically adopted as descriptions of differing byte orders only and are not used to differentiate bit orders (see the section called "Byte Order" earlier in this chapter).

Normal bit direction, least significant bit to most significant bit, is often used in transmitting data between devices, such as FAX machines and printers, and for storing unencoded bitmapped data. Reverse bit direction, most significant bit to least significant bit, is used to communicate data to display devices and in many data compression encoding methods. It is therefore possible for a bitmap image file to contain data stored in either or both bit directions if both encoded and unencoded data is stored in the file (as can occur in the TIFF and CALS Raster image file formats).

Problems that occur in reading or decoding data stored with a bit order that is the reverse of the expected bit order are called bit sex problems. When the bit order of a byte must be changed, we commonly refer to this as reversing the bit sex.

Color sex problems result when the value of the bits in a byte are the inverse of what we expect them to be. Inverting a bit (also called flipping or toggling a bit) is to change a bit to its opposite state. A 1 bit becomes a 0 and a 0 bit becomes a 1. If you have a black-on-white image and you invert all the bits in the image data, you will have a white-on-black image. In this regard, inverting the bits in a byte of image data is normally referred to as inverting the color sex.

It is important to realize that inverting and reversing the bits in a byte are not the same operation and rarely produce the same results. Note the different resulting values when the bits in a byte are inverted and reversed:

```
Original Bit Pattern: 10100001
Inverted Bit Pattern: 01011110
Original Bit Pattern: 10100001
Reversed Bit Pattern: 10000101
```

There are, however, cases when the inverting or reversing of a bit pattern will yield the same value:

```
Original Bit Pattern: 01010101
Inverted Bit Pattern: 10101010
Original Bit Pattern: 01010101
Reversed Bit Pattern: 10101010
```

Occasionally, it is necessary to reverse the order of bits within a byte of data. This most often occurs when a particular hardware device, such as a printer, requires that the bits in a byte be sent in the

Bit Order

reverse order in which they are stored in the computer's memory. Because it is not possible for most computers to directly read a byte in a reversed bit order, the byte value must be read and its bits rewritten to memory in reverse order.

Reversing the order of bits within a byte, or changing the bit sex, may be accomplished by calculation or by table look-up. A function to reverse the bits within a byte is shown below:

```
//
// Reverse the order of bits within a byte.
// Returns: The reversed byte value.
//
BYTE ReverseBits(BYTE b)
{
    BYTE c;
    c = ((b >> 1) & 0x55) | ((b << 1) & 0xaa);
    c |= ((b >> 2) & 0x33) | ((b << 2) & 0xcc);
    c |= ((b >> 4) & 0x0f) | ((b << 4) & 0xf0);
    return(c);
}
```

If an application requires more speed in the bit-reversal process, the above function can be replaced with the REVERSEBITS macro and look-up table below. Although the macro and look-up table is faster in performing bit reversal than the function, the macro lacks the prototype checking that ensures that every value passed to the function ReverseBits() is an 8-bit unsigned value. An INVERTBITS macro is also included for color sex inversion.

```
#define INVERTBITS(b)    (~(b))
#define REVERSEBITS(b)  (BitReverseTable[b])
static BYTE BitReverseTable[256] =
{
    0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
    0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
    0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
    0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
    0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
    0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
    0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
    0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
    0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
    0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
    0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
    0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
    0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
    0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
    0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
    0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
    0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
```

Bit Order

```
0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,  
0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,  
0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,  
0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,  
0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,  
0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,  
0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,  
0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,  
0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,  
0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,  
0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,  
0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,  
0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,  
0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,  
0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff  
};
```