USC Viterbi School of Engineering

Ming Hsieh Department of Electrical Engineering

EE 109L - Introduction to Embedded Systems

Spring 2015 Lab Assignment #6
Using the LCD Display

1 Introduction

In this lab exercise you will attach an LCD display to the Arduino Uno and write software to display short messages on it. The steps required to communicate between the microcontroller and the LCD display are typical of how most devices are interfaced to a microcontroller.

2 The LCD Module

The LCD module used in this exercise is a monochrome, 16 character by 2 line display that uses a parallel interface. LCD displays are designed with either a serial or parallel interface, and both types have advantages and disadvantages. A serial interface transfers data one bit at a time and requires only one or two I/O lines from the microcontroller but is slower and often more expensive. A parallel interface display transfers data multiple bits at a time and can usually transfer data faster but requires multiple I/O lines from the microcontroller in order to operate.

The LCD used in this exercise (Fig. 1) is called an "LCD shield" since it is mounted on a board that has pins to plug into the Arduino Uno sockets. Once mounted on the Arduino it uses six of the Uno's I/O lines for data and control signals (Arduino I/O pins D4-D9).

The LCD shield also has a cluster of six pushbuttons. Five of these are interfaced through a multistage voltage divider to ADC channel 0 of the Arduino. Depending on which of the five buttons is pressed, or none, one of six analog voltages appears on the ADC0 input. By using the ADC to convert the voltage to a number it's possible to easily determine if one of the five buttons was pressed. The sixth button, marked "RST" is connected to the RESET line of the Arduino and can be used to restart the program.

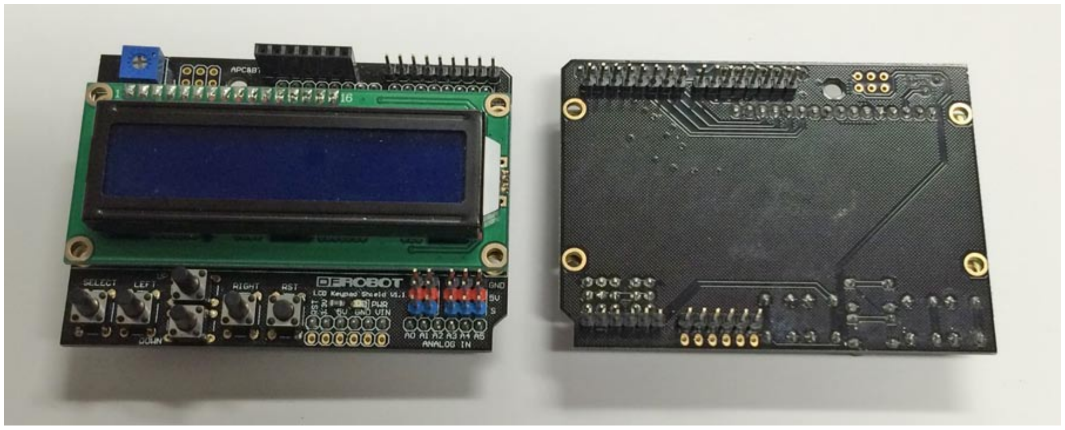

Figure 1: Front and back of the 16x2 character LCD shield

3 The LCD Interface

The parallel interface to the LCD is used to transfer data between the microcontroller and a special integrated circuit in the LCD module that controls where the dark and light spots appear on the display. The presence of this IC allows the microcontroller to display the letter 'A' by just sending the ASCII code for an 'A' (0x41) rather than having to send information about the individual dots that make up the letter on the screen. The circuit used on this module (Hitachi HD44780) is one of the most commonly used for controlling LCD modules and is designed to communicate with a microcontroller using eight data lines and three control lines. However in order to save I/O lines it can also communicate in a 4-bit mode that sends an eight bit byte in two 4-bit "nibbles", first the most significant four bits, then the least significant four bits. The four bit mode is used with this LCD shield.

In addition to the four data lines, the controller IC has three control lines: RS (Register Select), RW (Read/Write) and E (Enable). On the LCD shield the designers have eliminated the the RW line so data can only be written to the shield. The result of all this is that the shield can be used with just six I/O lines, four for data and two for control signals.

Command/data bits 4 through 7 - (Connected to Arduino I/O pins D4-D7 = PORTD, bits PD4-PD7) - These lines carry four bits of the command or data bytes that are sent to the LCD. Most operation require sending a full byte so transfers have to be done in two steps.

Register select - (Connected to Arduino I/O pin D8 = PORTB, bit PB0) - Register select. Set to 1 to write to the data register. Clear to 0 to write to the command registers.

Enable - (Connected to Arduio I/O pin D9 = PORTB, bit PB1) - All writes to the data or command registers are done by making the Enable line do a $0 \rightarrow 1 \rightarrow 0$ transition.

3.1 Command and Data Registers

Inside the controller IC on the LCD module are two 8-bit registers, the command register and the data register. The Register Select control line is used to determine which register the microcontroller transfers data to.

When the microcontroller wants the LCD to perform certain operations it writes the necessary 8-bit command code into the command register to initiate the operation. Typical commands include clearing the display, moving the cursor to some location, turning the display on or off, etc.

The data register is where the program writes character data that it wants to have displayed on the LCD. When a character is written to the data register it will appear on the display at the location that the cursor was at, and then cursor will move to the right one position. Writing a string of characters to the data register one after the other has the effect of making the string appear on a line of the display with the cursor left positioned after the last character in the string.

3.2 Command and Data Transfers

Transfering commands and data to the LCD involves a number of steps all involving setting or clearing bits in the six I/O port bits that are connected to the six interface lines described above. The LCD shield does all its data transfer in the 4-bit interface mode. In some cases only 4 bits have to be transferred, in others a full 8-bit must be transferred.

After most data transfers, it is necessary to insert a short delay to give the controller IC on the module time to execute the action. A 2msec delay should be sufficient. For 8-bit transfers, it is not necessary to delay between transferring the upper and lower parts of the byte, but there should be a delay after the second part of the transfer.

3.2.1 Tranferring 4 bits

For some of the control transfers, only 4-bits have to be transferred. In this situation the following steps are performed.

1. Configure the RS (Register Select) line to determine the destination of the data transfer. RS = 0 puts the 4 bits in the command register. RS=1 puts the 4 bits in the data register.
2. Put the four bits to be transferred on the data lines (Uno's D7-D4, or Port D, bits 7-4).
3. Make the E (Enable) line go from 0 to 1 and back to 0. E must be high for at least 230ns.

3.2.2 Tranferring 8 bits

For most transfers we have to send all eight bits of the byte. This has to be done using **two 4-bit transfers** so the following steps have to be performed.

1. Configure the RS (Register Select) line to determine the destination of the data transfer. RS = 0 puts the 4 bits in the command register. RS=1 puts the 4 bits in the data register.
2. Put the **upper** four bits of data (bits 4-7) on the data lines (Uno's D7-D4, or Port D, bits 7-4).
3. Make the E (Enable) line go from 0 to 1 and back to 0 to transfer the upper bits. E must be high for at least 230ns.
4. Put the **lower** four bits of data (bits 0-3) on the data lines
5. Make the E (Enable) line go from 0 to 1 and back to 0 to transfer the lower bits. E must be high for at least 230ns.

3.3 Initializing the Display

Unfortunately you can't just start sending character data to the LCD and have it appear. The module has to have a few initialization steps performed before it will accept data and display it. All of these steps are simply commands that must be sent to the LCD in the same way as other commands. In most cases a delay of some specified amount must be done after the command is sent. Delays can be implemented using the "`_delay_ms`" and "`_delay_us`" functions. The following sequence of commands and delays must be performed in order to get the display working.

1. Delay at least 15msec after power on and program running
2. Send the 4-bit command 0011, followed by a delay of at least 5msec.
3. Send the 4-bit command 0011, followed by a delay of at least 100μsec.
4. Send the 4-bit command 0011.
5. Send the 4-bit command 0010 to set the module to use 4-bit interfacing. Delay 2ms.
6. Send the 8-bit command 00101000 to set the module for 2-line display. Delay 2ms.
7. Send the 8-bit command 00001111 to turn on display and turn on a blinking cursor. Delay 2ms.

After the above steps are done the display is ready to accept data to display. If you now write data to the data register it should appear on the screen starting in the upper left position.

3.4 Displaying Characters

The LCD display uses the ASCII character codes to determine which characters to display. Once the LCD has been initialized as shown above, any byte written into the data register will result in the character with that ASCII code being displayed on the LCD and the cursor that indicates where the next character will go is advanced one position to the right. For example if the bytes with values 0x55, 0x53, 0x43 are written to the data register one after the other, the character "USC" will appear on the screen.

The position of the cursor can be controlled by sending a command byte to the LCD. The format of the cursor position command is the eight bit value 0x80+p where "p" is the number of the position on the screen where the cursor is to be moved to. The positions on the first line start at 0x00 at the left side and go up to 0x0F at the right side. Similarly, the positions on the second line start at 0x40 and go up to 0x4F.

For example, if you want to move the cursor to the 4th position on the 1st row, send the command byte 0x83 (0x80 + 0x03). To go to the 11th position on the second row send a 0xCA (0x80 + 0x4A).

In most cases programs should execute a small delay after sending each command or data byte to the LCD. A 2msec delay should be sufficient for most operations.

4 Programming the LCD

This lab exercise is a good example of how programs can be structured so the complexities of one part are hidden from other parts of the program. As much as possible the details of how the LCD module works should be handled by only a small portion of the code. Most of the program should not have to deal with knowing which I/O port bits are being used, setting the control bits, dealing with delays, etc. for each character it wants to display. To put characters on the display, the program just has to call a string output function that has a pointer to the string of characters to be displayed as its argument. All the details about how that gets done should be isolated in another part of the program. Doing this has an additional advantage that the LCD could be changed for a model with a different interface, and only the small number of routines that deal directly with the interface will have to be changed.

Try writing your programs in a layered manner as described below. Set it up with three layers of functions. The main program only calls the functions in the top layer. The top layer has a small number of simple functions that make use of the mid level functions. The mid level functions are more complex and make use of the low level functions. The low level functions deal with most of the details of changing bits in the I/O ports.

4.1 Top Level Functions

The top level routines are to initialize the LCD and to write strings of characters starting at specified locations. For example

```
void init_lcd()
{
    /* Configure the I/O ports and send the initialization commands to the LCD */
}

void stringout(char *str)
{
    /* Write the string pointed to by "str" at the current position
}

void moveto(unsigned char pos)
{
    /* Move the cursor to the position "pos" */
}
```

These routines should make use of the functions defined in the mid level (and if necessary the lower level) of the program. **Important:** Your `init_lcd` should only configure the DDR bits for the Port bits being used by he LCD. Don't modify any other DDR bits.

4.2 Mid Level Functions

Two mid level functions are used to send a byte to the command register and the data register.

```
void writecommand(unsigned char x)
{
    /* Send the 8-bit byte "x" to the LCD command register */
}

void writedata(unsigned char x)
{
    /* Send the 8-bit byte "x" to the LCD data register */
}
```

These routines must set the register select line to the correct state for a command or data transfer, and then make use twice of the low level “`writenibble`” function to first send the upper four bits of the byte, and then send the lower 4 bits. After an 8-bit transfer is complete, the function should delay for about 2msec to let the operation finish.

4.3 Low Level Functions

The low level function “`writenibble`” transfers a four bit value from the argument to whatever register has been selected by the register select signal.

```
void writenibble(unsigned char x)
{
    /* Send four bits of the byte "x" to the LCD */
}
```

All transfers of data and commands depend on this function to do the actual transfer. Note that it is very important that the `writenibble`, `writecommand` and `writedata` functions all agree on which four bits of the 8-bit argument are to be sent whenever `writenibble` is called. It could be the lower four bits, or the upper four bits, and there is no real advantage to doing it one way or the other. The only important thing is that the three functions agree on which to use.

Important: Your `writenibble` routine must only change the bits in registers B and D that need to be changed in order to affect the transfer. Don’t just copy a full byte into a register if you only need to modify a few bits. Any bits that are not part of the transfer should **not** be changed.

5 Formatting Strings

Since the LCD is used to display strings of characters, it’s necessary to have a efficient way to create the strings to be displayed. Constant strings are no problem and can be given as an argument to the “`stringout`” function.

```
stringout("Hello, world"};
```

It gets more complicated when you need to create a string containing variable values, such as numbers. The best tool for doing that is the “`snprintf`” function that is a standard part of most C languages. **Important:** In order use `snprintf`, or any other routine from the C standard I/O library, you must have the following line at the top of the program with the other `#include` statements.

```
#include <stdio.h>
```

The `snprintf` function is called in the following manner

```
snprintf(buffer, size, format, arg1, arg2, arg3, ...);
```

where the arguments are

buffer – A `char` array large enough to hold the resulting string.

size – The size of the `buffer` array. This tells the `snprintf` program the maximum number of character it can put in `buffer` regardless of how many you try to make it do.

format – The heart of the `snprintf` function is a character string containing formatting codes that tell the function exactly how you want the following arguments to be formatted in the output string. More on this below.

arguments – After the `format` argument comes zero or more variables containing the values to be placed in the `buffer` according to the formatting codes that appear in the `format` argument. For every formatting code that appears in the `format` argument, there must be a corresponding argument containing the value to be formatted.

The `format` argument tells `snprintf` how to format the output string and has a vast number of different formatting codes that can be used. The codes all start with a percent sign and for now we will only be working with two of them:

%d – Used to format decimal integer numbers. When this appears in the format string, the corresponding argument will be formatted as an decimal integer number and placed in the output string.

%s – Used to format string variables. When this appears in the format string, the corresponding argument will be assumed to be a string variable and the whole string variable will be copied into the output string.

The `format` string must have the same number of formatting codes as there are arguments that follow the format argument in the function call. Each formatting code tells how to format its corresponding argument. The first code tells how to format "`arg1`", the second code is for "`arg2`", etc. Anything in the `format` argument that is not a formatting code will be copied verbatim from the format string to the output string.

Example: Assume you have three unsigned char variables containing the month, day and year values, and a string variable containing the day of the week, and you want to create a string of the form "`Date is month/day/year = dayofweek`".

```
char date[30];
unsigned char month, day, year;
char *dayofweek = "Saturday";
month = 2
day = 21;
year = 15

snprintf(date, 30, "Date is %d/%d/%d = %s", month, day, year, dayofweek);
```

After the function has executed, the array "`date`" will contain "`Date is 2/20/15 = Saturday`". Since you told `snprintf` that the size of the array was 30, the maximum number of characters it will put in the array is 29 since as with all C strings, the string has to be terminated with a null byte (0x00).

6 Tasks

6.1 Test the LCD Shield

Install the LCD shield on the top of the Arduino Uno. Make sure that you are lining up the pins and connectors properly before trying to push them together. Two of the male headers on the shield are the same size as the mating connectors on the Uno and these go into the D0-D7 and A0-A5 connectors. The other two male headers have fewer pins than the connectors they are plugged into. Take care to make sure that all the pins are going into the correct openings before applying pressure to push the two boards together. If you have problems mounting the shield on the Uno bring the boards to one of the instructors for help.

Create a `lab6` folder in your `ee109` folder. From the class web sites download the file `lab6.zip` and put it in the `lab6` folder. Once you unzip it you should have these three files for Lab 6:

`Makefile` - this is a modification of the normal `Makefile` to allow testing of the LCD shield

`test.hex` - the binary test program for the LCD shield

`lab6a.c` - skeleton file containing functions declarations to get you started

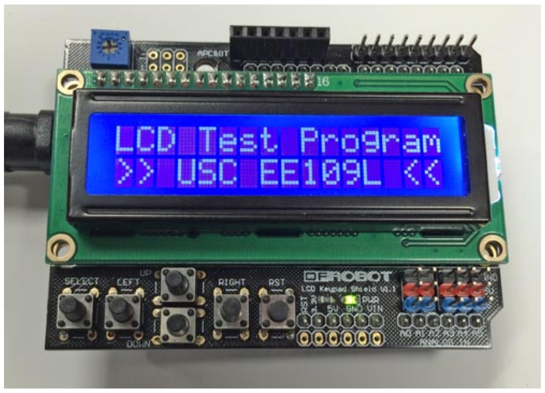

Figure 2: LCD shield with test program running

Make the usual changes to the `PROGRAMMER` line at the top of the Makefile to make it work on your computer. Attach your Uno+LCD to your computer and enter the command "`make test`". This will result in something similar to what you see when entering the `make flash` command but will download the data from the `test.hex` file to the Uno. Once the download is complete the LCD should show two lines of text (Fig. 2). If nothing shows up on the screen or it shows a lot of white boxes, try changing the display contrast by using your screwdriver to adjust the potentiometer in the upper left corner of the display. If you can't get the test program to work, ask one of the instructors or TAs for help. Don't try working on the rest of the lab assignment until the test program is working.

6.2 The Lab6a Program

Using the `lab6a.c` file as a starting point, write a program that does the initialization steps described above and then writes some text to the screen. At least one of the strings written to the LCD must have been created using the `snprintf` routine as described above in Sec. 5. For example, try writing your name (as much as will fit) to the first row, and then using `snprintf` to create a string with your birthday to write to someplace on the second row. Your program should demonstrate that you can write characters at whatever position you need them to be, not just a string of characters starting in the upper right corner.

Remember to use "`make flash`" to download your program. The "`make test`" is only used to download the test program in the `test.hex` file.

6.3 The Lab6b Program

Copy the `lab6a.c` file to a new file, `lab6b.c`. Edit this file to expand on the program you wrote above to create a program that uses the buttons on the LCD shield to change the value stored in a variable and display the results on the LCD. The program should have a signed variable that starts at zero and then is changed depending on which of the five buttons has been pressed:

Left – Increment the variable by one

Up – Increment the variable by ten

Down – Decrement the variable by ten

Right – Decrement the variable by one

Select – Reset the variable back to zero.

Use what you learned in Lab 5 to give the program the ability to use an ADC channel to read the analog signal coming from the voltage divider that the buttons are connected to. The ADC output can be used to determine which button was pressed and change the variable value accordingly. Display the value of the variable on the LCD and update it as the buttons are pressed. The goal is to be able to make the displayed number go up or down in increments of one or ten as the buttons are pressed.

In order to determine which button was pressed you have to know what the ADC output is for each button, but how do you know what those values are? Answer: write code to loop continuously reading the ADC result and writing the value to the LCD. Once the program is running, just press each button and see what value is displayed.

When your program is running, it should only write data to the LCD when needed. Unnecessary writing to the LCD, like when the value hasn't changed, can lead to the display flickering. Your code should check to see if the displayed value has to be changed and only then write to the LCD.

7 Results

Once you have the assignments working demonstrate them to one of the instructors. Turn in a copy of your source code (see website for details.)