Programming 101 - By Eric Weddington

To really understand what's going, it's best to learn C languages bit operators and about truth tables.

| bit OR
& bit AND
~ bit NOT
^ bit EXLUSIVE OR (XOR)
<< bit LEFT SHIFT
>> bit RIGHT SHIFT

These operators work on bits and not logical values. Take two 8 bit bytes, combine with any of these operators, and you will get another 8 bit byte according to the operator's function. These operators work on the individual bits inside the byte.

A truth table helps to explain each operation. In a truth table, a 1 bit stands for true, and a 0 stands for false.

The OR operation truth table:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

The AND operation truth table:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

The XOR operation truth table:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

The NOT operator inverts the sense of the bit, so a 1 becomes a 0, and a 0 becomes a 1.

So let's say I have a byte foo that is initialized to 0:
Code:
```
unsigned char foo = 0;
```

To set bit 0 in foo and then store the result back into foo:
Code:
```
foo = foo | 0x01;
```

The OR operation is used between the variable that we want to change and a constant which is called a BIT MASK or simply the MASK. The mask is used to identify the bit that we want changed.

Remember that we write the constants in hexadecimal because it's shorter than writing it in binary. It is assumed that the reader knows how to convert back and forth between hex and binary.

Usually, though the statement is made shorter in real programming practice to take advantage of C's compound assignment:

Code:
```
foo |= 0x01;
```

This is equivalent to the statement above.

To clear bit 0 in foo requires 2 bit operators:

Code:
```
foo = foo & ~0x01;
```

This uses the AND operator and the NOT operator. Why do we use the NOT operator? Most programmers find it easier to specify a mask wherein the bit that they are interested in changing, is set. However, this kind of mask can only be used in setting a bit (using the OR operator). To clear a bit, the mask must be inverted and then ANDed with the variable in question. It is up to the reader to do the math to show why this works in clearing the desired bit.

Again, the statement is made shorter with a compound assignment:

Code:
```
foo &= ~0x01;
```

To see if a bit is set or clear just requires the AND operator, but with no assignment. To see if bit 7 is set in the variable foo:

Code:
```
if(foo & 0x80)
{
}
```

The condition will be zero if the bit is clear, and the condition will be non-zero if the bit is set. NOTE! The condition will be NON-ZERO when the bit is set. But the condition will not NECESSARILY BE ONE. It is left to the reader to calculate the value of the condition to understand why this is the case.

There is another useful tool that is not often seen, and that is when you want to flip a bit, but you don't know and you don't care what state the bit is currently in. Then you would use the XOR operator:

Code:
```
foo = foo ^ 0x01;
```

Or the shorter statement:

Code:
```
foo ^= 0x01;
```

A lot of times the bit mask is built up dynamically in other statements and stored in a variable to be used in the assignment statement:

Code:
```
foo |= bar;
```

Sometimes, a programmer wants to specify the bit NUMBER that they want to change and not the bit MASK. The bit number always starts at 0 and increases by 1 for each bit. An 8 bit byte has bit numbers 0-7 inclusive. The way to build a bit mask with only a bit number is to LEFT SHIFT a bit by the bit number. To build a bit mask that has bit number 2 set:

Code:
```
(0x01 << 2)
```

To build a bit mask that has bit number 7 set:

Code:
```
(0x01 << 7)
```

To build a bit mask that has bit number 0 set:

Code:
```
(0x01 << 0)
```

Which ends up shifting the constant 0 bytes to the left, leaving it at 0x01.

MACROS

Because there are a number of programmers who don't seem to have a familiarity with bit

flipping (because they weren't taught it at school, or they don't need to know it because of working on PCs), most programmers usually write macros for all of these operations. Also, it provides a fast way of understanding what is happening when reading the code, or it provides additional functionality.

Below is a set of macros that works with ANSI C to do bit operations:

Code:
```
#define bit_get(p,m) ((p) & (m))
#define bit_set(p,m) ((p) |= (m))
#define bit_clear(p,m) ((p) &= ~(m))
#define bit_flip(p,m) ((p) ^= (m))
#define bit_write(c,p,m) (c ? bit_set(p,m) : bit_clear(p,m))
#define BIT(x) (0x01 << (x))
#define LONGBIT(x) ((unsigned long)0x00000001 << (x))
```

To set a bit:
Code:
```
bit_set(foo, 0x01);
```

To set bit number 5:
Code:
```
bit_set(foo, BIT(5));
```

To clear bit number 6 with a bit mask:
Code:
```
bit_clear(foo, 0x40);
```

To flip bit number 0:
Code:
```
bit_flip(foo, BIT(0));
```

To check bit number 3:
Code:
```
if(bit_get(foo, BIT(3)))
{
}
```

To set or clear a bit based on bit number 4:
Code:
```
if(bit_get(foo, BIT(4)))
{
   bit_set(bar, BIT(0));
```

```
}
else
{
   bit_clear(bar, BIT(0));
}
```

To do it with a macro:
Code:

```
bit_write(bit_get(foo, BIT(4)), bar, BIT(0));
```

If you are using an unsigned long (32 bit) variable foo, and have to change a bit, use the macro LONGBIT which creates un unsigned long mask. Otherwise, using the BIT() macro, the compiler will truncate the value to 16-bits.[/list]