PIC Programming in Assembly

(http://www.mstracey.btinternet.co.uk/index.htm)

Tutorial 1

Good Programming Techniques.

Before we get to the nitty gritty of programming the PIC, I think now is a good time to explain some good programming techniques.

If you type a ; (semicolon) anywhere in your program, the compiler will ignore anything after it until the carriage return. This means we can add comments in our program to remind us of what on earth we were doing in the first place. This is good practice, even for the simplest programs. You may well fully understand how your program works now, but in a few months time, you may be scratching your head. So, use comments wherever you can – there is no limit.

Secondly, you can assign names to constants via registers (more about these later). It makes it far easier to read in English what you are writing to, or what the value is, rather than trying to think of what all these numbers mean. So, use real names, such as COUNT. Notice that I have put the name in capitals. This makes it stand out, and also means that (by convention) it is a constant value.

Thirdly, put some kind of header on your programs by using the semi-colons. An example is below:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Author :                                                  ;
; Date :                                                    ;
; Version:                                                  ;
; Title:                                                    ;
;                                                           ;
 ; Description:                                             ;
;                                                           ;
;                                                           ;
;                                                           ;
;                                                           ;
;                                                           ;
;                                                           ;
;                                                           ;
;                                                           ;
;                                                           ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Notice that I have made a kind of box by using the semi-colons. This just makes it look neat.

Finally, try and document the program on paper as well. You can either use flow charts or algorithms or anything else you want. This will help you in writing your program, step by step.

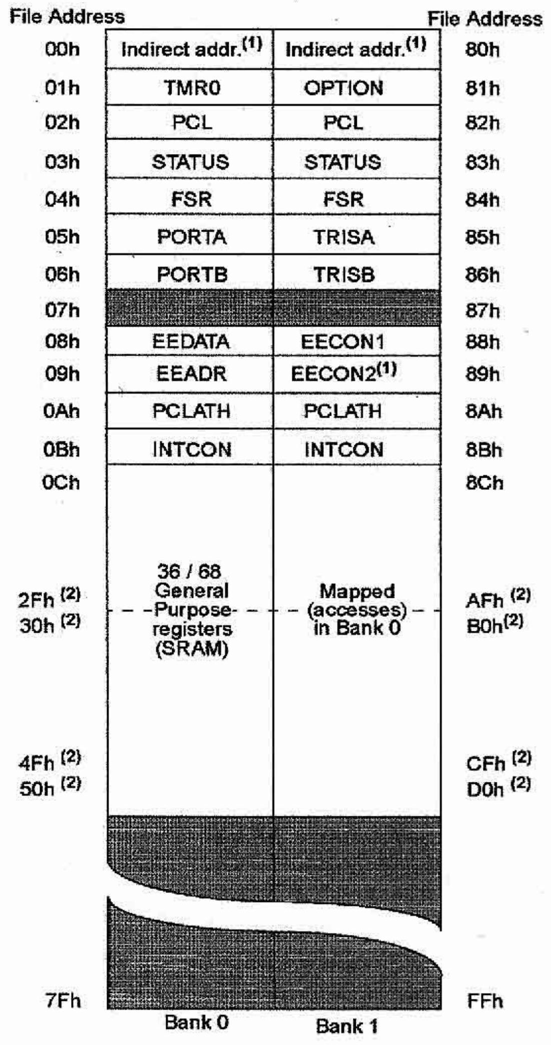
Right, that's the lecture over with, lets move on to the real stuff.

Tutorial 2

The Registers.

A register is a place inside the PIC that can be written to, read from or both. Think of a register as a piece of paper where you can look at and write information on.

The figure below shows the register file map inside the PIC16F84. Don't worry if you haven't come across anything like this before, it is only to show where the different bits and pieces are inside the PIC, and will help explain a few of the commands.



▨ Unimplemented data memory location; read as '0'.

Note 1: Not a physical register.
2: The address depends on the device used. Devices with 36 bytes end at 2Fh, devices with 68 bytes end at 4Fh.

First thing you will notice is that it is split into two - Bank 0 and Bank 1. Bank 1 is used to control the actual operation of the PIC, for example to tell the PIC which bits of Port A are input and which are output. Bank 0 is used to manipulate the data. An example is as follows: Let us say we want to make one bit on Port A high. First we need to go to Bank 1 to set the particular bit, or pin, on Port A as an output. We then come back to Bank 0 and send a logic 1 (bit 1) to that pin.

The most common registers in Bank 1 we are going to use are STATUS, TRISA and TRISB. The first allows us to come back to Bank 0, TRISA allows us to select which pins on Port A are output and which are input, TRISB allows us to select which pins on Port B are output and which are input. The SELECT register in Bank 0 allows us to switch to Bank 1.

Let us take a closer look at these three registers.

STATUS

To change from Bank 0 to Bank 1 we tell the STAUS register. We do this by setting bit 5 of the STATUS register to 1. To switch back to Bank 0, we set bit 5 of the STATUS register to 0. The STATUS register is located at address 03h (the 'h' means the number is in Hexadecimal).

TRISA and TRISB.

These are located at addresses 85h and 86h respectively. To program a pin to be an output or an input, we simply send a 0 or a 1 to the relevant bit in the register. Now, this can either be done in binary, or hex. I personally use both, as the binary does help visualize the port. If you are not conversant with converting from binary to hex and vice versa, then use a scientific calculator.

So, on Port A we have 5 pins, and hence 5 bits. If I wanted to set one of the pins to input, I send a '1' to the relevant bit. If I wanted to set one of the pins to an output, I set the relevant bit to '0'. The bits are arranges in exactly the same way as the pins, in other words bit 0 is RA0, bit 1 is RA1, bit 2 is RA2 and so on. Let's take an example. If I wanted to set RA0, RA3 and RA4 as outputs, and RA1 and RA2 as inputs, I send this: 00110 (06h). Note that bit zero is on the right, as shown:

Port A Pin	RA4	RA3	RA2	RA1	RA0
Bit Number	4	3	2	1	0
Binary	0	0	1	1	0

The same goes for TRISB.

PORTA and PORTB

To send one of our output pins high, we simply send a '1' to the corresponding bit in our PORTA or PORTB register. The same format follows as for the TRISA and TRISB registers. To read if a pin is high or low on our port pins, we can perform a check to see if the particular corresponding bit is set to high (1) or set to low (0)

Before I give an example code, I need to explain just two more register – w and f.

W

The W register is a general register in which you can put any value that you wish. Once you have assigned a value to W, you can add it to another value, or move it. If you assign another value to W, its contents are overwritten.

An Example Code.

I am going to give you some example code on what we have just learnt. Don't try and compile this yet, we will do that when we come to our first program. I am just trying to show how the above is actually programmed and introduce a couple of instructions along the way. I am going to set up Port A as per the example above.

First, we need to switch from Bank 0 to Bank 1. We do this by setting the STATUS register, which is at address 03h, bit 5 to 1.

```
BSF  03h,5
```

The BSF Means Bit Set F. The letter F means that we are going to use a memory location, or register. We are using two numbers after this instruction – 03h, which is the STATUS register address, and the number 5 which corresponds to the bit number. So, what we are saying is "Set bit 5 in address 03h to 1".

We are now in Bank 1.

```
MOVLW  b'00110'
```

We are putting the binary value 00110 (the letter b means the number is in binary) into our general purpose register W. I could of course have done this in hex, in which case our instruction would be:

```
MOVLW 06h
```

Either works. The MOVLW means ‘Move Literal Value Into W’, which in English means put the value that follows directly into the W register.

Now we need to put this value onto our TRISA register to set up the port:

```
MOVWF 85h
```

This instruction means “Move The Contents Of W Into The Register Address That Follows”, in this case the address points to TRISA.

Our TRISA register now has the value 00110, or shown graphically:

Port A Pin	RA4	RA3	RA2	RA1	RA0
Binary	0	0	1	1	0
Input/Output	O	O	I	I	O

Now we have set up our Port A pins, we need to come back to Bank 0 to manipulate any data.

```
BCF 03h,5
```

This instruction does the opposite of BSF. It means “Bit Clear F”. The two numbers that follow are the address of the register, in this case the STATUS register, and the bit number, in this case bit 5. So what we have done now is set bit 5 on our STAUS register to 0

We are now back in Bank 0.

Here is the code in a single block:

```
BSF
03h,5
;Go to Bank 1
MOVLW
06h
 ;Put 00110
into W
MOVWF
85h
;Move 00110
onto TRISA
BCF
03h,5
;Come back to
Bank 0
```

Read this through a couple of times, until it is you can follow it. So far we have looked at 4 instructions. Only 31 to go!

Tutorial 3

Writing To the Ports.

In the last tutorial, we I showed you how to set up the IO port pins on the PIC to be either input or output. In this tutorial, I am going to show you how to send data to the ports. In the next tutorial, we will finish off by flashing an LED on and off which will include a full program listing and a simple circuit diagram so that you can see the PIC doing exactly what we expect it to. Don't try and compile and program your PIC with the listings here, as they are examples only.

First, let us set up Port A bit 2 as an output:

```
bsf          03h,5        ;Go to Bank 1
movlw        00h          ;Put 00000 into W
movwf        85h          ;Move 00000 onto TRISA – all pins
set to output
bcf          03h,5        ;Come back to Bank 0
```

This should be familiar from the last tutorial. The only difference is that I have set all of the pins on Port A as output, by sending 0h to the tri-state register.

Now what he have to do is turn an LED on. We do this by making one of the pins (the one with the LED connected to it) high. In other words, we send a ‘1’ to the pin. This is how it’s done (note the comments for an explanation of each line):

```
movlw           02h     ;Write 02h to the W register.  In binary this is 00010, which
                         ;puts a ‘1’ on bit 2 (pin 18) while keeping the other pins to ‘0’

movwf           05h     ;Now move the contents of W (02h) onto the PortA, whose
                         ;address is 05h
```

So, now our LED is on, we now need to turn it off:

```
movlw           00h     ;Write 00h to the W register.  This puts a ‘0’ on all pins.

movwf           05h     ;Now move the contents of W (0h) onto the Port A, whose
                         ;address is 05h
```

So, what we have done is turn the LED on then off once.

What we want is for the LED to turn on then off continuously. We do this by getting the program to go back to the beginning. We do this by first defining a label at the start of our program, and then telling the program to keep going back there.

We define a label very simply. We type a name, say START, then type the code:

```
Start       movlw       02h     ;Write 02h to the W register.  In binary this is
                                  ;00010, which puts a '1' on pin 2 while keeping
                                  ;the other pins to '0'

    movwf       05h     ;Now move the contents of W (02h) onto the
                          ;PortA, whose address is 05h

    movlw       00h     ;Write 00h to the W register.  This puts a '0' on
                          ;all pins.

    movwf       05h     ;Now move the contents of W (0h) onto the Port
                          ;A, whose address is 05h

                goto        Start     ;Goto where we say Start
```

As you can see, we first said the word 'Start' right at the beginning of the program. Then, right at the very end of the program we simply said 'goto Start'. The 'goto' instruction does exactly what it says.

This program will continuously turn the LED on and off as soon as we power up the circuit, and will stop when we remove power.

I think we should look at our program again:

```
                bsf         03h,5
                movlw       00h
                movwf       85h
                bcf         03h,5
Start           movlw       02h
                movwf       05h
                movlw       00h
                movwf       05h
                goto        Start
```

OK, I know I have left the comments off. But, do you notice that all we can see are instructions and numbers? This can be a little confusing if you are trying to debug the program later, and also when you write the code you have to remember all of the addresses. Even with the comments in place, it can get a bit messy. What we need is to give these numbers names. This is accomplished by another instruction: 'equ'.

The ‘equ’ instruction simply means something equals something else. It is not an instruction for the PIC, but for the assembler. With this instruction we can assign a name to a register address location, or in programming terms assign a constant. Let us set up some constants for our program, then you will see how much easier to read the program is.

```
STATUS        equ  03h          ;this assigns the word STATUS to the value of 03h,
                                  ;which is the address of the STATUS register.

TRISA         equ 85h           ;This assigns the word TRISA to the value of 85h,
                                  ;which is the address of the Tri-State register for PortA

PORTA         equ 05h           ;This assigns the word PORTA to 05h which is the
                                  ;address of Port A.
```

So, now we have set up our constant values, let us put these into our program. The constant values must be defined before we can use them, so to be sure always put them at the start of the program. I will re-write the program without comments again, so that you can compare the previous listing to the new one:

```
STATUS        equ 03h
TRISA         equ 85h
PORTA         equ 05h

              bsf           STATUS,5
              movlw         00h
              movwf         TRISA
              bcf           STATUS,5

Start         movlw         02h
              movwf         PORTA
              movlw         00h
              movwf         PORTA
              goto          Start
```

Hopefully, you can see that the constants make following the program a little easier, even though we still have not put the comments in. However, we are not quite finished.

Tutorial 4

Delay Loops.

There is one slight drawback to our flashing LED program. Each instruction takes one clock cycle to complete. If we are using a 4MHz crystal, then each instruction will take 1/4MHz, or 1uS to complete. As we are using only 5 instructions, the LED will turn on then off in 5uS. This is far too fast for us to see, and it will appear that the LED is permanently on. What we need to do is cause a delay between turning the LED on and turning the LED off.

The principle of the delay is that we count down from a previously set number, and when it reaches zero, we stop counting. The zero value indicates the end of the delay, and we continue on our way through the program.

So, the first thing we need to do is to define a constant to use as our counter. We will call this constant COUNT. Next, we need to decide how big a number to start counting from. Well, the largest number we can have is 255, or FFh in hex. Now, as I mentioned in the last tutorial, the equ instruction assigns a word to a register location. This means that whatever number we assign our COUNT, it will equal the contents of a register.

If we try and assign the value FFh, we will get an error when we come to compile the program. This is because location FFh is reserved, and so we can't access it. So, how do we assign an actual number? Well, it takes a little bit of lateral thinking. If we assign our COUNT to the address 08h, for example, this will point to a general purpose register location. By default, the unused locations are set to FFh. Therefore, if COUNT points to 08h, it will have the value of FFh when we first switch on.

But, I hear you cry, how do we set COUNT to a different number? Well, all we do is 'move' a value to this location first. For example, if we wanted COUNT to have a value of 85h, we can't say COUNT equ 85h because that is the location of out Tri-State register for Port A. What we do is this:

```
movlw  85h              ;First put the value of 85h in the W register

movwf  08h              ;Now move it to our 08h register.
```

Now, when we say COUNT equ 08h, COUNT will equal the value 85h. Subtle, isn't it!

So, first we define our constant:

```
        COUNT       equ     08h
```

Next we need to decrease this COUNT by 1 until it reaches zero. It just so happens that there is a single instruction that will do this for us, with the aid of a 'goto' and a label. The instruction we will use is:

```
        DECFSZ      COUNT,1
```

This instruction says 'Decrement the register (in this case COUNT) by the number that follows the comma. If we reach zero, jump two places forward.' A lot of words, for a single instruction. Let us see it in action first, before we put it into our program.

```
        COUNT       equ 08h
        LABEL       decfsz   COUNT,1
                      goto LABEL
                      Carry on here.
                      :
                      :
                      :
```

What we have done is first set up our constant COUNT to 255. The next line puts a label, called LABEL next to our decfsz instruction. The decfsz COUNT,1 decreases the value of COUNT by 1, and stores the result back into COUNT. It also checks to see if COUNT has a value of zero. If it doesn't, it then causes the program to move to the next line. Here we have a 'goto' statement which sends us back to our decfsz instruction. If the value of COUNT does equal zero, then the decfsz instruction causes our program to jump two places forward, and goes to where I have said 'Carry on here'. So, as you can see, we have caused the program to stay in one place for a predetermined time before carrying on. This is called a delay loop. If we need a larger delay, we can follow one loop by another.

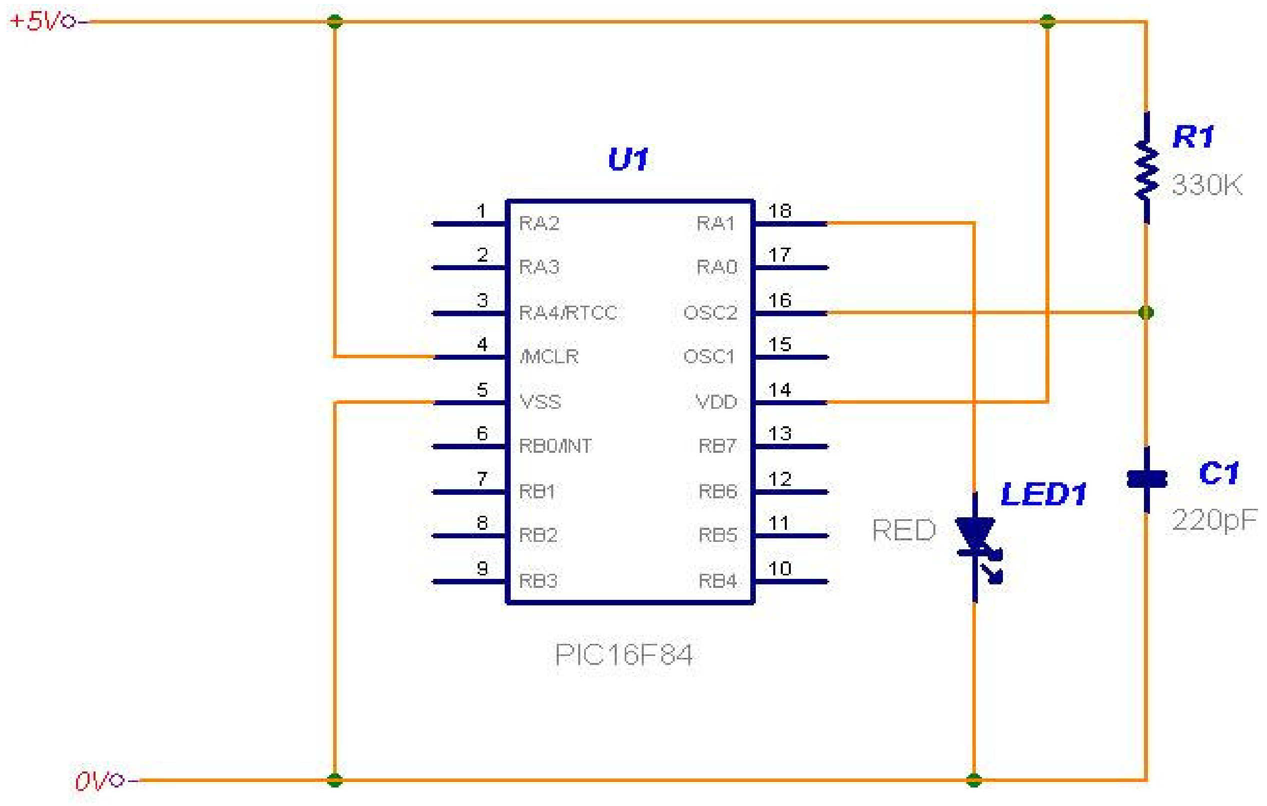
The more loops, the longer the delay. We are going to need at least two, if we want to see the LED flash..

Let us put these delay loops into our program, and finish off by making it a real program by adding comments:

```
;*****Set up the Constants****

STATUS      equu    03h         ;Address of the STATUS register
TRISA       equ     85h         ;Address of the tristate register for port A
PORTA       equ     05h         ;Address of Port A
COUNT1      equ     08h         ;First counter for our delay loops
COUNT2      equ     09h         ;Second counter for our delay loops

;****Set up the port****

            bsf         STATUS,5    ;Switch to Bank 1
            movlw       00h         ;Set the Port A pins
            movwf       TRISA       ;to output.
            bcf         STATUS,5    ;Switch back to Bank 0

;****Turn the LED on****

Start       movlw       02h         ;Turn the LED on by first putting
            movwf       PORTA       ;it into the w register and then
                                    ;on the port

;****Start of the delay loop 1****

Loop1       decfsz      COUNT1,1    ;Subtract 1 from 255
            goto        Loop1       ;If COUNT is zero, carry on.
            decfsz      COUNT2,1    ;Subtract 1 from 255
            goto        Loop1       ;Go back to the start of our loop.
                                    ;This delay counts down from
                                    ;255 to zero, 255 times

;****Delay finished, now turn the LED off****

            movlw       00h         ;Turn the LED off by first putting
            movwf       PORTA       ;it into the w register and then on
                                    ;the port

;****Add another delay****
```

```
Loop2       decfsz        COUNT1,1         ;This second loop keeps the
             goto          Loop2             ;LED turned off long enough for
             decfsz        COUNT2,1         ;us to see it turned off
             goto          Loop2             ;

;****Now go back to the start of the program

             goto          Start              ;go back to Start and turn LED
                                               ;on again

;****End of the program****

end                                     ;Needed by some compilers,
                                        ;and also just in case we miss
                                        ;the goto instruction.
```

You can compile this program and then program the PIC. Of course, you will want to try the circuit out to see if it really does work. Here is a circuit diagram for you to build once you have programmed your PIC.



Congratulations, you have just written your first PIC program, and built a circuit to flash an LED on and off. So far, if you have followed these tutorials, you have learnt a total of 7 instruction out of 35, and yet already you are controlling the I/O ports!

Why not try and alter the delay loops to make the LED flash faster – what is the minimum value of COUNT to actually see the LED flash? Or, why not add a third or even more delay loops after the first one to slow the LED down. You will need a different constant for each delay loop. You could then even adjust your delay loops to make the LED flash at a given rate, for example once a second.

In the next tutorial we will see how we can use a thing called a subroutine to help keep the program small and simple.

Tutorial 5

Subroutines

A subroutine is a section of code, or program, than can be called as and when you need it. Subroutines are used if you are performing the same function more than once, for example creating a delay. The advantages of using a subroutine are that it will be easier to alter the value once inside a subroutine rather than, say, ten times throughout your program, and also it helps to reduce the amount of memory your program occupies inside the PIC.

Let us look at a subroutine:

```
ROUTINE                     COUNT       equ 255
LABEL                      decfsz       COUNT,1
                             Goto         LABEL
                             RETURN
```

First, we have to give our subroutine a name, and in this case I have chosen ROUTINE. We then type the code that we want to perform as normal. In this case, I have chosen the delay in our flashing led program. Finally, we end the subroutine by typing the RETURN instruction.

To start the subroutine from anywhere in our program, we simply type the instruction CALL followed by the subroutine name.

Let us look at this in slightly more detail. When we reach the part of our program that says CALL xxx, where xxx is the name of our subroutine, the program jumps to wherever the subroutine xxx resides. The instructions inside the subroutine are carried out. When the instruction RETURN is reached, the program jumps back to our main program to the instruction immediately following our CALL xxx instruction.

You can call the same subroutine as many times as you want, which is why using subroutines reduces the overall length of our program. However, there are two things you should be aware of. First, as in our main program, any constants must be declared before they are used. These can be either declared within the subroutine itself, or right at the start of the main program. I would recommend that you declare everything at the start of your main program, as then you know that everything is in the same place. Secondly, you must ensure that the main program skips over the subroutine. What I mean by this is if you put the subroutine right at the end of your main program, unless you use a ‘Goto’ statement to jump away from where the subroutine is, the program will carry on and execute the subroutine whether you want it to or not. The PIC does not differentiate between a subroutine and the main program.

Let us look at our flashing led program, but this time we will use a subroutine for the delay loop. Hopefully, you will see how much simpler the program looks, and also you will see how the subroutine works for real.

```
;*****Set up the Constants****

STATUS      equ     03h                 ;Address of the STATUS register
TRISA       equ     85h                 ;Address of the tristate register for port A
PORTA       equ     05h                 ;Address of Port A
COUNT1      equ     08h                 ;First counter for our delay loops
COUNT2      equ     09h                 ;Second counter for our delay loops

;****Set up the port****

bsf             STATUS,5    ;Switch to Bank 1
movlw           00h         ;Set the Port A pins
movwf           TRISA       ;to output.
bcf             STATUS,5    ;Switch back to Bank 0

;****Turn the LED on****

Start       movlw       02h         ;Turn the LED on by first putting it
            movwf       PORTA       ;into the w register and then on the port
```

```
;****Add a delay

        call            Delay

;****Delay finished, now turn the LED off****

movlw       00h         ;Turn the LED off by first putting it
movwf       PORTA       ;into the w register and then on the port

;****Add another delay****

call            Delay

;****Now go back to the start of the program

                goto            Start           ;go back to Start and turn LED on again

;****Here is our Subroutine

Delay

Loop1       decfsz      COUNT1,1    ;This second loop keeps the LED
             goto        Loop1       ;turned off long enough for us to
             decfsz      COUNT2,1    ;see it turned off
             goto        Loop1       ;
return

;****End of the program****

end                                         ;Needed by some compilers, and also
                            ;just in case we miss the goto instruction.
```

Hopefully, you can see that by using a subroutine for our delay loop, we have reduced the size of the program. Each time we want a delay, either when the LED is on or off, we simply call the delay subroutine. At the end of the subroutine, the program goes back to the line following our ‘Call’ instruction. In the example above, we turn the LED on. We then call the subroutine. The program then returns so that we can turn the LED off. We call the subroutine again, and when the subroutine has finished, the program returns and the next instruction it sees is ‘goto Start’.

For those of you who are interested, our original program was 120 bytes long. By using the subroutine, we have reduced our program size down to 103 bytes. This may not seem to be that great, but seeing that we only have 1024 bytes in total inside the PIC, every little bit helps.

In the next tutorial, we will look at reading from the ports.

Tutorial 6

Reading from the I/O ports.

Up to now, we have been writing to Port A so that we can turn an LED on and off. Now, we are going to look at how we can read the I/O pins on the ports. This is so that we can connect an external circuit, and act on any outputs it gives.

If you recall from our previous tutorials, in order to set up the I/O ports, we had to switch from Bank 0 to Bank 1. Let us do that first:

```
STATUS        equ      03h                     ;Address of the STATUS register
TRISA         equ      85h                     ;Address of the tristate register for port A
PORTA         equ      05h                     ;Address of Port A
bsf           STATUS,5                         ;Switch to Bank 1
```

Now, to set up the port to be an output, we sent a 0 to the TrisA register. To set a pin on a port to be an input, we send a 1 to the TisA register.

```
movlw         01h           ;Set the Port A pins
movwf         TRISA         ;to input.
bcf           STATUS,5      ;Switch back to Bank 0
```

Now we have set bit 0 of Port A to input. What we need to do now is to check if the pin is high or low. For this, we can use one of two instructions: BTFSC and BTFSS.

The BTFSC instruction means ‘Do a bit test on the register and bit we specify. If it is a 0, then we skip the next instruction’. BTFSS means ‘Do a bit test in the register and bit we specify. If it is set to a 1, then we skip the next instruction.’

Which one we use, depends on how we want our program to react when we read the input. For example, if we are simply waiting for the input to be a 1, then we could use the BTFSS instruction like this:

```
Code here
:
BTFSS         PortA,0
Goto start
Carry on here
:
:
```

The program will only move onto 'Carry on here' only if bit 0 on PortA is set to a 1.

Let us now write a program which will flash an LED at one speed, but if a switch is closed it will flash the LED twice as slow. You can probably work this program out for yourself, but I have included the listing anyway. You could try and write the whole program, just to see if you have grasped the concepts. We are using the same circuit as before, with the addition of a switch connected RA0 of the PIC and the positive rail of our supply.

```
;*****Set up the Constants****

STATUS      equ     03h                 ;Address of the STATUS register
TRISA       equ     85h                 ;Address of the tristate register for port A
PORTA       equ     05h                 ;Address of Port A
COUNT1      equ     08h                 ;First counter for our delay loops
COUNT2      equ     09h                 ;Second counter for our delay loops

;****Set up the port****

        bsf         STATUS,5    ;Switch to Bank 1
        movlw       01h         ;Set the Port A pins:
        movwf       TRISA       ;bit 1to output, bit 0 to input.
        bcf         STATUS,5    ;Switch back to Bank 0

;****Turn the LED on****

  Start       movlw       02h           ;Turn the LED on by first putting it
  movwf       PORTA                     ;into the w register and then on the port


;****Check if the switch is closed


BTFSC       PORTA,0         ;Get the value from PORT A
                              ;BIT 0.  If it is a zero
call        Delay            ;a zero, carry on as normal.
                              ;If is is a 1, then add an
                              ;extra delay routine

;****Add a delay

call    Delay

;****Delay finished, now turn the LED off****

          movlw       00h         ;Turn the LED off by first putting it
          movwf       PORTA       ;into the w register and then on the port
```

```
;****Check if the switch is still closed

BTFSC                    PORTA,0         ;Get the value from PORT A
                                            ;BIT 0.  If it is a zero,
       call                    Delay              ;carry on as normal.
                                            ;If is a 1, then add an
                                            ;extra delay routine

;****Add another delay****

call       Delay

;****Now go back to the start of the program

                goto               Start              ;go back to Start and turn LED on again

;****Here is our Subroutine

Delay

Loop1            decfsz             COUNT1,1     ;This second loop keeps the LED
                  goto               Loop1           ;turned off long enough for us to
                 decfsz              COUNT2,1     ;see it turned off
                 goto                 Loop1             ;
return

 ;****End of the program****

 end                                                   ;Needed by some compilers, and also
                                                        ;just in case we miss the goto instruction.
```

What I have done here is to turn the LED on. I then check to see if the switch is closed. If it is closed, then I make a call to our delay subroutine. This gives us the same delay as before, but we are now calling it twice. The same goes for when the LED is off. If the switch is not closed, then we have our old on and off times.

You can compile and run this program. However a word of warning. The final circuit and code will look un-impressive to someone who is not interested in programming microcontrollers. So, don't be upset if, when you show your family and friends how you can change the speed of a flashing LED with a switch, they show very little interest – I am talking from personal experience, here!

If you have been following these tutorials from the start, then you may be interested to know that you have now learnt 10 of the 35 instructions for the PIC 16F84! And all of these have been learnt just by simply turning an LED on and off.

Tutorial 7

So far, we have made the PIC flash an LED on and off. Then we were able to interact with our PIC by adding a switch, and so altering the flash rate. The only problem is, the program is very long and very wasteful of memory. It was fine when I was introducing the commands for for the first time, but there must be a better way of doing it. Well there is (you knew that was coming, right?).

Let us examine how we were actually turning the LED on and off.

```
movlw        02h
movwf        PORTA
movlw        00h
movlw        PORTA
```

First we loaded our w register with 02h, then moved it to our PortA register to turn the LED on. To turn it off, we loaded w with 00h and then moved it to our PortA register. In between these routines we had to call a subroutine so that we could see the LED flashing. So, we had to move two sets of data twice (once into the w register then to PORTA) and call a subroutine twice (once for on and once for off).

So, how can we do this more efficiently? Simple. We use another instruction called XORF.

The XORF instruction performs an Exclusive OR function on the register that we specify with the data we give it. I think I need to explain what on earth an Exclusive OR is before we go on.

If we have two inputs, and one output, the output will only be a 1 if, and only if, the two inputs are different. If they are the same, then the output will be 0. Here is a truth table, for those who prefer to look at these:

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Let us now look to what happens if we make B the same as our previous output, and just changing the value of A:

A	B	F
0	0	0
0	0	0
1	0	1

1	1	0
1	0	1

If we keep the value of A equal to 1, and we Exclusive OR it with the output, the output will toggle. For those who can't see this from the truth table, here it is using binary:

```
                0       Current Output
EX-OR With 1    1       New Output
EX-OR With 1    0       New Output
```

Hopefully you can see that by exlusive ORing the output with 1, we are now toglling the output from 0 to 1 to 0.

So, to turn our LED on and off, we just need two lines:

```
MOVLW     02h
XORWF     PORTA,1
```

What we are doing is loading our w register with 02h. We are then Exclusive ORing this number with whatever is on our PortA. If bit 1 is a 1, it will change to a 0. If bit 1 is a 0, it will change to a 1.

Let's run through this code a couple of times, to show how it is working in binary:

```
                PORTA
                00010
xorwf           00000
xorwf           00010
xorwf           00000
xorwf           00010
```

We don't even need to load the same value into our w register each time, so we can do this once at the beginning, and just jump back to our toggle command. Also, we don't need to set up a value on our PortA register. Why? Well, because if on power up it is a 1, we will toggle it. I, on the other hand it is a 0 on power up, we will still toggle it.

So, let us now see our new code. The first one is our original flashing LED, and the second is where we added a switch:

Flashing LED

```
;*****Set up the Constants****

STATUS      equ     03h             ;Address of the STATUS register
TRISA       equ     85h             ;Address of the tristate register for port A
PORTA       equ     05h             ;Address of Port A
```

```
COUNT1        equ      08h                              ;First counter for our delay loops
COUNT2        equ      09h                              ;Second counter for our delay loops

;****Set up the port****

        bsf              STATUS,5      ;Switch to Bank 1
        movlw            00h            ;Set the Port A pins
        movwf            TRISA          ;to output.
        bcf              STATUS,5      ;Switch back to Bank 0
        movlw            02h            ;Set up our w register with 02h

;****Turn the LED on and off****

Start           xorwf           PORTA,1       ;Toggle the LED

;****Add a delay

call      Delay

;****Now go back to the start of the program

            goto             Start              ;go back to Start and turn LED on again

;****Here is our Subroutine

Delay

Loop1          decfsz            COUNT1,1      ;This second loop keeps the LED
               goto              Loop1           ;turned off long enough for us to
               decfsz            COUNT2,1      ;see it turned off
               goto              Loop1           ;

return

;****End of the program****

end                                                          ;Needed by some compilers, and also
                                                             ;just in case we miss the goto instruction.
```

Flashing LED With Switch:

```
;*****Set up the Constants****

STATUS         equ      03h                              ;Address of the STATUS register
TRISA          equ      85h                              ;Address of the tristate register for port A
PORTA          equ      05h                              ;Address of Port A
```

```
COUNT1        equ      08h                            ;First counter for our delay loops
COUNT2        equ      09h                            ;Second counter for our delay loops

;****Set up the port****

        bsf             STATUS,5       ;Switch to Bank 1
        movlw           01h            ;Set the Port A pins:
        movwf           TRISA          ;bit 1to output, bit 0 to input.
        bcf             STATUS,5       ;Switch back to Bank 0

        movlw           02h            ; Set up our w register with 02h

;****Turn the LED on and off****

Start           xorwf            PORTA,1       ;Toggle the LED

;****Check if the switch is closed

        BTFSC                   PORTA,0       ;  Get the value from PORT A
                                                 ;BIT 0.  If it is a zero,
        call                    Delay           ;carry on as normal.
                                                 ;If is a 1, then add an
                                                 ;extra delay routine

;****Add a delay

call      Delay

;****Check if the switch is still closed

        BTFSC                   PORTA,0       ;Get the value from PORT A
                                                 ;BIT 0.  If it is a zero,
        call                    Delay           ;carry on as normal.
                                                 ;If is a 1, then add an
                                                 ;extra delay routine

;****Add another delay****

call      Delay

;****Now go back to the start of the program

                goto             Start             ;go back to Start and turn LED on again

;****Here is our Subroutine

Delay
```

```
Loop1           decfsz          COUNT1,1     ;This second loop keeps the LED
                 goto             Loop1          ;turned off long enough for us to
                 decfsz          COUNT2,1     ;see it turned off
                 goto             Loop1          ;
return

;****End of the program****

end                                             ;Needed by some compilers, and also
                                                ;just in case we miss the goto instruction.
```

I hope you can see that by just using one simple instruction, we have reduced the size of our program. In fact, just to show how much we have reduced our programs by, I have shown the two programs, what changes were made, and their sizes in the table below:

Program	Change	Size (Bytes)
Flashing LED	Original	120
Flashing LED	Subroutine Added	103
Flashing LED	XOR Function Used	91
LED With Switch	Original	132
LED With Switch	XOR Function Used	124.

So, not only have we learnt some new instructions, we have also reduced the size of our coding!

Tutorial 8

Here, we are going to examine how to manipulate individual bits, perform some simple arithmetic, and data tables.

Logical Operators

In the last tutorial I introduced the Exclusive OR function. The ExOR function is known as a logical operator. In this tutorial I am going to explain the other logical operators that the PIC supports. There won't be any example programs, but I will explain how to use the operators by using small sections of code.

AND

The AND function simply compares two bits and produces a 1 if they are the same, and a 0 if they are different. For example, if we said 1 AND 1, the result is 1, whereas if we said 1 AND 0 the result will be 0. Of course, we can compare words as well, and all the AND function does is compare the two words bit by bit. The example below shows two 8-bit words being ANDed along with the result:

```
          11001011
AND       10110011
Equals    10000011
```

As you can see, the result will only have a 1 when two 1s coincide with each other in the two words. We can use the AND function to check the ports, for example. If we are monitoring some I/O pins which are connected to a circuit, and we need to monitor a certain condition where only some of the pins are high, then we can simply read the port, and then AND the result with the condition we are checking for, just like the example above.

The PIC gives us two flavors for AND. They are ANDLW and ANDWF. ANDLW allows us to perform an AND function with the contents of the W register, and a number that we specify. The syntax is:

ANDLW <number> where <number> is what we will AND the contents of W with. The result of the AND function will be stored back into the W register.

ANDWF allows us to perform an AND function on the W register and another register, such as a PORT. The syntax is:

ANDWF <register>,d where <register> is the register we are interested in, e.g. PORTA, and d tells the PIC where to place the result. If d=0, the result is placed in the W register, and of d=1 the result is stored in the register we specified.

The two sections of code below show an example of each AND function. The first is checking the status of the PORTA, where we need to see if the inputs are 1100. We will put the result back into the W register:

```
movlw       1100
ANDWF       05h,0
```

The second example will now check the contents of the W register:

```
ANDLW       1100
```

OR

We have already come across one OR function, namely the XOR. This produced a 1 if two bits are different, but not the same. There is a second OR function called IOR, which is the inclusive OR. This function will produce a 1 if either bit is a 1, but also if both bits are 1. Below is a simple truth table to demonstrate this:

A	B	O/P
0	0	0
0	1	1
1	0	1
1	1	1

Arithmetic Operators

ADD

This function does exactly what it says. It adds two numbers! If the result of adding the two numbers exceeds 8 bits, then a CARRY flag will be set. The CARRY flag is located at address 03h bit 0. If this bit is set, then the two numbers exceeded 8 bits. If it is a 0, then the result lies within 8 bits.

Again, the PIC gives us two flavors of ADD, namely ADDLW and ADDWF. As you may have guessed, this is very similar to the above function. ADDLW adds the contents of the W register to a number that we specify. The syntax is:

```
ADDLW  <number>
```

ADDWF will add the contents of the W register and any other register that we specify. The syntax is:

ADDWF <register>,d where <register is the register we specify and d tells the PIC where to place the result. If d=0, the result is placed in the W register, and is d=1 it placed in the register that we specified.

SUB

Now, I bet you can't guess what this function does! Yes, you guessed it, this function subtracts one bit from another. Once again the PIC gives us two flavors: SUBLW and SUBWF. The syntax is exactly the same as for the ADD function, except of course you type SUB instead of ADD!

Increment

If we wanted to add 1 to a number in the PIC, we could just simply use the ADD function, and use the number 1. ~The problem with this is that we have to first put the number into the W register, then use ADDLW 1 command to increment it. If we wanted to add 1 to a register, it is even worse. We first have to put the number 1 into the W register, then use ADDWF <register>,1. So, for example, to add 1 to location 0C, say, we would have to have the following section of code:

```
movlw      01
addwf      0c,1
```

There is a better way of doing this. We can use the command INCF. The syntax is:

INCF <register>,d where <register> is the register, or location, that we are interested in, and d tells the PIC where to place the result. If d=0, the result is in the W register, and if d=1, the result is placed in the register we specified. By using this single instruction we can literally half the coding. If we wanted the result put back into the W register, then using the example above, we would have had to add another command to move the contents of 0C back into the W register, and then put the 0C register back to whatever it was.

There is another increment command. It is INCFSZ. This command will increment the register that we specify, but if we the register equals 0 after the increment (which will happen when we add 1 to 127) then the PIC will skip the next instruction. The section of code below demonstrates this:

```
Loop      incfsz          0C
          Goto Loop
           :
           :
           Rest of program.
```

In the above section of code, 0C will be incremented by 1. We then have an instruction that tells the PIC to go back to our label called Loop, and increment 0C by 1 again. This carries on until 0C equals 127. This time, when we increment 0C by 1, 0C will now equal 0. Our INCFSZ instruction will then tell the PIC to skip the next instruction, which in this case is the goto statement, and so the PIC will continue with the rest of the program.

Decrement

I have already covered the decrement function in previous tutorials, so I won't repeat myself here.

Compliment

The last instruction in this group will invert all of the bits in the register that we specify. The syntax is:

COMF <register>,d where <register is the register that we want to invert, and d will tell the PIC where to store the result. If d=0, the result is stored in the W register. Id d=1, the result is stored back into the register we specified. The following example shows this instruction in action:

```
0C = 11001100

COMF 0C,1

0C = 00110011
```

This could be use, for example, to quickly turn the pins of a port from output to input and vice-versa.

Tutorial 9

BIT Operations

Bit operations allow us to manipulate a single bit within a word. They allow us to move, set and clear single bits in registers or numbers that we specify. At the end of this tutorial I will show you a program that will produce a set of running lights that go one way, then the other way. *We saw this done previously when we looked at the exclusive OR function, where we Exclusively ORed the ports with a word.*

We have already seen a couple of bit operations when we set up the ports on the PIC, and I will repeat their use here.

<u>BCF</u>

This instruction will clear a bit that we specify in a register that we specify. The syntax is:

```
BCF     <register>,<bit>
```

We used this previously to change from page 1 to page 0 by clearing a bit in the STATUS register. We can also use it to set a bit to 0 in any other register/location. For example, if we wanted to set the third bit in 11001101 stored in location 0C to 0, we would enter:

```
BCF     0C,03
```

<u>BSF</u>

This instruction will set any bit we specify to 1 in any register that we specify. We used this previously to go from Page 0 to Page 1. The syntax is:

```
BSF     <register>,<bit>, and is used in
exactly the same way as BCF above.
```

<u>BTFSC</u>

So far we have set or cleared a bit in a register. But what if we want to just simply test if a bit is a 1 or a 0 in a register? Well, we can use BTFSC. It says Bit Test Register F, and Skip If It Is Clear. This instruction will test the bit we specify in the register. If the bit is a 0, the instruction will tell the PIC to skip the next instruction. We would use this instruction if we wanted to test a flag, such as the carry flag. This saves us having to read the STATUS register and looking at the individual bits to see which flags are set. For example, if we wanted to test if the Carry flag had been set to 1 after we have added two numbers, then we would enter the following:

```
BTFSC           03h,0
carry on here if set to 1
or here if set to 0
```

If the status of the bit is a 1, then the instruction immediately following BTFSC will be carried out. If it is set to a 0, then the next instruction is skipped. The following section of code shows where it might be used:

```
Loop   :
       :
       :
       BTFSC 03,0
       Goto Loop
```

In the above code, the PIC will only come out of the loop if bit 0 of the STATUS register (or the Carry flag) is set to 0. Otherwise, the goto command will be carried out.

BTFSS

This instruction says Bit Test Register F, And Skip If Set. This is similar to the BTFSC instruction, except that the PIC will skip the next instruction if the bit we are testing is set to 1, rather than 0.

CLRF

This instruction will set the entire contents of a register to 0. The syntax is:

```
CLRF  <register>
```

We used this previously to set the output of the Ports to 0, by using CLRF 85h. We also used it to set the Ports to have all pins to output by using CLRF 05h.

CLRW

This is similar to the CLRF instruction, except is only clears the W register. The syntax is quite simply:

```
CLRW
```

RLF And RRF

These commands will move a bit in a register one place to the left (RLF) or the right (RRF) in a register. For example, if we had 00000001 and we used RLF, then we would have 00000010. Now, what happens if we have 10000000 and carried out the RLF instruction? Well, the 1 will be placed in the carry flag. If we carried out the RLF instruction again, the 1 will reappear back at the beginning. The same happens, but in reverse, for the RRF instruction. The example below demonstrates this for the RLF instruction, where I have shown the 8 bits of a register, and the carry flag :

```
       C 76543210
       0 00000001
RLF    0 00000010
RLF    0 00000100
RLF    0 00001000
RLF    0 00010000
RLF    0 00100000
RLF    0 01000000
RLF    0 10000000
RLF    1 00000000
RLF    0 00000001
```

Example Program

I am now going to give you an example code which you can compile and run. It will produce a running light starting at PortA bit 0, going to PortB bit 8 and then back again. Connect LEDs to all of the Port pins. You will see some of the bit operations mentioned in this tutorial.

```
TIME  EQU 9FH           ; Variable for the delay loop.
PORTB EQU 06H           ; Port B address.
TRISB EQU 86H           ; Port B Tristate address.
PORTA EQU 05H           ; Port A address.
TRISA EQU 85H           ; Port A Tristate address.
STATUS EQU 03H          ; Page select register.
COUNT1 EQU 0CH          ; Loop register.
COUNT2 EQU 0DH          ; Loop register.


BSF STATUS,5            ; Go to page 1
MOVLW 00H               ; and set up
MOVWF TRISB             ; both Ports A and B
MOVLW 00H               ; to output,
MOVWF TRISA                ; then return to
BCF STATUS,5               ; page 0.

MOVLW 00H                   ; Clear Port A.
MOVWF PORTA                ;

 ; Start of main program

RUN
MOVLW 01H                  ; Set the first bit
MOVWF PORTB               ; on Port B.
CALL DELAY                 ; Wait a while
CALL DELAY                 ;


; Move the bit on Port B left, then pause.


    RLF PORTB,1

    CALL DELAY
    CALL DELAY

    RLF PORTB,1

    CALL DELAY
    CALL DELAY

    RLF PORTB,1

    CALL DELAY
    CALL DELAY

    RLF PORTB,1

    CALL DELAY
    CALL DELAY

    RLF PORTB,1

    CALL DELAY
    CALL DELAY

    RLF PORTB,1

    CALL DELAY
    CALL DELAY

    RLF PORTB,1
    CALL DELAY
    CALL DELAY

    RLF PORTB,1             ; This moves the bit into the carry flag


; Now move onto Port A, and move the bit left.


RLF PORTA,1                 ; This moves the bit from the zero flag into PortA

CALL DELAY
CALL DELAY

RLF PORTA,1

CALL DELAY
CALL DELAY

RLF PORTA,1

CALL DELAY
CALL DELAY

RLF PORTA,1

CALL DELAY
CALL DELAY


; Move the bit back on Port A


RRF PORTA,1

CALL DELAY
CALL DELAY

RRF PORTA,1

CALL DELAY
CALL DELAY

RRF PORTA,1

CALL DELAY
CALL DELAY

RRF PORTA,1                 ; This moves the bit into the zero flag
```

```
; Now move the bit back on Port B


    RRF PORTB,1

    CALL DELAY
    CALL DELAY

    RRF PORTB,1

    CALL DELAY
    CALL DELAY

    RRF PORTB,1

    CALL DELAY
    CALL DELAY

    RRF PORTB,1

    CALL DELAY
    CALL DELAY

    RRF PORTB,1

    CALL DELAY
    CALL DELAY

    RRF PORTB,1

    CALL DELAY
    CALL DELAY

    RRF PORTB,1

    CALL DELAY
    CALL DELAY              ; Now we are back where we started,
                              ;
GOTO RUN                     ; let's go again.


; Subroutine to give a delay between bit movements.


DELAY

MOVLW TIME                   ; Get the delay time,
MOVWF COUNT1                ; and put it into a variable.
```

```
    LOOP1                        ;

         DECFSZ COUNT1  ; Decrement 1 from the delay time until it          GOTO
LOOP1         ; reaches zero.

MOVWF COUNT1           ; Get the delay time again,

    LOOP2                        ; and repeat the count down.
         DECFSZ COUNT1 ;
         GOTO LOOP2        ;

RETURN                           ; End of subroutine.

END                              ;
```

Tutorial 10

Data Tables

There is a nice feature in the instruction set that allows you to use a data table. A data table is simply a list of data values, where each one is read depending on some criteria. For example, you might have a circuit that uses a PIC where it counts the number of times an input pin goes high in 1 second. You can then display the number on a 7 segment display. Once the timing has started, the PIC counts the number of times the pin goes high. After 1 second it goes to the table and looks up the information it needs to display the number on the display that corresponds to the number of times the pin went high. This is useful, because we don't know what the number will be until the PIC has completed its count. By using a table, we can let the PIC decide which number to display.

Now, before I carry on to explain how the data table works, I have to explain how the PIC keeps track of whereabouts in the program it is when the program is running. It helps if you have done some programming in BASIC. If not, don't worry, you should still be able to see the concept.

Imagine we have a BASIC program like the one shown below:

```
10          LET K=0
11          K=K+1
12          IF K>10 THEN GOTO 20 ELSE GOTO 11
20                PRINT K
21                END
```

The program starts at line 10. Once K is set to 0, it then proceeds to line 11. After we have added 1 to K we then move on to line 12. Here we are asking if K is greater than 10. If it is, then we go to line 20, if not we go back to line 11. Line 20 prints the value of K, and line 21 ends the program. BASIC uses line numbers to help the programmer keep track of where things are, as labels are not allowed.

The PIC uses labels to jump between locations – or does it? We use the labels so that we know where things are, and also so that we can tell the PIC in an easy way where to go. What actually happens is the PIC uses an internal line counter called a Program Counter. The Program Counter (abbreviated to PC) keeps track of the memory location of where the current instruction is. When we tell the PIC to go to a particular label, it know the memory location and hence increase the PC until it reads that memory location. This is exactly the same way as we read the BASIC program above. Below is a section of code, with the memory locations, or the contents of the PC, next to each instruction:

```
PC              Instruction

0000             movlw  03
0001             movwf  0C
0002    Loop    decfsc  0C
0003             goto Loop
0004    end
```

In the example above, I have set the PC to 0000. At this location we have the instruction movlw 03. When the PIC has executed this instruction, it increments the PC so that the next instruction is read. Here the PIC sees movwf 0C. The PC is incremented again. Now the PIC reads decfsc 0C. If the contents of 0C are not 0, then the PC is incremented by 1, and the next instruction, goto Loop, tells the PC to go back to location 0003, which is where we have said Loop. If the contents of 0C is 0, then the PC is told to increment by 2, in other words skip the next instruction. This puts the PC at location 0004, where the program ends. The locations are set by the assembler, and we don't normally need to worry what the PC is doing. Until, that is we need to control it like we are about to do when using data tables.

The best way to explain how a data table works, is to start off with an example.

```
PC   equ  02


movlw  03
call      table
```

```
          :
table     addwf   PC
retlw     01
retlw     02
retlw     03
retlw     04
retlw     05
retlw     06
retlw     07

return
```

The first instruction is assigning the label PC with the address of the Program Counter (02h). We are then placing the value of 03h into the w register. We then make a call to table. The first line in the subroutine table adds the contents of the W register (03h) to the program counter. This causes the program counter to increase by 3, or to put it another way, causes the program counter to move down 3 lines. When the counter reaches 3 lines down it the PIC sees the instruction retlw. This command passes the value following it into the W register, and then returns from the subroutine. RETLW actually means Return, Literal to W. Notice I put a comma after the word Return. As we are in a subroutine, we need a Return instruction to come out of it. Hence the RET in the instruction. After the RETLW instruction is a number, and this is what is placed in the W register. In this case it is the number 3.

We can assign any number to the W register, as long as when this number is added to the Program Counter in the table subroutine, we will find a retlw instruction. In the above example this means we can have any number from 1 to 7. If we go past the subroutine, we could end up executing another part of the program. Because of this, it is always a good idea to put the data table right at the end of the PIC program, so if we do overshoot then we will reach the end of the program anyway.

Tutorial 11

Interrupts - An Introduction

The subject of interrupts is probably going to be the longest and most difficult to go through. There is no easy way of explaining interrupts, but hopefully by the end of this section you will be able to implement interrupts into your own programs. I have split the section into two parts. This is to help break the subject up, and to give you, the reader, a break.

So what is an interrupt? Well, as the name suggests, an interrupt is a process or a signal that stops a microprocessor/microcontroller from what it is doing so that something else can happen. Let me give you an every day example. Suppose you are sitting at home, chatting to someone. Suddenly the telephone rings. You stop chatting, and pick up the telephone to speak to the caller. When you have finished your telephone conversation, you go back to chatting to the person before the telephone rang. You can think of the main routine as you chatting to someone, the telephone ringing causes you to interrupt your chatting, and the interrupt routine is the process of talking on the telephone. When the telephone conversation has ended, you then go back to your main routine of chatting. This example is exactly how an interrupt causes a processor to act. The main program is running, performing some function in a circuit, but when an interrupt occurs the main program halts while another routine is carried out. When this routine finishes, the processor goes back to the main routine again.

The PIC has 4 sources of interrupt. They can be split into two groups. Two are sources of interrupts that can be applied externally to the PIC, while the other two are internal processes. I am going to explain the two external ones here. The other two will be explained in other tutorials when we come to look at timers and storing data.

If you look at the pin-out of the PIC, you will see that pin 6 shows it is RB0/INT. Now, RB0 is obviously Port B bit 0. The INT symbolizes that it can also be configures as an external interrupt pin. Also, Port B bits 4 to 7 (pins 10 to 13) can also be used for interrupts. Before we can use the INT or other Port B pins, we need to do two things. First we need to tell the PIC that we are going to use interrupts. Secondly, we need to specify which port B pin we will be using as an interrupt and not as an I/O pin.

Inside the PIC there is a register called INTCON, and is at address 0Bh. Within this register there are 8 bits that can be enabled or disabled. Bit 7 of INTCON is called GIE. This is the Global Interrngupt Enable. Setting this to 1 tells the PIC that we are going to use an interrupt. Bit 4 of INTCON is called INTE, which means INTerrupt Enable. Setting this bit to 1 tells the PIC that RB0 will be an interrupt pin. Setting bit 3, called RBIE, tells the PIc that we will be using Port B bits 4 to 7. Now the PIC knows when this pin goes high or low, it will need to stop what it's doing and get on with an interrupt routine. Now, we need to tell the PIC whether the interrupt is going to be on the rising edge (0V to +5V) or the falling edge (+5V to 0V) transition of the signal. In other words, do we want the PIC to interrupt when the signal goes from low to high, or from high to low. By default, this is set up to be on the rising edge. The edge 'triggering' is set up in another register called the OPTION register, at address 81h. The bit we are interested in is bit 6, which is called INTEDG. Setting this to 1 will cause the PIC to interrupt on the rising edge (default state) and setting it to 0 will cause the PIC to interrupt on the falling edge. If you want the PIC to trigger on the rising edge, then you don’t need to do anything to this bit. Now, unfortunately, the Option register is in Bank 1, which means that we have to change from bank 0 to bank 1, set the bit in the Option register, then come back to bank 0. The trick here is to do all of the Bank 1 registers in one hit, such as setting up the port pins, then coming back to Bank 0 when you are finished.

Ok, so now we have told the PIC which pin is going to be the interrupt, and on which edge to trigger, what happens in the program and the PIC when the interrupt occurs? Two things happen. First, a ‘flag’ is set. This tells the internal processor of the PIC that an interrupt has occurred. Secondly, the program counter (which I mentioned in the last tutorial) points to a particular address within the PIC. Let’s quickly look at each of these separately.

Interrupt Flag

In our INTCON register, bit 1 is the interrupt flag, called INTF. Now, when any interrupt occurs, this flag will be set to 1. While there isn’t an interrupt, the flag is set to 0. And that is all it does. Now you are probably thinking ‘what is the point?’ Well, while this flag is set to 1, the PIC cannot, and will not, respond to any other interrupt. So, let’s say that we cause an interrupt. The flag will be set to 1, and the PIC will go to our routine for processing the interrupt. If this flag wasn’t set to 1, and the PIC was allowed to keep responding to the interrupt, then continually pulsing the pin will keep the PIC going back to the start of our interrupt routine, and never finishing it. Going back to my example of the telephone, it’s like picking up the telephone, and just as soon as you start to speak it starts ringing again because someone else want to talk to you. It is far better to finish one conversation, then pick up the phone again to talk to the second person.

There is a slight drawback to this flag. Although the PIC automatically sets this flag to 1, it doesn’t set it back to 0! That task has to be done by the programmer – i.e. you. This is easily done, as I’m sure you can guess, and has to be done after the PIC has executed the interrupt routine.

Memory Location

When you first power up the PIC, or if there is a reset, the Program Counter points to address 0000h, which is right at the start of the program memory. However, when there is an interrupt, the Program Counter will point to address 0004h. So, when we are writing our program that is going to have interrupts, we first of all have to tell the PIC to jump over address 0004h, and keep the interrupt routine which starts at address 0004h separate from the rest of the program. This is very easy to do.

First, we start our program with a command called ORG. This command means Origin, or start. We follow it with an address. Because the PIC will start at address 0000h, we type ORG 0000h. Next we need to skip over address 0004h. We do this by placing a GOTO instruction, followed by a label which points to our main program. We then follow this GOTO command with another ORG, this time with the address 0004h. It is after this command that we enter our interrupt routine. Now, we could either type in our interrupt routine directly following the second ORG command, or we can place a GOTO statement which points to the interrupt routine. It really is a matter of choice on your part. To tell the PIC that it has come to the end of the interrupt routine we need to place the command RTFIE at the end of the routine. This command means return from the interrupt routine. When the PIC see this, the Program Counter points to the last location the PIC was at before the interrupt happened. I have shown below a short segment of code to show the above:

```
        ORG    0000h  ;PIC starts here on power up and reset
        GOTO start    ;Goto our main program

        ORG    0004h  ;The PIC will come here on an interrupt
        :             ;This is our interrupt routine that we
        :             ;want the PIC to do when it receives
        :             ;an interrupt

        RETFIE        ;End of the interrupt routine

start                 ;This is the start of our main program.
```

There are two things you should be aware of when using interrupts. The first is that if you are using the same register in your main program and the interrupt routine, bear in mind that the contents of the register will probably change when the interrupt occurs. For example, let's you are using the w register to send data to Port A in the main program, and you are also using the w register in the interrupt routine to move data from one location to another. If you are not careful, the w register will contain the last value it had when it was in the interrupt routine, and when you come back from the interrupt this data will be sent to Port A instead of the value you had before the interrupt happened. The way round this is to temporarily store the contents of the w register before you use it again in the interrupt routine. The second is that there is a delay between when one interrupt occurs and when the next one can occur. As you know, the PIC has an external clock, which can either be a crystal or it can be a resistor-capacitor combination. Whatever the frequency of this clock, the PIC divides it by 4 and then uses this for it's internal timing. For example if you have a 4MHz crystal connected to your PIC, then the PIC will carry out the instructions at 1MHz. This internal timing is called an Instruction Cycle. Now, the data sheet states (admittedly in very small print) that you must allow 3 to 4 instruction cycles between interrupts. My advice is to allow 4 cycles. The reason for the delay is the PIC needs time to jump to the interrupt address, set the flag, and come back out of the interrupt routine. So, bear this in mind if you are using another circuit to trigger an interrupt for the PIC.

Now, a point to remember is that if you use bits 4 to 7 of Port B as an interrupt. You cannot select individual pins on Port B to serve as an interrupt. So, if you enable these pins, then they are all available. So, for example, you can't just have bits 4 and 5 – bits 6 and 7 will be enabled as well. So what is the point of having four bits to act as an interrupt? Well, you could have a circuit connected to the PIC, and if any one of four lines go high, then this could be a condition that you need the PIC to act on quickly. One example of this would be a house alarm, where four sensors are connected to Port B bits 4 to 7. Any sensor can trigger the PIC to sound an alarm, and the alarm sounding routine is the interrupt routine. This saves examining the ports all the time and allows the PIC to get on with other things.

In the next tutorial, we will write a program to handle an interrupt.

Tutorial 12

Interrupts - Writing The Code

We covered quite a bit of ground in the last tutorial, and so I think it is time that we wrote our first program. The program we are going to write will count the number of times we turn a switch on, and then display the number. The program will count from 0 to 9, displayed on 4 LEDs in binary form, and the input or interrupt will be on RB0.

The first thing we need to do is tell the PIC to jump over the address where the Program Counter points to when an interrupt occurs. You will notice that I am using a different way of expressing hexadecimal numbers. Before I used to use F9h where h denoted hexadecimal. We can write this as 0xF9, and this is the format I am going to use from now on.

```
org             0x00            ;This is where the PC points to on power up and reset
        goto            main            ;Goto our main program

org             0x04            ;This is where our interrupt routine will start
```

```
retfie                                  ;This tells the PIC that the interrupt routine has
                                                    ;finished and the PC will point back to the main
                                                    ;program

         main                                    ;This is the start of our main program
```

Now we need to tell the PIC that we are going to use interrupts, and we are using RB0 pin 6 as an interrupt pin:

```
bsf                 INTCON,7      ;GIE – Global interrupt enable (1=enable)
bsf                 INTCON,4      ;INTE - RB0 interrupt enable (1=enable)
```

I am going to clear the interrupt flag just in case (I never trust anything!)

```
bcf                INTCON,1      ;INTF - Clear flag bit just in case
```

Now we need to set up our two ports. Remember that as we are using RB0 as an interrupt pin, this must be set up as an input:

```
  bsf                 STATUS,5      ;Switch to Bank 1

movw             0x01               ;

movwf            TRISB            ;Set RB0 as input

  movlw            0x10               ;

          movwf            TRISA            ;Set the first 4 pins on PortA as output

          bcf                STATUS,5      ;Come back to Bank 0
```

We are going to use a variable called COUNT to store the number of switch counts. We could just simply increment the value on Port A, but you will see why I am using a variable when we write our interrupt routine.

```
          loop

                    movf            COUNT,0                ;Move the contents of
          COUNT into W

                    movwf           PORTA                   ;Now move it to Port A

                    goto             loop                        ;Keep on doing this
```

```
        end                                     ;End of our program
```

So, our main program is written, and now we need to tell the PIC what to do when an interrupt happens. In this instance, our interrupt is going to be the switch. What we want the PIC to is add one to the variable COUNT each time the switch is closed. However, we only want to display the number of times the switch closes from 0 to 9. Above, I said we could have just simply incremented the value on Port A each time there was an interrupt. But, Port A has 5 bits, and if we just simply incremented the port, we will have a maximum count of 31. There are two reasons why I chose not to go up to 31. First, we are going to use a 7-segment display, which can at the most only go from 0 to 15 (0 to F in hex). Secondly, I also want to show you some of the arithmetic commands that you came across in the last couple of tutorials.

So lets get on with our interrupt routine.

Now the first thing we need to do is temporarily store the contents of our w register, as we are using this to transfer the contents of COUNT to PORTA. If we don't store it, then we could send a completely different number as a result of our arithmetic. So let's do that first:

```
movwf           TEMP            ;Store w register in a temporary location
```

Next we want to add 1 to our variable COUNT:

```
incf            COUNT,1         ;Increment COUNT by 1, and put the result

                                ;back into COUNT
```

Next we want to do a check on COUNT to se if we have gone past the value of 9. The way we can do this is to subtract it from 10.

```
        movlw           0x0A            ;Move the value 10 into w

        subwf           COUNT,0         ;Subtract w from COUNT, and put the

                                        ;result in w
```

From tutorial 8 we saw that if we subtract a large number from a small number a Carry flag will be set. This flag will also be set if the numbers are equal, and we subtract them.

```
btfss           STATUS,0        ;Check the Carry flag. It will be set if
```

```
                                        ;COUNT is equal to, or is greater than w,

                                        ;and will be set as a result of the subwf

                                        ;instruction
```

Now we know if the value of COUNT is 9 or more. What we want to do now is if COUNT is greater than 9, put it back to 0, otherwise go back to the main program so that we can send it to Port A. The BTFSS command as you know will skip the next instruction if the carry flag is set i.e COUNT = 10:

```
goto            carry_on                ;If COUNT is <10, then we can carry on

        goto            clear                   ;If COUNT is >9, then we need to clear it

 carry_on

        bcf             INTCON,0x01           ;We need to clear this flag to enable

                                                ;more interrupts

        movfw           TEMP                    ;Restore w to the value before the
interrupt

        retfie                                  ;Come out of the interrupt routine



clear

        clrf            COUNT                 ;Set COUNT back to 0

        bcf             INTCON,1              ;We need to clear this flag to enable

        ;more interrupts

 retfie                                       ;Come out of the interrupt routine
```

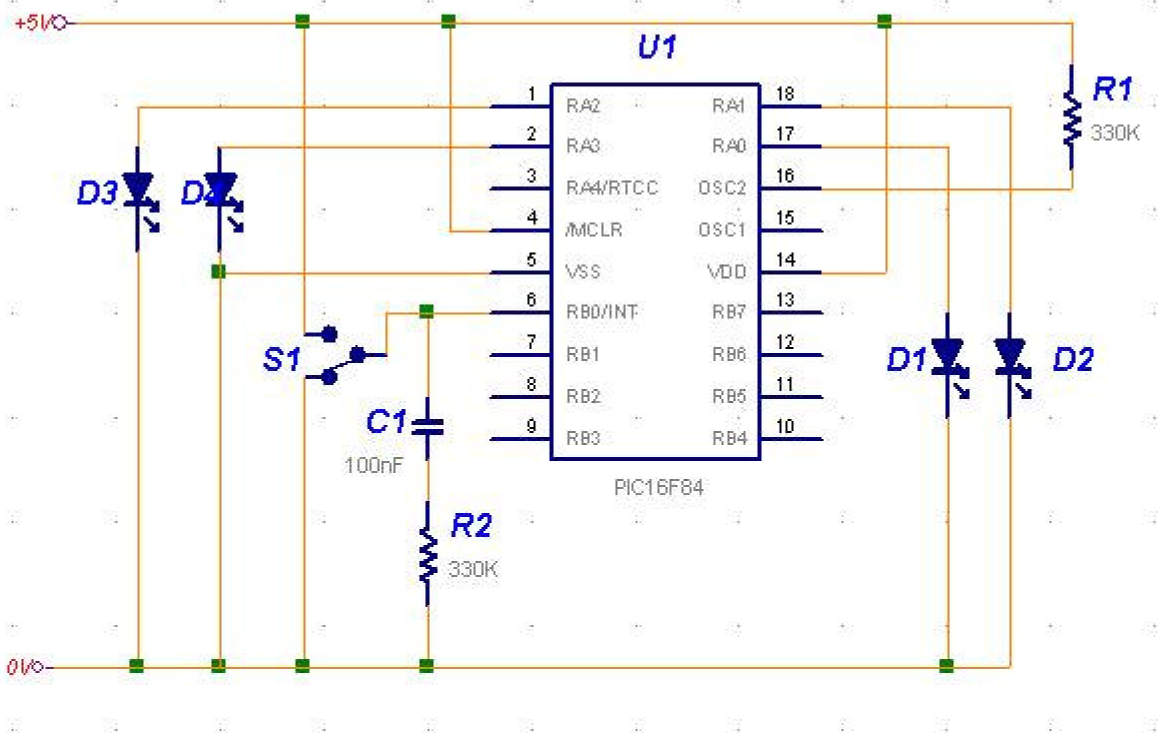
All that is left to do now is put everything together and also define values to our constants, which we can do right at the beginning of our program.

Below is the complete program listing. The circuit is shown after the program listing. Every time you turn the switch on, the LEDs will count up in binary from 0000 to 1010 then back to 0000.

```
org 0x00                  ;This is where we come on power up and reset

        ;*******************SETUP CONSTANTS*******************

        INTCON          EQU 0x0B        ;Interrupt Control Register

        PORTB           EQU 0x06        ;Port B register address

        PORTA           EQU 0x05        ;Port A register address

        TRISA           EQU 0x85        ;TrisA register address

        TRISB           EQU 0x86        ;TrisB register address

        STATUS          EQU 0X03        ;Status register address

       COUNT           EQU 0x0c        ;This will be our counting variable

        TEMP            EQU 0x0d        ;Temporary store for w
        register

 goto    main                          ;Jump over the interrupt address

;***************INTERRUPT ROUTINE***************

 org             0x04            ;This is where PC points on an interrupt

 movwf           TEMP            ;Store the value of w temporarily

incf             COUNT,1         ;Increment COUNT by 1, and put the result

                                  ;back into COUNT

movlw 0x0A                       ;Move the value 10 into w

subwf            COUNT,0         ;Subtract w from COUNT, and put the

                                 ;result in w

btfss            STATUS,0        ;Check the Carry flag. It will be set if

                                 ;COUNT is equal to, or is greater than w,

                                 ;and will be set as a result of the subwf

                                 ;instruction
 goto            carry_on        ;If COUNT is <10, then we can carry on

 goto            clear           ;If COUNT is >9, then we need to clear it


carry_on

 bcf             INTCON,0x01 ;We need to clear this flag to enable

                                  ;more interrupts

 movfw          TEMP            ;Restore w to the value before the interrupt

  retfie                          ;Come out of the interrupt routine


clear

 clrf            COUNT          ;Set COUNT back to 0

 bcf             INTCON,1       ;We need to clear this flag to enable

                                 ;more interrupts

retfie                           ;Come out of the interrupt routine


;*******************Main Program*********************

 main

 ;*******************Set Up The Interrupt Registers****

 bsf              INTCON,7       ;GIE – Global interrupt enable (1=enable)

 bsf              INTCON,4       ;INTE - RB0 Interrupt Enable (1=enable)

 bcf             INTCON,1       ;INTF - Clear FLag Bit Just In Case

 ;*******************Set Up The Ports******************

  bsf            STATUS,5      ;Switch to Bank 1

 movlw 0x01
 movwf TRISB                    ;Set RB0 as input

 movlw 0x10

 movwf TRISA                    ;Set R 0 to RA3 on PortA as output

  bcf                STATUS,5  ;Come back to Bank 0

 ;*******************Now Send The Value Of COUNT To Port A

 loop

movf               COUNT,0       ;Move the contents of Count into W

movwf              PORTA         ;Now move it to Port A

goto               loop          ;Keep on doing this

 end                              ;End Of Program
```

The Circuit Diagram

Below is the circuit diagram that will work for the code above. There are two things in the diagram that may throw you. First, I have not included a timing capacitor in the oscillator circuit. This is a clever little trick that you can try if you run out of capacitors. The capacitance comes from the stray capacitance between the oscillator pin and ground. so, with the resistor and the stray capacitance, we have an RC oscillator. Okay, this is not an accurate way of doing it, as the stray capacitance will vary from circuit to circuit. But, I thought you may be interested in seeing this sort of thing. Secondly, I have included a de-bouncing circuit across the switch. This is needed because every time you flick a switch, the contacts will bounce. This will make the PIC think there have been more than one switches. With the de-bouncing circuit, when the switch goes high, the capacitor charges up. no matter how many times the switch goes to +5V, the capacitor will only charge once. The capacitor is discharged when the switch is thrown the other way. If you want to see the effects of switch bounce, then disconnect the capacitor and resistor across the switch.


+5V
0V
U1
PIC16F84
1 RA2
2 RA3
3 RA4/RTCC
4 /MCLR
5 VSS
6 RB0/INT
7 RB1
8 RB2
9 RB3
18 RA1
17 RA0
16 OSC2
15 OSC1
14 VDD
13 RB7
12 RB6
11 RB5
10 RB4
R1 330K
R2 330K
C1 100nF
S1
D1
D2
D3
D4

Tutorial 13

The Watchdog Timer

We are now going to look at an internal timer, called a Watchdog Timer

So what is a Watchdog Timer?

Suppose you have written a program that is continuously running on a PIC. Now, you want to make sure that this program is always running, and that no matter what happens it will never stop. The first thing you would have, of course, is a loop back at the end of the program that brings us back to the start of the program. But consider this case. Let us say that the PIC is monitoring an input. When this input goes high, it jumps to another part of the program and waits for another pin to go high. If the second pin doesn't go high, the PIC will just sit there and wait. It will only exit if the second pin goes high. Let us consider another example. Suppose you have written a program. You have compiled it successfully, and you have even simulated it over and over again using a simulator such as MPLAB. Everything seems to work fine. You program the PIC and place it into a circuit. However after a long period of time, the program gets stuck somewhere and the PIC gets caught in a loop. What's needed in both cases is some kind of reset if the program gets stuck. This is the purpose of a watchdog circuit.

A watchdog circuit is nothing new. Many microprocessors and microcontrollers have them. But how does it work? Well, inside the PIC there is a resistor/capacitor network. This provides a unique clock, which is independent of any external clock that you provide in your circuit. Now, when the Watchdog Timer (abbreviated to WDT) is enabled, a counter starts at 00 and increments by 1 until it reaches FF. When it goes from FF to 00 (which is FF + 1) then the PIC will be reset, irrespective of what it is doing. The only way we can stop the WDT from resetting the PIC is to periodically reset the WDT back to 00 throughout our program. Now you can see that if our program does get stuck for some reason, then the WDT will not be set. The WDT will then reset the PIC, causing our program to restart from the beginning.

In order to use the WDT, we need to know three things. First, how long have we got before we need to reset the WDT, secondly how do we clear it. Finally, we have to tell the PIC programming software to enable the WDT inside the PIC. Let's look at these separately.

WDT Times

The PIC data sheet specifies that the WDT has a period from start to finish of 18mS. This is dependant several factors, such as the supply voltage, temperature of the PIC etc. The reason for the approximation is because the WDT clock is supplied by an internal RC network. The time for an RC network to charge depends on the supply voltage. It also depends on the component values, which will change slightly depending on their temperature. So, for the sake of simplicity, just take it that the WDT will reset every 18mS. We can, however, make this longer. Inside the PIC is a thing called a Prescaler. We can program this prescaler to divide the RC clock. The more we divide the RC clock by, the longer it takes for the WDT to reset.

The prescaler is located in the OPTION register at address 81h, bits 0 to 2 inclusive. Below is a table showing the bit assignments with the division rates and the time for the WDT to time out:

Bit *2,1,0*	*Rate*	*WDT Time*
0,0,0	1:1	18mS
0,0,1	1:2	36mS
0,1,0	1:4	72mS
0,1,1	1:8	144mS
1,0,0	1:16	288mS
1,0,1	1:32	576mS
1,1,0	1:64	1.1Seconds
1,1,1	1:128	2.3Seconds

Remember these times are irrespective of your external clock frequency. Think of these times as real time, rather than clock times. To help make this clear, let us suppose we want the WDT to reset our PIC after about half a second as a failsafe. The nearest we have is 576mS, or 0.576 seconds. All we do is send b'101' to our OPTION register, as follows:

```
movlw           b'101'  ;This is 0x05 in Hex
movwf            81h     ;This is the Option Register
```

Simple, really. Now, there is a catch. By default the prescaler is assigned to the other internal timer. This means that we have to change the prescaler over to the WDT. First, we have to reset the other counter to 0 first. We then have to change to Bank 1 to assign the prescaler to the WDT and to set up the time, and then come back to Bank 0. The code is below, where xx is the prescaler time:

```
bcf             STATUS,0      ;make sure we are in bank 0
clrf            01h            ;address of the other timer – TMR0
```

```
bsf         STATUS,0     ;switch to bank 1
clrwdt                    ;reset the WDT and prescaler
movlw       b'1xxx'      ;Select the new prescaler value and assign
movwf       OPTION       ;it to WDT
bcf         STATUS,0     ;come back to bank 0
```

The CLRWDT command above is how we clear the WDT before it resets the PIC. So, all we need to do is calculate where in our program the WDT will time out, and then enter the CLRWDT command just before this point to ensure the PIC doesn't reset. If your program is long, bear in mind that you may need more than one CLRWDT. For example, if we use the default time of 18mS, then we need to make sure that the program will see CLRWDT every 18mS.

So now we come to the point where we need to work out how long our code takes in real time. The principle is very simple, but could cause you to pull your hair out!

Instruction Timing

As you are probably already aware, the PIC takes the external clock timing and divides it by 4. This internal time is called an instruction cycle. Now if we have, say, a 4MHz xtal connected to the PIC, internally the PIC will run at 1MHz. In timing terms, this is 1/(4MHz/4) = 1uS. Now, some instructions take just one instruction cycle to complete, i.e. 1uS using a 4MHz crystal, while others take two cycles – 2uS – to complete. The data sheet tells us how many cycles each instruction takes. The easiest way to remember this is quite simple. Assume ALL instructions take 1 cycle. But, if an instruction causes the program to go somewhere else, then it will take 2 cycles. Let me give you a couple of examples. The movwf command takes only one cycle, because it is only moving data from one place to another. The goto command takes 2 cycles, because it is causing the Program Counter (PC) to go elsewhere in the program. The RETURN command takes 2 cycles, because it is causing the PC to go back in the program. I think you can see the pattern here. However, there are four commands which can take 1 or 2 cycles. These are DECFSZ, INCFSZ, BTFSC and BTFSS. These commands have one thing in common. They will skip the next instruction is a certain condition is met. If that condition is not met, then the next instruction will be carried out. For example, the DECFSZ command will decrement the value stored in the F register by 1. If the result is not 0, then the next instruction will be executed. This instruction therefore takes 1 cycle. If the result is 0, then the next instruction will be skipped, and the one following that will be executed. In this instance the instruction takes 2 cycles. The reason is that the instruction alters the value of the PC. It needs one cycle to carry out the function, and it will need another to alter the PC by an extra one.

To clarify this, let us look at a sample code, and work out how many instruction cycles it takes.

```
        movlw       02
        movwf       COUNT
loop    decfsz      COUNT
        goto        loop
end
```

Our first instruction simply moves the value 02 into w. This does not cause the program to off course, therefore it is only 1 cycle. The next instruction is similar, in as much that it moves the contents of the w register into COUNT. Again, this will be 1 cycle. Now, the next instruction will first decrement COUNT by 1. This is 1 cycle. It will then do a test to see if COUNT is equal to 0. At this stage it doesn't, and so we move onto the next instruction. The next instruction is a goto statement, and so is 2 cycles long. We come back to our decfsz instruction, which decrements COUNT by 1 again. This is another instruction cycle. It does a test to see if COUNT is equal to 0. This time it does, and so the next instruction is skipped. To skip the next instruction requires another cycle. We reach the end of the program. So in total, with the value 02 placed into COUNT, this program will take a total of 7 cycles. If we were using a 4MHz crystal for our clock, then the program will take:

1/(4MHz/4) = 1uS per cycle, therefore 7 cycles takes 7 x 1uS = 7uS.

So you can see that it can get a little confusing when you have instructions like DECFSZ.

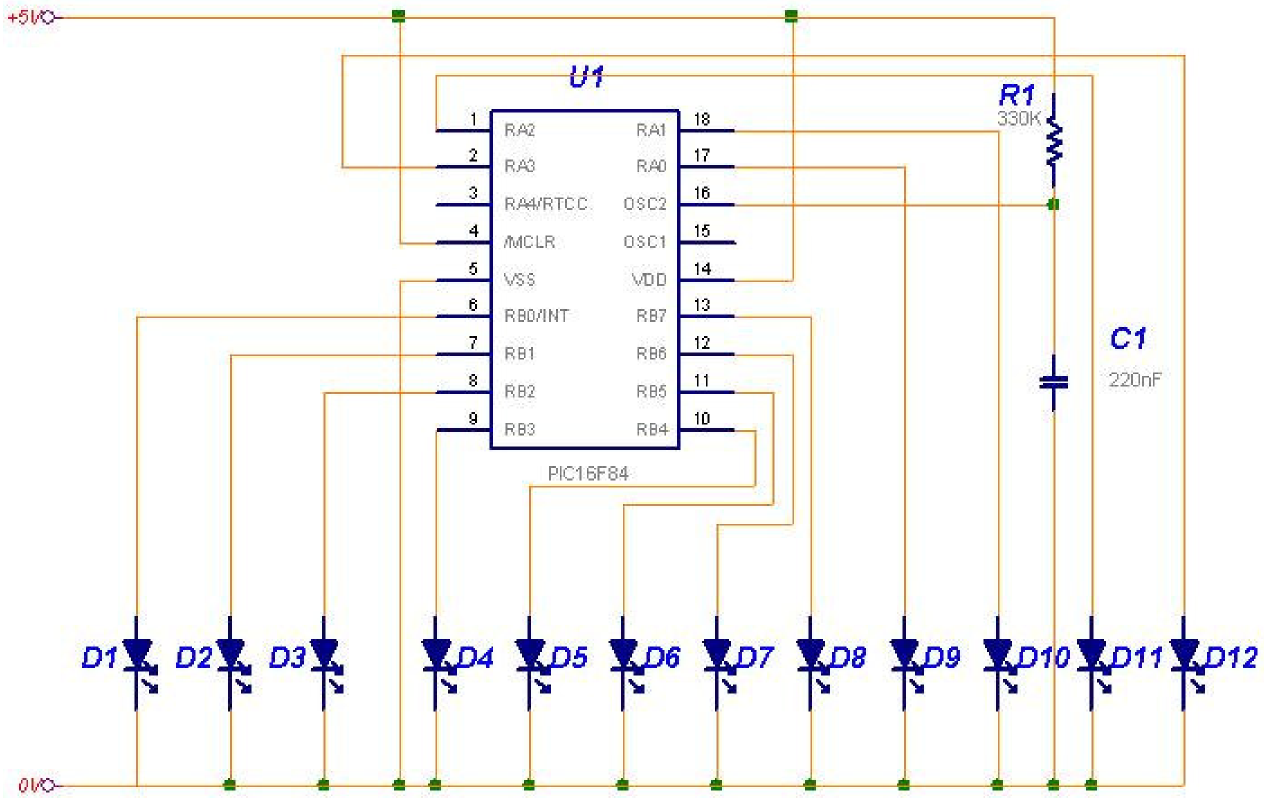
Programmer Software

Inside the PIC there are things called 'Fuses'. These are not the same as the fuses you would find in a mains plug, but electronic switches which are 'blown' by the programmer. Now, one of these fuses has to be 'blown' in order for the WDT to operate. There are two ways of doing this. One way is to write a couple of lines at the beginning of your program to tell the PIC programming software to enable or disable certain fuses. The other way is to tell the PIC programming software manually which fuses to enable. We will look at getting your program to instruct the programming software in a later tutorial, when we look at including other files and macros. To tell the programming software manually, varies from program to program. The documentation that came with the programmer should tell you how to do this. As I am using the PICALLW software, which is linked on my main page, I will explain how to do change fuses within this program. The fuses are configured by pressing the F3 key, or clicking on the 'Config' button. Then you can select the fuse you want enabled, in this case the WDT, by clicking on the box next to it.

Sample Program

Let us write a program, where we will turn on the WDT, and let the PIC perform a function. We will first of all periodically clear the WDT, to show that the program works, and then remove the CLRWDT command to show that the PIC will indeed reset.

The program I have chosen is the one used in tutorial 9 where we cause a row of LEDs to light up one at a time from left to right, then right to left. The circuit is shown below, and with the RC values shown will give us a clock frequency of 8KHz. This clock speed will allow us to actually see the LEDs moving one by one. I chose this program because it is slow enough for us to play with the WDT, and you can easily see when the PIC is reset. I have removed the original comments, and I have replaced them with a description of the WDT lines, a running total of the time from the start (assuming a 8KHz clock), and the number of clock cycles at each line.



```
TIME            equ     9FH             ; Variable for the delay loop.
PORTB           equ     06H             ; Port B address.
TRISB           equ     86H             ; Port B Tristate address.
PORTA           equ     05H             ; Port A address.
TRISA           equ     85H             ; Port A Tristate address.
STATUS          equ     03H             ; Page select register.
COUNT1          equ     0CH             ; Loop register.
COUNT2          equ     0DH             ; Loop register.

        bsf     STATUS,5                ; 1 cycle, 0.5mS
        movlw   00H                     ; 1 cycle, 1.0mS
        movwf   TRISB                   ; 1 cycle, 1.5mS
        movlw   00H                     ; 1 cycle, 2.0mS
        movwf   TRISA                   ; 1 cycle, 2.5mS
        bcf     STATUS,5                ; 1 cycle, 3.0mS
        movlw   00H                     ; 1 cycle, 3.5mS
        movwf   PORTA                   ; 1 cycle, 4.0mS

        ; Start of main program

 RUN

        movlw   01H                     ; 1 cycle, 4.5mS
        movwf   PORTB                   ; 1 cycle, 5.0mS
        call    DELAY                   ; 2 cycles, 486mS
        call    DELAY                   ; 2 cycles, 967mS

 ; Move the bit on Port B left, then pause.

        rlf     PORTB,1                 ; 1 cycle, 967.5mS
        call    DELAY                   ; 2 cycles, 1.45S
        call    DELAY                   ; 2 cycles, 1.93S
        rlf     PORTB,1                 ; 1 cycle, 1.93S
        call    DELAY                   ; 2 cycles, 2.41S
        call    DELAY                   ; 2 cycles, 2.89S
        rlf     PORTB,1                 ; 1 cycle, 2.89S
        call    DELAY                   ; 2 cycles, 3.37S
        call    DELAY                   ; 2 cycles, 3.85S
        rlf     PORTB,1                 ; 1 cycle, 3.85S
        call    DELAY                   ; 2 cycles, 4.34S
        call    DELAY                   ; 2 cycles, 4.82S
        rlf     PORTB,1                 ; 1 cycle, 4.82S
        call    DELAY                   ; 2 cycles, 5.30S
        call    DELAY                   ; 2 cycles, 5.78S
        rlf     PORTB,1                 ; 1 cycle, 5.78S
        call    DELAY                   ; 2 cycles, 6.26S
        call    DELAY                   ; 2 cycles, 6.74S
        rlf     PORTB,1                 ; 1 cycle, 6.74S
        call    DELAY                   ; 2 cycles, 7.22S
        call    DELAY                   ; 2 cycles, 7.70S
        rlf     PORTB,1                 ; 1 cycle, 7.70S

; Now move onto Port A, and move the bit left.

        rlf     PORTA,1                 ; 1 cycle, 7.70S
        call    DELAY                   ; 2 cycles, 8.19S
        call    DELAY                   ; 2 cycles, 8.67S
        rlf     PORTA,1                 ; 1 cycle, 8.67S
        call    DELAY                   ; 2 cycles, 9.15S
        call    DELAY                   ; 2 cycles, 9.63S
        rlf     PORTA,1                 ; 1 cycle, 9.63S
        call    DELAY                   ; 2 cycles, 10.11S
        call    DELAY                   ; 2 cycles, 10.59S
        rlf     PORTA,1                 ; 1 cycle, 10.59S
        call    DELAY                   ; 2 cycles, 11.07S
        call    DELAY                   ; 2 cycles, 11.55S

; Move the bit back on Port A

        rrf     PORTA,1                 ; 1 cycle, 11.55S
        call    DELAY                   ; 2 cycles, 12.04S
        call    DELAY                   ; 2 cycles, 12.52S
        rrf     PORTA,1                 ; 1 cycle, 12.52S
        call    DELAY                   ; 2 cycles, 12.99S
        call    DELAY                   ; 2 cycles, 13.48S
        rrf     PORTA,1                 ; 1 cycle, 13.48S
        call    DELAY                   ; 2 cycles, 13.96S
        call    DELAY                   ; 2 cycles, 14.44S
        rrf     PORTA,1                 ; 1 cycle, 14.44S

; Now move the bit back on Port B

        rrf     PORTB,1                 ; 1 cycle, 14.44S
        call    DELAY                   ; 2 cycles, 14.92S
        call    DELAY                   ; 2 cycles, 15.40S
        rrf     PORTB,1                 ; 1 cycle, 15.40S
        call    DELAY                   ; 2 cycles, 15.89S
        call    DELAY                   ; 2 cycles, 16.37S
        rrf     PORTB,1                 ; 1 cycle, 16.37S
        call    DELAY                   ; 2 cycles, 16.84S
        call    DELAY                   ; 2 cycles, 17.33S
        rrf     PORTB,1                 ; 1 cycle, 17.33S
        call    DELAY                   ; 2 cycles, 17.81S
        call    DELAY                   ; 2 cycles, 18.29S
        rrf     PORTB,1                 ; 1 cycle, 18.29S
        call    DELAY                   ; 2 cycles, 18.77S
        call    DELAY                   ; 2 cycles, 19.25S
        rrf     PORTB,1                 ; 1 cycle, 19.25S
        call    DELAY                   ; 2 cycles, 19.73S
```

```
          call      DELAY                    ; 2 cycles, 20.22S
          rrf       PORTB,1                  ; 1 cycle, 20.22S
          call      DELAY                    ; 2 cycles, 20.70S
          call      DELAY                    ; 2 cycles, 21.18S

          goto      RUN                      ; 2 cycles, 21.18S

 ; Subroutine to give a delay between bit movements.
 ;Total of 957 cycles, 480mS

DELAY
          movlw  TIME                        ; 1 cycle
          movwf  COUNT1                      ; 1 cycle

LOOP1                                        ;
         decfsz   COUNT1                     ; 9F x 1 cycle + 1 cycle = 160 cycles
         goto     LOOP1                      ; 9E x 2 cycles = 316 cycles
         movwf COUNT1                        ; 1 cycle

LOOP2                                        ;
          decfsz   COUNT1                    ; 9F x 1 cycle + 1 cycle = 256 cycles
          goto     LOOP2                     ; 9E x 2 cycles = 316 cycles

return                                       ; 2 cycles

 END                                         ;
```

With an 8KHz clock, it takes just under 1 second for the next LED illuminates, and it takes a total of about 21 seconds to run from one end to the other and back again i.e. to go through the routine once only. The delay routine takes 480mS, and we are calling it twice before moving the bit on the ports. Now, we need to periodically reset the WDT. The largest time we can set the WDT is 2.3 seconds, and the next one down form this is 1.1 seconds. We have two options here. We could make a call to a subroutine to clear the WDT after the two delays have finished, or we could incorporate the CLRWDT within the delay itself. I have decided, for no real reason at all, to incorporate the CLRWDT within the delay loop.

```
TIME         equ     9FH          ; Variable for the delay loop.
PORTB        equ     06H          ; Port B address.
TRISB        equ     86H          ; Port B Tristate address.
PORTA        equ     05H          ; Port A address.
TRISA        equ     85H          ; Port A Tristate address.
STATUS       equ     03H          ; Page select register.
```

```
COUNT1        equ      0CH              ; Loop register.
COUNT2        equ      0DH              ; Loop register.

OPT            equ      81h              ; Option Register to control the
WDT


;*************Set up the ports, WDT and prescaler******************

          clrf                01h                  ;Clear TMR0
          bsf       STATUS,5                ;Switch to bank 1
          clrwdt                                   ;reset the WDT and prescaler
          movlw            b'1101'           ;Select the new prescaler value and
assign
          movwf            OPT                ;it to WDT


          movlw  00H                          ; Now set up the ports
          movwf  TRISB                        ;
          movlw  00H                          ;
          movwf  TRISA                        ;
          bcf       STATUS,5               ;Come back to bank 0

          movlw  00H                          ;
          movwf  PORTA                        ;



 ;*************Start of main program*****************************

 RUN

          movlw 01H                           ;
          movwf PORTB                         ;
          call      DELAY                     ;
          call      DELAY                     ;

 ; *************Move the bit on Port B left, then pause.**************

           rlf        PORTB,1                 ;
           call      DELAY                    ;
           call      DELAY                    ;
          rlf        PORTB,1                  ;
           call      DELAY                    ;
           call      DELAY                    ;
          rlf        PORTB,1                  ;
           call      DELAY                    ;
           call      DELAY                    ;
          rlf        PORTB,1                  ;
           call      DELAY                    ;
```

```
         call     DELAY                 ;
        rlf       PORTB,1               ;
         call     DELAY                 ;
         call     DELAY                 ;
        rlf       PORTB,1               ;
         call     DELAY                 ;
         call     DELAY                 ;
        rlf       PORTB,1               ;
         call     DELAY                 ;
         call     DELAY                 ;
        rlf       PORTB,1               ;

; *************Now move onto Port A, and move the bit left.***********

        rlf       PORTA,1               ;
        call      DELAY                 ;
        call      DELAY                 ;
        rlf       PORTA,1               ;
        call      DELAY                 ;
        call      DELAY                 ;
        rlf       PORTA,1               ;
        call      DELAY                 ;
        call      DELAY                 ;
        rlf       PORTA,1               ;
        call      DELAY                 ;
        call      DELAY                 ;

 ;************** Move the bit back on Port A************************

         rrf       PORTA,1               ;
         call     DELAY                 ;
         call     DELAY                 ;
        rrf       PORTA,1               ;
         call     DELAY                 ;
         call     DELAY                 ;
        rrf       PORTA,1               ;
         call     DELAY                 ;
         call     DELAY                 ;
        rrf       PORTA,1               ;

 ;****************** Now move the bit back on Port B******************

         rrf       PORTB,1               ;
          call     DELAY                 ;
          call     DELAY                 ;
         rrf       PORTB,1               ;
          call     DELAY                 ;
          call     DELAY                 ;
         rrf       PORTB,1               ;
          call     DELAY                 ;
          call     DELAY              ;
         rrf       PORTB,1             ;
          call     DELAY              ;
          call     DELAY              ;
         rrf       PORTB,1             ;
          call     DELAY              ;
          call     DELAY              ;
         rrf       PORTB,1             ;
          call     DELAY              ;
          call     DELAY              ;
         rrf       PORTB,1             ;
          call     DELAY              ;
          call     DELAY              ;

          goto    RUN                  ;



; ******************Subroutine to give a delay between bit
movements.******

 DELAY

        movlw  TIME                  ;
        movwf  COUNT1               ;

LOOP1                               ;
        decfsz   COUNT1             ;
        goto     LOOP1              ;
        movwf COUNT1                ;

LOOP2                               ;
        decfsz   COUNT1             ;
        goto     LOOP2

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This part resets the WDT                                     ;;
;;Comment out or remove this command to see the WDT           ;;
;;    in action.  It should reset the PIC                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        clrwdt                       ;This simply resets the WDT.


;***************Return from our original DELAY routine***************

 return                              ;

END                                 ;
```

If you comment out, or remove the CLRWDT command, you will find that the PIC will not go past lighting the second LED. This is because the WDT is resetting the PIC. With the CLRWDT in place, the program works as it should.