Becoming Bit Wise

Content

by *Gene Myers*
last updated *2000/02/15* (version *1.2*)
also available as XML

Foreword

Bitwise operations often cause a great deal of confusion among beginning programmers. I credit this confusion to most entry-level texts on C/C++ programming; they often explain the syntax of the operations, but don't give the student a real-world reason for using them. Hence, the student just commits the syntax to memory for the short term. Its not until they have a need to use them, do they fully understand them. This article attempts to correct this problem, by establishing a real world senerio at the outset.

First, we're going to look at the Binary number system, though. I'm going to explain its strong relationship to the Hexadecimal and other number systems. Then, I'm going to discuss the most widely used application of bitwise operations - flags variables. I'll show you how to use bitwise operations set, clear, test and toggle bits in flag variables. We are also going to look simple encryption, and see some examples of bitwise arithmetic.

Skills Check: Before you begin, make sure you understand of the variable types (char, int, double, long) and the allowable bounds of their values, as well as the signed and unsigned modifiers. [See Appendix A "Data Types"]

Introduction - The Binary Number System

Simply put, bitwise operations are operations that manipulate values one or more bits at a time. As I hope anyone reading this already knows, all numbers on computers are represented by the binary number system. a series of 1's and 0's that represent the electrical state of On or Off. For instance, when you declare a numerical variable, the C compiler translates that number into it binary (Base 2) format. When displaying or

printing a variable, the compiler formats the binary number back into the Decimal numbering system, or the number system you specify. For example, when using printf , you can specify decimal (%d, %u, %l, etc),hexadecimal (%x), or Octal (%o). When initializing a variable, if the number is just digits, the C compiler defaults to decimal (ie- int var_name=36); if its preceded with an 0x, its interpreted as Hexadecimal (i.e. int var_name= 0x5E), or if its preceded by a 0, it's assumed to be Octal (i.e. int var_name=036).

Before we jump into bitwise operations, lets cover the Binary number system and how it relates to the other number systems.

Binary and Decimal numbers

Examine the diagram below:

7	6	5	4		3	2	1	0	Bit Position
1	0	1	0		1	0	1	0	Bit Value

In this example, we have the binary number, **10101010**. You will notice in the diagram, the 'Bit Position' of each 1 or 0. The bit farthest to the right, Bit 0, is known as the Least Significant Bit. Conversely, Bit 7 in this example, is known as the Most Significant Bit. The algorithm for translating a binary number to our standard Decimal number system is easy:
v = The value of the Bit (either 1 or 0 in Binary)
B = The Base numbering system. Binary is 2, Decimal is 10, Hexidecimal is 16, etc
p = The Bit Position
The basic formula v * (B^p) determines the value of each bit. If you have an 8 bit number as above in our example, you would add each of the values derived from the formula, together. Our example about would be:

(1 * (2^7)) + (0 * (2^6)) + (1 * (2^5)) + (0 * (2^4)) + (1 * (2^3)) + (0 * (2^2)) + (1 * (2^1)) + (0 * (2^0))


I hope everyone remembers, any number to the 0 power equals 1.

So, based on that, (1*128)+(0*64)+(1*32)+(0*16)+(1*8)+(0*4)+(1*2)+(0*1) = 170
Now, if you haven't already done so, take a look at the sidebar on Datatypes. [See Appendix B "Using Unsigned Data Types for Portability"]

You will notice something interesting. Look at the datatype, unsigned char. You'll notice it's 8 bits in length, and its maximum value is 255. Apply the above formula to 8 bits, all 1's; 255. Starting to make sense?
You might then notice, that the 'char datatype is also 8 bits, but its value range is -128 to 127. That is because the Most Significan Bit is used to signify the 'sign' of the number: if the MSB is 1, the number is Negative, if the MSB is 0, the number is positive.If a variable is declared as 'char', the value can be 'signed' and therefor bits 0-6 are for the number, and bit 7 holds the 'sign'. The maximum value of 6 bits is 127. So how do we get-128? Well, the value 00000000 would be 0, not Negative 0. So, 10000000 wouldn't be Positive zero, its -128.

When writing binary numbers, its good to segment them into 4 bit groups (4 bits are called a Nibble (or Nybble), and 8 bits are a Byte)...

... i.e. 1101 1111 0011 0001

it makes them much easier to read, and you're less likely to loose your place. And, there's also another reason for doing this, as you'll learn next.

The Connection Between Binary and Hexadecimal Numbers

While Binary numbers are used for the internal representation of numbers in computers, the most convenient system to represent them outside of the computer is the Hexadecimal numbering system, because of its close relationship to Binary. You can think of it as a kind of shorthand binary.

As I showed you a formula for converting binary numbers to decimal, imagine the same formula with a different Base; 16 for Hexadecimal. But in Binary you multiply either a 1 or 0 by the Base to the Bit Position (v * (B ^ p), but we only have 10 unique digits at our disposal (0-9). So how do we symbolize the other 5 digits?. For the digits 10 - 15, in Hex, we use the Letters A-F.

Now consider this hexadecimal number, and a formula like we used above:

5A2=(5*32)+(10*16)+(1*2)=1442 Decimal

(The letter 'h' is usually written after a Hex number to avoid confusion. But remember, in C/C++, Hexadecimal numbers are differentiated from Decimal's by preceding them with 0x ..ie- 0x5A2).

Now, for the connection I promised you. Every 'bit' in Hex can be represented by a four bit decimal number, and vice versa. Obviously, this property is due to the fact that 16=2 to the 4th power. To go from Hex to Binary, just replace each Hex digit, one by one, with the corresponding group of four binary digits.

5A2= 5 (0101) A(1010) 2(0010), or 0101 1010 0010

I told you that there was another reason for segmenting binary numbers into groups of four. This is it. And obviously, it works the same in reverse:

0000 0100 1010 1111= 04AF= 4AFh or 0x4AF

Another Number System, BCD (Binary Coded Decimal)

BCD's are probably the most rare and least understood numbering system. It was introduced in early computers, and was widely used in business applications. Its still used in COBOL and in some spreadsheets. But the reason I'm mentioning it here, is because in PC's, the current date and time stored in the internal CMOS memory is in the BCD format. BCD is a strange mix of decimal and binary. The Decimal number systems 0-9's binary equivalents 0000 - 1001 are the integers used in BCD's. Its not a very efficient numbering system, because the binary numbers 1010, 1011, 1100, 1101, and 1111 are never used. Therefore, the largest 8-bit number you can have is 99. This isn't a problem since when storing the date and time, that's the largest number you'll need.

In the CMOS, the second, minute, hour, day of week, and month each occupy one byte (the year occupies two bytes, one for the lower two digits, and one for the upper two).

Within a 'normal' 8 bit variable (a byte), the maximum value would be 255 for an unsigned value or 127 for a signed value. Remember that the most significant bit is used for the 'sign' in signed values.
[See Appendix B "Using Unsigned Data Types for Portability"]

This is how decimal numbers appear as BCD's:

BCD	Binary Representation							
1	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	1
9	0	0	0	0	1	0	0	1
10	0	0	0	1	0	0	0	0
15	0	0	0	1	0	1	0	1
55	0	1	0	1	0	1	0	1
99	1	0	0	1	1	0	0	1

Do you see how it relates to Hex numbers? i.e. 12 BCD= 1(0001) 2(0010) or 0001 0010

Binary Arithmetic

I personally think the safest way to perform arithmetic on Hex or Binary numbers outside of a program is to convert them to Decimal first, perform the calculations, and then convert the result back. Many calculator companies manufacture calculators that perform Hex and Binary arithmetic also. Users of Win95 can use the calculator that comes as part of the OS, by checking the Scientific option. Hexadecimal/Decimal/Octal/Binary conversions are easy, though somewhat tedious, by pressing the F5 to F8 buttons respectively. But the nicest thing about the Win95 calculator- it lets you perform the bitwise calculations of AND, OR, XOR (Exclusive OR), and NOT (Bitwise Inverse) as well as Bitwise Shift (Both Left and Right).

Understanding Flag Variables

One of the most common uses of bitwise operations is the manipulation of a Flag variable. Consider a program that needs to keep track of a number of different key 'states'. For example, a program keeps track of which arrow key or keys are being pressed at any one time. We could define a Boolean variable for each keys 'state'; True if its is pressed, and False if it is not. But each time we need to test the state of the keys, we would have to test each of the four variables, and use a complex conditional statement to determine which of the 16 possible combinations are being involked.

A much simpler solution as you might have guess, is the use of a Flag variable. If we declare the variable FLAG as an 'unsigned char', it will be one byte long and therefore have 8 bit positions (0-7). We could therefor store the state of 8 keys, although for this example we are only tracking 4 keys; the Arrow keys.

7	6	5	4	3	2	1	0	Bit Position
0	0	0	0	0	1	1	0	Bit Value

If the value of any bit position is 0, then the keys state is False, if it is 1, its True. It quickly becomes apparent the power of the FLAG variable- since the state of each key is contained within the same variable, testing for the combination of key states is much easier.

Would you like a simple program that illustrates setting, testing, clearing and toggling bits in a variable? Download a demo here.

Screen colour attributes are handled in the same way. See the sidebar that describes how console colours use a Flag variable. [See Appendix C "The DOS Colour Attribute

Byte"]

Bitwise Operations

We can compare Binary numbers bit by bit, with the six bit wise operations that C provides. They are:

Shift Left (<<),
Shift Right (>>),
AND (&),
OR (|), sometimes called Inclusive OR,
Exclusive OR (^), sometimes called XOR,
and Inverse(~) ,sometimes called NOT.

Be careful not to confuse bitwise operators (& and |) with the logical operators (&& and ||). The bitwise operators sometimes produce the same results as the logical operators, but they are not equivalent.

(Challenge: I've completely skipped discussing Octal number... Base 8, 3 bits... play with them,

and consider the effects of >> 3 and << 3)

Setting a Bit - inclusive OR (value | value)

Inclusive OR compares two values, and **if either bit is 1, it returns 1**. If we therefor supply one of the values with the bit we want set, we set that bit in the return value.

Decimal	Binary								Operation
5	0	0	0	0	0	1	0	1	SET bit
8	0	0	0	0	1	0	0	0	OR (|)
13	0	0	0	0	1	1	0	1	**Return**

Example: the variable FLAG currently equals 5.
We want to set Bit 3. Bit 3 = 2 to the 3rd power, or 8.

Code:

```
FLAG = 5;
result = ( FLAG | 8 );
```

result equals 13

Clearing a Bit - INVERSE and AND (value & ~value)

Clearing a Bit is a bit more complicated as it requires understanding two bitwise operands.INVERSE does exactly as its name suggests; it inverts the Bits of a single value, **it turns 0's to 1's, and 1's into 0's.** AND compares two values, and **if both bits are 1, it returns 1.** You MUST perform the INVERSE on the value you are using to clear the bit!!!

Decimal	Binary								Operation
11	0	0	0	0	1	0	1	1	CLEAR bit

~8	1	1	1	1	0	1	1	1	AND NOT (&~)
13	0	0	0	0	1	1	0	1	**Return**

:Testing a Bit - AND (value & value)

As stated above, AND compares two values, and **if both bits are 1, it returns 1.** Without using INVERSE, it can be used to test a bit.

Decimal	Binary								Operation
11	0	0	0	0	1	1	0	1	TEST a bit
4	0	0	0	0	0	1	0	0	AND (&)
4	0	0	0	0	0	1	0	0	**Return**

If the value used to test, equals the result, then the Bit is set. If the return is 0, then the bit was not set.

:Toggling a Bit - exclusive OR [XOR] (value ^ value)

Exclusive OR is used to toggle a bit; if the bit is 1, its changed to 0, if its 0, its changed to 1. This accomplished by using Exclusive OR [XOR] to compare the two values. **If both bits are the same, it returns 0, if the bits are different, it returns 1.**

Decimal	Binary								Operation
11	0	0	0	0	1	1	0	1	TEST a bit
4	0	0	0	0	0	1	0	0	XOR (^)
9	0	0	0	0	1	0	0	1	**Return**

:Bitwise Shifts (<< and >>)

Looking back at what I just showed you about binary and hexadecimal numbers, and just from what the name implies about bitwise shifts, you may be already thinking that to divide or multiply by 2,4,8,16,etc would be pretty easy, and you'd be right. The idea of shifts is pretty simple right shift of four places would turn 1010 1001 into 0000 1010 and a left shift of four places would turn 1010 1001 into 1001 0000.

When you shift values to the left, C zero-fills the lower bit positions. When you shift values to the right, the value that C places in the most-significant bit position depends on the variables type. If the variable is an unsigned type, C zero-fills the most significant bit. If the variable is a signed type, C fills the most significant bit with a 1 if the value is currently negative, or 0 if the value is positive.(This may vary between machines, though. **Use the 'unsigned' type with bit wise shifts to ensure portability**.)

Hopefully, you now have a firm grasp of the binary/decimal/hex relationship, so think about this: 0x5A >> 4 would be 0x5... .and 0x5A << 4 would be 0x5A0.

:Bitwise Operator Precedence

This is a good point to talk about precedence with bitwise operations. Bitwise shifts have a lower precedence that arithmetic operators (var_name << 4+10 would be evaluated as var_name<<(4+10), not (var_name << 4) +10). The following are in order of precedence, stating with the highest: ~,&,^,|

The precedence of the bitwise operators is lower than relational and equality operators. Be careful not to write statements like `if (value & 0x04 != 0). .` Instead of testing whether value & 0x04 isn't equal to zero, this statement will test 0x04 !=0 first, returning the result 1, which will result in value & 1.

At this point, we are through with out text attributes example. But I am going to give you another example, of one of the more common uses for the XOR.

For those of you who found the DOS Screen colour attributes example interesting, the following sidebar contains examples of manipulating the screen attribute byte. [See Appendix D "Manipulating the DOS Colour Attribute Byte"]

XOR Encryption

One of the easiest ways to encrypt data is to use the Exclusive OR operator. A value is chosen that become our 'key'. We then compare a character to be encrypted with XOR to our key. It's this simple...

Suppose we take the character G (ASCII decimal value=71) and for our key, we use the ASCII value for # (decimal 35)

XOR Encryption

Decimal	Binary								Operation
71	0	1	0	0	0	1	1	1	XOR (^)
35	0	0	1	0	0	0	1	1	
100	0	1	1	0	0	1	0	0	**Result**

The ASCII value for 100 decimal is 'd'.

To decrypt the character, we simply apply the same algorithm.

XOR Decryption

Decimal	Binary								Operation
100	0	1	1	0	0	1	0	0	XOR (^)
35	0	0	1	0	0	0	1	1	
71	0	1	0	0	0	1	1	1	**Result**

I've included the code for a very simple envryption program.. If you would like to try out this program, create a text file with the text you want to encrypt and call it txtfile.txt, compile this program, calling it..say.. xor, and at your command prompt, type
`xor <txtfile.txt >newfile.txt`

```
/*************************************************************************/
/* A Sample program using XOR encryption
/*************************************************************************/

#include <stdio.h>
#include <ctype.h>

#define KEY 0x84 /* the 'ä' character (ASCII 132) */

int main(void)
{
```

```
        int orig_char, new_char;
        while ((orig_char=getchar()) != EOF)
        {
                new_char=orig_char ^ KEY;
                if (iscntrl(orig_char) || iscntrl(new_char))
                        putchar(orig_char);
                else
                        putchar(new_char);
        }
        return 0;
}
```

For a more information on XOR Encryption, see CScene #4 "*SimpleFile Encryption using One-Time-Pad and Exclusive OR*" by Glen Gardner Jr.

I haven't throughly examined this article, but I did notice on the cover sheet his alternate title is "How I learned to love bitwise logical operations in C". While this is wrong, because Bitwise operators are NOT logical operators the article looks very interesting and well worth reading.

Bitwise Arithmetic

A few people submitted interesting examples of bitwise operators that I'm going to share with you here.

The first one was submitted by David Lee in the UK. Its very clever, although at least one person called it a "lame tired old hack", and "plain stupid now". I think you'll agree, if you haven't seen this before, you're going to find it incredibly interesting.

Its used to swap two integers in place without temporary storage:

In my sample program here, I'm going to assign two values so we can examine what's going on...

```
/*************************************************************************/
/* Swaping two integers without a temporary storage
/* - A Tired Lame Old Hack, or a Clever Example?
/* sumitted by David Lee, UK
/*************************************************************************/
#include <stdio.h>
int main(void)
{
        unsigned int a, b;
        a=112;
        b=32;
        a ^=b;  /* step 1 - 'a' now equals 80  */
        b ^= a; /* step 2 - 'b' now equals 112 */
        a ^=b;  /* step 3 - 'a' now equals 32  */
        printf("A=%d B=%d\n", a, b);
}
```

Lets look at each step:

Step 1: Tired Lame Old Hack

Decimal	Binary								Operation
112	0	1	1	1	0	0	0	0	XOR (^)
32	0	0	1	0	0	0	0	0	
80	0	1	0	1	0	0	0	0	**Result**

Step 2: Tired Lame Old Hack - this is kinda like how the encryption algorithm worked, eh?

Decimal	Binary								Operation
80	0	1	0	1	0	0	0	0	XOR (^)
32	0	0	1	0	0	0	0	0	
112	0	1	1	1	0	0	0	0	**Result**

Step 3: Tired Lame Old Hack - well, ain't this brilliant?

Decimal	Binary								Operation
112	0	1	1	1	0	0	0	0	XOR (^)
80	0	1	0	1	0	0	0	0	
32	0	0	1	0	0	0	0	0	**Result**

After seeing how its done, it's not *that* clever, is it?

```
/*************************************************************************/
/* A Bitwise Arithmetic Example
/* Submitted by Jos A. Horsmeier
/* © 1998 Jos A. Horsmeier
/*************************************************************************/
#include <stdio.h>
/* add two numbers without using the '+' operator */
unsigned int add(unsigned int a, unsigned int b)
{
        unsigned int c= 0;
        unsigned int r= 0;
        unsigned int t= ~0;
        for (t= ~0; t; t>>= 1)
         {
                r<<= 1;
                r|= (a^b^c)&1;
                c= ((a|b)&c|a&b)&1;
                a>>= 1;
                b>>= 1;
         }
        for (t= ~0, c= ~t; t; t>>= 1)
         {
                c<<= 1;
                c|= r&1;
                r>>= 1;
         }
        return c;
}

/* multiply two numbers without using the '*' operator */
unsigned int mul(unsigned int a, unsigned int b)
{
        unsigned int r;
        for (r= 0; a; b <<= 1, a >>= 1)
```

```
                if (a&1)
                        r = add(r, b);
        return r;
}

/* driver program for the above two functions */
int main(int argc, char* argv[])
{
        printf("%d*%d= %d\n", atoi(argv[1]), atoi(argv[2]),
                mul(atoi(argv[1]), atoi(argv[2])));
        printf("%d+%d= %d\n",atoi(argv[1]), atoi(argv[2]),
                add(atoi(argv[1]), atoi(argv[2])));
        return 0;
}
```

Bitwise arithmetic is generally regarded as being much faster than using the traditional C arithmetic operators, and programmers that are often resource greedy (ie- games programmers) are oftem huge proponents of Bitwise arithmetic. I haven't bench tested Horsmeier's Bitwise arithmetic functions above, so I have no idea if they are optimised. But, if you take the time to analysis Horsmeier's Bitwise arithmetic functions above, you'll be well on your way to fully understanding the power and beauty of Bitwise operations

When you feel comfortable with this tutorial, take a look at the Bitwise rotation functions in the stdlib library (_rotl and _rotr)... and bit fields.

If anyone has any questions, comments, or anything they'd like to share, please feel free to email me at gmyers@designandlogic.com.

Appendix A: Data Types - Their sizes and ranges

This is compiler dependant- see your compiler docs for your actual values.

16-bit data types, sizes, and ranges

Type	Bits	Value Range	Typical Usage
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
enum	16	-32,768 to 32,767	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4 ^ 10-38 to 3.4 ^ 1038	Scientific (7-digit precision)
double	64	1.7 ^ 10-308 to 1.7 ^ 10308	Scientific (15-digit precision)
long double	80	3.4 ^ 10-4932 to 1.1 ^ 104932	Financial (18-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

32-bit data types, sizes, and ranges

Type	Bits	Value Range	Typical Usage
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned int	32	0 to 4,294,967,295	Large numbers and loops
int	32	-2,147,483,648 to 2,147,483,647	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
enum	32	-2,147,483,648 to 2,147,483,647	Ordered sets of values
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4 ^ 10-38 to 3.4 ^ 1038	Scientific (7-digit precision)
double	64	1.7 ^ 10-308 to 1.7 ^ 10308	Scientific (15-digit precision)
long double	80	3.4 ^ 10-4932 to 1.1 ^ 104932	Financial (18-digit precision)

Appendix B: Using Unsigned Data Types for Portability

It is commonly said that for portability, you should perform bitwise shifts only on unsigned characters. You will remember that the most significant bit in a signed character is the sign bit.

When you shift values to the left, C zero-fills the lower bit positions. When you shift values to the right, the value that C places in the most-significant bit position depends on the variables type. If the variable is an unsigned type, C zero-fills the most significant bit. If the variable is a signed type, C fills the most significant bit with a 1 if the value is currently negative, or 0 if the value is positive. This may vary between machines, though. I've seen one case where the expression `a << -5` actually does a left shift of 27 bits - not exactly intuitive.

This is why it is said that you should only use the unsigned data types with bitwise shifts; while it is easy to test how your compiler will handle these shifts, using unsigned data types ensures portability.

Appendix C: The Colour Attribute Byte

It wasn't until I was developing a DOS console user interface, and I needed to be able to store the text screen attributes, then restore them, that I developed an appreciation for the use of bitwise operations. The attribute byte for a text screen stores the 16 possible text and 16 possible background colours (plus the ability to make the background 'blink') in the 8 bits. This is accomplished because all colours are made from the 3 primary colours; Red, Green, and Blue. (we are speaking in terms of light, not pigment- If you add Red, Green, and Blue paint together, you get Black. If you add Red, Green, and Blue light together, you get White).

Examine the diagram below illustrating the structure of **Attribute Byte**:

	Background				Foreground			
Bit Position	**7**	6	5	4	3	2	1	**0**
Bit Value	0	0	1	0	1	0	1	0
	X	**R**	**G**	**B**	**X**	**R**	**G**	**B**

In our example above, the binary number 0010 1010 = 42 = 0x2A

It is LIGHTGREEN text (GREEN + INTENSITY BIT) on a CYAN background (BLUE+GREEN).

The most significant bit in the text attribute is the Blink bit..if its 0, the character doesn't blink, if its 1, it does. If we wanted our example to blink, its binary value would be 1011 1010=0xBA=186 decimal.

Below is a chart of the Text (Foreground) and Background colours:

COLOUR	DEC	HEX	BIN	USE
Black	0	0	0000	Text/Bkgrnd
Blue	1	1	0001	Text/Bkgrnd
Green	2	2	0010	Text/Bkgrnd
Cyan (Blue+Green)	3	3	0011	Text/Bkgrnd
Red	4	4	0100	Text/Bkgrnd
Magenta (Red+Blue)	5	5	0101	Text/Bkgrnd
DarkYellow or Brown (Green+Red)	6	6	0110	Text/Bkgrnd
Lightgray (Red+Green+Blue)	7	7	0111	Text/Bkgrnd
Darkgray (Black+Intensity)	8	8	1000	Text
LightBlue (Blue+Intensity)	9	9	1001	Text
LightGreen (Green+Intensity)	10	A	1010	Text
LightCyan (Cyan+Intensity)	11	B	1011	Text
LightRed (Red+Intensity)	12	C	1100	Text
LightMagenta (Magenta+Intensity)	13	D	1101	Text
Yellow (DarkYellow+Intensity)	14	E	1110	Text
White (Lightgray+Intensity)	15	F	1111	Text

Appendix D: Manipulating the Colour Attribute Byte

The bitwise AND Operator (&)

For our example, we need to determine the colour of the text and the background independently. How do we read just the first four bits? Easy, the AND bitwise operator.

The bitwise AND operator examines each bit and returns a comparison. If a bit from Number A is 1, AND the corresponding bit from Number B is 1, the result is 1.Lets assume we have a BLUE Background with LIGHTCYAN text, and the attribute byte is stored in txtcolor.attr as 27 decimal, or 0001 10111 binary.

AND to Test the Foreground

Decimal	Binary									Operation
27	0	0	0	1		1	0	1	1	TEST bit
16	0	0	0	0		1	1	1	1	AND (&)
11	0	0	0	0		1	0	1	1	**Result**

we'd write that as... (16 & txtcolor.attr)... and the result would be decimal 11, LIGHTCYAN

And to Test the Background

Decimal	Binary									Operation
27	0	0	0	1		1	0	1	1	TEST bit
112	0	1	1	1		0	0	0	0	AND (&)
16	0	0	0	1		0	0	0	0	**Result**

and then the shift (112 & txtcolor.attr)>>3 the result would be 1, or BLUE

The bit wise AND can also be used to test a bit as follows:

Suppose we want to test for the Intensity bit (bit 3)

if ((txtcolor.attr & 0x08) == 0x08) {...} /* test if bit 3 is = 1 */

Decimal	Binary									Operation
27	0	0	0	1		0	0	1	1	TEST bit
8	0	0	0	0		1	0	0	0	AND (&)
8	0	0	0	0		1	0	0	0	**Result**

we could also write this as:

if ((txtcolor.attr & 0x08)>>3) {...}

this would shift the answer to the bit 0 position, and the result would be 1 if true and 0 if false then. In our case, the if statement would test true.

The Bitwise Inclusive OR (|)

The bitwise OR operator examines each bit and returns a comparison. If the bit from Number A is 1 OR the bit from Number B is 1, then the result is 1. Suppose our foreground is MAGENTA, and our background is GREEN , therefor our attribute byte is 37 decimal, and we want to make MAGENTA into LIGHTMAGENTA, and BLINKING so, we add the blinking bit 7 and the high intensity bit 3.

OR to Set a Bit

Decimal	Binary									Operation
37	0	0	1	0		0	1	0	1	SET bit
136	1	0	0	0		1	0	0	0	OR (	)
173	1	0	1	0		1	1	0	1	**Result**

We'd write that as (136 | txtcolor.attr)

A nice way to add a bit to the attribute. ..isn't it. But if the bit is already set as you want it... it doesn't change

We could get our bits as we did in the AND example, using OR, and a little subtraction.

Examine this example:

Foreground colour- CYAN, decimal 3, binary 0 0 1 1

Background colour- BLUE, decimal 1, binary 0 0 0 1

OR to Test a Bit

Decimal	Binary	Operation

19	0	0	0	1		0	0	1	1	TEST bit
240	1	1	1	1		0	0	0	0	OR (|)
243	1	1	1	1		0	0	1	1	**Result**

now, if we subtract 240 from the result, it would give us decimal 3, CYAN.

we'd write it like this (240 | txtcolor.attr)-240

INVERSE (~) and Exclusive OR [XOR] (^)

The next operator, INVERSE, does exactly as the name suggests, it turns 1's into 0's, and vice versa.

Consider how we could use Inverse to clear a bit. Suppose we have LIGHTCYAN text on a BLUE background, and we want clear the Intensity bit, to make the text CYAN

INVERSE and AND to clear a Bit

Decimal	Binary									Operation
27	0	0	0	1		1	0	1	1	CLEAR bit
~8	1	1	1	1		0	1	1	1	AND (&)
19	0	0	0	1		0	0	1	1	**Result**

And you'd write it as **txtcolor.attr &= ~0x08**

NOTE- txtcolor.attr should be of the unsigned char type (8 bits)

What if you wanted to note only if the bits are different?..You guessed it... the exclusive OR.

Exclusive OR results in 0 if the bits are the same (either both 0 or both 1) and results in 1 if they are different.

Exclusive OR

Decimal	Binary									Operation
162	1	0	1	0		0	0	1	0	XOR (^)
203	1	1	0	0		1	0	1	1	
105	0	1	1	0		1	0	0	1	**Result**

now, check this out... you can toggle a bit (if its 1, make it 0 or if its 0, make it 1) with Exclusive OR.

Lets look at bit 3 in our example value 162..

Toggling with Exclusive OR

Decimal	Binary									Operation
162	1	0	1	0		0	0	1	0	TOGGLE Bit
8	0	0	0	0		1	0	0	0	XOR (^)
170	1	0	1	0		1	0	1	0	**Result**

this could be written as, txtcolor.attr ^= 0x08;

Ain't that sweet?

So, an INVERSE of an EXCLUSIVE OR would notify you of what bits are the SAME..

Inverse

Decimal	Binary									Operation
211	1	1	0	1		0	0	1	1	Inverse (~)
44	0	0	1	0		1	1	0	0	**Result**

Acknowledgements

Taylor Carpenter, Jos Horsmeier, David Lee, Michael Rubenstein, James Hu and Luis Grave.

Bibliography

King, K.N., *C Programming, A Modern Approach*, W.W.Norton Company

Maljugin, V., **J. Izrailevich**, **S. Lavin**, and **A. Sopin**, *The Revolutionary Guide to Assembly language*, WROX Press.

Kernigan, B.W., and **D.M. Richie**, *The C Programming Language*, 2nd Edition, Prentice-Hall.

Jamsa, K., and **L. Klander**, *The C/C++ Programmers Bible*, Jamsa Press.

Summit, S., *C Programming FAQ*, ftp://rtfm.mit.com/.