C Microcontroller Bit Manipulations Explained

Setting Bits

If you're new to microcontroller C programming, statements such as the following may not be especially clear:

```
DDRC |= (1 << PC5);           // set Port C pin PC5 for output
```

Let's brake this down, starting with DDRC and PC5.

If you search through the Dependencies (see the .h files in Atmel Studio) you will find that "PC5" is defined to be "5". Keeping this in mind and using our knowledge of C operators we can break down the above statement as follows:

```
DDRC |= (1 << PC5);
DDRC = DDRC | (1 << 5);     // note that the "|" char is a bitwise OR
```

Next let's lake (1 << 5) in isolation. The 1 on the left is simply stating the number 1, and the << 5 are indicating to shift the bits on the left of the operator to the left by 5 positions. If we wrote out the number 1 in base 2 to fill an 8 bit register or 8 bit variable it would look like this:

```
00000001
```

That's all we're doing when we specify the 1 above, now if we apply the bit shift left by 5 we get:

```
00100000
```

```
In other words, (1 << 5) == 00100000
```

Before we get back to registers, lets quickly review bit-wise operator basics. "|" is bit-wise OR, "&" is bit-wise AND, and "X" is a bit that could be 0 or 1:

```
X | 0 = X
X | 1 = 1
X & 0 = 0
X & 1 = X
```

Also remember the tilde character "~" is a bitwise NOT.

Now let's get back to registers. If we would like to set the 5th bit of DDRC, we can bit-wise OR (1 << 5), which is 00100000, with DDRC, as follows:

```
DDRC contains:    XXXXXXXX
bit-wise OR with: 00100000
DDRC now equals:  XX1XXXXX
```

Let's put all of the above together to break down the original statement from the beginning:

```
DDRC |= (1 << PC5);       // set DDRC bit PC5, start with breaking out the "|=" operator:
```

```
DDRC = DDRC | (1 << PC5);        // next substitute 5 for PC5, as defined in the dependencies:
DDRC = DDRC | (1 << 5);          // next re-write the "1" as an 8 bit binary value:
DDRC = DDRC | (00000001 << 5);   // then apply the bit shift:
DDRC = DDRC | (00100000);        // now apply the bit-wise OR:

DDRC contains:    XXXXXXXX
bit-wise OR with: 00100000
DDRC now equals:  XX1XXXXX
```

So we have successfully set the 5th bit without changing the others. Make sense? Great, now on to clearing bits.

Clearing Bits

Let's look at the syntax for clearing bits, which at first looks more complicated but is not difficult to work through:

```
DDRD &= ~(1 << PD2);          // clear DDRD bit 2, sets PD2 (pin 4) for input
```

We'd like to use PD2 as output, so we clear DDRD bit 2. Let's break it down:

```
DDRD &= ~(1 << PD2);             // clear DDRD bit 2, start by breaking out the "&=" operator
DDRD = DDRD & ~(1 << PD2);       // next substitute 2 for PD2, as defined in the dependencies:
DDRD = DDRD & ~(1 << 2);         // next write out the "1" as an 8 bit binary value:
DDRD = DDRD & ~(00000001 << 2);  // next apply the bit shift:
DDRD = DDRD & ~(00000100);       // now apply the bit-wise NOT:
DDRD = DDRD & (11111011);        // now apply the bit-wise AND:

DDRD contains:     XXXXXXXX
bit-wise AND with: 11111011
DDRD now equals:   XXXXX0XX
```

So we have cleared the 2nd bit and left the others alone, as we were intending to do. For a final example, let's consider the bit toggle using the Exclusive OR operator "^" from the first tutorial in this series.

Toggling Bits

First a quick review of an XOR truth table:

```
A  B  XOR
0  0   0
0  1   1
1  0   1
1  1   0
```

In other words, if we XOR anything with a 0, it remains what is was before:

```
0 ^ 0 = 0
1 ^ 0 = 1
X ^ 0 = X
```

And if we XOR anything with a 1, it toggles what the value was before (Y means the toggled value of X):

```
0 ^ 1 = 1
1 ^ 1 = 0
X ^ 1 = Y
```

Next let's evaluate the toggle expression from the 1st tutorial in this series, recall we were attempting to toggle the state of the LED on PC5:

```
PORTC ^= (1 << PC5); // toggle state of LED on PC5, start by breaking out the "^=" operator:
PORTC = PORTC ^ (1 << PC5);      // next substitute 5 for PC5, as defined in the dependencies:
PORTC = PORTC ^ (1 << 5);        // next write out the "1" as an 8 bit binary value:
PORTC = PORTC ^ (00000001 << 5); // next apply the bit shift:
PORTC = PORTC ^ (00100000);            // now apply the XOR:

PORTC contains:    XXXXXXXX
bit-wise XOR with: 00100000
PORTC now equals:  XXYXXXXX
```

So we have successfully toggled the 5th bit without changing the others!

Conditionals

We can apply the same principles from above to conditional statements in our C program. For example let's evaluate the following:

```
if(PIND & (1 << PD2)) { // check if register PIND, pin PD2 is set, substitute 2 for PD2:
if(PIND & (1 << 2)) {            // next write out the "1" as an 8 bit binary value
if(PIND & (00000001 << 2)) {     // next apply the bit shift
if(PIND & (00000100)) {          // next apply the bit-wise AND

PIND contains:     XXXXXXXX
bit-wise AND with: 00000100
equals:            00000X00
```

Remember that any positive number in C will evaluate to true and zero will evaluate to false. Therefore we can say that if PIND bit 2 was set our result with be non-zero and therefore will evaluate to true, if PIND bit 2 was not set our result will be zero and therefore evaluate to false. Note that because we are not assigning anything to PIND the value is only being checked, not changed.

After considering the above to check if a bit is set, we can simply put an additional set of brackets around or expression and then apply the logical NOT operator to check if is a bit is clear:

```
if(!(PIND & (1 << PD2))) {       // check if register PIND, pin PD2 is clear
```

To check if bits are set or clear, many people prefer to use macros, for example:

```
#define BIT_IS_SET(byte, bit) (byte & (1 << bit))
```

```
#define BIT_IS_CLEAR(byte, bit) (!(byte & (1 << bit)))
```

Now we can simply do:

```
if(BIT_IS_SET(PIND, PD2)) {      // check if bit is set using the above macro
if(BIT_IS_CLEAR(PIND, PD2)) {    // check if bit is clear using the above macro
```

Atmel Studio has the predefined macros `bit_is_set(byte, bit)` and `bit_is_clear(byte, bit)`, but I suggest using your own for 2 reasons:

1) The predefined macros are in lower case, which makes them look like functions and can cause confusion
2) If you use your own macros you know exactly where they are and what they are doing

That completes the basic of bit manipulations in microcontroller C!