PIC Laboratory Board Manual

V1.2

By:
John Peatman
Georgia Tech

Table of Contents

Foreword
PIC Laboratory Board Manual 3
Project One
Introduction to the Microcontroller Development System 6
Project Two
Bar Graph Display 8
Project Three
Bar Graph Intensity 9
Project Four
LCD Display 11
Project Five
Display of Character Codes 14
Project Six
Temperature Measurement 16
Project Seven
RPG Parameter Entry 19
Project Eight
Square-wave Output 20
Project Nine
Rate-Sensitive RPG Parameter Entry 22
Project Ten
Parameter Entry via Keypad 23

PIC Laboratory Board Manual

John B. Peatman, Georgia Tech

The following projects are intended to provide a framework for putting together an instructional laboratory using a PIC16C74A microcontroller. They make use of Microchip Technology's new PIC Education Board. They assume that each station of the laboratory consists of a PC for creating and assembling code, either an emulator (preferred) or a programmer and ultraviolet eraser to put assembled code into a PIC chip, and a PIC Education Board. An emulator speeds up the development cycle and gives quicker insight into the sources of bugs in student-generated code.

The laboratory projects make reference to **Design with PIC Microcontrollers** (see http://www.picbook.com). My experience is to use the same first project each new time our microcontroller course is taught. The intent of this first project is to get students into the laboratory and to have them become familiar with the tools available and the lab procedures. As a lead-in to this first lab, my teaching assistants provide several one-hour demos during the first week of the course, explaining lab procedures and then going through the first lab project. We encourage students to work in teams of two, both to be more efficient in the use of the lab equipment and, perhaps more importantly, to have code debugging supported by two people who presumably understand their code in addition to the help of the TA.

Comments regarding Project One

Working code for the first project is provided in the lab. Each team copies the code onto a floppy (or into their own account on a server). Students learn to use the text editor and assembler in the lab. They make several minor modifications to the code, print a listing, and run their code. Since they do not understand anything about the PIC yet, any modifications must be well directed.

In the course, I begin with an explanation of the CPU register structure and addressing modes. Then, instead of going through an explanation of the instructions one by one, I use this Project One code as the introduction to what each instruction in the code does. This places the instruction's use in the context of a program they can understand. It also arms students to carry out Project Two, where they will need to understand enough about their code and the PIC to make a small functional addition to the code of Project One.

The code for this first project has a ten-millisecond mainline loop, used to time slow events. The PIC's Timer2 is used to define each ten-millisecond interval. The lighting of each of the top three LEDs is changed every half second. In addition to illustrating half-second-interval timing, this project also illustrates how tables are employed in PIC code.

One of the interesting resources on the board is an *encoder emulator*. This provides a low-cost alternative to the rotary pulse generator (RPG) shown on page 99 of **Design with PIC Microcontrollers**. Every time either the INC or the DEC key is depressed, the encoder emulator generates a narrow pulse that is connected to the PIC's RB0/INT interrupt input. A second output of the encoder emulator drives the PIC's RC0 input. It indicates the direction of rotation of the emulated RPG. It is high for INC presses and low for DEC presses, giving direction information analogous to that of the second output of an RPG. If either key is held down, a string of pulses is emitted, slowly at first and fast shortly thereafter. This emulates the turning of an RPG slowly to enter small parameter variations and fast for large variations. The code of Project One echoes the two encoder emulator outputs to the bottom two LEDs, to give an intuitive grasp of its operation. This code also illustrates the handling of PIC interrupts.

The intent of the Project One code is to provide students with a **template** for subsequent code writing. It includes

- a header saying what the code does and to identify their names
- a program hierarchy showing the relationship of subroutines within the mainline code and interrupt handlers within the interrupt service routine
- *list* and *include* and *__config* directives
- equates of names used in the code
- definitions of variables used in the code

- macro definitions
- reset and interrupt vectors
- tables used by the code
- the mainline program and its subroutines
- the interrupt service routine and its handlers

Different code for the first project from that provided here can of course be substituted for this project. However, it is my experience that even if the code used for Project One is never changed, this no impediment to having richly varied subsequent projects from one quarter (or semester) to the next.

Text references for Project One

Chapter 3	Pages 33-54	Assembly language program conventions
Section 3.6	Pages 51-54	Macro use
Section 3.5	Pages 46-51	Table use
Sections 4.2, 4.4	Pages 58-63	Timer2 use and its initialization
Section 4.3	Pages 60-61	Interrupt logic
Section 6.2	Pages 95-100	RB0/INT external interrupts; RPG use
Section 5.3	Pages 82-85	Interrupt service routine and its handlers

Functioning of Project One code

The **Mainline** program first calls **Initial** to initialize everything. Then it repeatedly calls the **Blink** and the **LoopTime** subroutines every ten milliseconds. Subsequent projects will add further tasks that need to be checked upon every ten milliseconds.

The **Initial** subroutine initializes every PIC I/O pin either as an input or as an output, as determined by the circuitry of the PIC Education Board. It also initializes **ADCON1**, a necessary prerequisite not only for our subsequent use of three **PORTA** pins as ADC inputs but also to enable the remaining pins of **PORTA** as well as the pins of **PORTE** as digital I/O pins. It initializes Timer2 as a ten-millisecond-period counter. It finishes up by enabling RB0/INT interrupts.

The **Blink** subroutine counts a **Blnkcnt** variable so as to change the LED pattern only every fiftieth time around the mainline loop (i.e., every 500 milliseconds). It uses the value returned by the **BlinkTable** subroutine to select which bits of **PORTD** to toggle.

The **BlinkTable** subroutine reads the state of the LEDs driven by **PORTD** into **W**. It moves bits 7,6,5 of **W** to bits 2,1,0 and forces the remaining bits to zero. Thereby, **W** contains a number between 0 and 7, depending upon which of the top three LEDs are turned on. This number is added to the program counter, producing a jump down to one of eight **retlw** instructions. The selected **retlw** instruction causes a return from the subroutine with ones in those bits of **W** which correspond to bits of **PORTD** to be toggled to produce the desired change in the LED pattern.

The **LoopTime** subroutine simply waits for Timer2 to complete its ten-millisecond count cycle. For this to provide a ten-millisecond loop time, we assume that even in the worst case, the calls of mainline subroutines plus all possible interrupt service routines never add up to more than ten milliseconds. Given this, we can count loops to time longer intervals (e.g., an action which is to be taken every second can be controlled by taking action every hundredth time around the mainline loop). This subroutine also toggles bit 5 of **PORTA**, providing a signal that can be used to measure the loop time with a scope or a universal counter.

The **IntService** interrupt service routine is entered whenever *any* interrupt occurs. It sets aside **W** and the **STATUS** register before polling the various possible interrupt sources to determine which one caused the interrupt. After servicing all interrupts requiring service, **STATUS** and **W** are restored and the CPU returns to the execution of the mainline program. Note that each interrupt handler is called with a **goto** instruction and terminates with a **goto** instruction. As discussed in Chapter 5, this is needed for the PIC to assign priority to interrupt sources; **call** instructions will not achieve the desired priority.

The **RPG** handler clears its interrupt flag and then toggles the **INTEDG** bit of **OPTION_REG**. As pointed out on page 98, this will cause the next edge (rising or falling) from the encoder emulator to produce an interrupt. Because **OPTION_REG** is located in Bank 1, students will see how indirect addressing can be used for its access. The two encoder emulator outputs are then copied to the bottom two LEDs so as to give a visual indication of the operation of the encoder emulator.

Students "To Do" Handout for Project One

Project One is prepared assuming the use of Microchip Technology's new MPLAB-ICE 2000 emulator. Given the tools actually used, this project should be rewritten to engage students in their use.

3-6-99

Project One
Introduction to the Microcontroller Development System

Complete by:
End of week two

References:
Emulator (summarizing handout)
Schematic for PIC Education Board

Before Lab:

[ ] Attend the demo.

[ ] On the **Lab Teams** sign-up sheet, list a *handle* together with your name and that of your partner (if you have one). You will need one or two 3.5-inch floppy disks.

Using the emulator:

Please be sure to familiarize yourself with the information in the handout. This information goes into further detail about the emulator than the information presented here in Project One.

[ ] Turn on one of the PCs, if necessary. Turn on power to both the PIC Education Board and the emulator.

[ ] Logon to the PC using the *userid* "PIC" (without quotes) and password "PIC" (without quotes). The login process will take 30-60 seconds while the PC's hard disk is scanned for viruses. If a warning or error occurs during login or bootup, please contact the TA.

[ ] See the P1 program attached to this handout. Copy it from a floppy borrowed from the Lab TA onto your floppy disk. You will keep your files on a floppy, rather than on the PC's hard disk. In this way you will have a backup copy that you carry away with you. We also want to avoid misuse of your files by someone else.

WARNING: All user files are deleted from the PC's hard disk whenever the computer is rebooted or a user logs on.

[ ] Select the emulator software by clicking the appropriate button on the taskbar at the bottom of the screen. Load the P1 source file (P1.ASM) into the emulator.

[ ] Insert your handle and name(s) somewhere within the first few lines of the program. Be sure that a semicolon is placed at the beginning of the line so that your handle and name(s) will be considered as comments by the assembler.

[ ] Save your updated P1 source file.

[ ] Assemble your P1 source file. If an error occurred during assembly, modify the source file, save it, and re-assemble until all errors have been corrected.

NOTE: Treat all warnings generated during assembly as errors. Several forms of *typos* will result in warnings being generated during assembly rather than errors. The warnings often result in assembled code that does not perform as expected.

[ ] Print a copy of the generated list file, P1.LST, to familiarize yourself with the printing process. Refer to the handout.

[ ] Download the assembled program (P1.COD) to the emulator.

NOTE: This step must be repeated every time the source is reassembled.

WARNING: Failure to reload the assembled program into the emulator will result in wasting countless hours trying to debug an outdated program.

[ ] Verify that the emulator has been setup correctly to emulate the PIC16C74A microcontroller with a 4 MHz clock (for a 1 MHz internal clock rate).

WARNING: Failure to configure the emulator correctly will result in wasting countless hours trying to debug a flawless program.

[ ] Run P1. Press the INC and the DEC keys to monitor how the program echoes the encoder emulator outputs to the lower two LEDs.

[ ] Stop the execution of the program.

[ ] Often when debugging a program, there is a known point at which you want to stop the program and view what has happened to variables up to that point. The emulator provides the ability to set breakpoints that allow program execution to be stopped automatically once a specific program location has been reached. Set up a breakpoint at the address label **RPG**.

[ ] Viewing the contents of various PIC registers/RAM variables can be accomplished by setting up *a watch window*. Add PORTD and **OPTION** as watch variables. Now that a watch window has been configured, whenever the program stops executing (e.g., by reaching a breakpoint or by single stepping), the values displayed in the watch window will be updated.

[ ] When you are ready to begin execution again, you have the option of beginning from reset or of beginning from the point where you stopped. Run from reset to the breakpoint at the beginning of the **RPG** interrupt handler. Press the **INC** key to initiate an interrupt.

[ ] To facilitate debugging, the emulator supports *single-step* instruction execution. Single step through the **RPG** handler, watching the effect of each instruction upon the two watch variables.

[ ] Remove the breakpoint.

[ ] Set up the MPLAB-ICE 2000 emulator's trace feature to trigger about "RPG". Run from reset and press the INC key to cause an interrupt. Stop execution. Look at the trace display, and examine the captured execution of instructions leading up to the RPG trigger point. In particular, back up to where the PIC was executing the mainline code just before it fielded the interrupt. What was the CPU doing at the time of the interrupt? Can the emulator be set up to show the cycle by cycle operation from mainline program to interrupt service routine so you can see how cycles were used in the transition (i.e., the transition indicated on page 76, in Figure 5-2)? What you do not want to do here is have the emulator filter out dummy cycles. Does the operation match that of Figure 5-2?

[ ] Now move past the trigger point in the trace display and monitor the operation of the successive instructions upon **PORTD** and **OPTION**. Does this give a more helpful view of how the instructions affected these registers?

[ ] Set up the emulator to capture just the instruction labeled **MainLoop** but to do so repeatedly so as to capture the successive times it takes to traverse the mainline loop. Run from reset and then examine the captured results in the trace buffer. Are these times all exactly 10,000 cycles? Or do they vary slightly? If so, how much do they vary and why?

Winding Up:

[ ] Be sure to save your file(s) onto your floppy disk.

Project Two
Bar Graph Display

Reference:

Chapter 10 Analog-to-Digital Converter

Project Description

For this project you are to turn on a number of LEDs, starting from the bottom, which is proportional to the output from the top potentiometer.

RPG Handler

Delete all the lines of this handler other than the

```
bcf     INTCON,INTF
```

instruction at the beginning and the

```
goto    Poll
```

instruction at the end.

When on a future project we want to use the "INC" and "DEC" keys, we will fill in this handler with the instructions to take appropriate action.

Analog-to-Digital Converter

The top potentiometer is connected to the RA3/AN3 input on the PIC. Referring to the **Initial** subroutine for Project One, **ADCON1** is already correctly initialized to B'00000100'. Referring to Figure 10-6 on page 187, ADCON0 might be loaded with B'01011001' each time a conversion is to be initiated so that if another of the ADC inputs is used on a later project, this will still work.

```
MOVLF   B'01011001', ADCON0     ;Select ADC's AN3 input
bsf     ADCON0, GO_DONE         ;Start conversion
```

Then test this **GO_DONE** bit to wait until it returns to zero (after the nineteen-microsecond conversion time) before reading the result from the **ADRES** (A-to-D Result) register.

Bar Graph Subroutine

This subroutine should be called each time around the mainline loop. It should initiate a conversion of the potentiometer's output voltage and then wait for the completion of the result. If the result is H'00', then turn off all **PORTD** LEDs. Otherwise, call the **BarTable** subroutine described below. Take the value returned by the subroutine in the **W register** and write it to **PORTD**.

BarTable Subroutine

We will not use the **BlinkTable** subroutine any more. However, you are to create a new **BarTable** subroutine that does a similar operation, so just rename and modify the **BlinkTable** subroutine to do this. It is important that whatever *tables* you create in your code be placed within the first 256 program locations. Locating any tables in the section of code following the *Vectors* section will accomplish this.

Noting that the upper three bits of the **ADRES** register divide the potentiometer's range into eight equal parts, move these three bits to bits 2, 1, 0 of **W**, blanking the remaining bits. Then return from the subroutine with **W** loaded with one to eight ones which can be written to **PORTD** to represent the Bar Graph display of the pot's output voltage. The highest voltage range should turn on all eight LEDs while the lowest of the eight ranges should turn on just the bottom LED. If **ADRES** = B'00000000', turn off even that bottom LED.

Project Three
Bar Graph Intensity

Reference:
Section 6.3 Timer0

Project Description

For this project, your code for Project Two should continue to work. In addition, the middle pot is to control the intensity of the LEDs by pulse-width modulating them. Use Timer0 to control the PWM duty cycle.

Analog-to-Digital Converter

The middle potentiometer is connected to the RA1/AN1 input on the PIC. Referring to Figure 10-6 on page 187, load **ADCON0** with B'01001001' each time a conversion is to be initiated:

```
MOVLF   B'01001001', ADCON0     ;Select ADC's AN1 input
bsf     ADCON0, GO_DONE         ;Start conversion
```

Then test this **GO_DONE** bit to wait until it returns to zero before reading the result from the **ADRES** (A-to-D Result) register.

Timer0 Interrupts

Change the initialization of the **INTCON** register to include the setting of the **T0IE** bit shown in Figure 6-5 on page 101. Add to the polling routine in the **IntService** interrupt service routine the following code

```
btfsc   INTCON,T0IF
goto    Intensity
```

where **Intensity** is the interrupt handler which does the pulse-width modulation steps discussed below. Conclude this handler with

```
goto    Poll
```

Intensity Interrupt Handler

Initialize (the bank 1 register) **OPTION_REG** as shown on page 102 in Figure 6-6. If you set the prescaler to 32, then TMR0 will take 8.192 milliseconds to count through its 256 states. We will divide this interval into two parts as determined by the value, N, obtained from AN1, the analog-to-digital converter input connected to the middle potentiometer. The 8.192 millisecond cycle will produce a pulse-width-modulation frequency which is fast enough to produce no visible blinking on the LEDs (with approximately 122 blinks per second) and yet slow enough so that the CPU will spend a very small percentage of its time controlling the intensity of the LEDs.

Change your code for Project Two so that instead of writing to **PORTD**, you write to **PORTD_CPY**. Now you will copy **PORTD_CPY** to **PORTD** to turn on the *on* LEDs for N counts of **TMR0**. You will clear **PORTD** to turn off all LEDs for 256-N counts of **TMR0**.

Note from Figure 6-5 on page 101 that an interrupt can be generated when **TMR0** rolls over from 255 to 0. On every other interrupt, clear the **T0IF** interrupt flag and then control the pulse-width modulation of the LEDs by loading 256-N into **TMR0** and turn on the *on* LEDs. When the alternate interrupts occur, clear the **T0IF** interrupt flag, turn off all LEDs, and load N into **TMR0**.

Define a variable called **INT_FLAG** to keep track of alternate interrupts. Complement it each time that the **Intensity** handler is entered. Then test bit 0 (or any bit) to distinguish between alternate interrupts.

For any value of N which is 5 or less, clear the **T0IF** interrupt flag, turn off all LEDs, and do not change **TMR0**. For any value of N which is 250 or more, clear the **T0IF** interrupt flag, copy **PORTD_CPY** to **PORTD**, and do *not* change **TMR0**. With these modifications, we will insure that at least 160 microseconds occur between Timer0 interrupts, plenty of time to complete one servicing before being asked to undertake the next one.

Measuring Duty Cycle

Immediately after copying **PORTD_CPY** to **PORTD** set bit 4 of **PORTA** with

```
bsf     PORTA,4
```

Immediately after clearing **PORTD** clear bit 4 of **PORTA** with

```
bcf     PORTA,4
```

A scope display of the resulting waveform will show the duty cycle.

Measuring CPU Overhead due to LED Intensity Control

Insert

```
bsf     PORTA,2
```

as the first instruction of the **IntService** interrupt service routine. Insert

```
bcf     PORTA,2
```

just before the final **retfie** instruction of **IntService**. Consider how a scope display of the resulting waveform will give an indication of the percentage of CPU time spent dealing with the LED intensity control.

Project Four
LCD Display

References:

Section 7.6	*LCD Display*
Section 8.5	*Display strings*
Section 8.7	*Display of Constant Strings*
Example file	*LED.asm*
Information file	*Character codes.tif*

Project Description

For this project, your code for Project Three should continue to work. In addition, initially display the following message, centered on the top line of the display:

Press ENTER

When the ENTER key is pressed, display the names of you and your lab partner, again centered on the display. When the ENTER key is released, redisplay the initial message. Update the display when ENTER is changed.

LED.asm File

When this file is assembled and executed by the PIC on the PIC Education Board, it will write UL in the upper-left-hand corner of the display, UR in the upper-right-hand corner of the display, LL in the lower-left-hand corner of the display, and LR in the lower-right-hand corner of the display. Try it.

Book versus PIC Education Board Differences

The Hitachi LCD display discussed in Section 7.6 uses the PIC's Serial Peripheral Interface (SPI) and a 74HC164 shift register to acquire bytes from the PIC eight bits at a time. The FEMA LCD display on the board uses the upper four bits of **PORTB** to acquire bytes from the PIC two four-bit *nibbles* at a time. While both the Hitachi and the FEMA displays can employ either a four-bit or an eight-bit interface to the PIC, the initialization required for the four-bit interface is somewhat different from that for the eight-bit interface. Furthermore, you will note that the LED.asm file's **Initial, InitLCD, LCDinit_Table, DisplayC, DisplayC_Table,** and **T40** subroutines are somewhat different from the corresponding subroutines in the book, mainly because of the four-bit interface.

Another set of differences arises because Hitachi displays (and most other LCD character displays) employ an HD44780U Hitachi controller chip and other parts mounted with the display on a PC board module. In contrast, the FEMA display uses a tiny FCS2314AK Fujitsu controller chip mounted right on the glass substrate of the LCD display itself. By the time Fujitsu made their controller chip, the technology had advanced such that what had been a board of parts for Hitachi was now designed as a special purpose microcontroller by Fujitsu. Functionally, the Fujitsu unit is an upwardly compatible extension of the Hitachi unit. It understands all of the Hitachi commands. While I have not been able to find the Fujitsu data sheet on the Internet, the Hitachi HD44780U data sheet can be found by going to Hitachi's web page (http://semiconductor.hitachi.com/) and then searching for HD44780U. This data sheet will correctly answer most questions concerning our display.

The 5x7 characters displayed for all of the ASCII codes shown in Figure 7-9 on page 138 are the same. The complete character set is shown in the attached **Character codes.tif** file. This Fujitsu unit includes characters that are useful for making an 80-element horizontal Bar Graph display extending horizontally across one of the rows of what is normally a 16x2-character display. Likewise, it includes characters that can be used to make a 14-element vertical Bar Graph display using any two characters located in the same column.

Initial Subroutine

Your **Initial** subroutine must be terminated by calling the **InitLCD** subroutine last thing (just before the "return" instruction is executed). By that point, the ports and the **LoopTime** subroutine needed by the **InitLCD** subroutine will have been set up correctly.

InitLCD and LCDinit_Table Subroutines

The **InitLCD** subroutine first pauses for a quarter of a second because both the Fujitsu controller chip and the PIC have a power-on reset circuit. The quarter-second pause insures that when the PIC starts to execute the code which will initialize the LCD display, the Fujitsu controller will be out of reset, ready to accept commands.

The subroutine next calls the **LCDinit_Table** subroutine with **LCD_TEMP** = 0. This subroutine adds the zero to the program counter and jumps to the very first **retlw** instruction, returning with H'33' in **W**. Each subsequent time **LCDinit_Table** is called, it will be with **LCD_TEMP** incremented by one. In this way the **InitLCD** subroutine gets byte after byte from the string of bytes stored via the successive **retlw** instructions in **LCDinit_Table**. When the byte retrieved equals zero, the end of the string has been reached.

The display is initialized to the *nibble mode* (whereby subsequent bytes are received as two successive 4-bit nibbles) by sending each of the nibbles of the first two bytes (i.e., 3 - 3 - 3 - 2) interspersed with the strobing of the display's E pin and a pause to allow each nibble to be digested. The RS pin of the display is held low for all nibble transfers, telling the display that these are commands, not displayable characters. Subsequent nibbles are paired together into bytes by the display. These four bytes tell the display controller that it is controlling a two-row display that is to be blanked initially and that it is not to display a cursor. The display controller is also told that a string of displayable characters received by the display is to be written from left to right across the display.

DisplayC and DisplayC_Table Subroutines

The **DisplayC** subroutine has a parameter passed to it in **W** which represents an offset from the CDS label in the **DisplayC_Table** subroutine to the beginning of the string of characters that will form a fixed message on the display. Each of the characters in the string is stored as part of a **retlw** instruction. For example, the _UR label identifies the beginning of a *display string* having the format described in Section 8.5 on page 149:

Cursor-positioning code
ASCII string of characters to be displayed
End-of-string designator, H'00'

For the _UR label, this becomes

C0	The *cursor positioning code* which will place the cursor at the first column of the second row of the 16x2 display. The cursor-positioning codes for every character position are shown in Figure 7-8 on page 137 for rows 1 and 2 and columns 1 to 16.
55	This is the ASCII code for the upper case letter U. The *dt* assembler directive tells the assembler to convert the characters between the quotes to their corresponding ASCII codes and to embed them into **retlw** instructions. The ASCII table shown in Figure 7-9 on page 138 shows U being coded as 0101 0101 which is the same as hex 55.
52	This is the ASCII code for R.
00	This is the *end-of-string designator*.

The **DisplayC** subroutine employs a loop of instructions. Each time around the loop, the next byte in the string is returned by the **DisplayC_Table** subroutine. The display receives the first byte with its RS (register select) input low. This signifies to the display that the C0 byte is to be treated as a control byte, to set the position where the subsequent characters are to be displayed. At the end of the first time around the loop, the RS input is set to one so that all subsequent characters sent to the display are treated as displayable characters, not control codes.

Modification of DisplayC_Table

The strings you are to display for this project will be known at the time of assembly (e.g., "Press ENTER"). That is, they are constant, or fixed, strings. You need to create a new display string in the **DisplayC_Table** subroutine for each one of them, with a label to name each string (e.g., "_Press") and the appropriate sequence of **retlw** instructions to handle each byte of the string. Then to display the new string, your mainline code will execute an instruction sequence analogous to

```
movlw   _Press-CDS
call    DisplayC
```

Use of Enter Switch

Note that this switch is connected to bit 2 of **PORTE**. When the switch is pressed, a zero is sensed on this pin.

Project Five
Display of Character Codes

Project Description

For this project, your code for Project Four should continue to work, subject to the modification that you are to use the ENTER key to toggle the display between several options. Initially, display the same "Press ENTER" message as for Project Four. Thereafter, presses of the ENTER key should toggle between two options. These are your names, from the last project, and the display of character codes, described below. When in the "display character codes" mode, you are to display the following. In the upper left hand corner of the LCD display:

<two hex digit number> <space> <the displayable character coded by this two hex digit number>

The entire bottom row of the LCD display is to display the sixteen displayable characters for the column of the ASCII table in which the above two-hex-digit number resides. For example, if the two-hex-digit number is 4B, then the top row will be

```
4B K
```

(because the ASCII code for the letter K is 4B) and the bottom row will be

```
@ABCDEFGHIJKLMNO
```

The RPG emulator introduced with Project One (whose function was deleted with Project Two) will be used to increment and decrement a one-byte variable called CHARCODE (initialized to H'30'). Each INC interrupt is to increment CHARCODE, stopping at FF. In like manner each DEC interrupt is to decrement CHARCODE, stopping at 00. It is this number which is to be expressed with two hex digits in the upper-left-hand corner of the display. Likewise, it is this number which is to be used as the code of the displayable character shown in row 1, column 4 of the display. And it is the left-most hex digit that is to determine which column of the ASCII table to show in the second row.

ENTER Key

Each time around the mainline loop, a subroutine named **EnterKey** is to be called. It compares the state of the ENTER key with what it was last time around the mainline loop. Only if it has changed from one to zero is action to take place. Initialize a variable called **ENTFLG** to zero. Then, each time the ENTER key is pressed, change **ENTFLG** as follows 0,1,2,1,2,1,2,1,2,1,2,....
Every tenth of a second, the **EnterKey** subroutine is to take one of the following three actions. With **ENTFLG=0**, rewrite "Press ENTER". With **ENTFLG=1**, rewrite your names. With **ENTFLG=2**, call a **Character** subroutine which will, in turn, update the display as described above. Each time the ENTER key is pressed, first clear the display to make way for the new display (which may not overwrite all character positions).

To clear the display, add two constant display strings to **DisplayC_Table**, one called **_ClrRow1** and the other called **_ClrRow2**. Calling the **DisplayC** subroutine when **W** has been loaded with **_ClrRow1-CDS** is to write sixteen blank characters to the top row of the display.

DisplayV Subroutine

For the last project, we had a **DisplayC** subroutine which extracted the bytes of a *display string* from program memory. For this project you will need a **DisplayV** subroutine which will extract the bytes of a display string from RAM. Make a copy of the **DisplayC** subroutine and edit it as follows to create a **DisplayV** subroutine.

- Change the name of the subroutine in the copy to **DisplayV**. Change labels in the subroutine accordingly.

- Save pointer in **W** to **FSR**.
- Get byte from string into **W** using indirect addressing (i.e., by using **FSR** as a pointer to the string entry)
- Increment **FSR** to point to the next byte

To update the LCD display with a variable display string, load **W** with its address and call **DisplayV**.

Row1 Display String

Add the following to your RAM variables:

Row1:6

This will reserve six characters for the display string needed for displaying the four characters in the top row. Initialize the content of **Row1** with H'80', the cursor-positioning code for the upper-left-hand corner of the display. Initialize the content of **Row1**+3 with H'20', the ASCII code for a blank character. Initialize the content of **Row1**+5 with H'00', the end-of-string designator.

Write a subroutine called **ASCIIcode** which first copies **CHARCODE** to **Row1**+4 and which then fills in **Row1**+1 and **Row1**+2 with the ASCII code for each hex digit making up the number in **Row1**+4. (Use the content of **Row1**+4, not the content of **CHARCODE**, when doing this.) For example, if **CHARCODE** contains H'4B' (the ASCII code for the letter K), then **Row1**+4 should be loaded with H'4B', **Row1**+1 should be loaded with H'34' (the ASCII code for the number 4), and **Row1**+2 should be loaded with H'42' (the ASCII code for the letter B).

Before calling **DisplayV** to display the **Row1**, call **ASCIIcode** to complete the display string, given a possibly changed value of **CHARCODE**.

Row2 Display String

Add the following to your RAM variables:

Row2:18

Initialize the content of **Row2** with H'C0', the cursor-positioning code for the lower-left-hand corner of the display. Initialize the content of **Row2**+17 with H'00', the end-of-string designator.

Write a subroutine called **FillRow2** that copies the content of **Row1**+4 into **W**, and forces the lower four bits of **W** to zero. Then, with **FSR** loaded with the address of **Row2**+1, store indirectly. In a loop, increment **FSR**, increment **W** by adding 1 to it, and store **W** indirectly. Keep this up until all sixteen entries between **Row2**+1 and **Row2**+16 have been filled. Then return.

Before calling **DisplayV** to display **Row2**, call **FillRow2** to complete the display string, given a possibly changed value of **CHARCODE**, as reflected in a changed value of **Row1**+4.

Character Subroutine

This subroutine calls **ASCIIcode**, calls **FillRow2**, loads **FSR** with the address of the **Row1** display string, calls **DisplayV**, loads **FSR** with the address of the **Row2** display string, calls **DisplayV** again, and then returns.

Project Six
Temperature Measurement

References:

Data sheet	*Tmp03_4.pdf (Analog Devices TMP04 temperature sensor data sheet) (http://www.analog.com/pdf/tmp03_4.pdf)*
Example file	*Math.asm*
Include file	*Math.inc*
Section 6.5	*CCP2 capture mode (Figure 6-10)*

Overview

For this project, all the old stuff should continue to work. In addition, add the final screen we will use with the LCD display. Every time the ENTER key is pressed, now change ENTFLG as follows 0,1,2,3,1,2,3,1,2,3,.. With ENTFLG=3, every tenth of a second update a display of the temperature having the format

71.4 °F (for temperatures under 100 °F)

or

104.7 °F (for temperatures over 100 °F)

located at the bottom right hand corner of the LCD display.

Temperature Algorithm

The temperature sensor emits a pulse-width-modulated waveform. Define the interval during each period when the waveform is high as T1 and the interval when it is low as T2. Then the temperature, expressed in tenths of a degree Fahrenheit, is

Temperature = 4550 - [7200xT1/T2]

T1 and T2 are going to be measured with the input capture feature of the PIC and will range between 10000 to 30000 (or thereabouts). To maintain accuracy, first do the multiply and then the divide using the math algorithms of math.asm and math.inc.

Example: T1 = 11424 µs and T2 = 21683 µs

7200 x 11424 = 82252800
82252800 / 21683 = 3793
4550 - 3793 = 757
The temperature in this case is 75.7 °F.

Incidentally, while the accuracy of the TMP04 temperature sensor is typically about ±2 °F, the incremental accuracy is much better than this. That is, if a temperature *change* from 75.7° to 75.9° occurs, this change of 0.2 °F is much more accurate than the ±2 °F absolute accuracy would indicate since most of the errors making up the absolute accuracy cancel out when an incremental temperature measurement is made.

Binary to ASCII Conversion

After you have completed the calculation of the temperature, you need to take the two-byte binary result and break it into a three-ASCII-character result. If you divide the two-byte binary value by ten, the remainder will be the units digit value. Add H'30' to this to convert it to the ASCII code for the digit. Dividing the quotient by ten will yield the tens digit in the new remainder and the hundreds digit in the new quotient.

Math Subroutines

The two files referenced, math.asm and math.inc, are derived from Microchip Technology's Application Note AN617 entitled "Fixed Point Routines". If you want to obtain this application note, it is available on Microchip's web site (http://www.microchip.com). Search that site for AN617 to get the Acrobat version (.pdf) of the file.

Copy into your file only those math subroutines contained in the "math.asm" file which you use. Then be sure to insert the line

```
#include    math.inc
```

immediately after the **cblock ... endc** construct already present in your code. The *include* file not only defines the math macros invoked by any of the math subroutines which you copy from the "math.asm" file, it also adds the RAM variables used by the math routines to the list of variables already defined in your code. The placement of this *include* statement into your code *must* occur *after* your variables have been assigned to addresses (beginning at address H'20'). Note that your code has the construct

```
cblock  Bank0RAM
  .
  .
endc
```

whereas the "math.inc" file has the construct

```
cblock
  .
  .
endc
```

The Microchip assembler uses a parameter with the **cblock** assembler directive as the address which is to be assigned to the first variable within the construct. The remaining variables are assigned to the consecutive addresses that follow this one. A **cblock** assembler directive that does not include a parameter (as is the case within the math.inc file) will assign its variables to addresses that begin where a previous "cblock ... endc" construct left off.

All of the math subroutines expect the two operands to be passed to them in two variables called AARG (An argument) and BARG (B argument). The bytes of these two arguments are names, **AARGB0, AARGB1**,... where the B0 byte represents the most-significant byte. The result of an operation is represented the same way. However, a common error is to extract the result from the wrong bytes. For example, if you divide a 32-bit number by a 16-bit number, the result (in general) can be as large as 32 bits. In our case, the result will be only 16 bits (as you would expect, since to form the temperature it is subtracted from 4550, another 16-bit number). You need to take the 16-bit result from **AARGB2 and AAARGB3**, not from **AARGB0 and AARGB1**.

Measuring Time Intervals

Every tenth of a second when the temperature display is to be updated, carry out the calculation of temperature using whatever values of T1 and T2 are available. Update the appropriate display string and send it to the display. Then begin the next temperature measurement by setting up an input capture interrupt when the next rising edge occurs on the CCP2 input. Be sure to clear the interrupt flag before enabling the interrupt. Also note that since this interrupt is enabled by setting a bit of a bank 1 register (i.e., the **CCP2IE** bit in the **PIE2** register), you should set it using indirect addressing. When the interrupt occurs, save **CCPR2H** and **CCPR2L**, the captured time of occurrence of the rising edge of the PWM output of the temperature sensor. Now set up to capture the time of occurrence of the next falling edge. Finally, set up to capture the time of occurrence of the next rising edge. When this has been done, disable further CCP2 interrupts and update T1 and T2 with the newly formed values. If T1+T2 is less than forty-five milliseconds, are we assured of collecting a new value of T1 and T2 within the one hundred milliseconds between display updates?

Display String

The temperature will be displayed as seven characters (e.g., 103.4 °F). Add the following to your list of variables:
TEMPSTG:9

This will reserve nine bytes, with the first byte named TEMPSTG. In your **initial** subroutine, initialize the first byte to the proper cursor positioning code so that the temperature will show up at the bottom right hand corner of the LCD display. Initialize TEMPSTG+4 with the ASCII code for a decimal point. Initialize TEMPSTG+6 with the "ASCII" code for the degree symbol which is coded as H'DF'. You also need to initialize the "F" and then clear TEMPSTR+8 to serve as the end-of-string designator. When you display a temperature under 100.0 °F, blank the hundreds digit with the ASCII code H'20'.

Project Seven
RPG Parameter Entry

Project Description

For this project, the action of encoder emulator keys, INC and DEC, is to depend upon what is being displayed on the LCD display. If **ENTFLG** =2, then continue to use these keys to cycle through the ASCII characters. If **ENTFLG**=3, then use these keys to increment or decrement a number in the upper right hand corner of the display ranging between 10 Hz and 9990 Hz.

Procedure

Define a variable called **NUMCOUNT** as a one-byte, two's-complement number, where "0" is coded as H'00', "+1" is coded as H'01', "-1" is coded as H'FF', etc. For each INC interrupt when **ENTFLG**=3, increment **NUMCOUNT**. For each DEC interrupt when **ENTFLG**=3, decrement **NUMCOUNT**.

Within the mainline subroutine **RPG_Count**, first test **NUMCOUNT** for zero and do nothing if it is zero. Otherwise, check the most-significant bit and either increment or decrement **NUMCOUNT** appropriately, then decrement or increment the ASCII string in **RPGSTR** appropriately. However, do not change beyond the range from 1 to 999.

Create a display string that will produce a display having the following format:

xxx0 Hz

(where the xxx is constrained to the range from 1 to 999), with leading zeros blanked, giving a frequency range of 10 Hz to 9990 Hz for the entry of a frequency parameter (which will be used to generate a square wave output with this frequency on the next project). Blanking leading zeros means that instead of displaying

0240 Hz

display instead

240 Hz

Note that if the output is 10 Hz, then a DEC interrupt should produce no change. Likewise, if the output is 9990 Hz, then a INC interrupt should produce no change.

Initialize the display to "1000 Hz".

SCALE10 Counter

For the last project, you updated the display every tenth of a second, counting ten loop times to keep track of tenths of a second. If the counter that did this counting was called **SCALE10**, then you might update the display with temperature when **SCALE10** equals zero. If you update the display with the string developed here when **SCALE10** equals one, then you will do the updating of both items every tenth of a second, but not during the same loop time. This will help to spread the load on the CPU among separate times around the loop (so as to help keep the time taken by everything other than the **LoopTime** subroutine to less than ten milliseconds).

Project Eight
Square-wave Output

Reference:

Compare Mode (Section 6.4)

Project Description

Using the entered value from Project Seven, generate a squarewave on the PIC's CCP1 output with a frequency equal to the displayed value. Initialize the frequency to 1000 Hz.

CCP1 Initialization

Refer to Figure 6-7 on page 104. Change the initialization of **TRISC** so that bit 2 is cleared. Add the initialization of **CCP1CON** (H'09'). Define a new sixteen-bit variable with the two one-byte variables **HALFPERL** and **HALFPERH**. Initialize these variables to one-half of the period of the initial frequency setting; that is, 500 microseconds or 500 cycles of the PIC's internal 1 MHz clock. Thus initialize **HALFPERH** to H'01' and **HALFPERL** to H'04' since D'500' = H'01F4'.

An interrupt to the CPU must be made to occur whenever the sixteen-bit register **TMR1** equals **CCPR1**. Accordingly, the interrupt source must be enabled by setting the **CCP1IE** bit in the (bank 1) **PIE1** register. You can use a **bsf PIE1,CCP1IE** instruction to do this since the power-on-reset content of the **PIE1** register is H'00' (refer to the "Value on POR" column of the table on page 250).

Be sure to take into account the bank location of each of these registers as you add code into the **Initial** subroutine.

CCP1 Interrupt Handler

Referring again to Figure 6-7, the sixteen-bit counter, **TMR1**, will increment continuously every microsecond (with our external crystal having a frequency of OSC = 4 MHz). A CCP1 interrupt will occur when the sixteen-bit counter, **TMR1**, equals the sixteen-bit register **CCPR1**. At the same time, the CCP1 pin will be either set or cleared, depending upon whether bit 0 of the **CCP1CON** register is cleared or set. The setting or clearing of the CCP1 pin takes place at this time even though the CPU is probably executing code in the mainline program at this time. That is, the CCP1/TMR1 circuitry controls the time of each edge, not the CPU. Thus, the output waveform can be made *jitter-free* by using the timer circuitry in this way.

The handler has four jobs. It must toggle an otherwise unused output pin to illustrate the jitter which can arise when the CPU is involved in generating output changes. (Use bit 5 of **PORTA** for this purpose, removing from your code any earlier use of this pin.) It must toggle bit 0 of **CCP1CON** (leaving the other bits unchanged) so that the opposite edge will occur on the CCP1 pin when the next interrupt occurs. It must add **HALFPER** to **CCPR1** so that the next compare will occur exactly one-half of the period of the waveform after the one that has just occurred. (Add the lower bytes, increment the upper byte if a carry occurs, then add the upper bytes.) Finally, it must clear the **CCP1IF** flag in the **PIR1** register. Note that if this flag is cleared first rather than last, erroneous behavior of the output waveform will occur occasionally, for some values of frequency. Why is this?

RPG_Count Subroutine

Modify this subroutine with the introduction of a new two-byte variable called **FREQH,FREQL**. Initialize it to D'1000' = H'03E8'. Each time that the value of the frequency is changed in response to presses of the INC or DEC key, change both **RPGSTR** and **FREQ**. Thus, these should always represent the same number, the one in ASCII string form, the other as a two-byte binary number. Also, set a flag called **RPGCHG** when such a change occurs. This flag

will serve as a signal to the **Period** subroutine, below, to recalculate the half-period value, **HALFPER**, from the new frequency.

Period Subroutine

This subroutine checks the **RPGCHG** flag. If set, the subroutine clears it, recalculates the value of **HALFPER**, and returns. If clear, the subroutine simply returns.

The relationship between **HALFPER** and **FREQ** is given by

FREQ x10= 1,000,000/(2x**HALFPER**)

Or

HALFPER = 50,000/**FREQ**

where **FREQ** has units of 10 Hertz while **HALFPER** has units of clock cycles (or microseconds). Accordingly, we need to execute a division of a sixteen-bit unsigned binary number by a sixteen-bit unsigned binary number, which will produce a result that, even with **FREQ**=1, will fit in a sixteen-bit binary number. Use the **FXD1616U** subroutine in the math.asm file.

Project Nine
Rate-Sensitive RPG Parameter Entry

Project Description

For Project Seven we entered a parameter having 999 values using an RPG emulator which produces a maximum of about 33 interrupts/second. Thus to change from one extreme to the other requires 999/33 = 30+ seconds. Instruments (e.g., function generators) commonly use an RPG for entering parameters, especially if the parameters must be *tweaked*, or varied back and forth, around some nominal value. To get around the problem of turning the RPG knob rotation after rotation to get to the desired nominal value, they resort to *rate-sensitive* RPG use. That is, if the RPG is turned fast, it produces larger increments than if turned slowly.

For this project, introduce a new one-byte variable, **RPGTIME**. Every ten milliseconds when **RPG_Count** is called, increment **RPGTIME**, stopping when **Threshold** (defined below) is reached. Modify the **RPG_Count** subroutine as follows. If **NUMCOUNT** <> 0, then compare **RPGTIME** against **Threshold**. If **RPGTIME** equals **Threshold** then change **RPGSTR** by one count. Otherwise, change by ten counts. In either case, reset **RPGTIME** to zero.

Threshold

We want to draw a distinction between the fast turning of an RPG and slow turning. Our encoder emulator translates this distinction into a fast sequence of interrupts, spaced just thirty milliseconds apart, and a slow sequence of interrupts, spaced 150 milliseconds apart when the INC or DEC button is first pressed or an even slower sequence of interrupts, if one of these buttons is pressed and quickly released several times. If we pick **Threshold** to be any number greater than 3 and less than 15, we will draw the distinction we want for our encoder emulator.

As an example of the analogous operation of an RPG, suppose that we have an RPG which generates 32 interrupts per revolution. Then setting **Threshold** to 6 will distinguish turning rates of greater than one-half revolution per second (i.e. 16 interrupts/second or 62 milliseconds between successive interrupts) from turning rates of less than this.

Project Ten
Parameter Entry via Keypad

Reference:

State machines and keyswitches (Section 8.3)

Project Description

For this project all the old stuff should continue to work. In addition, a keypad-entered four-digit number representing a voltage is to be displayed in the upper left-hand corner of the display

The four-digit number represents a second parameter which might be used in the setup of an instrument (e.g., the amplitude of the output of a function generator). Display it in the format of a voltage ranging from -9.999V to 9.999V:

-x.xxxV or +x.xxxV

KEYSTR String

Create a new nine-byte display string for writing the seven-character voltage string to the left-most seven character positions on the top row of the LCD display. Initialize it so that calling the **DisplayV** subroutine with it will produce a display of

+0.000V

AnyKey Subroutine

Section 8.3 includes some code which is written for a keypad having a somewhat different connection to the PIC from that which we have on our target system board. The question “Is any key pressed?” of Figure 8-3 is to be answered by calling an **AnyKey** subroutine which is modified from that on page 146 to drive bits 7,6,5 and 4 of **PORTB** low and then to test bits 3,2, and 1 of **PORTB** to see if all of them are high (corresponding to no keys pressed).

ScanKeys Subroutine

Modify the **ScanKeys** subroutine of page 148 to do the job described there, but for our target system. The subroutine uses a new variable, **TEMP**, to test each key, one at a time, beginning with the "0" key. It does so by accessing a table like that contained in the **ScanKeys_Table** subroutine, but changed (if necessary) to account for the different target system configuration. Each table entry must have a zero in the position which drives the selected row of keys (e.g., the top row with keys 1, 2, and 3) and ones in the other three row positions. In this way writing the table entry to the port will select the desired row by driving it low. For example, the column selecting the "2" key is selected by writing

0111xxxx

to the port. Bits 3,2, and 1 of each table entry should match what will be read back from the port if the selected key is pressed. Again for example, the table entry for the "2" key is

0111101x

The final bit of the table entry can be made zero.

The **ScanKeys** subroutine first initializes a **KEYCODE** variable to zero. Then it repeatedly tests key after key (key 0, key 1, key 2, etc.), using **KEYCODE** as an offset into the table. The table entry from **ScanKeys_Table** is returned in **W**. It is written to **PORTB**. Then **PORTB** is immediately exclusive-ORed with **W**, producing

0000000x

if the selected key is pressed.

Because bit 0 of **PORTB** represents an input that is independently driven by the encoder emulator, its value, when read back, is unrelated to what was written out to **PORTB**. Accordingly, W must be ANDed with

11111110

to force bit 0 to zero and to set the Z bit if the selected key (only) is pressed. The entire keypad is scanned in this way until either a key press has been detected or all twelve keys have been checked unsuccessfully (for whatever reason).

KeySwitch Subroutine

Implement the algorithm of Figure 8-3. The action to be taken when a new keypress has been detected is to call a new **Voltage** subroutine, described next.

Voltage Subroutine

This subroutine manipulates **KEYSTR**, which can be thought of as being represented by

sw.xyzV

If the key pressed is a digit key, it copies x to w, y to x, z to y, and finally copies **KEYCODE** (converted to ASCII) to z. Then it calls the **DisplayV** subroutine to display this new string.

CHS Key

If this key is pressed, then change the ASCII code representing the sign of the number.

Decimal Point Key

If the "." key is pressed, then reinitialize the display to

+0.000V