Laboratory 9

Programming a PIC Microcontroller - Part I

Required Components:

- 1 PIC16F84 (4MHz) or PIC16F84A (20MHz) or compatible (e.g., PIC16F88) microcontroller
- 1 4MHz microprocessor crystal (20 pF), **only if using the PIC16F84x**
- 2 22pF capacitors, **only if using the PIC16F84x**
- 1 0.1 μF capacitor
- 1 LED
- 1 330Ω resistor
- 1 1kΩ resistor
- 1 SPST microswitch or NO button

Required Special Equipment and Software:

- Mecanique's Microcode Studio integrated development environment software
- MicroEngineering Labs' PicBasic Pro compiler
- MicroEngineering Labs' U2 USB Programmer

9.1 Objective

Microcontrollers are important parts in the design and control of mechatronic systems. This laboratory introduces the architecture of Microchip's PIC microcontroller, describes the PIC's capabilities, shows how to create programs using microEngineering Lab's PicBASIC Pro, and shows how to wire simple circuits using the PIC and your software. The exercise also shows how to use interrupts in response to sensor inputs to the PIC.

9.2 Introduction

A PIC microcontroller adds sophisticated digital control capabilities when connected to other circuits and devices. A single PIC microcontroller can communicate with other electronic devices, and digitally switch them on or off to control simple operations.

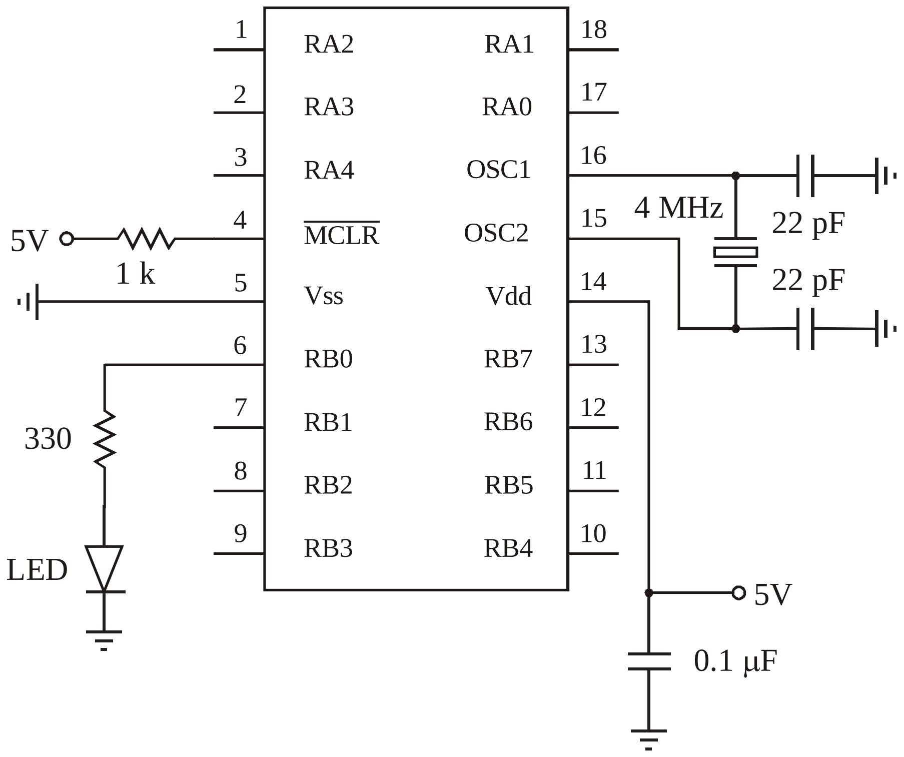
Microchip Technology, Inc. (*www.microchip.com*) produces a family of PIC processors capable of storing programs. The PIC16F84, which we will use in this Lab, contains electrically erasable programmable ROM (EEPROM), which is memory used to store programs. The program in EEPROM can be overwritten many times during the design cycle. The PIC has 64 bytes of data EEPROM and 1792 bytes of program EEPROM for storing compiled programs. It operates at 4 or 8 MHz depending upon an external crystal oscillator or a timer circuit. The PIC16F84A can function up to 20 MHz. The 18-pin PIC has 13 pins capable of operating as either inputs or outputs, designated by software, that can be changed during program execution. Five of the pins are grouped together and referenced as PORTA; another 8 pins are grouped together and referenced as PORTB.

Simple programs may be written in a form of BASIC called PicBasic Pro, which is available from microEngineering Labs, Inc. (*www.melabs.com*). The package includes a compiler that converts PicBasic to assembly language code, and then compiles the assembly code to hexadecimal machine code (hex) that is downloaded to the PIC. The hex executable code is downloaded via a serial port to a PIC using the Microchip Development Programmer hardware using a Windows interface. Once written, the program remains in PIC memory even when the power is removed.

9.2.1 PIC Structure

The PIC16F84 is an 18-pin DIP IC with the pin-out shown in the top of Figure 9.1. It has external power and ground pins (Vdd and Vss), 13 binary input/output (I/O) pins (RA[0-4] and RB[0-7]), and uses an external oscillator (the crystal and capacitor circuit attached to OSC1 and OSC2) to generate a clock signal. The master clear ($\overline{\mathrm{MCLR}}$) pin is active low, meaning the PIC is reset when the pin is grounded. $\overline{\mathrm{MCLR}}$ must be held high during PIC operation, or be driven low on purpose (e.g., by a reset button) to reset the PIC and restart your program. There are many PIC microcontrollers that are pin-compatible with the PIC16F84, including the PIC16F88 shown in the bottom of Figure 9.1. The PIC16F88, like most advanced PICs, provides alternative functions for the pins. Only some of the alternative functions are listed in Figure 9.1 (see the PIC16F88 datasheet online for more information). For example, the RA[0-4] pins can be used as

PIC16F84 (or any other compatible PIC requiring an external oscillator)



PIC16F88 (or any compatible PIC with an internal oscillator)

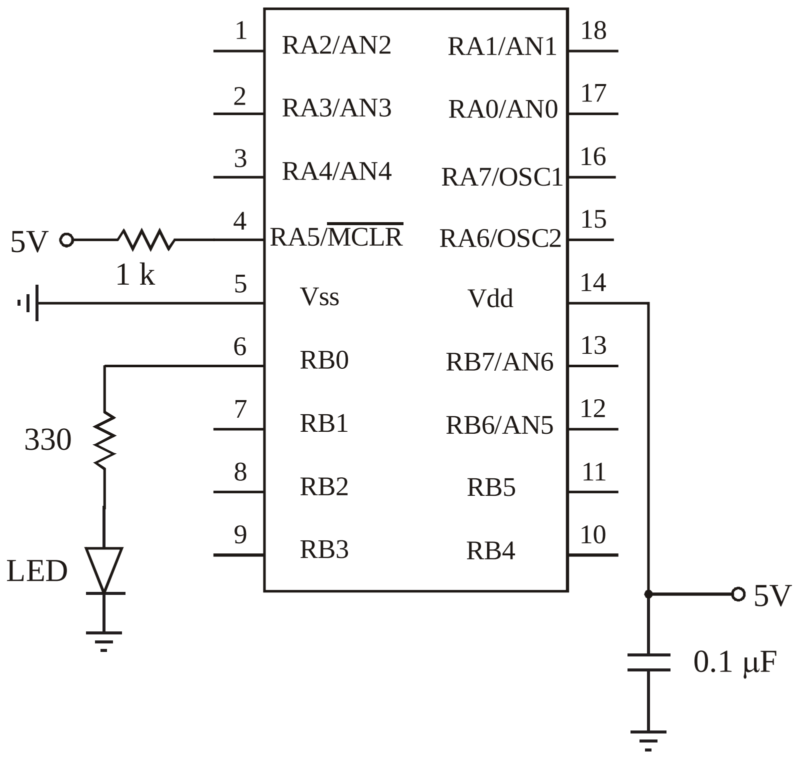


Figure 9.1 PIC and LED wiring diagram for the "blink.bas" example

analog inputs (AN[0-4]) instead, and the $\overline{\text{MCLR}}$ pin can be used as an additional I/O pin (RA5). Also, the PIC16F88, like many PICS, includes an internal oscillator, so it does not require an external clock circuit. Therefore, the OSC1/OSC2 pins can be used as additional I/O pins (RA6-7) instead (and the external crystals and capacitors are not required). Throughout all of the PIC Labs, we will use the PIC16F88 (or some other PIC compatible with the PIC16F84), but it will be configured to look just like the PIC16F84 (except for the oscillator).

The power supply voltage (5 Vdc) and ground are connected to pins labeled Vdd and Vss (pins 14 and 5), respectively. Pin 4 ($\overline{\text{MCLR}}$) is attached to 5Vdc with a 1k resistor to ensure continuous operation. If this pin were left unconnected (floating), the PIC could spontaneously reset itself. With the PIC16F84, an accurate clock frequency can be obtained by connecting a 4MHz crystal (sometimes indicated as XT) across pins 15 and 16 which are also connected to ground through 22pF capacitors. A less expensive and less accurate alternative for setting a clock frequency is to attach an RC circuit to pin 16 while leaving pin 15 unattached (referred to as an RC clock).

The pins labelled RAx and RBx provide binary I/O. They are divided into two groups called PORTs. PORTA refers to pins RA0 through RA4 and PORTB refers to pins RB0 through RB7. PORTA and PORTB are compiler variable names that provide access to registers on the PIC. Each bit within the PORT can be referred to individually by its bit location (e.g., PORTA.3 refers to bit 3 in the PORTA register). For both ports, bit zero (PORTA.0 or PORTB.0) is the least significant bit (LSB). The specifics of how the PORT bits are defined and accessed follow:

> <u>PORTA</u>: Designated in PicBasic Pro code as PORTA.0 through PORTA.4 (5 pins: 17, 18, and 1 through 3). For example, PORTA = %00010001 would set the PORTA.0 and PORTA.4 bits to 1, and set all other bits to 0. The % sign indicates binary number format. For PORTA, the three most significant bits are not required (i.e., %10001 would suffice).
>
> <u>PORTB</u>: Designated in PicBasic Pro code as PORTB.0 through PORTB.7 (8 pins: 6 through 13). For example, PORTB = %01010001 would set PORTB.0, PORTB.4, and PORTB.6 to 1. All other bits would be set to 0.

Each individual pin can be configured as an input or output independently (as described in the following Lab). When a pin is configured as an output, the output digital value (0 or 1) on the pin can be set with a simple assignment statement (e.g., PORTB.1 = 1). When a pin is configured as an input, the digital value on the pin (0 or 1) can be read by referencing the corresponding port bit directly (e.g., IF (PORTA.2 = 1) THEN ...).

9.3 An Example of PICBasic Pro Programming

PicBasic Pro is a compiler that uses a pseudocode approach to translate user friendly BASIC code into more cryptic assembly language code that is created in a separate *.asm file. The assembly code is then compiled into hexadecimal machine code (*.hex file), or hex code for short, that the PIC can interpret. The hex code file is then downloaded to the PIC and remains stored semi-permanently in EEPROM even when it is powered off. The code will remain in PIC memory until it is erased or overwritten using the Development Programmer.

For this laboratory you will program, compile, and test a very simple PicBasic example that controls the blinking of an LED. The code for this program, called blink.bas, is listed below, for both the standard PIC16F84 and a PIC with an internal oscillator (e.g., the PIC16F88). The hardware required is shown in Figure 9.1. Anytime a PIC with an internal oscillator or with a clock speed other than 4MHz is used, the extra OSC PICBasic code (shown in italics below) is required so all time-critical functions (e.g., Pause) will work properly. The A/D converter setting (ANSEL = 0) is required only for a PIC containing optional A/D converters that you wish to disable.

```
' blink.bas for the PIC16F84 with external oscillator
' Example program to blink an LED connected to PORTB.0 about once a second

myloop:
        High PORTB.0            ' turn on LED connected to PORTB.0
        Pause 500               ' delay for 0.5 seconds

        Low PORTB.0             ' turn off LED connected to PORTB.0
        Pause 500               ' delay for 0.5 seconds
Goto myloop                     ' go back to label "loop" repeatedly
End
```

```
' blink.bas for the PIC16F88 with an internal oscillator running at 8 MHz
' Example program to blink an LED connected to PORTB.0 about once a second

' Identify and set the internal oscillator clock speed (required for the PIC16F88)
DEFINE OSC 8
OSCCON.4 = 1
OSCCON.5 = 1
OSCCON.6 = 1

' Turn off the A/D converters (required for the PIC16F88, to use associated pins for digital I/O)
ANSEL = 0

myloop:
        High PORTB.0            ' turn on LED connected to PORTB.0
        Pause 500               ' delay for 0.5 seconds

        Low PORTB.0             ' turn off LED connected to PORTB.0
        Pause 500               ' delay for 0.5 seconds
Goto myloop                     ' go back to label "loop" repeatedly
End
```

NOTE - there is a code template file available on the Lab website that you can use as a starting point for all future labs and your project (if using the PIC16F88 and/or similar devices).

The blink.bas program turns a light emitting diode (LED) on for half a second, and then turns it off for half a second, repeating the sequence for as long as power is applied to the circuit. The first two optional lines in the program are comment lines that identify the program and its function. Comment lines must begin with an apostrophe. On any line, information on the right side of an apostrophe is treated as a comment and is ignored by the compiler. The label "loop" allows the program to return control to this line at a later time using the Goto command. "High PORTB.0" causes pin 6 (RB0) to go high which turns on the LED. The Pause command delays execution of the next line of code by a given number of milliseconds (in this case 500 corresponds to 500 milliseconds or 0.5 second). "Low PORTB.0" causes pin 6 (RB0) to go low which turns the LED off. The following Pause causes a 500 millisecond delay before executing the next line. The "Goto loop" statement returns control to the first executable program line labeled as "loop" to continue the process. The "End" statement on the last line of the program terminates execution. In this example, the loop continues until power is removed. Although the End statement is never reached in this example, it is good programming practice to end every program file with an End statement.

9.4 Procedure for Programming a PIC with the Microcode Studio IDE (Integrated Development Environment)

The Microcode Studio software is used to program PIC microcontrollers. It contains all the tools necessary to write and compile PicBasic code and to download the resulting hex file to the PIC. It also includes several debugging and simulation tools not used in this lab.

Programming a PIC always requires three sequential steps. 1) Write or edit PicBasic code. 2) Compile this code to hexadecimal (hex). 3) Download the hex code to the PIC EEPROM. The PIC is then able to execute the code until it is erased or programmed again. The details for using the software in the Lab follow.

PIC PROGRAMMING PROCEDURE:

1. Open MicroCode Studio

Double click on the MicroCode Studio desktop icon

or select from the *Start* menu:

Programs | *MicroCode Studio (MCSX)* | *MicroCode Studio (MCSX).*

2. Create or Open Your PicBasic Pro Program

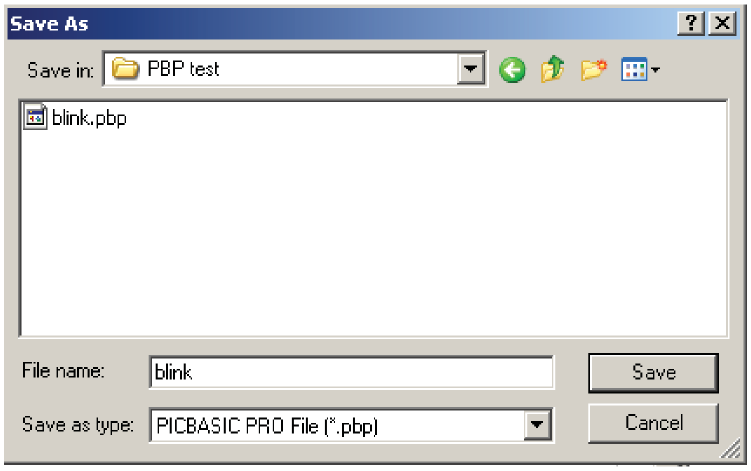
If you are starting a new project, either edit the file that comes up by default, **use the code template available on the Lab website**, or select *File* | *New* to start from scratch. We recommend always starting the code template when using the PIC16F88.

If you want to edit an existing project, select *File* | *Open* and browse to your code file. The file can be created initially in any text editor (e.g., Windows NotePad or Microsoft Word, saving the file as "Plain Text: *.txt").

Note - To disable Microcode Studio's command case changing, select *View* | *Editor Options ...*, click on the *Highlighter* tab, and under *Reserved word formatting*, select *Default*.

3. Save and Name Your Project File

Save the file to the folder where you want to store your project. Make sure you select the appropriate drive (e.g., your U-drive) in the *Save in* pull-down box. Use either PICBASIC PRO file (*.pbp) or BASIC file (*.bas) as the file type. **NOTE - Do not use periods in your file name.**

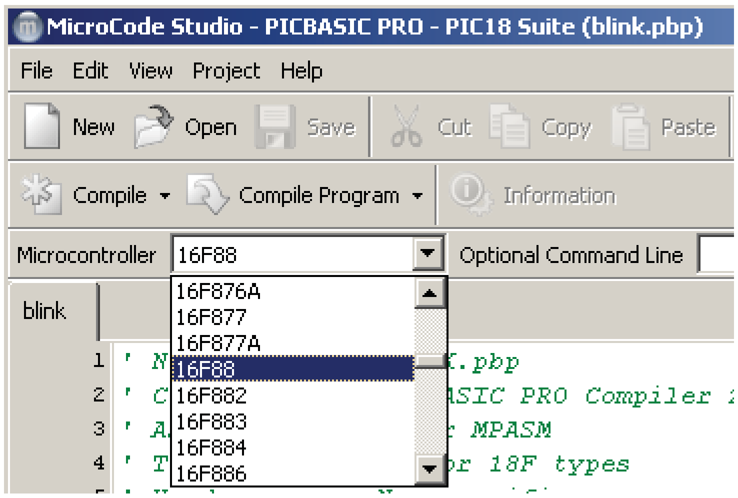


File name: blink

Save as type: PICBASIC PRO File (*.pbp)

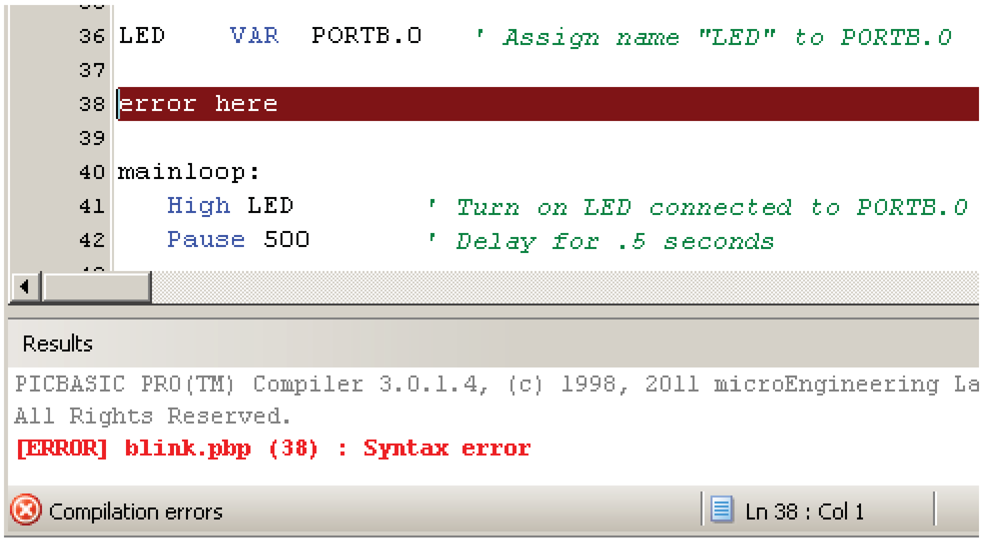
4. Choose the PIC Device You are Using

Select the appropriate PIC microcontroller (usually the 16F88) from the pull-down box in the *Microcontroller* (MCU) toolbar. MicroCode Studio and the U2 Programmer support only the devices listed.


MicroCode Studio - PICBASIC PRO - PIC18 Suite (blink.pbp)
File Edit View Project Help
New Open Save Cut Copy Paste
Compile Compile Program Information
Microcontroller 16F88
Optional Command Line
blink
16F876A
16F877
16F877A
16F88
16F882
16F883
16F884
16F886

5. Check For Errors

To make sure there are no errors in your code, click on the *Compile* button on the *Compile and Program Toolbar*. If there are any errors, MicroCode Studio will identify and locate them. Here's an example:



```
36 LED     VAR   PORTB.0    ' Assign name "LED" to PORTB.0
37
38 error here
39
40 mainloop:
41     High LED           ' Turn on LED connected to PORTB.0
42     Pause 500          ' Delay for .5 seconds
```

```
PICBASIC PRO(TM) Compiler 3.0.1.4, (c) 1998, 2011 microEngineering La
All Rights Reserved.
[ERROR] blink.pbp (38) : Syntax error
```

To have the line #'s appear in the editor window (if they aren't there already), select *View | Editor Options ...* and check the *Show line numbers in left gutter* box.

Correct any errors found in the code and *Compile* again until there are no more errors. After a successful compile, the status line at the bottom of the window will read "Success" and indicate how much memory your program is using on the PIC.

6. Prepare the PIC for Programming

Make sure the USB cable is plugged into the U2 Programmer. The green LED in the device should be on.

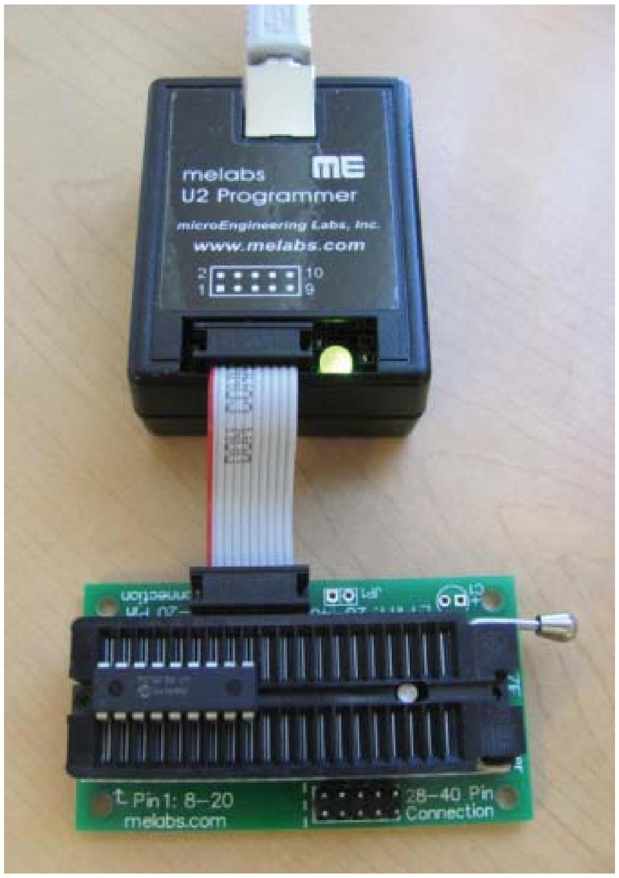
Make sure the metal lever on the U2 Programmer ZIF socket is in the up position.

NOTE - Always support the programmer socket with your spare hand while pivoting the lever up or down.

Insert your PIC into the socket with pin 1 in the position indicated on the socket board. **Make sure the PIC is in the correct orientation.**

NOTE - The "Pin 1" position is different depending on the # of pins on your PIC, as indicated on the green U2 socket board. The required ribbon cable connector position is also different.

Pivot the socket lever down to lock the PIC in place.

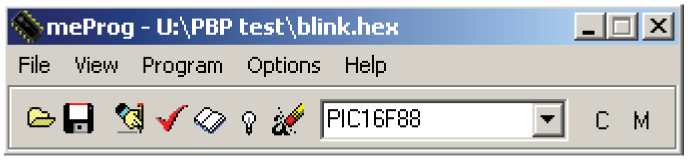


Make sure the PIC is positioned and oriented in the programmer properly before continuing.

7. Prepare the Code For Download Onto the PIC

Click on the *Compile Program* button to compile the code and generate the files needed for programming the PIC.

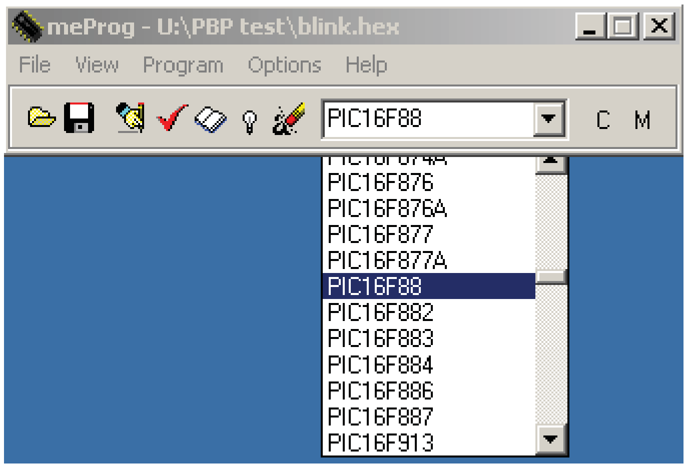
This will launch the meProg utility that allows you to store the code on the PIC. The following window will appear:


meProg - U:\PBP test\blink.hex
File View Program Options Help
PIC16F88
C M

NOTE - The window may take a while to appear, especially the first time you compile, while the software generates the files and searches for the U2 hardware, so be patient.

8. Identify the PIC Model Number

The PIC device number should transfer from Microcode Studio, but you should still verify this and change it if necessary in the meProg window pull-down list.



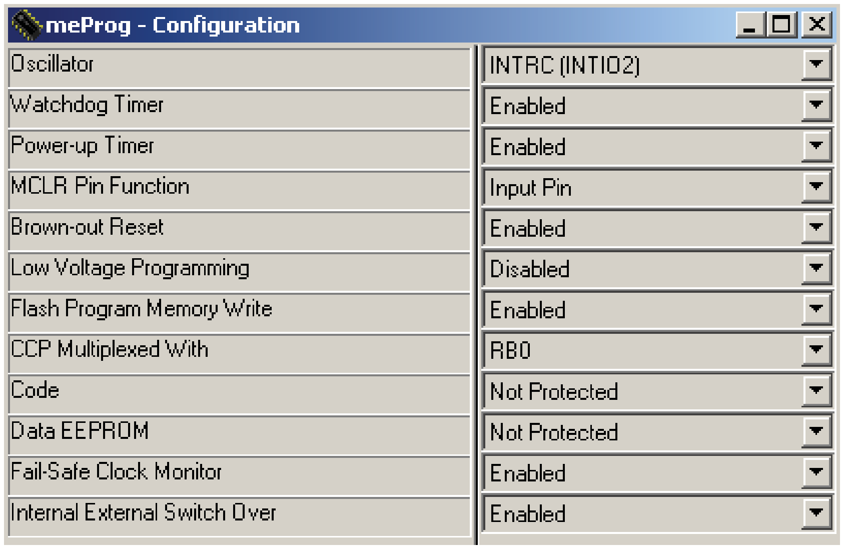
9. Select the Appropriate Configuration Bit Settings

Again in the meProg window, Select *View* | *Configuration* (or click on the "C" on the toolbar) to display the Configuration window (if it isn't visible already).

Click on the down-arrows to select the desired or appropriate choice for each feature listed.

NOTE - The configuration choices need to be set to the desired values every time you recompile your code, unless you define them in your code, as described in the next section.

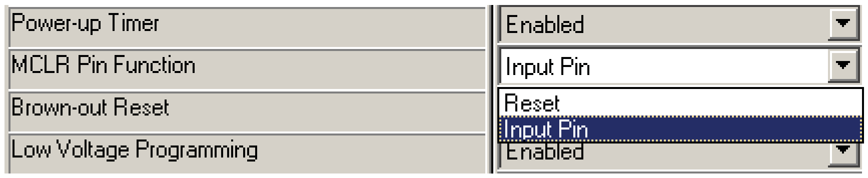
Typical choices (e.g., for the PIC16F88) are shown below.



meProg - Configuration

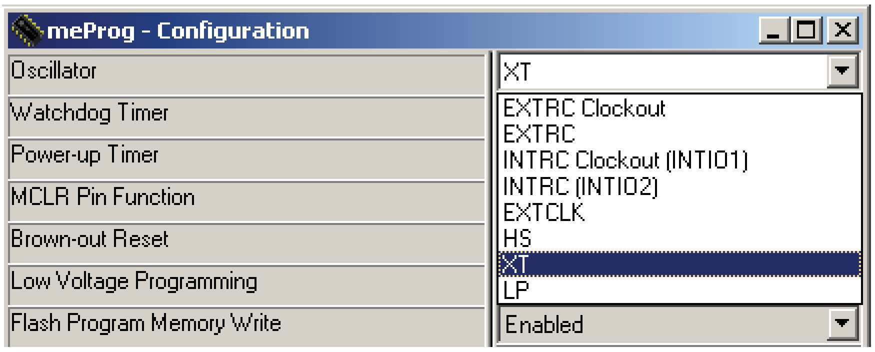
Oscillator	INTRC (INTIO2)
Watchdog Timer	Enabled
Power-up Timer	Enabled
MCLR Pin Function	Input Pin
Brown-out Reset	Enabled
Low Voltage Programming	Disabled
Flash Program Memory Write	Enabled
CCP Multiplexed With	RB0
Code	Not Protected
Data EEPROM	Not Protected
Fail-Safe Clock Monitor	Enabled
Internal External Switch Over	Enabled

With many PICs, some pins offer multiple functions, and you indicate the desired function with the configuration setting. For example, the MCLR pin can be used to activate a reset of the PIC, but it can also be used as an additional I/O pin (RA5):



Power-up Timer	Enabled
MCLR Pin Function	Input Pin
Brown-out Reset	
Low Voltage Programming	Enabled

And many PICs offer many options for the type of oscillator used. For example, if you wanted to use a more-accurate external crystal oscillator, or if you were using a PIC that did not have an internal oscillator, you would want to select the XT option:



meProg - Configuration

Oscillator	XT
Watchdog Timer	
Power-up Timer	
MCLR Pin Function	
Brown-out Reset	
Low Voltage Programming	
Flash Program Memory Write	Enabled

To learn about the different features and choices listed in the Configuration window, refer to appropriate sections in the datasheet for the specific PIC you are using.

NOTE - Depending on how multi-function pins are being used, bits in certain registers (e.g., OSCCON, ANSEL, and ADCON) must also be set in your code to have the functions operate as desired. For example, with the PIC16F88, to use PORTA pins for digital I/O, the ANSEL bits must be set to 0. See the relevant sections in the PIC datasheet for more information.

10. Changing Configuration Settings in Code

An alternative to setting the configuration bits manually, as described in the previous section, is to set them within your program.

You only need to add code for the settings for which the default values are different from what you want.

For example, you can automatically achieve the settings shown in the previous section for the PIC16F88, by adding the following code to your program (or by using the code template on the Lab book website that already contains the code):

```
#CONFIG
   __CONFIG _CONFIG1, _INTRC_IO & _PWRTE_ON & _MCLR_OFF & _LVP_OFF
#ENDCONFIG
```

Note that there are two underscores in front of the "CONFIG" and only one underscore in front of the "CONFIG1." There is also a comma between "CONFIG1" and the settings. All settings, including any that might be added, are separated by the bitwise AND operator (&).

The settings available for a given PIC can be found in the appropriate *.INFO file for the device. These files can be found in: *C:\PBP3\DEVICE_REFERENCE.*

11. Download Your Code Onto the PIC

After all of the configuration choices have been set to the desired values, click on the Program icon

or select *Program* from the *Program* menu in the meProg window.

The U2 programmer LED will glow red while the code is being downloaded, and it should glow green again when the process is completed.

After the program is written and verified, a *Program Verify complete* dialog box should appear, indicating that everything worked properly. Click on *OK.*

NOTE - Never insert or remove a PIC when the LED glows red. This can cause damage to the chip and/or the programmer.

12. Remove and Test the Programmed PIC

Lift the lever on the programmer to release the pin clamp. Then remove the PIC from the socket and insert it into your circuit for testing.

13. Shutdown the Software and Logoff

Close (Exit) the MicroCode Studio application. The programmer and configuration windows will close automatically with MicroCode Studio.

Be sure to log off your session on the PC so others won’t use (and/or abuse) your account.

9.5 Using Interrupts

An interrupt is a specially designated input to a microcontroller that changes the sequence of execution of a program. When there is a change of state of one or more of the input pins designated as interrupts, the program pauses normal execution and jumps to separate code called an interrupt service routine. When the service routine terminates, normal program execution resumes with the statement following the point where the interrupt occurred. The interrupt service routine is identified by the PICBasic ON INTERRUPT GOTO command. An example program called onint.bas follows:

```
' onint.bas
' Example use of an interrupt signal and interrupt handler
' This program turns on an LED and waits for an interrupt on PORTB.0.  When RB0 changes
'   state, the program turns the LED off for 0.5 seconds and then resumes normal execution.

' Identify and set the internal oscillator clock speed (required for the PIC16F88)
DEFINE OSC 8
OSCCON.4 = 1
OSCCON.5 = 1
OSCCON.6 = 1

' Turn off the A/D converter (required for the PIC16F88)
ANSEL = 0

led   var   PORTB.7          ' define variable led

     OPTION_REG = $FF        ' disable PORTB pull-ups and detect positive edges on interrupt
     On Interrupt Goto myint ' define interrupt service routine location
     INTCON = $90            ' enable interrupt on pin RB0

'  Turn LED on and keep it on until there is an interrupt
myloop:
     High led
     Goto myloop

' Interrupt handler
     Disable                 ' do not allow interrupts below this point
myint:
     Low led                 ' if we get here, turn LED off
     Pause   500             ' wait 0.5 seconds
     INTCON.1 = 0            ' clear interrupt flag
     Resume                  ' return to main program
     Enable                  ' allow interrupts again

End                          ' end of program
```

The onint.bas program turns on an LED using PORTB.7 until an external interrupt occurs. A switch or button connected to pin 6 (PORTB.0) provides the source for the interrupt signal.

When the signal transitions from low to high, the interrupt routine executes, causing the LED to turn off for half a second. Control then returns to the main loop causing the LED to turn back on again. More detail is provided in the following paragraphs.

NOTE: when using constants in a program, the dollar sign ($) prefix indicates a hexadecimal value percent sign (%) prefix indicates a binary value.

The first active line uses the keyword var to create the variable name led to denote the pin identifier PORTB.7. In the next line the OPTION_REG is set to $FF (or %11111111) to disable PORTB pull-ups and to configure the interrupt to be triggered when a positive edge occurs on pin RB0. When pull-ups are enabled, the PORTB inputs are held high until they are driven low by the external input circuit (e.g., a switch or button wired to pin RB0). The option register is defined in more detail below.

The label "myint" is defined as the location to which the program control jumps when an interrupt occurs. The value of the INTCON register is set to $90 (or %10010000) to properly enable interrupts. Setting the INTCON.7 bit to 1 globally allows all interrupts, and setting the INTCON.4 bit to 1 specifically enables the PORTB.0 interrupt. The INTCON register is described in more detail below.

The two lines starting with "loop" label cause the program to continually maintain the led pin (PORTB.7) high which keeps the LED on. The continuous cycle created by the "Goto loop" statement is called an infinite loop since it runs as long as no interrupt occurs. Note that an active statement (such as: "High led") MUST exist between the label and Goto of the loop for the interrupt to function because PICBASIC checks for interrupts only after a statement is completed.

The final section of the program contains the interrupt service routine. Disable must precede the label (myint:) and Enable must follow the Resume to prevent further interrupts from occurring until control is returned to the main program. The placement of these commands might seem awkward if you think about it, but this is the correct syntax. The interrupt routine executes when control of the program is directed to the beginning of this routine (labeled by "myint:") when an interrupt occurs on PORTB.0 (pin 6). At the identifier label "myint" the statement Low led sets PORTB.7 (pin 13) to a digital low turning off the LED in the circuit. The Pause statement causes a 500 milliseconds (half a second) delay, during which the LED remains off. The next line sets the INTCON.1 bit to zero to clear the interrupt flag. The interrupt flag was set internally to 1 when the interrupt signal was received on PORTB.0, and this bit must be reset to zero before exiting the interrupt routine. At the end of the myint routine, control returns back to the main program loop where the interrupt occurred.

9.5.1 Registers Related to Interrupts

In order to detect interrupts, two specific registers on the PIC must be initialized correctly. These are the option register (OPTION_REG) and the interrupt control register (INTCON). The function of the individual bits within both registers are defined below.

The definition for each bit in the first register (OPTION_REG) follows. Recall that the least significant bit (LSB) is on the right, and is designated as bit zero (b_0), while the most significant bit (MSB) is on the left, and is designated as bit 7 (b_7).

$$\text{OPTION\_REG} = \%b_7b_6b_5b_4b_3b_2b_1b_0$$

bit 7: RBPU: PORTB Pull-up Enable Bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled (by individual port latch values)
bit 6: Interrupt Edge Select Bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin
bit 5: T0CS: TMR0 Clock Source Select Bit
1 = Transition on RA4/TOCK1 pin
0 = Internal instruction cycle clock (CLKOUT)
bit 4: T0SE: TMR0 Source Edge Select Bit
1 = Increment on high-to-low transition on RA4/TOCK1 pin
0 = Increment on low-to-high transition on RA4/TOCK1 pin
bit 3: PSA: Prescaler Assignment Bit
1 = Prescaler assigned to the Watchdog timer (WDT)
0 = Prescaler assigned to TMR0
bits 2-0:PS2: PS0: Prescaler Rate Select Bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

In the onint.bas example above, OPTION_REG was set to $FF which is %11111111. Setting bit 7 high disables PORTB pull-ups and setting bit 6 high causes interrupts to occur on the positive edge of a signal on pin RB0. Bits 0 through 5 are only important when using special timers and are not used in this example.

The definition for each bit in the second register (INTCON) follows:

bit 7:	GIE: Global Interrupt Enable Bit
	1 = Enables all unmasked interrupts
	0 = Disables all interrupts
bit 6:	EEIE: EE Write Complete Interrupt Enable Bit
	1 = Enables the EE Write Complete interrupt
	0 = Disables the EE Write Complete interrupt
bit 5:	T0IE: TMR0 Overflow Interrupt Enable Bit
	1 = Enables the TMR0 interrupt
	0 = Disables the TMR0 interrupt
bit 4:	INTE: RB0/INT Interrupt Enable Bit
	1 = Enables the RB0/INT interrupt
	0 = Disables the RB0/INT interrupt
bit 3:	RBIE: RB Port Change Interrupt Enable Bit (for pins RB4 through RB7)
	1 = Enables the RB Port Change interrupt
	0 = Disables the RB Port Change interrupt
bit 2:	T0IF: TMR0 Overflow Interrupt Flag Bit
	1 = TMR0 has overflowed (must be cleared in software)
	0 = TMR0 did not overflow
bit 1:	INTF: RB0/INT Interrupt Flag Bit
	1 = The RB0/INT interrupt occurred
	0 = The RB0/INT interrupt did not occur
bit 0:	RBIF: RB Port Change Interrupt Flag Bit
	1 = When at least one of the RB7:RB4 pins changed state (must be cleared in software)
	0 = None of the RB7:RB4 pins have changed state

In the onint.bas example above, INTCON was set to $90 which is %10010000. For interrupts to be enabled, bit 7 must be set to 1. Bit 4 is set to 1 to check for interrupts on pin RB0. Bits 6, 5, 3, and 2 are for advanced features and are not used in this example. Bits 0 and 1 are used to indicated interrupt status during program execution.

If more than one interrupt signal were required, bit 3 would be set to 1 which would enable interrupts on pins RB4 through RB7. In that case, INTCON would be set to $88 (%10001000). To check for interrupts on RB0 and RB4-7, INTCON would be set to $98 (%10011000). PORTA has no interrupt capability, and PORTB has interrupt capability only on pins RB0 and pins RB4 through RB7.

NOTE - PicBasicPro does not handle interrupts very efficiently, and the code can be confusing. Hardware interrupts can be very effective when using Assembly language or C, but PicBasicPro software interrupts should usually be avoided. It is better to just use polling loops instead (see the next Lab for more info).

9.6 Procedure

(1) Use an ASCII editor (e.g., Windows Notepad or MS Word - Text Only) to create the program "blink.bas" listed in Section 9.3. Save the file in a folder in your network file space.

(2) Follow the procedure in Section 9.4 to compile the "blink.bas" program into hexadecimal machine code ("blink.hex") and to load this code onto a PIC.

(3) Assemble and test the circuit shown in Figure 9.1. When power is applied, the LED should immediately begin to blink on and off, cycling once each second.

(4) Repeat steps (1) through (2) for the "onint.bas" program listed in Section 9.5. **Be sure to create a new project and follow the entire procedure.** Before constructing the circuit for onint.bas, identify the additional components required in Figure 9.2. Indicate the necessary changes in the figure and check with your Teaching Assistant to verify that your changes are appropriate. The program should turn on an LED attached to PORTB.7. An interrupt on PORTB.0 should cause the LED to turn off for half a second, and then turn back on again. This should be signaled by an single pole, single throw (SPST) switch or a normally open (NO) button. You must make sure to wire the switch or button such that the ON state applies 5 Vdc to the pin, and the OFF state grounds the pin. The input should not be allowed to "float" in the OFF state (i.e., it must be grounded through a 1kΩ resistor). When you are sure you have all components wired properly, apply power to the circuit and test it to determine if it is working properly.

NOTE - When removing ICs from a breadboard, always use a "chip puller" tool to lift both ends together. Alternatively, use a small flat-head screwdriver to pry each end up a little at a time to release the IC without causing damage (e.g., bent or broken pins).

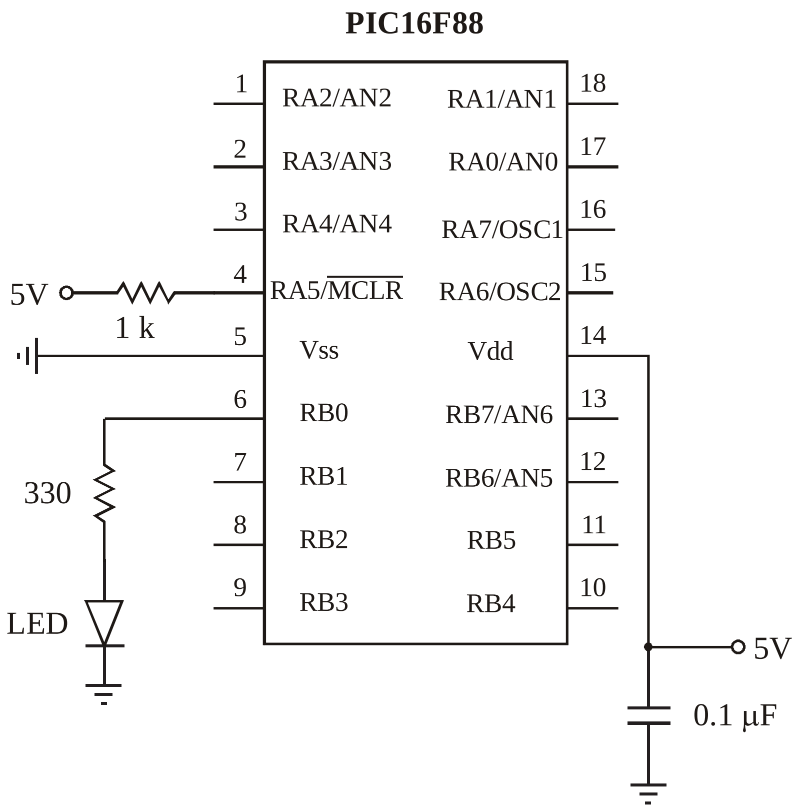


Figure 9.2 Circuit to be modified to run onint.bas

LAB 9 QUESTIONS

Group: ____ Names: ______________________ ______________________

______________________ ______________________

(1) Explain all differences between PORTA and PORTB if using the pins for inputs. Refer to Section 7.8 in the textbook for more information.

(2) For the onint.bas interrupt example, if the button is held down for more that 0.5 second and then released, is it possible that the LED would blink off again? If so, explain why. (Hint: consider switch bounce.)

(3) Show two different ways to simply and properly interface an LED to a PIC output pin. One circuit should light the LED only when the pin is high (this is called positive logic) and the other circuit should light the LED only when the pin is low (this is called negative logic).

(4) Explain what you would observe if power were applied to a PIC loaded with the following code if an LED is connected to RB0 as shown in Figure 9.1. Note - *Goto* is being used, not *Gosub*.

```
before: High PORTB.0
        Pause 500
        Low PORTB.0
        Goto during
        Pause 100
        High PORTB.0
        Goto after
during: Low PORTB.0
        Pause 300
        High PORTB.0
        Pause 400
after:  Pause 200
        Low PORTB.0
        End
```