Chapter 15: Writing to an LCD Text Display with Parallel Data Communication

Often in a project you will have a need to display information for a user. The easiest way to do this is to use one of the 1, 2 or 4 line text displays that are commonly available either new or removed for old medical equipment. These displays are available from a variety of sources including the following:

All Electronics (www.allelectronics.com)
BG Micro (www.bgmicro.com)
Mouser Electronics (www.mouser.com)
CrystalFontz (www.crystalfontz.com)

When searching for a display, look for one that has a Hitachi HD44780 controller chip. This will include most of the cheaper displays that use parallel data communication. All Electronics and BG Micro will likely have these for under $10. You may want to avoid the one line displays because the addressing is somewhat confusing. The one line display is really a two line display where the first half of the display is the first line and the second half of the display is the second line. So if you have a 16 x 1 display, you write the first eight characters and then you have to advance to the second line to write the second eight characters.

Assuming you are planning to use this in conjunction with your bread board, it would also be useful to have a display that has a single row of pins for electronic connections, since a double row will be more difficult to plug into a bread board. At this writing, All Electronics has a 16 character by 2 line display for $6.50 (LCD-129).

These displays will have a set of either 14 or 16 holes along the bottom of them which can be used to connect to your bread board (you only need to use the first 14 holes of a 16 hole strip). The easiest way to connect the display to your board will be to add a row of 14 pins (called a "header") spaced .1 inch. All Electronics sells a 40 pin header for $1.50 (DHS-40) which would be enough pins to supply 2 displays. Cut 14 pins off the end of the header and they should fit in the holes in the display. You will need to use a soldering iron and some solder to attach the pins in such a way as to make a solid electrical connection.

The holes will be numbered 1 through 14, though there may not be any indication on the display as to which pin is number 1. Often All Electronics has documentation for these display that show you which pin is number 1, but if documentation is not available, you can figure it out by knowing that pin number 1 is ground. Use an ohmmeter to find which end pin has zero resistance between the pin and mounting hardware on the display. Alternatively, the ground pin will almost always have a wider trace running into it than the pin on the other end of the row of holes.

As with the RS232 example in Chapter 14, the PICKIT 3 does not supply enough current to power the LCD display. Furthermore, unlike the MAX3232, the LCD display requires 5 volts, not 3 volts, to work. You can provide this voltage by using an AA battery holder that holds 3 batteries as described in Chapter 3.

Connecting the LCD Display to a Microcontroller

The LCD uses parallel data communication to write an entire byte to the display at once. The ideal way to do this would be to use eight pins of one port to do this. The only port on this particular chip that has 8 pins is Port C, and, unfortunately, two of those pins are being used for programming. For the first part of this chapter we will use all of the pins on Port C by unplugging the LCD display when programming the chip and unplugging the programmer and reinstalling the LCD display when we run it. Using this configuration it will not be possible to use the PICKit 3's hardware debugger.

The second part of the chapter will show how to run the display in “four bit mode” which will allow us to run the display and the PICKit 3 at the same time.

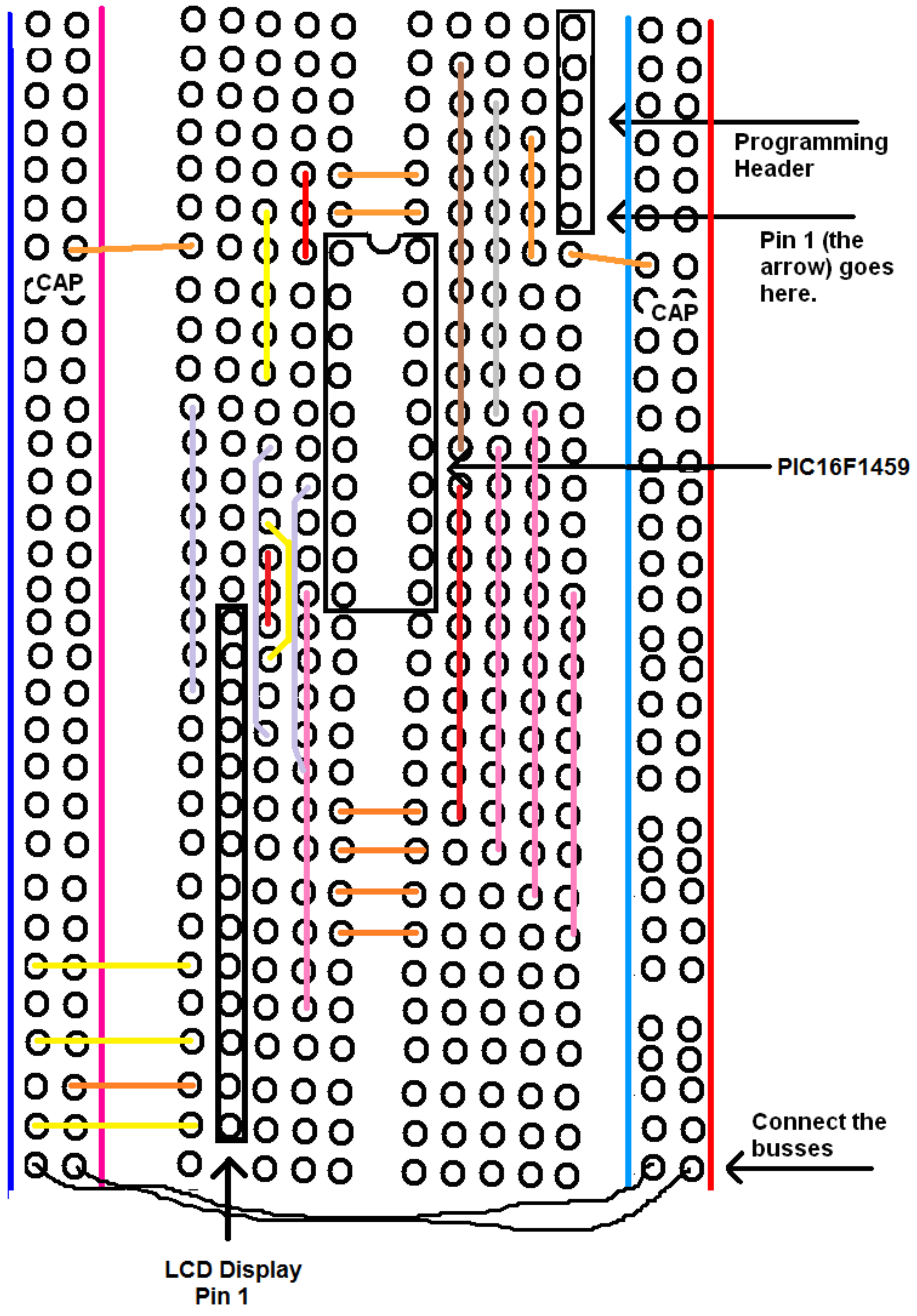

Figure 1: Wiring Diagram for an LCD Display

Figure 1 shows how to connect the LCD display to your bread board. The four wires that are shown in pink are either white wires (which would not show up on the diagram) or lengths that are not available in a standard bread board wire kit. In the latter case you will need to use the longer lengths in the kit and cut them to fit. Here is a summary of the wire connections:

Display Pin	Display Name	PIC Pin	PIC Description
1	GND		
2	PWR		
3	Contrast (GND)		
4	RS	10	RB7
5	R/W (GND)		
6	E	11	RB6
7	D0	16	RC0
8	D1	15	RC1
9	D2	14	RC2
10	D3	7	RC3
11	D4	6	RC4
12	D5	5	RC5
13	D6	8	RC6
14	D7	9	RC7

Table 1: Connections between the 16F1459 and the LCD Display

The first two pins provide power to the display. The third pin is labeled "Contrast" and calls for a potentiometer to be connected in a voltage divider circuit that provides a variable voltage to this pin to control the display contrast. However, my experience is that the contrast on the display is almost always optimal when zero volts is fed into this pin, so I have simply skipped the potentiometer and connected it directly to ground.

Sometimes you will be issuing a command to the display, for example to clear the screen, and sometimes you will be sending data to the display to be placed on the screen. This is determined by pin 4, the RS line. When it is low, it is set to receive instructions, when it is high, it is set to receive data to display on the screen.

It is possible to read the memory contents of the display as well as write to it. However, I find that it is almost never necessary to do this. The display is configured for being written to when Pin 5 (labeled R/W) is connected to ground and it is configured to be read from when 5 volts appears on this pin. In Figure 1, this pin is connected to ground, causing the display to always be written to and never be read. Pin 6, which is labeled "E", is the clock line which causes the display to read the data on the eight data lines. Those eight data lines are pins 7 through 14.

There are a number of commands that can control the state of the display. The most important ones are[1]:

Binary Value	**Command**	**Notes**
0000 0001	Clear Display	Also returns the cursor to the home position.
0000 0010	Cursor Home	Does not clear the display.
0000 01AB	Entry Mode Set	A = 0, cursor moves to left. A = 1, cursor moves to right. B = 1, display shifts, B = 0, no display shift.
0000 1ABC	Display on/off	A = 1, display on; B = 1, cursor on; C = 1, cursor blink.

0001 AB**	Cursor/Display Shift	A = 0 shift cursor; A = 1, shift display.
		B = 0, shift left; B = 1, shift right.

001A BC**	Function Set	A = 0, four bit interface. A = 1, 8 bit interface.
		B = 0, one line; B = 1; two lines.
		C = 0, 5x8 dots; C = 1 5x10 dots.

1NNN NNNN	Address	Set address (NNN NNNN) of next character to be written.

Table 2: Display Commands

Note that in the Function Set command it is possible to run this display using only a 4 bit interface. In that case the data of the characters to be displayed are written 4 bits (one nibble) at a time. In that case, only four data lines (those on pins 11 through 14) are required.

Programming the PIC Microcontroller to Drive an LCD Display

When learning a new programming language, programmers often begin with a "Hello World" program. This is a program that simply prints the text "Hello World" to the screen. When we started programming with a PIC, this was not possible because the PIC had no screen. Often it is said that the blinking LED is the "Hello World"program of embedded programming. Now, however, we have the tools to write a real "Hello World" program.

The initialization section of our program will have to do the usual initialization of the TRIS, ANSEL and OSCCON registers, but in addition, it will set up the display for writing. All of the PIC pins in Table 1 must be set as outputs. So the TRIS registers will look like this:

```
TRISB = 0;   //RB0 – RB3 do not exist on this chip so the associated TRIS values do not matter.
TRISC = 0;
```

The ANSEL registers must be setup as all digital pins:

```
ANSELB = ANSELC = 0;
```

And the OSCCON register will be setup to provide a 4 MHz oscillator:

```
OSCCON = 0b00110100;
```

[1] In this table, the use of an * means the bit is ignored.

Every time a byte is sent to the display it will be necessary to cycle the E (think clock) pin (RB6). Thus, it might make sense to spin this off into a separate function called “pulse()”. Different instructions require different amount of times to execute, but none of them require more than 2 ms, so we will incorporate a 2 ms delay in the pulse function. The function will look like this:

```
void pulse(){
        RB6 = 1;
        RB6 = 0;
        __delay_ms(2);
}
```

It is the dropping of RB6 that causes the data to be read by the display. We will initialize that line to 0 since the pulse function will raise it and then lower it. Initialization of the display will appear as follows:

```
        RB6 = 0;                 //set E low to prepare it for pulsing
        RB7 = 0;                //set RS low because we are about to issue commands not data
        __delay_ms(125);        //wait for display to power up
        PORTC = 0b00111000;     //set 8 bit, 2 lines, 5x7 matrix
        pulse();                //send the command
        PORTC = 0b00001111;     //display on, cursor on, cursor blinking
        pulse();                //send the command
        PORTC = 1;              //clear display
        pulse();                //send the command
        RB7 = 1;                //switch to data mode
```

Notice that the bit patterns for the commands are taken from the command table (Table 2). Note that when we are done setting up the display, we make the R/S line (RB7) high so that we can write data to the display. Writing data to the display involves writing data to specific memory locations in the HD44780 display driver chip. The chip then takes care of writing the pixels to the screen for each character in memory. To write text, first create an array to hold the text to be transmitted:

```
                char hellotext[] =  {'H','e','l','l','o',' ','W','o','r','l','d'};
```

Then write the code to print out the array to the display:

```
                        char i;
                        for (i = 0; i< 11; i++){
                                PORTC = hellotext[i];
                                pulse();
                        }
```

In summary, here is the entire eight bit version of the LCD Hello World program:

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
//pragma statements to set configuration bits go here…
#define _XTAL_FREQ 4000000
void initial();
void pulse();

int main(int argc, char** argv) {
    char i;
    char hellotext[]=  {'H','e','l','l','o',' ','W','o','r','l','d'};
    initial();
        for (i = 0; i< 11; i++){
            PORTC = hellotext[i];
            pulse();
        }
    while(1);
    return (EXIT_SUCCESS);
   }

void pulse(){
                RB6 = 1;
                RB6 = 0;
                __delay_ms(2);
}

void initial(){

        OSCCON  =  0b00110100;            //4 MHz clock
        ANSELB  =  ANSELC = 0;            //all pins digital
        TRISB = TRISC = 0;                //set up the pins as output
        RB6 = 0;                          //set E low to prepare it for pulsing
        RB7 = 0;                          //set RS low because we are about to issue commands
        __delay_ms(125);                  // make sure the display is fully powered up before proceeding
        PORTC = 0x38;                     //set 8 bit, 2 lines, 5x7 matrix
        pulse();                          //send the command
        PORTC = 0x0F;                     //display on, cursor on, cursor blinking
        pulse();                          //send the command
                PORTC = 1;                //clear display
                pulse();                  //send the command
        RB7 = 1;                          //switch to data mode

}
```

Moving On to Four Bit Mode

PIC pins are a scarce resource, and there's no reason to use eight of them when the display will allow us to get by with using only four. In addition, doing this will allow us to connect the PICKit 3 and the display at the same time. To accomplish this we will need to make two changes to the above program. First, we need to break each byte into two four bit nibbles and send them one after the other on pins RC4 – RC7. In addition, we need to change the display initialization routine so that it puts the display in four bit mode instead of eight bit mode.

Putting the display in four bit mode requires that we have the capability to either send the two nibbles of a byte or to send just one nibble. As a result, we will use two functions to accomplish this. Sending a single nibble (just the four most significant bits) could be accomplished with the following code:

```
void sendnibble(char b){
                PORTC = b;
                pulse();
}
```

The problem with this approach is that it may alter the state of the RC0 – RC3 pins as well as the ones we intended to set. Two of those pins (RC0 and RC1) are connected to the PICKit 3, but it is possible that we might be using the other two pins for some other purpose.[2] As a result, we may want to insure that we do not alter the state of the four least significant bits of Port C. This can be accomplished by changing the contents of the sendnibble() function as follows:

```
void sendnibble(char b){
   RC7 = (b & 0b10000000)>>7;
   RC6 = (b & 0b01000000)>>6;
   RC5 = (b & 0b00100000)>>5;
   RC4 = (b & 0b00010000)>>4;
   pulse();
}
```

This code uses a bitwise and operation to strip off the upper four bits of the byte that is being transmitted one by one. Then it left shifts the result into the appropriate location in the Port C register.

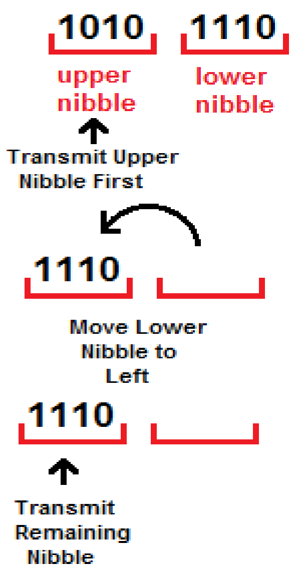
1010
1110
upper nibble
lower nibble
Transmit Upper Nibble First
1110
Move Lower Nibble to Left
1110
Transmit Remaining Nibble

Figure 2: Sending in Four Bit Mode

Sending an entire byte will require that we send the four most significant bits, then shift the four least significant bits four bits to the left and then send these four bits. The following function accomplishes this:

```
void sendbyte(char t){
 sendnibble(t);
 t = t << 4;
 sendnibble(t);
}
```

The last step will be to modify the display setup routine so that it puts the display in four bit mode. This involves sending the upper four bits of the function set command (see Table 2) four times. The first three times we command the display to 8 bit mode and the fourth time we command it to four bit mode. At this point the display is ready to receive the entire byte of the function set command which will also allow us to set the number of lines that the display has. For a two line display, the code will be as follows:

```
sendnibble(0b00110000);
sendnibble(0b00110000);
sendnibble(0b00110000);
sendnibble(0b00100000);
sendbyte(0b00100000);
```

You may have noticed that the only choices for the number of lines on a display are one or two. Yet, there are many four line displays available that use the HD44780 controller chip. On these displays the text appears as four lines, but from the display's perspective, there are really only two. The third line is simply a continuation of the first line and the fourth line is a continuation of the second.

[2] Note that it is also possible that C0 and C1 are being used for another purpose as well, once the PICKit 3 is disconnected.

Placing Text Precisely on the Display

So far, all we have done is to write text to the display starting at the top left corner. It is possible to achieve much more interesting effects if you can force each byte written to appear precisely where you want. In doing this you can center text, or produce special effects, such as having the text appear to fly in from the side of the display. To accomplish this you use the last command in Table 2, which allows you to specify the location of the next character that will be displayed.

To understand how to use this, you must understand how memory locations are specified for the display. Assuming we are dealing with a 4 line display, the four lines of the display begin and end with the following addresses:

	First Character Address	**Last Character Address** (Assuming 20 characters/line)
Line 1:	0x80 = 0b1000 0000	0x93 = 0b1001 0011
Line 2:	0xC0 = 0b1100 0000	0xD3 = 0b1101 0011
Line 3:	0x94 = 0b1001 0100	0xA7 = 0b1010 0111
Line 4:	0xD4 = 0b1101 0100	0xE7 = 0b1110 0111

The reason that the display starts at 80 and not zero is that the initial 1 is the 1 in the command table that specifies that this command is a display data memory address. The controller was originally designed to handle lines that are up to 40 characters in length and two lines. Four line displays are achieved by splitting the lines in half. Line 1 continues on line 3 and line 2 continues on line 4. This is clearly shown by looking at the last address on line 1 and the first address on line 3. Note also that bit 7 of the address indicates whether the byte is on line 1 (if it is a 0) or line 2 (if it is 1). This is why line 2 starts at 0xC0 instead of 0xA8 (the next byte after the end of line 3).

To place a character at a specific location use the following procedure:

1. Make the R/S line (Pin RB6) low to enter command mode.
2. Write the address to the display.
3. Make the R/S line high to enter data mode.
4. Write the desired character to the display.

The code below would take the Hello World text and have it appear to fly in from the right side of the screen character by character. Each character starts at the far right of the display. It is then replaced at that location with a space and is written to the next address to the left. This continues until it arrives at its final location on the display. The process repeats for each of the twelve characters.

```
char i,j;
char text[] = {'H','e','l','l','o',' ','W','o','r','l','d', '!'};
for (j = 0; j< 12; j++){
        for (i = 0x8F; i>= 0x80+j;i--){
                RB7 = 0;
                PORTC = i+1;
                pulse();
                RB7 = 1;
                PORTC = ' ';
                pulse();
                RB7 = 0;
                PORTC = i;
                pulse();
                RB7 =1;
                PORTC = text[j];
                pulse();
                if (text[j] != ' ') __delay_ms(60);
        }
}
```

Using Tables to Simulate Arrays in Assembly Language

Anything that involves text would seem to be a natural place to use an array, if you are writing in C. However, there are no arrays in assembly language. Microchip has introduced something that works in a very similar way, however, called a table. Consider the follow code:

```
lookup
        addwf   PCL, f
        retlw ‘H’
        retlw ‘e’
        retlw ‘l’
        retlw ‘l’
        retlw ‘o’
        retlw ‘ ’
        retlw ‘W’
        retlw ‘o’
        retlw ‘r’
        retlw ‘l’
        retlw ‘d’
```

This code involves several new instructions. The **addwf** instruction takes the specified register (PCL in this case) and adds to it whatever is in the working register. The result can be placed either back in the file (by specifying ",f") or in the working register (by specifying ",w").

The PCL register is the program counter. It keeps track of the location of the next instruction to be executed. As a result, when **lookup** is called, the effect of the **addwf** instruction will be to skip however many instructions are in the w register at the time **lookup** is called. For example suppose the working register contains 4 when lookup is called. After executing the **addwf** instruction, if the PCL register had not been changed, the next instruction to be executed would have been the **retlw ‘H’** instruction. Instead, because 4 had been added to the program counter, execution will skip down 4 instructions to **retlw ‘o’.**

The **retlw** instruction is similar to the usual **return** instruction except in addition to returning to the point where this subroutine was called, it also places the specified byte in the working register. As a result this code works in a manner that is very similar to an array lookup; place the number of the entry that you want in the working register and call lookup. The subroutine will return with the desired value in the working register. This makes it relatively easy to deal with groups of characters, for example, when writing to an LCD display.

The above method will work with any 16 series PIC microcontroller. Some of the newer ones (including the 16F1459) have added an additional assembly language instruction to make this even easier. Simply replace the `addwf PCL`, f instruction with a `brw` instruction:

```
lookup
        brw
        retfie ‘H’
        retfie ‘e’
```
…etc.

TRY THIS: Create a digital voltmeter. Connect the potentiometer to one of the analog inputs as you did in Chapter 12. Use the analog to digital converter to calculate the voltage and use the display to show the result. For example:

3.5 Volts

If you use 8 bit analog to digital conversion, you can compute the voltage to the nearest tenth of a volt. If you use 10 bit analog to digital conversion, you can compute the voltage to the nearest hundredth of a volt. In either case this is the precision of your measurement. Accuracy will depend on the accuracy of the reference voltage being used.