EE319K Laboratory Manual

Univ of Texas at Austin

Instructor: *Ramesh Yerraballi*; Teaching Assistant: *Sean Duffy*

Do not print the entire document; we will be making many changes.

Summer 2014

Table of Contents

If you find a mistake, or if you find a section vague or confusing, please email me at **ramesh@mail.utexas.edu** . If you enjoy EE319K, you might also enjoy EE445L. Furthermore, if you really like EE319K, you might consider specializing in embedded systems.

http://edx-org-utaustinx.s3.amazonaws.com/UT601x/EE319K_Install.exe to download all examples

Laboratory grading policies: This is a programming class. Therefore, the quality of the software you write will significantly affect your grade. In addition to writing software, each lab has specific activities that must be documented and turned in with the lab. These deliverables will be screenshots, circuit diagrams, measurements or data analyses. When you get the program finished, create one file of your source code, and all deliverables. Save this file as a pdf. Upload the pdf file to Canvas/SVN as instructed, and bring the grading sheet to the demonstration. The TA will record the performance and demonstration grades on the grading sheet. Labs are due during your scheduled lab period. A detailed schedule follows. Late assignments incur a penalty. Exceptionally "good" programs may be given extra credit. If you do extra functions in your program, point out and demonstrate the extra functions to the TA when you demonstrate your program. Partial credit, not to exceed a maximum of 75% of full credit, may be awarded to a program that does not meet all specifications. Often it is better to checkout late incurring penalties than to turn in an incomplete program. There are three components to the lab grade:

1. Deliverables 20% (completion grade, one **pdf** file, uploaded to Canvas/SVN, and showed to TA during demonstration):
2. Performance 35% (graded by the TA at the time of checkout):
   How well does it work? Does it crash?
   Does it handle correctly all situations as specified?
   How clean is the user interface?
   Possible 5% bonus, does it do more than specified?
3. Adhere to coding standard 5% (organization, style, comments etc.) %%%
   Comments explaining each subroutine (inputs, outputs, summary of behavior)
   Modular programming (subroutines do only the tasks they were created for)
   Code is clearly organized (not too many branches)
   Good variable names (divisor is called divisor, result is called result etc.)
   Consistent coding style throughout (bracket placement, proper indentation, variable naming scheme)
4. Demonstration 40% (graded by the TA at the time of checkout):
   Can you explain to the TA how your software works? Be prepared to explain the data flow through your system.
   Can you explain why you made certain software engineering tradeoffs?
   Both partners must be present during the demonstration.

There is a lot of starter code in the book and on the internet. It is natural and appropriate for you to look at these examples. However, during demonstration you are expected to understand both the algorithm used and the implementation details of ALL software you are running at the time of checkout. In particular, the contents of these starter files will be examinable for the lab checkouts as well as for in-class written exams. Even if your program works, you may get very poor score on the entire lab (e.g., 0 to 50%) if you do not understand how it works. For Labs 3 through 9, if your system runs on the simulator, but does not fully run on the real hardware, you may be assessed at least a 20% penalty on the entire lab.

Comments and policies about EE319K

Clearly describe the data and program structures in the software comments. Simple well-structured software with descriptive labels is much easier to understand than complex software with voluminous comments. Use either top-down or bottom up structure. Modular programming divides the problem into "natural divisions" with minimal coupling. The interfaces should be simple and natural. The software will be graded on the correctness of the program function, the comments, the interface to the user, the style, and the organization. The effort spent to write high quality software will be rewarded both during debugging and when it is time to add features in the future.

1) EE319K is a "hands on" class and therefore, even though the labs are only 30% of the total grade, there is NO WAY to make it through this class without taking all the labs seriously. It is during the lab experience that you will really learn.

2) This lab manual describes the tasks for each lab and provides and examples of questions you can be asked. The questions are meant to give you an idea of the topics, depth and scope of the questions. The TA may ask similar but different questions. You may also be asked ANY question related to the material in previous labs.

3) In addition, the lab manual specifies a list of "deliverables". There is a lab grade component for the deliverables and missing some of components will affect the grade. You will create one **pdf file** with all the deliverables. You upload this file to Canvas/SVN as instructed before the beginning of your check-out time. You also open the pdf file on the computer for the TA to see during demonstration. You do not need to make hard copy prints. It is OK to capture low resolution screen shots, select the interesting parts in Paint, and then paste them into the document. (Ctrl-Alt-PrintScreen captures the active window)

4) You must upload (to Canvas) the deliverables before your time slot. Each lab will also have a Lab grading sheet which the TA will print for you. If you begin to create the deliverables file after the checkout started, you will be considered as not ready, and it will impact the grade.

5) You will need to come to lab ahead of your assigned time, turn on your laptop, connect your laptop to your microcontroller if needed and download your code. This way you are ready to demonstrate at the beginning of your assigned checkout time. The checkout times are VERY short, and there will not be time for the TA to check you out if you need to spend half of the checkout time setting up or creating documents.

6) Student(s) not ready with everything set-up on the time assigned, won't be able to check-out during their slot, they will have to do a "late-check-out" and loose at least 5 points if checked the same day. For more info, see table below.

7) For all EE319K labs, students work in teams of two. If you are late or miss your slot, your grade will likely be lower because you will not have time to present your lab to the TA. Partners must have the same lecture and lab time (same unique number). Both partners should signup for the same demonstration slot. You will need one Stellaris LaunchPad (either LM4F120 or TM4C123 version) per team, but you are encouraged to purchase a second board as a back up (boards get lost or broken). If you and your partner did not get along, our first, second and third option is to help you work out your differences. Both partners are still both responsible for the lab even if you have difficulties working together. Only under extreme cases will we split a partnership. It is common for the TA to assign a different grade to each partner based on individual performances during your checkout demonstration. Both partners MUST be present for your assigned time. If your partner does not show up for your assigned time, then one student may demonstrate, and the other student (the one not present) will receive a zero and not be allowed to make it up.

8) Grading: you will receive your lab grade through the Canvas. The TA should upload your lab grade within a week after the check-out. There is no need to send emails and ask about it during that week. However, if you don't see your grade after seven days, please send an email to the TA cc the email to the professor.

9) Please DO NOT PRINT this whole lab manual since it will be updated multiple times throughout the semester. Changes made to the manual after the first lab is due will be marked with **%%%** and colored in RED or BLUE. You can search for the three percent signs and see exactly what has changed. Therefore if you decide to print, it will be best if you print only the part of the manual for the lab on which you are currently working.

10) Office hours: The professor and TA have office hours posted in the Syllabus and on Canvas.

11) Late check-outs: In case you didn't make it to check-out during the slot (on Wednesday) you were assigned, you still can check a lab out later during office hours on Thursday. There is no need to ask/register for that, just show up. If you show up at the end of a TA office hour, there may not be time to check you out. There is a penalty of 10 points late checkout. If you check out during your scheduled time, it is possible to get full credit, but checking out later on Wednesday will be considered late (-5 points). No check-outs will be allowed after Thursday.

12) Lab lecture: The TA will conduct a Lab lecture on Thursday from 2:00 to 3:00 pm to show you the working of the lab for the following week. During the lecture a TA will answer questions you might have, present a possible solution (how your solution is suppose to operate - not the code). You may not take photographs of the constructed hardware shown by the TA. The TA may give some hints and will answer your questions. Therefore we strongly recommend you read the lab manual before coming to the lecture so you'll have some time to think of the lab and see if you have any questions.

13) Communication: Piazza??

14) You can download the software tools we use for the class (Keil uVision and PCB Artist) from the web, install it on you own machine. We know that these tools were successfully used on XP, Vista, Windows 7, Windows 8, Mac* and Linux* (*-using virtual machines). However, we cannot assure you will be able to run it on any OS/hardware.

15) We do pass your lab software through a plagiarism checker. Any copy/pasting from current or previous students constitutes scholastic dishonesty. Basically, you should not be looking at solutions to EE319K labs written by other students (other than your partner of course). See the course syllabus for more information on plagiarism.

Useful web sites

Course material	http://users.ece.utexas.edu/~ryerraballi/sa2014	(information specific to EE319K Study-abroad)
Data sheets	http://users.ece.utexas.edu/~valvano/Datasheets/	
Book examples	http://users.ece.utexas.edu/~valvano/arm/	(example code from the book)
C programming	http://users.ece.utexas.edu/~valvano/embed/toc1.htm	
Setup Keil uVision	http://users.ece.utexas.edu/~valvano/Volume1/uvision/	
Starter files	http://edx-org-utaustinx.s3.amazonaws.com/UT601x/EE319K_Install.exe	

Checkout procedure

1) TA will have a checkout sheet. If you want to look at what the TA will be looking for, you can download and print one from EE319K Study-abroad Course material site (above).

2) Before you checkout you must upload the one pdf file with all the deliverables as listed for each lab.

3) You will need to come to lab ahead of your assigned time, turn on your laptop, connect your laptop to your microcontroller if needed and download your code.

4) The one pdf file should be open on the computer for the TA to see.

5) At the start of your checkout time, you will show the deliverables to your TA and run your program for the TA. The TA may wish to see certain conditions, so be ready to make small changes to your programs.

6) You will be asked questions to determine your level of understanding.

7) At the end of checkout, the TA will record all parts of your lab grade on the grading sheet, so you will know your score at that time. These sheets will not be returned.

To checkout early or late, go to a TA office hour. TA time during office hours is first come first served.

Lab 1. Automobile Door Signal Implemented on a Microcontroller

Preparation

Read Chapters 1 and 2 of the book
Read Sections 3.1, 3.2, 3.3.1-3.3.5, 3.3.9, 3.3.10, 4.1.2 and 4.2 of the book
Install and run Keil uVision on your personal computer
Run EE319K_Install_in.exe to install starter code for labs 1 and 2 and other common files

Purpose

The purpose of this laboratory is to familiarize you with the software development steps using the **uVision** simulator. Starting with Lab 2, we will use uVision for both simulation and debugging on the real board, but for this lab, we will use just the simulator. You will learn how to perform digital input/output on parallel ports of the LM4F120/TM4C123. Software skills you will learn include port initialization, logic operations, and unconditional loop.

System Requirements

The specific device you will create is a alarm signal with two binary sensor inputs and one LED output. The LED output represents the alarm signal, and the operator will toggle the switches to simulate the doors. Let **PE2** (Port E bit 2) be the Boolean variable representing the alarm signal (0 means LED is off and it is safe, 1 means LED is on and it is unsafe). Let **PE3** (Port E bit 3 is the left-door sensor) and **PE4** (Port E bit 4 is the right-door sensor) be Boolean variables representing the state of the two doors using negative-logic (1 means the door is open, and 0 means the door is closed). The LED should come ON if either of the doors or both are open.

Procedure

The basic approach to this lab will be to develop and debug your system using the simulator.

Part a) To create a Lab 1 Project, perform these tasks. Find a place on your harddrive to save all your LM4F120/TM4C123 software.

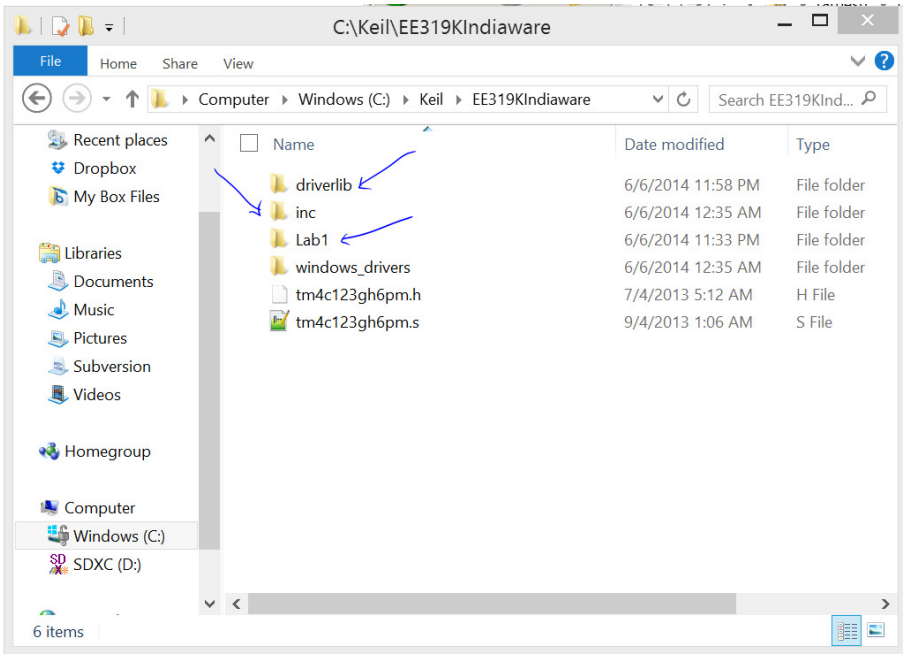


Figure 1.1. Directory structure with your Lab1 together with driverlib and inc directories.

Download and unzip the starter configuration from http://users.ece.utexas.edu/~ryerraballi/UT601x/EE319KIndiaware.zip into this location. It is important for the directory structure to look like Figure 1.1. As you work on each new lab you will download and extract the provided zip file into this location.

Begin by opening the **Lab1** project folder. Either double click the **uvproj** file or open the project from within uVision. Make sure you can compile it and run on the simulator. Please contact the TA if the starter project does not compile or run on the simulator. **`Startup.s`** contains assembly source code to define the stack, reset vector, and interrupt vectors. All projects in

this class will include this file, and you should not need to make any changes to the **`Startup.s`** file. **`LabX.c`** will contain your C source code for this lab. You will edit the C file.

Add your names and the most recent date to the comments at the top of Lab1.c. This code shows the basic template for the first few labs.

```
// ****************** Lab1.c ***************
// Program written by: put your names here
// Date Created: 6/5/2014
// Last Modified: 6/5/2014
// Brief description of the program
//Preprocessor directives #define and #include go here

int main(void){    volatile unsigned long delay;
// one time initialization goes here

  while(1){
    // operations to be executed over and over go here
  }
}
```

Program 1.1. C language template.

To run the Lab 1 simulator, you must do two things. First, execute Project->Options and select the Debug tab. The debug parameter field (Figure 1.2) must include **-dInLab1**. Second, make sure the InLab1.dll file is added to your Keil\ARM\BIN folder. The debugging dynamic link libraries should go into Keil\ARM\BIN. These steps are done automatically when you run the **EE319K_Install_in.exe** file.

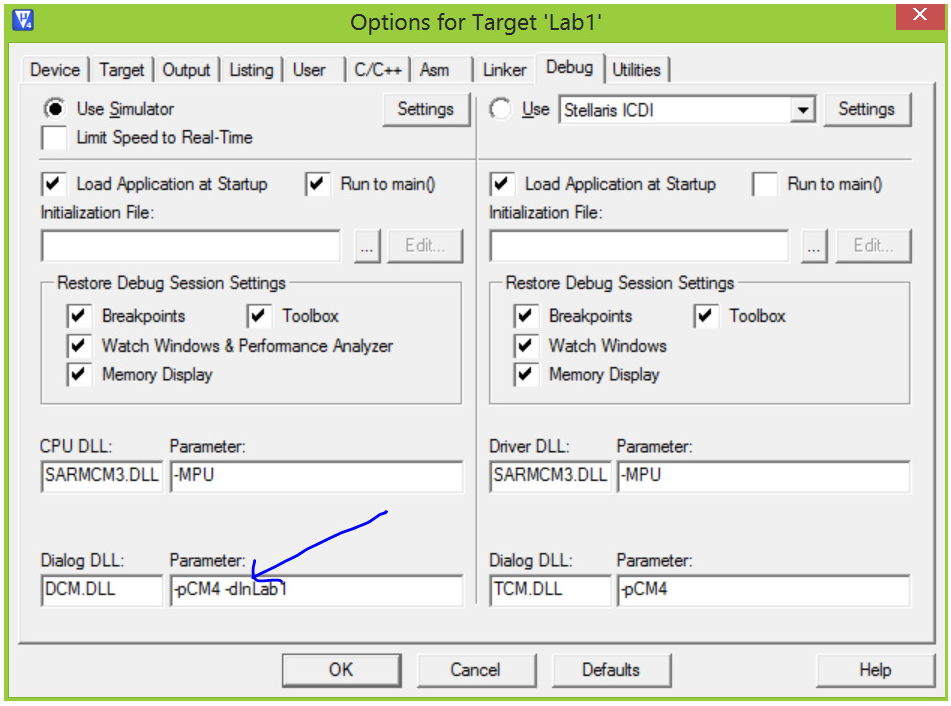


Figure 1.2. Debug the software using the simulator (DCM.DLL -pCM4 -dInLab1).

Part b) You will write C code that inputs from PE3 and PE4, and outputs to PE2. The address definitions for Port E device registers are included in the starter code.

The opening comments include: file name, overall objectives, hardware connections, specific functions, author name, and date. The **`#define`** pre-processor directive is used here to define port device register addresses. We will decipher this declaration later in the course, for now know that it allows us to refer to an address using a symbolic name.

To interact with the I/O during simulation, execute the **Peripherals->TExaS Port E** command. When running the simulator, we check and uncheck bits in the I/O Port box (Figure 1.3) to change input pins (1) and (2). We observe output pins in the window (3). You can also see the other DIR AFSEL DEN and PUR registers. *Ignore the Grading controls section as they were used in an online version of the course and are not applicable to you.*,

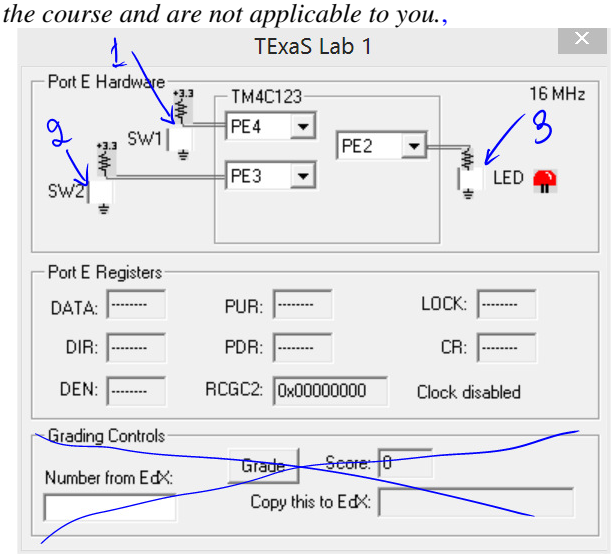


Figure 1.3. In simulation mode, we interact with virtual hardware.

Demonstration

During the demonstration, you will be asked to run your program to verify proper operation. You should be able to single step your program and explain what your program is doing and why. You need to know how to set and clear breakpoints. You should know how to visualize Port E input/output in the simulator.

Deliverables (Items 1-4 are one pdf file uploaded to Canvas, have this file open during demo.)

0) Lab 1 grading sheet. You can print it yourself or pick up a copy in lab. You fill out the information at the top.
1) Assembly source code of your final **Lab1.c** program
2) One screen shot of the Port E window, showing PE2, PE3, and PE4 showing the case with the LED on.

Lab 2. Switch and LED interfacing

Preparation

Read Sections 2.7, 3.3.8, and 4.2

Download

EE319K_TM4C123_artist.sch PCB artist example drawing with all EE319K components

Purpose

The purpose of this lab is to learn how to interface a switch and an LED. You will perform explicit measurements on the circuits in order to verify they are operational and to improve your understanding of how they work. This is the only lab in which you will write your software in assembly.

System Requirements

Figure 4.13 in the book shows a switch interface that implements positive logic. In this lab you will connect it to PE0. The right side of Figure 4.14 shows an LED interface that implements positive logic. You will attach this switch and LED to your protoboard, and interface them to your MicrocontrollerTM4C123. Overall functionality of this system is as follows:

1) Make **PE1** an output and make **PE0** an input.
2) The system starts with the LED on (make **PE1** =1).
3) Wait about 62 ms
4) If the switch is pressed (**PE0** is 1), then toggle the LED once, else turn the LED on.
5) Steps 3 and 4 are repeated over and over

Procedure

First, you will build and test your software first in simulation. Second you will build the hardware on a protoboard connecting it to your Microcontroller. You will test and debug both using the Keil IDE running on your PC. The final demonstration will be run stand-alone without connection to the PC. This stand-alone operation illustrates how embedded systems operate.

To run the Lab 2 simulator, you must do two things. First, execute Project->Options and select the Debug tab. The debug parameter field must include **-dInLab2**. Second, make sure the **InLab2.dll** file is added to your Keil\ARM\BIN folder. These steps are done automatically when you run the **EE319K_Install_in.exe** file. *The screenshot below of the virtual hardware reads Lab3 instead of Lab2, which will be the case for you as well, please ignore it.*

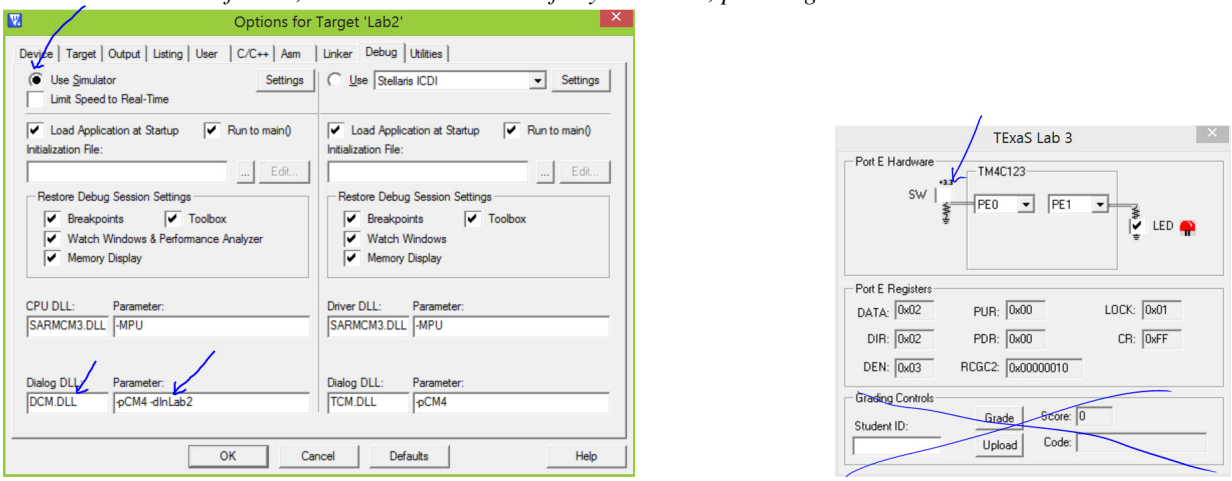


Figure 2.1. Using TExaS to debug your software in simulation mode (DCM.DLL -pCM4 -dInLab2).

Part a – optional but recommended learning experience) Make a copy of the **EE319K_TM4C123_artist.sch** file and rename it to Lab2. Edit this file to show the interface for the switch and the LED. When you are done, print the circuit as a pdf file, and email this pdf file to the TA for verification. Alternatively, you can handdraw and send the circuit picture to the TA for verification. If you wish to build something different from Figure 2.3, please have the TA check your design. Your circuit should look similar to book Figure 4.13 and the right side of Figure 4.14. If using PCB Artist, the first step is to click and drag components you will need so they are close to each other

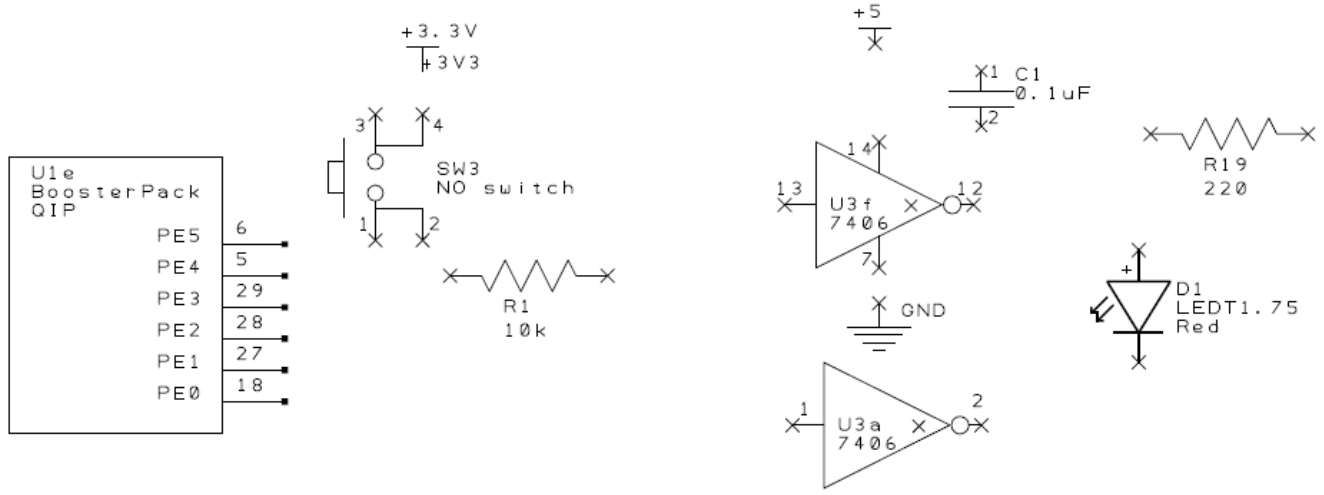


Figure 2.2. PCB Artist drawing showing Port E, 3.3V power, a 10kΩ resistor, a switch, one 7406 gate, an LED, a 220Ω resistor, a 0.1 μF capacitor, +5V power and ground.

The second step is to add traces (these will be actual wires when you build it). In general, we draw circuit diagrams so current flows from top to bottom on the page.

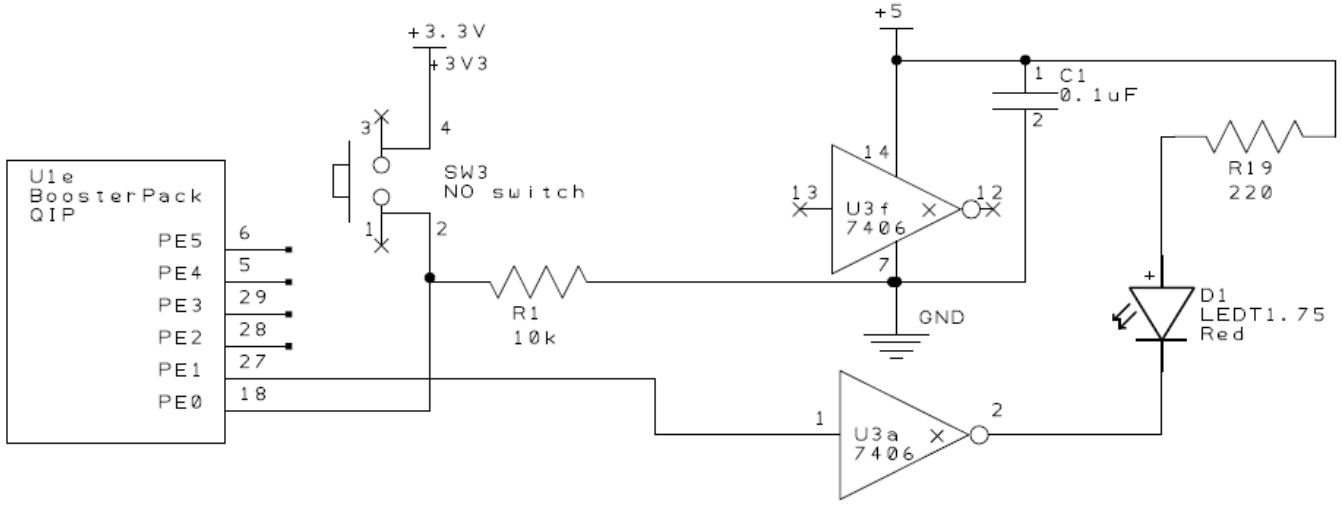


Figure 2.3. PCB Artist drawing one possible Lab2 drawing. When building the circuit, the +3.3V comes from the +3.3V signal on the LaunchPad. The +5V comes from the VBUS signal on the LaunchPad. The LaunchPad ground must be connected to the ground signals on your external circuit.

In the left side of Figure 2.4, wire1 and wire2 cross in the circuit diagram, but are not electrically connected, because there is NO dot at the point of crossing. In the right side of Figure 2.4, wire1 and wire2 cross in the circuit diagram, and are electrically connected, because there IS a dot at the point of crossing.

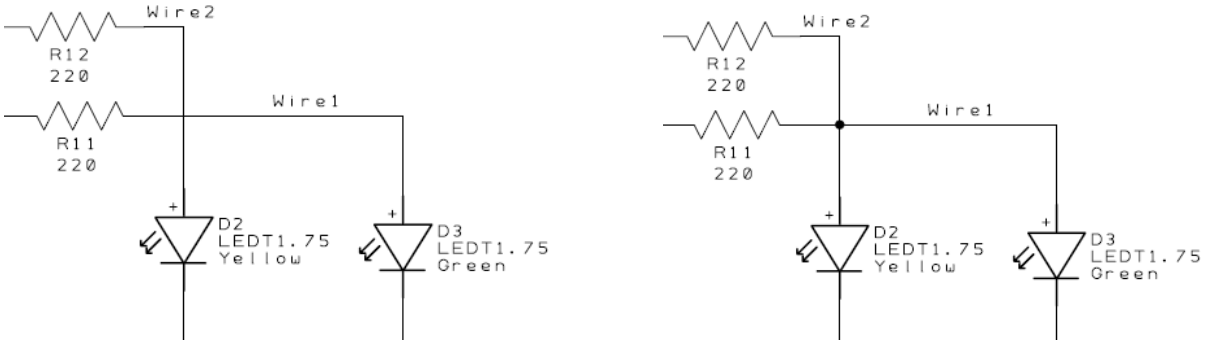


Figure 2.4. When traces cross without a dot they are not connected. When traces cross with a dot they are connected.

Part b) The best approach to time delay will be to use a hardware timer, which we will learn in Lab 3. In this lab, however, we do not expect you to use the hardware timer. Instead write a delay loop:
On the TM4C123 the default bus clock is 16 MHz ±1%. Starting in Lab 3 we will activate the phase lock loop (PLL) and the bus clock will be exactly 80 MHz. For now, however, we will run at about 16 MHz. The following is a portion of a listing file with a simple delay loop.

```
0x00000158 F44F7016        MOV R0,#800
0x0000015C 3801     wait   SUBS R0,R0,#0x01
0x0000015E D1FD            BNE  wait
```

The SUBS and BNE are executed 800 times. The SUBS takes 1 cycle and the BNE takes 1 to 3 (a branch takes 0 to 3 cycles to refill the pipeline). The minimum time to execute this code is 800*(1+1)/16 µs = 100 µs. The maximum time to execute this code is 800*(1+4)/16 µs = 250 µs. Since it is impossible to get an accurate time value using the cycle counting method, we will need another way to estimate execution speed. An accurate method to measure time uses a logic analyzer or oscilloscope. In the simulator, we will use a simulated logic analyzer, and on the real board we will use an oscilloscope. To measure execution time, we cause rising and falling edges on a digital output pin that occur at known places within the software execution. We can use the logic analyzer or oscilloscope to measure the elapsed time between the rising and falling edges. In this lab we will measure the time between edges on output PE1.
*(note: the **BNE** instruction executes in 3 cycles on the simulator, but in 2 cycles on the real board)*
Note that the delay of 62ms will require you to make appropriate adjustments to the code above. Also, the 62ms delay was an arbitrary choice, one with which you will be able to see the LED flash with your eyes.

Part c) Engineers must be able to read datasheets during the design, implementation and debugging phases of their projects. During the design phase, datasheets allow us to evaluate alternatives, selecting devices that balance cost, package size, power, and performance. For example, we could have used other IC chips like the 7405, 74LS05, or 74HC05 to interface the LED to the TM4C123. In particular, we chose the 7406 or 74LS06 because it has a large output current (I_{OL} = 40 mA), 6 drivers, and is very inexpensive (59¢). During the implementation phase, the datasheet helps us identify which pins are which. During the debugging phase, the datasheet specifies input/output parameters that we can test. Download the 7406 and LED datasheets from the web

http://users.ece.utexas.edu/~valvano/Datasheets/7406.pdf
http://users.ece.utexas.edu/~valvano/Datasheets/LED_red.pdf

and find in the datasheet for the 7406 the two pictures as shown in Figure 3.5. Next, hold your actual 7406 chip and identify the location of pin 1. Find in the datasheet the specification that says the output low voltage (V_{OL}) will be 0.4V when the output low current (I_{OL}) is 16 mA (this is close to the operating point we will be using for the LED interface).

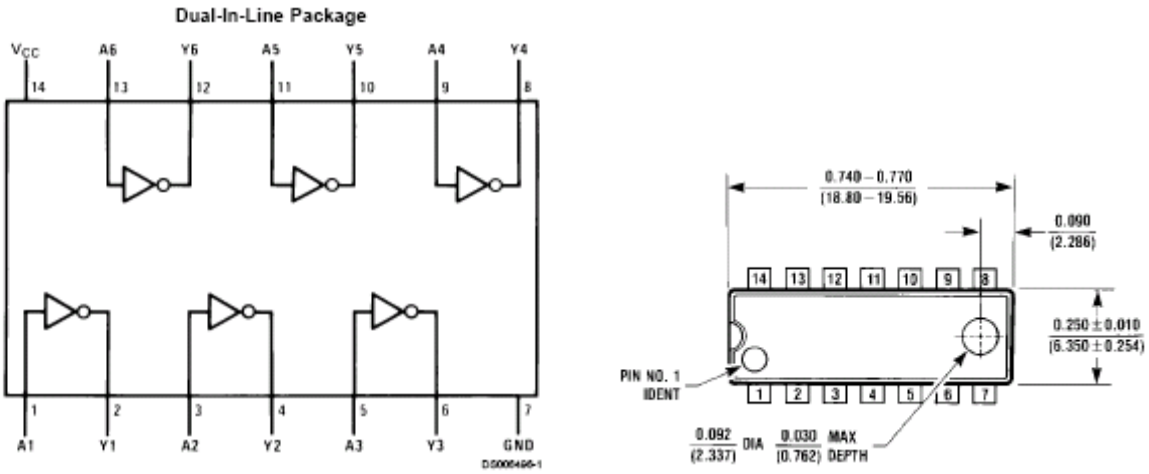


Figure 2.5. Connection diagram and physical package diagram for the 7406.

Using the data sheet, hold an LED and identify which pin is the anode and which is the cathode. Sometimes we are asked to interface a device without a data sheet. Notice the switch has 4 pins in a rectangular shape, as shown in Figure 2.6. Each button is a single-pole single-throw normally-open switch. All four pins are connected to the switch. Using your ohmmeter determine which pairs of pins are internally connected, and across which pair of pins is the switch itself. In particular, draw the internal connections of the switch, started in Figure 3.6, showing how the four pins are connected to the switch.

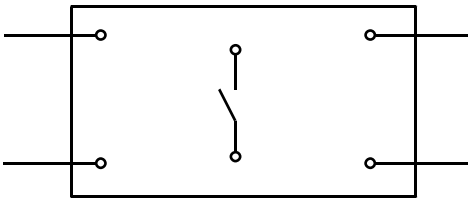


Figure 2.6. Connection diagram for the normally open switch.

To build circuits, we'll use a solderless breadboard, also referred to as a protoboard. The holes in the protoboard are internally connected in a systematic manner, as shown in Figure 2.7. The long rows of of 50 holes along the outer sides of the protoboard are electrically connected. Some protoboards like the one in Figure 2.7 have four long rows (two on each side), while others have just two long rows (one on each side). We refer to the long rows as power buses. If your protoboard has only two long rows (one on each side, we will connect one row to +3.3V and another row to ground. If your protoboard has two long rows on each side, then two rows will be ground, one row will be +3.3V and the last row will be +5V (from VBUS). Use a black marker and label the voltage on each row. In the middle of the protoboard, you'll find two groups of holes placed in a 0.1 inch grid. Each adjacent row of five pins is electrically connected. We usually insert components into these holes. IC chips are placed on the protoboard, such that the two rows of pins straddle the center valley. To make connections to the TM4C123 we can run male-male solid wire from the bottom of the microcontroller board to the protoboard. For example, assume we wish to connect TM4C123 PE1 output to the 7406 input pin 1. First, cut a 24 gauge solid wire long enough to reach from PE1 and pin 1 of the 7406. Next, strip about 0.25 inch off each end. Place one end of the wire in the hole for the PE1 and the other end in one of the four remaining of the 7406 pin 1.

If you are unsure about the wiring, please show it to your TA before plugging in the USB cable. I like to place my voltmeter on the +3.3V power when I first power up a new circuit. If the +3.3V line doesn't immediately jump to +3.3V, I quickly disconnect the USB cable.

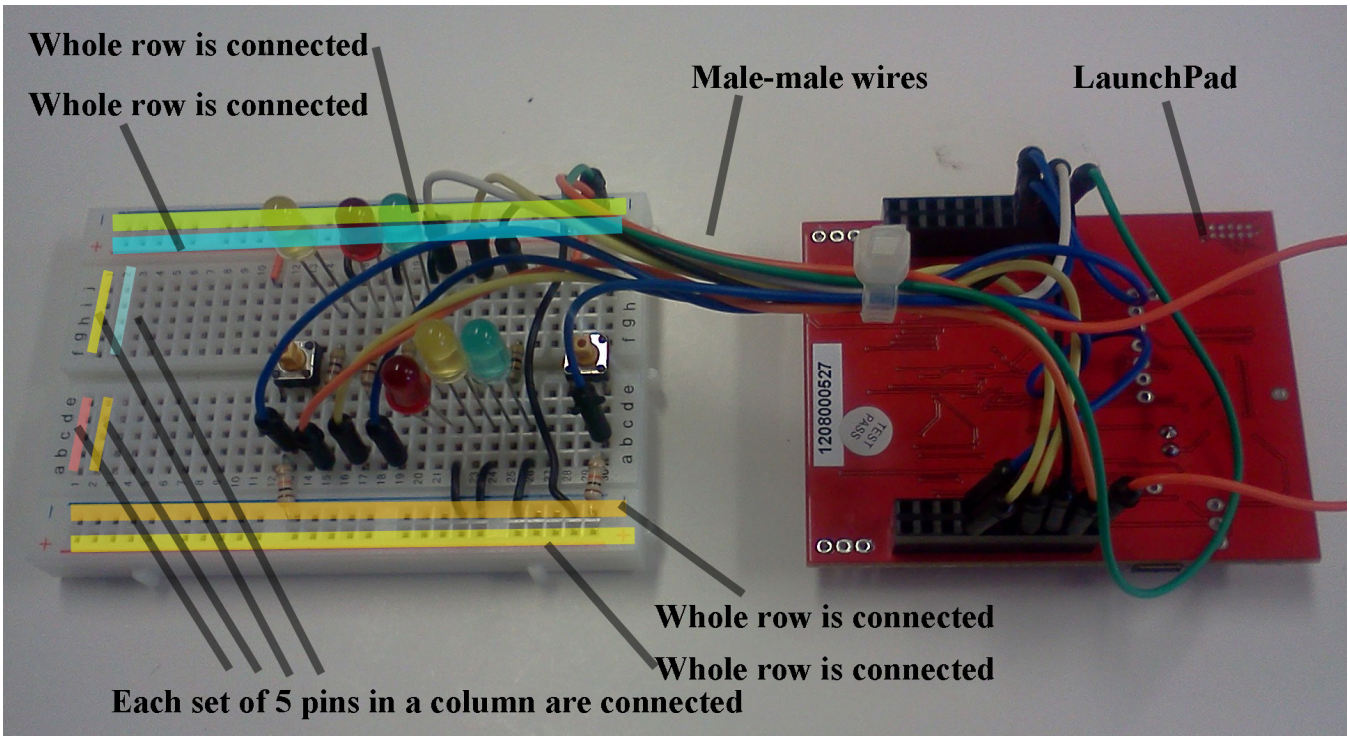


Figure 2.7. All the pins on each of the four long rows are connected. The 5 pins in each short column are connected. Use male-male wires to connect signals on the LaunchPad to devices on the protoboard. Make sure ground wire connected between the LaunchPad and your circuit. The +3.3V and VBUS (+5V) power can be wired from the LaunchPad to your circuit. I like to connect the two dark blue rows to ground, one red row to +3.3V and the other red row to VBUS(+5V).

Part d) After the software has been debugged on the simulator, you will build the hardware on the real board. Please get your design checked by the TA before you apply power (plug in the USB cable). *Do not place or remove wires on the protoboard while the power is on.*

Before connecting the switch to the microcomputer, please take the measurements in Table 2.1 using your digital multimeter. The input voltage (V_{PE0}) is the signal that will eventually be connected to **PE0**. With a positive logic switch interface, the resistor current will be V_{PE0}/10kΩ. The voltages should be near +3.3 V or near 0 V and the currents will be less than 1 mA. The goal is to verify the V_{PE0} voltage is low when the switch is not pressed and high when the switch is pressed.
The circuit in Figures 2.1 and 2.2 used R1 as the 10kΩ resistor. There are other 10k resistors in the PCB artist starter file; any one of which could have been used.

Parameter	Value	Units	Conditions
Resistance of the 10kΩ resistor, R1		ohms	with power off and disconnected from circuit (measured with ohmmeter)
Supply Voltage, $V_{+3.3}$		volts	Powered (measured with voltmeter)
Input Voltage, V_{PE0}		volts	Powered, but with switch not pressed (measured with voltmeter)
Resistor current		mA	Powered, but switch not pressed $I=V_{PE0}/R1$ (calculated and measured with an ammeter)
Input Voltage, V_{PE0}		volts	Powered and with switch pressed

			(measured with voltmeter)
Resistor current		mA	Powered and switch pressed $I=V_{PE0}/R1$ (calculated and measured with an ammeter)

Table 2.1. Switch measurements.

Next, you can connect the input voltage to **PE0** and use the debugger to observe the input pin to verify the proper operation of the switch interface. You will have to single step through the code that initializes Port E, and PE0. You then execute the **Peripherals->TEXaS Port E** command. As you single step you should see the actual input as controlled by the switch you have interfaced, see Figure 2.1.

The next step is to build the LED output circuit. LEDs emit light when an electric current passes through them, as shown in Figure 2.8. LEDs have polarity, meaning current must pass from anode to cathode to activate. The anode is labeled **a** or **+** , and cathode is labeled **k** or **-**. The cathode is the short lead and there may be a slight flat spot on the body of round LEDs. Thus, the anode is the longer lead. LEDs are not usually damaged by heat when soldering. Furthermore, LEDs will not be damaged if you plug it in backwards. LEDs however won't work plugged in backwards of course. Look up the pin assignments in the 7406 data sheet. Be sure to connect +5V power to pin 14 and ground to pin 7. The 0.1 µF capacitor from +5V to ground filters the power line. Every digital chip (e.g., 7406) should have a filter capacitor from its power line (i.e., pin 14 V_{CC}) to ground. The capacitor in your kit is ceramic, which is not polarized, meaning it can be connected in either direction.

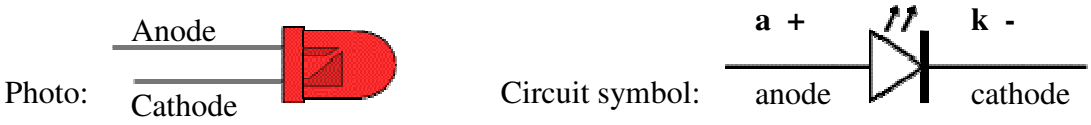


Figure 2.8. LEDs.

The circuit in Figures 2.2 and 2.3 used R10 as the 220Ω resistor. There are six 220Ω resistors R10 – R15 in the PCB artist starter file, any one of which could have been used.

Take the measurements as described in Table 2.2. The R10 measurement occurs before R10 is inserted into the circuit. Single step your software to make **PE1** to output. Initially **PE1** will be low. So take four measurements with **PE1** low, rows 2,3,4,5 in Table 2.2. Then, single step some more until **PE1** is high and measure the three voltages (rows 8,9,10 in Table 2.2). When active, the LED voltage should be about 2 V, and the LED current should be about 10 mA. The remaining rows are calculated values, based on these 8 measurements. The LED current (row 12) can be determined by calculation or by direct measurement using the ammeter function. You should perform both ways to get LED current.

Part e) Debug your combined hardware/software system on the actual TM4C123 board.

Warning: NEVER INSERT/REMOVE WIRES/CHIPS WHEN THE POWER IS ON.

Row	Parameter	Value	Units	Conditions
1	Resistance of the 220Ω resistor, R10		ohms	with power off and disconnected from circuit (measured with ohmmeter)
2	+5 V power supply V_{+5}		volts	(measured with voltmeter, *notice that the +5V power is not exactly +5 volts*)
3	TM4C123 Output, V_{PE1} input to 7406		volts	with **PE1** = 0 (measured with voltmeter)
4	7406 Output, V_{k-} LED k-		volts	with **PE1** = 0

				(measured with voltmeter)
5	LED a+, V_{a+} Bottom side of R10		volts	with **PE1** = 0 (measured with voltmeter)
6	LED voltage		volts	calculated as V_{a+} - V_{k-}
7	LED current		mA	calculated as (V_{+5} - V_{a+})/R10 and measured with an ammeter
8	TM4C123 Output, V_{PE1} input to 7406		volts	with **PE1** = 1 (measured with voltmeter)
9	7406 Output, V_{k-} LED k-		volts	with **PE1** = 1 (measured with voltmeter)
10	LED a+, V_{a+} Bottom side of R10		volts	with **PE1** = 1 (measured with voltmeter)
11	LED voltage		volts	calculated as V_{a+} - V_{k-}
12	LED current		mA	calculated as (V_{+5} - V_{a+})/R10 and measured with an ammeter

Table 2.2. LED measurements (assuming the 220 Ω resistor is labeled R10).

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 switch and LEDs work. Also be prepared to explain how your software works and to discuss other ways the problem could have been solved. Why the 7406 was used to interface the LED? I.e., why did we not connect the LED directly to the TM4C123. Why do you think you need the capacitor for 7406 chip? Why was the delay increased from 1 to 62 ms? How would you modify the software to change the rate at which LED flickers? What operating point (voltage, current) exists when the LED is on? Sketch the approximate current versus voltage curve of the LED. Explain how you use the resistor value to select the operating point. What is the difference between a positive logic and negative logic interface for the switch or the LED? We may test to see if you can measure voltage, current and/or resistance with your meter (so bring your meter to the demonstration).

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

0) Lab 2 grading sheet. (TA will have this)
1) Circuit diagram, using PCBArtist, or hand drawn
2) Screenshot like Figure 2.9 showing your debugging in the simulator
3) Switch measurements (Table 2.1)
4) LED measurements (Table 2.2)
5) Assembly source code of your final program

Precautions to avoid damaging your system

0) Do not attach or remove wires on the protoboard when power is on. Always shut power off when making hardware changes to the system.
1) Touch a grounded object before handling CMOS electronics. Try not to touch any exposed wires.
2) Do not plug or unplug the modules into the LaunchPad while the system is powered.
3) Do not use the TM4C123 with any external power sources, other than the USB cable. In particular, avoid connecting signals to the TM4C123 that are not within the 0 to +5V range. In particular, voltages less than 0V or greater than +5V will damage the microcontroller.
4) Do not use PC3,PC2,PC1,PC0. These are needed for the debugger.

5) You can't use PA1 PA0 PD4 and PD5. These are connected to the serial port and USB.
6) See Figure 4.8 in the textbook. If you use both PD0 and PB6, remove R9 from your board. If you use both PD1 and PB7, remove R10 from your board.

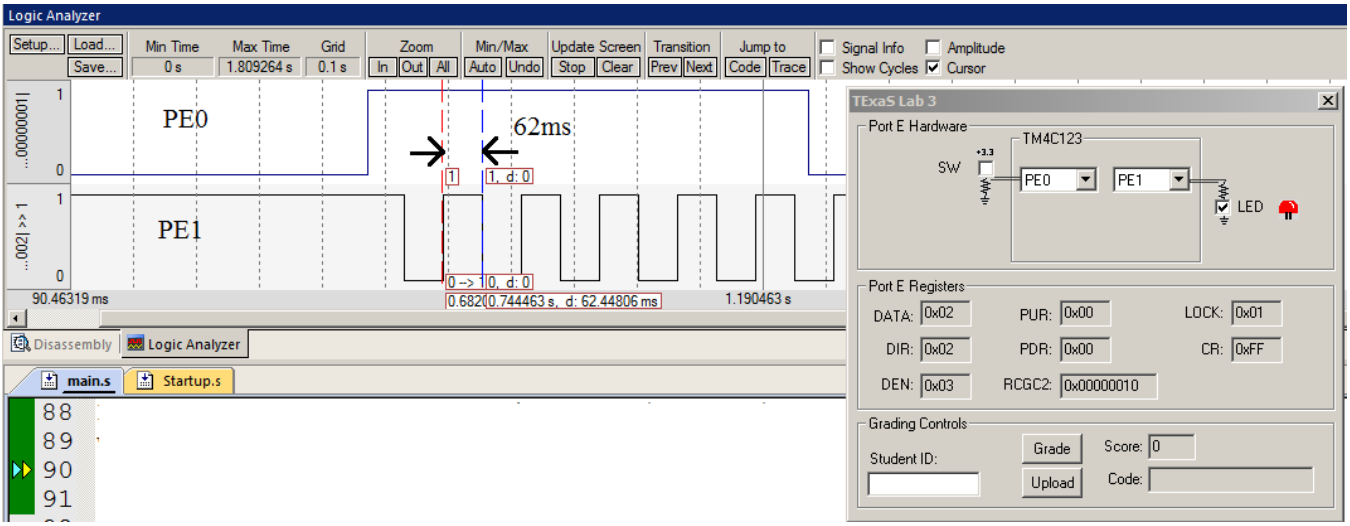


Figure 2.9. Simulation of Lab 3, showing PE0 high, and the PE1 output toggling at 8 Hz.

Lab 3. Minimally Intrusive Debugging Methods

Preparation

Read Sections 4.3, 4.4, 5.1, 5.2, 5.5, 5.7, 6.1, 6.2, and 6.9

Download, unzip, open, compile, and run the project **Lab3.zip** from class site:

http://users.ece.utexas.edu/~ryerraballi/sa2014/EE319K.html

Purpose

The purpose of this lab is to learn minimally intrusive debugging skills. When visualizing software running in real-time on an actual microcomputer, it is important use minimally intrusive debugging tools. The first objective of this lab is to develop an instrument called a dump, which does not depend on the availability of a debugger. A **dump** allows you to capture strategic information that will be viewed at a later time. Many programmers use the printf statement to display useful information as the programming is being executed. On an embedded system we do not have the time or facilites to use printf. Fortunately, we can use a dump in most debugging situations for which a printf is desired. The second useful debugging technique you will learn is called a heartbeat. A **heartbeat** is a visual means to see that your software is still running. The debugging techniques in this lab use both hardware and software and are appropriate for the real TM4C123. Software skills you will learn include indexed addressing, array data structures, the PLL, the SysTick timer, and subroutines.

System Requirements

In this lab, you will design, implement, test and employ software debugging instruments to experimentally verify the correct operation of your Lab 2 system. But, first you will rewrite parts of your code in C. Specifcally:

- You will take your delay code and write it as an assembly module called Delay.s with a single subroutine **`Delay`**.
- You will see how the linkage is done between the C and assembly code and do the linkage for the Delay module.
- You will rewrite a your **`main`** in C following the same steps listed in Lab2. Complete routines **`PortF_Init`**, **`PortE_Init`**.
- You will also write the bebug instrumentation code (**`Debug_Init and Debug_Capture`**)
- All code you are responsible for is tagged as **`–UUU––`**

For the functional debugging aspect of the lab you will record (**`Debug_Capture`**) the Port E value (includes both the input and output signals) and the time during each execution through the loop of your main program of Lab 2 as your system runs in real time. You will activate the **PLL** (call **`PLL_Init`** shown in Program 4.6 of the book) to make the microcontroller run at 80 MHz. You will also activate the **SysTick** timer (call **`SysTick_Init`** shown in Program 4.7 of the book), which will make the 24-bit counter decrement every 12.5 ns. We will use this counter (**`NVIC_ST_CURRENT_R`**) to measure time differences up to 335.5 ms. To measure the current time, you simply read the 24-bit **`NVIC_ST_CURRENT_R`** value. This software dump should store Port E and **`NVIC_ST_CURRENT_R`** data into arrays while the system is running in real-time, and the information will be viewed at a later time. Software dumps are an effective technique when debugging real-time software on an actual microcomputer. We will define the debugging instrument as **minimally intrusive** if the time to collect and store the information is short compared to the time between when information is collected. You will define an array capable of storing about 3 seconds worth of Port E measurements, and an array capable of storing about 3 seconds worth of time measurements. For example, if the loop of your Lab executes in about 62 ms, then the loop will be executed about 50 times in 3 seconds (3000/62), so the array sizes will need to be 50 elements each. You will collect performance data on the system with a no-touch, touch, no-touch sequence during the 3-second measurement.

PF2 is already interfaced to an LED on the TM4C123 board itself. Write debugging software and add it to the Lab 3 system so that this LED always flashes while your program is running. In particular, initialize the direction register so **PF2** is an output, and add code that toggles the LED each time through the loop. A heartbeat of this type will be added to all software written for Labs 4, 5, 6, 7, and 8 in this class. In Lab 4 will use PF2 as part of the system output. It is a quick and convenient way to see if your program is still running.

Procedure

The basic approach to Lab 3 through Lab 9 will be to first develop and test the system using simulation mode. After debugging instruments themselves are tested, you will collect measurements on the real TM4C123.

Part a) Edit your main program so it activates the **PLL** making bus clock 80 MHz (the given code sets it to 50MHz). Activate the SysTick timer by calling **`SysTick_Init`** function (provided as a stub). Write two debugging subroutines that implement a **dump** instrument. If we saved just the input/output data, then the dump would be called *functional debugging,*

because we would capture input/output data of the system, without information specifying when the input/output was collected. However, you will save both the input/output data and the time, so the dump would be classified as *performance debugging*. The first subroutine (**`Debug_Init`**) initializes your dump instrument. The initialization should place 0xFFFFFFFF into the two arrays, and initialize pointers and/or counters as needed. The second subroutine (**`Debug_Capture`**) that saves one data-point (**PE0** input data, and **PE1** output data) in the first array and the **`NVIC_ST_CURRENT_R`** in the second array. Since there are only two bits to save in the first array, pack the information into one value for ease of visualization when displayed in hexadecimal. Put the PE0 value into bit 4 and the PE1 value into bit 0. I.e.,

Input (PE0)	Output (PE1)	save data
0	0	000**0**,000**0**$_2$, or 0x0000.0000
0	1	000**0**,000**1**$_2$, or 0x0000.0001
1	0	000**1**,000**0**$_2$, or 0x0000.0010
1	1	000**1**,000**1**$_2$, or 0x0000.0011

In this way, you will be able to visualize the data in an effective manner. Place a call to **`Debug_Init`** at the beginning of the system, and a call to **`Debug_Capture`** at the start of each execution of the main loop. The basic steps involved in designing the data structures for this debugging instrument are as follows

Allocate **`DataBuffer`** in RAM (to store 3 seconds of input/output data)
Allocate **`TimeBuffer`** in RAM (to store 3 seconds of timer data)
Allocate and **`index`** to remember the position to save the next data

The basic steps involved in designing **`Debug_Init`** are as follows

Set all entries of the first buffer to 0xFFFFFFFF (meaning no data yet saved)
Set all entries of the second buffer to 0xFFFFFFFF (meaning no data yet saved)
Initialize the **`index`** to the beginning of each buffer

The basic steps involved in designing **`Debug_Capture`** are as follows

Save any registers needed
Return immediately if the buffers are full (pointer past the end of the buffer)
Read Port E and the SysTick timer (**`NVIC_ST_CURRENT_R`**)
Mask capturing just bits 1,0 of the Port E data
Shift the Port E data bit 0 into bit 4 position, and bit 1 into bit 0 position
Dump this port information into **`DataBuffer`** using the **`index`** position
Dump time into **`TimeBuffer`** using the **`index`** position
Increment **`index`** to the next location

You can observe the debugging arrays using a **`Watch`** and/or **`Memory`** window. After you have debugged your code in simulation mode, capture a screenshot showing the results as the switch is pressed then not pressed. The dumped data should start with some 0x01 data, next it should oscillate between 0x10, 0x11 as the switch is pressed, then return to 0x01 when the switch is released.

Part b) One simple way to estimate the execution speed is to assume each instruction requires about 2 cycles. By counting instructions (look at the disassembly) and multiplying by two, you can estimate the number of cycles required to execute your **`Debug_Capture`** subroutine. Assuming the 12.5 ns bus cycle time, convert the number of cycles to time. Next, estimate the time between calls to **`Debug_Capture`**. Calculate the percentage overhead required to run the debugging instrument (100 times execution time divided by time between calls, in percent). This percentage will be a quantitative measure of the intrusiveness of your debugging instrument. Type the estimations and calculations in as comments to your program.

Part c) Write the debugging instruments that implement the heartbeat (initialization and toggle). The heartbeat instruments can be embedded as simple C code without making them separate subroutines.

Part d) Debug your combined hardware/software system first with the simulator, then on the actual TM4C123 board.

Figure 4.2 is screenshot of Lab 4 in simulation mode. You can find the address of the buffers by looking in the map file. You can dump the memory to a file using the command (the data is formatted in a little-endian hexadecimal format)

SAVE data.txt 0x20000000 , 0x20000190

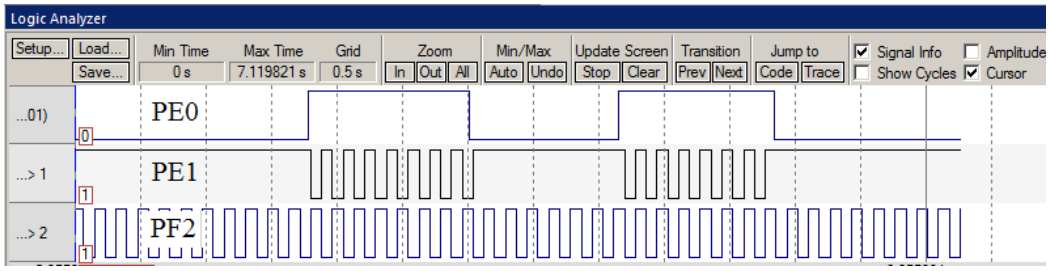


Figure 3.2. Simulation output showing the input, output and heartbeat.

Let's look at one measurement 0x00ABE5DA-0x0057CDB4 = 11265498-5754292=5511206. Since each cycle is 12.5ns, 5511206*12.5ns = 69ms. For this solution, the measurement agrees with the data collected by an oscilloscope and shown in Figure 4.4.

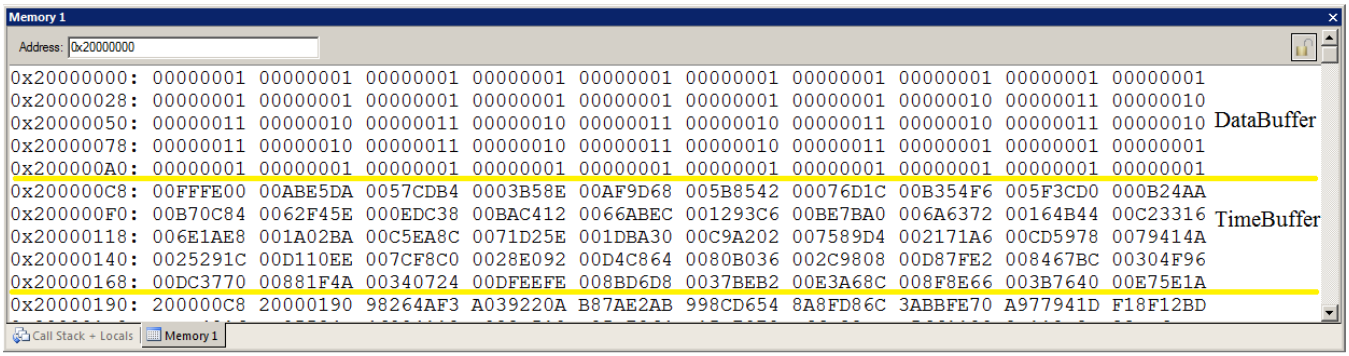


Figure 3.3. Similar data is observed in the memory window in simulation and on the real board showing results of the dump.

Part e) Run your debugging instrument capturing the sequence of input/outputs as you touch, then release the switch. Use the collected data to measure the period of the flashing LED. Your measurement should be accurate to 12.5 ns.

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. You should be able to connect PF2 to an oscilloscope to verify the main loop is running at about every 62 ms. The TA may look at your data and expect you to understand how the data was collected and what the data means. Also be prepared to explain how your software works and to discuss other ways the problem could have been solved. Why did you have to change the delay function after the PLL was activated? How did you change it? Why? The TA will pick an instruction in your program and ask how much time does it take that instruction to execute in µsec. Does it always take same amount of time to execute? You will be asked to create a breakpoint, and add the port pin to the simulated logic analyzer. Is **`Debug_Capture`** minimally intrusive or non-intrusive? What do you mean by intrusiveness? Is your code "friendly"? How do you define masking? How do you set/clear one bit in without affecting other bits? How do you initialize the SysTick? You should understand every step of the function **`SysTick_Init`**. How do you change the rate at which SysTick counts? Describe three ways to measure the time for a software function to execute? How do you calculate the sizes of the port data and the timestamp data? If you used 32-bit data for **`DataBuffer`** what would be the advantages of 8-bit data? If you used 8-bit data for **`DataBuffer`** what would be the advantages of 32-bit? Could you have stored the time-stamp data in 8-bit, 16-bit, or 24-bit arrays? Why does the pointer to the time-stamp array need to be incremented by four, if you want to point to the next element in the array? How do you allocate global variables?

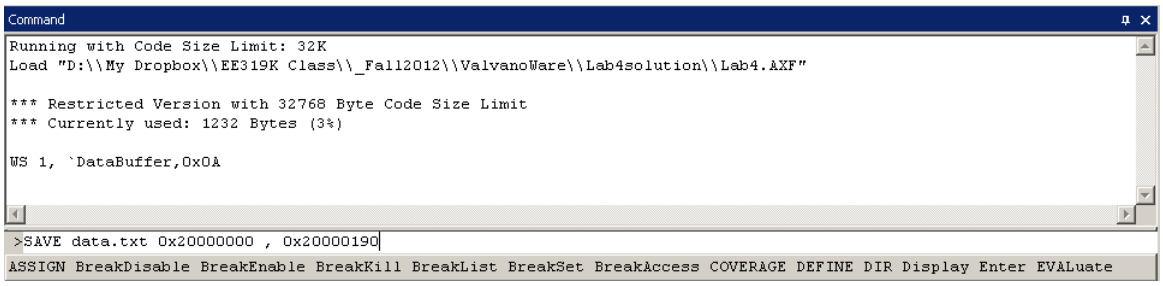
Deliverables (Items 1, 2, 3, 4 are one pdf file uploaded to Canvas, have this file open during demo.)

1) A screenshot showing the system running in simulation mode. In the screenshot, please show the dumped data in a memory window and the I/O window, as illustrated in Figures 4.1 and 4.2

2) Assembly+C listing of your final program with both the dump and heartbeat instruments

3) Estimation of the execution time of your debugging instrument **`Debug_Capture`** (part b)
4) Results of the debugging instrument (part e) and the calculation of the flashing LED period in msec.

How to transfer data from debugger to a computer file
1) Run your system so data is collected in memory, assume interesting stuff is from 0x20000000 to 0x20000190
2) Type **SAVE data.txt 0x20000000 , 0x20000190** in the command window after the prompt (“>”), type enter



3) Open the **data.txt** file in NotePad, it looks like this

```
:020000042000DA
:10000000010000000100000001000000010000000EC
:10001000010000000100000001000000010000000DC
:10002000010000000100000001000000010000000CC
:10003000010000000100000001000000010000000BC
:10004000010000000100000011000000010000000 7E
:10005000110000001000000011000000010000005E
:10006000110000001000000011000000010000004E
:10007000110000001000000011000000010000003E
:10008000110000001000000011000000010000002E
:10009000110000000100000001000000010000004C
:1000A000010000000100000001000000010000004C
:1000B000010000000100000001000000010000003C
:1000C000010000000100000000FEFF00DAE5AB00C7
:1000D000B4CD57008EB50300689DAF0042855B002C
:1000E0001C6D0700F654B300D03C5F00AA240B003F
:1000F000840CB7005EF4620038DC0E0012C4BA0053
:10010000ECAB6600C6931200A07BBE0072636A006F
:10011000444B16001633C200E81A6E00BA021A00E9
:100120008CEAC5005ED2710030BA1D0002A2C9007F
:10013000D4897500A67121007859CD004A41790013
:100140001C292500EE10D100C0F87C0092E02800A8
:1001500064C8D40036B0800008982C00E27FD80034
:10016000BC678400964F30007037DC004A1F88005F
:1001700024073400FEEEDF00D8D68B00B2BE370075
:100180008CA6E300668E8F0040763B001A5EE70087
:01019000C8A6
:00000001FF
```

4) Strip off the first 9 characters of every line, and the last two characters of every line. Delete the first and last lines, leaving the data shown above in bold. Each two characters is a byte in hex. 32-bit and 16-bit data are of course little endian.

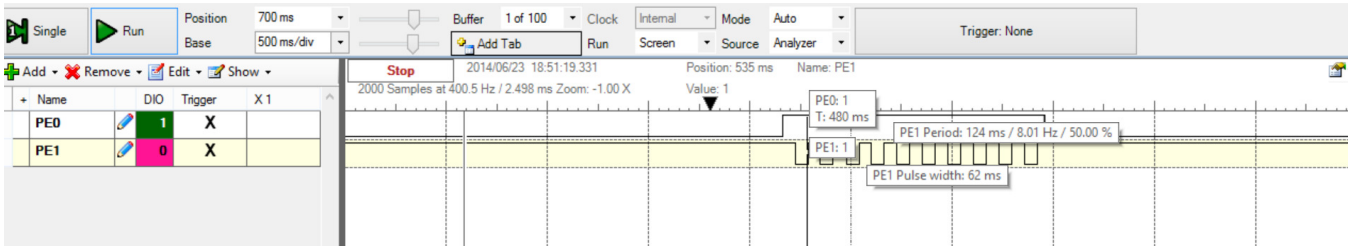


Figure 3.4. The input and output signals are measured with a two channel oscilloscope. When the switch is pressed, PE0 goes high. When the software sees PE0 high it toggles PE1 every 62ms.

Lab 4. Traffic Light Controller

Preparation

Read Sections 4.4, 6.1, 6.4, 6.5 in the textbook and Chapter 10 in the E-Book
See http://users.ece.utexas.edu/~valvano/Datasheets/LED_yellow.pdf
See http://users.ece.utexas.edu/~valvano/Datasheets/LED_green.pdf
Download, unzip, build and simulate the starter code: **Lab4.zip**.

Purpose

This lab has these major objectives: 1) the understanding and implementing linked data structures; 2) learning how to create a segmented software system; and 3) the study of real-time synchronization by designing a finite state machine controller. Software skills you will learn include advanced indexed addressing, linked data structures, creating fixed-time delays using the SysTick timer, and debugging real-time systems.

System Requirements

Consider a 4-corner intersection as shown in Figure 4.1. There are two one-way streets are labeled South (cars travel South) and West (cars travel West). There are three inputs to your TM4C123, two are car sensors, and one is a walk button. The *South* sensor will be true (1) if one or more cars are near the intersection on the South road. Similarly, the *West* sensor will be true (1) if one or more cars are near the intersection on the West road. The *Walk* button will be pushed by a pedestrian when he or she wishes to cross in any direction. To request a walk, a pedestrian must push and hold the walk button for at least 2 seconds, after which person can release the button, and the walk request will be remembered. In contrast, when a car sensor is released, it means no cars are waiting to enter the intersection. You will use 8 outputs from your microcomputer that control the traffic lights, the "walk" light, and the "don't walk" light. The "walk" light will be the green LED (PF3) on the LaunchPad and the "don't walk" light is the red LED (PF1) on the LaunchPad. When the "walk" condition is signified, pedestrians are allowed to cross. When the "don't walk" condition flashes (and the two traffic signals are red), pedestrians should hurry up and finish crossing in any direction. When the "don't walk" condition is on steady, pedestrians should not enter the intersection. The red and green LED connected to PF1 and PF3 respectively must be used. All other inputs and outputs must be built on the protoboard.

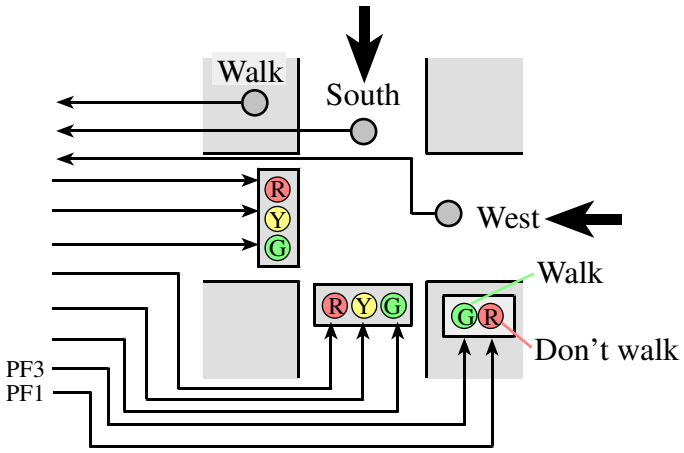


Figure 4.1. Traffic Light Intersection (three inputs and eight outputs).

Traffic should not be allowed to crash. I.e., there should not be only a green or only a yellow LED on one road at the same time there is only a green or only a yellow LED on the other road. You should exercise common sense when assigning the length of time that the traffic light will spend in each state; so that the system changes at a speed convenient for the TA (stuff changes fast enough so the TA doesn't get bored, but not so fast that the TA can't see what is happening). Cars should not be allowed to hit the pedestrians. The walk sequence should be realistic, showing three separate conditions: 1) "walk", 2) "hurry up" using a flashing (at least twice) LED, and 3) "don't walk". You may assume the two car sensors remain active for as long as service is required. However a request for walk is signified by a push and release, where the walk button is pushed for at least 2 seconds.

There is no single, "best" way to implement your system. However, your scheme must use a linked data structure stored in ROM. There should be a 1-1 mapping from the FSM states and the linked elements. A "good" solution will have about 15 to 30 states in the finite state machine, and provides for input dependence. Try not to focus on the civil engineering issues. I.e., the machine does not have to maximize traffic flow or minimize waiting. On the other hand if there are multiple

requests, the system should cycle through, servicing them all. Build a quality computer engineering solution that is easy to understand and easy to change. We expect your solution to behave reasonably and have 15 to 30 states. It is unacceptable if your machine is less than 15 states. If your machine has more than 30 states you have made it more complicated than we had in mind. Because we have three inputs, there will be 8 next state arrows. One way to draw the FSM graph to make it easier to read is to use X to signify don't care. For example, compare the Figure 6.9 in the book to the FSM graph in Figure 4.2 below. Drawing two arrows labeled **01** and **11** is the same as drawing one arrow with the label **X1**. When we implement the data structure, we will expand the shorthand and explicitly list all possible next states.

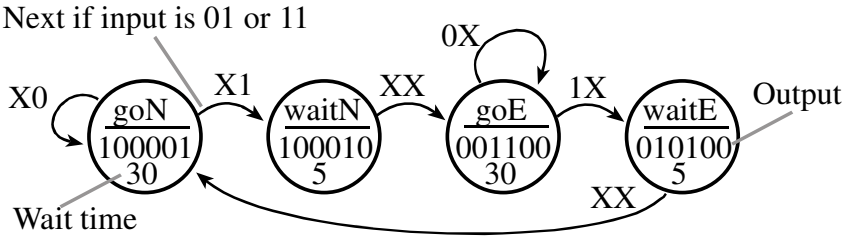


Figure 4.2. FSM from the book Figure 6.20 redrawn with a short hand format.

This lab was written in a manner intended to give you a great deal of flexibility in how you draw the FSM graph, while at the same time require very specific boundaries on how the assembly controller must be written. This flexibility causes students to be question "when am I done?" or "is this enough for an A?" To clarify the distinction between computer engineering, and civil engineering, I restate the computer engineering requirements. In particular do these 9 requirements well and you can get a good grade on this lab.

0) The system provides for input dependence. This means each state has 8 arrows such that the next state depends on the current state and the input. This means you can not solve the problem by simply cycling through all the states regardless of the input. *You should not implement a Mealy machine.*

1) There must be a 1-1 mapping between state graph and data structure. For a Moore machine, this means each state in the graph has a name, an output, a time to wait, and 8 next state arrows (one for each input). The data structure has exactly these components: a name, an output, a time to wait, and 8 next state indices (one for each input). There is no more or no less information in the data structure then the information in the state graph.

2) There can be no conditional statements in program, other than those in **`SysTick_Wait`** and **`SysTick_Wait10ms`**. See the Example 6.5 in the book.

3) The state graph defines exactly what the system does in a clear and unambiguous fashion.

4) Each state has the same format of each state. This means every state has exact one name, one 7-bit output, one time to wait, and 8 next indices.

5) Please use good names and labels (easy to understand and easy to change). Examples of bad state names are **S0** and **S1**.

6) Do not allow cars to crash into each other. Do not allow pedestrians to walk in one direction while any cars are allowed to go. Engineers do not want people to get hurt.

7) There should be 15 to 30 states with either a Moore finite state machine. What if I have less than 10 states? Usually students with less than 10 states did not flash the don't walk light, or they flashed the lights using a counter. Counters and variables violate the "no conditional branch" requirement. If you are less than 15 states, you probably did not handle the 2 second walk button recognition.

8) If the pedestrian pushes the walk button for 2 or more seconds, eventually a walk light must occur. If the pedestrian pushed the walk button for less than 2 seconds, it is ok for the system to generate a walk signal or to not generate a walk signal.

One way to handle the 2-second walk button requirement is to add a duplicate set of states. The first set of states means the walk button is not pressed. The second set of states means the walk operation is requested. Go from the first set to the second set whenever the walk is pushed. Go from the second back to the first whenever the walk condition is output. The two sets of states allow you to remember that a walk has been requested; in this way the request is serviced when appropriate.

There are many civil engineering questions that students ask. How you choose to answer these questions will determine how good a civil engineer you are, but will not affect your grade on this lab. For each question, there are many possible answers, and you are free to choose how you want to answer it. It is reasonable however for the TA to ask how you would have implemented other answers to these civil engineering questions using the same FSM structure.

0) How long should I wait in each state? *Possible answer*: 1 to 2 seconds of real TA time.
1) What happens if I push 2 or 3 buttons at a time? *Possible answer*: cycle through the requests servicing them in a round robin fashion.
2) What if I push the walk button, but release it before 2 seconds are up? *Possible answer*: ignore the request as if it never happened. *Possible answer*: service it or ignore it depending on exactly when it occurred.
3) What if I push a car button, but release it before it is serviced? *Possible answer*: ignore the request as if it never happened (e.g., car came to a red light, came to a full stop, and then made a legal turn). *Possible answer*: service the request or ignore it depending on when it occurred.
4) Assume there are no cars and the light is green on the North, what if a car now comes on the East? Do I have to recognize a new input right away or wait until the end of the wait time? *Possible answer:* no, just wait until the end of the current wait, then service it. *Possible answer*: yes; break states with long waits into multiple states with same output but shorter waits.
5) What if the walk button is pushed while the don't walk light is flashing? *Possible answer*: ignore it, go to a green light state and if the walk button is still pushed, then go to walk state again. *Possible answer:* if no cars are waiting, go back to the walk state. *Possible answer*: remember that the button was pushed, and go to a walk state after the next green light state.
6) Does the walk occur on just one street or both? *Possible answer*: stop all cars and let people walk across either or both streets.
7) How do I signify a walk condition? *Answer*: You must use the on board green LED for walk, and on board red LED as the don't walk.

In real products that we market to consumers, we put the executable instructions and the finite state machine linked data structure into the nonvolatile memory such as flash EEPROM. A good implementation will allow minor changes to the finite machine (adding states, modifying times, removing states, moving transition arrows, changing the initial state) simply by changing the linked data structure, without changing the executable instructions. Making changes to executable code requires you to debug/verify the system again. If there is a 1-1 mapping from FSM to linked data structure, then if we just change the state graph and follow the 1-1 mapping, we can be confident our new system operate the new FSM properly. Obviously, if we add another input sensor or output light, it may be necessary to update the executable part of the software, re-assemble and retest the system.

Procedure

You will create a segmented software system putting global variables into RAM, local variables into registers, constants and fixed data structures into EEPROM, and program object code into EEPROM. Most microcontrollers have a rich set of timer functions. For this lab, you will the SysTick timer to wait a prescribed amount of time. You may use the functions in Section 4 of the book, but you are responsible for understanding all the details. The code to activate the PLL is given to you, so you are not responsible for the details of how the PLL works, but are responsible for what the PLL does and why we use the PLL.

In general, using cycle-counting (for-loops) for delays has the problem of conditional branches and data-dependent execution times. If an interrupt were to occur during a cycle-counting delay, then the delay would be inaccurate using the cycle-counting method. In the above method, however, the timing will be very accurate, even if an interrupt were to occur while the microcomputer was waiting. In more sophisticated systems, using timer interrupts provide a more flexible and even more accurate mechanism for microcomputer synchronization. Interrupt synchronization will be used in Labs 5 through 9.

The basic approach to this lab will be to first develop and debug your system using the simulator. After the software is debugged, you will interface actual lights and switches to the TM4C123, and run your software on the real TM4C123. As you have experienced, the simulator requires different amount of actual time as compared to simulated time. On the other

hand, the correct simulation time is maintained in the SysTick timer, which is decremented every cycle of simulation time. The simulator speed depends on the amount of information it needs to update into the windows. Because we do not want to wait the minutes required for an actual intersection, the cars in this traffic intersection travel much faster than "real" cars. In other words, you are encouraged to adjust the time delays so that the operation of your machine is convenient for you to debug and for the TA to observe during demonstration.
Port pins you will use for the inputs and outputs are given below.

Red south	PB5
Yellow south	PB4
Green south	PB3
Red west	PB2
Yellow west	PB1
Green west	PB0

Table 4.1. Ports to interface the traffic lights (PF1=red don't walk, PF3=green walk).

Walk sensor	PE2
South sensor	PE1
West sensor	PE0

Table 4.2. Ports to interface the sensors.

Part a) Design a finite state machine that implements a good traffic light system. Include a graphical picture of your finite state machine showing the various states, inputs, outputs, wait times and transitions.

Part b) Write and debug the assembly code that implements the traffic light control system. During the debugging phase with the simulator. In simulation mode, capture logic analyzer screen shots showing the operation when cars are present on both roads, like Figure 4.3.

Part c) After you have debugged your system in simulation mode, you will implement it on the real board. Use the same ports you used during simulation. Use the PCB Artist starter file you used in Lab3 to draw your hardware circuit diagram using the PCB Artist program. The first step is to interface three push button switches for the sensors. You should implement positive logic switches. *Do not place or remove wires on the protoboard while the power is on.* Build the switch circuits and test the voltages using a digital voltmeter. You can also use the debugger to observe the input pin to verify the proper operation of the interface.

The next step is to build six LED output circuits. Build the system physically in a shape that matches a traffic intersection, so the TA can better visualize the operation of the system. Look up the pin assignments in the 7406 data sheet. Be sure to connect +5V power to pin 14 and ground to pin 7. Write a simple main program to test the LED interface. Set the outputs high and low, and measure the three voltages (input to 7406, output from 7406 which is the LED cathode voltage, and the LED anode voltage).

Part d) Debug your combined hardware/software system on the actual TM4C123 board.

Demonstration

During checkout, you will be asked to show both the simulated and actual TM4C123 systems to the TA. The TA will expect you to know how the **`SysTick_Wait`** function works, and know how to add more input signals and/or output signals. An interesting question that may be asked during checkout is how you could experimentally prove your system works. In other words, what data should be collected and how would you collect it? If there were an accident, could you theoretically prove to the judge and jury that your software implements the FSM? What type of FSM do you have? What other types are there? How many states does it have? In general, how many next-state arrows are there? Explain how the linked (using indexes) data structure is used to implement the FSM. List some general qualities that would characterize a good FSM.

Deliverables (Items 1, 2, 3, 4 are one pdf file uploaded to Canvas (UT student does this), have this file open during demo.)

1) Logic analyzer screen shot showing the system running in simulation mode when cars are present on both roads.
2) Circuit diagram (with your name and date) using PCB Artist
3) Drawing of the finite state machine

Your Logic Analyzer window will look like the following screnshots

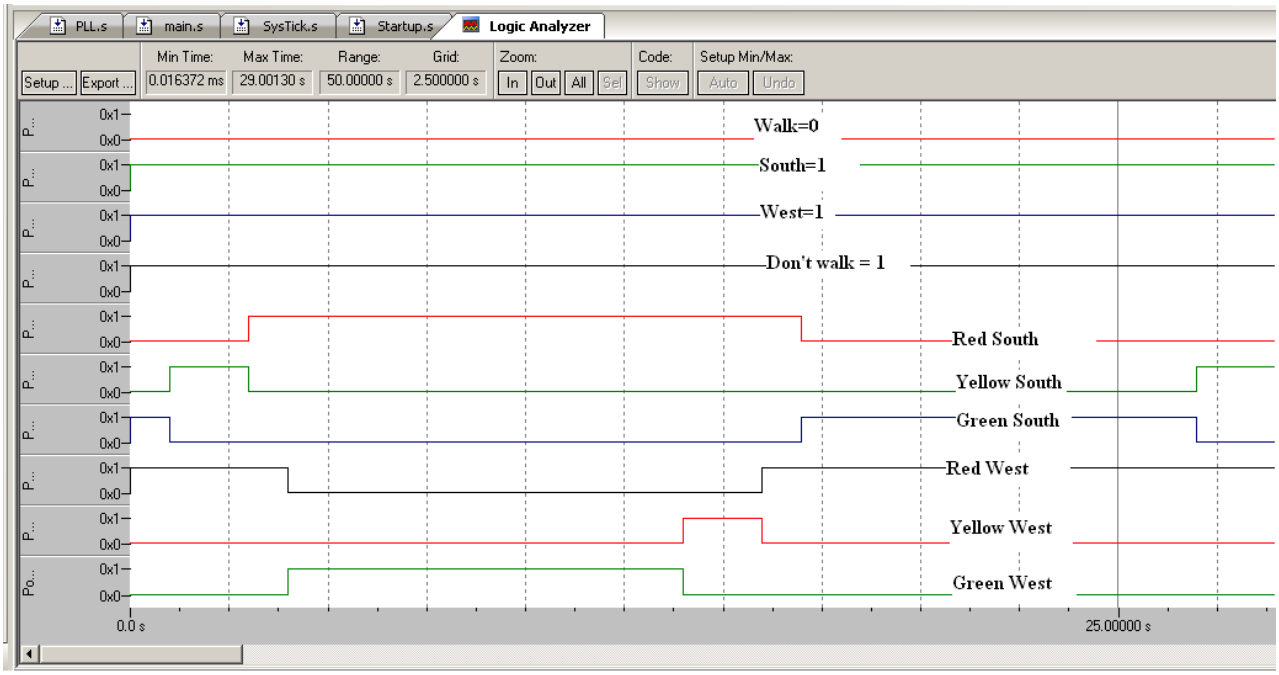


Figure 4.3. Simulation showing cars on both South and West

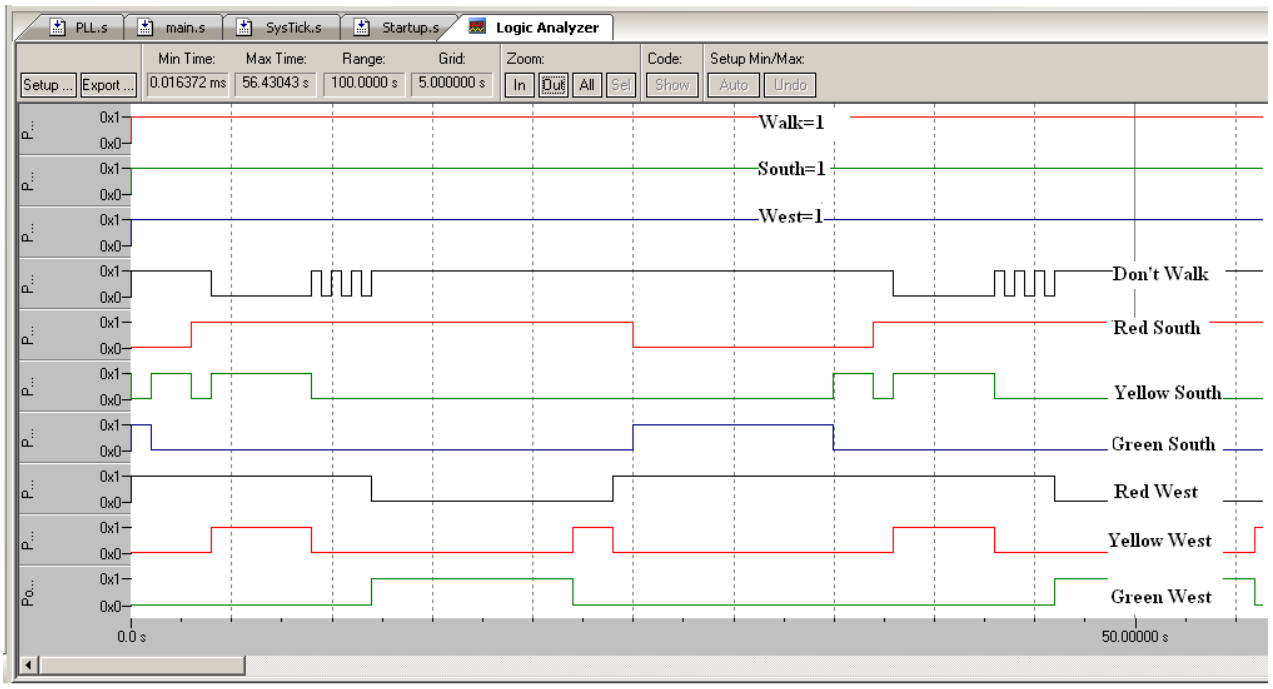


Figure 4.4. Simulation showing walk pushed, and cars on both South and West

Lab 5. Digital Piano using a Digital to Analog Converter

Preparation

Read Sections 9.1, 9.2, 9.4, 9.6, 10.1, 10.2 and 10.3,
Read chapters 12 and 13 from e-Book
Download, unzip, build and simulate the starter code: **Lab5.zip** (see **dac.xls** inside this project)
Install the **LaunchPadDLL.dll** file into Keil/ARM/Bin directory

Purpose

There are four objectives for this lab: 1) to learn about DAC conversion; 2) to understand how digital data stored in a computer could be used to represent sounds and music; 3) to study how the DAC can be used to create sounds; 4) Learn to use Interrupts for responsiveness and precise timing.

Programming Requirements

All software for this lab must be written in C. You can debug your code in the simulator but your final code must run on the board with a DAC circuit.

System Requirements

Most digital music devices rely on high-speed DAC converters to create the analog waveforms required to produce high-quality sound. In this lab you will create a very simple sound generation system that illustrates this application of the DAC. Your goal is to create an embedded system that plays three notes, which will be a digital piano with three keys. The first step is to design and test a 4-bit **binary-weighted** DAC, which converts 4 bits of digital output from the TM4C123 to an analog signal. You are free to design your DAC with a precision of more than 4 bits. You will convert the four digital bits to an analog output using a simple resistor network. During the static testing phase, you will connect the DAC analog output to your voltmeter and measure resolution, range, precision and accuracy. During the dynamic testing phase you will connect the DAC output to headphones, and listen to sounds created by your software. It doesn't matter what range the DAC is, as long as there is an approximately linear relationship between the digital data and the speaker current. The performance score of this lab is not based on loudness, but sound quality. The quality of the music will depend on both hardware and software factors. The precision of the DAC, external noise and the dynamic range of the speaker are some of the hardware factors. Software factors include the DAC output rate and the number of data points stored in the sound data. You can create a 3k resistor from two 1.5k resistors. You can create a 6k resistor from two 12k resistors.

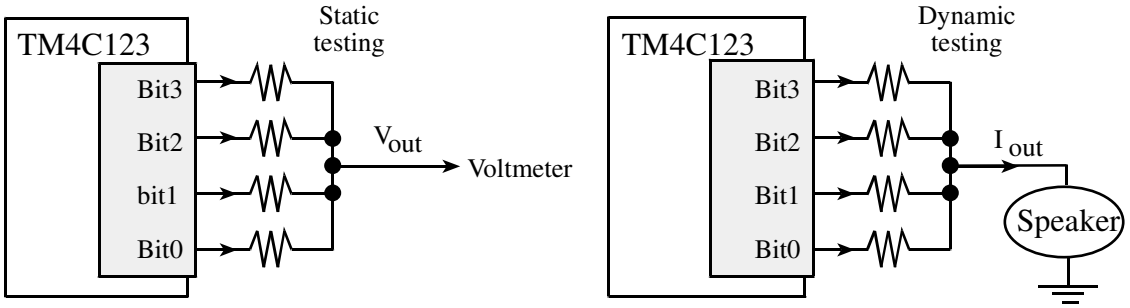


Figure 5.1. DAC allows the software to create music.

The second step is to design a low-level device driver for the DAC. Remember, the goal of a device driver is to separate what the device does from how it works. "What it does" means the general descriptions and function prototypes of **`DAC_Init`** and **`DAC_Out`** that are placed in DAC.h. "How it works" means the implementations of **`DAC_Init`** and **`DAC_Out`** that will be placed in DAC.c. The third step is to design a low-level device driver for the three keys of the piano. For example, if you could create public functions **`Piano_Init`** and **`Piano_In`**, where **`Piano_In`** returns a logical key code for the pattern of keys that are pressed. You may design this driver however you wish, but the goal is to abstract the details how it works (which port, which pin) from what it does (which key pressed). Using Edge-triggered interrupts for reading the keys pressed will make your code responsive but in this application you may also be okay using a busy-wait approach. The fourth step is to organize the sound generation software into a device driver. You will need a data structure to store the sound waveform. You are free to design your own format, as long as it uses a formal data structure. Compressed data occupies less storage, but requires runtime calculation. Although you will be playing only three notes, the design should allow additional notes to be added with minimal effort. For example, if you could create public functions **`Sound_Init`** and

`Sound_Play(note)`, where the parameter note specifies the frequency (pitch) of the sound. For example, calling **`Sound_Play(Off)`** makes it silent and calling **`Sound_Play(C)`** plays the note C. A background thread within the sound driver implemented with SysTick interrupts will fetch data out of your music structure and send them to the DAC. The last step is to write a main program that links the modules together creating the digital piano.

Procedure

Part a) Decide which port pins you will use for the inputs and outputs. Avoid the pins with hardware already connected. Table 4.4 in the book lists the pins to avoid. To use the simulator/grader the choices are listed in Tables 5.1 and 5.2. In particular, Table 5.1 shows you three possibilities for how you can connect the DAC output. Table 5.2 shows you three possibilities for how you can connect the three positive logic switches that constitute the piano keys. Obviously, you will not connect both inputs and outputs to the same pin.

DAC bit 5	PA7	PB5	PE5
DAC bit 4	PA6	PB4	PE4
DAC bit 3	PA5	PB3	PE3
DAC bit 2	PA4	PB2	PE2
DAC bit 1	PA3	PB1	PE1
DAC bit 0	PA2	PB0	PE0

Table 5.1. Possible ports to interface the DAC outputs (DAC bits 4 and 5 are optional).

Piano key 3	PA5	PD3	PE3
Piano key 2	PA4	PD2	PE2
Piano key 1	PA3	PD1	PE1
Piano key 0	PA2	PD0	PE0

Table 5.2. Possible ports to interface the piano key inputs (piano key 3 is optional).

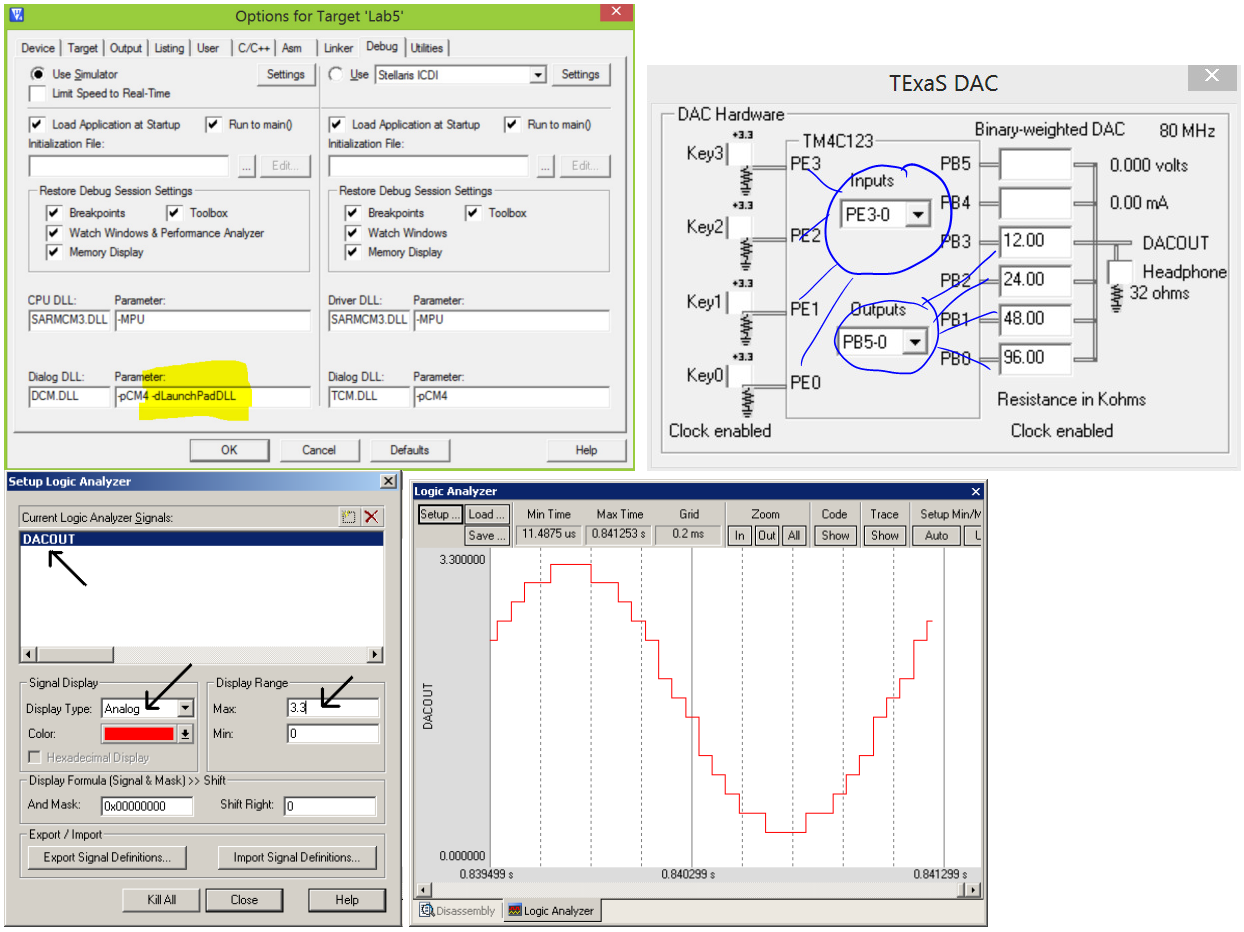


Figure 5.1b. The screens needed to run Lab 5 in simulation.

Draw the circuit required to interface the binary-weighted DAC to the TM4C123. Design the DAC converter using a simple resistor-adding technique. A 4-bit binary-weighted DAC uses resistors in a 1/2/4/8 resistance ratio. Select values in the 1.5 kΩ to 240 kΩ range. For example, you could use 1.5 kΩ, 3 kΩ, 6 kΩ, and 12 kΩ. Notice that you could create double/half resistance values by placing identical resistors in series/parallel. It is a good idea to email a pdf file of your design to your TA and have him/her verify your design before you build it. You can solder 24 gauge solid wires to the audio jack to simplify connecting your circuit to the headphones. Plug your headphones into your audio jack and use your ohmmeter to determine which two wires to connect. You have the option of connecting just the left, just the right, or both channels of the headphones. Draw interface circuits for three switches, which will be used for the piano keyboard.

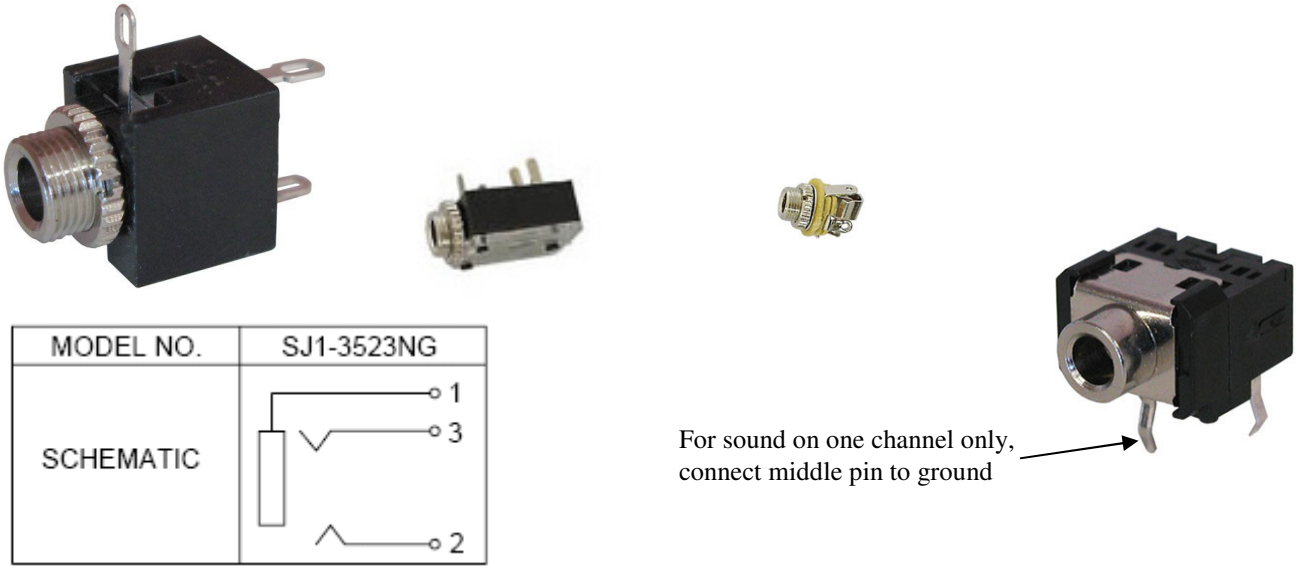


Figure 5.2. A mono jack has three connections (for one channel connect pin 1 to ground and pin 3 to the DAC output).

Part b) You must write the entire code in C. In addition, if you perform the extra credit, it too should be implemented in C. Write the C language device driver for the DAC interface. Include at least two functions that implement the DAC interface. For example, you could implement the function **`DAC_Init()`** to initialize the DAC, and the function **`DAC_Out`** to send a new data value to the DAC. Place all code that accesses the DAC in a separate DAC.c code file. Add a **DAC.h** header file with the prototypes for public functions. Describe how to use a module in the comments of the header file. Describe how the module works, how to test the module, and how to change module in the comments of the code file.

Part c) Begin with the static testing of the DAC. You will write a simple main program to test the DAC, similar in style as Program 6.1. You are free to debug this system however you wish, but you must debug The DAC module separately. You should initially debug your software in the simulator (Figure 6.3). You can single step this program, comparing digital **`Data`** to the analog voltage at the V_{out} without the speaker attached (i.e., left side of Figure 6.1).

```
#include "DAC.h"
int main(void){
  unsigned long Data; // 0 to 15 DAC output
  PLL_Init();         // like Program 4.6 in the book, 80 MHz
  DAC_Init();
  for(;;) {
    DAC_Out(Data);
    Data = 0x0F&(Data+1); // 0,1,2...,14,15,0,1,2,...
  }
}
```

Program 5.1. A simple program that outputs all DAC values in sequence.

Using Ohm's Law and fact that the digital output voltages will be approximately 0 and 3.3 V, make a table of the theoretical DAC voltage and as a function of digital value (without the speaker attached). Calculate resolution, range, precision and accuracy. Complete Table 6.3 and attach it as a deliverable (alternatively you could enter this data into **DACdata.xls**).

Bit3 bit2 bit1 bit0	Theoretical DAC voltage	Measured DAC voltage
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Table 5.3. Static performance evaluation of the DAC (if you implement a 6-bit DAC, then make a table with 16 measurements: 0,1,7,8,15,16,17,18,31,32,33,47,48,49,62,63).

Figure 5.3 shows the static testing using program 5.1 running in the simulator.

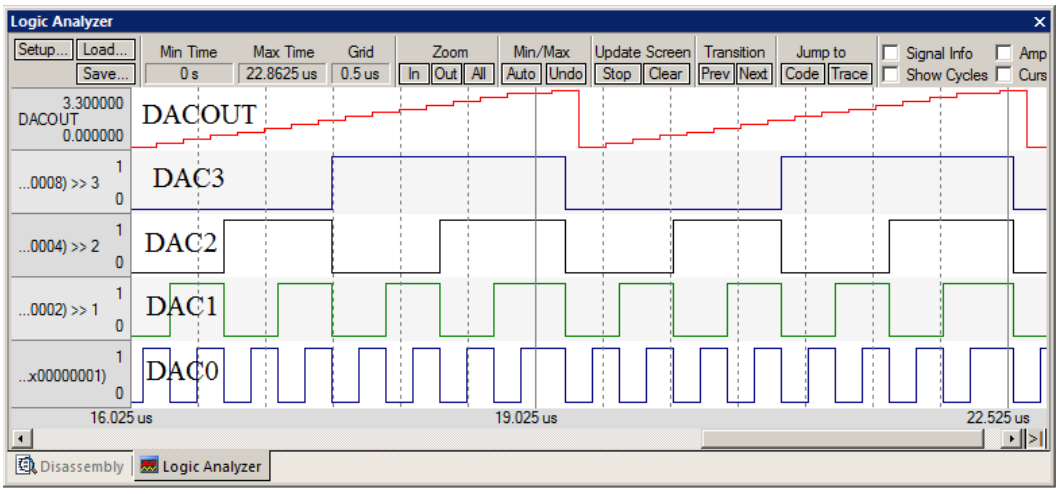


Figure 5.3. A screenshot in simulation mode showing the static testing of the DAC.

Part d) Design and write the piano keyboard device driver software. These routines facilitate the use of the four piano keys. Include at least two functions that implement the piano keyboard interface. For example, you could implement the function **`Piano_Init()`** to initialize the switch inputs, and the function **`Piano_In`** that returns a logical key code for the pattern of switches that are pressed. Place all code that directly accesses the three switches in a separate **Piano.c** code file. Add a Piano.h header file with the prototypes for public functions. Add comments that describe what it does in the **Piano.h** file and how it works in the **Piano.c** file.

Part e) Design and write the sound device driver software. The input to the sound driver is the pitch of the note to play. SysTick interrupts will be used to set the time in between outputs to the DAC. Add minimally intrusive debugging instruments to allow you to visualize when interrupts are being processed. Include at least two functions that implement the sound output. For example, you could implement the function **`Sound_Init()`** to initialize the data structures, calls **`DAC_Init`**, and initializes the SysTick interrupt. You could implement a function **`Sound_Play(note)`** that starts sound output at the specified pitch. Place all code that implements the waveform generation in a separate **Sound.c** code file. Add a Sound.h header file with the prototypes for public functions. Add comments that describe what it does in the **Sound.h** file and how it works in the **Sound.c** file. When you wish to play a new note you should write to **`NVIC_ST_RELOAD_R`**, changing the interrupt period, without completely initializing SysTick.

One approach to debugging is to attempt to create a sine wave with a constant frequency as shown in Figures 5.4 and 5.5. Just run the SysTick periodic interrupts and output one DAC value each interrupt. The toggling digital line shows you the interrupts are happening, and the top sine wave shows your table and DAC output are working.

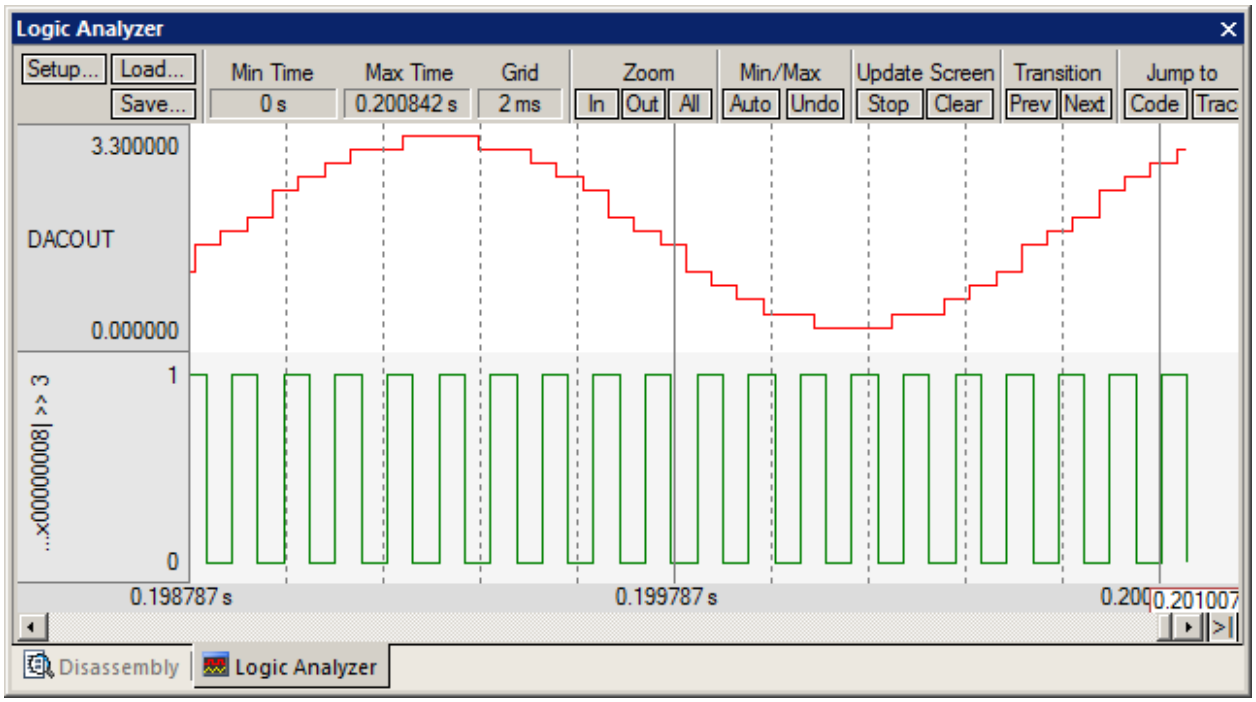


Figure 5.4. The 4-bit DAC with a 32-element table is used to create a sine wave in simulation. The top trace is the DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt. (the current version has a bug requiring you place a break point at the line that you output to the DAC so the scope window is updated)

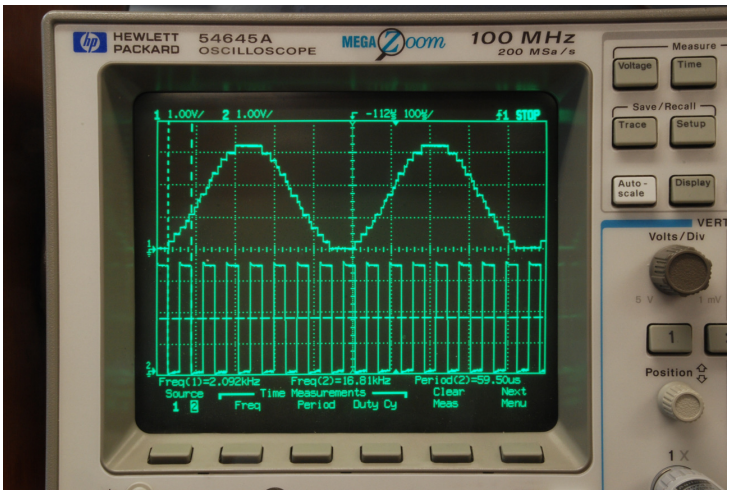


Figure 5.5. The 4-bit DAC with a 32-element table is used to create a sine wave. In this system, the software was attempting to create 2093 Hz, and the measured frequency was 2092 Hz. The top trace is the DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt.

Part f) Write a main program to implement the three-key piano. Make a heartbeat connected to an LED so you can see your program is running. Document clearly the operation of the routines. Figure 5.6 shows a possible data flow graph of the music player. Debug the system first in the simulator then on the real TM4C123 with an oscilloscope. Take a photograph of the scope traces (like Figure 5.10) to capture the waveform generated by your digital piano. When no buttons are pressed, the output will be quiet. When Button 1 is pressed, output a sine wave at one frequency. When Button 2 is pressed, output a sine wave at a second frequency. When Button 3 is pressed, output a sine wave at a third frequency. Only one button will be pressed at a time. The sound lasts until the button is released.

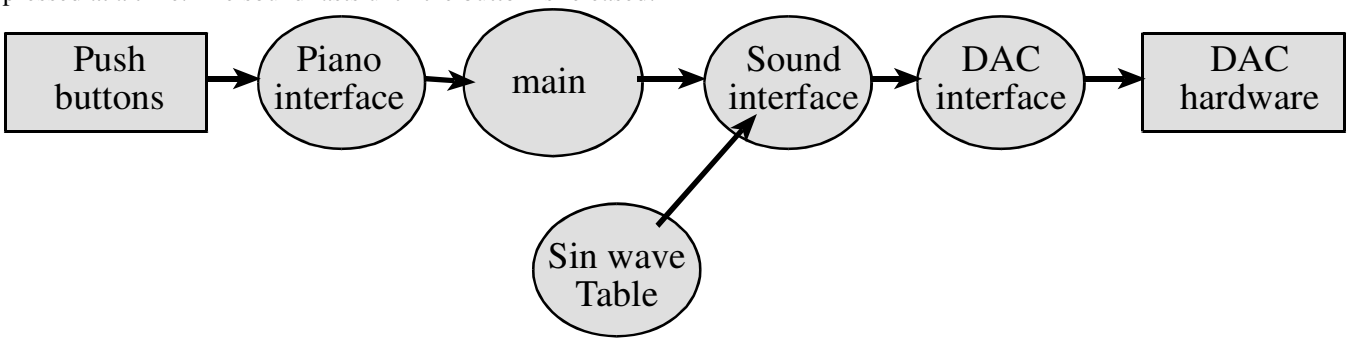


Figure 5.6. Data flows from the memory and the switches to the speaker.

Figure 5.7 shows a possible call graph of the Lab 5 system. Figure 5.8 is one possible way to implement the extra credit. Dividing the system into modules allows for concurrent development and eases the reuse of code.

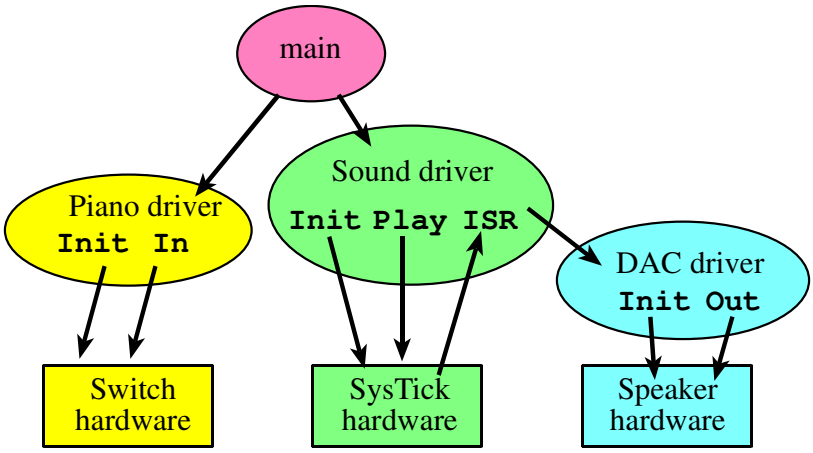


Figure 5.7. A call graph showing the three modules used by the regular part of Lab 5. Notice the SysTick hardware calls the SysTick ISR.

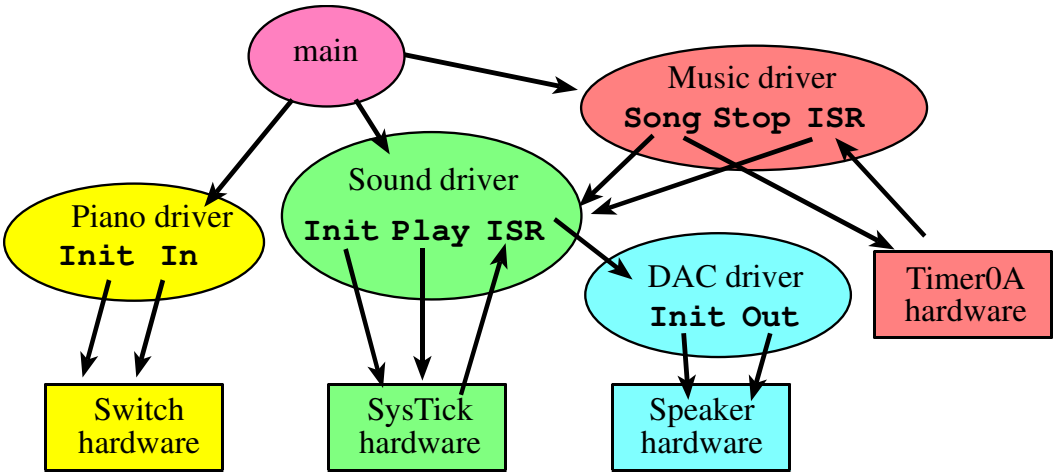


Figure 5.8. A call graph showing the three modules used by the extra credit part of Lab 6. Notice there are two interrupts.

Part g) Add a debugging monitor to your interrupt service routine. I.e., set a digital output pin high at the start of the ISR and clear it low at the end of the ISR. The scopes in the lab should have probes attached. For this part you need to use the oscilloscope debugging tool. Connect one channel to a digital output showing when the interrupts are occurring and a second channel to the DAC output (without the headphones). Set the oscilloscope to trigger on the digital output occurring in the ISR. Observe the relationship between software and hardware.

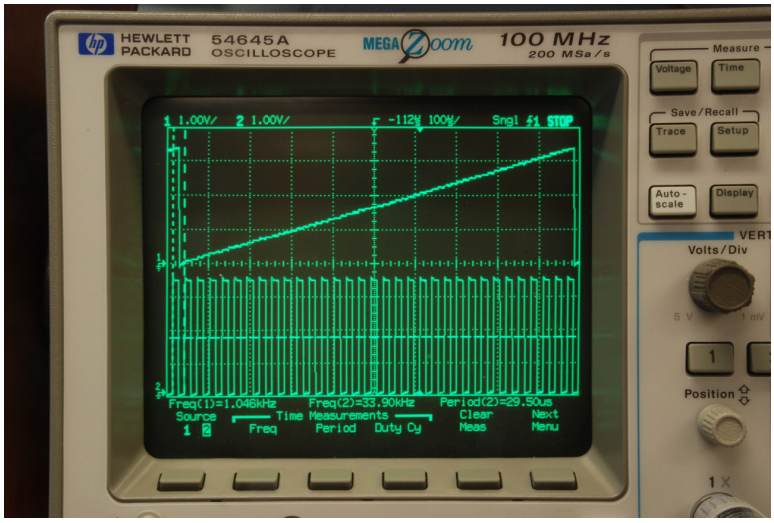


Figure 5.9. Photograph of the DAC output and debugging monitor. This system uses a 6-bit DAC and a 64-element table. Notice the DAC is monotonic (each increment of the digital input causes the DAC output to go up. (You only need 4 bits). The top trace is the 6-bit DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt.

Demonstration

You should be able to demonstrate the three notes. Be prepared to explain how your software works. You should be prepared to discuss alternative approaches and be able to justify your solution. The TA may look at your data and expect you to understand how the data was collected and how DAC works. In particular, you should be able to design a DAC with 5 to 10 bits. What is the range, resolution and precision? You will tell the TA what frequency you are trying to generate, and they may check the accuracy with a frequency meter or scope. TAs may ask you what frequency it is supposed to be, and then ask you to prove it using calculations. Just having three different sounding waves is not enough, you must demonstrate the frequency is proper and it is a sinewave (at least as good as you can get with a 4-bit DAC). You will be asked to attach your DAC output to the scope (part g). Many students come for their checkout with systems that did not operate properly. You may be asked SysTick interrupt and DAC questions. If the desired frequency is **f**, and there are **n** samples in the sine wave table, what SysTick interrupt period would you use?

This lab mentions 32 samples per cycle. Increasing the DAC output rate and the number of points in the table is one way of smoothing out the "steps" that in the DAC output waveform. If we double the number of samples from 32 to 64 to 128 and so on, keeping the DAC precision at 4-bit, will we keep getting a corresponding increase in quality of the DAC output waveform?

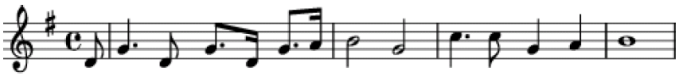
As you increase the number of bits in the DAC you expect an increase in the quality of the output waveform. If we increase the number of bits in the DAC from 4 to 6 to 8 and so on, keeping the number of points in the table fixed at 32, will we keep getting a corresponding increase in quality of the DAC output waveform?

Deliverables (Items 1, 2, 3, 4, 5 are one pdf file uploaded to Canvas by UT Student)

1) Circuit diagram showing the DAC and any other hardware used in this lab, PCB Artist
2) Software Design
   - Draw pictures of the data structures used to store the sound data
   - If you organized the system different than Figure 5.6 and 5.7, then draw its data flow and call graphs
3) A picture of the dual scope (part g) like Figures 5.5 or 5.11.
4) Measurement Data
   - Show the theoretical response of DAC voltage versus digital value (part c, Table 5.3)
   - Show the experimental response of DAC voltage versus digital value (part c, Table 5.3)
   - Calculate resolution, range, precision and accuracy
5) Brief, one sentence answers to the following questions
   - When does the interrupt trigger occur?
   - In which file is the interrupt vector?

List the steps that occur after the trigger occurs and before the processor executes the handler.
It looks like **`BX LR`** instruction simply moves LR into PC, how does this return from interrupt?

Extra Credit. Extend the system so that it plays your favorite song (a sequence of notes, set at a specific tempo). The song should contain at least 5 different pitches and at least 20 notes. But, what really matters is the organization is well-done using appropriate data structures that is easy to understand and easy to adapt for other songs. This extra credit provides for up to 20 additional points, allowing for a score of 120 out of 100 for this lab. One possible approach is to use two interrupts. A fast SysTick ISR outputs the sinewave to the DAC (Figures 5.4, 5.11). The rate of this interrupt is set to specify the frequency (pitch) of the sound. A second slow Timer ISR (see Timer started code) occurs at the tempo of the music. For example, if the song has just quarter notes at 120, then this interrupt occurs every 500 ms. If the song has eight notes, quarter notes and half notes, then this interrupt occurs at 250, 500, 1000 ms respectively. During this second ISR, the frequency of the first ISR is modified according to the note that is to be played next. The fourth step is to organize the music software into a device driver. The device driver will perform all the I/O and interrupts to make it happen. The driver will have public functions **`Song`** and **`Stop`**, which perform operations like a cassette tape player. The **`Song`** function has an input parameter that defines the song to play (passes the song "data"). If you complete the extra credit (with input switches that can be used to start and stop), then the three-key piano functionality still must be completed. In other words, the extra credit part is in addition to the regular part. You may interface more switches or you could use the on-board switches, or you can use some combination of the piano switches, to activate the **`Song`** and **`Stop`** operations



Some web links about music

Simple Music Theory http://library.thinkquest.org/15413/theory/theory.htm
Free sheet music http://www.8notes.com/piano/

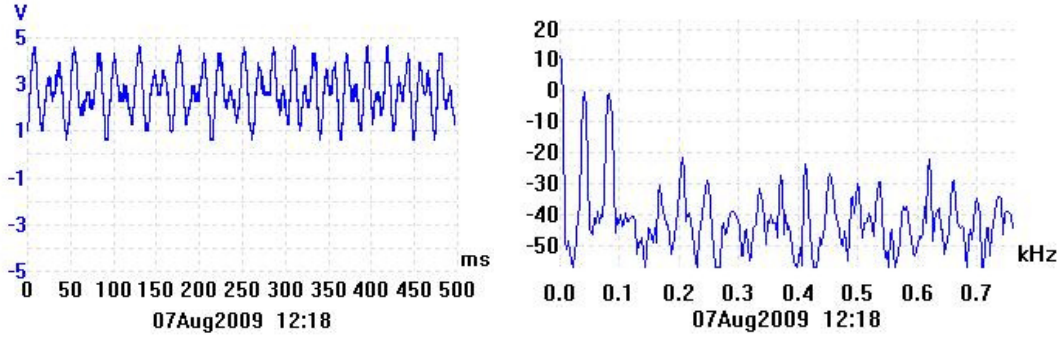


Figure 5.10. A song being played with a harmony and melody (see two peaks in the spectrum).

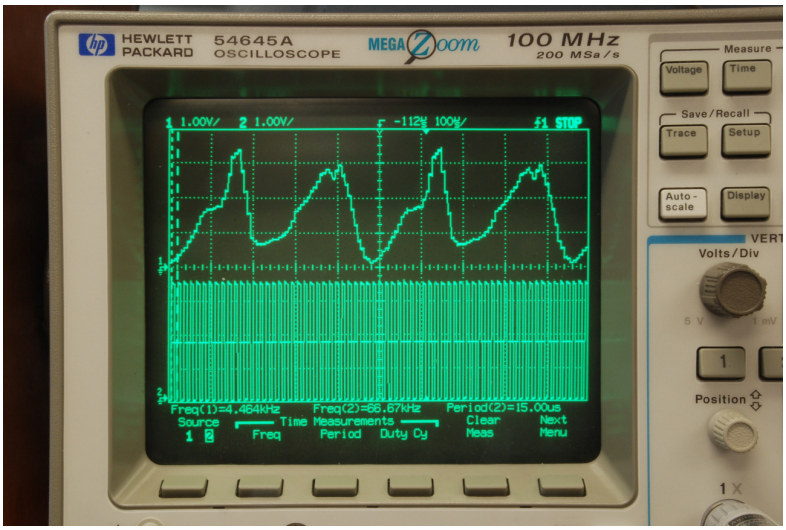


Figure 5.11. This waveform sounds like a horn (6-bit DAC, 64-element table). The top trace is the 6-bit DAC output (without headphones) and the bottom trace is a debugging monitor that toggles at each SysTick interrupt.

There should be no **`DAC_Out`** in the main loop. There should be zero or one **`DAC_Out`** each execution of the ISR. Some students will discover they can make sounds without interrupts, by simply using a software delay like Labs 4, and 5. This approach is not allowed. Other students will discover they can create sounds by performing the **`DAC_Out`** in the main loop. This approach is also not allowed. 20 point Lab 6 point deduction if zero or one calls to **`DAC_Out`** do not occur in the ISR. These other approaches are not allowed because we will be employing sounds in Lab 9, and we will need the sound to completely run within the ISR. The Lab 9 main program will start and stop sounds, but the ISR will do all the **`DAC_Out`** calls.

Soldering Guide

Reference: © John Hewes 2006, The Electronics Club, www.kpsec.freeuk.com

Safety Precautions

- **Do not solder if you are pregnant or think you might be pregnant**
- **Never touch the element or tip of the soldering iron.**
  They are very hot (about 400°C) and will give you a nasty burn.
- **Take great care to avoid touching any metal parts of the iron.**
  The iron should have a heatproof flex for extra protection.
- **Always return the soldering iron to its stand when not in use.**
  Never put it down on your workbench, even for a moment!
- **Allow joints a minute or so to cool down before you touch them.**
- **Work in a well-ventilated area.**
  The smoke formed as you melt solder is mostly from the flux and quite irritating. Avoid breathing it by keeping you head to the side of, not above, your work.
- **Wash your hands after using solder.**
  Solder contains lead.

Treatment for minor burns

Most burns from soldering are likely to be minor and treatment is simple:

- **Immediately cool the affected area under gently running cold water.**
  Keep the burn in the cold water for at least 5 minutes (15 minutes is recommended). You do not wish to freeze the tissue, so running water is much better than ice.
- **Do not apply any creams or ointments.**
  The burn will heal better without them. A dry dressing, such as a clean handkerchief, may be applied if you wish to protect the area from dirt.
- **Seek medical attention if the burn breaks the skin.**

Preparing the soldering iron

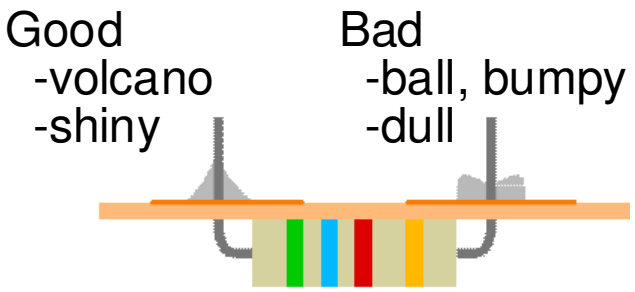
- **Place the soldering iron in its stand and plug in.**
  The iron will take a few minutes to reach its operating temperature of about 400°C.
- **Dampen the sponge in the stand.**
  The best way to do this is to lift it out the stand and hold it under a cold tap for a moment, then squeeze to remove excess water. It should be damp, not dripping wet. Don't put the electronics near the tap water
- **Wait a few minutes for the soldering iron to warm up.**
  You can check if it is ready by trying to melt a little solder on the tip.
- **Wipe the tip of the iron on the damp sponge.**
  This will clean the tip.
- **Melt a little solder on the tip of the iron.**
  This is called 'tinning' and it will help the heat to flow from the iron's tip to the joint. It only needs to be done when you plug in the iron, and occasionally while soldering if you need to wipe the tip clean on the sponge.

Preparation

- **You have two hands.**
  One hand will hold the iron and the second hand will hold the solder. This means everything else must be immobilized.
- **Immobilize the two ends to be soldered such the two ends are in physical contact.**
  In EE319K we will be soldering a wire to a component. One simple way is to 1) use masking tape to fix the component to a ceramic tile; 2) cut and strip solid wire; and 3) use masking tape to hold the wire so that the two metallic parts are touching. Another approach is to 1) cut and strip solid wire; and 2) tightly wrap the stripped portion of the wire around the component.

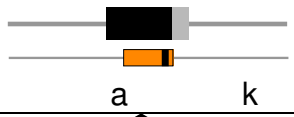
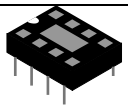
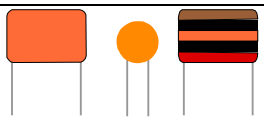
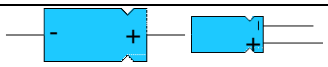
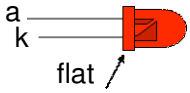
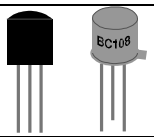
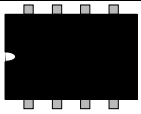
Making soldered joints

- **Hold the soldering iron like a pen, near the base of the handle.**
  Imagine you are going to write your name! Remember to never touch the hot element or tip.
- **Touch the soldering iron onto the joint to be made.**
  Make sure it touches both the component lead and the track. Hold the tip there for a few seconds and...
- **Feed a little solder onto the joint.**
  It should flow smoothly onto the lead and track to form a volcano shape as shown in the diagram below. Make sure you apply the solder to the joint, not the iron.
- **Remove the solder, then the iron, while keeping the joint still.**
  Allow the joint a few seconds to cool before you move the circuit board.
- **Inspect the joint closely.**
  It should look shiny and have a 'volcano' shape. If not, you will need to reheat it and feed in a little more solder. This time ensure that both the lead and track are heated fully before applying solder.



Soldering advice for components

Some components require special care when soldering. Many must be placed the correct way round and a few are easily damaged by the heat from soldering. Appropriate warnings are given in the following, together with other advice which may be useful when soldering.

Components	Pictures	Advice
Resistor		No special precautions are required. Connect either way round.
Diodes		Diodes must be connected the correct way round: a = anode, k = cathode. Use a heat sink with germanium diodes.
Chip holder		Ensure the notch is at the correct end. Do not insert the IC at this stage to prevent it being damaged by heat.
Ceramic, or nonpolarized capacitor		No special precautions are required. Connect either way round. Take care to identify their value.
Electrolytic or polarized capacitor		Electrolytic capacitors must be connected the correct way round, they are marked with + or - near one lead.
LED		LEDs must be connected the correct way round: a = anode (+), k = cathode (-).
Transistors		Transistors have three leads and must be connected the correct way round.
Integrated circuits		When all soldering is complete, carefully insert ICs the correct way round in their holders. Make sure all the pins are lined up before pushing in firmly.

Lab 6. LCD Device Driver for the Nokia 5110

Preparation

Read Sections 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, and 7.7 from text
Read the datasheet for the LCD display (in project folder)
Nokia5110.pdf
Download **Lab6..zip**

Purpose

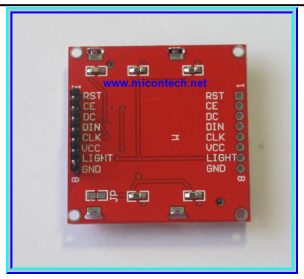
This lab has these major objectives: 1) to interface an LCD interface that can be used to display information on the embedded system; 2) to use indexed addressing to access strings; 3) to learn how to design implement and test a device driver using busy-wait synchronization; 4) to learn how to allocate and use local variables on the stack; 5) to use fixed-point numbers to store non-integer values.

Who should this lab instead of the regular lab

Do this lab if you are little more adventurous or if you wish to have a complete operational game for yourself after lab 10. The Kentec displays as part of the regular lab are 320 by 240 color LCD, and this one is 84 by 48 monochrome. However, all Kentec displays must be returned at the end of the semester. Presently they are expensive to purchase $60 plus international shipping from Europe.

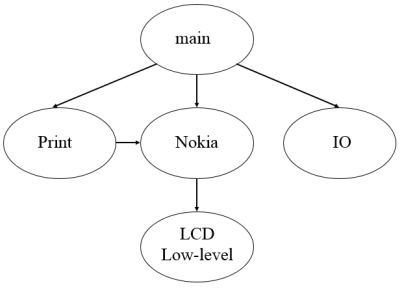
System Requirements

In this lab you will interface a Nokia 5110 LCD to the TM4C123:

Signal	**(Nokia 5110)**		**LaunchPad pin**
3.3V	(VCC, pin 6)		power
Ground	(GND, pin 8)		ground
SSI0Fss	(CE, pin 2)		connected to PA3
Reset	(RST, pin 1)		connected to PA7
Data/Command	(DC, pin 3)		connected to PA6
SSI0Tx	(DIN, pin 4)		connected to PA5
SSI0Clk	(CLK, pin 5)		connected to PA2
back light	(LIGHT, pin 7)		not connected

This 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 (wait for previous command to complete).

For the 10ms wait function, *we suggest you use the cycle-counting approach to blind wait (like Lab 2) instead of SysTick (like Lab 4) because you will need SysTick periodic interrupts for Labs 7, 8, and 9.*



One of the objective of this lab is to develop a low-level device driver for the Nokia 5110 LCD display. A higher-level driver (Nokia5110.c) is provided to you. 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. First component is the description of the driver. Since this driver will be developed in C, your descriptions are placed in the comments before each subroutine implementation (LCD.c) and the Prototype (LCD.h). In Labs 7, 8, 9, we will call these driver functions from other modules and the main, so we create a header file **LCD.h** that is included in other C files using a `#include` statement. It is during the design phase of a project that this information is specified. When developing a driver in assembly, the implementations are the instructions and comments placed inside the body of the subroutines. In addition to public functions, a device driver can also have private functions. This interface will require a private function that outputs commands to the LCD (notice that private functions do not include **`LCD_`** in their names).In this lab, you are required to develop and test seven public functions (notice that public

functions include **LCD_** or **IO_** in their names). The third component is a main program that can used to test these functions. We have given you this main program, which you can find in the **Lab6.c** file.

In the **LCD.c** file you will implement and test two functions to communicate directly with the Nokia 5110 LCD. You will not write the initialization ritual, because it is given in the **Nokia5110.c** file. However you will write these two functions that output to the LCD. Your **LCD_WriteCommand** function will be used to output 8-bit commands to the LCD, and your **LCD_WriteData** function will be used to output 8-bit data to the LCD. Built on top of your **LCD.c** file is the **Nokia5110.c**, which implements 6 public functions that can be used to display characters and graphics on the display. *You should not modify the **Nokia5110.c** file*. One of the functions, **LCD_OutChar**, outputs one ASCII character on the display, which in turn calls your low-level LCD routines. Another function, **LCD_OutString**, displays a given null-terminated String of ASCII characters by repeatedly calling **LCD_OutChar**.

```
// ************** LCD_WriteCommand ************************
// This is a helper function that sends an 8-bit command to the LCD.
// inputs: R0  8-bit command to transmit
// outputs: none
// assumes: SSI0 and port A have already been initialized and enabled
// 1) Read SSI0_SR_R and check bit 4,
// 2) If bit 4 is set loop back to step 1 (wait for BUSY bit to be zero)
// 3) Clear D/C=PA6 to zero (D/C pin configured for COMMAND)
// 4) Write the command to SSI0_DR_R
// 5) Read SSI0_SR_R and check bit 4,
// 6) If bit 4 is set loop back to step 5 (wait for BUSY bit to be zero)

// ************** LCD_WriteData ************************
// This is a helper function that sends an 8-bit data to the LCD.
// inputs: R0  8-bit data to transmit
// outputs: none
// assumes: SSI0 and port A have already been initialized and enabled
// 1) Read SSI0_SR_R and check bit 1,
// 2) If bit 1 is clear loop back to step 1 (wait for TNF bit to be one)
// 3) Set D/C=PA6 to one (D/C pin configured for DATA)
// 4) Write the data to SSI0_DR_R
```

In the IO module (**IO.c**) file you will implement and test three functions to handle a user switch and heartbeat LED.

```
//------------IO_Init------------
// Initialize GPIO Port for a switch and an LED
// Input: none
// Output: none
void IO_Init(void);

//------------IO_HeartBeat------------
// Toggle the output state of the LED.
// Input: none
// Output: none
// Invariables: This function must not permanently modify registers R4 to R11
void IO_HeartBeat(void);

//------------IO_Touch------------
// wait for release and touch of the switch; Input: none
// Output: none
// This is a public function
// Invariables: This function must not permanently modify registers R4 to R11
```

In your Print module (**Print.c**), you will implement and test two more display functions. Your **LCD_OutDec** function will be used to output an unsigned 32-bit integer to the LCD, and your **LCD_OutFix** function will be used to output an unsigned 32-bit fixed-point number to the LCD.

```
//----------------------LCD_OutDec----------------------
// Output a 32-bit number in unsigned decimal format
// Input: R0 (call by value) 32-bit unsigned number
// Output: none

// ----------------------LCD_OutFix----------------------
// Output characters to LCD display in fixed-point format
// unsigned decimal, resolution 0.001, range 0.000 to 9.999
// Inputs:  R0 is an unsigned 32-bit number
// Outputs: none
// E.g., R0=0,    then output "0.000 "
//       R0=3,    then output "0.003 "
//       R0=89,   then output "0.089 "
//       R0=123,  then output "0.123 "
//       R0=9999, then output "9.999 "
//       R0>9999, then output "*.*** "
```

LCD_OutDec (input **n** is a 32-bit number) may be implemented using either iteration or recursion. You must have at least one local variable. If you use recursion, the base case (n<10) requires one call to **LCD_OutChar**.

LCD_OutFix (input **num** is a 32-bit number) performs the following function

Parameter (**num**)	LCD display
0	"0.000 "
1	"0.001 "
999	"0.999 "
1000	"1.000 "
9999	"9.999 "
10000 or more	"*.*** "

Table6.1. Specification for the **LCD_OutFix** *function (do not display the quotes").*

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 subroutine) from the mechanisms (how the programs are implemented, which are described in the comments placed within the body of the subroutine.)

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 TA) examples of how to use your driver.

Procedure

The basic approach to this lab will be to first develop and debug your system using the simulator. During this phase of the project you will use the debugger to observe your software operation. After the software is debugged, you will run your software on the real TM4C123. 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_OutDec** or **LCD_OutFix** processes the input 187. Later, while you are debugging the actual functions on the simulator, you can single step the program and compare the actual data with your expected data.

Part a) There is a simulator for the Nokia LCD device so you are expected to first debug your code in the Simulator before you debug on the real board.

.

Part b) If your three functions in **LCD.c** are correct, the main program will output the **`"Lab 7, welcome to 319K!"`** message on LCD screen.

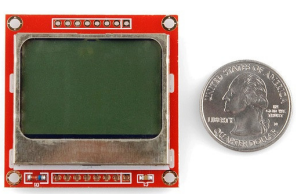
Successive refinement is a development approach that can be used to solve complex problems. If the problem is too complicated to envision a solution, you should redefine the problem and solve an easier problem. If it is still too complicated, redefine it again, simplifying it even more. You could simplify **`LCD_OutFix`**

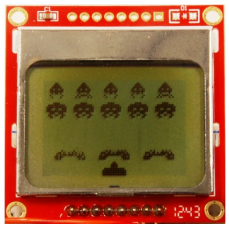
1) ignore special cases with illegal inputs
2) implement just one decimal digit

During the development phase, you implement and test the simpler problem then refine it, adding back the complexity required to solve the original problem. You could simplify **`LCD_OutDec`** in a similar fashion.

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

You will also be required to demonstrate the proper operation on the actual microcontroller. During demonstration to the TA, you will run your system in the debugger and show the binding, allocation/initialization, access and deallocation of the local variables. Each time a function is called, an **activation record** is created on the stack, which includes parameters passed on the stack (none in this lab), registered saved, and the local variables. You will be asked to observe the stack in the debugger and identify the activation records created during the execution of **`LCD_OutDec`**. TAs may ask you questions on LCD interfacing, and programming. What is the difference between post and pre-increment modes of addressing? What do the CS, RS and WR signal lines on the LCD signify? What does blind cycle synchronization mean in the context of communication with the LCD? The TA will ask to see your implementation local variables and ask you to explain the four step process (binding, allocation, access and deallocation). You should be able to draw stack pictures. How does AAPCS apply to this lab? Why is AAPCS important?





Hints

1) In simulation mode, you can look at the waveforms on the simulated scope.

2) It will be important to pass data into your functions in Register R0; this way you will be able to call your functions from C code later in Labs 8, 9, and 10. It is important to save and restore Registers R4-R11 if you use them.

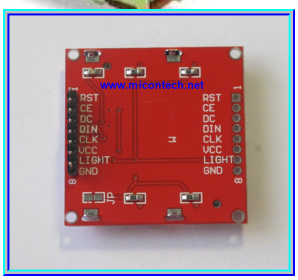
3) You MUST use the cycle-counting approach to implement the blind waits (Lab 1) instead of SysTick (Lab 4) because you will need SysTick periodic interrupts for Labs 7, 8, and 9.

4) Because we are using blind cycle synchronization, please leave the compiler optimization on **Level 0** and do not click **Optimize for Time**

Lab 7. Real-Time Position Measurement System

Preparation, starter project Lab7.zip

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



Purpose

This 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 a mixture of assembly and C; and 7) learn how to debug one module at a time.

System Requirements

You will design a **position meter**. Your software will use the 12-bit ADC built into the microcontroller. A linear slide potentiometer (Bourns SSHA20B20300) converts position into resistance ($0 \le R \le 20$ kΩ). Depending on how you solder the wires to the pot, 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 7.1. You may use any ADC channel. The TM4C123 ADC will convert voltage into a 12-bit digital number (0 to 4095). This ADC is a successive approximation device with a conversion time on the order of several µsec. 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 the Lab 6. You may use either the Nokia or Kentec display. 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 as part of Lab 6. The ADC and PLL code can be written in either assembly or C, your choice. The device drivers should be in separate files: **adc.s** (or **adc.c**), and **pll.s** (or **pll.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.

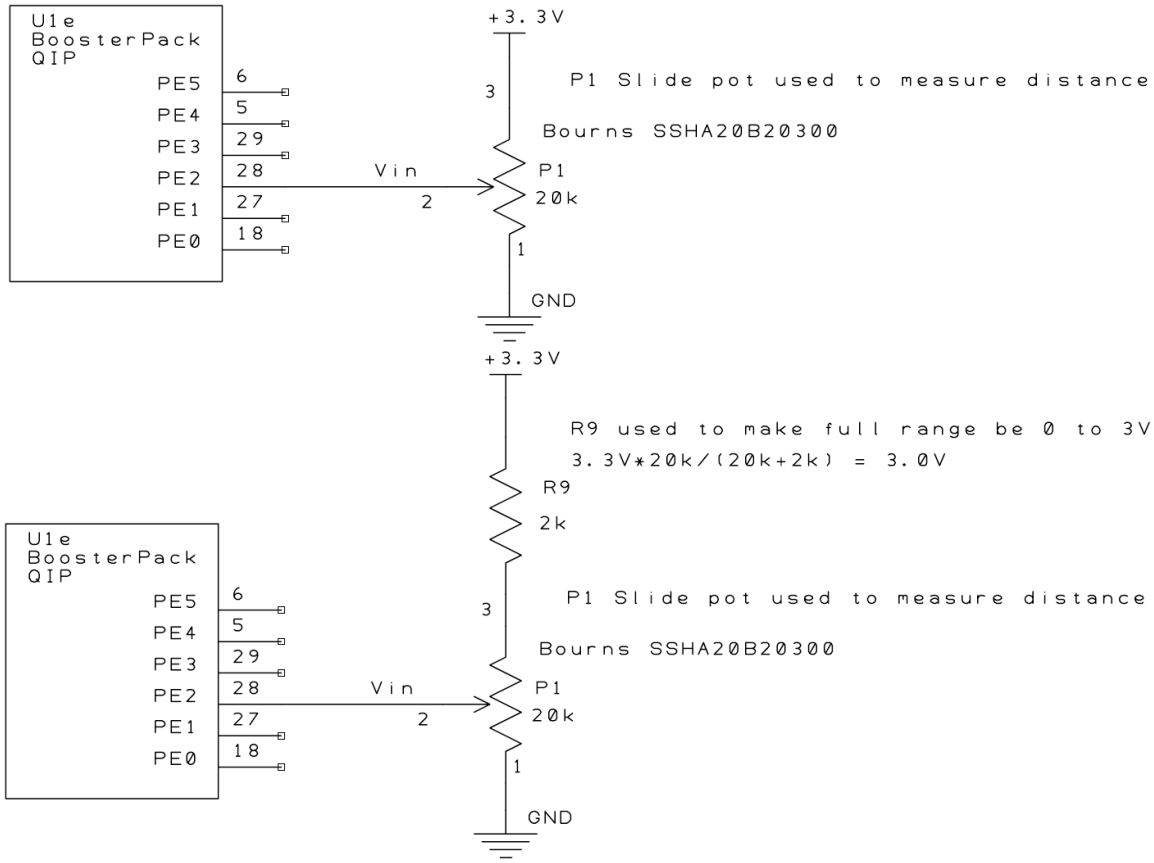


Figure 7.1. Possible circuits to interface the sensor (use your ohmmeter on the sensor to find pin numbers 1,2,3).

The left of Figure 7.2 shows a possible data flow graph of this system. Dividing the system into modules allows for concurrent development and eases the reuse of code. The right of Figure 7.2 shows a possible call graph.

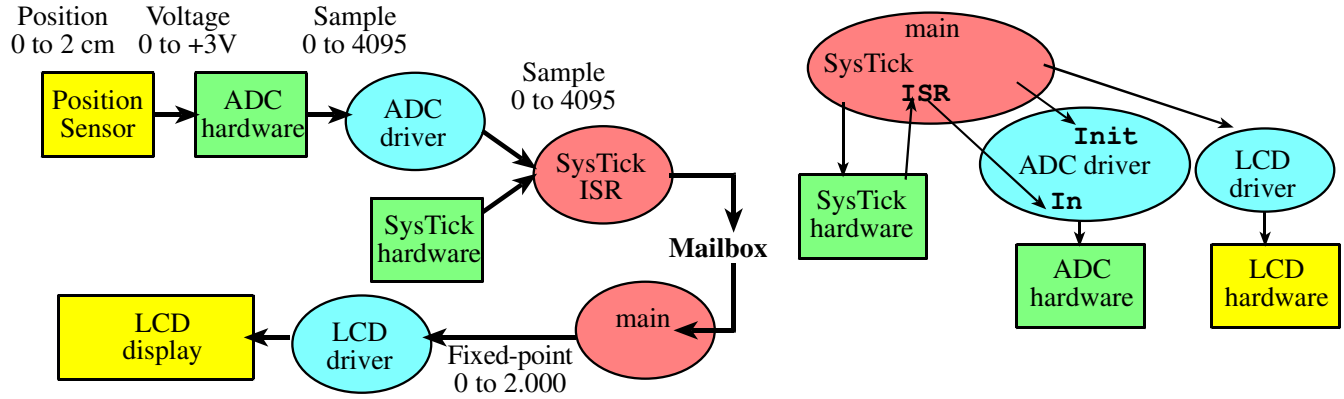


Figure 7.2. Data flow graph and call graph of the position meter system. Notice the hardware calls the ISR.

You should make the position resolution and accuracy as good as possible using the 12-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 amount of electrical noise, the number of ADC bits, and the resolution of the output display software. Considering just the errors due to the 12-bit ADC, we expect the resolution to be about 2cm/4096 or about 0.0005 cm. **Accuracy** is defined as the absolute difference between the true position and the value measured by your device. Accuracy is dependent on the same parameters as 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.

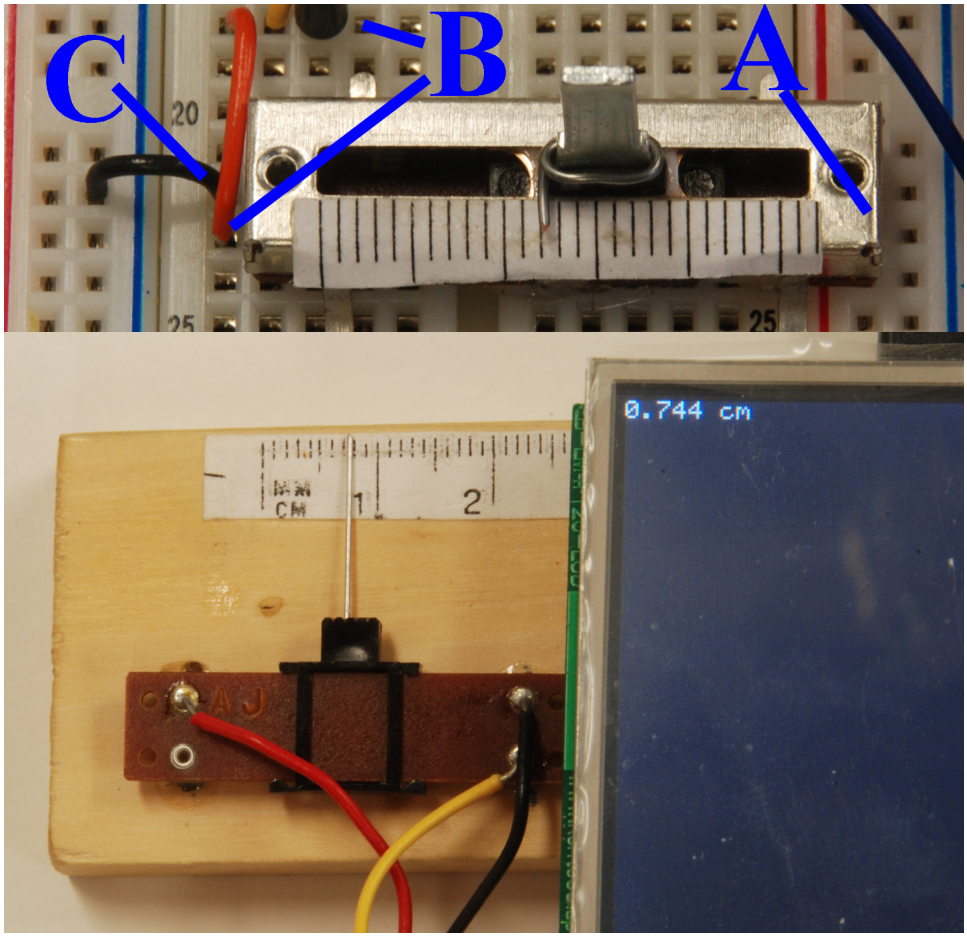


Figure 7.3. Hardware setup for Lab 7, showing the LCD and slide pot. The slide pot (similar to the one you have) is used to measure distance. In this system the label A is +3.3V, which is also connected to pin 3 of the potentiometer. The label B is pin 2 of the potentiometer, which is also connected to PE2. Label C is ground, which is also connected to pin 1 of the potentiometer.

In this lab, you will be measuring the position of the armature (the movable part) on the slide potentiometer, see Figure 7.3. Due to the mass of the armature and the friction between the armature and the frame, the position signal has a very low frequency response. One way to estimate the bandwidth 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, we need a sampling rate greater than 20 Hz. Consequently, you will create a system with a sampling rate of 40 Hz. You will sample the ADC exactly every 0.025 sec and calculate position using decimal fixed-point with Δ of 0.001 cm. You should display the results on the LCD, including units. In general, when we design a system we choose a display resolution to match the expected measurement resolution. However in this case, the expected measurement resolution is 0.0005 cm, but the display resolution is 0.001 cm. This means the display may be the limiting factor. However we expect electrical noise and uncertainty about exactly where the measurement point is to determine accuracy and not the display or the ADC resolution. In most data acquisition systems the noise and transducers are significant sources of error. A SysTick interrupt will be used to establish the real-time periodic sampling.

Nyquist Theorem: If f_{max} is the largest frequency component of the analog signal, then you must sample more than twice f_{max} in order to faithfully represent the signal in the digital samples. For example, if the analog signal

is A + B sin(2πft + ϕ) and the sampling rate is greater than 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.

When a transducer is not linear, you could use a piece-wise linear interpolation to convert the ADC sample to position (Δ of 0.001 cm.) One approach is to use piece-wise linear interpolation. In this approach, there are two small tables **`Xtable`** and **`Ytable`**. The **`Xtable`** contains the ADC results and the **`Ytable`** contains the corresponding positions. The ADC sample is passed into the lookup function. 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. You are free to implement the conversion in any acceptable manner, however your conversion function must be written in C.

A second approach to the conversion is to implement *Cubic Interpolation*. One description of Cubic Interpolation can be found in the following document online: **http://paulbourke.net/miscellaneous/interpolation/**. A third approach, shown in Figure 7.7, is to fit a linear equation to convert the ADC sample to position (Δ of 0.001 cm.)

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; 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 +3V as the position ranges from 0 to P_{max}, where P_{max} may be any value from 1.5 to 2 cm. The minimum position is established by where the paper scale is glued to the frame. Glue the paper so that a position of 0 can be measured.

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 12-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 4095) into a 16-bit unsigned decimal fixed-point number, with a Δ of 0.001 cm. Lastly, your main program will use your **`LCD_OutFix`** function from the previous lab to display the sampled signal on the LCD. Include units on your display.

Part a) Your transducer fits into the protoboard directly, see TA for help.
Label these three wires +3.3 (Pin3), Vin (Pin2), and ground (Pin1), as shown in Figure 7.1. The potentiometer may have places where it is nonlinear. The full scale range may be any value from 1.5 to 2.0 cm.

Part b) You will notice the Lab 7 starter project has four modules. Nokia5110.c module is the same one you were given in Lab6. Two of the other modules, Print.c and LCD.c are ones you wrote in Lab6 and are used here without any modifications – copy them (you will not need the IO module from Lab6). You are responsible for the fourth module ADC.c.

Part c) Write two functions: **`ADC_Init`** 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. The example code in the book uses channel 9 and sequencer 3. You can use whichever sequencer you wish, but you must use a different analog pin than PE4=Ain9. You are required to understand every line of the your ADC code. 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 TM4C123 to show the sensor and ADC are operational. The first main program will look something like the following:

```
#include "pll.h"
#include "ADC.h"
unsigned long Data;
int main(void){       // single step this program and look at Data
  PLL_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
```

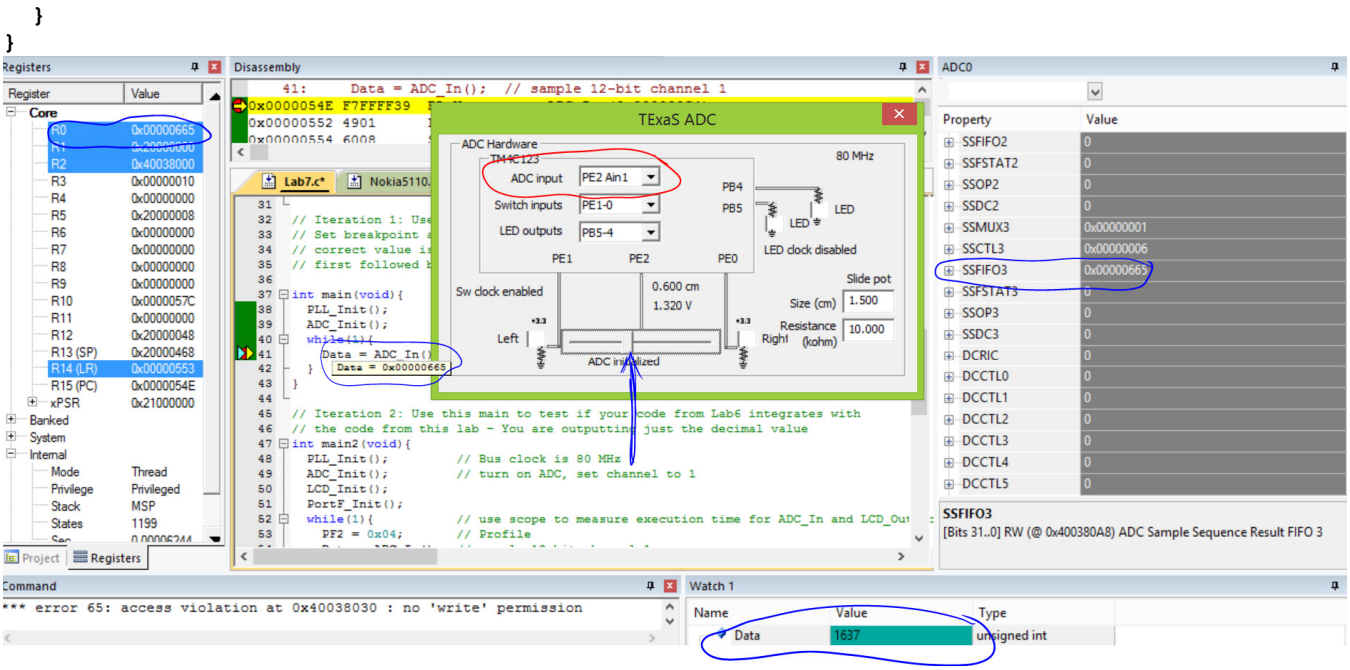


Figure 8.5. Screen shot showing one way to debug the ADC software in Simulation. You adjust the slide potentiometer, and then see the ADC result. You can observe the ADC result in four places(blue circles). The ADC result is in R0 after the call to the function; it is stored in the variable Data(hover/watch); and it is in the ADC register SSFIFO3. You can also see in the simulator the pin on which the slide-pot is connected (red circle).

Part d) 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_OutDec** developed in the lab6. Connect PF2 to a real oscilloscope or real logic analyzer 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 7.6 shows for this system the ADC requires 9 µsec to convert and the LCD requires xx µsec to display. The second main program will look something like the following:

```
unsigned long Data;
int main(void){
  PLL_Init();          // Bus clock is 80 MHz
  ADC_Init();          // turn on ADC, set channel to 1, 64-point hardware averaging
  LCD_Init();
  LCD_SetTextColorRGB(YELLOW);
  PortF_Init();
  while(1){            // use scope to measure execution time for ADC_In and LCD_OutDec
    PF2 = 0x04;        // Profile
    Data = ADC_In();   // sample 12-bit channel 1
    PF2 = 0x00;        // Profile
    LCD_Goto(0,0);
    LCD_OutDec(Data); LCD_OutString("    ");
  }
}
```

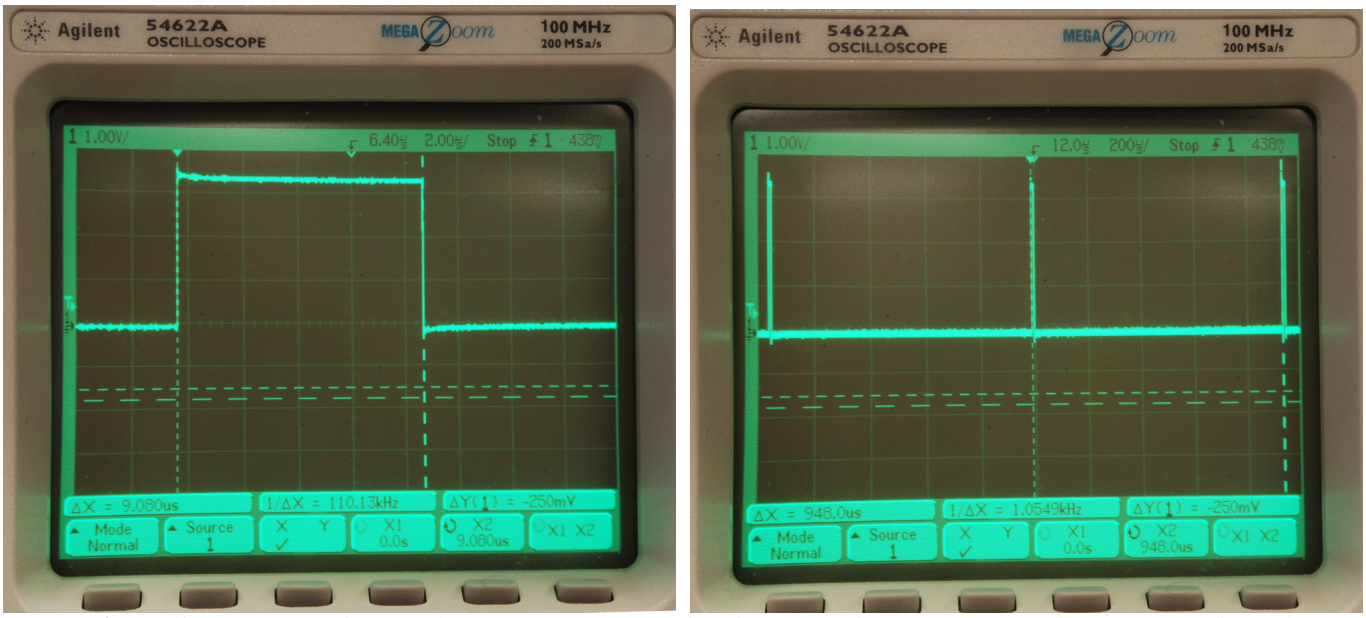


Figure 7.6. Oscilloscope trace showing the execution time profile. There are a lot of parameters that effect speed, so please expect your time measurements to be different. For this system it takes 9µs for the ADC to sample and 948 µs to output.

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 7.1. Rather, collect your own data like the first three columns of Table 7.1.

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

Part e) 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 7.1 shows some example results; this data is plotted in Figure 7.7. You are allowed to use a linear equation to convert the ADC sample into the fixed-point number. Please consider overflow and dropout during this conversion. 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
```

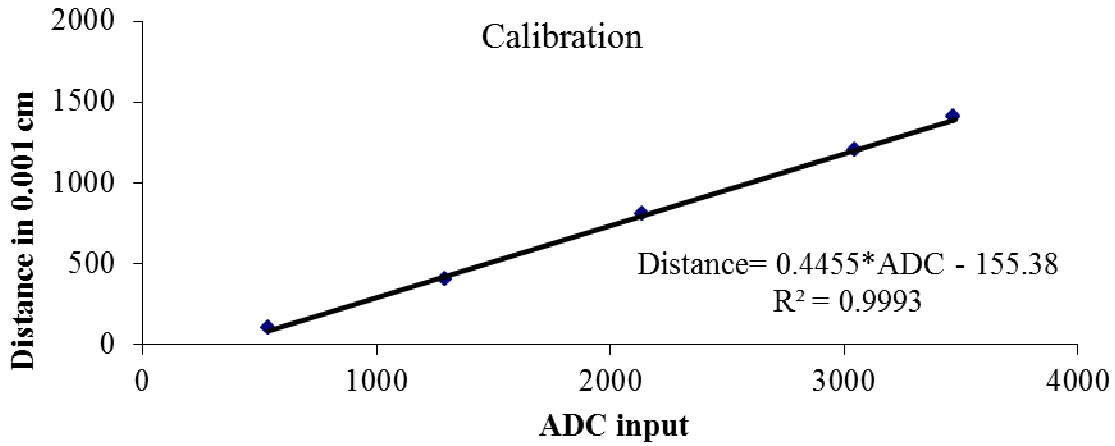


Figure 7.7. Plot of the conversion transfer function calculating fixed-point distance as a function of ADC sample.

The third main program will look something like

```
unsigned long Data;       // 12-bit ADC
unsigned long Position;  // 32-bit fixed-point 0.001 cm
int main(void){
  PLL_Init();          // Bus clock is 80 MHz
  LCD_Init();
  LCD_SetTextColorRGB(YELLOW);
  PortF_Init();
  ADC_Init();         // turn on ADC, set channel to 1
  while(1){
    PF2 ^= 0x04;      // Heartbeat
    Data = ADC_In(); // sample 12-bit channel 1
    Position = Convert(Data);
    LCD_Goto(0,0);
    LCD_OutDec(Data); LCD_OutString("    ");
    LCD_OutFix(Position);
  }
}
```

Part f) Write a C function: **`SysTick_Init`** 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 6 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 5 to implement the 25-ms periodic interrupt needed for this lab.

Part g) Write a C SysTick interrupt handler 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 each time the ADC is sampled. The frequency of the toggle is a measure of the sampling rate. The interrupt service routine performs these tasks

1) toggle heartbead 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

Part h) Write main program number 4, which initializes the PLL, timer, LCD, ADC and SysTick interrupts. 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 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 (LCD_GoTo is faster than LCD_Clear)

Debug this system on the real TM4C123. Use a logic analyzer or oscilloscope to observe the sampling rate. Take a photograph or screen shot of the LED toggling that verifies the sampling rate is exactly 40 Hz, see Figure 7.8.

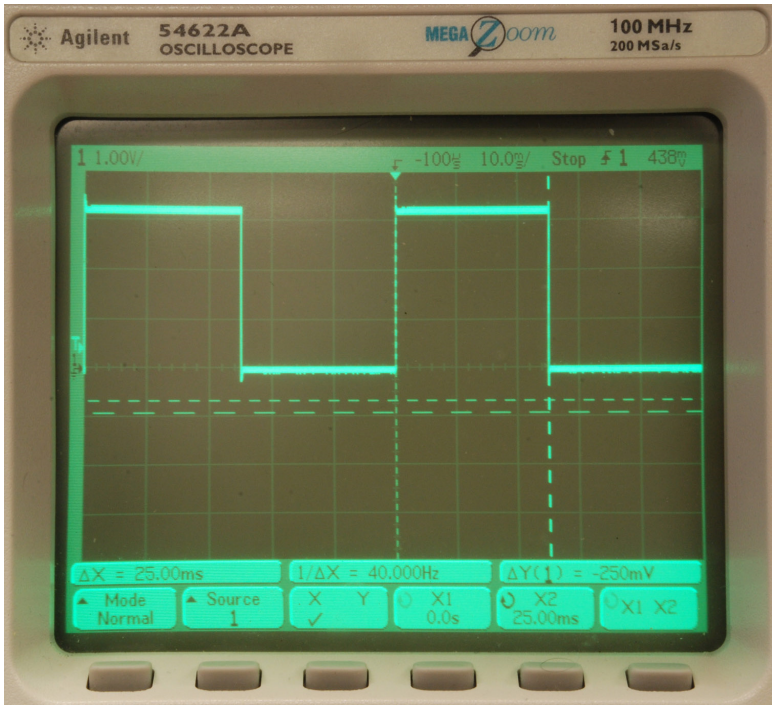


Figure 7.8. The oscilloscope connected to the LED pin verifies the sampling rate is 40 Hz.

Part i) 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). Calculate average accuracy by calculating the average difference between truth and measurement,

$$\text{Average accuracy (with units in cm)} = \frac{1}{n}\sum_{i=1}^{n}\left|x_{ti} - x_{mi}\right|$$

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

Table 7.2. Accuracy results of the position measurement system.

Demonstration

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)

1) Circuit diagram showing the position sensor, hand-drawn or PCB Artist, like Figure 7.1 (part a),
2) Time measurements and a photo showing the ADC/LCD execution time profile, like Figure 7.6 (part d)
3) Calibration data, like the first three columns of Table 7.1 (part d)
4) A photo or screenshot verifying the sampling rate is 40 Hz, like Figure 7.8 (part h)
5) Accuracy data and accuracy calculation, Table 7.2 (part i)

All code in the source files that you have changed must be cut-and-pasted in the pdf document.

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 the 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 7.1 is the desired software output, assuming a linear fit between ADC and position. The fifth column of Table 7.1 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.

Lab 8. Distributed Data Acquisition System

Preparation

Read Sections 8.2, and 11.4 on the UART
Read Sections 11.1, 11.2, and 11.3 on communication systems and the FIFO queues
Download these examples FIFO_1968.zip, UART_4F120.zip, and UART2_4F120
Starter project Lab8.zip

Teams

To fully test your system you will need two boards. However, you can perform initial testing by connecting the transmitter output to the receiver input on one board. We suggest you find another group with whom you will finish testing your system. You will not be checking out the same team as a team with whom you debug. Both groups will have to design, build and test separate systems. That is you WILL NOT split the lab into two parts with one team working on the Sender and other working on the Receiver. Each team is responsible for their own system. At the time of checkout, the TA will select another group with whom you will be checked out. It is also possible for there to be a TA board with which you will check out. During checkout your system must operate as both a transmitter and a receiver. You cannot share code with other EE319K teams past, present, or future.

Purpose

The objectives of this lab are: 1) to learn how the UART works using both busy-wait and interrupt synchronization; 2) to employ a FIFO queue to buffer data; and 3) to study synchronization issues in a distributed data acquisition system. This lab is demonstrated on two TM4C123 boards. Although it is strongly suggested you debug first in the simulator, there is no requirement that you must first debug in the simulator.

System Requirements

You will extend the system from Lab 7 to implement a distributed system. In particular, each TM4C123 will sample the data at 40 Hz, transmit the data to the other system, and the other system will display the results on its LCD. Basically the hardware/software components from Lab 7 will be divided and distributed across two TM4C123 microcontrollers. Communication will be full duplex, so the slide pot position will be displayed to the other system. Figure 8.2 shows a possible data flow graph of the distributed data acquisition system.

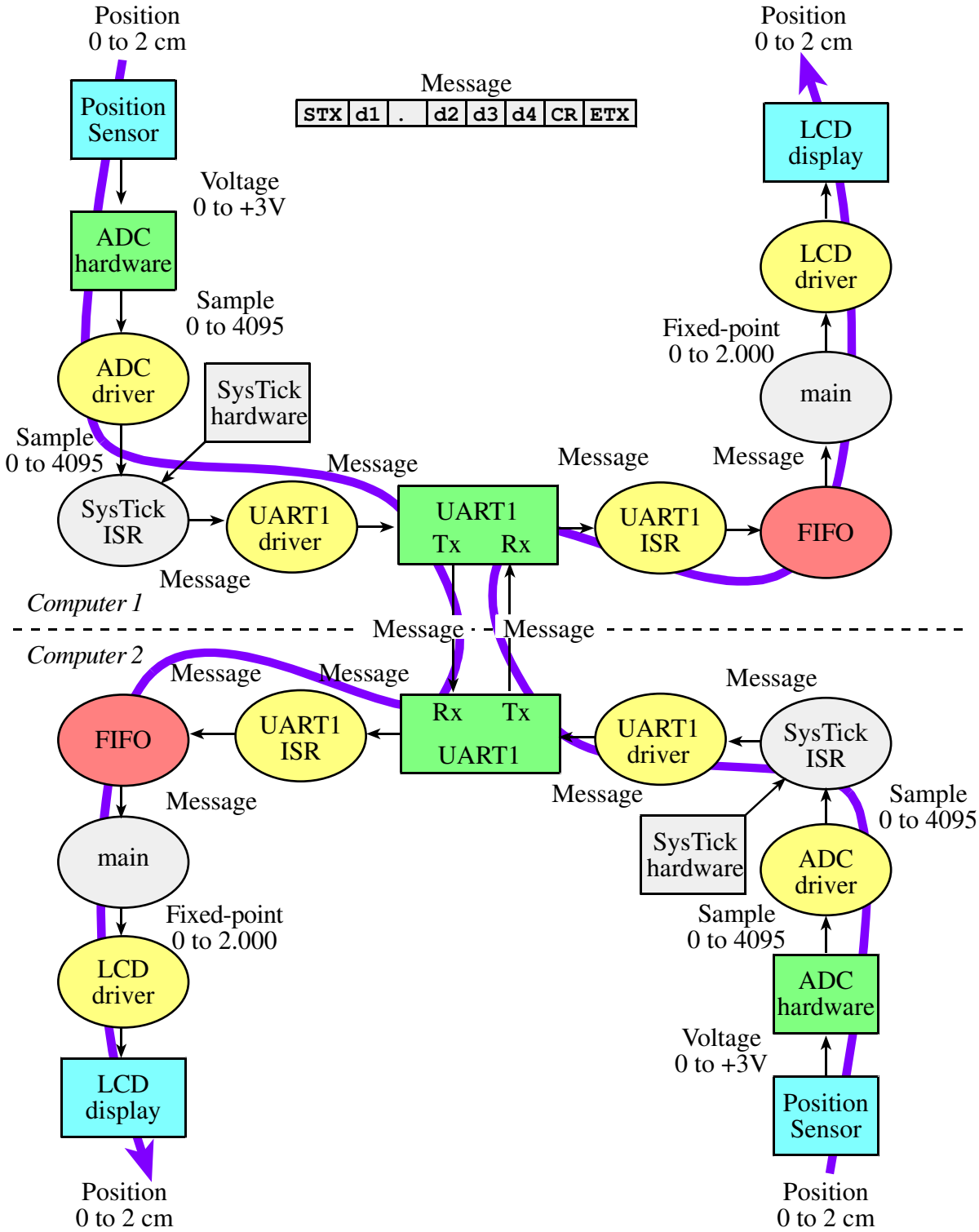


Figure 8.2. Data flows from the sensor on one computer to the LCD on the other computer. SysTick interrupts are used to trigger the real-time ADC sampling. You will use a 3-wire serial cable to connect the two UART ports. The full scale range of distance is exactly the same as Lab 7 and depends on the physical size of the slide potentiometer. Shown here as UART1, but any available UART can be used.

A sensor is attached to each computer, and the ADC (**`ADC_In`** function) generates a 12-bit digital value from 0 to 4095. The SysTick periodic interrupt establishes the real-time ADC sampling at 40 Hz. Your convert function will create a calibrated fixed-point representation of the distance. The calibration conversion occurs on the same computer as the sensor. You will then encode this information into an 8-byte message and send the message from one computer to the other computer using the universal asynchronous serial transmitter (UART). There are five choices for the UART hardware:

- PC4, PC5 UART1
- PC4, PC5 UART4
- PC5, PC6 UART3
- PE0, PE1 UART7
- PD6, PD7 UART2 (PD7 will need unlocking, similar to the way PF0 needed unlocking)

The UART transmitter must use busy-wait synchronization, and the UART receiver must use interrupt synchronization. In order to allow your board to operate with any other student's board, we will fix the communication protocol. Each measurement will be sent as exactly eight serial frames at 100,000 bits/sec:

STX, $ASCII_1$, '.', $ASCII_2$, $ASCII_3$, $ASCII_4$, CR, ETX

For example, if the distance is 1.234 cm, then the following message should be sent.

0x02, 0x31, 0x2E, 0x32, 0x33, 0x34, 0x0D, 0x03

Students can borrow extra cable from the TA to use to connect the UART ports between two TM4C123 boards. One wire connects the grounds together, and the other two wires connect Tx to Rx and Rx to Tx. *Ask the TA to cut you a section of cable.*

If you were transmitting on a network with more than two nodes, the protocol would need to include the destination address. Furthermore, if you wanted to transmit other types of data, the protocol would need include data type. If the channel were noisy, then extra frames could be added to the message to detect and correct errors. Network addressing, message type specification, data compression, error detection, and error correction are not to be implemented in this lab.

Figure 8.3 shows a possible call graph of the system. The **`main`** program will initialize SysTick, the ADC, the FIFO, the LCD and the UART, but after initialization the **`main`** program only performs receiving functions. Notice the race condition where **`main`** can be calling **`Get`** at the same time as the UART ISR calls **`Put`**. The call graph shows UART1, but you can use any of the available UARTs. There is no race condition for the shared UART because the transmitting and receiving functions are separate. Dividing the system into modules allows for concurrent development and eases the reuse of code.

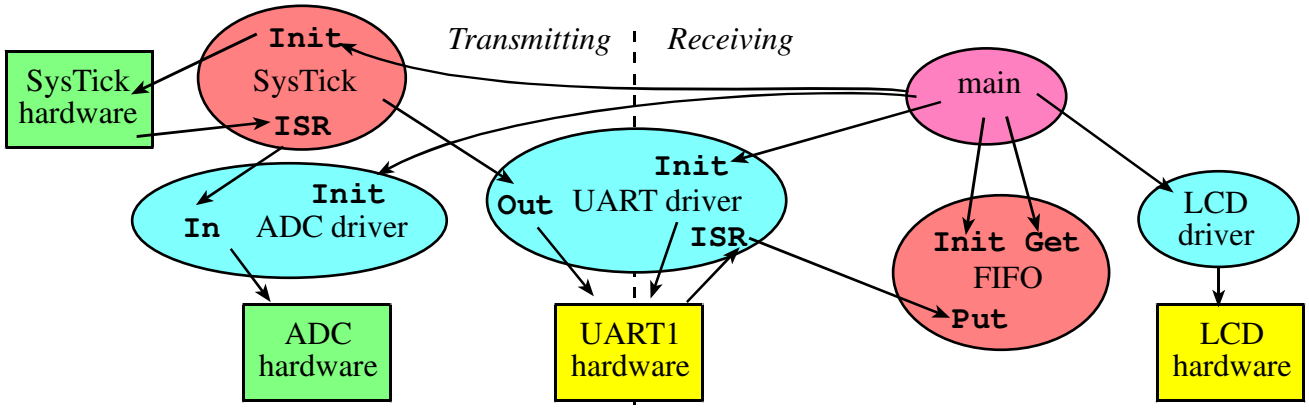


Figure 8.3. A call graph showing the modules used by the distributed data acquisition system.

There are many options for arming interrupts for the receiver. However, we require you to arm just RXRIS with ½ full FIFO. This way you get one interrupt for each 8-frame message received. The main program in each computer will output the position of the other sensor on its LCD.

Think about what the baud rate means and how it is used. Can you find a mathematical relationship between the ADC sampling rate (assuming the 8-byte message is transmitted in each sample) and the slowest baud rate possible for the transmission UART? For example, if I were to reduce the ADC's sampling rate, then I could also reduce the baud rate of the transmission UART. Think about how the baud rate for the receiver is selected. Is there an advantage to setting the baud rate of the receiver UART faster or slower than the baud rate of the transmitter? Assume that hardware and software FIFOs are deep enough to not affect these decisions.

Procedure *(shown below as UART1, but could be any of the available UARTs)*
Transmitter software tasks
Part a) Write a C function: **`UART1_Init`** that will initialize your UART1 transmitter. This will be similar to book Program 8.1, but with UART1 and with 100,000 bps baud rate. Busy-wait synchronization will be used in the transmitter.

1) enable UART1 transmitter (no interrupts)
2) set the baud rate

Part b) Write a C function: **`UART1_OutChar`** for the transmitter that sends one byte using busy-wait synchronization. Place all the UART1 routines in a separate UART1.s or UART1.c file.

1) Wait for TXFF in `UART1_FR_R` to be 0
2) Write a byte to `UART1_DR_R`

Part c) Modify the SysTick interrupt handler from Lab 8 so that it samples the ADC at 40 Hz and sends the 8-frame message to the other computer using UART1. The SysTick interrupt service routine performs these tasks

1) toggle a heartbeat (change from 0 to 1, or from 1 to 0),
2) sample the ADC
3) toggle a heartbeat (change from 0 to 1, or from 1 to 0),
4) convert to distance and create the 8-byte message
5) send the 8-byte message to the other computer (calls `UART1_OutChar` 8 times)
6) increment a `TxCounter`, used as debugging monitor of the number of ADC samples collected
7) toggle a heartbeat (change from 0 to 1, or from 1 to 0),
8) return from interrupt

Toggling the heartbeat three times allows you to see the LED toggle every 25ms, as well as measure the total execution time of the ISR using an oscilloscope attached to the heartbeat. To understand the transmitter better answer these questions

Q1) The UART transmitter has a built-in hardware FIFO. How deep is the FIFO? (How many frames can it store?)
Q2) At a baud rate of 100,000 bits/sec (1 start, 8 data, 1 stop) how long does it take to send 8 frames?
Q3) The SysTick ISR runs every 25 ms, and each ISR calls **`UART1_OutChar`** 8 times. Does this hardware FIFO ever fill? In other words will the busy-wait loop in **`UART1_OutChar`** ever loop back?

Part d) Write a temporary main program for testing the transmitter, which initializes the PLL, SysTick, ADC, UART1, heartbeat, and enables interrupts. After initialization, this transmitter test main (foreground) performs a do-nothing loop. This main program will only be used for testing, and will be replaced by a more complex main once the receiver is built. The ADC sampling and UART1 transmissions occur in the SysTick interrupt service routine running in the background. It is not possible to debug this Lab in simulation mode as we would need two instances of the simulator to communicate with each other.

To debug the transmitter on the real TM4C123, you can place the ADC sample and the interrupt counter in a debugger Watch window. The interrupt **`TxCounter`** should increase by 40 every second, and the 12-bit ADC sample should respond to changes in the slide pot. Attach a dual-channel oscilloscope to the heartbeat and UART outputs, you will see the 8 frames being transmitted at 100,000 bits/sec every 25 ms.

Questions to think about:

Q4) Using the photo captured in Figure 8.5, approximately how long does it take to sample the ADC?

Receiver software tasks

Part e) Design, implement and test a C language module implementing a statically allocated FIFO queue. You are allowed to look at programs in the book and posted on the internet, but you are required to write your own FIFO. Examples of how to make your FIFO different from the ones in the book include 1) store the data so the oldest is on top and one size counter defines the FIFO status; 2) implement it so 5, 6, or 8 bytes are put and get at a time; 3) reverse the direction of the pointers so both PutPt and GetPt decrement rather than increment; Make sure your FIFO is big enough to hold all data from one message. Place the FIFO code in a separate **`fifo.c`** file. Add a **`fifo.h`** file that includes the prototypes for the functions.

The software design steps are

1) Define the names of the functions, input/output parameters, and calling sequence. Type these definitions in as comments that exist at the top of the functions.

2) Write pseudo-code for the operations. Type the sequences of operations as comments that exist within the bodies of the functions.

3) Write C code to handle the usual cases. I.e., at first, assume the FIFO is not full on a put, not empty on a get, and the pointers do not need to be wrapped.

4) Write a main program in C to test the FIFO operations. An example main program is listed below. The example test program assumes the FIFO can hold up to 6 elements. This main program has no UART, no LCD and no interrupts. Simply make calls to the three FIFO routines and visualize the FIFO before, during and after the calls. Debug either in the simulator or on the real board. In this example, Put and Get return a true if successful, Put takes a call by value input, and Get takes a return by reference parameter. You are free to design the FIFO however you wish as long as it is different from the ones in the book, and you full understand how it works.

5) Iterate steps 3 and 4 adding code to handle the special cases.

```
uint32_t Status[20];             // entries 0,7,12,19 should be false, others true
char GetData[10];  // entries 1 2 3 4 5 6 7 8 should be 1 2 3 4 5 6 7 8
int main_fifo(void){ // --UUU-- make this main to test FiFo
  FiFo_Init();
  for(;;){
    Status[0]  = FiFo_Get(&GetData[0]);  // should fail,    empty
    Status[1]  = FiFo_Put(1);            // should succeed, 1
    Status[2]  = FiFo_Put(2);            // should succeed, 1 2
    Status[3]  = FiFo_Put(3);            // should succeed, 1 2 3
    Status[4]  = FiFo_Put(4);            // should succeed, 1 2 3 4
    Status[5]  = FiFo_Put(5);            // should succeed, 1 2 3 4 5
    Status[6]  = FiFo_Put(6);            // should succeed, 1 2 3 4 5 6
    Status[7]  = FiFo_Put(7);            // should fail,    1 2 3 4 5 6
    Status[8]  = FiFo_Get(&GetData[1]);  // should succeed, 2 3 4 5 6
    Status[9]  = FiFo_Get(&GetData[2]);  // should succeed, 3 4 5 6
    Status[10] = FiFo_Put(7);            // should succeed, 3 4 5 6 7
    Status[11] = FiFo_Put(8);            // should succeed, 3 4 5 6 7 8
    Status[12] = FiFo_Put(9);            // should fail,    3 4 5 6 7 8
    Status[13] = FiFo_Get(&GetData[3]);  // should succeed, 4 5 6 7 8
    Status[14] = FiFo_Get(&GetData[4]);  // should succeed, 5 6 7 8
    Status[15] = FiFo_Get(&GetData[5]);  // should succeed, 6 7 8
    Status[16] = FiFo_Get(&GetData[6]);  // should succeed, 7 8
    Status[17] = FiFo_Get(&GetData[7]);  // should succeed, 8
    Status[18] = FiFo_Get(&GetData[8]);  // should succeed, empty
    Status[19] = FiFo_Get(&GetData[9]);  // should fail,    empty
  }
}
```

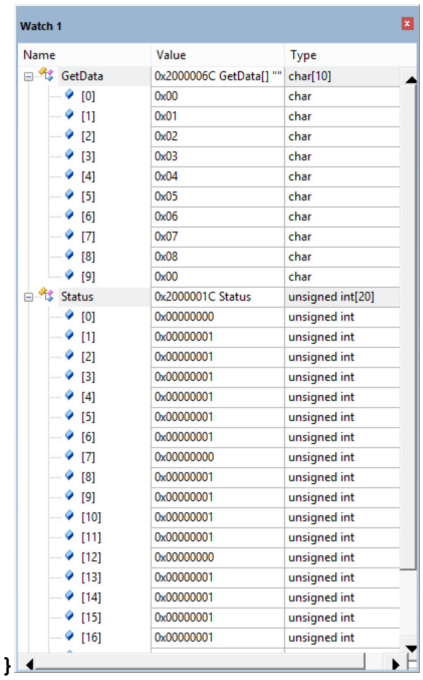


Figure 8.7. Watch window screen shot debugging the FIFO after one iteration of the for loop

Part f) Write a C function: **`UART1_Init`** that will initialize the UART1 receiver for the receiver.

1) clear a global error count, initialize your FIFO (calls `Fifo_Init`)
2) enable UART1 receiver (arm interrupts for receiving)
3) set the baud rate to 100,000 bits/sec

In general, one must enable interrupts in order for UART1 interrupts to occur. It is good style however to finish all initializations, then enable interrupts from the high-level main program. The receiver initialization with interrupts should be added to the transmitter initialization using busy-wait.

Part g) Write the UART interrupt handler in C that receives data from the other computer and puts them into your FIFO queue. Your software FIFO buffers data between the ISR receiving data and the main program displaying the data. If your software FIFO is full, increment a global error count. If you get software FIFO full errors, then you have a bug that must be removed. Do not worry about software FIFO empty situations. The software FIFO being empty is not an error. Define the error count in the UART.c file and increment it in the UART ISR.

Arm RXRIS receive FIFO ½ full (interrupts when there are 8 frames): the interrupt service routine performs these tasks

1) toggle a heartbeat (change from 0 to 1, or from 1 to 0),
2) toggle a heartbeat (change from 0 to 1, or from 1 to 0),
3) as long as the RXFE bit in the UART1_FR_R is zero
Read bytes from UART1_DR_R
Put all bytes into your software FIFO (should be exactly 8 bytes, but could be more possibly)
If your software FIFO is full (data lost) increment a global error count (but don't loop back)
(the message will be interpreted in the main program)
4) Increment a `RxCounter`, used as debugging monitor of the number of UART messages received
5) acknowledge the interrupt by clearing the flag which requested the interrupt
UART1_ICR_R = 0x10; // this clears bit 4 (RXRIS) in the RIS register
6) toggle a heartbeat (change from 0 to 1, or from 1 to 0),
7) return from interrupt

Toggling heartbeat three times allows you to see the LED flash with your eyes, as well as measure the total execution time of the ISR using an oscilloscope attached to heartbeat. Place all the UART1 routines in a separate **`UART1.c`** file.

Part h) The body of the main program reads data from the FIFO and displays the measurement using the same LCD routines developed in Lab 6 and used in Lab 7. The main program in this data acquisition system performs these tasks

1) initialize PLL
2) initialize transmitter functions (ADC, heartbeat, SysTick, UART transmission busy-wait)
3) initialize receiver functions (FIFO, LCD, heartbeat, UART receiver with interrupts)
4) enable interrupts
5) calls your FIFO get waiting until new data arrives. FIFO should return an Empty condition if the main program in the receiver calls get and the FIFO is empty. For more information on how the ISR and main use the FIFO, read Section 11.3 in the book and look at the left side of Figure 11.6. Also read Section 11.4 and look the the function **`UART_InChar`** in Program 11.6**. In this step you keep reading until you receive an STX,**
6) output the fixed-point number (same format as Lab 7) with units.
The next five characters gotten from the FIFO should be the ASCII representation of the distance
7) repeat steps 5,6 over and over

To debug the system on one TM4C123, connect the transmitter output to the receiver input. In this mode, the system should operate like Lab 7, where the sensor position is displayed on the LCD. Use the oscilloscope to visualize data like Figures 8.5 and 8.6.

To debug the distributed system on the real TM4C123, use two PC computers close enough together so you can connect the TM4C123s with the special cable, yet run **uVision4** on both computers. You can place strategic variables (e.g., ADC sample, interrupt counter) in the Watch windows on both PCs. If you start both computers at the same time all four interrupt counters should be approximately equal, and the data on both computers should respond to changes in the slide pot.

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

You will show the TA your program operation on the two TM4C123 boards. Your TA will connect a scope to either transmitter heartbeat and TxD→RxD or receiver heartbeat and TxD→RxD and expect you to explain the relationship between your executing software and the signals displayed on the scope. Also be prepared to explain how your software works and to discuss other ways the problem could have been solved. How do you initialize UART? How do you input and output using UART? What is the difference between busy-wait and interrupt synchronization? What synchronization method does the transmitter UART use? What synchronization method does the receiver UART use? What sets RXFE, TXFF, RXRIS, RTRIS and when are they set? What clears RXFE, TXFF, RXRIS, RTRIS and when are they cleared? What does the PLL do? Why is the PLL used? There are both hardware and software FIFOs in this system. There are lots of FIFO code in the book and on the web that you are encouraged to look at, but you are responsible for knowing how your FIFO works. What does it mean if the FIFO is full? Empty? What should your system do if the FIFO is full when it calls PUT? Why? What should your system do if the FIFO is empty when it calls GET? Why?

Hints

1) Remember the port name **UART1_DR_R** links to two hardware FIFOs at the same address. A write to **UART1_DR_R** is put into the transmit hardware FIFO and a read from **UART1_DR_R** gets from the receive hardware FIFO
2) You should be able to debug your system with any other EE319K group. You are not allowed to look at software written by another group, and you certainly are not allowed to copy software from another group. You are allowed to discuss strategies, flowcharts, pseudocode, and debugging techniques.
3) This system can be debugged on one board. Connect TxD to RxD.
4) This hint describes in more detail how to create a message. Assume the distance is 1.424 and your software calculates the appropriate integer 1424. First you convert the digits to 1, 4, 2, 4 in a similar way as LCD_OutFix. Next, you make the digits ASCII 0x31, 0x34, 0x32, 0x34. Add the STX(02), the decimal point (2E), the CR (0D) and ETX (03), and the message will be 0x02, 0x31, 0x2E, 0x34, 0x32, 0x34, 0x0D, 0x03. To send this message you call **`UART_OutChar`** 8 times using busy-wait. These 8 bytes will go in the hardware FIFO of the UART transmitter. The 8 bytes will be sent one at a time, one byte every 100usec.

NOT READY YET

Lab 9. Embedded System Design

Preparation

Download the starter project Lab9.zip (place BMP images on the LCD) **USE AS YOUR PROJECT**

Purpose

The objectives of this lab are: 1) design, test, and debug a large C program; 2) to review I/O interfacing techniques used in this class; and 3) to design a system that performs a useful task. There are three options for Lab 9. Option 1 is to design a 80's-style shoot-em up game like **Space Invaders**. Option 2 is to design a game called PipeDreams. The third option is to propose an alternative project similar in scope to space invaders. If you would like to propose a project prepare a one-page description by the Monday (8/11).

Interrupts must be appropriately used control the input/output, and will make a profound impact on how the user interacts with the game. You could use an edge-triggered interrupt to execute software whenever a button is pressed. You could output sounds with the DAC using a fixed-frequency periodic interrupts. You could decide to move a sprite using a periodic interrupt, although the actual LCD output should always be performed in the main program.

System Requirements for all games

- There must be at least one externally-interfaced button and one slide pot. Buttons and slide pot must affect game play. The slide pot must be sampled by the ADC.
- There must be at least three images on the LCD display that move in relation to user input and/or time.
- There must be sounds appropriate for the game. The sounds must be generated by the DAC developed in Lab 6.
- The score should be displayed on the screen (but it could be displayed before or after the game action).
- At least two interrupt ISRs must be used in appropriate manner.
- The game must be both simple to learn and fun to play.

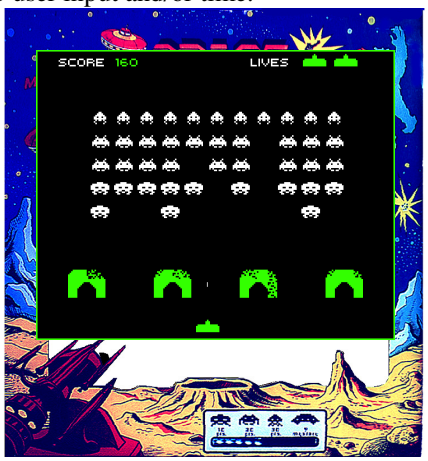


Figure 9.1. Space Invaders
http://www.classicgaming.cc/classics/spaceinvaders/index.php

System Requirements for Space Invaders, Asteroids, Missile Command, Centipede, Snood, or Defender

You will design, implement and debug 80's or 90's-style video game. You are free to simplify the rules but your game should be recognizable as one of these six simple games. Buttons and the slide pot are inputs, and the LCD and sound (Lab 6) are the outputs. The slide pot is a simple yet effective means to move your ship.

System Requirements for Pipe Dream

Pipe Dream is a game played on a grid. The player receives pipe pieces that go on the grid. Possible pieces are a 90 degree turn, a straight piece, and a cross (two straight pieces on top of each other.) Players are given these pieces randomly and cannot rotate them, and they cannot replace any already placed pieces. Which piece is to be placed next is shown to the player. The object of the game is to construct a path for water to flow using the pipe pieces given from a designated starting point on the grid to another designated end point before time runs out. Players receive a score based on how long their final path of piping was, but if they fail to connect the two points the game will be over. Inputs will select a grid space, and a single button to place the next piece available in the selected space. There will be a sound for a pipe section being placed, as well as game over and win sounds. This game must satisfy the **System Requirements for all games** listed above. Possible additions include (but not required):

- Use two slide pots for moving
- Add a "queue" of pieces that shows not only the next piece to be placed, but the next 3-5 pieces as well
- Add obstacles on the grid where pipes may not be placed (or can be placed with a score penalty)
- Add already placed pipe sections that give bonus points if used in the solution

- Instead of a time limit, have a flow of water start running through the pipes. If the water gets to an open pipe the player loses, and if the water gets to the end point the player wins.

Figure 9.2 A pipe dream screen.

System Requirements for Design Your Own Project

The first step is to make a proposal. Look for a proposal for Super Breakout (SuperBreakout.doc) on the class website. If selected your game will be added as an official choice that you and other EE319K students would be free to choose. Good projects require students to integrate fundamental EE319K educational objects such as I/O interfacing, data structures, interrupts, sound, and the effective implementation of real-time activities. This project must satisfy the System Requirements for all games listed above.The TA or professor must approve an alternate game.

Procedure

Part a) In a system such as this each module must be individually tested. Your system will have four or more modules. Each module has a separate header and code file. Possible examples for modules include slide pot input, button input, LCD, and sound output. For each module design the I/O driver (header and code files) and a separate main program to test that particular module. Develop and test sound outputs as needed for your game. There should be at least one external switch and at least one slide pot.

Part b) In the game industry an entity that moves around the screen is called a *sprite*. You will find lots of sprites in the Lab10Files directory of the starter project. You can create additional sprites as needed using a drawing program like Paint. Most students will be able to complete Lab 10 using only the existing sprites in the starter package. Because of the way pixels are packed onto the screen, we will limit the placing of sprites to even addresses along the x-axis. Sprites can be placed at any position along the y-axis. Having a 2-pixel black border on the left and right of the image will simplify moving the sprite 2 pixels to the left and right without needing to erase it. Similarly having a 1-pixel black border on the top and bottom of the image will simplify moving the sprite 1 pixel up or down without needing to erase it. You can create your own sprites using Paint by saving the images as 16-color BMP images. 24-bit BMP images can also be converted, but the files are 8 times larger and will use up your allotment of 32k on the free version of the compiler. Figure 10.3 is an example BMP image. Because of the black border, this image can be moved left/right 2 pixels, or up/down 1 pixel. Use the **BmpConvert.exe** program to convert the BMP image into a two-dimensional array that can be plced in the screen buffer at a position of your choice using the function **`Nokia5110_PrintBMP()`**. This function does not display to the LCD, to do that you call the **`Nokia5110_DisplayBuffer()`** function. For this part of the procedure you will need to write main programs for drawing and animating your sprites.

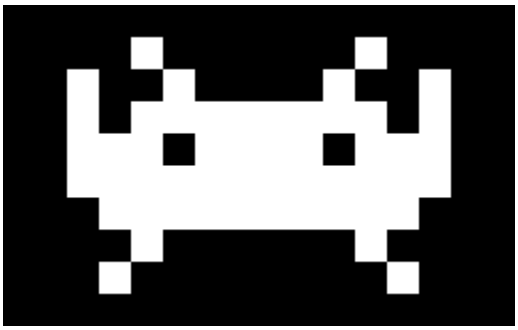


Figure 9.3. Example BMP file. Each is 16-color, 16 pixels wide by 10 pixels high.

Program 9.1 shows an example BMP file in C program format. There are 0x76 bytes of header data. At locations 0x12-0x15 is the width in little endian format. In this case (shown in blue) the width for this sprite is 16 pixels. At locations 0x16-0x19 is the height also in little endian format.

```
const unsigned char Enemy10Point1[] = {
0x42,0x4D,0xC6,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x76,0x00,0x00,0x00,0x28,0x00,
```

```
0x00,0x00,
0x10,0x00,0x00,0x00,  // width is 16 pixels
0x0A,0x00,0x00,0x00,  // height is 16 pixels
0x01,0x00,0x04,0x00,0x00,0x00,
0x00,0x00,0x50,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x80,0x00,0x00,0x80,
0x00,0x00,0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x00,0x80,0x00,0x80,0x00,0x80,0x80,
0x00,0x00,0x80,0x80,0x80,0x00,0xC0,0xC0,0xC0,0x00,0x00,0x00,0xFF,0x00,0x00,0xFF,
0x00,0x00,0x00,0xFF,0xFF,0x00,0xFF,0x00,0x00,0x00,0xFF,0x00,0xFF,0x00,0xFF,0xFF,
0x00,0x00,0xFF,0xFF,0xFF,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  // bottom row
0x00,0x0F,0x00,0x00,0x00,0x00,0xF0,0x00,
0x00,0x00,0xF0,0x00,0x00,0x0F,0x00,0x00,
0x00,0x0F,0xFF,0xFF,0xFF,0xFF,0xF0,0x00,
0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,
0x00,0xFF,0xF0,0xFF,0xFF,0x0F,0xFF,0x00,
0x00,0xF0,0xFF,0xFF,0xFF,0xFF,0x0F,0x00,
0x00,0xF0,0x0F,0x00,0x00,0xF0,0x0F,0x00,
0x00,0x00,0xF0,0x00,0x00,0x0F,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,  // top row
0xFF};
```

Program 9.1. Example BMP file written as a C constant allocated in ROM.

In this case (shown in red) the height for this sprite is 10 pixels. The **`Nokia5110_PrintBMP()`** function assumes the entire image will fit onto the screen, and not stick off the side, the top, or the bottom. There is other information in the header, but it is ignored. As mentioned earlier, you must save the BMP as a 16-color image. These sixteen colors will map to the grey scale of the LCD. The 4-bit color 0xF is maximum brightness or white, and the color 0x0 is off or black. Starting in position 0x76, the image data is stored as 4-bit color pixels, with two pixels packed into each byte. Program 9.1 shows the purple data with 16 pixels per line in row-major. If the width of your image is not a multiple of 16 pixels, the BMP format will pad extra bytes into each row so the number of bytes per row is always divisible by 8. In this case, no padding is needed. The **`Nokia5110_PrintBMP()`** function will automatically ignore the padding.

Part c) The starter project includes a random number generate. A C version of this function can be found in the book as Programs 2.6, 2,7, and 3.12. To learn more about this simple method for creating random numbers, do a web search for **linear congruential multiplier**. The book example will seed the number with a constant; this means you get exactly the same random numbers each time you run the program. To make your game more random, you could seed the random number sequence using the SysTick counter that exists at the time the user first pushes a button (copy the value from NVIC_ST_CURRENT_R into the private variable M). You will need to extend this random number module to provide random numbers as needed for your game. For example, if you wish to generate a random number between 1 and 5, you could define this function

```
unsigned long Random5(void){
  return (Random()%5)+1;  // returns 1, 2, 3, 4, or 5
}
```

Seeding it with 1 will create the exact same sequence each execution. If you wish different results each time, seed it once after a button has been pressed for the first time, assuming SysTick is running

```
Seed(NVIC_ST_CURRENT_R);
```

Part d) When designing a complex system, it is important to design implement and test low-level modules first (parts a,b,c). In order to not be overwhelmed with the complexity, you must take two working subsystems and combine them. Do not combinate all subsystems at the same time. Rather, we should add them one at a time. Start with a simple approach and add features as you go along.

The Competition

Between at 2:00-4:00 on Wednesday you will demonstrate your games. Each game is peer evaluated by 7 teams who will rank them. Your rank determines your score on the lab as described by the percentile cutoffs in the slides.