Fast Track to PIC Programming

You may know that when it comes to machines, programming is often used to tell the machine how to interact with its world. But have you ever wondered how this programming is actually physically implemented? One way is to use PICs (Programmable Interrupt Controllers). These chips allow you to write code that reads input signals, performs functions and sends signals to outputs based on conditions that you define. This tutorial will explain the basic process of writing programs for PICs and “burning” those programs to the device.

MOTIVATION AND AUDIENCE

The focus of this tutorial is to get you quickly acquainted with PICs so that you can start using them in your applications. As such, this tutorial will teach you how to:

- ***Write code to define outputs.***
- ***Write code to read inputs and use those inputs to affect outputs.***
- ***Write code to react to a clock cycle.***
- ***“Burn” code into the device.***

To do this, it is assumed that you already know how to:

- ***Read an electrical schematic.***
- ***Construct and solder an electrical circuit onto a protoboard.***

The rest of the tutorial is presented as follows:

PARTS LIST AND SOURCES

The majority of the parts will be required to construct a circuit to test your programs on. The parts listed in Table 1 are consumables used in the circuit:

TABLE 1

PART DESCRIPTION	VENDOR	PART	PRICE (2002)	QTY
PIC16F84-04/P	JAMECO	145111	5.95	1
40-PIN ZIF SOCKET	JAMECO	104029	10.95	1
PUSHBUTTON SWITCH	JAMECO	71642	1.49	1
8-POSITION DIP SWITCH	JAMECO	38842	0.79	1

4 MHZ CRYSTAL CLOCK OSCILLATOR	JAMECO	27967	1.89	1
0.1 UF CAP	JAMECO	151116	1.00 FOR BAG OF 10	1
0.1 INCH HEADERS	JAMECO	160881	0.39	1
SIPP 30-PIN WIREWRAP SOCKET	JAMECO	104053	1.95	1
T1-3/4 GREEN LED	JAMECO	104256	0.29	1
100 OHM RESISTOR				1
10 KILO OHM RESISTOR				1
220 OHM RESISTOR				9
3.3 KILO OHM RESISTOR				8
6 INCH PROTOTYPING CIRCUIT BOARD	RADIO SHACK	276-170	2.99	1
2-3/4 X 3-3/4 PROTOTYPING CIRCUIT BOARD	RADIO SHACK	276-158	2.39	1

The PIC we will be using (PIC16F84) was chosen because it is a very common PIC and because it can be programmed and reprogrammed without additional hardware (many PICs must be exposed to UV light to erase existing programs).

To construct the circuit, you will also need:

- ***a soldering iron with a fine point***
- ***materials for soldering (solder, flux, etc.)***
- ***small gauge wire***
- ***wire strippers***
- ***multimeter***
- ***DC power supply***

The items listed above can all be purchased from an electronics store such as Radio Shack. Some hardware such as Home Depot carry tools like wire strippers and multimeters.

There are many utilities for writing, compiling and burning PIC code. This tutorial uses the following software/hardware to program the PIC:

- ***MPLAB for Windows (Microchip)***
- ***PICSTART Plus device programmer***

The MPLAB software contains the text editor, compiler (MPASM) and device programmer software (PICSTART Plus) in a single program, thereby centralizing all your PIC programming needs. The book Easy Pic'n by David Benson is also an invaluable resource in learning how to use PICs. This tutorial refers to the book to clarify some of the code.

CONSTRUCTION

The circuit used to test your PIC programs is depicted in Figure 1.

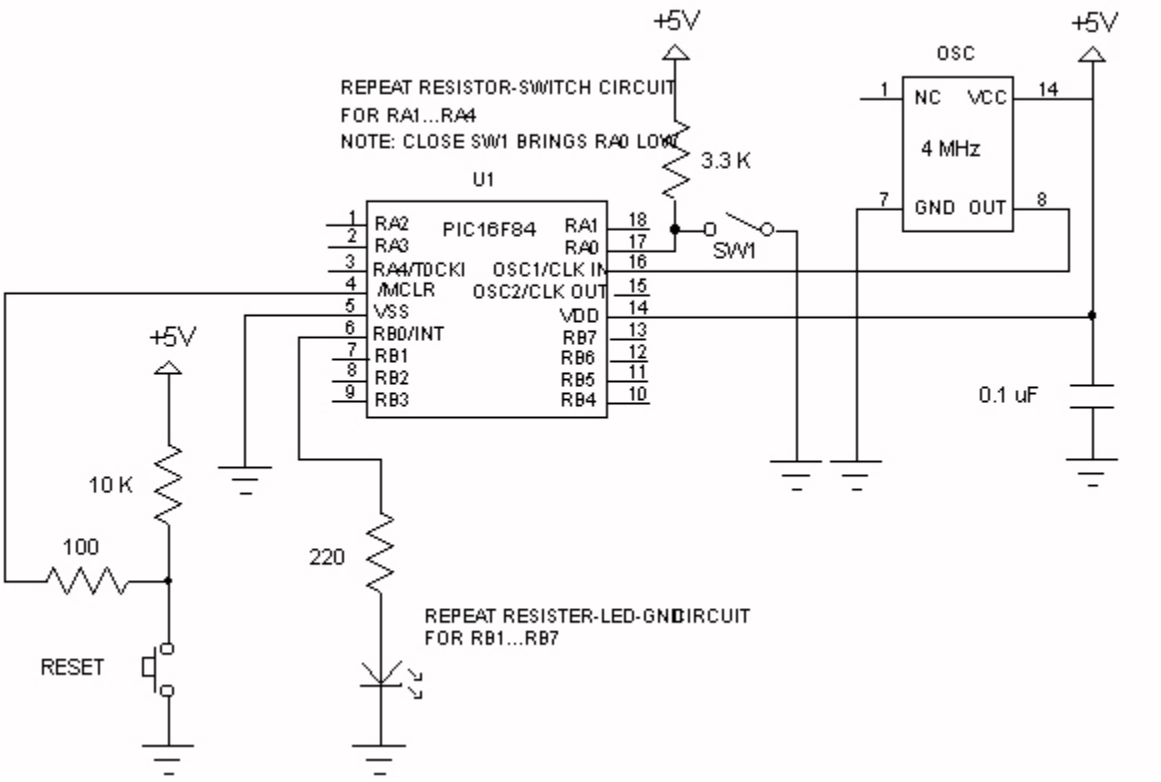

Figure 1

This circuit is set up to test and display basic PIC functions. In this tutorial, Port B on the PIC (Pins 6- 13) is used as an output. LED's are connected to all 8 of the Port B lines, and will light up when the line is set to a logic high, or '1'. Port A (Pins 17, 18, 1, 2 and 3) is used as an input. Its lines are connected to dip switches, which will set the line to a logic high when the switch is in the 'On" position.

It is recommended that the LED and dip switch circuits be constructed on a separate board and connected to the PIC via a cable. This allows you to construct more complicated circuits in the future and easily switch between circuits.

A ZIF (Zero Insertion Force) socket is used to make repeated installation and removal of the PIC easy, and to help prevent the pins on the PIC from being damaged.

PROGRAMMING

What follows are a few sample codes that illustrate the functionality of the PIC. In each instance, the code will be given, followed by an explanation of how the code works.

Example 1: Outputting to LEDs

In this example you will learn how to define pins as output and how to set those pins to a logic hi or logic low. The following code sets pins corresponding to B0,B2,B4,B6 to a logic low and pins corresponding to B1,B3,B5,B7 to a logic hi.

outLed.asm

```
; FILE: outLed.asm
; AUTH: (Your name here)
; DATE: (date)
; DESC: Makes B0,B2,B4,B6 low and B1,B3,B5,B7 hi
; NOTE: Tested on PIC16F84-04/P.
; REFs: Easy Pic'n p. 23 (Predko p. 173 is bogus?)

        list    p=16F84
        radix   hex

;-----------------------------------------------------------------------
;       cpu equates (memory map)
myPortB equ     0x06            ; (p. 10 defines port address)
;-----------------------------------------------------------------------

        org     0x000

start   movlw   0x00            ; load W with 0x00 make port B output (p. 45)
        tris    myPortB         ; copy W tristate, port B outputs (p. 58)

        movlw   b'10101010'     ; load W with bit pattern (p. 45)
        movwf   myPortB         ; load myPortB with contents of W (p. 45)

circle  goto    circle  ; done

        end

;-----------------------------------------------------------------------
; at burn time, select:
;       memory uprotected
;       watchdog timer disabled
;       standard crystal (4 MHz)
;       power-up timer on
```

In evaluating the code above, first and foremost it is important to realize that everything following a semicolon is a comment, and is ignored by the compiler. The actual code used by the compiler to program the PIC is shown below:

```
        list    p=16F84
        radix   hex

myPortB equ     0x06

        org     0x000

start   movlw   0x00
        tris    myPortB
        movlw   b'10101010'
        movwf   myPortB
circle  goto    circle
```

```
        end
```

HEADER

The first portion of code is called the header. This information helps the compiler to format the code correctly. In our case, every header will be identical.

The first line

```
        List    p=16F84
```

describes the type of device that the program is to be burned to. The line

```
        radix   hex
```

tells the compiler what format numbers are in unless otherwise specified. In this case, the format is hexadecimal.

EQUATES

The next section of the program is called the equates. This is similar to variable declaration in other programming languages. Labels are assigned to addresses. Later, whenever that label is referred to in the program, the compiler looks up its address.

The line

```
portB   equ     0x06
```

assigns ‘portB’ to the file register located at 0x06. Port B is always located at this file register.

ORG

In the line

```
        org     0x000
```

org stands for origin. The org function has a few special uses, but this tutorial only uses the org statement as shown. When used in this manner, the org statement defines the beginning of the code.

INSTRUCTIONS

The next portion of code contains the actual instructions that tell the PIC what to do.

```
start   movlw   0x00
```

This line is labeled as the start of the code. The function ‘movlw’ moves a literal (a number) to the file register W. You can not directly assign values to file registers. All values must first be passed through the W register.

```
        tris    myPortB
```

This command is outdated, though its still compatible with the version of software in use. The ‘tris’ command tells the compiler that the current W value will map the lines of the selected port as inputs or outputs (a ‘1’ in W means input, a ‘0’ in W means output).

```
        movlw   b'10101010'
```

This command moves the binary number ‘10101010’ to the W register.

```
        movwf   myPortB
```

The contents of the W register are now assigned to Port B, setting the appropriate pins as hi or low.

```
circle  goto    circle
```

This command labels the line of code as ‘circle’, and then refers bac to itself, thereby setting the program in a continuous loop.

END

Finally the compiler is told that it has reached the end of the code with an end statement.

```
        end
```

All programs must have an end statement.

Example 2: Inputting from dip switches

Now that you have the fundamentals down, this program will illustrate how inputting is used to control outputs. In this example, all of the pins in port A are set as inputs (an should be connected to dip switches). When a dip switch is turned on, its value is passed to the corresponding output on port B, thereby lighting an LED.

Dip2Led.asm

```
; FILE: Dip2Led.asm
; AUTH: (Your name here)
; DATE: (date)
; DESC: Read Port A DIP switch and display on Port B LEDs
; NOTE: Tested on PIC16F84-04/P.
; REFs: Easy Pic'n p. 60

        list    p=16F84
```

```
        radix   hex

;------------------------------------------------------------------------
;       cpu equates (memory map)
myPortA equ     0x05
myPortB equ     0x06            ; (p. 10 defines port address)
;------------------------------------------------------------------------

        org     0x000
start   movlw   0x00            ; load W with 0x00 make port B output (p. 45)
        tris    myPortB         ; copy W tristate to port B outputs (p. 58)

        movlw   0xFF            ; load W with 0xFF make port A input
        tris    myPortA         ; copy W tristate to port A

        movf    myPortA, w      ; read port A DIP and store in W
        movwf   myPortB         ; write W value to port B LEDs

circle  goto    start           ; loop forever

        end

;------------------------------------------------------------------------
; at burn time, select:
;       memory uprotected
;       watchdog timer disabled
;       standard crystal (4 MHz)
;       power-up timer on
```

This code is very similar to the previous code. Port B is declared as outputs in the same manner. In this case, port A is defined as inputs by the code

```
        movlw   0xFF
        tris    myPortA
```

This fills the W register with 1's, and uses those 1's to declare port A as inputs. Instead of assigning a literal to port B, port A is read into the W register.

```
        movf    myPortA, w
```

This command says move the file register 'myPortA' into the W register. And as before, the contents of the W register are assigned to the LED's at port B

```
        movwf   myPortB
```

The code is then told to return to the start and execute again.

Example 3: Reacting to a clock cycle

There is one other important functionality to the PIC that can be extremely useful. The PIC has a built in counter whose frequency is dependent upon the external oscillator in the circuit and upon certain options you set. These options along with the calculations for determining the frequency are explained below.

Clock.asm

```
; FILE: Clock.asm
; AUTH: (Your name here)
; DATE: (date)
; DESC: 1.0 - Internal timer, blink LED every 32.8 msec
; NOTE: Tested on PIC16F84-04/P.
;       4 MHz crystal yields 1 MHz internal clock frequency.
;       "option" is set to divide internal clock by 256.
;       This results in 1 MHz/256 = 3906.25 Hz or 256 usec.
;       tmr0 bits 0 through 7 (255 decimal) is checked, thus yielding
;       255*256 usec = 65.28 msec delay loop
; REFs: Easy Pic'n p. 113

        list    p=16F84
        radix   hex

;-----------------------------------------------------------------------
;       cpu equates (memory map)
portB   equ     0x06            ; (p. 10 defines port address)
tmr0    equ     0x01
;-----------------------------------------------------------------------

        org     0x000
start   clrwdt                  ; clear watchdog timer
        movlw   b'11010111'     ; assign prescaler, internal clock
                                ; and divide by 256 see p. 106
        option
        movlw   0x00            ; set w = 0
        tris    portB           ; port B is output
        clrf    portB           ; port B all low
go      bsf     portB, 0        ; RB0 = 1, thus LED on p. 28
        call    delay
        call    delay
        bcf     portB, 0        ; RB0 = 0, thus LED off
        call    delay
        call    delay
        goto    go              ; repeat forever

delay   clrf    tmr0            ; clear TMR0, start counting
again   btfss   tmr0, 0         ; if bit 0 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 1         ; if bit 1 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 2         ; if bit 2 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 3         ; if bit 3 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 4         ; if bit 4 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 5         ; if bit 5 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 6         ; if bit 6 = 1
        goto    again           ; no, then check again
        btfss   tmr0, 7         ; if bit 7 = 1
        goto    again           ; no, then check again
        return                  ; else exit delay

        end
```

```
;-----------------------------------------------------------------------
; at burn time, select:
;       memory uprotected
;       watchdog timer disabled
;       standard crystal (4 MHz)
;       power-up timer on
```

The first thing notable about this code is a new equate

```
tmr0    equ     0x01
```

This is an 8 bit, read/write counter stored at this particular file register. In our case, the internal clock frequency is 1 MHz (external frequency gets divided by 4). This frequency can be further divided by setting a prescaler value (the value the internal frequency will be divided by). Based on the setting of 3 option bits, the prescaler value can be varied between 8 values ranging from 1 to 256. In our case, all 3 option bits are set, dividing the internal clock frequency by 256. This gives the frequency of the counter to be 1 MHz/256 = 3906.25 Hz. The next command

```
start   clrwdt
```

Clears the watchdog timer and the prescaler value. The following commands

```
        movlw   b'11010111'
        option
```

set the option bits. The significance of the option bits are as follows:

```
Bit     Purpose
0       Prescaler Value
1       Prescaler Value
2       Prescaler Value
3       Prescaler Assignment (0=tmr0, 1=watchdog timer)
4       tmr0 external edge clock select (0=rising, 1=falling)
5       tmr0 clock source (0=internal instruction cycle, 1=external)
6       Interrupt edge select (0=falling, 1=rising)
7       Port B Pullup Enable (0=enabled, 1=disabled)
```

As you can see, in this code, the Prescaler bits are set to 111 (divide by 256), prescaler is sent to tmr0, tmr0 increments on the rising edge, the tmr0 source is the internal instruction cycle clock, interrupts occur on rising edges, and Port B pull-ups are disabled.

Proceeding through the code, the next portion defines Port B as outputs, as has been seen in previous examples. The code following that is the meat of this program.

```
go      bsf     portB, 0
        call    delay
        call    delay
        bcf     portB, 0
        call    delay
        call    delay
        goto    go
```

This loop turns an LED off and on repeatedly. First, the 0 bit in port B is set (made to logical hi). A delay subroutine is then called (explained later) to pause the program for a bit. The 0 bit is then cleared (made to logical low). The program is delayed once more, and then the process repeats. The new concept presented is that of a subroutine

```
delay   clrf    tmr0
again   btfss   tmr0, 0
        goto    again
        btfss   tmr0, 1
        goto    again
        btfss   tmr0, 2
        goto    again
        btfss   tmr0, 3
        goto    again
        btfss   tmr0, 4
        goto    again
        btfss   tmr0, 5
        goto    again
        btfss   tmr0, 6
        goto    again
        btfss   tmr0, 7
        goto    again
        return
```

When delay is called, the program skips to this portion of the code. tmr0 (the counter) is first cleared (and then begins to count). The remaining code tests every bit in tmr0 to see if it is set to hi. If it isn't, it returns to 'again'. If it is set to hi, it skips the next line of code ("goto again") and testes the next bit. This loop continues until the return statement is reached, in which case the subroutine is exited and the code proceeds from where delay was originally called. As can be seen, the delay subroutine will continue until all tmr0 bits equal '1', or until tmr0 counts to 255.

"BURNING" CODE INTO A PIC

Now that you know how to write code, you need to know how to get your code into the PIC. The process of programming a PIC is often referred to as "burning". To burn code into the PIC, we will be using the windows version of MPLAB.

In the MPLAB program, all the information about your program is stored in a project files. Project files contain information about the program, the device you're using, one or more assembly language codes and compiled hex files. The process of burning a PIC contains three major steps. The first is to write the code in assembly language. Once the code is written, it must be compiled into a hex file for programming into the device. After successfully compiling the code, the final step is to program the device.

Begin by opening up the program MPLAB.

Create a new project by going to Project>New Project. In the new project dialogue box select a directory to place the project in and give the new project a name. When you've finished, click "Ok".

You will now see the Edit Project dialogue box. Select the appropriate device, and set the development mode to MPLAB-SIM Simulator. In the project files window click the file that has the name of your file with a .hex extension. Click on the Node Properties button and in the window that appears, click “Ok”. This sets default node properties and allows you to add nodes later.

Exit the Edit Properties box by clicking “Ok”.

Now you may create assembly code. Got to File>New. A blank text editor box should appear. Enter your assembly code into this window. When you are done, save the code (File>Save, give the code a name and click “Ok”).

You must now assign the source code you just made to the project. Go to Project>Edit Project. In the Edit Project dialogue box, click the Add Node button. Browse to find the assembly code you just found. Select it and click “Ok”. The file name for the assembly code should now appear in the window. Click “Ok” to close the Edit Project box.

Now that the source code is associated with the project, its time to compile the code. Go to Project>Make Project. If the compile is successful, you will see the Build Results window appear with the message “Build completed successfully”. If there were errors they will be listed in the window. After compiling successfully, save the project (Project>Save Project).

The final step is to program the device. Select Picstart Plus>Enable Programmer. The Programmer Status dialogue box should appear. At this point, you should have the Picstart Plus device programmer plugged in and th serial cable connected to the serial port on your computer. Place the PIC in the ZIF socket with pin 1 in the top left corner, and lock it in place.

The most important part of this step is to ensure the configuration bits are set appropriately. If you noticed, at the end of each example code was a note indicating how the configuration bits should be set

```
; at burn time, select:
;       memory uprotected
;       watchdog timer disabled
;       standard crystal (4 MHz)
;       power-up timer on
```

Watchdog timer should be set to “off”, Oscillator should be set to “XT” (this is the setting for standard crystal), the memory setting should be set to “off” and the powerup timer should be set to “on”.

When the configuration bits are set correctly, click “Program” and wait for the programmer to finish. If the programmer completes successfully, your code is now in the PIC.

FINAL WORDS

After completing this tutorial you should be able program a PIC to send outputs, read inputs, and create functions using a clock. After creating a program in assembly code, you should be able to burn that program into the device for use in your application.

The concepts shown here were presented in relatively trivial situations. However, these concepts are fundamental to using PIC's for larger, more complex applications.

If you have questions about this tutorial you can email me at **Keithicus@drexel.edu**. Happy PIC'n!