C Programming for Techies

Bits, the most fundamental element of data, can be tricky to deal with. While C offers some bitwise operators, unless you have a clear fix on what they're up to, bits may still escape you.

by Steve Rimmer

Bitwise manipulations under C language programs can be confusing. In many applications it would be convenient to deal with byte oriented data objects as bit fields... something C isn't really set up to let you do. However, if you really understand what's happening at the bit level of C's operations you can make it do a lot of clever, and hitherto unexplored, things.

Unfortunately, bitwise operations are exceedingly hard to fathom at first, and their operators under C are among the most obtuse.

This month we're going to look at some of the mechanics of bits and C.

Bits of Data

A bit is, of course, a simple binary object. Bits can be either on or off. In binary notation... something which C doesn't actually support... a bit that's off is represented by zero and a bit that's on is represented by one.

In microprocessor based computers a byte has largely standardized on being eight bits wide. This isn't true in other environments. If you read through some of the obscure parts of Kernighan and Ritchie's book *The C Language*, you'll find reference to mainframe environments with odd sized bytes. However, you can probably ignore the possibilities of a mainframe system in your basement for the time being.

An eight bit byte can be regarded as being comprised of two nybbles, each having four bits. This works out well when you look at the hexadecimal notation for a byte... something most PC based C compilers *do* support. Here's the binary representation of a byte:

```
11000111
```

and here's its hexadecimal representation:

```
0xc7
```

This works out to two nybbles in both cases. The upper nybble of the byte is 1100 in binary and 0xc in hexadecimal. This is twelve in human numbers.

You only really need to know sixteen values, then, as both nybbles work the same way.

```
DecimalBinaryHex
000000
100011
200102
300113
401004
501015
601106
701117
810008
910019
101010A
111011B
121100C
131101D
141110E
151111F
```

C provides a rich set of operators to deal with bits. These bitwise, or *boolean* operators allow you to both test and manipulate bits. However, using them effectively calls for a bit of cunning.

The most commonly used... and most frequently misapplied... operator is AND, represented by &. The AND operator will return a value which contains those bits which are common in the two objects being ANDed together. As with all C language bitwise operators, you can AND *char*s, *int*s and *long*s. For the moment, we'll work with *char*s, which are equivalent to bytes.

Here's an example of the use of AND.

```
int a=0xc7,b=0x7f;

printf("a AND b = %X",a & b);
```

Let's see what this should do. The binary representation of this expression would be:

```
a = 11000111
b = 01111111
```

The result will be a byte having its bits set in those positions where both *a* and *b* also have them set.

```
r = 01000111
```

This has a hexadecimal value of 0x47.

There are a number of obvious uses for the AND operator. It works as a crude... but fast... form of modulus operator if you want to take the modulus of an integer and the modules happens to be an even power of two. For example, you could replace *n % 8* with *n & 7*.

The AND operator is also used to mask off unwanted bits. For example, older versions of WordStar produced text files which were essentially pure ASCII save that some of the characters had their most significant bits set as a signal to some of WordStar's internals. You could turn a WordStar file back into an ASCII file by ANDing every byte with 0x7f. This value has all its bits set save for the most significant one.

It's probably worth pointing out the difference between the bitwise AND operator and the logical one under C. The former is *&* and the latter is *&&*. This often sneaks up and bites you somewhere private if you forget about it. For example, consider this conditional statement.

```
if(a && b) {
/* some code goes here */
}
```

This means to do whatever's in the conditional if both *a* and *b* are non-zero. Occasionally people forget the second ampersand.

```
if(a & b) {
/* some code goes here */
}
```

This means to do whatever is in the conditional if *a* AND *b* works out to a non-zero value, that is, if *a* and *b* have some bits in common. This can be a very hard bug to track down.

The OR operator is represented by the vertical rule character, /. It will return a value which contains set bits in all those positions which had them set in either argument to it. For example,

```
int a=0x38,b=0x81;

printf("a OR b = %X",a | b);
```

Once again, we can see how this works if we look at the two values in question in binary.

```
a = 00111000
b = 10000001
```

The result, then, would be

```
r = 10111001
```

This works out to 0xb9.

Once again, there's a logical OR operator, //, which should not be confused with the bitwise one.

The exclusive OR operator is perhaps the most confusing and the least used. It's represented by the carat character, ^. Fortunately, there is no logical exclusive OR operator to muddy the waters. The operation of the exclusive OR function is to invert bits.

If *a* and *b* are bytes, *a ^ b* will cause all the bits in *a* to be inverted wherever there are set bits in *b*. Let's see how that works.

```
int a=0x0f,b=0x55;

printf("a XOR b = %X",a ^ b);
```

Once again, we can work out the whole seething mess in binary.

```
a = 00001111
b = 01010101
```

The result would be

```
r = 01011010
```

This amounts to 0x5a in hexadecimal.

The exclusive OR operator is useful for toggling bits.

Finally, there's the negation operator, which is represented by the tilde character, ~. This simply inverts all the bits in a byte. These two expressions will produce the same result... you might want to stop and see if you can figure out why.

```
b = ~a;
b = a ^ 0xff;
```

This assumes in both cases that *a* is a *char*.

Aside from being able to manipulate bits under C, you can alter their positions in a byte. There are two operators for this, the left shift operator, << and the right shift operator, >>. These often get confused with the greater than and lesser than logical operators, with the same sorts of results as the confusion about the logical and bitwise AND operators discussed above.

Shifting bits involves moving all the bits in an object left or right by a defined amount. For example,

```
int a=0x34;

printf("a shifted left
one = %X\n",a << 1);
printf("a shifted right
one = %X\n",a >> 1);
```

Turning once more to the binary representation of things,

```
a = 00110100
```

In order to shift *a* left, we must lose the leftmost bit and add a zero bit onto the right end of the byte.

```
r = 01101000
```

This works out to 0x68.

To shift *a* right by a bit, you would do the opposite, that is, throw away the rightmost bit and add a zero bit to the left end.

```
r = 00011010
```

This works out to 0x1a.

The decimal values for these numbers may be more enlightening. The original value of *a* was 52. Shifted left by one it became 104. Shifted right it became 26.

Each time you shift a value left by one place, you multiply it by two. Each time you shift it right by one place you divide it by two. If you have to perform integer multiplication or division by even powers of two, using bit shifts is eminently more desirable than using normal integer math. A bit shift will typically take about a fiftieth of the processor time of a multiplication or division instruction.

Plowing the Bit Fields

In working with applications which require a lot of bitwise manipulation... especially in dealing with bitmapped graphics... you will usually find that you have to treat a string of bytes as a string of bits. For example, if you want to set a pixel on a graphics screen, you must locate the pixel in question in the line of bytes which makes up the screen line in memory.

In the following examples, *n* will be the location of the bit to be dealt with a *p* will point to the line of bytes which contains the bits in question.

Finding the byte which contains bit *n* in the bit field is easy. Since there are eight bits in a byte, bit *n* must reside somewhere in the byte numbered *n / 8*. Because eight is an even power of two... the third power... we can make this calculation much faster by representing it as *n >> 3*. The byte in question, then is *p[n>>3]*.

Finding the bit in question in the specific byte requires a bit more stealth. We wish to create what's called a *mask*, a byte having a single bit set representing the position of the bit in question. This can be done using the expression (0x80 >> (n & 7)).

Let's see what's going on here. The value 0x80 is a byte having one bit set, this being its most significant bit. The expression *n & 7* is, in fact, *n* mod eight, or the portion of the bit position value *n* which represents the bit position in the byte in question. Note that when we found the byte by shifting *n* right by three, the bits masked by the value seven... the first three... are the ones which were thrown away.

This is how you would turn on bit *n* in bit field *p*.

```
p[n>>3] |= (0x80 >> (n & 7));
```

This expression will locate the byte which bit *n* resides in and create a mask to represent the appropriate bit. It will then OR the byte with the mask. The single bit which is set in the mask will turn on the corresponding bit in the byte in question. If the bit is already on, nothing will happen.

This is how you would turn off bit *n* in bitfield *p*. It's a bit more involved.

```
p[n>>3] &= ~(0x80 >> (n & 7));
```

To turn off a bit, we must mask off that bit. You could so this by ORing the bit to make sure it was on and the exclusive ORing it to turn it off. An easier way is to use the same mask as we did in turning the bit on and inverting it, such that it becomes a mask selecting all the bits except for the one in question. If we AND this with the byte in question, the bit to be turned off will be masked, or set to zero.

Finally, this will toggle bit *n* in bitfield *p*. Note that this expression does not know the state of bit *n*... it simply inverts it.

```
p[n>>3] ^= (0x80 >> (n & 7));
```

Just a Bit Faster

Code optimization... the process of trying to streamline a program to make it run faster... would look at the foregoing expressions a bit suspiciously. Bitfield operations usually involve a lot of bits, and there seem to be more operations in these expressions than there need be. Code optimization invariably tries to pre-package some of the computations in a complex expression into a table. This is one of those cases wherein you can speed things up with one.

Consider the expression *(0x80 >> (n & 7))*. It has two actual operations going on, to wit, ANDing *n* by seven and then shifting 0x80 right by the result. It can only produce eight possible results. We can speed up all the aforementioned bitfield operations by creating a table of mask values. Here it is.

```
char masktable[8]=
0x80,0x40,0x20,0x10,0x08,0x0
4,0x02,0x01;
```

We can the reduce the expression by one operation. It becomes *masktable[n & 7]*. Hence, to turn on bit *n* in bitfield *p*, you would say

```
p[n>>3] |=masktable[n & 7];
```

As you begin to work with bitwise operators under C you'll probably find a wealth of uses for them. Aside from speeding up some multiplication and division operations, they provide a powerful way to compress the space you need to store flags, small numbers, pixels and other things which don't fit neatly into the rigid confines of a byte. ❐