

Lesson 17

Liquid Crystal Displays

Overview

Introduction	Most PIC projects seem to involve no more than some sort of input conditioning and a liquid crystal display. In this lesson, we examine how to control an LCD module.	
In this section	Following is a list of topics in this section:	
	Description	See Page
	LCD Modules	2
	The LCD Electrical Interface	4
	The HD44780 Controller	5
	Details of the LCD commands	6
	Sending Data to the LCD	9
	Regression Testing	10
	Writing the Test Harness	12
	Reviewing the map	17
	The structure of LCDlib	18
	Sending a nybble	20
	Sending a command byte	23
	Sending a Character	24
	Clearing the LCD	25
	Setting the Cursor Address	26
	Making the LCD Scroll	27
	Turning off Scrolling	28
	Initializing the LCD	29
	The 16 character display	30
	Additional Experiments	31
	Wrap Up	32

LCD Modules

Introduction	<p>Many PIC projects involve liquid crystal displays. LCDs require far fewer connections than LED displays and use much less power. As a result, they can be a lot more convenient as a frequency display, text display, any kind of display. But for many, controlling these things can be daunting. With an LED display, a few volts on the right pins will cause a segment to light. With an LCD, they will sit there and do absolutely nothing until you get a lot of things right! Once you do, however, they can be very convenient.</p>
Types of LCD displays	<p>LCD displays are available in two broad categories. The basic LCD is usually a few digits, or some application-specific format. This type of LCD tends to have a lot of connections ... from dozens to hundreds! Because they are complex to use, they are rarely applied to hobbyist projects. They have the advantage of being very cheap, especially in quantity.</p> <p>LCD modules are generally alphanumeric displays from 8 to 80 characters. These modules have either 14 or 16 pin connections. Almost universally, they include a Hitachi HD44780 controller or a compatible clone.</p> <p>There are also graphic LCDs available, which we will not discuss here.</p>
Extended Temperature Displays	<p>Typical LCDs only operate down to 0° C. Some specialized LCDs can operate down to -40° C. These are called <i>extended temperature</i> displays. Extended temperature displays require that their contrast pin be supplied with a negative voltage. The current demands are not high, and are easily supplied by a charge pump, but the additional complexity makes them unpopular for hobbyist applications.</p>
Backlights	<p>LCD displays may be backlit either with LED backlights, or electroluminescent backlights. LED backlights can draw prodigious current. For larger displays, backlight currents on the order of an ampere are not uncommon. Electroluminescent backlights require high voltage AC. Some displays get by with 100 volts, others require over 1000 volts. Frequencies tend to be between 400 Hz and 1 kHz.</p>

Continued on next page

LCD Modules, Continued

Display Formats

LCD modules tend to be available in a few common formats. Early PIC-EL's shipped with an 8 character display. Later PIC-EL's used a 16 character display. Formats of 16 characters by 2 lines, 20x2, 24x2, 20x4 and 40x1 are also common.

The organization of the lines on the display can often be quite confusing. For example, it is not uncommon for the third line of a four line display to be "line 2". The 16 character display on later PIC-EL's is actually a 2 line by 8 display, with the second line occupying the right eight characters.

The HD44780 has limited memory, so displays involving more than 80 characters require more than one controller. These displays are fairly uncommon (and expensive).



Ocular 16x1 showing HD44780 and HD44100



Lumex 16x2 showing potted controllers

Most 16x1 displays are actually 8x2. You can recognize a 16x1 by the 44100 in addition to the 44780 on the back (see picture at left). The 8x2 will have only the 44780. Hitachi makes the controllers available both in flat packs and as bare chips. On larger volume displays, the bare chips are often more cost effective, so many displays will have the chips hidden under potting material.

Character Generators

If you notice the picture of the Ocular LCD, the controller is marked HD44780A00. The A00 refers to the character generator ROM in the controller. There are two models of character generator. Both models have the standard ASCII character set stored as characters H'20' through H'7D'. However, the parts differ in the characters available in the H'7E' through H'FF' locations. The A00 part has Japanese characters in the from H'A1' through H'FF', along with a few graphic, Greek, and math symbols. H'80' through H'A0' are unused. The A02 part has the European characters plus a few graphic, Greek and Cyrillic characters. The A02 part also includes additional graphic characters at H'10' through H'1F'. The A00 part is by far the more common.

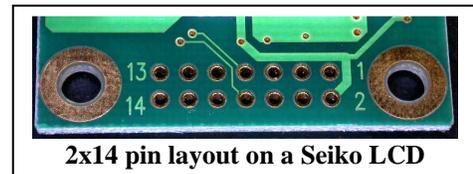
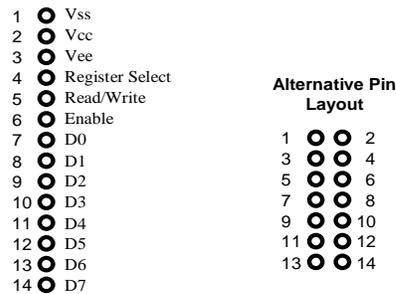
Serial LCDs

Some LCD modules are provided with a serial interface. The serial interface allows the LCD to be controlled by one to three PIC pins rather than the seven or eleven needed by a standard interface. Serial interface controllers often provide additional features, as well. However, serial LCDs frequently cost five to ten times as much as their parallel counterparts. Since there is no "standard" serial interface, operations like clearing the display will vary between serial LCDs.

The LCD Electrical Interface

Introduction

LCDs have fairly standardized interfaces. There are 14 pins for LCDs without a backlight. Backlit LCDs add two more pins for the backlight. Sometimes these two pins are 15 and 16 on the same connector as the data. Sometimes they are on a separate connector. Sometimes non-backlit LCDs have these two extra pins but they are unused. Eight of the pins are used for the eight data bits. There are pins for power and ground, and a pin for a voltage to control the contrast. Finally, there are pins for selecting whether the data represents data to be displayed or a command, a pin to select reading or writing, and a pin to strobe data into the LCD.



The pin spacing is typically 0.1", but smaller LCD modules sometimes use 0.05" spacing. When the connector is arranged as a 14x1 array, the assignments shown are universal. Sometimes, when the pins are arranged as 7x2, the assignments are different, but typically they have the same meaning as the corresponding pin numbers on a 14x1 connector.

Levels and Timing

All of the LCD data lines use standard CMOS logic levels. The LCD controller is a small microprocessor, and it is thus important not to send data to the controller faster than it can be processed. With the exception of the enable pulse, there is no such thing as too slow. Even the enable pulse can be quite wide, but there is a maximum risetime specified for the enable pulse (140 ns).

The enable pulse must be at least 450 ns wide. There must be at least one microsecond between the rising edge of successive enable pulses. Once the data for a command has been sent to the controller, it is necessary to wait long enough for the command to be processed. This depends on the command but is on the order of a few milliseconds.

Shared Pins

The controller ignores data until the enable pin is raised. As a result, the other LCD lines may be connected to PIC pins that can have other uses. On the PIC-EL, for example, these pins are shared with the DDS Daughtercard and the LEDs.

The HD44780 Controller

Introduction	In dealing with the LCD, we need to keep in mind the capability of the HD44780 controller, and to understand its view of the world.																																																																																	
Controller Memory	<p>The HD44780 controller has 128 bytes of character memory, called DDRAM. At reset, the first byte of this memory is mapped to the first character on the display. If the display has more than one line, the 65th byte of this memory is mapped to the first character of the second line of the display.</p> <p>The processor, however, has the ability to change the relative positions of the memory against the display, so after some characters have been written, this mapping might no longer hold. Further, the LCD manufacturer’s idea of what constitutes the “second line” is often surprising. On 16x1 displays the second line most commonly the rightmost 8 characters of the display.</p> <p>In addition to the DDRAM, the controller includes CGRAM. CGRAM allows the user to define graphics for a few special characters. The dot patterns for the characters are loaded into CGRAM and will be displayed when the “character” to be displayed matches the index of the user specified graphic.</p>																																																																																	
LCD Commands	<p>The LCD commands must generally fit within 8 bits. However, some commands are going to take some data. The various LCD commands are differentiated by the first non-zero bit. The bits to the right of the first one bit comprise the data for the command, if needed.</p> <p>The following drawing shows the various commands:</p> <table style="border-collapse: collapse; margin-left: 20px;"> <tr> <td style="text-align: center; padding: 5px;">7</td><td style="text-align: center; padding: 5px;">6</td><td style="text-align: center; padding: 5px;">5</td><td style="text-align: center; padding: 5px;">4</td><td style="text-align: center; padding: 5px;">3</td><td style="text-align: center; padding: 5px;">2</td><td style="text-align: center; padding: 5px;">1</td><td style="text-align: center; padding: 5px;">0</td><td></td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td>Clear Display</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td style="background-color: #ccccff; border: 1px solid black;"></td><td>Return Home</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">D</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">S</td><td>Entry mode set</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">D</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">C</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">B</td><td>Display on/off</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">S</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">D</td><td style="background-color: #ccccff; border: 1px solid black;"></td><td style="background-color: #ccccff; border: 1px solid black;"></td><td>Cursor or</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">D</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">N</td><td style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">L</td><td style="background-color: #ccccff; border: 1px solid black;"></td><td style="background-color: #ccccff; border: 1px solid black;"></td><td>Function ser</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">0</td><td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td colspan="6" style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">← address →</td><td>CGRAM address</td> </tr> <tr> <td style="border: 1px solid black; text-align: center; padding: 2px;">1</td><td colspan="7" style="background-color: #ffcc99; border: 1px solid black; text-align: center; padding: 2px;">← address →</td><td>DDRAM Address</td> </tr> </table> <p style="margin-left: 20px;"> Data or parameter Don't care bit </p> <p>The first two take no data. The rest accept varying amounts of data.</p>	7	6	5	4	3	2	1	0		0	0	0	0	0	0	0	1	Clear Display	0	0	0	0	0	0	1		Return Home	0	0	0	0	0	1	D	S	Entry mode set	0	0	0	0	1	D	C	B	Display on/off	0	0	0	1	S	D			Cursor or	0	0	1	D	N	L			Function ser	0	1	← address →						CGRAM address	1	← address →							DDRAM Address
7	6	5	4	3	2	1	0																																																																											
0	0	0	0	0	0	0	1	Clear Display																																																																										
0	0	0	0	0	0	1		Return Home																																																																										
0	0	0	0	0	1	D	S	Entry mode set																																																																										
0	0	0	0	1	D	C	B	Display on/off																																																																										
0	0	0	1	S	D			Cursor or																																																																										
0	0	1	D	N	L			Function ser																																																																										
0	1	← address →						CGRAM address																																																																										
1	← address →							DDRAM Address																																																																										

Details of the LCD commands

Introduction

We explain our intention to the LCD by sending it a sequence of commands. Some commands, like Clear, take some action immediately. Others determine how the LCD will respond when it receives data.

LCD command set

Earlier we outlined the commands that can be sent to the LCD. In the LCD library sources there is a file, LCDmacros.inc, which contains the bit patterns for the various commands. As part of the LCD initialization we need to send a number of commands. Using the symbols defined in LCDmacros.inc will make the code a little more obvious.

Many of the LCD commands have parameters. Different commands have different numbers of possible parameters. To make the commands as short as possible, they are arranged so that the command portion is a variable length, leaving a different length parameter portion for each command.

Command	Code	Parameters
Set DDRAM address	80	Address
Set CGRAM address	40	Address
Function Set	20	1 or 2 line, character size, data length
Shift	10	Display or Cursor, Left or Right
Display Control	08	Display on/off, cursor on/off, blink on/off
Entry mode	04	Shift or not, increment or decrement
Address and cursor home	02	None
Clear display	01	None

Function Set Command

The LCD Function Set command is generally set only once. It controls some hardware aspects of the display, and is therefore dependent on the particular display and how it is wired, so it is not something that needs to be changed frequently.

The function set command establishes whether we want to communicate with the LCD 4 or 8 bits at a time. In the case of the PIC-EL, we have only four bits connected, so that answer is fairly obvious.

Function set also determines whether the display is a one or two line display. Since this is also totally dependent on the hardware, the answer is also pretty obvious.

Continued on next page

Details of the LCD commands, Continued

Function Set Command (continued)	<p>The HD44780 controller includes a 5x7 character generator, but there are a few locations for user defined characters. If we set the character size to 5x10, we will be able to make these characters a little larger. Also, the underline cursor will be below the character instead of covering the last line. However, depending on the hardware, setting 5x10 may cause some cosmetic glitches.</p> <ul style="list-style-type: none"> • Bit 2 – 5x7 or 5x10 dot font • Bit 3 – Single line or two line display • Bit 4 – 4 or 8 bit data length
Entry Mode command	<p>The Entry Mode command determines how the display will be modified to accept future characters. The cursor may be moved after each character, or the memory may be shifted left or right under the character:</p> <p>The entry mode command allows us to be somewhat flexible with how we manage the display. We would normally expect a new character to be placed at the cursor position, and then the cursor position incremented by one to prepare for the next character.</p> <p>However, the entry mode command allows us to decrement the cursor position (move the cursor to the left) if we wish. Also, instead of moving the cursor, we can choose to move the display memory with respect to the display itself, giving the appearance of scrolling the message across the display. Typically, we would set this mode once and never change it, unless we wanted some special display effects.</p> <ul style="list-style-type: none"> • Bit 0 – Shift/No shift • Bit 1 – Increment/Decrement
Shift command	<p>The cursor or display shift command causes either the cursor to move one position, or the characters on the display to be shifted one position. Unlike the entry mode commands which determine what will happen when a character is sent, the cursor or display shift command simply performs the action:</p> <p>The shift command takes options very similar to the entry mode command. The difference is that the shift command simply takes the action, that is, shifts the display or the cursor, rather than configuring the LCD to take the action for each character.</p> <ul style="list-style-type: none"> • Bit 2 – Right/Left • Bit 3 – Display/Cursor

Continued on next page

Details of the LCD commands, Continued

Display Control	<p>The display command controls whether the display is visible, whether the cursor will be visible, and whether the character under the cursor will blink:</p> <p>The display control allows us to turn the display off without resetting it's contents. It also allows us to set whether the cursor will be visible or not, and whether the character at the cursor position will blink.</p> <ul style="list-style-type: none"> • Bit 0 – Blink on/off • Bit 1 – Cursor on/off • Bit 2 – Display on/off <p>'Blink' actually blinks the character cell in reverse video, so it gives the appearance of a blinking black block cursor.</p>
Clear	The LCD clear command erases the LCD display RAM.
Home	The Home command sets the cursor to address zero, and aligns display memory address zero with the upper left hand character of the display.
Set CGRAM address	Sending the set CGRAM address command causes subsequent data to be placed in the character generator RAM, rather than the display RAM.
Set DDRAM address	<p>Sending the set DDRAM address causes the next character sent to be placed at the display RAM address specified. Notice that if there has been some shifting prior to sending this command, the address might not match up with the character location on the display.</p> <p>There are 128 DDRAM locations. Very few displays actually can display 128 characters, so at least some of the locations are invisible until a shift is performed.</p>

Sending Data to the LCD

Introduction	Since the LCD contains an onboard processor, the HD44780, controlling the LCD involves communicating with the onboard processor. Operations must be performed in a particular sequence.
Sending the Data	Conceptually, sending the data is relatively straightforward. The LCD has 8 data lines to accept a byte of data. The data is placed on the data lines. The Register Select line is either raised or lowered depending on whether the data to be sent is a character to be displayed or a command to the LCD. The Enable line is then raised for a specific amount of time, then lowered. Finally, the LCD's processor must be given enough time to perform the action requested.
Four-bit Data	<p>The LCD can accept data a full byte at a time, or it can accept the byte as two, four-bit nybbles. Although sending data four bits at a time is a little more complex, it saves four output pins on the PIC. Because of this, the four bit data transfer is the more common way of dealing with the LCD.</p> <p>In addition to being more complex, sending data four bits at a time is obviously slower than eight bits. However, the actual LCD display takes a very long time, at least in PIC terms, to actually reflect a change on the display, so this performance hit is rarely meaningful.</p>
Processing Delays	The LCD has a busy flag which can be read, but reading the flag is a multiple step operation. Rather than reading the flag, most applications simply wait enough time for the command to complete. Different commands can take differing amounts of time, but there are only a handful of different times, so providing a few different delay times is not a large burden.

Regression Testing

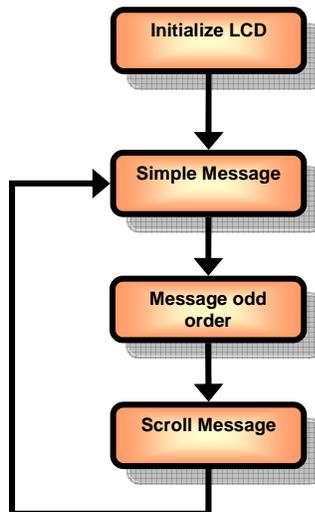
Introduction

When programmers manage an application through a series of changes or over a period of time, they use a technique called *regression testing*. The idea is that you write a program, or series of programs, called a *test harness*, that exercises all the functions of the code under test. As changes are made, this test program is run after each change to make sure that nothing was broken by the changes.

Since a number of things have to happen right with an LCD before it gives us any kind of satisfying behavior, we will use this technique to understand how the LCD behaves. We will write a fairly elaborate program to exercise the LCD using the LCDlib from Lesson 16. Then, we will change out the routines in the library with our own versions, running the regression test each time. Eventually we will end up with code to perform all the major LCD functions.

Flowchart

We won't replace every routine in the LCD library, but we will try to cover all the major functions. Our program, then, must initialize the LCD, display a simple message on the LCD, display a message that requires controlling the position we place letters on the LCD, and scroll a message across the LCD. Between each message, we will want to clear the display. In this way, we will cover all the major LCD operations:



Setting up the project

Set up a project, `Less17`, and create four empty `.asm` files, one for the mainline, `Less17.asm`, and one for each of our message routines, creatively named `L17msg1.asm`, `L17msg2.asm` and `L17msg3.asm`. Also, copy `16f84a.lkr` to the project directory and rename it `Less17.lkr`. Add all five files, plus `LCDlib84a.lib`, to the project. Don't forget to check that the PIC16F84A is selected under `Configure->Select Device...`

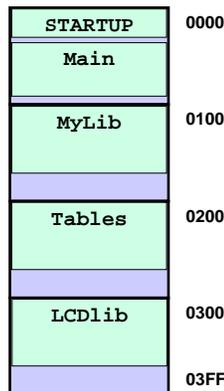
Continued on next page

Regression Testing, Continued

Thinking about the linker script

As we go through the development of our own LCD routines, we will be making a lot of changes, and looking at the map file a lot. It would be helpful if we managed the placement of things in memory, rather than just letting the linker place things where it would like. Notice that this step isn't absolutely necessary, but it will make things a little easier for us.

We will have our main program and the LCD library. We will also have our replacement routines for the LCD library. Finally, we will probably store our messages in lookup tables, and it is helpful to place these at the beginning of a memory page so we can have as many messages as possible before we cross a page boundary.



This will leave a lot of holes in memory of various sizes. Since we don't plan to use much of the F84's memory, this isn't a big issue. However, later on the experimenter is likely to want to do a fair amount of tinkering. Depending on what parts grow, some adjustment of these areas might be needed.

Editing the Linker Script

Double-click `Less17.lkr` in the project window which will open up the script for editing. We want to break the codepage page:

```
CODEPAGE NAME=page START=0x5 END=0x3FF
```

Into four parts:

```
CODEPAGE NAME=page START=0x5 END=0xff
CODEPAGE NAME=mylib START=0x100 END=0x1ff PROTECTED
CODEPAGE NAME=tables START=0x200 END=0x2ff PROTECTED
CODEPAGE NAME=lcdlib START=0x300 END=0x3FF PROTECTED
```

We added the `PROTECTED` attribute to keep the linker from assigning unused space in those code pages to other uses. Now we want to add sections for each of the three new codepages toward the end of the linker scripts. The section names will be visible from our program:

```
SECTION NAME=MYLIB ROM=mylib // Replacement library
SECTION NAME=TABLES ROM=tables // Message tables
SECTION NAME=LCDLIB ROM=lcdlib // LCD library
```

Our main routine will go into the default code section, but each of the other parts will go into specific, named sections.

Writing the Test Harness

Introduction

We will do one message at a time, and test each one, rather than do a whole lot of work with no result. The mainline, though, we will completely write. Open L17msg2.asm, and fill it with a dummy for now:

```

                                global      Msg2
                                code
Msg2
                                return
                                end

```

Do the same for L17msg3.asm.

The mainline

In the mainline, we want to initialize the LCD, then call Msg1, Msg2, and Msg3, and then do it again. However, we want to wait a little bit for the user to see the display, and then clear the LCD after each test. This means we will need to declare Msg1, Msg2, and Msg3 as extern, as well as LCDinit, LCDclear, and Dells.

```

                                include      p16f84a.inc
__config                        _XT_OSC & _WDT_OFF & _PWRTE_ON

                                extern      Msg1,Msg2,Msg3
                                extern      LCDinit,LCDclear
                                extern      Dells

```

Just as we did in Lesson 16, we need to declare a STARTUP section and place a goto Start in it, and then put our label, Start, in a new, unnamed section. Then we can add our (relatively obvious) code:

```

STARTUP                          code
                                goto      Start

                                code
Start
                                call      LCDinit      ; Initialize the LCD

Loop
                                call      Msg1          ; Display the 'Pig' message
                                call      Dells         ; Wait a second to see it
                                call      LCDclear      ; Clear the LCD

                                call      Msg2          ; Display the 'Elecraft' message
                                call      Dells         ; Wait a second to see it
                                call      LCDclear      ; Clear the LCD

                                call      Msg3          ; Display the 'Watson' message
                                call      Dells         ; Wait a second to see it
                                call      LCDclear      ; Clear the LCD

                                goto      Loop         ; Do it again

                                end

```

We will go ahead and write the first message routine, but on our first test there will be a three second delay with the LCD blank since we haven't done the other two messages. Three seconds isn't a horribly long wait.

Continued on next page

Writing the Test Harness, Continued

The first message routine

In the `Msg1` routine, we simply want to display a message on the LCD. It is convenient to put the message in a table, as we have done several times before. However, this time, we will add two embellishments to the table. First, if you noticed in the linker script, we have put the tables up at `H'0200'`. In the past, we have always put tables in addresses under `H'0100'`. This means that we will need to set `PCLATH` before we do the lookup.

Our second little addition is that we will end our message with a zero byte. This allows us to test for the end of the message in our code instead of counting bytes. Instead of getting the index passed in through the `W` register, we will read it from a memory location. The caller will need to keep a memory location anyway, and since the table will be used by only a single routine, this isn't really much of a loss.

We need to declare `Msg1` as global, but we don't need to set the configuration word since that was done in the mainline:

```

        include p16f84a.inc

        global Msg1
        extern LCDletr

        udata
MsgIdx  res    1                ; Counter for message index

;      Table containing the message to display

TABLES code
Msg1T   movlw  high Msg1Ts      ; Pick up high byte of table address
        movwf  PCLATH          ; and save into PCLATH
        movf   MsgIdx,W        ; Pick up index
        addwf  PCL,F           ; And look up in table
Msg1Ts  dt     "MultiPig",0    ; Message, terminated with a zero

```

There are a couple of interesting things going on here.

Notice that we placed the code in the section `TABLES`, which matches the name we gave in our linker script file to the section at `H'0200'`.

The first thing we do is move a literal, `high Msg1Ts`, to the `W`. The `high` operator takes the high byte of a two byte value. Later we will see that `Msg1Ts` ends up at around `H'0204'`, so this is the same as a `movlw H'02'`. However, by coding it this way, we can later edit the linker file to put the tables elsewhere and not have to change our code.

Now, we move that 2 to `PCLATH`. `PCLATH`, which stands for **Program Counter LATch High**, forms the high order part of the address whenever we change the program counter. Thus, when we `addwf PCL,F` the address formed will be the current program counter low byte, which will be the low byte of the address of

Continued on next page

Writing the Test Harness, Continued

The first message routine (continued)

Msg1Ts with a high byte of 02. Had we not done this, the next address executed after the `addwf PCL,F` would have used the current value of `PCLATH` which might have been anything.

We have chosen an 8-character message to fill the 8 character display. This will use the left 8 characters if you have a 16 character display.

Now, to display this, we need to clear the `MsgIdx` location, then call `Msg1T` followed by a call to `LCDletr` (as we did in lesson 16), then increment the index, and repeat the process until the table routine returns a zero

```

                code
Msg1   clrf     MsgIdx      ; Clear out the message index
Msg1L  call    Msg1T      ; Go get the character
        xorlw  H'00'      ; Test to see if it was a zero
        btfsc STATUS,Z   ; Was it?
        return ; Yes, all done
        call  LCDletr    ; Display the letter on the LCD
        incf  MsgIdx,F   ; Point to the next letter
        goto  Msg1L     ; And go do it again

```

The `retlw` instruction in the table routine doesn't affect the `Z` flag, so we use an `xorlw` instruction to set the `Z` bit without affecting the contents of `W`. Other than that, this routine should be fairly self-explanatory.

Testing the first message

At this point, if we assemble the program and load it into our PIC-EL, we should see the message "MultiPig" for about a second, then the display should clear for about three seconds, and the process repeats.

The second message

The second message is a little more interesting. We want to thoroughly test the `LCDaddr` routine, which positions the cursor to the address specified in the `W`. This allows us to place characters wherever we want on the display.

To do this, we will have two tables. One table looks up the position of the character to be written. The second table contains the actual letters to be displayed. A little quirk here is that our position table needs to have one extra position to pick up the zero at the end of the text:

```

TABLES          code
;
; Table containing the order in which to display the message
Msg2P           movlw  high Msg2Ps   ; Pick up high byte of table address
                movwf  PCLATH      ; And save into PCLATH
                movf   PosIdx,W    ; Pick up index
                addwf  PCL,F       ; And look up in table
Msg2Ps          dt     7,0,6,1,5,2,4,3,8 ; Position for each character
                ; Last pos 8 to move cursor off LCD
;
; Table containing the message to display
Msg2T           movlw  high Msg2Ts   ; Pick up high byte of table address
                movwf  PCLATH      ; And save into PCLATH
                movf   MsgIdx,W    ; Pick up index
                addwf  PCL,F       ; And look up in table
Msg2Ts          dt     "Elecraft",0 ; Message, terminated with a zero

```

Continued on next page

Writing the Test Harness, Continued

The second message (continued)

The body of the message display function is almost the same as `Msg1`, except that we call the `Msg2P` table first, save the returned value, then set the cursor position with `LCDaddr`, before calling `Msg2T` to look up the character to display.

Also, after we display each character we will wait a quarter second so that the display builds slowly enough that we can see it. The `Del256ms` routine from the LCD library delays about 256 ms.

```

;      Subroutine to display message in a strange order
code
Msg2   clrfs   PosIdx      ; Clear out the message index
Msg2L  call    Msg2P       ; Get the position
       movwf  MsgIdx     ; Remember which char to pick up
       call  LCDaddr    ; And position the cursor
       call  Msg2T     ; Go get the character
       xorlw H'00'     ; Test to see if it was a zero
       btfsc STATUS,Z ; Was it?
       return        ; Yes, all done
       call  LCDletr   ; Display the letter on the LCD
       call  Del256ms ; Wait a bit to see it
       incf  PosIdx,F  ; Point to the next letter
       goto  Msg2L    ; And go do it again

```

Of course, we need to add the appropriate extern and global statements.

Testing Msg2

When we run our test program this time, we should see our “MultiPig” message, and then an “Elecraft” message. However, the Elecraft message will build slowly from the outermost characters inward.

The third message

The final LCD behavior we would like to exercise is scrolling a message. For this experiment, we will use code almost the same as message 1. However, we want a longer message, and we will position the cursor past the right end of the display before we display the message. We will also set the LCD into scrolling mode using the `LCDshift` routine from the library. When we are done, we will turn off the LCD scrolling by calling `LCDunshf`. As in message 2, we will delay a little after each character to prevent the display from being filled too quickly.

The table:

```

TABLES code
Msg3T  movlw  high Msg3Ts ; Pick up high byte of table address
       movwf PCLATH     ; And save into PCLATH
       movf  MsgIdx,W   ; Pick up index
       addwf PCL,F      ; And look up in table
Msg3Ts dt   "Watson, come here please.      ",0

```

The trailing spaces will cause the message to shift off the end of the display.

Continued on next page

Writing the Test Harness, Continued

The third message (continued)

And now the display code:

```

Msg3      code
          clrfs      MsgIdx      ; Clear out the message index
          call      LCDshift     ; Set the LCD to shift mode
          movlw     H'08'        ; Position to the right end
          call      LCDaddr      ; of the LCD display
Msg3L     call      Msg3T        ; Go get the character
          xorlw     H'00'        ; Test to see if it was a zero
          btfsc     STATUS,Z     ; Was it?
          goto     Msg3Q        ; Yes, all done
          call      LCDletr      ; Display the letter on the LCD
          call      Del256ms     ; Wait so message isn't total blur
          incf     MsgIdx,F      ; Point to the next letter
          goto     Msg3L        ; And go do it again
Msg3Q     call      LCDunshf     ; Leave shifting turned off
          return

```

Notice that the addresses visible on the 8 character display are 0 through 7, so 8 is the character past the right end. However, as soon as the character is written, the entire display is shifted to the left one character causing the last character written to be displayed.

Testing the final version

After adding the necessary extern and global definitions, and reserving GPR space for the one variable needed, and assembling and loading the program, the first two messages will remain unchanged, but the third message will scroll across the LCD from the right to the left.

In all cases, we have written the routines for the eight character display. The routines will need a little extra work for the 2 line by 8 character LCD that shipped with later PIC-ELs. However, for 1x20 or 1x40 LCDs, the messages could be lengthened but no other changes would be needed.

On multi-line LCDs, these tests only exercise the first line. For the newer PIC-ELs this means the display will be constrained to the left 8 characters.

Reviewing the map

Introduction

As we build out our own versions of the LCD routines, we will be constantly referring to the map to validate that the library versions have been replaced with the new versions.

Symbols – Sorted by Address

The section of the map most interesting for this exercise is the last section, sorted by address. Because of the work we did to the linker script, we expect our program to be spread among four memory areas.

The first part contains our test harness:

Start	0x000007	program	static C:\Projects\PIC\Lesson17\Less17.asm
Loop	0x000008	program	static C:\Projects\PIC\Lesson17\Less17.asm
Msg1	0x000012	program	extern C:\Projects\PIC\Lesson17\L17msg1.asm
Msg1L	0x000013	program	static C:\Projects\PIC\Lesson17\L17msg1.asm
Msg2	0x00001a	program	extern C:\Projects\PIC\Lesson17\L17msg2.asm
Msg2L	0x00001b	program	static C:\Projects\PIC\Lesson17\L17msg2.asm
Msg3	0x000026	program	extern C:\Projects\PIC\Lesson17\L17msg3.asm
Msg3L	0x00002a	program	static C:\Projects\PIC\Lesson17\L17msg3.asm
Msg3Q	0x000032	program	static C:\Projects\PIC\Lesson17\L17msg3.asm

The second group contains our tables:

Msg1T	0x000200	program	static C:\Projects\PIC\Lesson17\L17msg1.asm
Msg1Ts	0x000204	program	static C:\Projects\PIC\Lesson17\L17msg1.asm
Msg2P	0x00020d	program	static C:\Projects\PIC\Lesson17\L17msg2.asm
Msg2Ps	0x000211	program	static C:\Projects\PIC\Lesson17\L17msg2.asm
Msg2T	0x00021a	program	static C:\Projects\PIC\Lesson17\L17msg2.asm
Msg2Ts	0x00021e	program	static C:\Projects\PIC\Lesson17\L17msg2.asm
Msg3T	0x000227	program	static C:\Projects\PIC\Lesson17\L17msg3.asm
Msg3Ts	0x00022b	program	static C:\Projects\PIC\Lesson17\L17msg3.asm

The final code section is for the LCD library:

Del1s	0x000300	program	extern H:\PIC\LCDlib\Del1s.asm
Del1256ms	0x000303	program	extern H:\PIC\LCDlib\Del1256ms.asm
Del128ms	0x000306	program	extern H:\PIC\LCDlib\Del128ms.asm
outer_loop	0x000309	program	static H:\PIC\LCDlib\Del128ms.asm
inner_loop	0x00030b	program	static H:\PIC\LCDlib\Del128ms.asm
Del1512ms	0x000310	program	extern H:\PIC\LCDlib\Del1512ms.asm
LCDaddr	0x000313	program	extern H:\PIC\LCDlib\LCDaddr.asm
Del2ms	0x00031c	program	extern H:\PIC\LCDlib\Del2ms.asm
lloop	0x00031e	program	static H:\PIC\LCDlib\Del2ms.asm
mloop	0x00031f	program	static H:\PIC\LCDlib\Del2ms.asm
Del40us	0x000324	program	extern H:\PIC\LCDlib\Del40us.asm
jloop	0x000326	program	static H:\PIC\LCDlib\Del40us.asm
kloop	0x000327	program	static H:\PIC\LCDlib\Del40us.asm
LCDclear	0x00032f	program	extern H:\PIC\LCDlib\LCDclear.asm
LCDinit	0x000333	program	extern H:\PIC\LCDlib\LCDinit.asm
_LCDLIB_0008	0x00033b	program	static H:\PIC\LCDlib\LCDinit.asm
LCDdig	0x00034b	program	extern H:\PIC\LCDlib\LCDletr.asm
LCDletr	0x00034d	program	extern H:\PIC\LCDlib\LCDletr.asm
LCDsend	0x000355	program	extern H:\PIC\LCDlib\LCDsend.asm
LCDshift	0x00035e	program	extern H:\PIC\LCDlib\LCDshift.asm
LCDsndD	0x000362	program	extern H:\PIC\LCDlib\LCDsnd.asm
_LCDLIB_0002	0x000364	program	static H:\PIC\LCDlib\LCDsnd.asm
LCDsndI	0x000365	program	extern H:\PIC\LCDlib\LCDsnd.asm
LCDunshf	0x00036c	program	extern H:\PIC\LCDlib\LCDunshf.asm

There is nothing in H'01xx' because we assigned that to MyLib, and we haven't put anything in that section yet. As we develop our own routines, we will add to that section, and take code from the LCDlib section.

The structure of LCDlib

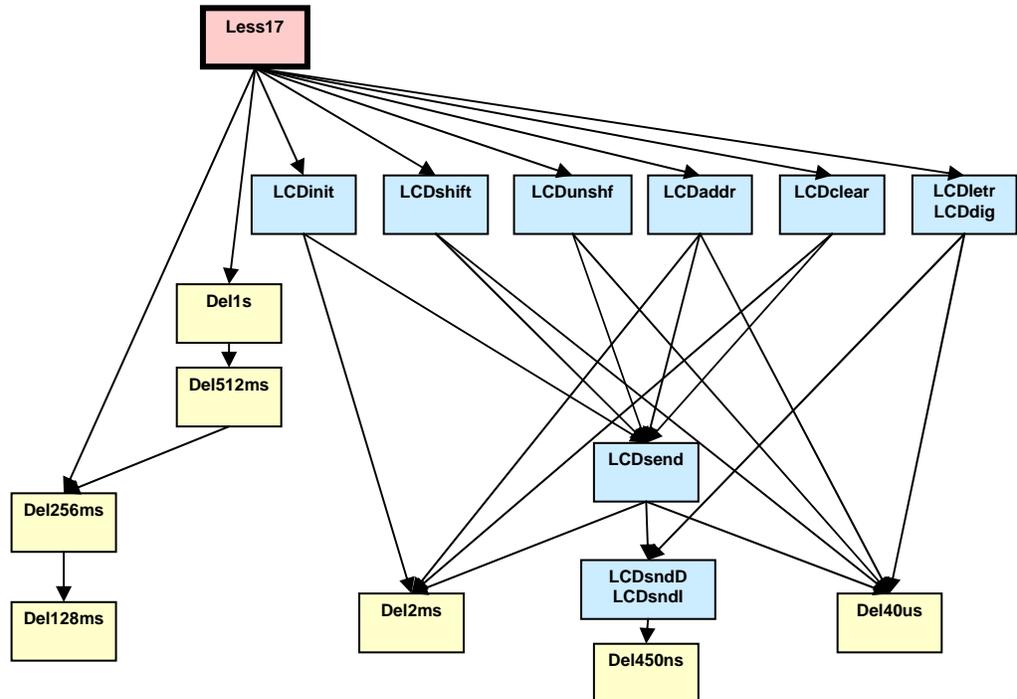
Introduction

Looking at the map, we see a lot of routines gathered from the library. Back in Lesson 16 we noticed a few routines, but now that we have called a few more library routines, there are still more routines that we never asked for.

LCDlib call tree

As we mentioned, routines in the library call other routines in the library, and the linker takes care of collecting whatever we need, and nothing more.

The routines in the map are connected as follows:



The delay routines are shown in yellow, and the LCD routines in blue. We will replace each of the LCD routines with our own code in the following pages. We will not replace the delay routines, as there is little learning there. The student may wish to provide his own delay routines to allow for later experimentation with less than prime LCDs which might not be quite up to specification.

What do they do?

We have already seen the routines in the top blue row, from LCDinit to LCDletr. LCDdig is another entry point in LCDletr which takes a value in W from 0 to 9, converts it to ASCII, and falls through to LCDletr.

LCDsend is almost the same as LCDletr in that it sends a byte to the LCD one nybble at a time. It differs from LCDletr in that it sends a command byte instead of a character.

LCDsndD and LCDsndI each send a nybble to the LCD. LCDsndD sends a data nybble and LCDsndI sends a command (instruction) nybble.

Looked at this way, all those LCD routines sort of make sense.

Continued on next page

The structure of LCDlib, Continued

Planning our attack

We want to replace all the LCD routines (the blue ones) with our own code. We won't deal with the various delay routines since we are already experts at wasting time!

It probably makes the most sense to start at the bottom. `LCDsndD` and `LCDsndI` are practically the same routine, indeed, they are in the same file because they happen to share a good fraction of their code. These two routines send a half-byte to the LCD. `LCDsndD` sends a character nybble and `LCDsndI` sends a command nybble.

Next up the food chain are `LCDsend`, and `LCDletr/LCDDig`. Like `LCDsndD` and `LCDsndI`, `LCDletr` and `LCDDig` share a lot of code and so are in the same file.

`LCDclear`, `LCDaddr`, `LCDshift` and `LCDunshf` are actually all very similar. We will do these next.

Finally, we will address `LCDinit`, perhaps the most complex of all the LCD routines.

Recall from lesson 16, that the linker looks through the files in our project before it looks in the library. Because of this behavior, we can simply add files to our project and if they contain symbols that are already in the library, the linker will choose *our* versions instead of the library version.

As we add routines, we can keep checking back with the map. Since we will be putting our routines in their own code segment that we called `MYLIB`, that segment should start growing as `LCDLIB` shrinks. Besides showing up in a different segment, our routines will also have different file names, so we can be confident which version of, say, `LCDletr` we are using.

LCDmacs.inc

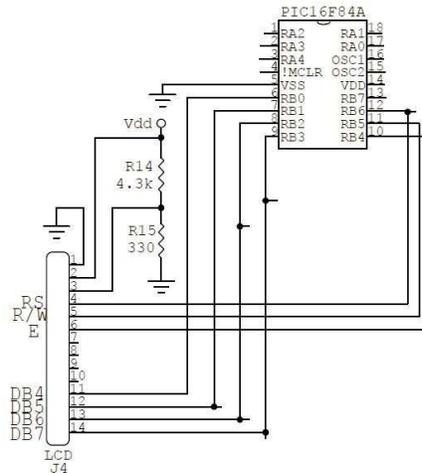
We will use an include file from the LCD library called `LCDmacs.inc`. This file provides symbols for the various bits in the LCD command, as well as providing a place to keep some hardware characteristics. If we do it right, you should be able to re-use your LCD routines on future projects, even if you wire the LCD differently.

`LCDmacs` uses an include file, `Processor.inc`. This is nothing more than a convoluted way of including `P16F84A.inc`. `Processor.inc` allows the library to be assembled for other PICs without editing any of the files. We can leave this file in there, but if you feel uncomfortable with that, replace `Processor.inc` in `LCDmacs` with `p16f84a.inc`. As long as you stick to the 16F84A, it won't make any difference.

Sending a nybble

Introduction

Since the LCD is wired to the PIC with only four data lines (and three control lines), data we send to the PIC has to go a nybble at a time. If we examine the schematic for the PIC-EL:



We can see that the data goes on RB0-RB3, RB4 is the enable line, RB5 read/write, and RB6 is Register Select. So the process for sending the nybble will be to prepare a byte with the data in the low four bits, and bit 6 set depending on whether the data is a command or a character. Then that data will be placed on PORTB. We will then raise bit 4, wait 450 nanoseconds or more, and then lower bit 4.

Adding a routine

For this routine, we will create a new .asm file, and add it under Source Files to our project. What we call it doesn't matter, as long as the name is unique. We've done this sort of thing plenty of times before, so in the future we won't mention it. If it's time for a new routine, add a new file. If you think that the new routine logically belongs with an existing file, add it to that file.

MySnd logic

We need to replace two routines because in the library, they are in the same file. So, our logic needs to look something like:

LCDsndD:

```
Mask off high order bits
Turn on Register Select Bit
Goto Send
```

LCDsndI:

```
Mask off high order bits
Goto Send
```

Send:

```
Move the data to the LCD port
Turn on LCD Enable
Wait 450ns (or more)
Turn off LCD Enable
Wait 450ns
Return
```

Relatively simple. Now, what does that look like in code?

Continued on next page

Sending a nybble, Continued

MySnd logic (continued)

As we said before, we will use the symbols in LCDmacs for the values of the various bits. That way we can move our routines to another project easily. Here we care about the symbols LCDRS, LCDEN, and LCDPORT.

```

include          "LCDMac.inc"

; Sends a nybble to the LCD.  Two entry points are provided, LCDsndI to
; send a command nybble, LCDsndD to send a data nybble.

; Provided Routines
global LCDsndI      ; Send a command nybble to the LCD
global LCDsndD      ; Send data to the LCD
; Required routines
extern Del450ns      ; Delay 450 nsec

MYLIB  code
; -----
; Send data to the LCD
LCDsndD:
    andlw  00fh          ; only use low order 4 bits
    iorlw  LCDRS         ; Select register
    goto   Send          ; Skip over LCDsndI

; -----
; Send a command nybble to the LCD
LCDsndI:
    andlw  00fh          ; only use low order 4 bits
; FALL THROUGH to Send

; -----
; Actually move the data
Send:
    movwf  LCDPORT       ; Send data to LCDPORT
    iorlw  LCDEN         ; Turn on enable bit
    movwf  LCDPORT       ; Send to port again
    call   Del450ns      ; 450ns
    xorlw  LCDEN         ; Turn off enable bit
    movwf  LCDPORT       ; Send to port yet again
    call   Del450ns      ; 450ns
    return
end

```

There are a couple of things to notice here. We used the subroutine `Del450ns`. We count on this routine not to change the `W`. For a 4 MHz processor, we could have simply used a `nop` instruction. In fact, 3 `nop`'s would be enough for a 20 MHz PIC. But keeping it in a separate subroutine does make it easier if we should encounter an especially slow LCD at a hamfest.

In addition, we could have used a `bsf` and `bcf` against the port rather than the `iorlw` and `xorlw`. This would have saved two instructions, and on the PIC16F84A is perfectly valid. However, when the port is shared with other peripherals, as it often is on more complicated PICs, these instructions can cause the peripherals to fail. Since we want to build a library to re-use, it is better not to build in these little traps that will bite us long after we have forgotten what we did in the library.

Continued on next page

Sending a nybble, Continued

Examining the map

If we test our program, it should still work. But did we use our version of LCDsndD, or the one in the library? The way to know for sure is to look at the map. If you recall, we placed our code in the section MYLIB which we had started at H'0100' in our linker script. Without our new function, there was no code in that section. Now, we would expect to see our new code there, and expect to see the functions missing from the LCDLIB section at H'0300'.

If we look at the portion of the map sorted by address we will see our routines between our table code and the tables themselves:

```
Msg3Q 0x000032 program static C:\Projects\PIC\Lesson17\L17msg3.asm
LCDsndD 0x000100 program extern C:\Projects\PIC\Lesson17\MySnd.asm
LCDsndI 0x000103 program extern C:\Projects\PIC\Lesson17\MySnd.asm
Send 0x000104 program static C:\Projects\PIC\Lesson17\MySnd.asm
Msg1T 0x000200 program static C:\Projects\PIC\Lesson17\L17msg1.asm
```

And if we look at the portion of the map sorted by name, we will see only one version of the routines, and that version is the one in our project:

```
LCDshift 0x000361 program extern C:\Projects\PIC\Lesson17\LCDlib\LCDshift.asm
LCDsndD 0x000100 program extern C:\Projects\PIC\Lesson17\MySnd.asm
LCDsndI 0x000103 program extern C:\Projects\PIC\Lesson17\MySnd.asm
LCDunshf 0x000365 program extern C:\Projects\PIC\Lesson17\LCDlib\LCDunshf.asm
```

What is happening?

At this point, we have taken control of the LCD. If you review the diagram on page 18 showing the relationships between all the LCD library routines, you can see that all the communications to the LCD are now going through our code.

Sending a command byte

Introduction

We indicated that we need to break each byte up into two nybbles to send to the LCD. We now have the routine to send a nybble, so the next logical step is to write the routine to send a byte. The data byte routine has two entry points, so let's do the command byte first, since it might be a little simpler.

The Logic

We will receive a command byte in the W register. We want to send the high order nybble first, then the low order. To do this, we will have to save off the original byte, move the high four bits into the low four, call `LCDsndI`, get back the original byte, then call `LCDsndI` again.

We could shift the byte four times to get the high four bits into the low order positions, but the `swapf` instruction swaps the two halves in one instruction. We also need a little delay between the calls to `LCDsndI` and a longer delay to allow the command to process.

```

MYLIB   code
LCDsend
    movwf   Save           ; Save off the incoming byte
    ; High byte
    swapf  Save,W         ; Get high nybble into low nybble of W
    call   LCDsndI       ; LCDsndI takes care of masking
    call   Del40us       ; 40us
    ; Low byte
    movf   Save,W         ; Grab the original value
    call   LCDsndI       ; And again, LCDsndI masks
    call   Del2ms        ; Delay to allow for processing
    return
    end

```

Of course, we have to reserve space for `Save` and identify our globals and externals:

```

global  LCDsend
extern  LCDsndI           ; Send a command nybble to the LCD
extern  Del40us          ; Delay 40 usec
extern  Del2ms           ; Delay 1.8 msec

    udata
Save   res    1

```

Building and Testing

After adding the new source to the project, one needs only to click the "Make" button and a new version of the program gets assembled including the new routine in place of the old. Loading and running the program should produce the same behavior as before, and a check of the map file should show that the `LCDsend` routine from the library has been replaced by our own.

Sending a Character

Introduction

Now that we have sent a command byte, sending a data byte should be pretty straightforward. However, if we carefully study the map there is a slight complication. The LCDletr routine is in the same source file as the LCDdig routine, so we need to replace both of them.

LCDletr simply takes a character. LCDdig takes a digit, 0-9. LCDdig needs to convert the 0-9 to the character H'30' through H'39', then call LCDletr.

The logic

Clearly, LCDdig can perform the conversion, then simply fall through to LCDletr. Other than calling LCDsndD instead of LCDsndI, the logic behind LCDletr will be pretty similar to LCDsend:

LCDdig:

Mask off high order bits
OR in a H'30'

LCDletr:

Save off the character
Swap the bytes
Call LCDsndD
Retrieve the saved character
Call LCDsndD
Wait
Return

The Code

After seeing the other routines, the code should be pretty rote by now:

```

                                udata
SaveLetr      res     1           ; Storage for letter

MYLIB
LCDdig:
                                andlw  00fh
                                iorlw  030h           ; note falls thru

LCDletr:
                                movwf  SaveLetr       ; save off the letter
                                swapf  SaveLetr,W     ; Swap bytes
                                call   LCDsndD

                                movfw  SaveLetr       ; get it
                                call   LCDsndD

                                call   Del40us       ; delay a while
                                return

```

Running and testing

After adding the necessary extern, global and include statements, adding the file to our project, and assembling, once again the behavior of the display shouldn't change. We now have done all the basics, the remaining routines we need to write send commands to the LCD.

Clearing the LCD

Introduction

In addition to characters, we can send commands to the LCD. The commands are simply values that we pass to the `LCDsend` routine instead of the `LCDletr` routine. The most used command is the command to erase the LCD.

The Logic

Erasing the LCD is quite straightforward. The 'clear' command is a H'01', but to make our code a little more readable, in `LCDmacs.inc` we have defined a constant, `LCD_DISP_CLEAR`. We need to wait a couple of milliseconds for the command to be processed by the display controller:

```
LCDclear:
            movlw      LCD_DISP_CLEAR
            call       LCDsend
            call       Del2ms
            return
```

That's all there is to it, now simply calling `LCDclear` will cause the display to be erased.

Setting the Cursor Address

Introduction

Sometimes we don't want to write letter after letter in sequence to the LCD. We may want to write just a part of the display. In this case, we need to set the LCD cursor address to begin writing at a specified character.

The Logic

Unlike clearing the display, setting the address requires that we send not only a command to the LCD, but also data; the address we desire. The `LCDaddr` routine is called with the desired address in the `W` register. The address may have a value between 0 and 127 (decimal). This leaves one bit, the high bit, for the command to tell the LCD that this byte contains an address. If you look at the figure on page 5, you will see that all we need to do is to take the seven bit address, and turn on the high bit;

```
LCDaddr:
    iorlw  LCD_SET_DDRAM ; OR in command byte
    call  LCDsend       ; Send to LCD
    call  Del2ms        ; Delay for processing
    return
```

Making the LCD Scroll

Introduction

The LCD can be commanded to scroll characters instead of moving the cursor. Typically, we want to scroll characters to the left, to make room for a new character added to the right. The LCD can scroll left or right, but we will only do a routine to scroll left.

The Logic

The command to set the LCD into scroll mode is very similar to clearing; we simply send a command byte to `LCDsend`. However, the “shift” command is actually a subcommand of the “set entry mode” command. In addition, we need to tell the LCD what direction to shift.

Of course, we have defined constants in `LCDmacs` for all these subcommands. We simply OR them together to send off to the LCD:

```
LCDshift:
    movlw   LCD_ENTRY_MODE | LCD_DISP_SHIFT | LCD_DIS_INCR
    call    LCDsend
    call    Del40us          ; Leave a little longer wait
    return
```

We could have simply sent a 7 (1+2+4) but that would have been a lot less readable than specifying the list of things we want to do. Notice that the required delay is a little shorter.

Turning off Scrolling

Introduction

After the LCD has been placed in scrolling mode, we may want to stop scrolling.

The Logic

This is getting old. Turning off scrolling is identical to turning on scrolling, we simply send different subcommands:

```
LCDunshf:
    movlw   LCD_ENTRY_MODE | LCD_NO_SHIFT | LCD_DIS_DECR
    call    LCDsend
    call    Del40us          ; Leave a little longer wait
    return
```

Initializing the LCD

Introduction	Before we can do any of the preceding, we need to initialize the LCD. This is by far the most complicated LCD operation, so we have saved it for last.
The Logic	<p>Before we can do anything with the LCD, we need to make it recognize that we have chosen to talk to it through the four bit interface, rather than eight. This is done by first, giving the LCD a chance to get its controller started, then sending three H'03' nybbles to the LCD which lets the LCD know we want to initialize it, followed by a H'02', which identifies the four bit mode.</p> <p>Before we can do that, however, we need to establish the PIC pins as outputs. Once the LCD has been set to four bit mode, we then want to set the entry mode, turn on the display, and identify the kind of cursor we want so that we are all ready to send characters.</p>
The Code	<p>The first thing we want to is set the port:</p> <pre> movlw H'80' ; Turn off low 7 bits errorlevel -302 ; Suppress message banksel LCDTRIS ; Now set the low 7 bits of andwf LCDTRIS,F ; LCDPORT to outputs banksel LCDPORT ; Back to bank zero and errorlevel +302 ; re-enable the error message </pre> <p>Then we want to hang around a while to give the LCD a chance to get through its internal initialization:</p> <pre> movlw 020h ; Need >15.1ms after 4.5V movwf Count ; we will wait 65ms (after 2V) call Del2ms ; in the case of LF parts) decfsz Count,F ; goto \$-2 </pre> <p>Next, we send the three 3's followed by a 2 to set the four bit mode:</p> <pre> movlw H'33' call LCDsend movlw H'32' call LCDsend </pre> <p>Finally we set the operating modes the way we want:</p> <pre> movlw LCD_FUN_SET LCD_DL_4 LCD_1_LINE LCD_5X7_FONT call LCDsend movlw LCD_DISPLAY LCD_DISP_OFF ; Display Off call LCDsend movlw LCD_DISP_CLEAR ; Display clear call LCDsend movlw LCD_ENTRY_MODE LCD_NO_SHIFT LCD_DIS_INCR call LCDsend movlw LCD_DISPLAY LCD_DISP_ON LCD_CURS_ON LCD_BLINK_ON call LCDsend </pre> <p>If we wanted to hide the cursor, we could have selected LCD_CURS_OFF LCD_BLINK_OFF for the final command. The LCD_CURS_... selects the underline cursor, the LCD_BLINK_... selects the blinking of the character under the cursor. Students with the 16 character display may prefer to set LCD_2_LINE.</p>

The 16 character display

Introduction

The original PIC-EL's shipped with an 8 character display. Later PIC-ELs had a 16 character display. Like most 16 character displays, this one is actually two lines by 8 characters.

In these experiments, everything has been done on the first eight characters of the display. This works on all PIC-ELs, but how to display characters on the right of the display?

Second Line Addressing

If we were to simply make the messages in the test harness longer, we would still only use the left eight characters of the display. The remaining characters are getting written to the LCD's DDRAM, but they are being written to addresses that aren't visible.

To address the second line of the display, we need to begin writing at address 64 (decimal). We could take our first message routine and do two messages, each 8 characters long. Between calls to those two routines, we could call `LCDaddr` with `H'40'` in the `W` register. Alternatively, we could modify our message routine to test for an index of 8, and call `LCDaddr` when that happens.

In the second message routine, it is a little more obvious. We always call `LCDaddr` so all we need to do is test whether we are calling it with an address of 8 or more, and add `H'40'` when that happens.

Scrolling is a little more interesting. The student is directed to the code in `LCDsc16`. This code is fairly complex, there are other strategies that might work better.

Additional Experiments

Introduction	At this point, all of the major LCD operations have been covered. However, there are a number of improvements the experimenter may wish to make.
Saving Memory	<p>A number of the LCD routines use some general purpose register storage. The GP registers can be a scarce resource, especially on a processor like the PIC16F84A, with only 56 locations. However, few of the routines call each other, so the storage could be shared.</p> <p>The experimenter may wish to use the techniques from Lesson 16 to reduce the general purpose register demands of the LCD routines.</p>
LEDs	<p>The LCD data lines are shared with the LEDs on the PIC-EL. This causes the LEDs to flash randomly whenever data is sent to the LCD. However, the LCD only pays attention to the data lines when the enable line is asserted. The experimenter could modify the send a byte routines (or the send a nybble routine) to turn off the LEDs when the operation is complete.</p> <p>An even more elaborate version might remember the LED condition and restore it on exit. This way the LEDs could be used even when the LCD is being used.</p>
Delay Routines	Most of the library delay routines calculate the number of cycles needed from the clock speed provided in <code>LCDmcs</code> . This may make these routines a little longer than they need to be. For certain processor speeds, program memory could be saved by making the delays specific to the particular processor speed. Obviously, this is a tradeoff against making the routines more portable to other projects.
Other Processors	<p>The <code>LCDinit</code> routine in the library is conditionalized to be used with a number of other 18 pin processors. These routines have been tested with the 16F628, 16F648A, 16F819 and 16F88. Unfortunately, the only one of these which is supported by the FPP software is the 16F628 (<u>not</u> the 16F628A).</p> <p>Students considering other projects might review what is handled differently on these other processors. All the listed chips can be programmed in the PIC-EL with other programming software. Some of the larger parts give the student an opportunity to study the paging issues when the program memory size exceeds 2K.</p>

Wrap Up

Summary

In this lesson, we have examined the differences in a number of common LCD modules and we have seen how to control them.

Coming Up

The next lesson will explore techniques for converting binary numbers to ASCII so they may be displayed on the LCD.
