

# Lesson 19

## In-Circuit Programming

### Overview

<b>Introduction</b>	<p>When the designer makes a new circuit, there is often some time spent in developing the software for that circuit. Removing the PIC from the circuit and inserting it into a programmer, then reversing the process for each change soon gets tedious. Programming the PIC while still in the target circuit certainly speeds up debugging (and reduces wear and tear on the PIC's pins.). However, allowance for in-circuit programming needs to be made when designing the target circuit.</p> <p>In this lesson, we will examine the requirements for designing circuits including PICs, and look at special considerations required if we wish to program the PIC in-circuit. We will use our demonstration circuit as an excuse to explore two peripherals lacking in the 16F84A, the analog input and the pulse width modulation output.</p>	
<b>In this section</b>	Following is a list of topics in this section:	
	<b>Description</b>	<b>See Page</b>
	PIC Support Circuitry	2
	ICSP Considerations	6
	Designing the test board	10
	Building the test board	12
	Testing the Board	14
	The A/D Converter	16
	Pulse Width Modulation	19
	Wrap Up	22

## PIC Support Circuitry

### Introduction

All microprocessors and microcontrollers need some form of support. As a minimum, they need to be provided power. Some require elaborate clock circuitry, bus controllers, power management controllers, and often other types of support.

The PIC is appealing for embedded applications partly because it requires very little support. The PIC has program and data memory on-chip, some models have an onboard clock. Depending on the part, the designer can get away with only power and ground, although most require some clock support, and most also require some reset circuitry.

Up until now, we have run our 16F84A on the PIC-EL board. In this lesson we examine the requirements to make a PIC operate on its own stand-alone board.

### Power

Obviously, the PIC requires power. What can be a bit of a surprise is how tolerant the PIC is of the power it gets.

Nominally, all the 16F PICs are specified to run at 5 volts. All will run from 4.5 to 5.5 volts. All PICs have an absolute maximum voltage they can tolerate without damage, although they may not be specified to run at that voltage. On the 16F84A, the maximum voltage on the Vdd pin is specified at 7.5 volts.

PICs have a wide range of low voltages, however. Most PICs can run at somewhat lower voltages if the clock frequency is reduced. Newer 'LF' PICs can run down to two volts. This can make these parts attractive for battery powered projects. The PIC16LF88, for example, can operate at 10 MHz at 3 volts, and 4 MHz at 2 volts.

The current demanded by the PIC is dependent on a number of factors. The current required by the PIC itself is dependent on the clock frequency, the supply voltage, and the type of oscillator used. The 16F84A, for example, takes typically 1.8 mA with a 4MHz XT oscillator and Vdd = +5.5V. With a 20MHz HS oscillator the part requires 10 mA. A 16LF88 at 32kHz and 2 volts sips a mere 15 microamps.

In addition to the PIC itself, the Vdd pin must supply the current required by any outputs. Different PICs are capable of sourcing different amounts of current, so it is important to review the "Electrical Characteristics" portion of the datasheet.

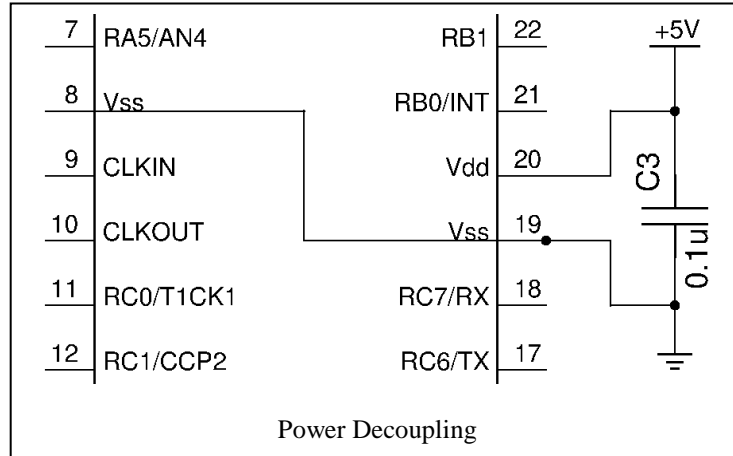
The datasheet shows the current available on each pin, each port, and total for the part. Sometimes there are unique characteristics to particular pins. On the PIC16F84A, a single pin can source 25 mA. PORTB can source 100 mA, and the entire PIC can only source 100 mA. If your circuit demands I/O that needs significant current, it is important to review the datasheet before designing your circuit.

*Continued on next page*

## PIC Support Circuitry, Continued

### Power Decoupling

If you ask professional PIC consultants, they will tell you that you must place a 0.1 uF capacitor at each power pin (some PICs have more than one power pin).



If the power supply should pick up noise, either from relays or motors, or from significant current changes in the supply, the effect on the PIC can be unpredictable.

In hobbyist practice, relays are relatively unusual, and rarely do we place large current demands on the PIC. As a result, the decoupling capacitor can often be omitted without ill effect. However, if it is needed, the symptoms can be very hard to diagnose. The experimenter would be well advised to simply keep a supply of 0.1's around and use them liberally.

### Reset Circuitry

The PIC has a reset pin named MCLR which can reset the processor if brought to ground. For normal operation, the pin needs to be near Vdd (typically 5 volts).

The external circuitry must be given time to stabilize before the program starts. Some processors require elaborate reset circuitry to ensure that the part doesn't start until the circuit is stable.

The PIC will not start for a few clock cycles after power is applied. The configuration flag `_PWRTE_ON` allows the designer to insert another 72 ms. of startup delay. As a result, the PIC reset circuitry can consist of nothing more than a pullup resistor to Vdd. Since MCLR takes very little current, a high value pullup is adequate. For the same reason, there is no penalty for a low value pullup. In other words, almost anything will work, from a meg to a wire.

Some designers also like to decouple MCLR. Some circuits may require more time, and designers might like an RC circuit on MCLR. In these cases, the value of the pullup will affect the RC delay.

*Continued on next page*

## PIC Support Circuitry, Continued

### Reset Circuitry (continued)

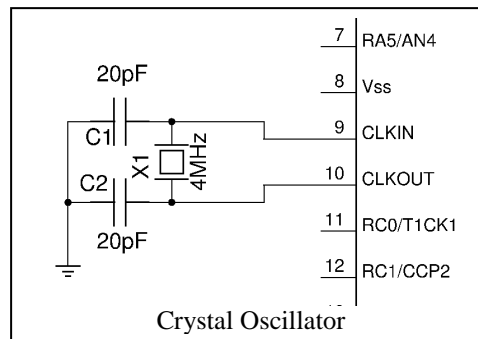
Most 16F PICs have a PGM pin, normally RB3 or RB4. This pin enables low voltage programming. The 16F84 does not have this pin, but most others do. For example the 16F628, which is often the hobbyist's next step up from the 16F84, does have a PGM pin. On PICs with PGM, it is important that this pin not be raised to a high logic level before the MCLR pin becomes true (+5 volts). The designer needs to take care that this pin will be at a low logic level during the startup of the circuit.

Typically, a high-value pulldown is adequate to keep PGM low. How high, of course, depends on your external circuitry, but the PGM pin as an input draws only microamps, so values on the order of tens to hundreds of K are usually adequate.

### Clock Circuitry

Some of the newer PICs have an onboard clock. For these, no external clock circuitry will be needed. All PICs can accept an external oscillator, but this is generally more complex than is needed. Most designs will use a crystal, a ceramic resonator, or a simple RC.

It seems that hobbyists generally prefer to use quartz crystals. Microprocessor crystals are inexpensive and if one is only going to make a single copy of the circuit, the cost of the supporting capacitors is generally insignificant.



If the capacitors are omitted, the crystal will oscillate at a slightly different frequency than its marked frequency, but generally it will oscillate. For applications where the clock frequency can be critical, such as a frequency counter or time of day clock, the designer may choose to replace one of the capacitors with a trimmer.

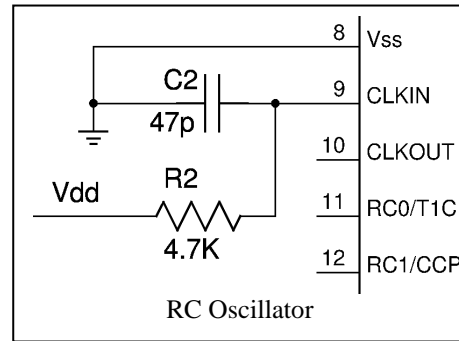
Ceramic resonators may also be used, and generally do not require the capacitors. Resonators are generally less expensive than crystals, and are not nearly as fragile. As a result, most professional designers prefer resonators to crystals. However, they are not quite as accurate.

*Continued on next page*

## PIC Support Circuitry, Continued

### Clock Circuitry (continued)

For most hobbyist applications, the processor frequency is not at all critical. For these applications a simple RC circuit can be used to set the frequency.



The RC oscillator frequency is much more temperature dependent than the crystal, and is also dependent on the supply voltage. Nonetheless, it is a cheap and simple way to provide a clock signal to the PIC. The RC oscillator, however, takes a little more current than the crystal.

The required resistor and capacitor values must be selected based on graphs in the datasheet. RC oscillators may be used for clocks from a few kHz up to about 2 MHz. The values shown will result in a clock of a little over 1 MHz.

### Summary

As you can see, the PIC allows a lot of flexibility in the way it is implemented. Perhaps less obvious, much of this is terribly non-critical. The fact that “almost anything” will probably work has probably contributed to the PIC’s popularity among hobbyists.

## ICSP Considerations

<b>Introduction</b>	<p>As mentioned earlier, in-circuit serial programming allows us to test new software in our circuit without the hassle of inserting and removing the PIC. It also makes an expensive ZIF socket unnecessary. Indeed, our programmer really needs no socket, and some popular programmers intended for in-circuit programming do not provide them.</p>
<b>How is the PIC programmed</b>	<p>All PICs can be programmed in an electrically similar fashion (the actual commands sent to different parts can be quite different). The MCLR pin may actually have three states; when MCLR is near <math>V_{ss}</math> (ground) the PIC is held in a reset state, when MCLR is near <math>V_{dd}</math> (+5) the PIC runs its program, and when MCLR is at <math>V_{pp}</math> (usually around +13), the PIC is placed in programming mode.</p> <p>Some PICs can also be placed into programming mode by raising the PGM pin to a logic high before MCLR is raised. The 16F84 does not have this capability, and the very newest PICs also lack this capability, but since many 16F (and 18F) parts have it, handling the PGM pin properly should be addressed to avoid going into program mode accidentally.</p> <p>After the PIC has been placed in program mode, programming instructions are fed into the PGD pin and clocked with the PGC pin. PGC/PGD are usually shared with RB6/RB7.</p>
<b>Handling PGC/PGD</b>	<p>Generally, the designer is not interested in giving up pins, especially on the smaller parts, so the PGC/PGD pins need to be available for some other use.</p> <p>When the PIC is placed in reset mode, all I/O pins are set as inputs and thus present a high impedance. If MCLR reaches <math>V_{pp}</math> quickly enough, the PIC program does not start, and thus PGC/PGD cannot be placed in output mode. As long as the external circuitry can be driven by the programmer these pins can be used by the application and still be available for programming.</p> <p>The critical consideration is that the application impose a small enough load that the programmer can overcome the external circuitry. This depends on the particular programmer. The PIC-EL can provide fairly substantial drive, so as long as the designer takes reasonable care to keep the capacitance down on PGC/PGD, most any circuitry can be used on those pins.</p> <p>Some PICs provide the capability to interface with an external debugger. However, the debugger interface requires that PGD/PGC be dedicated and cannot be used for any other purpose. So the designer planning to use a debugger must avoid the use of these pins for the application.</p>

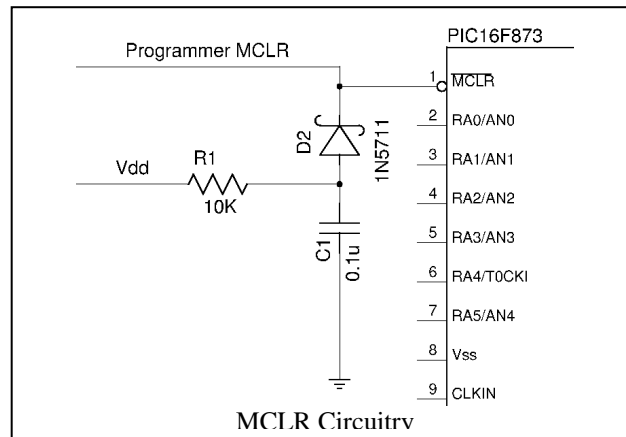
*Continued on next page*

## ICSP Considerations, Continued

### Handling MCLR

While the PGC, PGD, and PGM pins are quite straightforward, MCLR presents some new challenges. If we were not planning in-circuit programming, we could simply pull MCLR up to Vdd (+5). However, the programmer is going to need to bring MCLR up to Vpp, which is typically around 13 volts. A simple resistor could allow this 13 volts to find its way to the rest of the circuit, which could potentially damage the PIC or other components. In a noisy environment, we may well want some capacitance on MCLR, but such capacitance could prevent the programmer from raising MCLR quickly enough.

To handle this, Microchip recommends the following circuit:



The diode prevents 13 volts from getting back into the Vdd bus, and at the same time, “hides” the capacitor from the programmer. Microchip recommends a Schottky diode such as a 1N5711 for its speed. However, with a programmer that can provide substantial drive, and short cables, a normal 1N4148 should work here.

On the PIC-EL, this problem is solved by a switch which removes the pullup resistor during programming. A switch on the target circuit is certainly an option, but dealing with this automatically is more convenient and less expensive.

Values for the resistor and capacitor depend on the time constant the designer wants on reset, the potential noise level, and the amount of drive available from the programmer.

Once again, in “experimental” practice, we don’t always need everything to be perfect. MCLR draws very little current, so a pullup on the order of 100K or so is adequate. If the remainder of the circuit can be counted on to draw enough current, we can be guaranteed of sufficient voltage drop across R1 that the diode is really unnecessary. If our circuit was *only* to be powered from the PIC-EL, the MCLR circuitry could be eliminated altogether, and we could rely on the PIC-EL switch to manage MCLR.

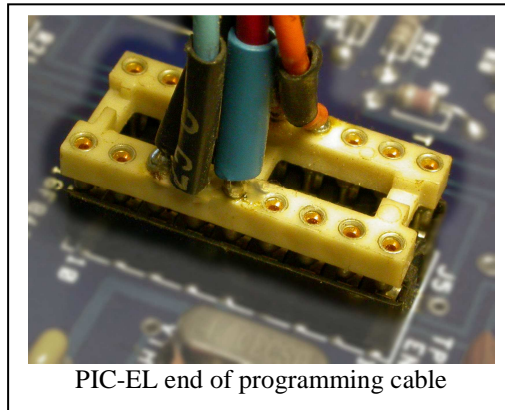
*Continued on next page*

## ICSP Considerations, Continued

### Connecting the programmer

The final detail that must be handled is providing a means to connect the programmer. On the PIC-EL, there is a “programming connector”, but on this connector, the MCLR signal must be either at Vss or Vpp. There is no provision for holding MCLR at Vdd, so in order to test our application, we need to disconnect the PIC-EL, and provide power to our circuit. It would be more convenient to connect directly to the PIC socket on the PIC-EL. This would allow use of the PIC-EL “programming” switch, and we could then test our application much as we test PIC-EL applications, by simply throwing the switch.

If we are making up a cable for programming, the PIC-EL end can be handled by using an 18 pin machine pin socket. The cable should be fairly short; Microchip recommends no more than 11 inches, but with the heavy drive available on the PIC-EL it is likely that longer cables will work.



For the target device end, we have some flexibility. We need at least 5 conductors; Power, Ground, PGC, PGD and Vpp. We would like the connector to be small. We would also like it to be inexpensive, and provide protection against inserting the connector backwards (we will likely plug and unplug many times as we experiment).

Your author favors clipping 6 pins from a long SIP connector. One socket can be filled with solder and the corresponding pin clipped to provide polarity protection. Although long SIP connectors are not terribly inexpensive, they can serve a variety of purposes, and the same part serves as both the cable and board ends.

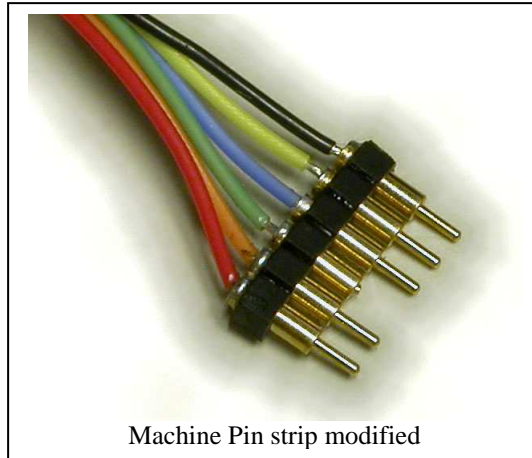
The downside of this approach is that the cable is not held very firmly, which can be quite annoying when testing, and the socket seems to wear relatively quickly, resulting in a looser fit after just a few insertions.

*Continued on next page*

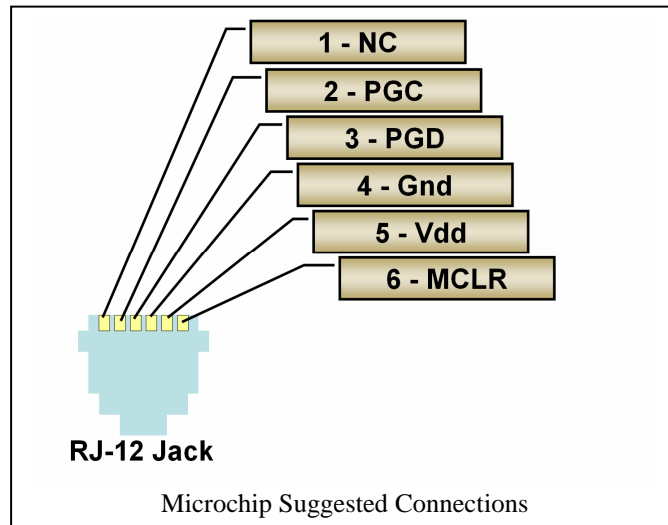


## ICSP Considerations, Continued

### Connecting the programmer (continued)



Microchip recommends using an RJ12 connector. This has the advantage of a quick and firm fit, and is widely available. RJ12 cables can be purchased at the local discount department store. Jacks are not quite as readily available, but they are available from all the electronics parts houses.



The main downside of the RJ12 is that the jacks are not compatible with the common 0.1" perfboard often used for prototyping. This leaves the experimenter with the choice of etching some sort of adapter board, or using a pigtail to connect the jack to the board.

## Designing the test board

<b>Introduction</b>	For our experiments, we want to be able to program a PIC on a separate board with the PIC-EL, and use the board to demonstrate the analog input and pulse width modulation output of some PIC models.
<b>PIC Selection</b>	The PIC choice is constrained by the parts available through FPP. The 16F872, 16F873, 16F874, 16F876, and 16F877 all fit the basic requirements. In a future experiment, we would like to examine program memory paging, which means we need to choose a PIC with more than 2K of memory, eliminating the 16F872 (although for <i>this</i> lesson, that part is adequate). The 873, 4, 6, and 7 are all part of the same family, and are very close. The 873 and 876 are 28 pin parts, while the 874 and 877 are 40 pin parts. The 873 and 874 have 4K of program memory, while the 876 and 877 have 8K. The same code will run on these four parts with the only change being the processor include file. The 872 requires a different linker script.
<b>Analog Input</b>	For testing the analog input, we will provide a variable voltage on AN0 (pin 2 on the 28 pin parts). This can be provided by a simple pot acting as a voltage divider between +5 and ground, with the wiper connected to AN0. Since we are using it as a voltage divider, and the analog input draws little current, almost any value pot will work. A value for the pot lower than about 1K will draw more current than necessary. A value much higher than 100K might be noisy.
<b>PWM Output</b>	We can gauge a change in the pulse width modulation duty cycle by the change in brightness of an LED. By connecting an LED to CCP1 (pin 13 on the 28 pin parts) through an appropriate value resistor to +5, we can use the PWM output to vary the brightness of the LED. The resistor will need to be sized for the LED; a value between 300 and 500 ohms will work with most LEDs.
<b>Handling MCLR</b>	<p>We saw that Microchip recommends a resistor, capacitor, and Schottky diode. With the substantial drive of the PIC-EL, an ordinary 1N4148 will probably work as long as the programming cable is short.</p> <p>The pot on AN0 will draw some current. If we arrange the value of the pot to be small enough compared to the value of the pullup on MCLR, we can avoid the diode altogether, since we can count on the drop through the pullup preventing +13 from getting back into the +5 supply. For example, if we have a 100K pullup and a 10K pot, we won't be able to drive the +5 bus above the 7.5 volt rating of the PIC.</p> <p>If we <u>only</u> want to power our circuit from the PIC-EL, we can avoid the pullup entirely and just connect MCLR straight through to the PIC-EL.</p>
<b>Clock Circuitry</b>	We saw that we have a number of options for controlling the microcontroller frequency. For our experiments, any design that results in a clock between about 500 kHz and 20 MHz will be adequate.



## Building the test board

### Introduction

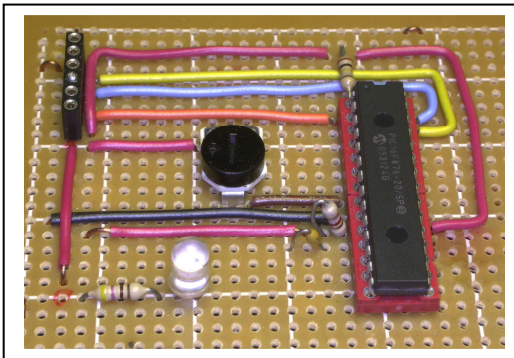
For reference purposes, we will use the schematic on the previous page. This is almost as simple as it can get. The resistor on MCLR could be omitted, in which case the board could only be run from the PIC-EL. With the resistor, we can apply 5 volts to the programming connector and run the board independent of the PIC-EL.

Many substitutions are possible, as discussed earlier. If the experimenter would like to use a higher value potentiometer, then the diode circuitry shown on page 7 should be used in place of the resistor.

A crystal or ceramic resonator may be used in place of the RC circuit for the clock if the experimenter would prefer more precise frequency control.

### Physical Construction

It is recommended that the circuit be built up on a perfboard. The student should choose a larger board than necessary. Melting solder on a perfboard is a little like eating potato chips; it's kind of hard to stop. There are plenty of PIC pins left over for additional experiments. A Radio Shack 276-147 is widely available and allows plenty of room for future experiments.



In laying out the parts, the experimenter should consider future possibilities. Leave room to reach unused PIC pins, but keep much of the board open.

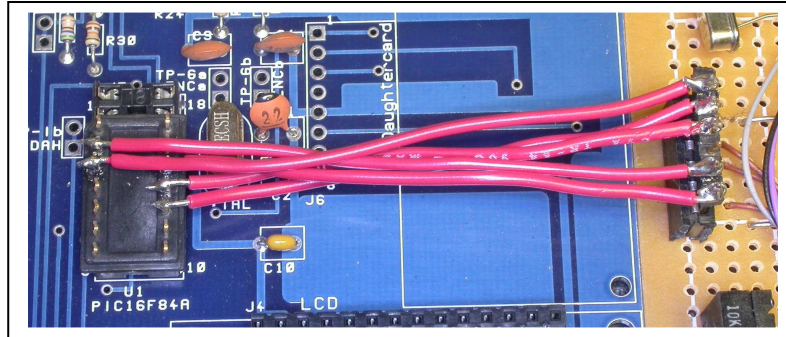
Wiring is non-critical. This is not RF so square corners are just fine. Wire wrap wire is often convenient for this sort of construction, but keep power and ground connections a little larger.

*Continued on next page*

## Building the test board, Continued

### Connecting to the PIC-EL

The PIC-EL cable should use a machine-pin socket or 18 pin DIP header on the PIC-EL end, and a connector of the experimenter's choice on the test board end. The cable should be short and relatively low capacitance. The figure below shows a minimum length cable, but a cable around 10 or 12 inches will work just fine.



### Going Forward

The PIC16F87x datasheet (30292c) is an important tool when doing the construction. When we move ahead to programming, it becomes vital. Although the various PICs share an instruction set, and even the pinouts of PICs with the same number of pins tend to be the same, each PIC has a different complement of I/O devices. The developer needs to know what registers control what devices, and what the various bits of each register mean.

## Testing the Board

### Introduction

Once the board is constructed, check that the PIC Vdd pin connects to the PIC-EL Vdd (+5), and that both PIC Vss (ground) pins connect to the PIC-EL Vss. Review the board for obvious shorts.

Having satisfied ourselves that we have caught the obvious, we need to load a program into the PIC to ensure that we can program the part. The obvious thing to do is the traditional flash a LED program.

### The test program

For our first test, we will do the classic flash an LED. Pretty boring, but a quick and simple way to see that we can program the 16F873.

Create a new project, but remember this time to go to Configure->Select Device and select the 16F873. Copy 16f873.lkr to your project directory for easy access and add it to the project. Our source, L19a.asm, will of course include p16f873.inc instead of p16f84a.inc, but the `__config` line will also be a bit different:

```
__config  _RC_OSC&_WDT_OFF&_PWRTE_ON&_BODEN_OFF&_LVP_OFF&_DEBUG_OFF
```

The 873 has a number of features that the 84A doesn't. The `_BODEN_OFF` turns off brownout detection. Brownout detection causes an interrupt when the voltage gets low, and since we will not be handling that interrupt, we need to turn it off.

`_LVP_OFF` turns off low voltage programming. We will be using the normal programming, so we don't want low voltage programming enabled.

Since we are using an RC oscillator instead of a crystal, we must specify `_RC_OSC` instead of `_XT_OSC`. A designer choosing a different oscillator must use the appropriate setting for the oscillator fuse.

Finally, `_DEBUG_OFF` turns off the debugging interface. Since we don't have an in-circuit debugger, we don't want the part to go into debug mode.

### Blinking the LED

We won't belabor the details of flashing an LED, that's been done a thousand times before. However, it is worth pointing out a few details in the example program.

First, the DELAY value was chosen to produce a reasonable flashing on clocks from 1 MHz to 20 MHz. At 20 the flashing will be fast, and at 1 somewhat lethargic. Decreasing DELAY lengthens the flashing cycle.

Notice that we put our code segment in PROG1:

```
STARTUP      code
             goto      Start
PROG1        code
             Start
```

This is necessary to avoid using the high program memory of the 873. We will talk about something called "paging" in later lessons. For now, controlling the location of our code allows us to avoid that detail. The linker script file shows us that the segment named PROG1 lives at address 0x5.

*Continued on next page*

## Testing the Board, Continued

### Blinking the LED (continued)

The LED is wired to PORT C on the PIC. The 84A doesn't have a PORTC, but for this example, it is just like PORT B.

Finally, we seem to have taken some extra steps in actually blinking the LED:

```

Loop
    btfss    LEDBIT    ; Is LED bit zero?
    goto    SetHi     ; No, go set it
    bcf     LEDBIT    ; Yes, clear it
    goto    SetLED    ; skip over set
SetHi
    bsf     LEDBIT    ; Set the LED bit
SetLED
    movwf  LEDstate,W ; Pick up the LED state
    movwf  PORTC     ; and send it to PORTC
    call   Snore     ; Wait a long while
    goto   Loop      ; Go do it again

```

We maintain a copy of the desired value for PORT C in `LEDstate`, and write it to PORTC instead of using `bcf/bsf` instructions directly on PORT C. It turns out that sometimes it is a problem to use bit set and clear instructions directly on a port. In this particular case, it happens it really doesn't matter, but it is generally considered good form to avoid read-modify-write instructions directly on a port. In those (relatively rare) cases where it does matter, the results can be very confusing, so the use of a *shadow register* (`LEDstate`) is preferred.

Using an XOR rather than the BCF/BSF could have saved a couple of instructions, but in this case, your author wanted to emphasize the shadow register. An XOR to memory is also a read-modify-write instruction, so there is still a need for the shadow register.

We have chosen to use the default linker script provided by Microchip. In a very simple case like this, we can sometimes use the template script. However, the developer should not feel that editing the script is something to be avoided. In most cases, tailoring the script for the application can make the application easier to understand and maintain.

### Programming the PIC

With the auxiliary board connected to the PIC-EL's PIC socket, programming the remote part is just like programming the 16F84A on the PIC-EL. You must remember to select the 16F873 on FPP, but otherwise, throw the programming switch, tell FPP to program the part, and the LED should flash when the PIC-EL's programming switch is returned to the run position.

### What if it doesn't work

Debugging a PIC circuit is no different than any other circuit. Check for bridges and cold solder joints. Remove the PIC, and check voltages. With the PIC-EL switch in the run position, approximately 5 volts should appear at the Vdd and MCLR pins of the 873. In the program position, the MCLR pin should drop to around zero. Each of the 5 connected pins on the PIC-EL socket should connect to the pins of the same name on the 873 socket.

## The A/D Converter

### Introduction

Now that we know that we can program the 873, it's time to move on to something more interesting. Since the 84A doesn't have analog inputs, the analog input on the 873 seems like a fun thing to try.

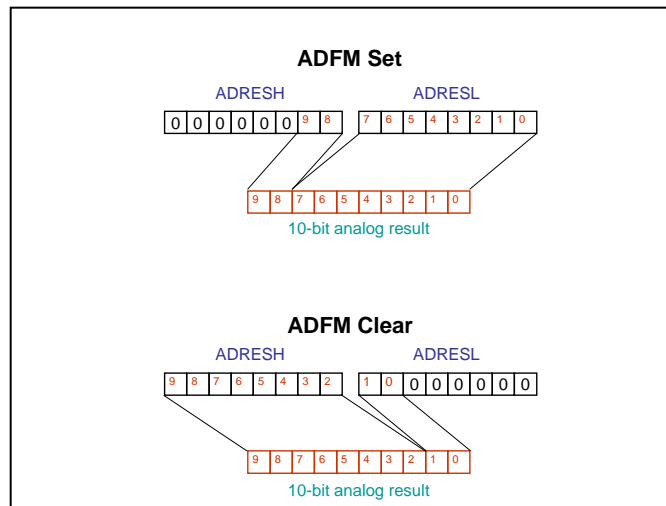
Unfortunately, the only output device we have on our test board is a single LED. So for our initial experiment, we will simply see if the voltage exceeds some threshold, and turn on the LED if it does. In our third experiment we will do something a little more interesting.

The student may wish to review section 11 of the datasheet.

### Configuring the A/D

To configure the analog to digital converter, there are two steps we need to take. The first is dealing with ADCON1.

ADCON1 has two parts. The highest bit, ADFM, controls whether we want the results left or right justified. The A/D produces a 10-bit result. The result is placed in two registers, ADRESH and ADRESL. Since there are 16 bits in the two registers, that leaves 6 bits left over, which will be cleared. If we set ADFM to 1, the high order six bits of ADRESH will be set to zero. If we clear ADFM to 0, the low order six bits of ADRESL will be set to zero.



For our purposes, eight bits of resolution is enough, so we will clear ADFM and simply ignore the low two bits in ADRESL.

The low order four bits of ADCON1 determine how we will assign the various bits of the RA register. The RA bits might be the digital inputs and outputs we have gotten used to on the 16F84. But they may also be assigned to analog inputs, and some of them might be used for analog reference voltages. See the table 11-2 in the datasheet

*Continued on next page*



## The A/D Converter, Continued

### Configuring the A/D (continued)

to view the various options. We only have RA0 connected, so at first glance, we might consider any setting for which RA0 is an analog input. However, if we have an unconnected pin which *could* be an analog, setting it to be a digital could cause excessive current to be drawn, potentially damaging the chip. We should set all the unused analog pins to be analog, so PCFG0 to PCFG3 should all be clear. (Since the 16F873 doesn't have a PORTE, setting PCFG1 wouldn't hurt anything, but for a 40 pin part it would matter). Since we are not providing voltage references but instead using the PIC's supply voltage, we don't want any of the configurations that provide for reference voltages.

```

; Set the A/D to left justified, use Vdd, Vss as refs
errorlevel    -302
banksel       ADCON1           ; Set all to analog since
clrf          ADCON1           ; AN1-4 all disconnected

```

The next thing we need to do is set ADCON0. If we were using multiple channels on the A/D, we would do this in our main program loop. However, since we are only using one channel, it makes sense to set up the A/D for that channel. Then the only thing we need to do in our main program is to signal the start of the conversion.

The high two bits of ADCON0 set the clock for the A/D converter. For simplicity, we will use the internal RC clock. The constant ADCOSC sets these two bits appropriately. Then, we need to select the A/D channel. We are using channel 0, and the constant CHANNEL0 sets these bits. Finally, we need to turn on the A/D converter. ADCON does this.

```

banksel       ADCON0
movlw         ADCOSC | CHANNEL0 | ADCON
movwf        ADCON0

```

That is all the initialization we need to do for the A/D, but don't forget we need to set PORTC bit 2 as an output to control our LED.

### Using the A/D Converter

Now that the A/D converter is turned on and ready, there are a few steps to be able to use it. The A/D works by charging a capacitor, then reading the charge on the capacitor. So, the steps are:

- Wait for the capacitor to charge
- Start the conversion
- Wait for the conversion to complete
- Read the result

The time it takes for the capacitor to charge is dependent on the external circuitry. There are equations in the datasheet to figure that out. In our case, the voltage will be changing fairly slowly, and we aren't in a hurry, so we can wait a (relatively) long time and call it good. The conversion is begun by setting the GO bit in ADCON0. When the conversion is complete, the GO bit will be cleared by the hardware. So our code looks like:

*Continued on next page*

## The A/D Converter, Continued

### Using the A/D Converter (continued)

```

; Hang around a while to charge the cap
    decfsz    c1,F          ; This is a longer wait
    goto     Loop         ; than needed but easy
; Start the conversion
    bsf      ADCON0,GO     ; GO=1 starts conversion
; Wait for conversion to complete
Conv
    btfsc    ADCON0,GO     ; Hardware clears GO
    goto     Conv         ; when A/D complete
; Pick up the value
    movf     ADRESH,W      ; Only using 8 MSBs
    movwf    ADCval        ; of analog value

```

Comparing the result to some value isn't anything special, nor is sending the result to the LED:

```

    clrf     LEDstate      ; Initially set to on
    movlw   H'80'         ; Compare ADC value
    subwf   ADCval,W      ; to midscale
    btfss   STATUS,C      ; Greater?
    bsf     LEDstate,LED  ; No, turn off LED
    movf    LEDstate,W    ; Pick up LED state
    movwf   PORTC         ; and send to LED

```

### Running the program

After assembling and downloading the program, you should be able to turn the LED on and off by turning the pot. Changing the value h'80' should move the position on the pot where the LED goes on and off.

The A/D converter, with its 10-bit output, produces a value between 0 and 1023 (decimal). The 0 result is generated when the input is at the negative reference voltage, in this case, 0 volts, and the 1023 when the input is at V<sub>dd</sub> (5 volts). The range could have been changed by providing reference voltages. The designer could, for example, have a 0 result represent 1 volt, and 1023 represent 4 volts, by providing 1 and 4 volt reference supplies.

## Pulse Width Modulation

### Introduction

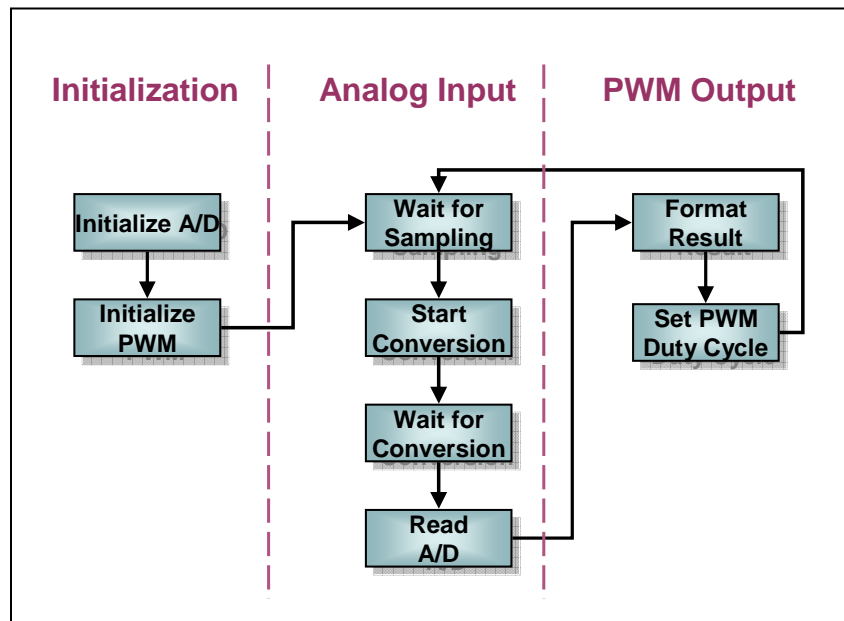
Another peripheral found on most PICs is a pulse width modulation output. Certainly we could do pulse width modulation on a F84 by toggling an output in a loop, but by providing a peripheral, many PICs make this a low overhead operation.

Pulse width modulation can be useful in a number of applications. Most commonly PWM is used for motor control, but in hobbyist applications, often the result is filtered to provide an analog output. The PIC PWM peripheral can generate frequencies over 200 kHz, making it easy to filter. PWM is also an effective way to adjust the brightness of an LED, which is what we will do in our example.

The PIC16F873 has two PWM modules. A few of the 16F parts have three. Some of the 16 and 24-bit core parts have as many as eight.

### Overall Program Flow

Since there are only limited peripherals on the little test board, the choices for testing are rather limited. The A/D converter could be used to provide an input, which could then be used to adjust the PWM output. The PWM from the CCP1 port will affect the brightness of the LED, so the feedback from the pot will be immediate. This leads to a program flow like the following:



### Setting the frequency

Before using the PWM output, it is necessary to set the PWM period (the frequency is 1/period). The PWM module uses timer 2 as it's time base, so the period is dependent on the oscillator frequency, and the prescale value set for TMR2. The PWM period restarts whenever the TMR2 value reaches the value stored in the PR2 register, so the PWM period is:

$$[(PR2)+1] * 4 * T_{osc} * (TMR2 \text{ prescale value})$$

*Continued on next page*

## Pulse Width Modulation, Continued

### Setting the frequency (continued)

The TMR2 prescale can have a value up to 16. PR2 can be loaded with a value as high as 255 (H'FF'). Using the values for RC shown on the schematic gives a clock of about 1.4 MHz, so that leads to a maximum period of about 11 milliseconds, which ends up at about an 85 Hz frequency. The resolution of the duty cycle is dependent on the value entered in PR2, so generally we want a fairly high value in PR2. For our purposes, even 85 Hz is fast enough that we can't see the LED flashing, so almost any period is adequate.

It turns out that we can't get the PWM output to 100% on if PR2 is set to H'ff', so if we want to be able to turn off our LED, the value in PR2 cannot be higher than H'fe'.

### Setting up for PWM

In order to make PWM work then, we need to set PR2, set CCP1 (PORTC bit 2) to be an output, turn on TMR2 and set its prescale value, and configure CCP1 for PWM.

```

; PR2 sets the period
banksel    PR2
movlw     H'fe'      ; Setting period to max
movwf     PR2       ; gives max resolution

; Set PWM pin to be an output
movlw     B'11111011' ; Clear TRISC<2>
movwf     TRISC     ;

; TMR2 prescaler scales the period and duty cycle
banksel    T2CON
movlw     B'00000101' ; TMR2 on <2>, prescale
movwf     T2CON     ; to 1:4 <0:1>

; Configure CCP1 for PWM
movlw     H'0f'      ; <3:0> all 1's for PWM
movwf     CCP1CON   ; <5:4> is duty cycle LSB
banksel    PORTA

```

### Setting the Duty Cycle

Now that the frequency for the pulse train has been established, it is necessary to set the duty cycle. Since the intent is to control the brightness of the LED depending on the pot position, the analog input value can be used to set the duty cycle.

Conveniently, both the A/D converter and the PWM duty cycle have 10 bit resolution. Somewhat inconveniently, the least significant bits of the PWM duty cycle are located in the middle of the CCP1CON control register.

Note that setting PR2 near full scale gives the full PWM resolution. If a lower value had been chosen for PR2, then the full resolution would not have been available for the duty cycle. For many applications, a few bits of resolution is adequate.

*Continued on next page*

## Pulse Width Modulation, Continued

### Setting the duty cycle (continued)

To set the duty cycle it is necessary to read the analog input value, shift the least significant bits into the correct position, then send the high and low bytes to the correct registers. CCP1RL contains the high eight bits, but CCP1CON contains not only the low two bits, but also 4 bits that configure the CCP port for PWM. These 4 bits must be kept high, so the least significant bits of the analog value must be ORed with H'0f'.

```

; Save the analog value
movf      ADRESH,W      ; Get top 8 bits
movwf     AnaH          ; Save into hi 8 bits
movf      ADRESL,W      ; Grab low 2 bits and
movwf     AnaL          ; save into low

; Move the high two bits of AnaL to bits <5:4>
movlw     H'c0'         ; First mask off low
andwf     AnaL,F        ; bits of AnaL
bcf       STATUS,C      ; Rotate high one bit
rrf       AnaL,F        ; making sure <7> clear
rrf       AnaL,F        ; Here we know <7> clear

; Set the high 8 bits of the duty cycle
movf      AnaH,W        ; Set high 8 bits of
movwf     CCP1L         ; duty cycle

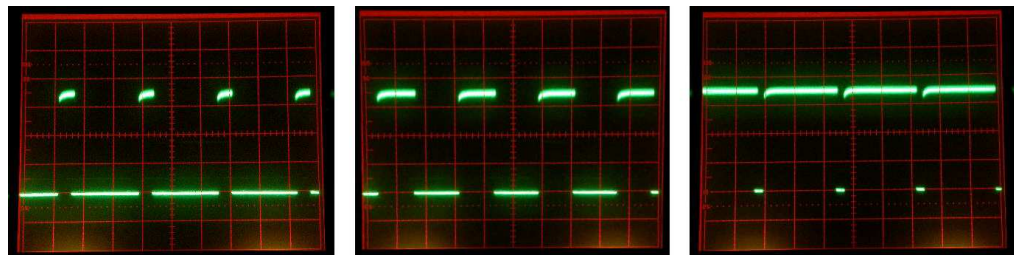
; and now the low 8 bits
movlw     H'0f'         ; Need to keep PWM mode
iorwf     AnaL,W        ; OR in low analog bits
movwf     CCP1CON       ; and send to CCP1CON

```

Notice that both the CCP1L and the CCP1CON registers are in Bank 0, so there is no need to be concerned with bank switching. Also, the duty cycle registers are double-buffered on the chip, so changing the duty cycle will not cause glitching of the output.

### Testing the program

When loaded and run, the LED brightness should vary with the potentiometer position. If a scope is available, a probe on PIC pin 13 should show a square wave that varies in duty cycle:



While a varying duty cycle pulse is a good choice for changing the brightness of an LED, a simple RC filter can change this into a varying DC level for other applications.

## Wrap Up

### Summary

In this lesson, we have shown how we can use the PIC-EL as an in-circuit programmer for our own circuit, even if our circuit should use a different, possibly more capable, PIC than the 16F84A used in the PIC-EL. We have also examined two new peripherals, the Analog to Digital Converter, and the Pulse Width Modulation output.

We are now positioned to design our own circuits using a wider range of PICs. PICs may have many peripherals that we haven't touched on yet, but they are all accessible merely by reading the data sheet.

Designers considering more advanced in-circuit serial programming projects are encouraged to study Olin Lathrop's excellent paper on the topic at:

<http://www.embedinc.com/picprg/icsp.htm>

### Coming Up

In the next lesson, we will return to our PIC-EL and examine interrupts. Interrupts allow us to make our code more responsive, and in many cases, to make it simpler, as well.