Lab 8. Real-Time Position Measurement System (Spring 2021)

Preparation

Read Sections 9.1, 9.2, 9.3, 9.4, 9.6, 9.8, 10.1, 10.4, and 10.6 (old Volume 1 book)
Chapter 8 (new book); Chap14 in E-Book
http://users.ece.utexas.edu/%7Evalvano/Volume1/E-Book/C14_ADCdataAcquisition.htm

In Spring 2021, the EE312H/EE319H students will use **Lab8_C++**
Review the example projects in EE319Kware using ADC:

ADCSWTrigger_4C123
Lab8_EE319K
Lab8_C++ (for EE319H)

Rulers: http://www.vendian.org/mncharity/dir3/paper_rulers/UnstableURL/rules_cm_narrow_20cm.pdf

Purpose

This lab has these major objectives:

1. introduction to sampling analog signals using the ADC interface
2. development of an ADC device driver
3. learning data conversion and calibration techniques
4. use of fixed-point numbers
5. development of an interrupt-driven real-time sampling device driver
6. development of a software system involving multiple files and a mixture of assembly and C
7. learn how to debug one module at a time
8. In Spring 2021, EE312H/EE319H students will implement one class in C++

System Requirements

You will design a position meter.This position meter will:

- use the PD2 analog channel 5, 12-bit ADC built into the microcontroller.
- sample the ADC with the SysTick at 10 Hz.
- use a linear slide potentiometer.
- display the distance along the potentiometer on the LCD as a decimal fixed-point number with a resolution of 0.01 cm.
- In Spring 2021, EE312H/EE319H students will implement the class **SlidePot**

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 most likely be able to recognize the change.

Procedure

The basic approach to this lab will be to debug each module separately. Remember to copy your Lab 7 solution files into your Lab 8 project. SysTick and ADC will simulate, but the SSD1306 will not. So, for Labs 7 to 10 we will debug on the board and not the simulator. After each module is debugged, you will combine them one at a time. For example:

1. Just the ADC.
2. ADC and LCD.
3. ADC, LCD and SysTick.

Part a - Design Ruler

Design a ruler using the slide potentiometer that converts a position on the slide potentiometer to a distance. Your design may require cutting, gluing, and/or soldering. A linear slide potentiometer, like your Bourns PTA2043-2015CPB103, converts position into a resistance $0 \leq R \leq 10$ kΩ. The system in Figure 8.2 was created by

plugging the slide pot into the protoboard and wrapping a solid wire to the armature to create a cursor, and gluing a photocopy of a metric ruler on the potentiometer. Minimum distance is 0 cm, and the maximum distance may be any value from 1.5 to 2.0 cm. The maximum distance is determined by the physical size of the potentiometer.

After you decide how to attach the measurement device to your potentiometer, you must ensure that you can connect the potentiometer to your breadboard. Label the three wires connected to the potentiometer +3.3 (Pin3), Vin (Pin2), and ground (Pin1), as shown in Figure 8.1.

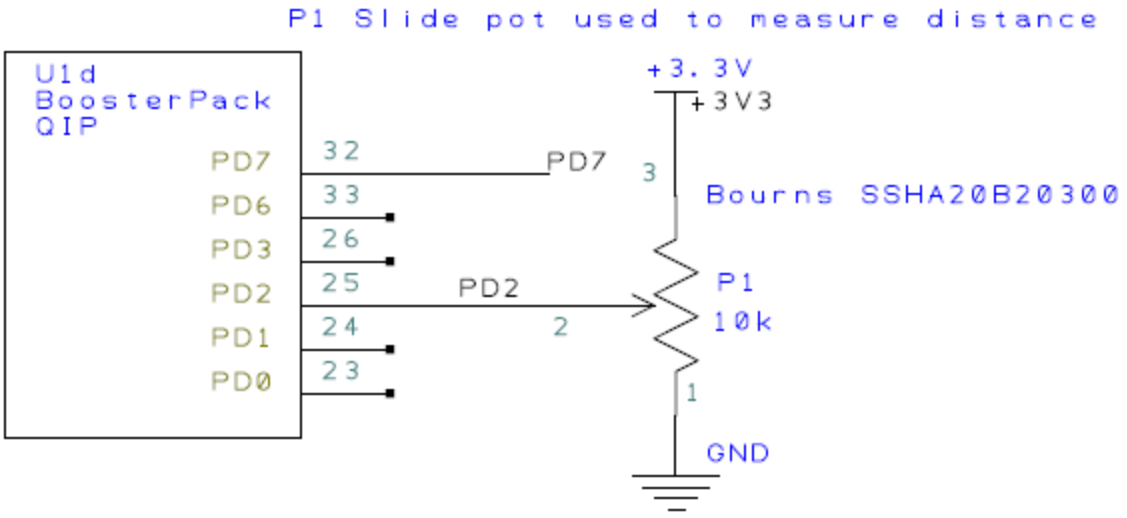


Figure 8.1. Possible circuit to interface the sensor (use your ohmmeter on the sensor to find pin numbers 1,2,3). To use the TExaS oscilloscope, also connect the analog signal to PD3.

In Figure 8.2, the side labeled B has one pin, and the side labeled with C/D has two pins. Connect

B to 3.3V	(shown as pin 3 in Figure 8.1)
C to PD2, the analog signal	(shown as pin 2 in Figure 8.1)
D to ground	(shown as pin 1 in Figure 8.1)

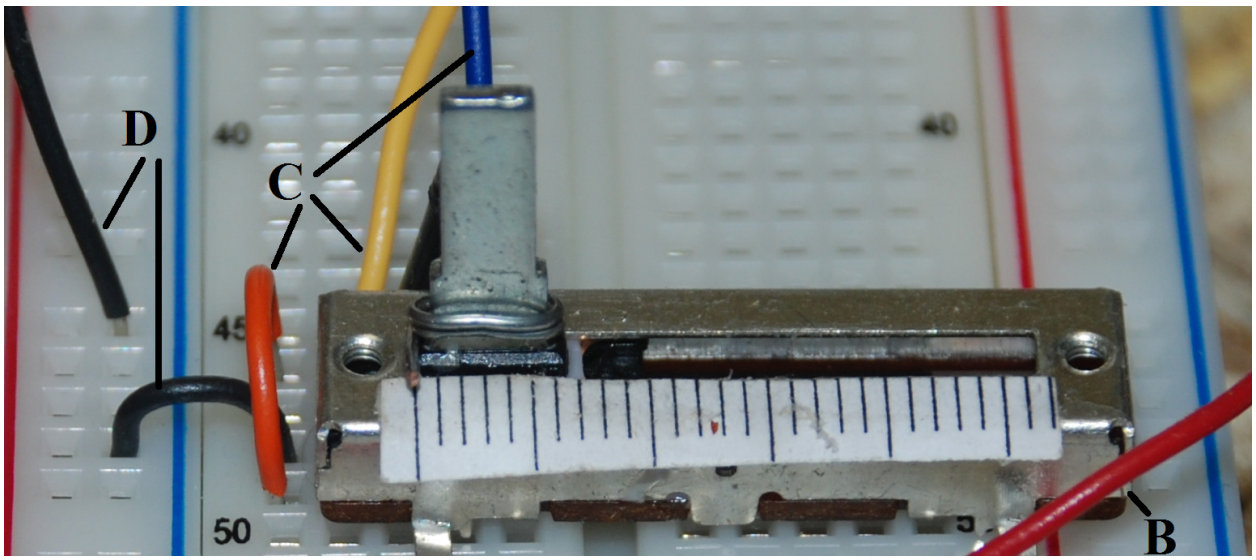


Figure 8.2. Hardware setup for Lab 8, showing the slide pot. The slide pot is used to measure distance on the display you developed in Lab 7. [replace ST7735 with SSD1306]

~~As an **option**, you could solder three male-male wires to the slide potentiometer and glue the pot to a piece of wood or plastic. If you do not know how to solder, ask your TA for a lesson. There is a soldering station in the lab floor. Please wash your hands after handling the solder, please solder in a well ventilated area, and if you drop the soldering iron let it fall to the ground (don't try to catch it). If you are pregnant or think you might be pregnant, please do not solder.~~

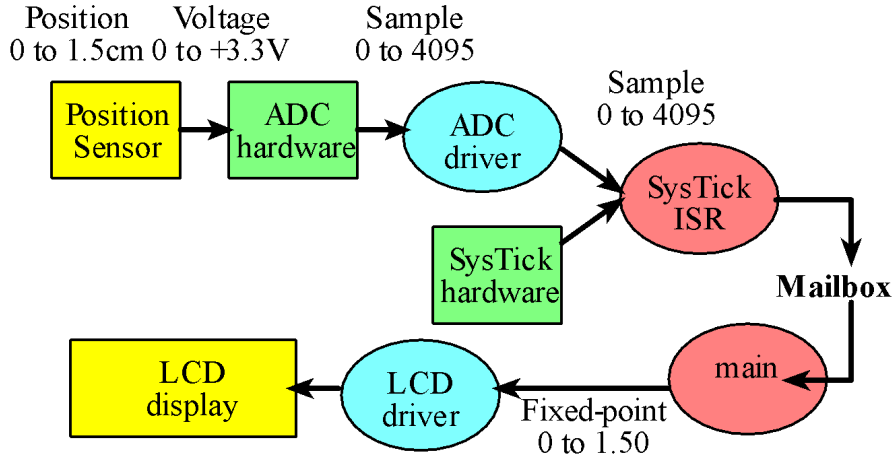


Figure 8.3. Data flow graph and call graph of the position meter system. Notice the hardware calls the ISR. The fixed-point resolution in Lab 8 will be 0.01 cm. In Spring 2021, the sampling rate is 10 Hz.

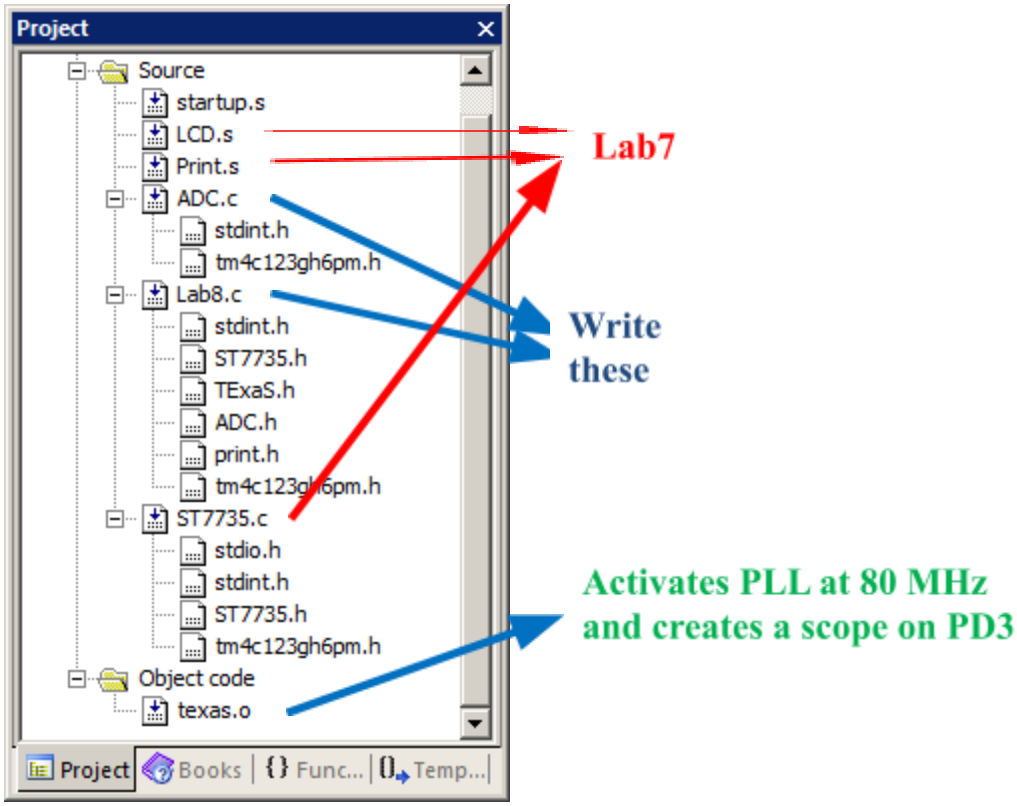
Part b - Design Software System

You will notice the Lab 8 starter project has a main program Lab8.c and three modules (SSD1306, ADC, and TExaS). The main program must be in C, but ADC, and SysTick can be in C or assembly. One possible organization is illustrated in Figure 8.4. Each module must have a header file containing the prototypes for public functions (e.g., SSD1306.h, ADC.h and TExaS.h). The modules will be responsible for the different functions of the embedded system as follows:

- The ADC module abstracts the ADC hardware
- The SSD1306 module abstracts the I2C and LCD hardware (developed in lab 7)
- The TExaS module abstracts Timer5, ADC1, PD3, UART0 and the PLL

In Spring 2021, EE312H/EE319H students: You will notice the Lab 8 starter project a main program Lab8.cpp and four modules (SSD1306, SlidePot, PLL, and TExaS). The main program, SysTick interrupts, and SlidePot must be in C++. One possible organization is illustrated in Figure 8.4b. Each module must have a header file containing the prototypes for public functions (e.g., SSD1306.h, SlidePot.h and TExaS.h). The modules will be responsible for the different functions of the embedded system as follows:

- The SlidePot module abstracts the ADC hardware and SlidePot software
- The SSD1306 module abstracts the I2C and LCD hardware (developed in lab 7)
- The TExaS module abstracts Timer5, ADC1, PD3, UART0
- The PLL module abstracts the phase lock loop (sets clock to 80 MHz)



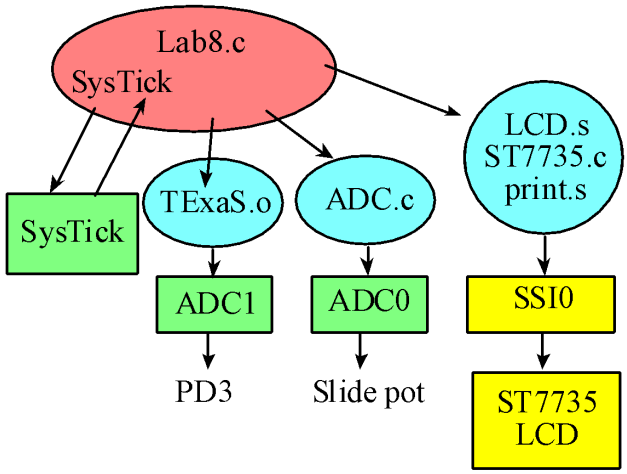


Figure 8.4. Screenshot showing one way the project could be organized. The main program must be C and the LCD driver must be the combination of the three files LCD.s, SSD1306.c and Print.s. The ADC could be C or assembly. [replace ST7735 with SSD1306]

The bottom part of Figure 8.4 shows a possible call graph and Figure 8.3 shows a possible data flow graph. The Lab8 (main) calls ADC, SSD1306, and TExaS. The TExaS module will activate the PLL at 80 MHz, sample PD3 using ADC1 at 10 kHz, and send the data via UART0 to the PC so it can be displayed by **TExaSdisplay**.

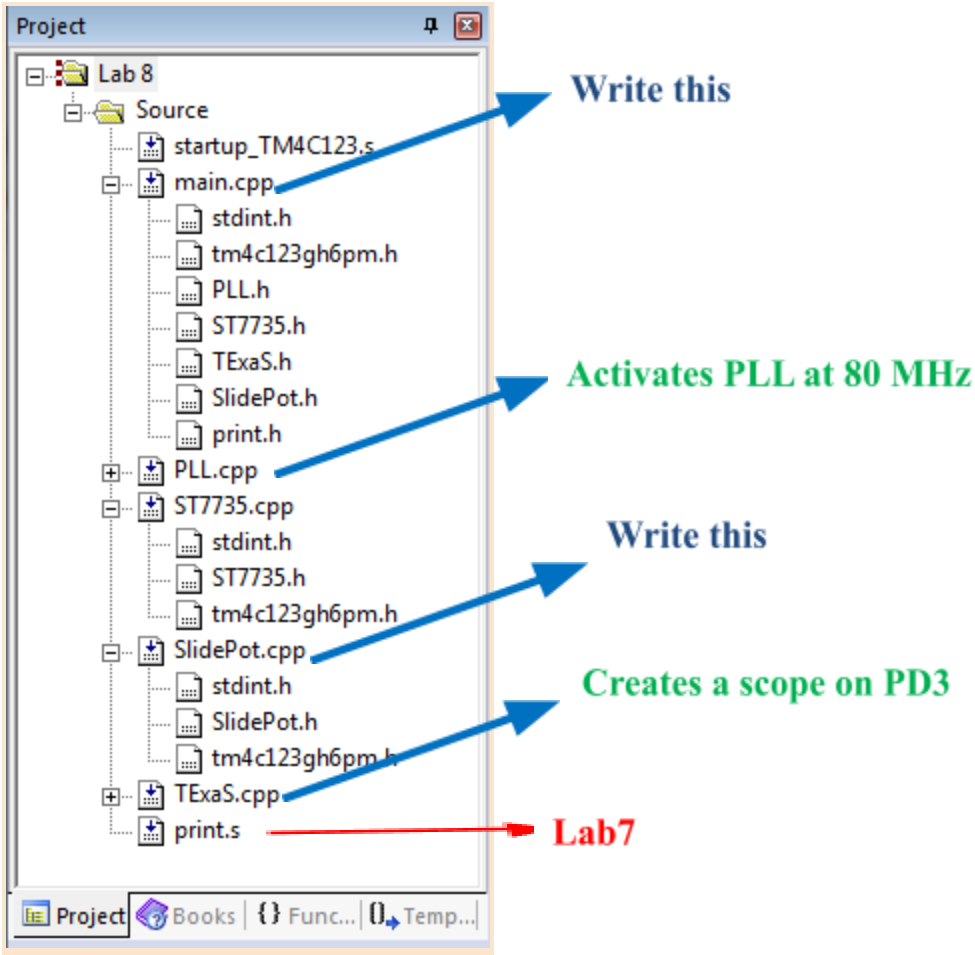


Figure 8.4b. In Spring 2021, EE312H/EE319H screenshot showing one way the project could be organized. The main program and SlidePot must be C++ and the LCD driver must be the combination of the two files SSD1306.cpp and print.s. [replace ST7735 with SSD1306]

Part c - Implement ADC Driver

Write two functions, **ADC_Init** and **ADC_In**, which will initialize the ADC hardware and will sample the ADC, respectively. You are free to pass parameters to these two functions however you wish. Unless your PD2 pin is broken, you must use it for the ADC input. You can use whichever sequencer you wish. You cannot use PD3, because PD3 is used by TExaS.

A **`main1()`** function which tests these two ADC functions is provided in the starter code. In this system, there is no LCD, and there are no interrupts. Debug this system on the real TM4C123 to show the sensor and ADC are operational. The first main program looks something like the following:

```
uint32_t Data;      // 12-bit ADC
int main1(void){    // single step this program and look at Data
  TExaS_Init();     // Bus clock is 80 MHz
  ADC_Init(SAC_NONE);   // turn on ADC, set channel to 5
  while(1){
    Data = ADC_In();    // sample 12-bit channel 5
  }
```

}

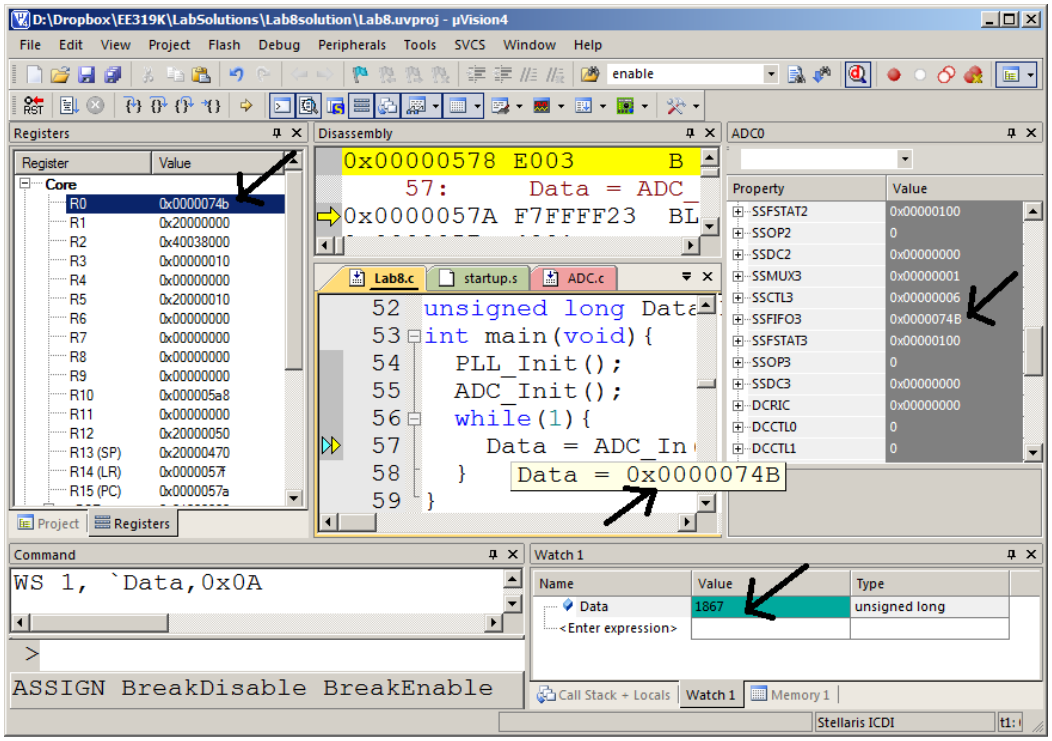


Figure 8.5. Screen shot showing one way to debug the ADC software on the board. You adjust the slide potentiometer, and then see the ADC result. You can observe the ADC result in three places. The ADC result is in R0 after the call to the function; it is stored in the variable Data; and it is in the ADC0 register SSFIFO3.

Part d - Calibrate Sensor and ADC

The starter code provides another main function called **main2** which you can use to collect calibration data. In particular, **main2** will sample the ADC and display the results as unsigned decimal numbers on the OLED. The results can also be seen by observing the variable **Data** in the debugger watch window.

The first step is to choose the amount of hardware averaging you will use in Lab 8, which is set by the parameter passed into **ADC_Init**. More specifically, your **ADC_Init** function will set the register **ADC0_SAC_R** with a number from 0 to 6. Setting **ADC0_SAC_R** equal to 5 specifies a 32-point hardware averaging, meaning the ADC is converted 32 times and the average result is returned. Hardware averaging improves signal to noise ratio but increases the time to perform the ADC conversion.

Once you have chosen your hardware averaging, the second step is to observe how long various functions take to execute. Each time you sample the ADC and print to the LCD, main2 measures the time to perform the ADC_In, OutDec, Convert, and OutFix operations. In this system, there is no mailbox, and there are no interrupts. The function **LCD_OutDec** and **LCD_OutFix** were developed in lab 7. The **main2** uses SysTick to measure the execution time. Since we will eventually be sampling at 10 Hz. Each of these four operations will be executed once every 100ms. Estimate the percentage of time (execution time divided by 100ms) for each of the four operations. The **main2** program is similar to the following:

```
uint32_t startTime,stopTime;
uint32_t ADCtime,OutDectime; // in usec
uint32_t Data;       // 12-bit ADC
int main2(void){
  TExaS_Init();      // Bus clock is 80 MHz
  NVIC_ST_RELOAD_R = 0x00FFFFFF; // maximum reload value
  NVIC_ST_CURRENT_R = 0;     // any write to current clears it
```

```
  NVIC_ST_CTRL_R = 5;
  ADC_Init(SAC_32);    // turn on ADC, set channel to 5
  // 32-point averaging
  SSD1306_Init(SSD1306_SWITCHCAPVCC);
  while(1){          // use SysTick
    startTime= NVIC_ST_CURRENT_R;
    Data = ADC_In();  // sample 12-bit channel 5
    stopTime = NVIC_ST_CURRENT_R;
    ADCtime = ((startTime-stopTime)&0x0FFFFF)/80; // usec
    SSD1306_SetCursor(0,0);
    startTime= NVIC_ST_CURRENT_R;
    LCD_OutDec(Data);
    SSD1306_OutString("    ");  // spaces cover up characters from last output
    stopTime = NVIC_ST_CURRENT_R;
    OutDectime = ((startTime-stopTime)&0x0FFFFF)/80; // usec
  }
}
```

The third step is to calibrate your sensor. Collect five calibration points and create a table showing the true position (as determined by reading the position of the hair-line on the ruler), and the ADC sample (**Data** as measured with **main2**). The full scale range of your slide pot will be different from the slide pot of the other students, which will affect the gain (**Position** versus **Data** slope). Where you attach the paper ruler will affect the offset. Do not use the data in Table 8.1. Rather, collect your own data. Plot **Position** (in 0.01cm units) as a function of ADC sample, like Figure 8.7. Fit this data to a straight line to find the calibration curve. The Excel file **Calibration.xls** can be used to complete the calibration process. You enter your data, and it uses linear regression to find a fixed-point calibration equation.

Position	ADC **Data**
30	192
70	1190
110	2238
140	3006
180	4016

Table 8.1. Calibration results of the conversion from ADC sample to fixed-point (collect your own data).

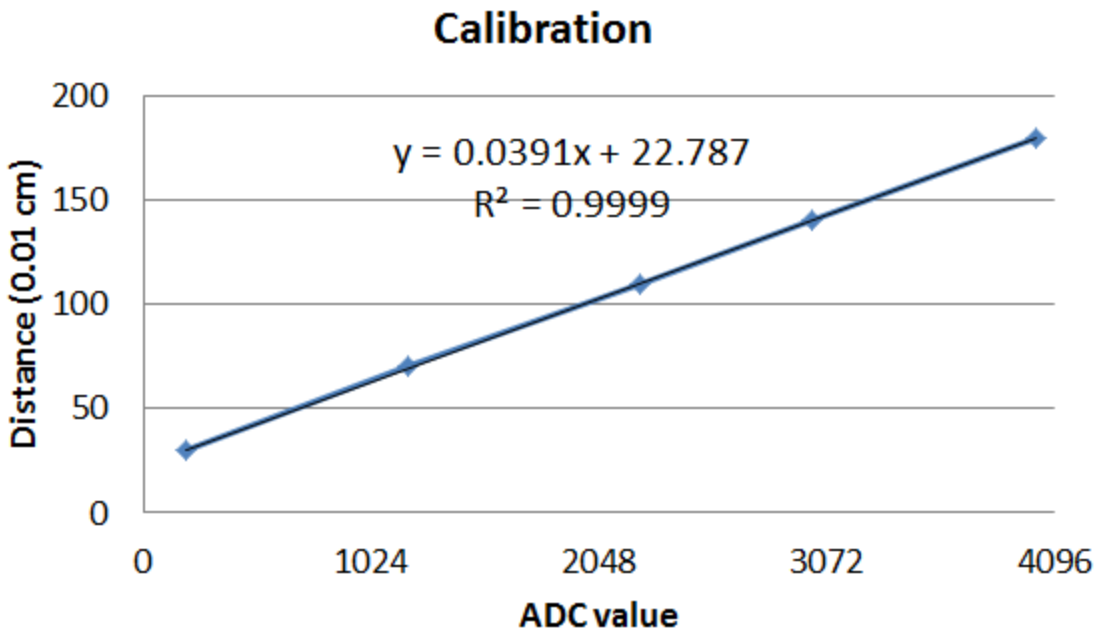


Figure 8.7. Plot of the conversion transfer function calculating fixed-point distance as a function of ADC sample.

Convert the calibration curve from floating point to fixed point. For this data, we change

Position = 0.0391***Data**+22.787

Into

Position = (160***Data**)/4096+23

Part e - Test your Convert function

Use this calibration data to write a function in C that converts a 12-bit binary ADC sample into a 32-bit unsigned fixed-point number (units of 0.01 cm). The input parameter (12-bit 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 can use a linear equation to convert the ADC sample into the fixed-point number. Please consider overflow and dropout during this conversion. **Floating-point numbers are not allowed in this lab**.

```
uint32_t Convert(uint32_t data){
  return (160*data)/4096+23;
}
// DO NOT USE THESE CONSTANTS,  MEASURE YOUR OWN
```

If you notice that your ADC values are not linear, you could use a piecewise linear interpolation to convert the ADC sample to position (Δ of 0.01 cm). In this approach, there are two small tables **Xtable** and **Ytable**. The **Xtable** contains the ADC results and the **Ytable** contains the corresponding positions. This function first searches the **Xtable** for two adjacent of points that surround the current ADC sample. Next, the function uses linear interpolation to find the position that corresponds to the ADC sample. *We do not expect you to use linear interpolation.*

Use the third main function in the starter code, called **main3**, that samples the ADC and displays the results to the LCD as a fixed point number to verify your calibration. The third program look similar to the following example.

```
int main3(void){
  TExaS_Init();         // Bus clock is 80 MHz
  SSD1306_Init(SSD1306_SWITCHCAPVCC);
  ADC_Init(SAC_32);  // turn on ADC, set channel to 5
 // 32-point averaging
  while(1){
    Data = ADC_In();  // sample 12-bit channel 5
    Position = Convert(Data);
    SSD1306_SetCursor(0,1); // second row
    LCD_OutDec(Data); SSD1306_OutString("   ");
    SSD1306_SetCursor(0,2); // third row
    LCD_OutFix(Position);   // your Lab 7 solution
  }
}
```

Part f1 - Use Systick to sample at 10 Hz (EE319K students)

Write a C function, **SysTick_Init,** which will initialize the SysTick system to interrupt at exactly 10 Hz (every 0.1 second). We initialize the SysTick timer to 10 Hz, as this sample rate is large enough to capture most changes in the position along the ruler, and small enough so that if we wanted to we could save much power by turning the microcontroller off for much of the time.

Write a C SysTick interrupt handler that samples the ADC and enters the data into the mailbox. Using the interrupt as synchronization, the ADC will be sampled at equal time intervals. Then toggle a heartbeat LED once at the beginning of the ISR creating one edge each time the ADC is sampled. The frequency of the pulses is a measure of the sampling rate. If you connect the oscilloscope to the LED pin, you can also measure the execution time of the ISR. The interrupt service routine performs these tasks, in this order:

Since you do not have access to a real scope, toggle the heartbeat once

1. toggle heartbeat LED (change from 0 to 1, or from 1 to 0)
2. sample the ADC
3. save the 12-bit ADC sample into the mailbox **`ADCMail`**
4. set the mailbox flag **`ADCStatus`** to signify new data is available
5. return from interrupt

You must use a mailbox to pass data between ISR and main program.

Part f2 - Sample ADC with Systick (EE312H/EE319H students)

Write a C++ SysTick interrupt handler that samples the ADC and saves the data into the SlidePot class. The class will implement the mailbox communication. Using the interrupt as synchronization, the ADC will be sampled at equal time intervals. Toggle a heartbeat LED once at the beginning of the ISR so the LED frequency will be ½ the sampling frequency. The interrupt service routine performs these tasks, in this order:

Since you do not have access to a real scope, toggle the heartbeat once

1. toggle heartbeat LED (change from 0 to 1, or from 1 to 0)
2. sample the ADC
3. save the 12-bit ADC sample using the member function **Save**
4. return from interrupt

The member function **Save** performs these tasks, in this order:

1. Saves the 12-bit ADC sample into a private variable of the class (part of mailbox)
2. Calculates distance
3. Sets the semaphore flag **`flag`** to signify new data is available (part of mailbox)

You must use the **Save** and **Sync** functions to synchronize between ISR and main program.

Part g - Study the Nyquist Theorem

Nyquist Theorem - If f_{max} is the largest frequency component of the analog signal, then you must sample at least as frequently as twice f_{max} in order to correctly represent the signal. For example, if the analog signal is A + B sin(2πft + φ) and the sampling rate is greater than or equal to 2f, you will be able to determine A, B, f, and φ from the digital samples.

Valvano Postulate - If f_{max} is the largest frequency component of the analog signal, then you must sample more than ten times f_{max} in order for the reconstructed digital samples to look like the original signal to the human eye when plotted on a voltage versus time graph.

A derivation of the 10 Hz sample rate follows.

One way to estimate the required sample rate of the position signal is to measure the maximum velocity at which you can move the armature. For example if you can move the armature 2 cm in 0.1sec, its velocity will be 20cm/sec. If we model the position as a signal sine wave $x(t)$=1cm*sin(2πft), we calculate the maximum velocity of this sine wave to be 1cm*2πf. Therefore, we estimate the maximum frequency using 20cm/sec = 1cm*2πf, to be 3 Hz.

A simpler way to estimate maximum frequency is to attempt to oscillate it as fast as possible. For example, if we can oscillate it 4 times a second, we estimate the maximum frequency to be 4 Hz. According to the Nyquist Theorem (stated below), we need a sampling rate greater than 8 Hz. Consequently, you will create a system with a sampling rate of 10 Hz.

Run **main5** to study the Nyquist Theorm. Data are sampled at 10 Hz and plotted on the OLED. Move the slide pot sinusoidally about 1 cycles per second and observe the position versus time plot. Next oscillate the slide pot faster than 5 times per second and notice you are no longer able to observe the true position.

Part h- Study the CLT (EE312H/EE319H students)

Central Limit Theorem (CLT) states as independent random variables are added, their sum tends toward a Normal or Gaussian distribution. To use the CLT we assume the noise of one sample is independent from the noise in other samples. Run **main4** and observe the PMF of the noise as a function of hardware averaging. There are two major observations you should look for: the shape of the noise PMF and the signal to noise ratio (SNR=the average value divided by width of the PMF).

Part i1 - Bring It All Together (EE319K)

Write your own main program which initializes the PLL, timer, LCD, ADC and SysTick interrupts. After initialization, this main program performs these five tasks in order, in the foreground, repeatedly.

1. wait for the mailbox flag **`ADCStatus`** to be true
2. read the 12-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 200)
5. output the fixed-point number on the LCD with units

Debug this system on the real TM4C123. Use the TExaSdisplay oscilloscope to observe the sampling rate (connect LED signal to PD3). Take a photograph or screenshot of the LED toggling that verifies the sampling rate is exactly 10 Hz, as is shown in Figure 8.8.

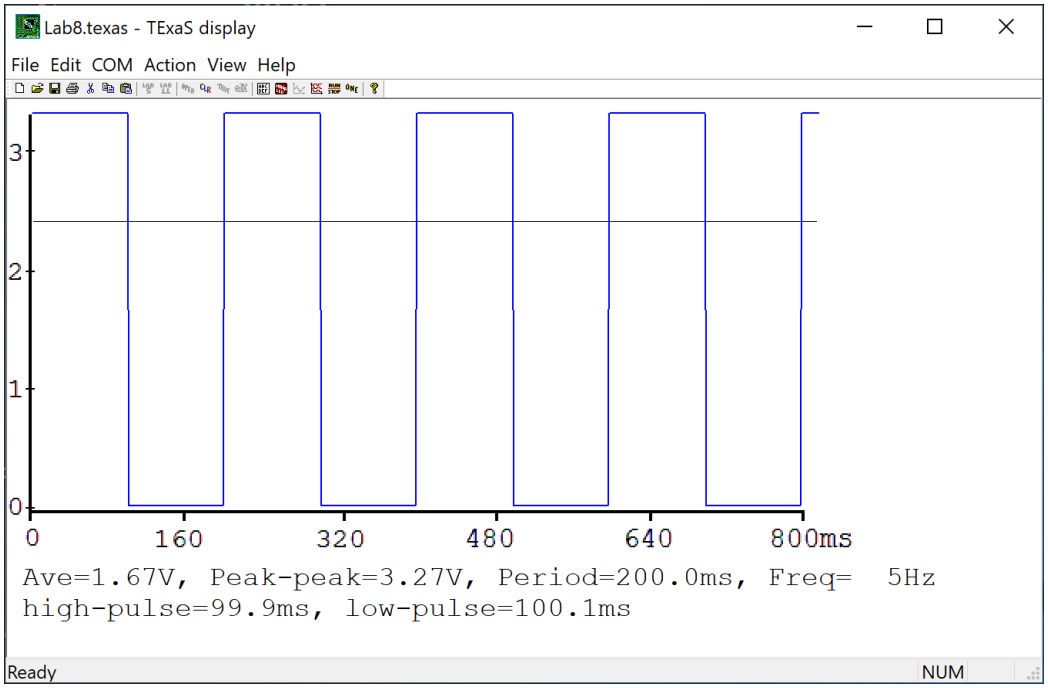


Figure 8.8. The oscilloscope connected to the LED pin verifies the sampling rate is 10 Hz .

Part i2 - Bring It All Together (EE312H/EE319H students)

Write your own main program which initializes the PLL, timer, LCD, ADC and SysTick interrupts. After initialization, this main program performs these five tasks in order, in the foreground, repeatedly.

1. Call sync to wait for the semaphore **`flag`** to be true
2. Retrieve the new distance measurement as a fixed-point number (variable integer is 0 to 200)
3. output the fixed-point number on the LCD with units

Debug this system on the real TM4C123. Use the TExaSdisplay oscilloscope to observe the sampling rate (connect LED signal to PD3). Take a photograph or screenshot of the LED toggling that verifies the sampling rate is exactly 10 Hz, as is shown in Figure 8.8.

Part j - Determine maximum and average error

Use the system to collect another five data points, creating a table showing the true position (x_{ti} as determined by reading the position of the hair-line on the ruler), and measured position (x_{mi} using your device). See the **Lab8_Accuracy** section in chapter 14 of the ebook

http://users.ece.utexas.edu/%7Evalvano/Volume1/E-Book/C14_ADCdataAcquisition.htm

If you collect data near the two extremes of the slide pot you will have larger errors. Don't worry about errors near the extremes, these errors are typical of inexpensive transducers. In the left column place the true distances as determined by your eyes looking at the cursor and the ruler. In the right column place the measured distances as determined by your system. When you have entered at least five sets of data, click the "Calculate" button. Record the maximum and average error in Lab report. *We do not care how small your error is, just that you understand the process.*

To calculate average error by calculating the average difference between truth and measurement,

Average accuracy (with units in cm) =

True position x_{ti}	Measured Position x_{mi}	Error $x_{ti} - x_{mi}$

Table 8.2. Accuracy results of the position measurement system.

In your lab report, answer these open-ended questions (one or two sentences each):

1) What factors limited the accuracy of the device?
2) What conclusions can you make based on this data in Table 8.2?
3) EE319H: what conclusions can you make based on the CLT?

Demonstration

(both partners must be present, and demonstration grades for partners may be different)

Only one of the partners needs to build/show the circuit

There are grading sheets for every lab so you know exactly how you will be evaluated. You will show the TA your program operation on the actual TM4C123 board. The TA may look at your data and expect you to understand how the data was collected and how the ADC and interrupts work. You should be able to explain how the potentiometer converts distance into resistance, and how the circuit converts resistance into voltage. Also be prepared to explain how your software works and to discuss other ways the problem could have been solved. What will you change in your program if the potentiometer were to be connected to a different ADC pin? How would this system be different if the units of measurement were inches instead of cm? What's your sampling rate? What do you mean by sampling rate? What is the ADC range, resolution and precision? How do you initialize SysTick interrupt? How can you change your sampling rate? Be prepared to prove what the sampling rate is using a calculator and the manual. Explain how, when an interrupt occurs, control reaches the interrupt service routine. Why is it extremely poor style to output the converted data to the LCD inside the SysTick ISR? Where is the interrupt vector located? What are the differences between an interrupt and a subroutine? What will happen if you increase your sampling rate a lot? At what point do you think your program will crash? What is the Nyquist Theorem? How does it apply to this lab?

Do all these well in advance of your checkout

1. **Signup for a Zoom time with a TA. If you have a partner, then both must be present**
2. **If you cannot live-stream video, create a 60-sec YouTube video and upload it**
3. **Upload your software to canvas, make sure your names are on all your software**
4. **Upload your one pdf with deliverables to Canvas**

Do all these during Zoom meeting

1. **Have your one pdf with deliverables open on your computer so it can be shared**
2. **Have Keil Lab 8 open so TA can ask about your code**
3. **Start promptly, because we are on a schedule. If you have a partner, then both must be connected**
4. **Demonstrate lab to TA (YouTube video or livestream video)**
5. **Answer questions from TA in the usual manner**
6. **TA tells you your score (later the TA will upload scores to Canvas)**

Deliverables

Submit your ADC.c and Lab8.c as two individual C files to Canvas. You can remove the main1 main2 main3 main4 and main5 test code. Combine the following components into one pdf file and upload this file also to Canvas. UPLOAD ONLY ONE COPY PER TEAM (names on both). Have the pdf file and Keil open on the computer during demonstration.

0. Your names, professors, and EIDs.
1. Circuit diagram showing the position sensor, hand-drawn or PCB Artist, like Figure 8.1 (part a),
2. Four time measurements showing the ADC/LCD execution time (part d)
3. Calibration data, like Table 8.1 (part d)
4. Observations about the Nyquist Theorem (part g)
5. Observations about the Central Limit Theorem (EE319H) (part h)
6. A photo or screenshot verifying the sampling rate is 10 Hz, like Figure 8.8 (part i)
7. Accuracy data and accuracy calculation, Table 8.2 (part j), including the open ended questions

Optional Feedback : http://goo.gl/forms/rBsP9NTxSy

Hints

1. Debug this lab in parts: debugging each module separately. If you combine two working systems (e.g., ADC and LCD), you should retest the system.
2. There are lots of details in this lab, please ask your instructor or your TA if you are confused.
3. Use your voltmeter to measure the voltage at the ADC input
4. A useful debugging monitor is to count the number of interrupts (to see if the interrupts are occurring)
5. When running on the real TM4C123, you cannot use breakpoints effectively in a real-time system like this, because it takes too long to service a breakpoint (highly intrusive). However, you can dump strategic variables into memory and view that data in a memory window.
6. It takes a long time for the LCD to initialize. It is necessary to allow the LCD to finish initialization before starting the SysTick interrupt sampling. One way to do this is to enable interrupts only after all initializations are complete.
7. The fourth column of Table 8.1 is the desired software output, assuming a linear fit between ADC and position. The fifth column of Table 8.1 is the actual software output using the linear equation implemented as integer operations.

Be careful when connecting the potentiometer to the computer, because if you mistakenly reverse two of the wires, you can cause a short from +3.3V to ground.

Do not use your Lab 7 project because the startup.s file for an assembly main (Lab 7) is different from the startup.s file for a C language main (Lab 8). Use the EE319K_Lab8 starter project. EE312H/EE319H students will use **Lab8_C++**

We expect everyone to use their Lab 7 LCD driver for Lab 8. However, if you cannot get your Lab 7 LCD driver to work, you could use the UART0 instead. The UART0 generates output to any serial terminal, like PuTTy, on your PC. You will need to write a new **OutFix** function, remove TExaS and add the PLL. However, there will be a 10 point penalty if you do not use your Lab 7 software.

If your Lab 7 solution works, but your LCD is broken, ask the TA to show you how to run with the Nokia emulator (runs on the real board, but displays in a window on the PC).

FAQ

1. Is anyone having issues with main2? My program gets stuck at OutDec and when I stop it, it jumps to the hard fault handler.

Make sure you have disabled the interrupts for the ADC. Remember you will be doing a software trigger on the ADC read. To check if the ADC_INTERRUPT configuration is the issue, you could do a DisableInterrupt() call before writing to the LCD and follow it with EnableInterrupt().

2. Where should put the code for the SysTick Init and Handler?

SystickInit can be in any file as long as you can call from the main(). As for the handler, it should be able to share the Mailbox and the corresponding Status Flag with the main(). So, it should be in the same file where main() function exist. Ideally, we would like to see a corresponding .h & .c file for Systick module Modular codes are reusable and thus preferred.

3. I'm confused about where exactly we're supposed to solder wires to the potentiometer. Is it the skinny pins on the ends, the shorter pins on the sides, or the holes at the ends?

You solder on the pins on the end. There should be 3 pins. Two are close together and on the opposite side is 1. The other pins on the sides can be ignored.the pins that are on the long sides, that are extensions of the frame, aren't actually pins I believe. One of the pins on the side with two pins goes to V_in; the other two remaining pins on the potentiometer go to either ground or 3.3V. The V_in pin is specific but the other two can be connected to either ground or 3.3V

You have an option of simply plugging the slide pot into the breadboard, without the need to solder.

4. When I was making the measurements to calibrate the measured length with the ADC samples, when I measure the ADC samples for the distances 1.8 - 2.0 cm, they all have the same exact range of values of ADC samples. Does anyone know what I can do about this?

The ends of the ADC tend to be non-linear. You need to either implement a piece-wise function or see if you can gather enough points to make an accurate conversion. (i.e 2 does not read as 1.8)

5. When testing the main2 on the screen, is it suppposed to be unstable? (i.e. going from 2022-2023-2021 very fast like a blink)

That should be fine if the final product doesn't blink too much. (i.e the screen doesn't flicker between 1.8 and 1.9, but a slow flicker between 1.855 and 1.856 is tolerable.) Also double check your connections and try pressing the slide-pot into the bread board to make sure your design is as stable as possible.

Activate hardware averaging. See the ADC0_SAC_R register.

6. What are ADCMail and ADCStatus? They are not, to my knowledge, defined in any of the programs. Are we supposed to define them ourselves? If so, what do we do with them later? I think it's safe to assume that there is an alternate name in one of the files somewhere.

The term mailbox and "passing data" just refers to how we are getting data from one function to another. In this case, if we are running out SysTick handler at 10Hz and reading ADC_In in that call, how can we get the result of ADC_In to our main loop? Likewise how does the main loop know when there is new data in the mailbox? By writing to a global variable and setting a flag each time ADC_In is read within the SysTick_Handler, our main loop can poll that flag and output the global when it is set, then clear the flag. If your systick is in another C file, make sure to extern it inside of that file's corresponding header file and include it in your main C fil

7. Our solution works in simulator, but is jumps to the hardfault handler on the real board somewhere during a waitforinterrupt(). What would be a reason for this?

Every time you have to wait for the clock or another register to initialize, add more wait commands(NOPs). The hardware may take longer to initialize everything since there are more registers and polling needed. Also, do not use RCGC0 RCGC1 RCGC2 registers anywhere in your software. Using these registers will lead to hard faults. Using RCGCGPIO and RCGCADC, along with friendly coding solved their problems.

8. What if I break one of the pins on the pot? Assuming it is not pin2, you could make a weird (nonlinear) transducer with just pins 1,2. connect pin 2 to 220 resistor, other end of 220 ohm connects to 3.3V; connect pin 2 to your input PD2; connect pin 1 to ground

let d be the integer part of distance 0 to 200 (meaning 0 to 2.0cm)
Let R be the resistance between pins 1,2, 0<R<10000, linear with distance, let R=10000*d/200
V= R/(220+R) (goes from 0 to 3.22), but it will be nonlinear
Let N be the 12-bit ADC (0 to 4095)
3.3*N/4096= 50d/(220+50d)
(220+50d)*3.3*N/4096= 50d

220*3.3*N/4096= 50d-50d*3.3*N/4096
220*3.3*N/4096/50=d*(1-3.3*N/4096)
220*3.3*N/4096/50/(1-3.3*N/4096)=d
calibrate this way, let d = A*N/(1-B*N), where A and B are calibration coefficients