

Lab 7 - LCD

# Preparation

* Read Sections 7.1 to 7.7, 9.1, 9.2, 9.3, 9.4, 9.6, 9.8, 10.1, 10.4, and 10.6
* Go through Lectures 20 to 22 again
* Download and load into CCS “EGCP\_450\_Lab\_7\_Part\_1.zip”
* [Watch the related video(s) here](https://www.youtube.com/watch?v=-FeUNmyqSrg&list=PLxYT_0WQMg35FiE5EDnCwW-1p7-E3gNtJ)

# Purpose

## Part 1: LCD Driver

This part of the lab has these major objectives: 1) to interface a LCD display that can be used to display information on the embedded system, 2) to use indexed addressing to access strings, 3) to learn how to design implement and test a device driver using busy-wait synchronization, and 4) to use fixed-point numbers to store non-integer values.

## Part 2: Real-Time Position Measurement System

This part of the lab has these major objectives: 1) an introduction to sampling analog signals using the ADC interface, 2) the development of an ADC device driver, 3) learning data conversion and calibration techniques, 4) the use of fixed-point numbers, 5) the development of an interrupt-driven real-time sampling device driver, 6) the development of a software system involving multiple files, and 7) learn how to debug one module at a time.

# System Requirements

## Part 1: LCD Driver

In this part of the lab, you will interface a 16x2 character LCD to the MSP432 using a Hitachi HD44780 LCD controller (Figure 1). This part of the lab will use “busy-wait” synchronization, which means before the software issues an output command to the LCD, it will wait until the display is not busy. In particular, the software will wait for the previous LCD command to complete.

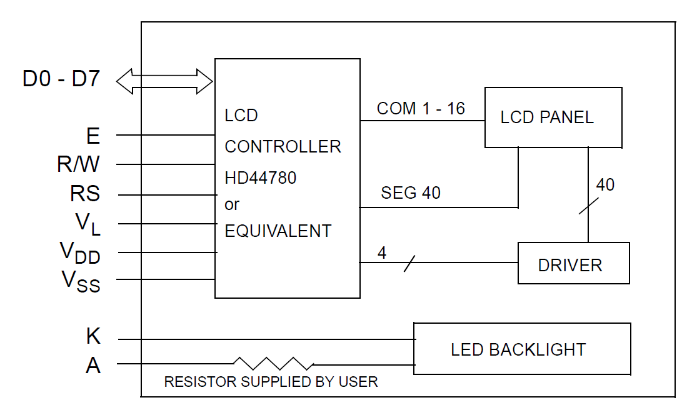


Figure : Hitachi HD44780 LCD controller.

The objective of this part of the lab is to develop a device driver for the LCD display. A device driver is a set of functions that facilitate the usage of an I/O port. In particular, there are three components of a device driver. The first component is the description of the driver. Place the function prototypes for the public functions in the header file **LCD.h**. It is during the design phase of a project that this information is specified. The second component of a device driver is the implementation of the functions that perform the I/O. Place the implementations in the corresponding code file (e.g., **LCD.c**). In addition to public functions, a device driver can also have private functions. This interface will require some private functions that output commands and data to the LCD (private functions should not include **LCD\_** in their names). In this part of the lab, you are required to develop and test the developed public functions (public functions must include **LCD\_** in their names). The third component is a main program that can be used to test these functions.

In the **LCD.c** file, you will implement and test the functions to communicate directly with the LCD. You will write the initialization ritual (i.e., **LCD\_Init**) which must be a public function in the **LCD.c** file. You will also write two functions that output to the LCD. Your **LCD\_OutCmd** function will be used to output 8-bit commands to the LCD, and your **LCD\_OutChar** function will be used to output 8-bit data to the LCD. However, the LCD **must be operating in 4-bit mode**, so only data lines DB4-DB7 of the LCD will be used (see Figure 1). So, when sending commands or data, you have to do the following steps:

1. Set RS signal
   1. Data 🡪 RS = 1
   2. Command 🡪 RS = 0
2. Mask out (make zeros) lower 4-bits and shift
3. Send to the LCD port (DB4-7)
4. Send enable signal (pulse for ~6µs)
5. Mask out (make zeros) higher 4-bits
6. Send to LCD port (DB4-7)
7. Send enable signal (pulse)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Public Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------LCD\_Clear------------------------

// Clear the LCD

// Inputs: none

// Outputs: none

void LCD\_Clear**()**

//-----------------------LCD\_Init--------------------------

// Initialize LCD

// Inputs: none

// Outputs: none

void LCD\_Init**()**

//-----------------------LCD\_OutChar-----------------------

// Output a character to the LCD

// Inputs: letter is ASCII character, 0 to 0x7F

// Outputs: none

void LCD\_OutChar**(**char letter**)**

//-----------------------LCD\_OutCmd------------------------

// Output a command to the LCD

// Inputs: 8-bit command

// Outputs: none

void LCD\_OutCmd**(**unsigned char command**)**

Also in **LCD.c**, you will implement and test four more public display functions. The **LCD\_OutString** function will be used to output an entire string to the LCD, the **LCD\_OutUDec** function will be used to output an unsigned 32-bit integer to the LCD, the **LCD\_OutUHex** function will be used to output an unsigned 32-bit HEX number to the LCD, and finally the **LCD\_OutUFix** function will be used to output an unsigned 32-bit fixed-point number to the LCD. An example of how **LCD\_OutUFix** should output values is shown in Table 1 below.

//----------------------LCD\_OutString----------------------

// Output String (NULL termination)

// Input: pointer to a NULL-terminated string to be transferred

// Output: none

void LCD\_OutString**(**char **\***pt**)**

//-----------------------LCD\_OutUDec-----------------------

// Output a 32-bit number in unsigned decimal format

// Input: 32-bit number to be transferred

// Output: none

// Variable format 1-10 digits with no space before or after

void LCD\_OutUDec**(**uint32\_t n**)**

//-----------------------LCD\_OutUHex-----------------------

// Output a 32-bit number in unsigned hexadecimal format

// Input: 32-bit number to be transferred

// Output: none

// Variable format 1 to 8 digits with no space before or after

void LCD\_OutUHex**(**uint32\_t number**)**

// -----------------------LCD\_OutUFix----------------------

// Output characters to LCD display in fixed-point format

// unsigned decimal, resolution 0.001, range 0.000 to 9.999

// Inputs: an unsigned 32-bit number

// Outputs: none

void LCD\_OutUFix(uint32\_t number)

Table : Specification for the LCD\_OutUFix function.

|  |  |
| --- | --- |
| Parameter | LCD display |
| 0 | 0.000 |
| 1 | 0.001 |
| 999 | 0.999 |
| 1000 | 1.000 |
| 9999 | 9.999 |
| 10000 or more | \*.\*\*\* |

An important factor in device driver design is to separate the policies of the interface (how to use the programs, which are defined in the comments placed at the top of each function) from the mechanisms (how the programs are implemented, which are described in the comments placed within the body of the functions).

The third component of a device driver is a main program that calls the driver functions. This software has two purposes. For the developer (you), it provides a means to test the driver functions. It should illustrate the full range of features available with the system. The second purpose of the main program is to give your client or customer (e.g., the instructor) examples of how to use your driver.

### Procedure

The basic approach to this part of the lab will be to first develop and test each component separately. During this phase of the project, you will use the debugger to observe your software operation. After each component is debugged, you will combine the components into one system on the MSP432. There are many functions to write in this lab, so it is important to develop the device driver in small pieces. One technique you might find useful is desk checking. Basically, you hand-execute your functions with a specific input parameter. For example, using just a pencil and paper think about the sequential steps that will occur when **LCD\_OutUDec** or **LCD\_OutUFix** processes the input 187. Later, while you are debugging the actual functions on the debugger, you can single step the program and compare the actual data with your expected data.

#### Task A

This part of the lab is sufficiently complex that I suggest that you test the first 4 public functions of **LCD.c** in the debugger. You can do this by verifying that the relevant variables, memory, and/or register get set with the appropriate data at the appropriate times.

#### Task B

Next, test the hardware display. Try to output a single character and, if your **LCD\_OutChar** and **LCD\_OutChar** functions in **LCD.c** are correct, you should see the character on the LCD screen. If you are sure your functions are correct and you are still not getting any output or it is incorrect, check your wiring before calling me over to debug for you.

#### Task C

Implement and test the reaming 4 public functions. I would recommend that you test them in the debugger first and then test the output on the LCD. Your test should be sufficient enough so that you are confident that it will perform as expected before you demo.

## Part 2: Position Measurement System

You will design a **position meter**. Your software will use the 14-bit ADC built into the microcontroller. A linear slide potentiometer (Bourns PTA2043-2015CPB103) converts position into resistance (0 ≤ *R* ≤ 10 k). The full scale range of position may be anywhere from 1.5 to 2 cm. You will use an electrical circuit to convert resistance into voltage (*Vin*). Since the potentiometer has three leads, one possible solution is shown in Figure 2. Add a *R1* resistor in the circuit so the input to the ADC ranges from 0 to 2.5V. You may use any ADC channel. The MSP432 ADC will convert voltage into a 14-bit digital number (0 to 16383). This ADC is a successive approximation device with a conversion time on the order of several sec. Your software will calculate position from the ADC sample as a decimal fixed-point number (resolution of 0.001 cm). The position measurements will be displayed on the LCD using the LCD device driver developed in Part 1. A periodic interrupt will be used to establish the real-time sampling. The main program and SysTick ISR must be written in C. You must use your LCD device drivers that you developed in Part 1. The ADC code must be written in C. The device drivers should be in **adc.c**. Each driver file will have a corresponding header file with the prototypes to public functions. The SysTick initialization, SysTick ISR, mailbox and the main program will be in the **main.c** file.

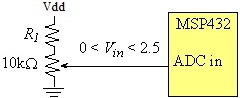


Figure : Possible circuit to interface the sensor (use your ohmmeter on the sensor to find pin numbers 1, 2, and 3).

Figure 3 shows a possible data flow graph of this system. Dividing the system into modules allows for concurrent development and eases the reuse of code.



Figure : Data flow graph and call graph of the position meter system. Notice the hardware calls the ISR.

You should make the position resolution and accuracy as good as possible using the 14-bit ADC. The **position resolution** is the smallest change in position that your system can reliably detect. In other words, if the resolution were 0.01 cm and the position were to change from 1.00 to 1.01 cm, then your device would be able to recognize the change. Resolution will depend on the number of ADC bits. The resolution can be calculated using:

(1)

where Vmax is the max input voltage of the ADC, Vmin minimum detectable input voltage change, n is the number of bits. For example, if Vmax=2.5V and n=14, then the minimum detectable input voltage change for the ADC is:

NOTE: **accuracy** is defined as the absolute difference between the true position and the value measured by your device. Accuracy is dependent on resolution, but in addition it is also dependent on the reproducibility of the transducer and the quality of the calibration procedure. Long-term drift, temperature dependence, and mechanical vibrations can also affect accuracy.

### Procedure

The basic approach to this part of the lab will be to debug each module separately. After each module is debugged, you will combine them one at a time. For example: 1) just the ADC, 2) ADC and LCD, and 3) ADC, LCD and SysTick.

The analog signal connected to the microcomputer comes from a position sensor, such that the analog voltage ranges from 0 to Vdd as the position ranges from 0 to *Pmax*, where *Pmax* may be any value from 1.5 to 2 cm.

In the final system, you will use SysTick interrupts to establish 40 Hz sampling. In particular, the ADC should be started exactly every 25 msec. The SysTick ISR will store the 14-bit ADC sample in a global variable (called a MailBox) and set a flag. Read Section 9.3 in the book to see how a **Mailbox** can be used to pass data from the background into the foreground. The main program will collect data from the Mailbox and convert the ADC sample (0 to 16383) into a 32-bit unsigned decimal fixed-point number, with a ∆ of 0.001 cm. Lastly, your main program will use your **LCD\_OutUFix** function from the previous lab to display the sampled signal on the LCD. Include units on your display (Figure 4).

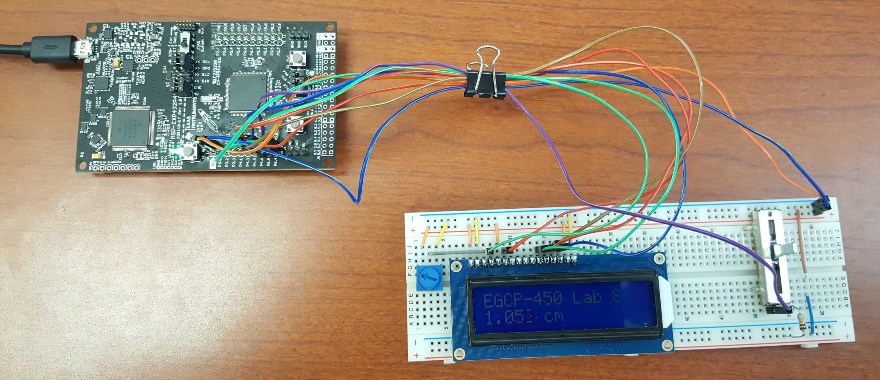


Figure 4: Hardware setup, showing the LCD and slide POT. The slide POT is used to measure distance. The R1 resistor shown in this figure creates a Vin ranging from 0 to 2.5V.

#### Task A

You will notice in the “EGCP\_450\_Lab\_7\_Part\_1” project a main program and three submodules (ADC, SysTick, and LCD). Each module has a header file containing the prototypes for public functions (SysTick.h, ADC.h, and LCD.h). Use the LCD driver you developed in Part 1. Figure 5 shows the call graph. Main calls ADC, LCD, and SysTick. The ADC module accesses the ADC hardware, and the LCD module access the LCD hardware.

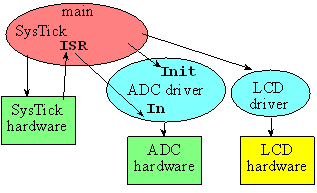


Figure : Possible call graph.

#### Task B

Write two functions: **ADC\_Init** that will initialize the ADC interface and **ADC\_In** will sample the ADC. You are free to pass parameters to these two functions however you wish. You are free to use any of the ADC channels. Write main program number 1, which tests these two ADC functions. In this system, there is no LCD, and there are no interrupts. Debug this system on the real MSP432 to show the sensor and ADC are operational. To debug it, you will have to use the debugging methods discussed.

#### Task C

Write main program number 2, which you can use to collect calibration data. In particular, this system should first sample the ADC and then display the results as unsigned decimal numbers. In this system, there is no mailbox, and there are no interrupts. You should use your **LCD\_OutUDec** developed in the previous lab. Again, using the debugging methods discussed in class, measure some the analog inputs (measured with a digital voltmeter) and see what the corresponding ADC samples are (measured with main program 2). The full scale range of your slide pot will be different from the slide pot of the other students, which will affect the gain (voltage vs. position slope).

#### Task D

Write a function in C that converts an ADC sample into a 32-bit unsigned fixed-point number. The input parameter (ADC sample) to the function will be passed by value, and your function will return the result (integer portion of the fixed-point number). You are allowed to use a linear equation to convert the ADC sample into the fixed-point number.

#### Task E

Use SysTickInts (interrupting version of SysTick) from Lab 6 to initialize the SysTick system to interrupt at exactly 40 Hz (every 0.025 seconds). If you used SysTick to implement the blind wait for the LCD driver, you will have to go back to Part 1 and remove all accesses to SysTick from the LCD driver. If you did not use SysTick for the LCD waits, then there is no conflict.

#### Task F

Write an interrupt handler in SysTickInts that samples the ADC and enters the data in the mailbox. Using the interrupt synchronization, the ADC will be sampled at equal time intervals. Toggle a heartbeat LED (P1.0) each time the ADC is sampled. The frequency of the toggle is a measure of the sampling rate. The ISR performs these tasks:

1. Toggle heartbeat LED (change from 0 to 1, or from 1 to 0)
2. Sample the ADC
3. Save the 14-bit ADC sample into the mailbox ADCMail
4. Set the mailbox flag ADCStatus to signify new data is available
5. Toggle heartbeat LED (change from 0 to 1, or from 1 to 0)
6. Return from interrupt

#### Task G

Write the main program number 4, which initializes the timer, LCD, and ADC. After initialization, this main program (foreground) performs these five tasks over and over.

1. Wait for the mailbox flag ADCStatus to be true
2. Read the 14-bit ADC sample from the mailbox ADCMail
3. Clear the mailbox flag ADCStatus to signify the mailbox is now empty
4. Convert the sample into a fixed-point number (variable integer is 0 to 2000)
5. Output the fixed-point number on the LCD with units (NOTE: it is better to move the cursor then to clear the LCD).

After each component has been separately debugged, combine them into one system on the real MSP432.

# Demonstration

You will show the instructor both parts (parts 1 and 2) of your program operation on the actual MSP432 board.

# Lab Report

The lab report is how I will grade you on your labs. Usually, the report is due 1 week following the completion of the lab (see TITANium for due dates). However, do to unforeseen circumstances, due dates may change. I will try my best to keep everyone informed of any changes. With this said, it is your responsibility to turn the report in during the scheduled due date. **There is a 10% penalty for late reports**. **Your report must be in MS Word Doc format. Submitting the report in another format will result in a 10% penalty.** Your lab report must include the cover sheet from the lab report template available on TITANium. **Not including the cover sheet will result in a 10% penalty.** The template contains instructions for the report and the rubric used for grading the labs. Please, read it thoroughly. **Don’t forget to include pertinent information such as code, flowcharts, waveform output, etc. in your report**. Please, no “spaghetti” code, keep your code clean and use comments. Remember, your code will affect your lab grade. If I can’t understand it, then I will assume it’s incorrect.