Lab 8. Real-Time Position Measurement System

Preparation

Read Sections 9.1, 9.2, 9.3, 9.4, 9.6, 9.8, 10.1, 10.4, and 10.6

starter project Lab8_EE319K http://users.ece.utexas.edu/~valvano/arm/ADCSWTrigger_4C123.zip

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

System Requirements

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

- use the 12-bit ADC built into the microcontroller.
- sample the ADC with the SysTick at 40 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 0.001 cm.

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.

Procedure

The basic approach to this 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.
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 full scale range may be any value from 1.5 to 2.0 cm. 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.

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

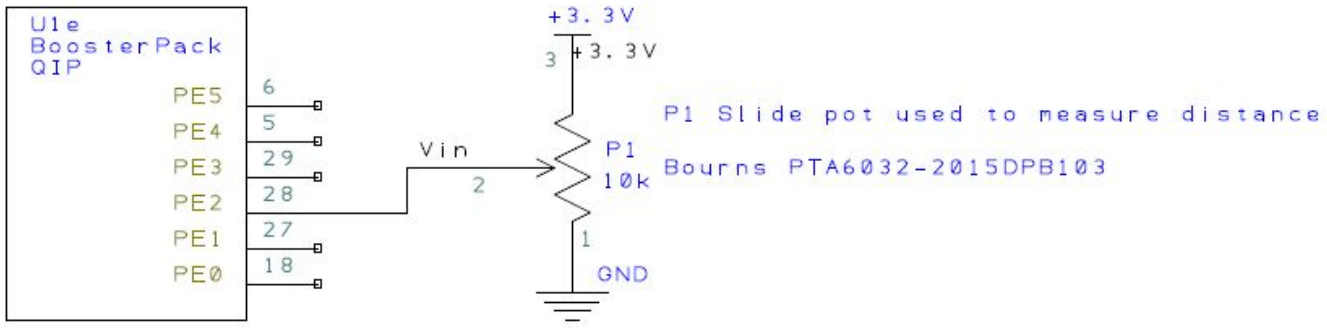

Figure 8.1. Possible circuit to interface the sensor (use your ohmmeter on the sensor to find pin numbers 1,2,3).

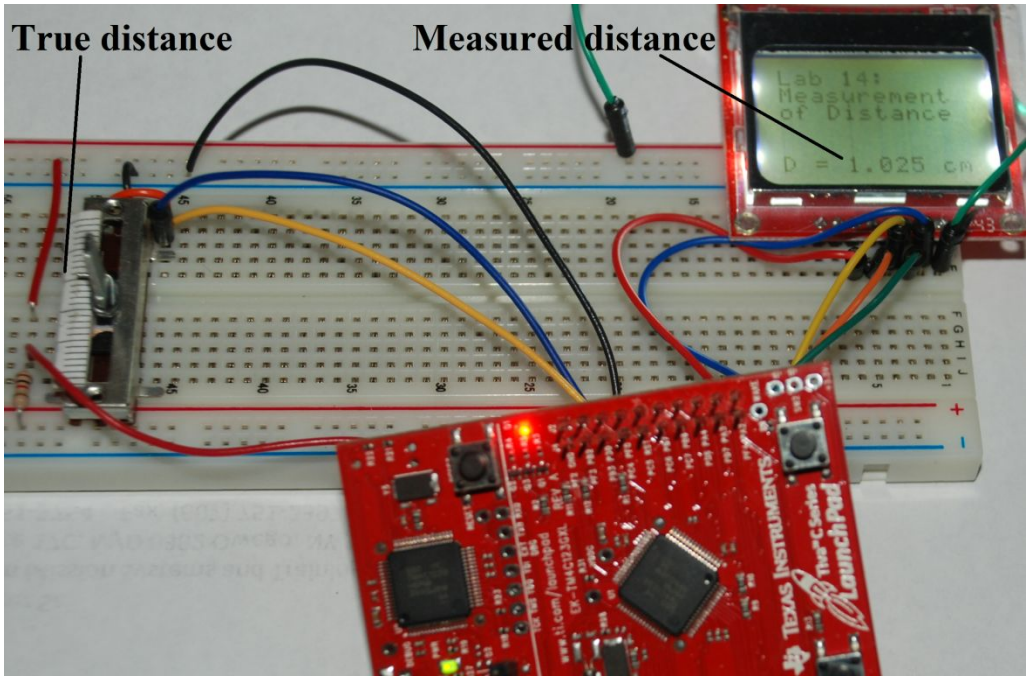

Figure 8.2. Hardware setup for Lab 8, showing the LCD and slide pot. The slide pot is used to measure distance. the display you developed in Lab 7. (The 1k resistor shown in this figure should not be used).

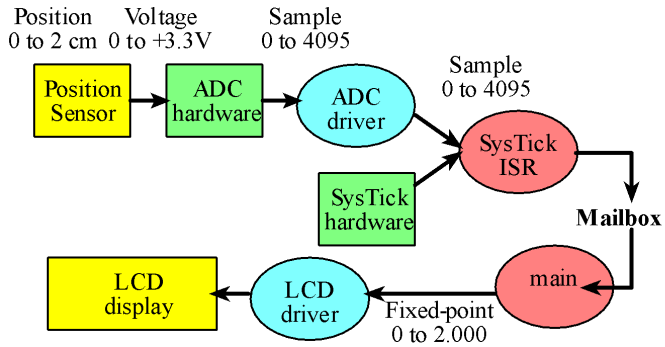

Figure 8.3. Data flow graph and call graph of the position meter system. Notice the hardware calls the ISR.

Part b - Design Software System

You will notice the Lab 8 starter project a main program Lab8.c and three modules (ST7735, 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., ST7735.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 ST7735 module abstracts the SSI and LCD hardware (developed in lab 7)
- The TExaS module abstracts Timer5, ADC1, PD3, UART0 and the PLL

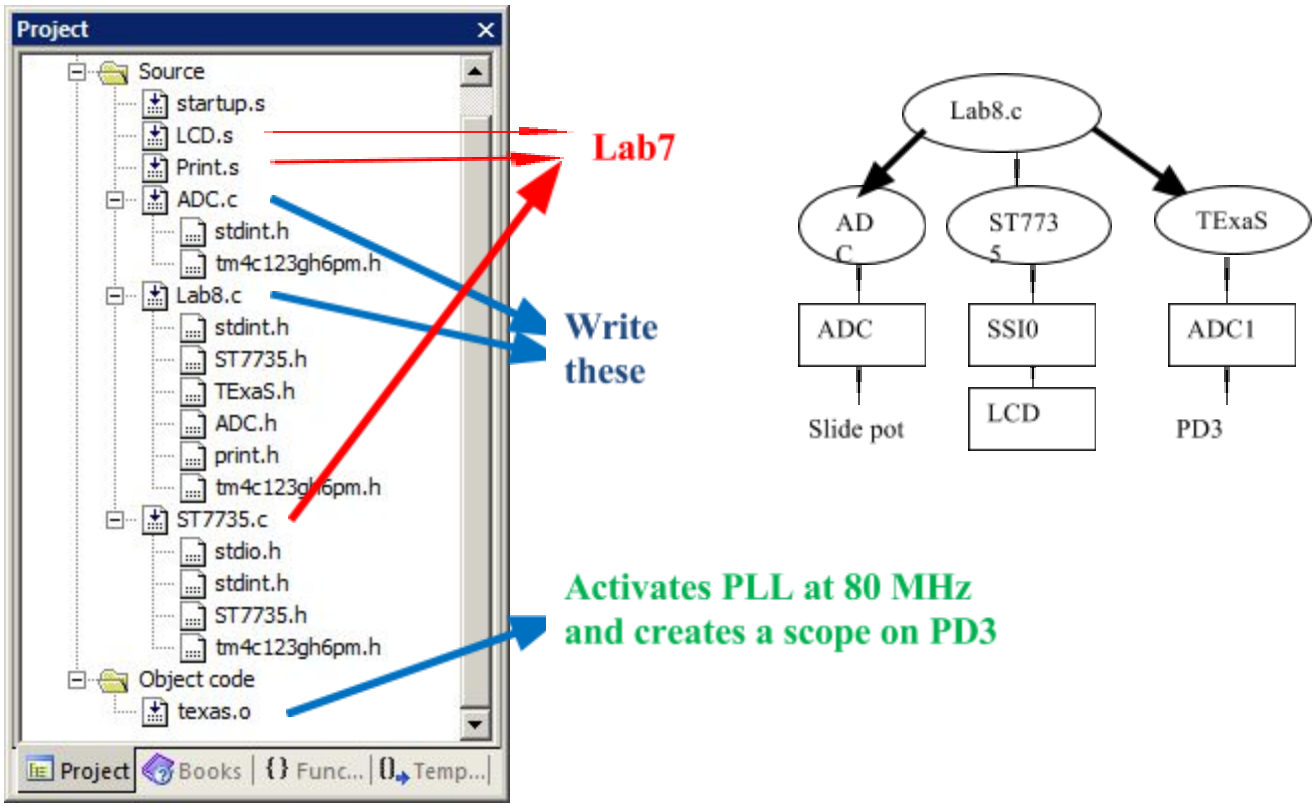

Figure 8.4. Screen shot showing one way the project could be organized. The main program must be C and the LCD driver must be the combination of the three files LCD.s, ST7735.c and Print.s. The ADC could be C or assembly.

The right side of Figure 8.4 shows a possible call graph and Figure 8.3 shows a possible data flow graph. The Lab8 (main) calls ADC, ST7735, 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**.

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 and you are free to use any of the ADC channels. You can use whichever sequencer you wish, but, if you use a different sequencer, you must use a different analog that corresponds the sequencer you use (e.g., pin PE4=Ain9). 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:

```
#include "TExaS.h"
#include "ADC.h"
uint32_t Data;
int main(void){       // single step this program and look at Data
  TExaS_Init();       // Bus clock is 80 MHz
  ADC_Init();         // turn on ADC, set channel to 1, PE2
  while(1){
    Data = ADC_In();  // sample 12-bit channel 1, PE2
```

```
  }
}
```

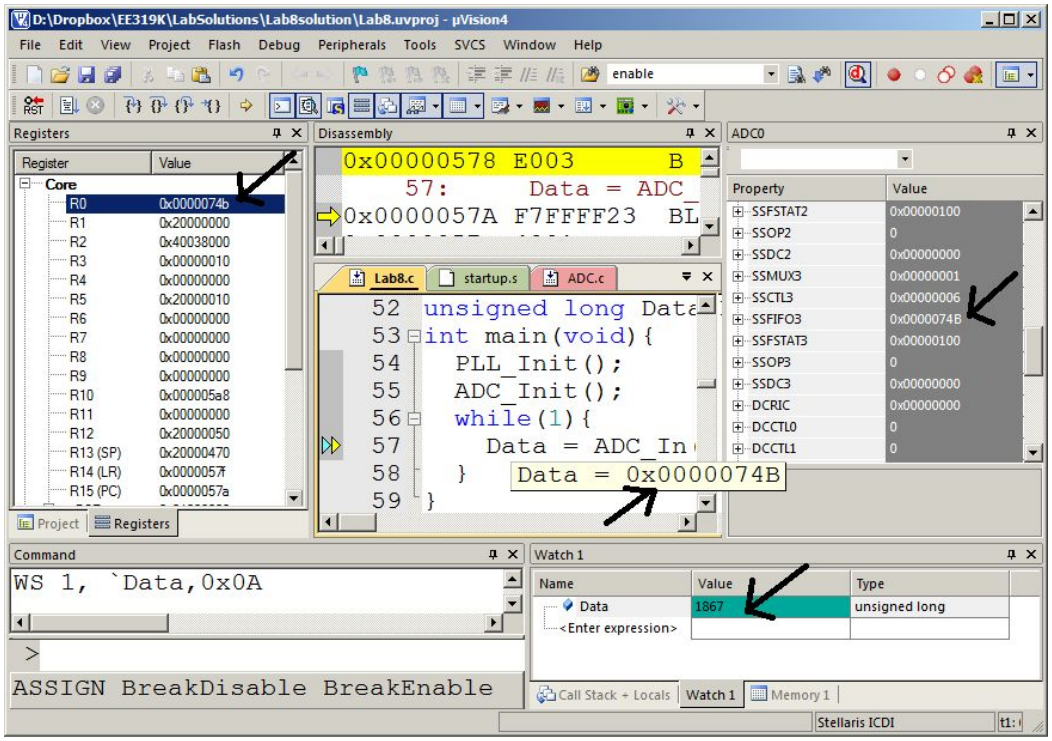

Figure 8.5. Screen shot showing one way to debug the ADC software on the board. You adjust the slide potentiometer, and then see the ADC result. You can observe the ADC result in three places. The ADC result is in R0 after the call to the function; it is stored in the variable Data; and it is in the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

Part d - Calibrate ADC

The starter code provides another main function called **`main2()`** 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. Each time you sample the ADC and print to the LCD, you should toggle the heartbeat LED attached to PF2. In this system, there is no mailbox, and there are no interrupts. You should use your **`LCD_OutDec`** developed in the previous lab.

Connect PF2, your heartbeat LED, to an oscilloscope and use it to measure the time it takes the ADC to perform one conversion, and the time it takes the LCD to output one number. Figure 8.6 shows for this system the ADC requires 9 µsec to convert and the LCD requires (you measure this) µsec to display. The second main program look something like the following:

```
uint32_t Data;
int main(void){
  TExaS_Init();       // Bus clock is 80 MHz
  ADC_Init();         // turn on ADC, set channel to 1
  LCD_Init();
  PortF_Init();
  while(1){           // use scope to measure execution time for ADC_In and LCD_OutDec
    PF2 = 0x04;       // Profile ADC
    Data = ADC_In();  // sample 12-bit channel 1
    PF2 = 0x00;       // end of ADC Profile
    LCD_Goto(0,0);
    PF1 = 0x02;       // Profile LCD
    LCD_OutDec(Data);
```

```
    LCD_OutString("     ");  // these spaces are used to coverup characters from last
output
    PF1 = 0;           // end of LCD Profile
  }
}
```

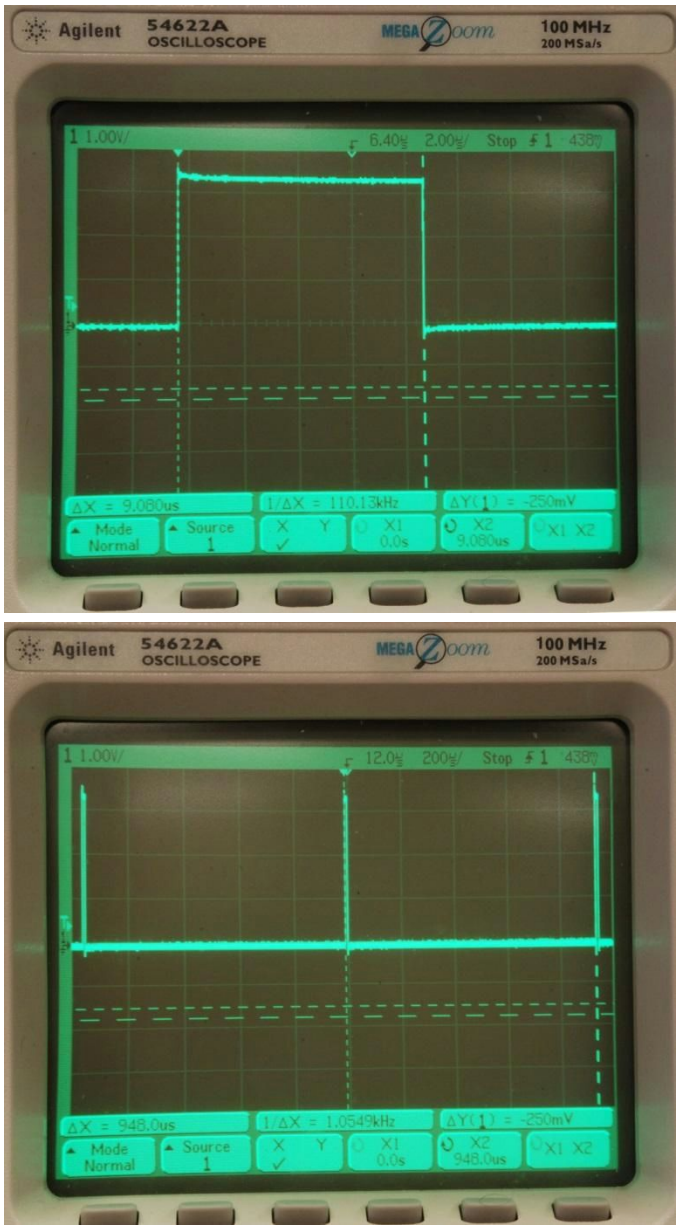

Figure 8.6. Oscilloscope trace showing the execution time profile. There are a lot of parameters that affect speed, so please expect your time measurements to be different. For this system it takes 9µs for the ADC to sample and 9ms to output.

Collect five to ten calibration points and create a table showing the true position (as determined by reading the position of the hair-line on the ruler), the analog input (measured with a digital voltmeter) and the ADC sample (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 versus position 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 like the first three columns of Table 8.1.

Position	Analog input	ADC sample	Correct Fixed-point	Measured Fixed-point Output
0.10 cm	0.432	538	100	84
0.40 cm	1.043	1295	400	421
0.80 cm	1.722	2140	800	797
1.20 cm	2.455	3050	1200	1202
1.40 cm	2.796	3474	1400	1391

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

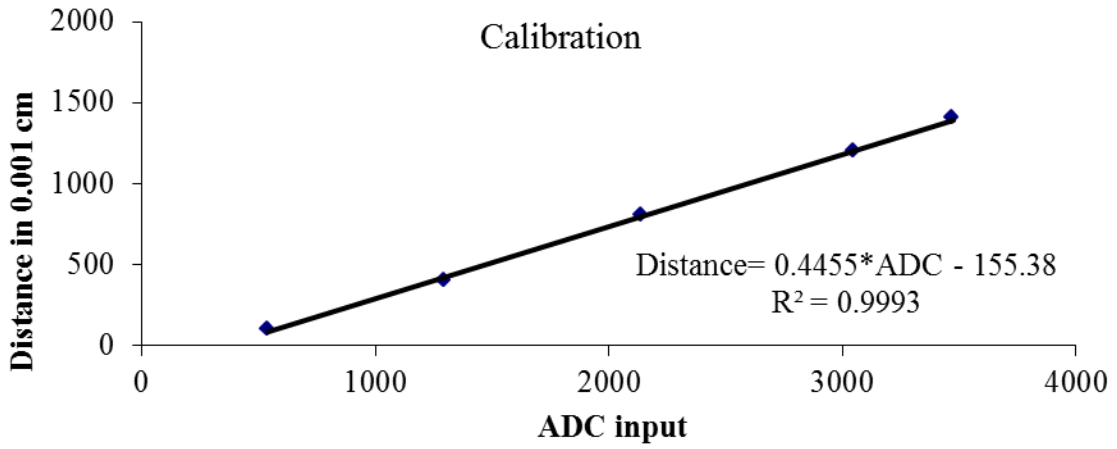

Figure 8.7. Plot of the conversion transfer function calculating fixed-point distance as a function of ADC sample.

Part e - Write Unit Conversion

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. 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). Table 8.1 shows some example results; this data is plotted in Figure 8.7. You may want to 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**. In this system, the calculation **Position = 0.4455*Data-155.38** can be approximated as

```
Position = (114*Data-39777)/256;   // 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 0.001 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.

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.

```
uint32_t Data;      // 12-bit ADC
uint32_t Position;  // 32-bit fixed-point 0.001 cm
```

```
int main(void){
  TExaS_Init();         // Bus clock is 80 MHz
  LCD_Init();
  PortF_Init();
  ADC_Init();         // turn on ADC, set channel to 1
  while(1){
    PF2 ^= 0x04;      // Heartbeat
    Data = ADC_In();  // sample 12-bit channel 1
    PF3 = 0x08;       // Profile Convert
    Position = Convert(Data);
    PF3 = 0;          // end of Convert Profile
    PF1 = 0x02;       // Profile LCD
    LCD_Goto(0,0);
    LCD_OutDec(Data); LCD_OutString("    ");
    LCD_Goto(6,0);
    LCD_OutFix(Position);
    PF1 = 0;          // end of LCD Profile
  }
}
```

Part f - Initialize Systick to 40 HZ

Write a C function, **`SysTick_Init,`** which will initialize the SysTick system to interrupt at exactly 40 Hz (every 0.025 second). If you used SysTick to implement the blind wait for the LCD driver, you will have to go back to Lab 7 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, and you can use SysTick code similar to Lab 6 to implement the 25-ms periodic interrupt needed for this lab.

We initialize the SysTick timer to 40 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. A derivation of the 40 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 10 times a second, we estimate the maximum frequency to be 10 Hz. According to the Nyquist Theorem (stated below), we need a sampling rate greater than 20 Hz. Consequently, you will create a system with a sampling rate of 40 Hz.

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.

Part g - Sample ADC with Systick

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 twice at the beginning of the ISR and once at the end of the ISR, creating a pulse 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:

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

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

Part h - Bring It All Together

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 2000)
5. output the fixed-point number on the LCD with units

Debug this system on the real TM4C123. Use an oscilloscope to observe the sampling rate. Take a photograph or screenshot of the LED toggling that verifies the sampling rate is exactly 40 Hz, as is shown in Figure 8.8.

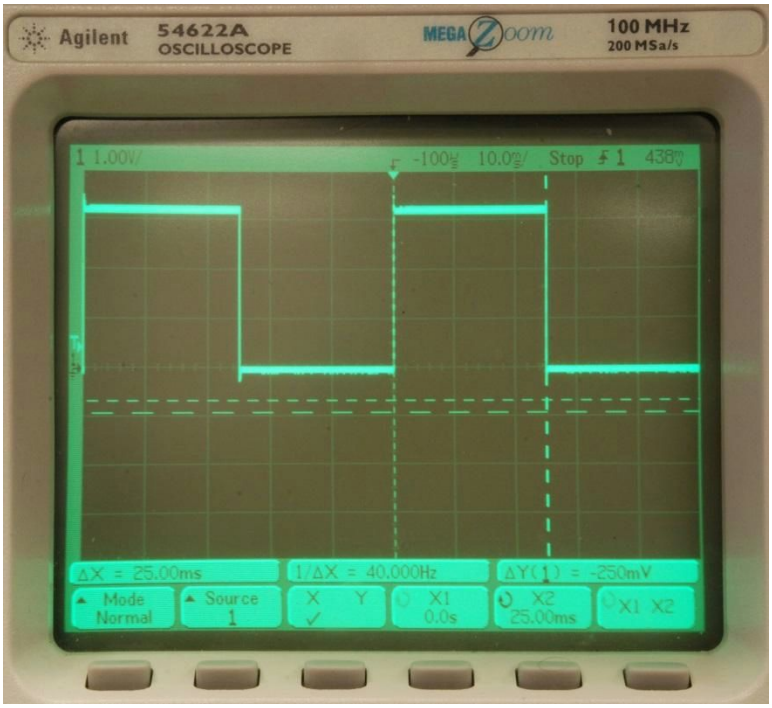

Figure 8.8. The oscilloscope connected to the LED pin verifies the sampling rate is 40 Hz. If you were to zoom in, you could also measure the execution time of the ISR.

Part i - Collect Measurement Data

Use the system to collect another five to ten 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). You can use the interactive tool at http://users.ece.utexas.edu/~valvano/Volume1/Lab8Accuracy.html to calculate average accuracy 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.

Demonstration

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

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?

Deliverables

(Items 1, 2, 3, 4, 5 are one pdf file uploaded to SVN, have this file open during demo.)

0. Lab 8 grading sheet. You can print it yourself or pick up a copy in lab. You fill out the information at the top.

Create one pdf file with the following for electronic submission (UTEID1_UTEID2.pdf***)***

1. Circuit diagram showing the position sensor, hand-drawn or PCB Artist, like Figure 8.1 (part a),
2. Time measurements and a photo showing the ADC/LCD execution time profile, like Figure 8.6 (part d)
3. Calibration data, like the first three columns of Table 8.1 (part d)
4. Final version of distance meter with SysTick, ADC, convert, and main (your code for parts c, e, f, g and h)
5. A photo or screenshot verifying the sampling rate is 40 Hz, like Figure 8.8 (part h)
6. Accuracy data and accuracy calculation, Table 8.2 (part i)
7. Optional Feedback : http://goo.gl/forms/rBsP9NTxSy

Commit to SVN the final version of distance meter with SysTick, ADC, convert, and main (parts c, e, f, g and h). All source files that you have changed or added should be committed to SVN. Please do not commit other file types.

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.

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

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.

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 40Hz 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 RCGC and RCGC2 registers for PORT initialization. Some students have used RCGC for port-e and RCGC2 for port-f, which lead to hard faults. Using RCGC only, along with friendly coding solved their problems.