Lab Manual
Embedded Systems

Table of Contents

Lab 2. Hello LaunchPad

Preparation

You will need a LaunchPad and access to TM4C123_LaunchPadUsersManual.pdf

Book Reading Chapters 1 and 2 in Embedded Systems: Introduction to ARM Cortex-M Microcontrollers

Starter project Lab2_HelloLaunchPad

Purpose

When first learning a new programming language it is tradition to begin by running a program that outputs the message "Hello World". Later you will write your own programs, but in this lab you will run simply a program that we have written for you. The input and output on the microcontroller comes from physical devices like switches and LEDs. Consequently, our "Hello World" will ask you to push a switch and observe an LED. The purpose of this lab is to work through the process of configuring the development system for the microcontroller board, and to learn how we will be grading labs in this course.

System Requirements

The system will have one input and three outputs. The input is a switch called SW1, which is connected port pin PF4. Three outputs (PF3, PF2, PF1) are connected to one multi-color LED. The color of the LED is determined by the 3-bit value written to the outputs. If SW1 is not pressed the LED toggles blue-red, and if SW1 is pressed the LED toggles blue-green.

Procedure

The basic approach to all labs will be to first develop and debug your system using the simulator. You will get a lab grade for this simulation phase of development. After the software is debugged, you will interface actual components to the LaunchPad and run your software on the real microcontroller. You will get a second lab grade for this real-board phase of development. In this lab you will not need to write any software (just run the software we have provided) or build any hardware (just use the LaunchPad with the switches and LED already connected). There are three steps to the software installation and the details can be found at http://users.ece.utexas.edu/~valvano/edX/download.html 400M

Keil Installation) The first step is to install the Keil uVision integrated development environment (IDE) for the ARM. This IDE includes the editor, compiler, simulator, and debugger. The **MDK-Lite (32KB) Edition** does not require a serial number or license key. A slide show of the process can be found at

http://users.ece.utexas.edu/~valvano/edX/KeilInstall.html

TExaS Installation) The second step is install Test EXecute and Simulate (**TExaS**). TExaS is a set of enhancements to Keil that will serve as your helper and grader during this course. This installation will also setup the example projects and starter files for labs. A slide show of the process can be found at http://users.ece.utexas.edu/~valvano/edX/TExaSInstall.html 40M

Windows drivers installation) The third step will be to install the hardware device drivers for your LaunchPad. You will need the LaunchPad hardware to perform this step. If you do not have it yet or do not plan to buy the hardware kit, you can skip this step and perform the labs in simulation. Driver installation depends on which OS you have. Slide shows of the process can be found at

Windows XP

Windows Vista

Windows 7

Windows 8

Windows 8.1

Part a) In order to test the software development system is properly installed you will run an existing project called Lab 2. You should not need to edit any of the code. First you will run Lab 2 in simulation mode. I suggest you modify your Windows system to show file extensions.

1) Open the Lab2 project file, which is the file Lab2 of type **uvproj** in the Lab2 directory.
2) Verify it is configured to run in the simulator
3) Build the project
4) Start the debugger in simulation mode
5) Run the program and interact with the switches, notice the LED outputs

Part b) In order to test the software development system for the real board, you will run Lab 2 on the real board. Again, you will not need to edit any of the code. You will compile, download, and grade the Lab 2 project on the actual LaunchPad. To run the grader on the real board perform these steps

0) Open the Lab2 project file, which is the file Project.uvproj in the Lab2 directory.
1) In Keil, execute **Project->OptionsForTarget**. In the **Target** tab, select **TExaS** in the **Operating System** drop-down menu. In the **Debug** tab, click the Use radio on the right and select the **Stellaris ICDI**.
2) Compile the project by executing **Project->BuildProject**
3) Download the object code into Flash EEPROM by executing **Flash->Download**
4) Start the debugger by executing **Debug->Start/StopDebuggingSession**
5) Show the TExaS real board grader by executing **Debug->OSsupport->TExaSGrader2.0**
6) Run your program executing by **Debug->Run**
7) Invoke the grader by clicking the **Grade** button (follow directions in the ActionMsg Window)

Interesting questions (things to think about but NOT implement in your lab)

How you could make the LED flash slower?

How you could make the LED flash blue-yellow instead of blue-red?

Lab 5. Functions in C

Preparation

You will need a LaunchPad and access to TM4C123_LaunchPadUsersManual.pdf.

Book Reading Sections 1.8, 1.9, 2.8, 5.1, 5.3 in Volume 1 Embedded Systems

Starter project Lab5_FunctionsInC

Purpose

In Lab5 you will learn how to write software that involves functions, parameters, and if-then conditionals.

System Requirements

The Lab5 starter project is the similar to **C5_Room** example in that it takes input from the keyboard using **`scanf`** and performs output to the display using **`printf`**. It also includes connections to the Lab5 grader. The goal is for you to write a function that accepts two input parameters (length and width) and calculates the area of rectangular room with this length and width. The length and width are in meters and the returned area should be in square meters. Calculate the area only if both the length and width are between 3 and 20 inclusively. Return a result of zero if the length is less than 3, the width is less than 3, the length is greater than 20 or the width is greater than 20. You are asked to write the function **`Calc_Area.`**.

Procedure

The basic approach to all labs will be to first develop and debug your system using the simulator. You will get a lab grade for this simulation phase of development. After the software is debugged, you will interface actual components to the LaunchPad and run your software on the real microcontroller. You will get a second lab grade for this real-board phase of development.

Part a) The first step is to write the function **`Calc_Area.`** Next you can test your function in simulation mode. This main program inputs using **`scanf`**, calls your function and outputs the results to the UART window using **`printf`**.

```
int main (void) {
  unsigned long length, width, area;
  UART_Init();    // initialize UART for printing
  printf("\nThis program calculates areas of rectangular rooms\n");
  while(1) {
    printf("\nGive length: "); scanf("%ld", &length);  // Get input
    printf("\nGive width: ");  scanf("%ld", &width);   // Get input
    area = Calc_Area(length, width);
    printf("\nArea of the room is %ld\n",  area);
  }
}
```

Part b) The second step is to run the grader in simulation mode. The automatic grader will interact with your function using the above main program (so while grading do not change this main program).

Part c) To run on the real board you will need to start the TExaSdisplay application. This application allows you to interact with the scanf and printf operations occurring on the microcontroller. The default settings (search for COM port and 115200 bits/sec) should work. If your computer has only one COM port device, then the default values for the settings will search for the current COM port to open and you can skip to step 2.

Step 1) If your computer has multiple COM port devices, then open the device Manager and make note of the COM port for the Stellaris Virtual Serial Port (in this figure it is COM13).

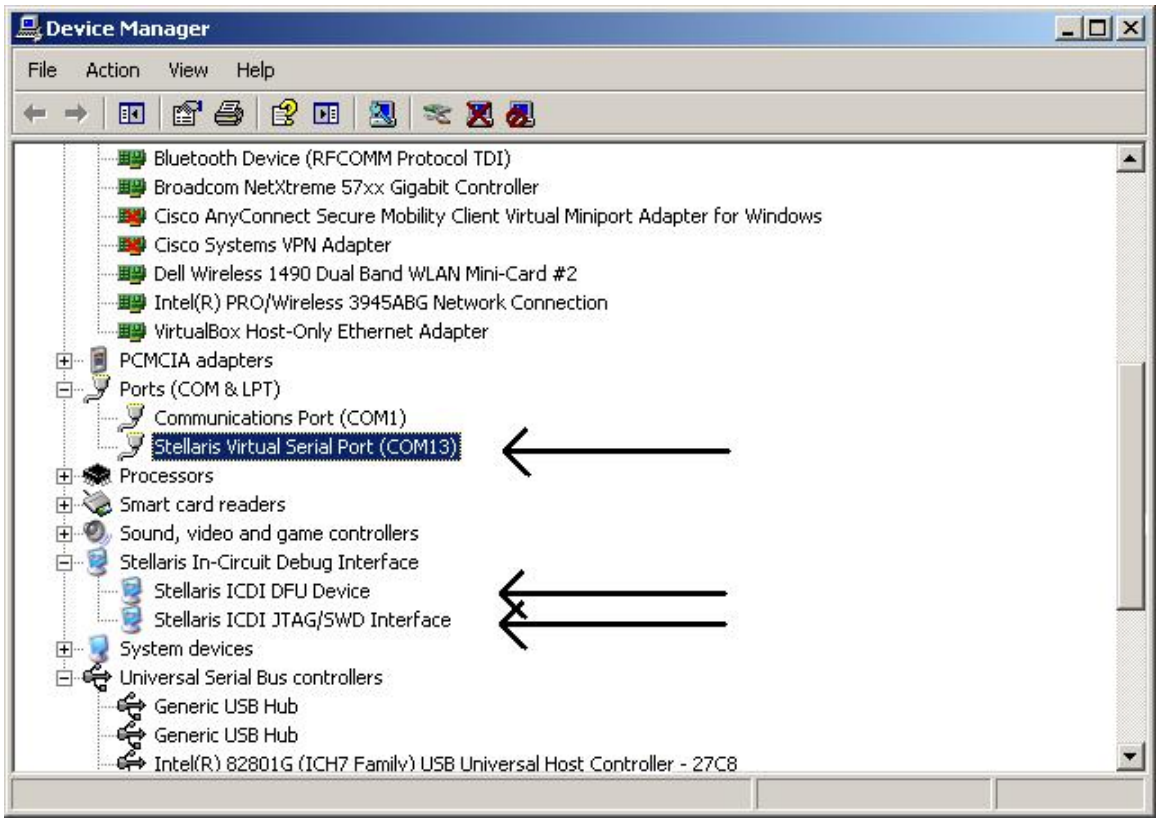


Figure 5.1. Device manager showing a computer with multiple COM port devices. On this computer the Stellaris Virtual Serial Port is on COM13.

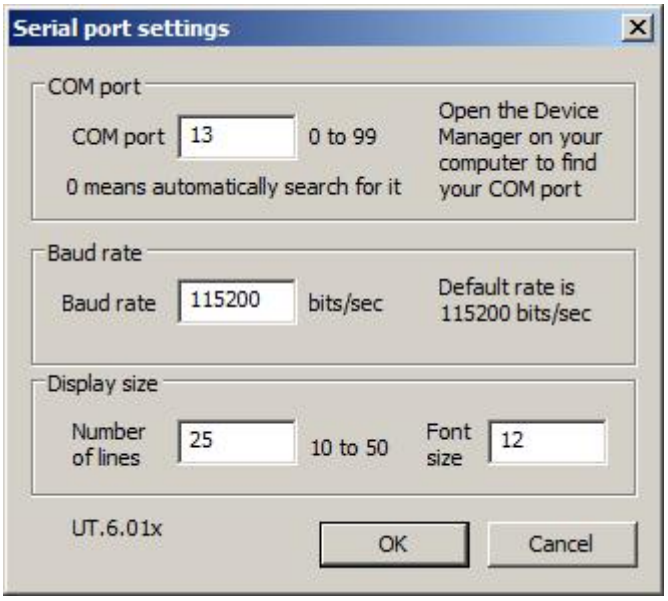


Figure 5.2. To configure TExaSdisplay to communicate with the proper COM port, execute COM->Settings and specify the COM port number (in this case 13) .

Step 2) To connect TExaSdisplay to the microcontroller, execute COM->OpenPort. You can have both the Keil debugger and the TExaSdisplay application open while debugging your combined hardware/software system on the actual LaunchPad.

Interesting questions to think about (things to think about but NOT implement in your lab)

The automatic grader performs hundreds of tests of your program. In what way is the automatic grader superior to manual testing using **`scanf`** and **`printf`**?

Lab 6. Branching, Functions, and Time Delays

Preparation
You will need a LaunchPad and access to TM4C123_LaunchPadUsersManual.pdf

Book Reading Sections 2.6, 3.3, 3.4, 4.1 and 4.2 in Volume 1 Embedded Systems

Starter project
Lab6_BranchingFunctionsDelays

Purpose
The purpose of this lab is to learn simple programming structures in C. You will also learn how to estimate how long it takes to run software, and use this estimation to create a time delay function. You learn how to use the oscilloscope to measure time delay. C software skills you will learn include masking, toggling, if-then, subroutines, and looping.

System Requirements
The system has one input switch and one output LED. Figure 6.1 shows the system when simulated as the switch is touched. A negative logic switch means the PF4 signal will be 1 (high, 3.3V) if the switch is not pressed, and the PF4 signal will be 0 (low, +0V) if the switch is pressed. A positive logic blue LED interface means if the software outputs a 1 to PF2 (high, +3.3V) the LED will turn ON, and if the software outputs a 0 to PF2 (low, 0V) the blue LED will be OFF. Here in Lab 6, you first debug in simulation and then run on the real board, but no external hardware will be required. The switch and LED are already built into the LaunchPad. However in Lab 8, you will attach a real switch and a real LED to your protoboard and interface them to your microcontroller. The functionality of this system is described in the following rules.

1) Make PF2 an output and make PF4 an input (enable PUR for PF4).
2) The system starts with the LED ON (make PF2 =1).
3) Delay for about 100 ms
4) If the switch is pressed (PF4 is 0), then toggle the LED once, else turn the LED ON.
5) Repeat steps 3 and 4 over and over.

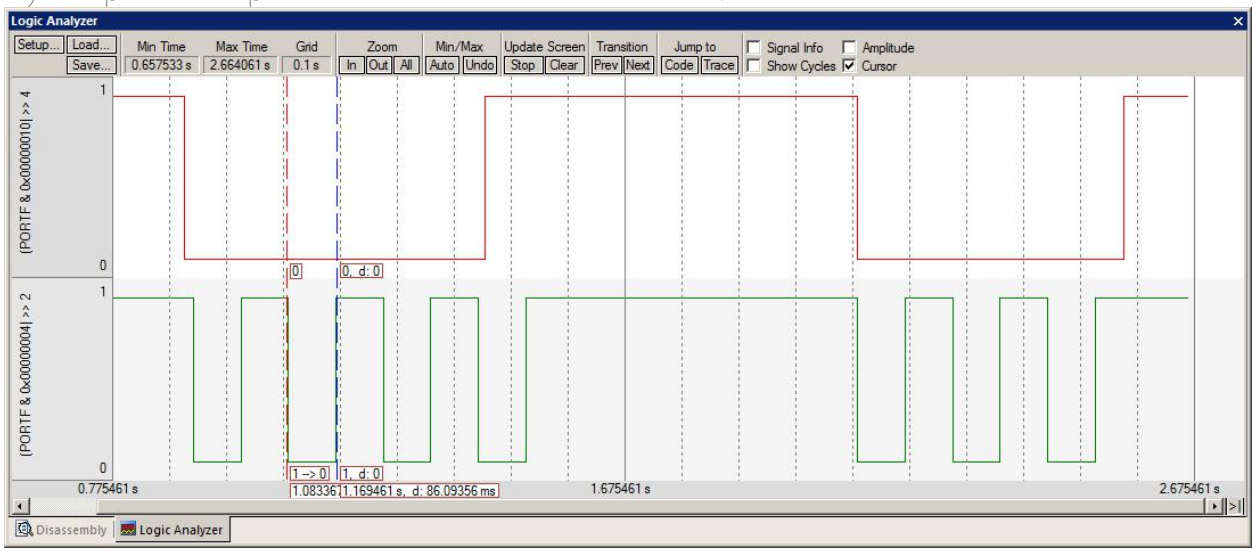


Figure 6.1. Example screenshot in simulation mode. Notice the output toggles at about 86ms when switch is pressed (PF4=0) and the output is high when the switch is not pressed. (the 86ms time in the simulator will result in 100 ms on the real board, because of a bug in the Keil uVision).

The basic approach to Lab 6 will be to develop and debug your system using the simulated negative logic switch (PF4) and positive logic LED (PF2). In Lab 8, you will build and test an external switch and LED, which you will need to connect yourself. However, this lab will run on the LaunchPad using SW1 on PF4 and the blue LED on PF2, which come pre-built into the LaunchPad.

To run the Lab 6 grader simulation verify the two settings in Figure 6.1. In particular, execute Project->Options and select the Debug tab. The "Use Simulator" option must be selected. Second, notice the parameter field includes **–dTExaSLab6**.

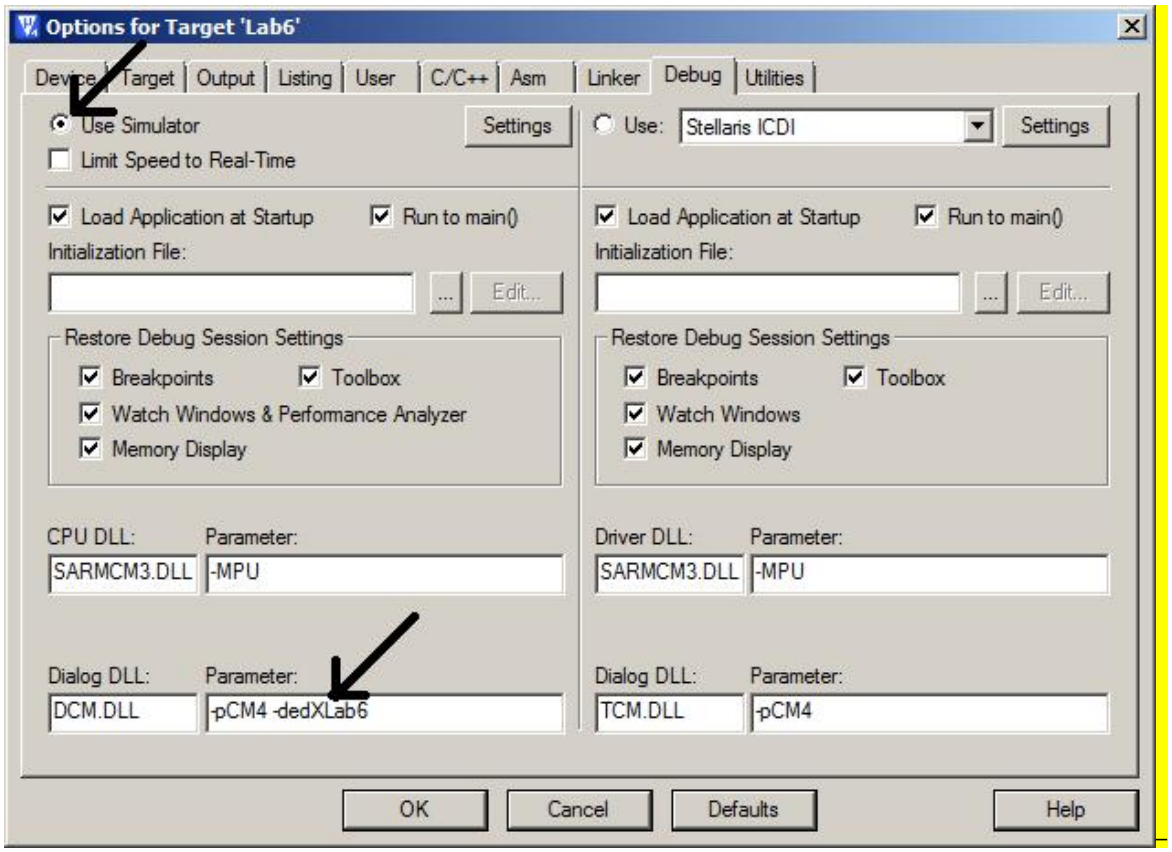


Figure 6.2. Configuration used to run Lab 6 in simulation.

To run the Lab 6 grader simulation verify the two settings in Figure 6.1. In particular, execute Project->Options and select the Debug tab. The "Use Simulator" option must be selected. Second, notice the parameter field includes **–dedXLab6**.

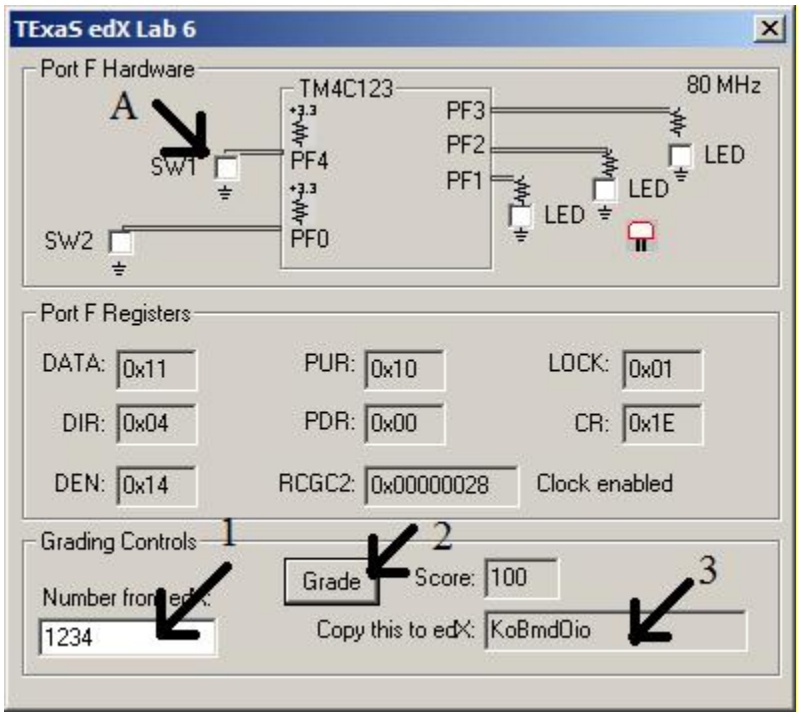


Figure 6.3. Using TExaS to debug your software in simulation mode. Press the switch at "A" to press and release the input SW1. When you are finished: 1) get the Number from edX, 2) hit the Grade button, and 3) Copy the code back into edX.

Time is very important in embedded systems. One of the simplest ways in which we manage time is determining how long it takes to run our software. There are two ways to determine how long each instruction takes to execute. The first method uses the ARM data sheet. For example, Figure 6.4 is a page from the Cortex-M4 Technical Reference Manual. E.g., see pages 34-38 of: **CortexM4_TRM_r0p1.pdf**

Operation	Description	Assembler	Cycles
Count	Leading zeroes	`CLZ Rd, Rn`	1
Load	Word	`LDR Rd, [Rn, <op2>]`	2^b
	To PC	`LDR PC, [Rn, <op2>]`	$2^b + P$
	Halfword	`LDRH Rd, [Rn, <op2>]`	2^b
	Byte	`LDRB Rd, [Rn, <op2>]`	2^b
	Signed halfword	`LDRSH Rd, [Rn, <op2>]`	2^b
	Signed byte	`LDRSB Rd, [Rn, <op2>]`	2^b

Figure 6.4. From the Technical Reference Manual page 34.

On the TM4C123 the default bus clock is 16 MHz ±1%. When using the automatic grader, we will activate the phase lock loop (PLL), and the bus clock will be exactly 80 MHz. The following is a portion of a listing file with a simple delay loop. The SUBS and BNE are executed 800 times. The SUBS takes 1 cycle and the BNE takes 1 to 4 (a branch takes 0 to 3 cycles to refill the pipeline). The minimum time to execute this code is 800*(1+1)/12.5 ns = 128 ns. The maximum time to execute this code is 800*(1+4)/12.5 ns = 320 ns. 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 PF2.

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

*(note: the **BNE** instruction executes in 3 cycles on the simulator, but in 4 cycles on the real board)*

The following function can be used to delay. The C is on the left and the corresponding assembly is on the right. The number 1333333 assumes 6 cycles per loop (100ms/12.5ns/6). The Keil optimization is set at Level 0 (-O0) and the "Optimize for Time" mode is unchecked.

```
void Delay1ms(unsigned long time){
  unsigned long i;
  while(time > 0){
    i = 1333333;
    while(i > 0){
      i = i - 1;
    }
    time = time - 1;
  }
}
```

```
0x000003A8 E005         B     test1
0x000003AA 4904 loop1   LDR   r1,=1333333
0x000003AC E000         B     test2
0x000003AE 1E49 loop2   SUBS  r1,r1,#1   ;1 cycle
0x000003B0 2900 test2   CMP   r1,#0x00   ;1 cycle
0x000003B2 D1FC         BNE   loop2      ;4 cycles
0x000003B4 1E40         SUBS  r0,r0,#1
0x000003B6 2800 test1   CMP   r0,#0x00
0x000003B8 D1F7         BNE   loop1
0x000003BA 4770         BX    lr
```

Program 6.1. A function you should use to delay time.

Part a) Write a main program in C that implements the input/output system. Pseudo code and flowchart are shown, illustrating the basic steps for the system. We recommend at this early stage of your design career you access the entire I/O port using GPIO_PORTF_DATA_R. After you fully understand how I/O works, then you can use bit-specific addressing to access I/O ports.

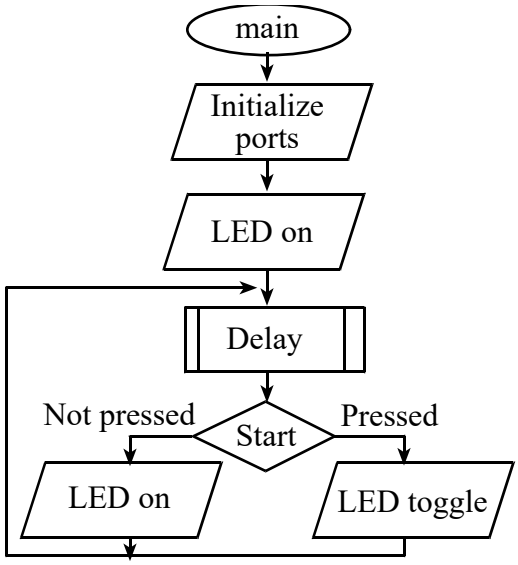


Figure 6.5. Flowchart for Lab6

main Turn on the clock for Port F
Clear the **PF4** and **PF2** bits in the Port F AMSEL register to disable analog
Clear the **PF4** and **PF2** bit fields in the Port F PCTL register to configure as GPIO
Set the Port F direction register so
PF4 is an input and
PF2 is an output
Clear the **PF4** and **PF2** bits in the Port F AFSEL register to disable alternate functions
Set the **PF4** and **PF2** bits in the Port F DEN register to enable digital
Set the **PF4** bit in the Port F PUR register so pull-up resistor will pull **PF4** up until switch touched
Set the **PF2** bit in the Port F DATA register so the LED is initially ON
loop Delay about 100 ms
Read the switch and test if the switch is pressed
If **PF4**=0 (the switch is pressed),
toggle **PF2** (flip bit from 0 to 1, or from 1 to 0)
If **PF4**=1 (the switch is not pressed),
set **PF2**, so LED is ON
Go to **loop**

Program 6.2. Lab 6 pseudo code.

Part b) Test the program in simulation mode. Use the built-in logic analyzer to verify the LED is toggling at the rate at which it was designed. In particular, capture two screenshots like Figure 6.1 showing when the switch is pressed, the LED is ON for 100 ms and OFF for 100 ms. The same code runs at a speed just a little bit slower on the real board as compared to running on the simulator. For this reason the graders will allow for some tolerance when measuring times of your lab solution. For example, a 100-ms delay running on the real board may look like a 86-ms delay running in simulation.

Simulators typically run slower than real hardware. Use the built in logic analyzer to measure how much Cortex-M time is simulated in 10 seconds. Hit the reset twice to clear the time axis on the plot. Run the simulator for 10 human seconds (real time with your watch), and then stop simulation. Obverse the logic analyzer time to determine the amount of Cortex-M simulated. For example, my computer simulated 15 sec of Cortex-M time in 10 sec of human time, meaning the simulator was running 150% faster than a real Cortex-M. There are many factors that affect this ratio, so do expect to see this ratio vary a lot. The point of the exercise is to get a sense of how a simulator manages time.

Part c) Load your software onto the board and test it again. If the PF4 switch is not pressed, then the blue LED is solidly on. If the PF4 switch is pressed, then the blue LED toggles 5 times a second. To run the grader on the real board perform these steps
1) In Keil, execute **Project->OptionsForTarget**. In the **Target** tab, select **TExaS** in the **Operating System** drop-down menu. In the **Debug** tab, click the Use radio on the right and select the **Stellaris ICDI**.
2) Compile the project by executing **Project->BuildProject**
3) Download the object code into Flash EEPROM by executing **Flash->Download**
4) Start the debugger by executing **Debug->Start/StopDebuggingSession**
5) Show the TExaS real board grader by executing **Debug->OSsupport->TExaSGrader2.0**
6) Run your program executing by **Debug->Run**
7) Please debug your software before attempting to grade. When you are ready to run the grader, press reset and then run. Invoke the grader by clicking the **Grade** button (follow directions in the ActionMsg Window)

Things to think about but NOT implement in your lab:
How you could make the LED flash slower?

How you could make the LED flash blue-yellow instead of blue?

Lab 7. Design and Development

Preparation
You will need a LaunchPad and access to TM4C123_LaunchPadUsersManual.pdf

Book Reading Sections 2.3, 2.4, 2.5, Chapter 5 in Volume 1 Embedded Systems

Starter project
Lab7_SOS

Purpose
In Lab7 you will learn how to write software that reads from two switches, makes decisions, and outputs to an LED. You will learn and understand the steps required to initialize parallel ports.

System Requirements
The Lab7 starter project is the same as **C7_SOS** example but includes the connections to the Lab7 grader. You will make three changes. First, you will modify the system so a yellow SOS is flashed instead of the green. Yellow is created by mixing red and green (PF3 high, PF2 low, and PF1 high). Second, make the SOS flash only if both switches SW1 and SW2 are pressed (this means both PF4 and PF0 inputs are 0). Third, you will decrease the time between SOS outputs from 5 to 4 seconds. In summary the system should perform these steps:

```
1) Make PF1, PF2, and PF3 outputs.
      Make PF0 and PF4 inputs (enable PUR for PF0 and PF4).
2) If either SW1 or SW2 are off, the LEDs should be off.
    If both SW1 and SW2 are on, the SOS is sent on the
yellow LED
     a) Send an 'S' as short short short pulses on the
yellow LED
     b) Send an 'O' as long long long pulses on the yellow
LED
     c) Send an 'S' as short short short pulses on the
yellow LED
     d) Wait 4 seconds
3) Repeat step 2 over and over.
```

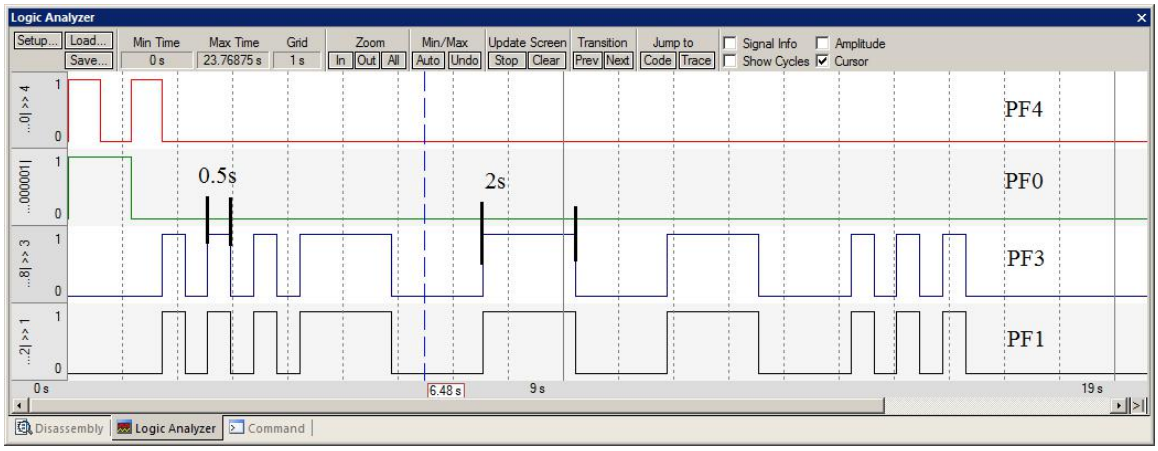


Figure 7.1. Example screenshot in simulation mode.

The basic approach to this lab will be to first develop and debug your system using the simulator. You will get a lab grade for this simulation phase of development. After the software is debugged, you will run your software on the real microcontroller. You will get a second lab grade for this real-board phase of development. In this lab you will not need to build any hardware (just use the LaunchPad with the switches and LED already connected).

To run the Lab 7 grader, you must do two things. First, execute Project->Options and select the Debug tab. The debug parameter field must include **–dedXLab7**. Second, the **edXLab7.dll** file must be added to your Keil\ARM\BIN folder. These configurations should have been made during the download and installation steps in Lab 2.

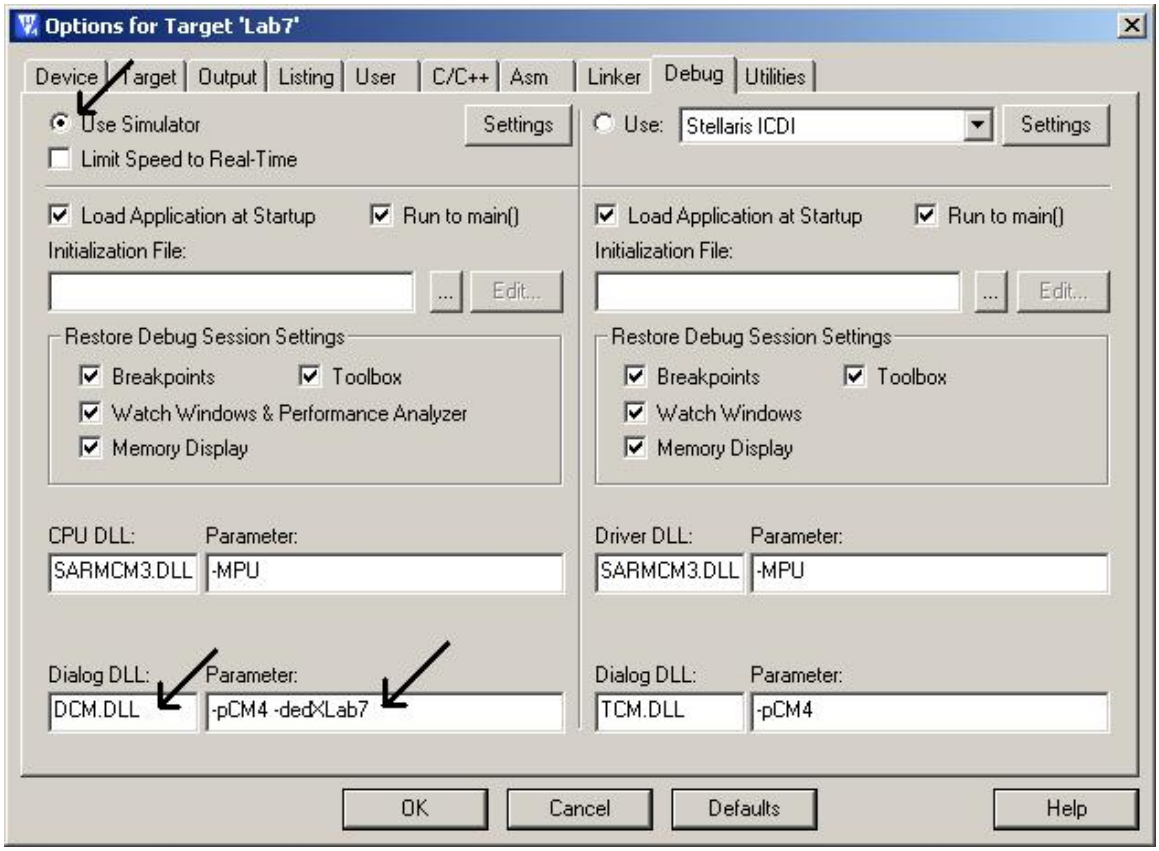


Figure 7.1. Setting up TExaS to debug your software in simulation mode.

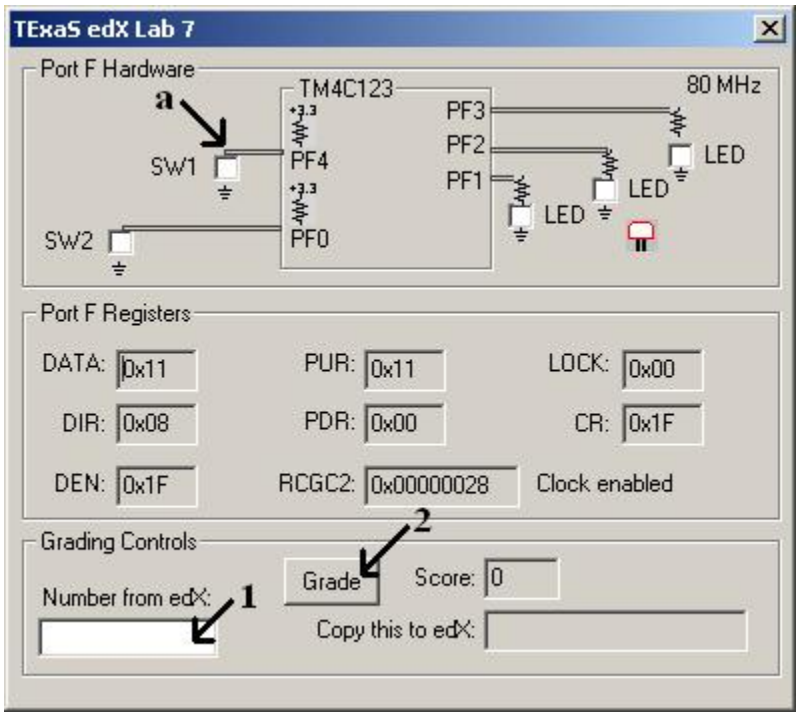


Figure 7.2. Debugging with TExaS in simulation mode. When running push the switch (a) to set and release the switch. When grading first place the number from edX (1), then click grade (2).

Part a) Make the necessary changes to the software given the fact that the yellow LED will be flashed. Test your software in the simulator and fix any software bugs.

Part b) Change the logic in the main loop so that the LED flashes SOS only if both switches are pressed.

Part c) Change the logic in the SOS output software so it delays 4 seconds between outputs, rather than 5 seconds.

Part d) During checkout, I will grade your system in both simulation and on the real board. During the simulation grading I will automatically set the input and check your output. The same code runs at a speed just a little bit slower on the real board as compared to running on the simulator. For this reason the graders will allow for some tolerance when measuring times of your lab solution. For example, a 4-second delay running on the real board may look like a 3.75 second delay running in simulation.

Part e) Load your software onto the board and test it again. While grading on the real board, your system will be tested as you push and release the two switches. To run the grader on the real board perform these steps

1) In Keil, execute **Project->OptionsForTarget**. In the **Target** tab, select **TExaS** in the **Operating System** drop-down menu. In the **Debug** tab, click the Use radio on the right and select the **Stellaris ICDI**.
2) Compile the project by executing **Project->BuildProject**
3) Download the object code into Flash EEPROM by executing **Flash->Download**
4) Start the debugger by executing **Debug->Start/StopDebuggingSession**
5) Show the TExaS real board grader by executing **Debug->OSsupport->TExaSGrader2.0**
6) Run your program executing by **Debug->Run**

7) Please debug your software before attempting to grade. When you are ready to run the grader, press reset and then run. Invoke the grader by clicking the **Grade** button (follow directions in the ActionMsg Window)

Things to think about but NOT implement in your lab:
There are many ways to test if two switches are pressed. For example, assume Port E is an input connected to two positive logic switches on PE3 and PE1, and we wish to execute SOS if switches are pressed. The first way involves an **&&**. The second method involves **&**, and the third way uses **==**. Why does the first use **&&** and the second use **&**? Which do you like? Does the use of the variables make it easier to debug?

```
In=GPIO_PORTE_DATA_R;
S1 = In&0x08;
S2 = In&0x02;
if(S1&&S2){
  FlashSOS();
}
```

```
In=GPIO_PORTE_DATA_R;
S1 = (In&0x08)>>2;
S2 = In&0x02;
if(S1&S2){
  FlashSOS();
}
```

```
In=GPIO_PORTE_DATA_R;
if((In&0x0A)==0x0A){
  FlashSOS();
}
```

Lab 8. Switch and LED Interface

Preparation You will need a LaunchPad, a switch, a 10kΩ resistor, an LED, and a 470Ω resistor.

Book Reading Sections 2.7, 4.2, 4.6, and 4.7 in Volume 1 Embedded Systems

Starter project Lab8_SwitchLEDinterface

Purpose

Lab 8 is our first lab requiring you to build circuits on the breadboard and connect them to the LaunchPad. 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.

System Requirements

In this lab you will build a switch interface that implements positive logic, and you will build an LED interface that implements positive logic. You will attach this switch and LED to your protoboard (the white piece with all the holes), and interface them to your TM4C123. Overall functionality of this system is similar to Lab 6, with five changes: 1) the pin to which we connect the switch is moved to PE0, 2) you will have to remove the PUR initialization because pull up is no longer needed. 3) the pin to which we connect the LED is moved to PE1, 4) the switch is changed from negative to positive logic, and 5) you should decrease the delay so it flashes about 5 Hz.

1) Make PE1 an output and make PE0 an input.
2) The system starts with the LED on (make PE1 =1).
3) Wait about 100 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

The basic approach to this lab will be to first develop and debug your system using the simulator. You will get a lab grade for this simulation phase of development. After the software is debugged, you will interface actual components to the LaunchPad and run your software on the real microcontroller. You will get a second lab grade for this real-board phase of development.

Part a) You should draw a circuit diagram connecting the switch and LED to the microcontroller, similar to Figure 8.1.

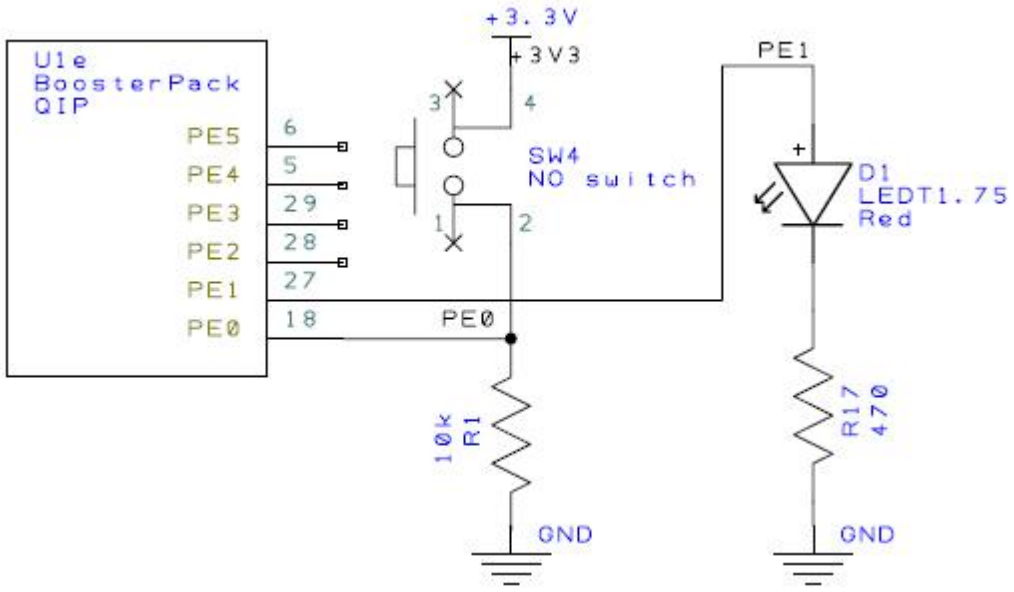


Figure 8.1. One possible Lab 8 interface connecting the input to PE0 and the output to PE1.

The lab describes using PE0 for input and PE1 for output, but Tables 8.1 and 8.2 show other options that the automatic grader can handle. You could connect the output to PA3, PB1 or PE1, and you could connect the input to PA2, PB0, and PE0.

Output	PA3	PB1	PE1

Table 8.1. Possible ports to interface the output (PE1 is default).

Input	PA2	PB0	PE0

Table 8.2. Possible ports to interface the input (PE0 is default).

Notice in Figure 8.1 that the PE0 and PE1 wires in the circuit diagram, but they are NOT electrically connected, because there is NO dot at the point of crossing. Conversely, the PE0 signal to the microcontroller, pin2 of the switch and one end of the 10K R1 resistor ARE electrically connected, because there IS a dot at the point of crossing.

Part b) Write the software that satisfies the requirements for this lab. The best approach to time delay will be to use a hardware timer, which we will learn in Lab 9. In this lab, however, we do not expect you to use the hardware timer. Again, use the logic analyzer on the simulator to verify the software operates properly.

Part c) After the software has been debugged on the simulator, you will build the hardware on the real board. 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 8.2. The long rows of holes along the outer sides of the protoboard are electrically connected. Some protoboards like the one in Figure 8.3 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, and one row will be +3.3V. 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 470 ohm resistor as shown in Figure 8.1. First, cut a 24 gauge solid wire long enough to reach from PE1 and pin 1 of the resistor. 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 holes in the 5-hole row shared by the resistor.

I like to watch the power LED when I first power up a new circuit. If the power LED does not illuminate, I quickly disconnect the USB cable.

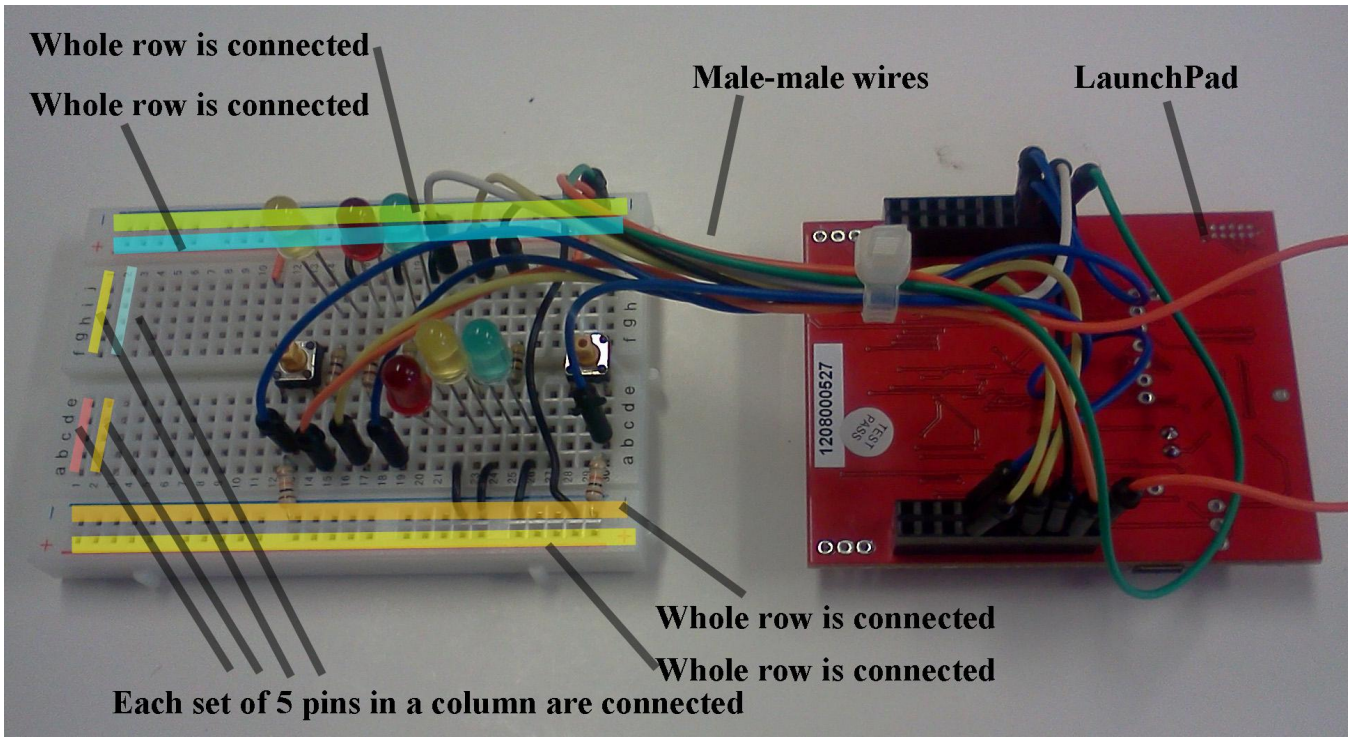


Figure 8.2. All the pins on each of the four long rows are connected. The 5 pins in each short column are connected. Use male–male wires to connect signals on the LaunchPad to devices on the protoboard. Make sure ground wire

connected between the LaunchPad and your circuit. The +3.3V power can be wired from the LaunchPad to your circuit. I like to connect the two dark blue rows to ground, red rows to +3.3V.

Notice the switch has 4 pins in a rectangular shape, as shown in Figure 8.3. Each button is a single-pole single-throw normally-open switch. All four pins are connected to the switch. Pins 1 and 2 are connected inside the switch; similarly pins 3 and 4 are connected. The switch itself is positioned between pins 1-2 and 3-4.

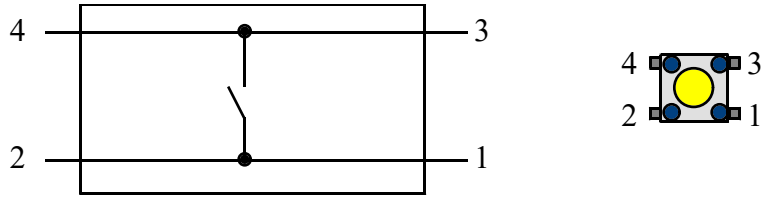


Figure 8.3. Connection diagram for the normally open B3F–style switch.

Do not place or remove wires on the protoboard while the power is on.

The next step is to build the LED output circuit. Using the data sheet, hold an LED and identify which pin is the anode and which is the cathode. LEDs emit light when an electric current passes through them, as shown in Figure 8.4. 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. Furthermore, LEDs will not be damaged if you plug it in backwards. LEDs however won't work plugged in backwards of course.

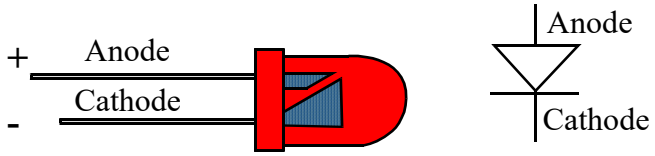


Figure 8.4. A drawing and the circuit symbol for an LED.
"Big voltage is on the big wire."

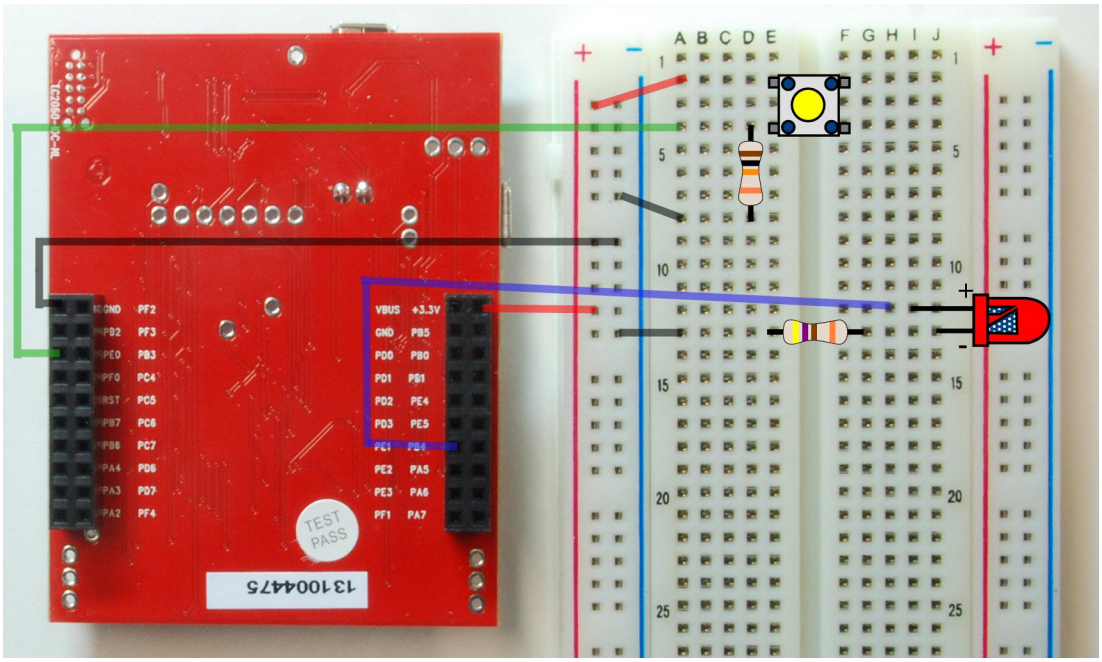


Figure 8.5. A photo showing one possible layout of the circuit. Black wires are ground. Red wires are +3.3V. Purple wire is PE1. Green wire is PE0. Brown–black–orange resistor is 10kΩ. Yellow–purple–brown resistor is 470Ω.

Part d) Debug your combined hardware/software system on the actual TM4C123 board. First, we will use the debugger to observe the input pin to verify the proper operation of the switch interface. You will have to single step through your code that initializes Port E, and PE0. You then execute the **Peripherals->SystemViewer->GPIO->GPIOE** command. As you single step you should see the actual input as controlled by the switch you have interfaced, see Figure 8.1.

Next, we will debug the output and the LED. PD3 is a debugging probe. With the power off connect a wire between PD3 and the LED cathode (-) (which is also connected to one end of the 470 resistor). Power it back up, start the debugger, run your program and do not press the switch. The LED should be on. We can verify all aspects of the LED interface are proper by beginning in the **Peripherals->xx** debugger window. In the debugging window you should see bit 1 of Port E is 1. In the Grader window (with your program running) you can observe the voltage seen at the PD3-probe. You can calculate the current across the LED using Ohm's Law. The current in the resistor is

$I = V/470\Omega$

During the real board grading you will have to push/release the external switch as instructed by the grader.

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

Interesting questions (things to think about but NOT implement in your lab)

How would you make the LED brighter?

What would happen if you plugged the LED in backward?

Lab 9. Functional Debugging

Preparation

Run the Lab9 starter file in the simulator and on the real board

Book Reading Sections 6.1, 6.2, and 6.3 in Volume 1 Embedded Systems

Starter project Lab9_FunctionalDebugging

Purpose

In Lab09 you will learn about time delays, arrays and functional debugging. The data you collect will be physical proof that the system operates within specifications.

System Requirements

The Lab9 starter project is the same as **C9_Debugging** example but includes the connections to the Lab9 grader. You will make three changes. First, make the LED flash at 10 Hz. In other words, make it turn on for 0.05 seconds, and then turn off for 0.05 seconds. Second, make the LED flash if either switch SW1 or SW2 are pressed (this means either PF4 or PF0 is 0). Third, record PortF bits 4,1,0 every time the input changes or the output changes. For example, if your system detects a change in either PF4 or PF0 input, record PortF bits 4,1,0. If your system causes a change in PF1, record PortF bits 4,1,0. In order for the grading engine to see/grade your data, please leave the debugging array defined exactly as it is in the starter project

```
unsigned long Data[50];
```

Your system will be graded on its ability to satisfy the following requirements.

If both PF4 and PF0 switch are not pressed, the PF1 output should be low.
If either PF4 or PF0 switches is pressed, the output toggles at 10 Hz (±10%).
Information collected in the Data array matches the I/O on PortF.
50 data points are collected only on a change in input or a change in output.
(i.e., no adjacent elements in the array are equal).

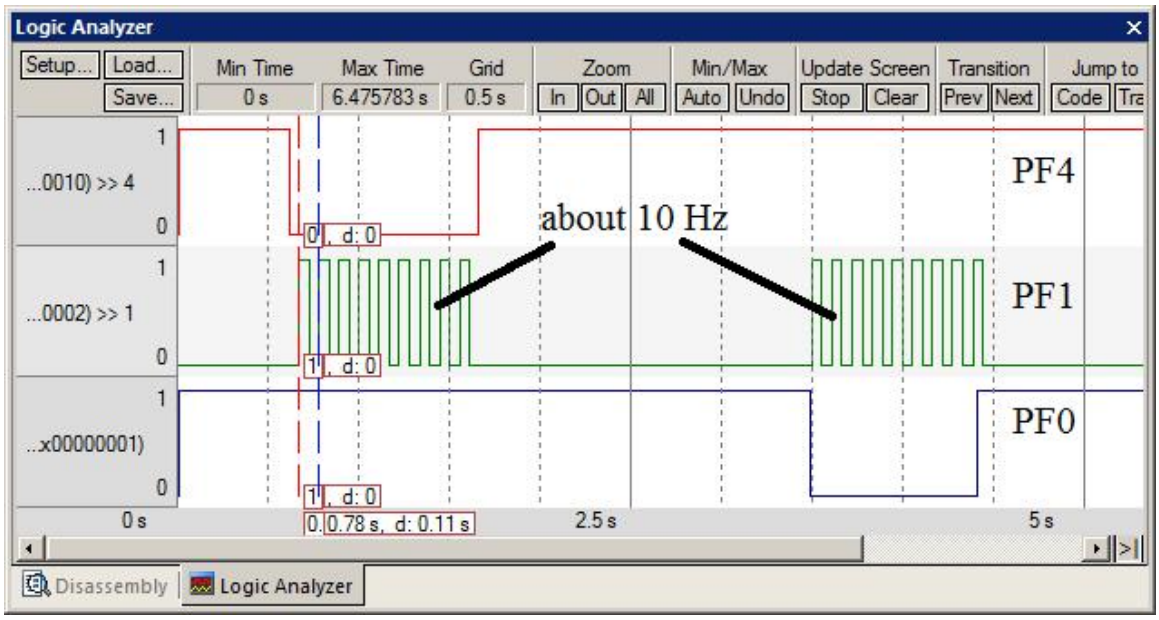


Figure 9.1. Logic analyzer output showing PF1 toggles at 10 Hz whenever PF0 or PF4 is low. If both PF0 and PF4 are high, then the output PF1 should be low.

Procedure

Part a) Run the starter code in both the simulator and on the real board. You should be able to repeat the measurements as demonstrated in the teaching videos of this chapter.

Part b) First, change the oscillation rate to 10 Hz, and use the existing debugging dump to prove it is toggling 0.05 seconds on and 0.05 seconds off. You should run this modification both in simulation and on the real board. You will notice that simply dividing the wait counter in half does not result in an exact solution. This is because each time through the loop includes waiting, outputting, and dumping. Dividing the wait counter in half just divides the waiting, without changing the time to output and dump. You only have to make it operate within 10% of 10 Hz. The simulator is not perfect so we will accept ±25% time accuracy in simulation, but will require ±1% time accuracy on the real board.

Part c) Next, add the ability to read the input and toggle the output only when either switch is pressed. The automatic grader in simulation mode should give you some points for meeting the input/output/time specifications. You may have to adjust your delay function so the grader sees it with ±25% of the expected 10 Hz.

Part d) Lastly, add the debugging instrument that dumps the input/output information into the **`Data[]`** array. You should mask the Port F data with 0x13 to select just bits 4, 1, and 0.

Grading and uploading the score

During checkout, I will grade your system in both simulation and on the real board. During the simulation grading I will automatically set the inputs, check your outputs and check that the **`Data[]`** array is properly filled.

While grading on the real board, your system will be tested in its ability to collect debugging data into this array as you push one switch and release, push the other switch and release, push the first switch, push the second switch, release the first switch and then release the second switch. You must then continue to push switches until the entire array is filled with debugging information. The grader will then test your **`Data[]`** array to verify correct data has been recorded.

Interesting questions (things to think about but NOT implement in your lab)

We removed the time measurements for this lab because SysTick running at 80 MHz could only measure time differences up to 200 ms. How could we have developed a method to record time with a longer range (e.g., minutes, hours, days, years)?

If we had two LaunchPads, how could we have connected them together so one computer tests the other?

Which was harder, writing the original system, or developing the debugging code that verifies the original system operates within specifications?

Lab 10. Traffic Light Controller

Preparation

Run the Lab10 starter file in the simulator and on the real board just to make sure the configurations are correct. Download the data sheet for the LED /static/HLMP-4700.pdf

Book Reading Sections 6.4 to 6.9, and 8.7 in Volume 1 Embedded Systems

Starter project Lab10_TrafficLight

Purpose

This lab has these major objectives: 1) the understanding and implementing of indexed 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. Please read the entire lab before starting.

System Requirements

Consider a 4-corner intersection as shown in Figure 10.1. There are two one-way streets are labeled South (cars travel South) and West (cars travel West). There are three inputs to your LaunchPad, two are car sensors, and one is a pedestrian sensor. The *South* car sensor will be true (3.3V) if one or more cars are near the intersection on the South road. Similarly, the *West* car sensor will be true (3.3V) if one or more cars are near the intersection on the West road. The *Walk* sensor will be true (3.3V) if a pedestrian is present and he or she wishes to cross in any direction. This walk sensor is different from a walk button on most real intersections. This means when you are testing the system, you must push and hold the walk sensor until the FSM recognizes the presence of the pedestrian. Similarly, you will have to push and hold the car sensor until the FSM recognizes the presence of the car. In this simple system, if the walk sensor is +3.3V, there is pedestrian to service, and if the walk sensor is 0V, there are no people who wish to walk. In a similar fashion, when a car sensor is 0V, it means no cars are waiting to enter the intersection. You will interface 6 LEDs that represent the two Red-Yellow-Green traffic lights, and you will use the PF3 green LED for the "walk" light and the PF1 red LED for the "don't walk" light. When the "walk" condition is signified, pedestrians are allowed to cross. When the "don't walk" light flashes (and the two traffic signals are red), pedestrians should hurry up and finish crossing. When the "don't walk" condition is on steady, pedestrians should not enter the intersection.

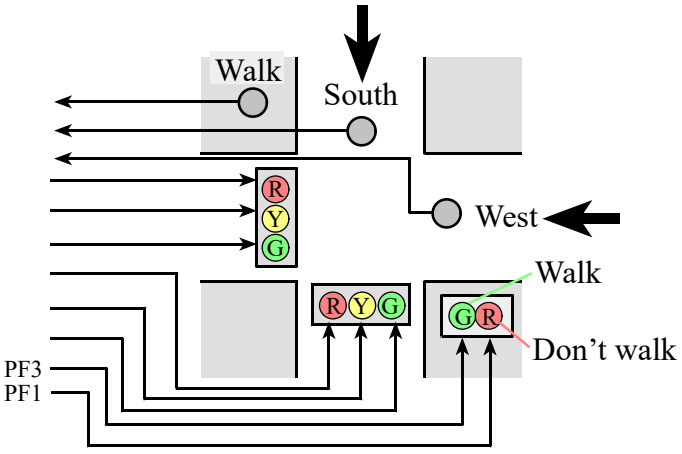


Figure 10.1. Traffic Light Intersection.

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 grading engine can complete the testing in a reasonable amount of time. Each traffic light pattern must be on for at least ½ second but for at most 5 seconds. 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 LED, and 3) "don't walk". You may assume the three sensors remain active for as long as service is required. The "hurry up" flashing should occur at least twice but at most 4 times.

The automatic grader can only check for function (does a pattern of inputs, result in the correct outputs.) In particular, the grader performs these checks:

0) At all times, there should be exactly one of the {red, yellow, green} traffic lights active on the south road. At all times, there should be exactly one of the {red, yellow, green} traffic lights active on the west road. To switch a light from green to red it must be yellow for at least ½ sec. The "walk" and "don't walk" lights should never both be on at the same time.

1) Do not allow cars to crash into each other. This means there can never be a green or yellow on one road at the same time as a green or yellow on the other road. Engineers do not want people to get hurt.

2) Do not allow pedestrians to walk while any cars are allowed to go. This means there can never be a green or yellow on either road at the same time as a "walk" light. Furthermore, there can never be a green or yellow on either road at the same time as the "don't walk" light is flashing. If a green light is active on one of the roads, the "don't walk" should be solid red. Engineers do not want people to get hurt.

3) If just the south sensor is active (no walk and no west sensor), the lights should adjust so the south has a green light within 5 seconds (I know this value is unrealistically short, but it makes the grading faster). The south light should stay green for as long as just the south sensor is active.

4) If just the west sensor is active (no walk and no south sensor), the lights should adjust so the west has a green light within 5 seconds. The west light should stay green for as long as just the west sensor is active.

5) If just the walk sensor is active (no west and no south sensor), the lights should adjust so the "walk" light is green within 5 seconds. The "walk" light should stay green for as long as just the walk sensor is active.

6) If all three sensors are active, the lights should go into a pattern such within one 20-second period that the west light is green for at least 1 second, the south light is green for at least 1 second and the "walk" light is green for at least 1 second.

The grading engine can only check for function, not for the quality of your software. This section describes, in qualitative manner, what we think is good design. There is no single, "best" way to implement your system. A "good" solution will have about 9 to 30 states in the finite state machine, and provides for input dependence. Try not to focus on the civil engineering issues. I.e., first build a quality computer engineering solution that is easy to understand and easy to change, and then adjust the state graph so it passes the functional tests of the automatic grader. Because we have three inputs, there will be 8 next state links. 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 two FSM graphs in Figure 10.2. Drawing two arrows labeled **01** and **11** is the same as drawing one arrow with the label **X1**. When we implement the data structure, however, we will expand the shorthand and explicitly list all possible next states.

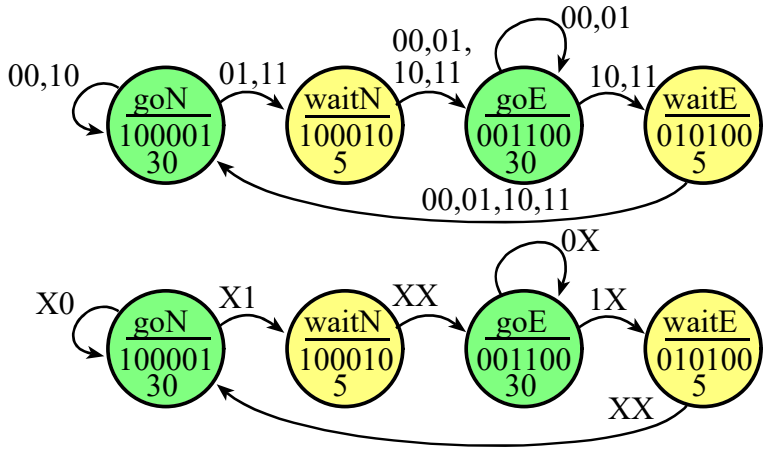


Figure 10.2. FSM from Chapter 10 redrawn with a short hand format.

The following are some qualitative requirements, which we think are important, but for which the automatic grader may or may not be able to evaluate.

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 cannot solve the problem by simply cycling through all the states regardless of the input.

1) Because we think being in a state is defined by the output pattern, we think you should implement a Moore and not a Mealy machine. However, your scheme should use a linked data structure stored in ROM.

2) There should be a 1-1 mapping between FSM 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 links (one for each input). The data structure has exactly these components: a name, an output, a time to wait, and 8 next state pointers (one for each input). There is no more or no less information in the data structure then the information in the state graph.

3) There can be no conditional branches in program, other than the **`while`** in **`SysTick_Wait`** and the **`for`** in **`SysTick_Wait10ms`**. This will simplify debugging make the FSM engine trivial.

4) The state graph defines exactly what the system does in a clear and unambiguous fashion. In other words, do not embed functionality (e.g., flash 3 times) into the software that is not explicitly defined in the state graph.

5) Each state has the same format of each state. This means every state has exact one name, one 8-bit output (could be stored as one or two fields in the struct), one time to wait, and 8 next indices.

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

7) There should be 9 to 30 states with a Moore finite state machine. If your machine has more than 30 states, you have made it more complicated than we had in mind. Usually students with less than 9 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.

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 should 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.

0) How long should I wait in each state? *Possible answer*: traffic lights at 1 to 2 seconds of real people time. *Flashing "don't walk"* on for ½ sec, off for a ½ sec and repeat 3 times.

1) What happens if I push 2 or 3 buttons at a time? *Required operation*: cycle through the requests servicing them in a round robin fashion.

2) What if I push the walk button, but release it before the light turns to walk? *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? *Required operation*: stop all cars and let people walk across either or both streets. A green (or yellow) light in any direction while the "walk" light is on will cause the automatic grader to penalize you for failing check #2. The pedestrian sensor does not know which street the pedestrian(s) want to cross, so you must direct all cars to stop while pedestrians may be in the road. You are not allowed to add additional pedestrian sensor because the automatic grader is built to handle only the configuration shown in Figure 10.13.

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.

As with all graders, it begins by checking initialization registers. During the I/O portion of the grading, we get a notification whenever you write to either of the output ports. The grader checks for valid output pattern sequences. We have defined 9valid output patterns, listed below. For each valid output pattern, there are a only finite number of valid output patterns that could be next.

Pattern 1) All lights are red. Once the output is at this pattern, the valid next patterns are {1,2,4,6,7,8,9}
Pattern 2) West is green, south is red, don't walk is

red. Once the output is at this pattern, the valid next patterns are {2,3}
Pattern 3) West is yellow, south is red, don't walk is red. Once the output is at this pattern, the valid next patterns are {1,3,4,6}
Pattern 4) South is green, west is red, don't walk is red. Once the output is at this pattern, the valid next patterns are {4,5}
Pattern 5) South is yellow, west is red, don't walk is red. Once the output is at this pattern, the valid next patterns are {1,2,5,6}
Pattern 6) Walk is green, south is red, west is red. Once the output is at this pattern, the valid next patterns are {1,6,7}
Pattern 7) Don't Walk is off, south is red, west is red. Once the output is at this pattern, the valid next patterns are {1,2,4,6,7,8,9}
Pattern 8) Don't Walk is off, west is red, south is green. Once the output is at this pattern, the valid next patterns are {4,5}
Pattern 9) Don't Walk is off, west is green, south is red. Once the output is at this pattern, the valid next patterns are {2,3}

There are a couple of consequences of this grading algorithm:

1) You should output to the road lights first and then to the walk lights,
2) You should output to the walk and don't walk lights at the same time
3) For simulation, we do not check for timing, so make the delays short during simulation testing.

Procedure

If you are using PD0, PD1, PB7, PB6, PB1 or PB0, make sure R9, R10, R25, and R29 are removed from your LaunchPad as shown in Figure 10.3. R25 and R29 are not on the older LM4F120 LaunchPads, just the new TM4C123 LaunchPads. The TM4C123 LaunchPads I bought over summer 2013 did not have R25 and R29 soldered on, so I just had to remove R9 and R10.

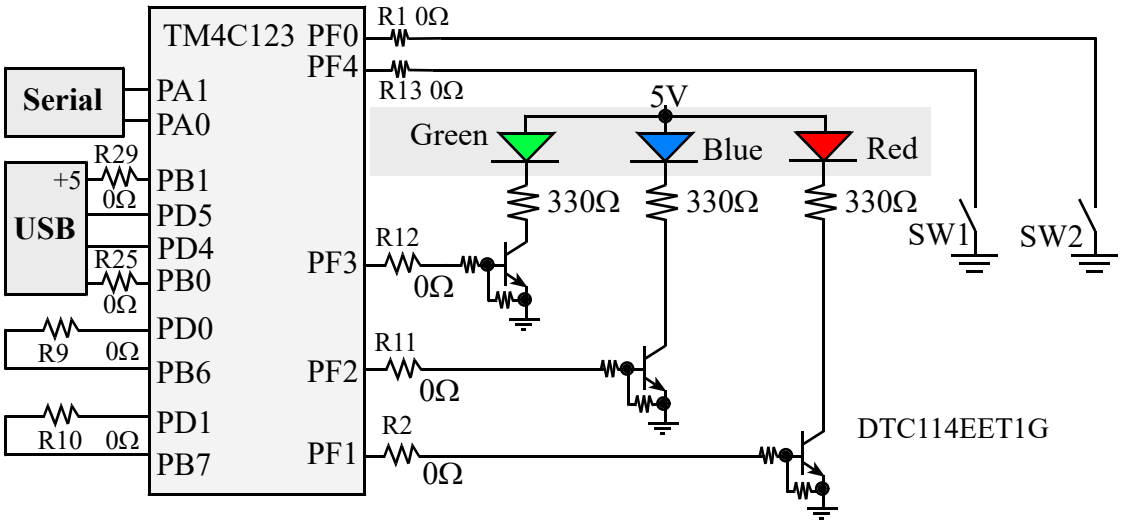


Figure 10.3. Switch and LED interfaces on the Stellaris® LaunchPad Evaluation Board. The zero ohm resistors can be removed so the corresponding pin can be used for its regular purpose.

Most microcontrollers have a rich set of timer functions. For this lab, you will the PLL and SysTick timer to wait a prescribed amount of time.

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 LaunchPad and run your software on the real microcontroller. 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 grading engine to observe during demonstration.

Part a) Decide which port pins you will use for the inputs and outputs. Avoid the pins with hardware already connected. Run the starter code in the simulator to see which ports are available for the lights and switches; these choices are listed in Tables 10.1 and 10.2. The "don't walk" and "walk" lights must be PF1 and PF3 respectively, but where to attach the others have some flexibility. In particular, Table 10.1 shows you three possibilities for how you can connect the six LEDs that form the traffic lights. Table 10.2 shows you three possibilities for how you can connect the three positive logic switches that constitute the input sensors. Obviously, you will not connect both inputs and outputs to the same pin.

Red east/west	PA7	PB5	PE5
Yellow east/west	PA6	PB4	PE4
Green east/west	PA5	PB3	PE3
Red north/south	PA4	PB2	PE2

Yellow north/south	PA3	PB1	PE1
Green north/south	PA2	PB0	PE0

Table 10.1. Possible ports to interface the traffic lights (yellow is suggested or default).

Walk sensor	PA4	PB2	PE2
North/south sensor	PA3	PB1	PE1
East/west sensor	PA2	PB0	PE0

Table 10.2. Possible ports to interface the sensors (yellow is suggested or default).

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

Part c) Write and debug the C code that implements the traffic light control system. During the debugging phase with the simulator, use the logic analyzer to visualize the input/output behavior of your system. Another debugging technique is to dump {the state index, the input, the output} each time through the FSM controller to create a log of the operation

During the simulation grading I will automatically set the inputs and check your outputs.

Hint: I recommend reducing the wait times for all states to be less than a second, so the simulation grading runs faster. Once you get a 100 in simulation, you can increase the wait times for the real board to be more reasonable.

Part d) 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. 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 and test the switch circuits. You should also use the debugger to observe the input pin to verify the proper operation of the interface similar to Lab 8.

The next step is to build the six LED output circuits. Build the system physically in a shape that matches a traffic intersection. You will use the PF3-2-1 LED interface for the walk light (green for walk and red for don't walk). Write a simple main program to test the LED interface, similar to the way you tested Lab 8.

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

During the real board grading you will have to push the sensors so that the four cases are tested (just west, just south, just walk, and all three).

What Real TExaS is looking for in Lab 10.

1) "Checking the TExaS_Init() in parameter" This verifies that the first parameter to TExaS_Init() is one of the enumerated types specified in TExaS.h with the expected number value. This first parameter tells TExaS which port you are using for inputs. If you are using the correct and unmodified TExaS.h for this lab, and if your program compiled, then this test

will always pass.
2) "Checking the TExaS_Init() out parameter" This verifies that the second parameter to TExaS_Init() is one of the enumerated types specified in TExaS.h with the expected number value. This second parameter tells TExaS which port you are using for outputs. If you are using the correct and unmodified TExaS.h for this lab, and if your program compiled, then this test will always pass.
3) "Checking the SysTick control register" SysTick must get its clock from the system clock; SysTick interrupts must be disabled; SysTick must be enabled.
4) "Checking the input port clock" The GPIO module specified by the first parameter to TExaS_Init() must have its clock enabled in the legacy register RCGC2. If you use the new RCGCGPIO register, your program might work, but it will not pass this test.
5) "Checking the output port clock" The GPIO module specified by the second parameter to TExaS_Init() must have its clock enabled in the legacy register RCGC2. If you use the new RCGCGPIO register, your program might work, but it will not pass this test.
6) "Checking the Port F clock" GPIO Port F must have its clock enabled in the legacy register RCGC2. If you use the new RCGCGPIO register, your program might work, but it will not pass this test.
7) "Checking the input AMSEL reg." Since all inputs for this lab are digital, the analog functionality must be disabled for all input pins. The appropriate bits of the appropriate AMSEL register must be 0.
8) "Checking the output AMSEL reg." Since all outputs for this microcontroller are digital, the analog functionality must be disabled for all output pins. The appropriate bits of the appropriate AMSEL register must be 0.
9) "Checking the Port F AMSEL reg." Since the red and green LEDs (here used as “walk” and “don’t walk” lights) on Port F are digital, the analog functionality must be disabled for bits 1 and 3. Bits 1 and 3 of the GPIO_PORTF_AMSEL_R register must be 0.
10) "Checking the input port PCTL reg." All pins being used as inputs are standard GPIO inputs. The appropriate bit fields of the appropriate PCTL register must be 0.
11) "Checking the output port PCTL reg." All pins being used as outputs are standard GPIO outputs. The appropriate bit fields of the appropriate PCTL register must be 0.
12) "Checking the Port F port PCTL reg." The red and green LEDs on Port F are standard GPIO outputs. Bits 15-12 and 7-4 of the GPIO_PORTF_PCTL_R register must be 0.
13) "Checking the input direction register" To make a GPIO pin an input, clear the corresponding bit in the direction register. The appropriate bits of the appropriate DIR register must be 0.
14) "Checking the output direction register" To make a GPIO pin an output, set the corresponding bit in the direction register. The appropriate bits of the appropriate DIR register must be 1.
15) "Checking the Port F direction register" To make the pins connected to the red and green LEDs on Port F outputs, set bits 1 and 3 in the direction register. Bits 1 and 3 of the GPIO_PORTF_DIR_R register must be 1.
16) "Checking the input AFSEL register" No inputs for this lab use the alternate function, so the alternate function must be disabled for all input pins. The appropriate bits of the appropriate AFSEL register must be 0.
17) "Checking the output AFSEL register" No outputs for this lab use the alternate function, so the alternate function must be disabled for all output pins. The appropriate bits of the appropriate AFSEL register must be 0.
18) "Checking the Port F AFSEL register" The red and green LEDs on Port F do not use the alternate function, so the alternate function must be disabled for bits 1 and 3. Bits 1 and 3 of the GPIO_PORTF_AFSEL_R register must be 0.
19) "Checking the input pullup PUR register" No internal pull-up resistors are allowed. Switches must be positive logic. The appropriate bits of the appropriate PUR register must be 0.
20) "Checking the output pullup PUR register" Outputs do not need internal pull-up resistors. The appropriate bits of the appropriate PUR register must be 0.
21) "Checking the Port F pullup PUR register" The red and green LEDs on Port F do not

need internal pull-up resistors. Bits 1 and 3 of the GPIO_PORTF_PUR_R register must be 0.
22) "Checking the input pulldown PDR register" External pull-down resistors must be used with the positive-logic switches. The appropriate bits of the appropriate PDR register must be 0.
23) "Checking the output pulldown PDR register" Outputs do not need internal pull-down resistors. The appropriate bits of the appropriate PDR register must be 0.
24) "Checking the Port F pulldown PDR register" The red and green LEDs on Port F do not need internal pull-down resistors. Bits 1 and 3 of the GPIO_PORTF_PDR_R register must be 0.
25) "Checking the input DEN register" Since all inputs for this lab are digital, the digital functionality must be enabled for all input pins. The appropriate bits of the appropriate DEN register must be 1.
26) "Checking the output DEN register" Since all outputs for this microcontroller are digital, the digital functionality must be enabled for all output pins. The appropriate bits of the appropriate DEN register must be 1.
27) "Checking the Port F DEN register" Since the red and green LEDs on Port F are digital, the digital functionality must be enabled for bits 1 and 3. Bits 1 and 3 of the GPIO_PORTF_DEN_R register must be 1.
28) "Checking all switches released" Release all switches and allow your machine to settle to its initial state. The initial state is not specified in the assignment. To pass this test, you have 10 seconds for all three switch input pins to read a 0 at the same time. All switches must be positive-logic.
29) "Checking west switch pressed" Press and hold the switch that corresponds to the car sensor in the west lane while releasing the other two switches. To pass this test, you have 10 seconds for the west switch input pin to read a 1 while the other two input pins read a 0.
30) "Checking west green/south red" Keep holding the west switch while releasing the other two switches. Your FSM should eventually cycle to the state where the west lane is green, the south lane is red, and the “walk” LED is red. How long this takes depends on the length of your delays and how many states are between the initial state and the “west green” state. The assignment requires your delays to be unrealistically short for ease of testing. To pass this test, you have 30 seconds for the west green LED, south red LED, and “don’t walk” LEDs to be on. All other LEDs must be off.
31) "Checking south switch pressed" Press and hold the switch that corresponds to the car sensor in the south lane while releasing the other two switches. To pass this test, you have 10 seconds for the south switch input pin to read a 1 while the other two input pins read a 0.
32) "Checking west yellow/south red" Keep holding the south switch while releasing the other two switches. Your FSM should quickly cycle to the “west yellow” state. To pass this test, you have 10 seconds for the west yellow LED, south red LED, and “don’t walk” LEDs to be on. All other LEDs must be off.
33) "Checking west red/south green" Keep holding the south switch while releasing the other two switches. Your FSM should quickly cycle to the “south green” state. To pass this test, you have 10 seconds for the west red LED, south green LED, and “don’t walk” LEDs to be on. All other LEDs must be off.
34) "Checking west switch pressed" Press and hold the switch that corresponds to the car sensor in the west lane while releasing the other two switches. To pass this test, you have 10 seconds for the west switch input pin to read a 1 while the other two input pins read a 0.
35) "Checking west red/south yellow" Keep holding the west switch while releasing the other two switches. Your FSM should quickly cycle to the “south yellow” state. To pass this test, you have 10 seconds for the west red LED, south yellow LED, and “don’t walk” LEDs to be on. All other LEDs must be off.
36) "Checking west green/south red" Keep holding the west switch while releasing the other two switches. Your FSM should quickly cycle to the “west green” state. To pass this test, you have 10 seconds for the west green LED, south red LED, and “don’t walk” LEDs to be on. All other LEDs must be off.
37) "Checking walk switch pressed" Press and hold the switch that corresponds to the pedestrian sensor while releasing the other two switches. To pass this test, you have 10

seconds for the walk switch input pin to read a 1 while the other two input pins read a 0.
38) "Checking west yellow/south red" Keep holding the walk switch while releasing the other two switches. Your FSM should quickly cycle to the “west yellow” state. To pass this test, you have 10 seconds for the west yellow LED, south red LED, and “don’t walk” LEDs to be on. All other LEDs must be off.
39) "Checking west red/south red" Keep holding the walk switch while releasing the other two switches. Your FSM should quickly cycle to the “walk” state. To pass this test, you have 10 seconds for the west red LED and south red LED to be on. The “walk” LEDs should be green, but it will be checked in the next step (this step only looks at Port A, Port B, or Port E and the “walk”/”don’t walk” LEDs are on Port F). All other LEDs must be off.
40) "Checking "walk" light on" Keep holding the walk switch while releasing the other two switches. Your FSM should already be in the “walk” state. To pass this test, you have 10 seconds for the “walk” LEDs to be green. The 10 second window is not strictly needed. Realistically, it is a good idea to force all lanes of traffic and pedestrians to briefly stop for a short time. This makes it less likely that someone running a questionably legal yellow light will still be in the intersection at the beginning of someone else’s green light. This functionality is not required and not graded.
41) "Checking all switches pressed" Press and hold the switches that correspond to the car sensor in the west lane, the car sensor in the south lane, and the pedestrian sensor. To pass this test, you have 10 seconds for all three switch input pins to read a 1 simultaneously. Do not be sloppy about this; be careful to press all three buttons as close to simultaneously as possible. As soon as you press a car sensor, your FSM will proceed to the next step, flash the “walk” LEDs red-off-red, and continue. If you are slow pressing all three switches, the TExaS real-board grader may miss seeing the “walk” LEDs flash and incorrectly penalize you in the next test.
42) "Checking west red/south red" Keep holding all three switches. Your FSM should quickly cycle to the “don’t walk 1” state. To pass this test, you have 10 seconds (minus your physical reaction time) for the west red LED and south red LED to be on. The “walk” LED should be flashing red, which will be checked in the next steps (this step only looks at Port A, Port B, or Port E and the “walk”/”don’t walk” LEDs are on Port F). All other LEDs must be off.
43) "Checking "don't walk" light on" Keep holding all three switches. Your FSM should already be in the “don’t walk 1” state. To pass this test, you have 10 seconds for the “walk” LEDs to be red. The 10 second window is not strictly needed.
44) "Checking "don't walk" light off" Keep holding all three switches. Your FSM should quickly cycle to the “don’t walk 2” state. To pass this test, you have 10 seconds for the “walk” LEDs to be dark (i.e. “walk” and “don’t walk” off).
45) "Checking "don't walk" light on" Keep holding all three switches. To pass this test, you have 10 seconds for the “walk” LEDs to be red. At this point, the “walk” LED has flashed red-off-red at least once, which is all that is required to demonstrate to TExaS that the LED has flashed. Additional flashes are not graded.
46) "Checking west green/south red" Keep holding all three switches. Your FSM should eventually cycle to the state where the west lane is green, the south lane is red, and the “walk” LED is red. All other LEDs must be off. To pass this test, you have 30 seconds to reach this state. Your solution can step through any valid states before reaching this state as long as it does not take too long.
47) "Checking west red/south green" Keep holding all three switches. Your FSM should eventually cycle to the state where the west lane is red, the south lane is green, and the “walk” LED is red. All other LEDs must be off. To pass this test, you have 30 seconds to reach this state. Your solution can step through any valid states before reaching this state as long as it does not take too long.
48) "Checking "walk" light on" Keep holding all three switches. Your FSM should eventually cycle to the state where the west lane is red, the south lane is red, and the “walk” LED is green. All other LEDs must be off. To pass this test, you have 30 seconds to reach this state. Your solution can step through any valid states before reaching this state as long as it does not take too long.

“Background Tests”: Situations That Will Immediately Cause You to Fail.

Crash! South green LED and green “walk” LED are on simultaneously.
Crash! West green LED and green “walk” LED are on simultaneously.
Crash! South yellow LED and green “walk” LED are on simultaneously.
Crash! West yellow LED and green “walk” LED are on simultaneously.
Crash! West green LED and south green LED are on simultaneously.
Crash! West green LED and south yellow LED are on simultaneously.
Crash! West yellow LED and south green LED are on simultaneously.
Crash! West yellow LED and south yellow LED are on simultaneously.
Invalid State for American Traffic Lights: West red LED and west yellow LED are on simultaneously.
Invalid State for American Traffic Lights: West red LED and west green LED are on simultaneously.
Invalid State for American Traffic Lights: West yellow LED and west green LED are on simultaneously.
Invalid State for American Traffic Lights: South red LED and south yellow LED are on simultaneously.
Invalid State for American Traffic Lights: South red LED and south green LED are on simultaneously.
Invalid State for American Traffic Lights: South yellow LED and south green LED are on simultaneously.
Invalid State for American Traffic Lights: Green “walk” LED and red “don’t walk” LED are on simultaneously, causing the LED to appear yellow-ish.

Interesting questions (things to think about but NOT implement in your lab)

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?

In general, how many next-state arrows are there?

List some general qualities that would characterize a good FSM.

Lab 11. UART Serial Interfacing

Preparation

You will need the LaunchPad. You will use the terminal application TExaSdisplay, which can perform serial port communication on the PC.

Book Reading Sections 7.6, 7.7, 8.1, and 8.2 in Volume 1 Embedded Systems

Starter project Lab11_UART

Purpose

In Lab11 you will learn how to write software that converts numbers into ASCII strings and display the string on the display connected to UART0.

System Requirements

The Lab11 starter project is the similar to **C11_Network** example and includes the connections to the Lab11 grader. When debugging in the simulator you will observe output in the UART debugging window. When running on the real board you will run the terminal program TExaSdisplay. The main program is given and should be used to test your software.

Part a) Make the necessary changes to the UART driver so that it uses UART0 instead of UART1.

Part b) You will write an I/O driver routine that outputs strings to the UART0 device. See the comments in the UART.h and UART.c for more detailed descriptions of how this **`UART_OutString`** function is to work.

Part c) You will write an I/O driver routine that outputs an unsigned decimal number to the UART0 device. See the comments in the UART.h and UART.c for more detailed descriptions of how these **`UART_ConvertUDec`** and **`UART_OutUDec`** functions are to work.

Parameter	UART display
`0`	`"   0 "`
`10`	`"  10 "`
`999`	`" 999 "`
`1000`	`"1000 "`
`9999`	`"9999 "`
`10000 or more`	`"**** "`

Table 11.4. Specification for the **`UART_OutUDec`** function (do not display the quotes").

Part d) Assume the system stores the integers 0 to 9999, but the values mean 0.000 to 9.999 cm. For example, in the software a variable might contain 1234, but that value actually means 1.234 cm. You will write an I/O driver routine that outputs the value of the distance to the UART0 device. See the comments in the UART.h and UART.c for more detailed descriptions of how these **`UART_ConvertDistance`** and **`UART_OutDistance`** functions are to work.

Parameter	UART display
`0`	`"0.000 cm"`
`1`	`"0.001 cm"`
`999`	`"0.999 cm"`
`1000`	`"1.000 cm"`
`9999`	`"9.999 cm"`
`10000 or more`	`"*.*** cm"`

Table 11.5. Specification for the **UART_OutDistance** function (do not display the quotes").

Grading and uploading the score

During checkout, I will grade your system in both simulation and on the real board. During the simulation grading I will automatically set the input and check your output.

While grading on the real board, your system will be tested as it runs on the real board.

Interesting questions

This function tests to see if two strings are equal, explain how it works.

```
//*********strcmp*************
// Compares two strings
// Inputs: two null-terminated strings
// outputs: 1 if equal, 0 if different
int strcmp(unsigned char first[], const unsigned char second[]){
unsigned long i = 0;
  while(1){
    if((first[i]==0)&&(second[i]==0))return 1; // equal
    if(first[i]==0)                  return 0; // different
    if(second[i]==0)                 return 0; // different
    if(first[i] != second[i])        return 0; // different
    i++;
  }
}
```

Programs 11.6 and 11.7 both output a decimal number. How are they the same? How are they different?

Lab 12. Square-wave Tuning Fork using Interrupts

Preparation

You will need a LaunchPad, a 1kΩ resistor, the headphone jack, and headphones.

Book Reading Sections 8.5, 8.6, 9.1-9.6, 9.8, and 9.9 in Volume 1 Embedded Systems

Starter project Lab12_TuningFork
Data sheet SJ1-355XNG SJ1-3553NG.pdf

Purpose

This lab has these major objectives: 1) the understanding and implementing of interrupt software; 2) interfacing an output pin to the speaker, so that the software can generate a buzzing sound at 440 Hz; and 3) the study the accuracy of the pitch created with interrupts. Please read the entire lab before starting.

System Requirements

In this lab you will make a square wave sound at 440 Hz, which is a standard frequency created by a tuning fork. You will interface a positive logic switch and the headphones to two pins, as listed in Tables 12.1 and 12.2. A resistor placed in series with the headphones will control the loudness of the sound. Any value between 680 Ω and 2 kΩ will be OK. Selecting a larger the resistor will make the sound quieter.

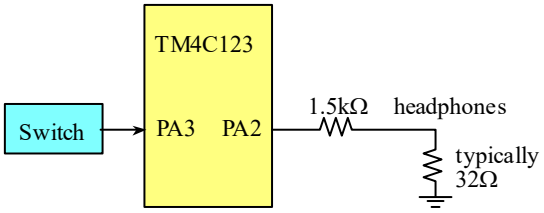


Figure 12.1. Hardware interface for Lab 12.

The lab describes using PA3 for input and PA2 for output, but Tables 12.1 and 12.2 show other options that the automatic graders can handle.

Output	PA2	PB2	PE2

Table 12.1. Possible ports to interface the output (PA2 is default).

Input	PA3	PB3	PE3

Table 12.2. Possible ports to interface the input (PA3 is default).

Figure 12.2 illustrates the operation of the system. You will push the switch to start a quiet 440 Hz tone on the headphones. The sound should continue to be generated until you push the switch a second time. Each time you press the switch the sound should either start or stop.

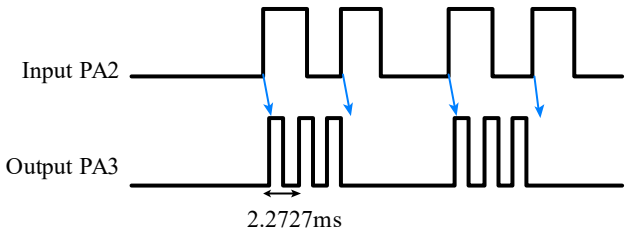


Figure 12.2. Desired input/output behavior. Notice the first time the switch is pressed the 440 Hz tone starts, the second time it is pressed the tone stops, the third time the switch is pressed, the tone starts again, and the fourth time stops the tone again. This on/off pattern should repeat each time the switch is pressed.

Procedure

Power is V*I or V^2/R or I^2R. Assume the output high voltage on PA2 is 3.3V. Let R be the series resistor, shown as 1500 Ω in Figure 12.1. Assume the resistance of your headphones is 32 Ω. Using Ohm's Law we can estimate the current delivered to your headphones

$$I = 3.3V/(R+32\Omega)$$

The power delivered to the headphones will be

$$P = I^2 * 32\Omega$$

You should limit the power to the headphones to less than 1 mW.

Sound will be created only when the voltage across the speaker oscillates. The frequency of oscillation will determine the pitch of the sound. A frequency of 440 Hz will be the note A, which is above middle C on the piano. To output silence, it requires less energy to leave the output pin low. An unchanging high output results in silence from the speaker, but it also causes current to flow through the electromagnet in the speaker for no useful purpose.

Part a) First design and test the system in the simulator. Just like most of the other labs, the **TExaS_Init** will configure the PLL to run at 80 MHz. You can observe the two Port A pins using the logic analyzer available in the simulator. Figure 12.3 shows a zoomed-out view illustrating the proper relation between input and output. Figure 12.4 shows a zoomed-in view illustrating how to measure the frequency of the tone. In previous labs we created delays with software loops. Software delays are difficult to get the time just right and are extremely inaccurate. With the SysTick interrupt you will find creating the 440-Hz wave with an 880-Hz interrupt will be both straightforward and accurate. Furthermore, the same software will create an output wave that will be accurate both in simulation and on the real board.

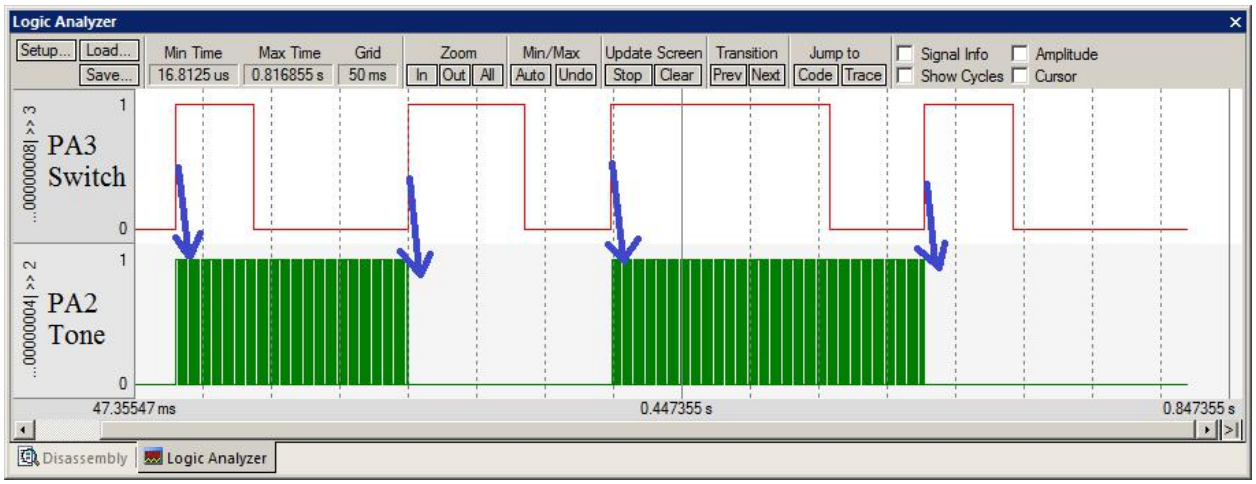


Figure 12.3. Logic analyzer waveforms showing Lab 12 running in simulation mode.

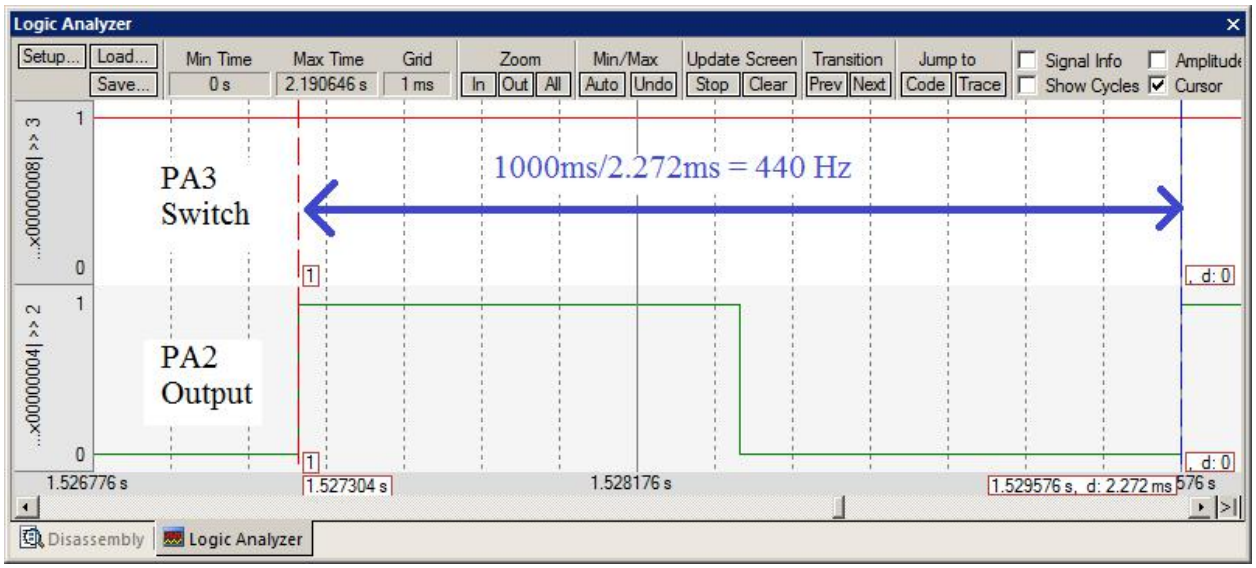


Figure 12.4. Logic analyzer waveforms showing Lab 12 running in simulation mode zoomed in to see the tone is exactly 440 Hz.

There are six steps to the simulation grader:

0) Initialization tests will look specifically to make sure SysTick is interrupting at 880Hz (in particular there is only one value the grader will accept for the SysTick RELOAD register);

1) It will turn the switch off and make sure the output is low;

2) It will turn the switch on and make sure the output toggles;

3) It will turn the switch off and make sure the output continues to toggle;

4) It will turn the switch on and the output should go low;

5) It will turn the switch off and the output should remain low.

Part b) Next, build the external circuit on the breadboard as described in Figure 12.1. Connect the input to the positive logic switch and connect the output to the headphones. DO NOT FORGET THE RESISTOR between the microcontroller and the headphones. Be sure to connect one side of the headphones to ground. *Do not place or remove wires on the protoboard while the power is on.* Since the sound is caused by the oscillations, it doesn't matter which side of the headphones go to the microcontroller and which side goes to ground. Figure 12.5 shows a stereo jack with three pins. The jack is used to connect the headphones to the circuit. Luckily for us, sound will be created if we connect any two of these three pins to the circuit.

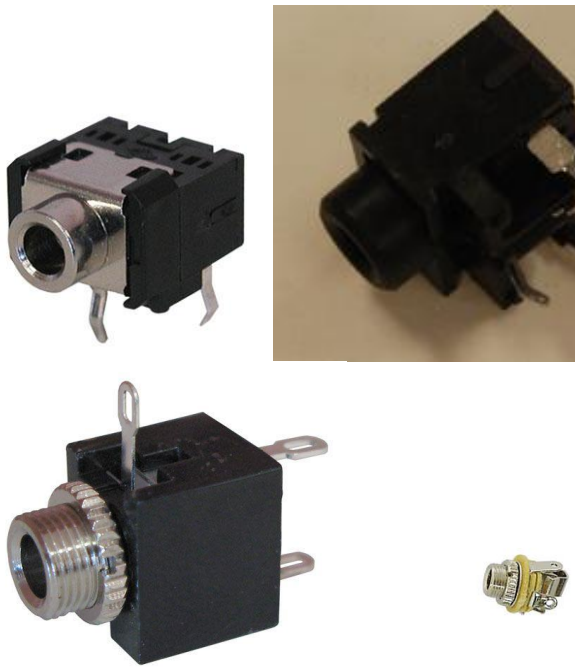


Figure 12.5. You can use either a stereo or mono audio jack.

Figure 12.6 shows one way to connect the headphone jack to the microcontroller. This figure shows which pin I connected to ground and which pin I connected to the 1 kΩ resistor. This jack will plug into the breadboard, but there are many other ways to build the connection

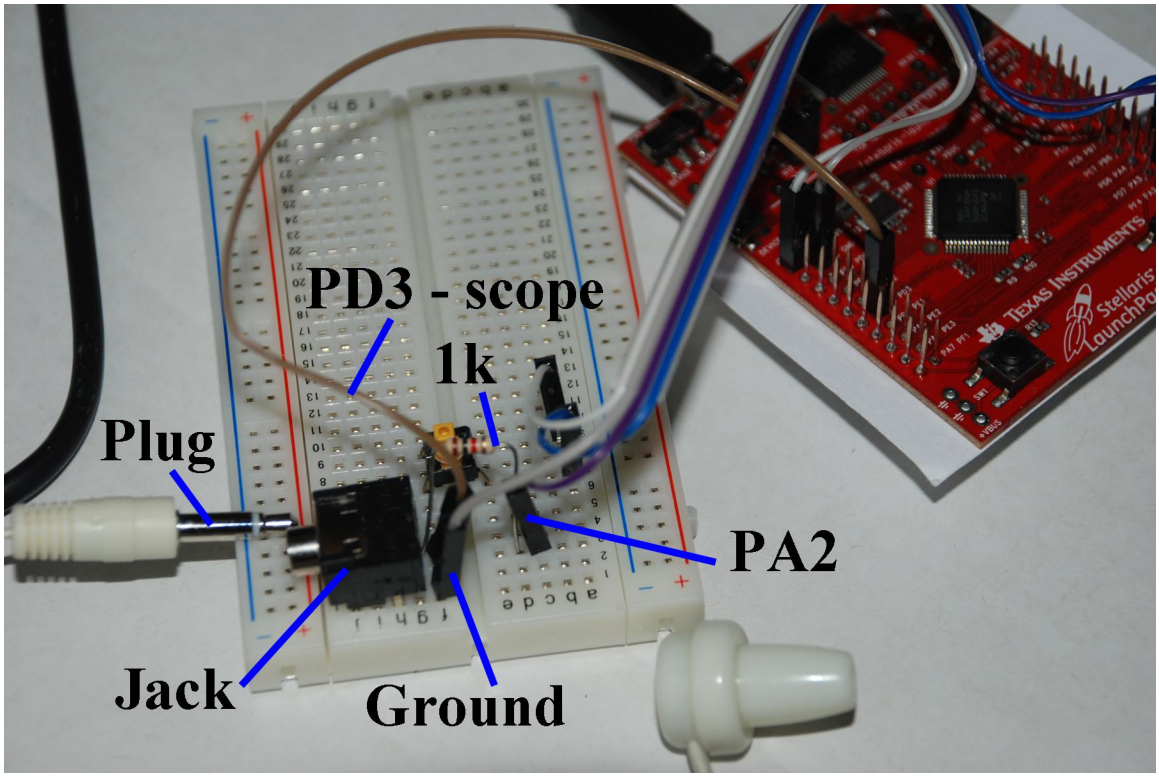


Figure 12.8. Lab 12 built with an audio jack that plugs into the breadboard.

If your audio jack does not plug into your breadboard and you have access to a soldering iron you can solder solid wires to the audio jack. The solid wires are the same type of wires used to build circuits on the breadboard. This means the other end of the wire will plug into the breadboard.

Figure 12.9 shows a system without an audio jack. This last method can be used if your audio jack does not plug into your breadboard and you do not have access to a soldering iron. It is not very elegant, but if you strip off about 3 cm off one end of a solid wire, you can twist the striped portion of the wire around the audio plug of the headphones. Be careful not to short the two wires together.

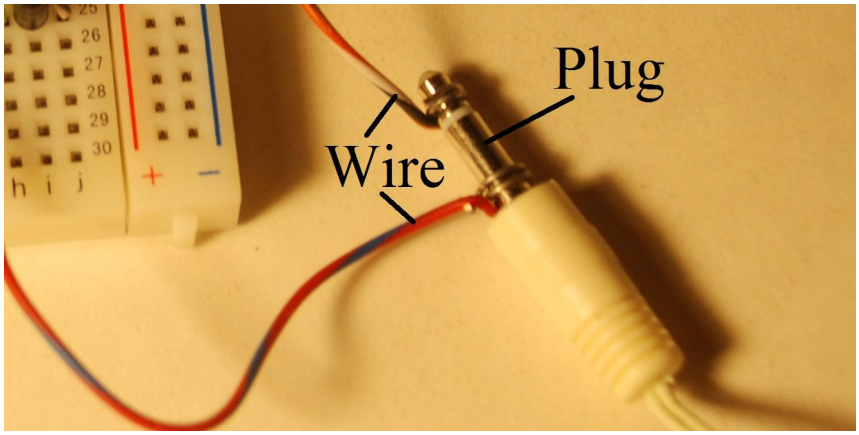


Figure 12.9. A simple way to connect the headphones to the circuit.

How to run the TExaS oscilloscope.

Debug your combined hardware/software system on the actual LaunchPad.To use the TExaS oscilloscope you need these things

1) Connect PD3 to the place you wish to measure;

2) You must be running Lab 12 (or Lab 13) project that calls TExaS_Init with the **ScopeOn** parameter;

3) Open the **TExaSdisplay** application. Within TExaSdisplay, set the COM port to match the board , open the COM port, and select Scope mode.

4) The TM4C123 must be running for data to be streamed from PD3, sampled at 10 kHz by ADC1, and passed through UART0 to the PC.

When using the scope, your software cannot use or change the mode on PD3, ADC1, Timer4, and UART0. Since the scope runs on your computer as a Timer4 interrupt service routine, you will need to leave interrupts enabled for the scope to operate.

The real-board grader is similar to the simulation grader:

0) Initialization tests will check for one input, one output, and SysTick interrupts;

1) It will turn the switch off and make sure the output is low;

2) It will turn the switch in and make sure the output toggles;

3) It will turn the switch off and make sure the output continues to toggle;

4) It will turn the switch on and the output should go low;

5) It will turn the switch off and the output should remain low.

During the real-board grading you will have to push the external switch so that all cases are tested. This grader will arm edge-triggered interrupts on your output pin. So, when your software makes the output go low to high, the grader ISR is run. This ISR will make a very accurate measurement of the actual rate at which you are toggling. Your grader may fail if the switch bounces a lot; so touch the switch gently during grading.

Interesting questions (things to think about but NOT implement in your lab)

Why does it sound so ugly?

Middle C is 261.6 Hz. How would you change the system so a middle C is generated?

What gets pushed on the stack after the hardware trigger and before the ISR is run? Why doesn't it push all the registers?

Assume switch bounce is less than 1 ms. If you check the status of the input switch in your SysTick ISR, and your runs every 1.1ms, what effect will switch bounce have on your system.

Lab 13. Digital Piano

Preparation You will need a LaunchPad, four switches, four 10-kΩ resistors to interface the four switches, resistors to build a 4-bit DAC, the headphone jack, and headphones.

Book Reading Sections 9.6, 10.1, 10.2, and 10.3 in Volume 1 Embedded Systems

Starter project Lab13_DAC

Purpose

This lab has these major objectives: 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.

System Requirements

Most digital music devices rely on high-speed DACs 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 four notes, which will be a digital piano with four 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. The simulator supports up to 6 bits. You will convert the digital output signals to an analog output using a simple resistor network. During the static testing phase, you will connect the DAC analog output to the voltmeter and measure resolution, range, precision, and accuracy. During the dynamic testing phase, you will connect the DAC output to the scope to see the waveform verses time. If you connect the DAC to headphones you will be able to hear the 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 3-kΩ resistor from two 1.5-kΩ resistors. You can create a 6-kΩ resistor from two 12-kΩ resistors.

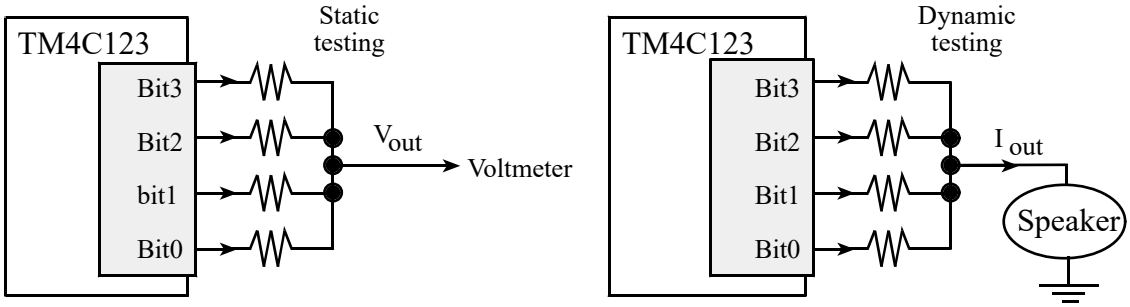


Figure 13.1. DAC allows the software to create music. In the simulator mode, the output voltage Vout is called DACOUT.

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 four keys of the piano. For example, 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 of how it works (which port, which pin) from what it does (which key pressed). You will not need to handle the situation where multiple keys are simultaneously pressed. This interface will be similar to the switch inputs on Lab 10.

The fourth step is to organize the sound generation software into a device driver. You will need a data structure to store the data versus time waveform. The real-board grader expects you to create sinusoidal waveforms. You are free to design your own format, although most students will choose to place the data into a simple array. Although you will be playing only four notes, the design should allow additional notes to be added with minimal effort. For example, you could create public functions **Sound_Init Sound_Off()** and **Sound_Play(note)**, where the parameter **note** specifies the frequency (pitch) of the sound. For example, calling **Sound_Off()** makes it silent by setting the output to zero, and calling **Sound_Play(C)** plays the note C by outputting the stored waveform at a rate such that the resulting sine wave has a frequency of 523.251 Hz. A background thread within the sound driver implemented with SysTick interrupts will fetch data out of your data structure and send them to the DAC. A good design uses the same data array for the DAC outputs, but adjusts the SysTick **RELOAD** value to change the pitch. Remember to output only one DAC value each interrupt. This means there will be a simple relation between the size of the sine table (N), the frequency of the SysTick interrupt (f_s), and the resulting sound frequency (f)

$$f = f_s / N$$

The last step is to write a main program that links the modules together creating the digital piano. After initialization, the main loop will input from the piano keyboard, and then call **Sound_Play(C)**, **Sound_Play(D)**, **Sound_Play(E)**, **Sound_Play(G)**, or **Sound_Off()**, depending on the input pattern. You may ignore multiple keys pressed at the same time.

Procedure

Part a) Decide which port pins you will use for the inputs and outputs. Avoid the pins with hardware already connected. If you plan to do the game in Lab 15, we suggest you place the DAC on Port B. To use the simulator/grader the choices are listed in Tables 13.1 and 13.2. In particular, Table 13.1 shows you three possibilities for how you can connect the DAC output. Table 13.2 shows you three possibilities for how you can connect the four positive logic switches that constitute the piano keys. Obviously, you will not connect both inputs and outputs to the same pin.

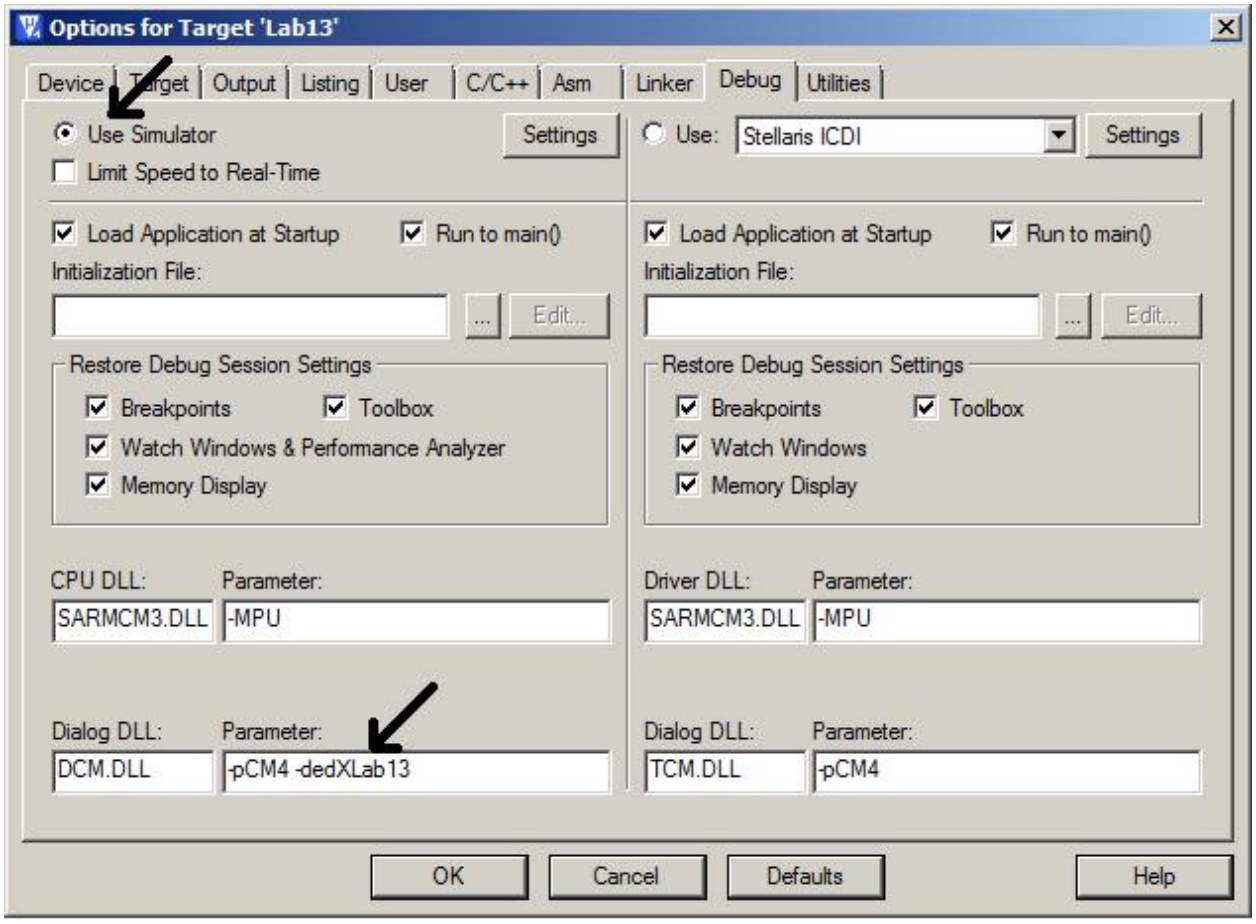
DAC bit 5	PA7	PB5	PE5
DAC bit 4	PA6	PB4	PE4
DAC bit 3	PA5	PB3	PE3
DAC bit 2	PA4	PB2	PE2
DAC bit 1	PA3	PB1	PE1
DAC bit 0	PA2	PB0	PE0

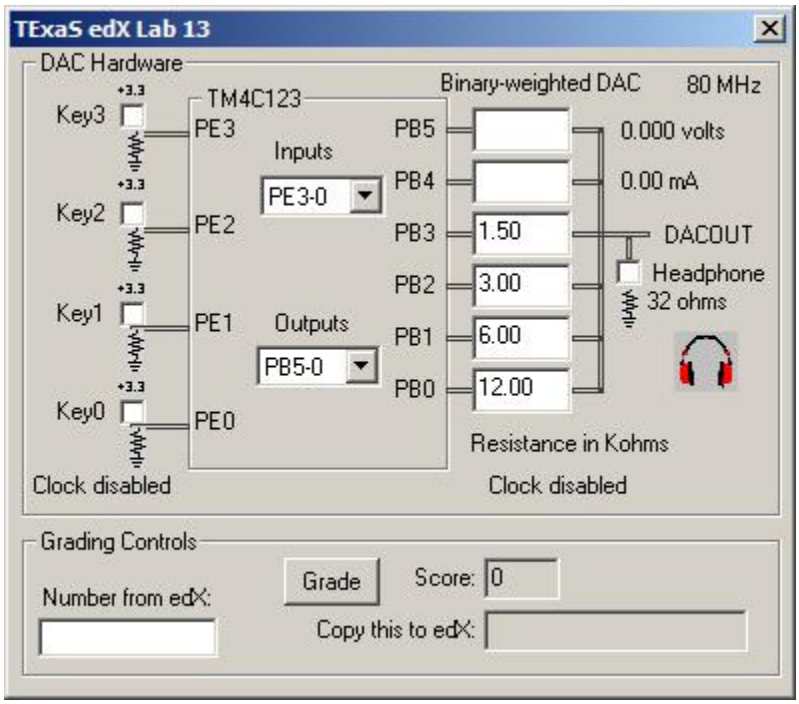
Table 13.1. Possible ports to interface the DAC outputs (DAC bits 4 and 5 are optional).

Piano key 3: G (783.991 Hz)	PA5	PB3	PE3
Piano key 2: E (659.255 Hz)	PA4	PB2	PE2
Piano key 1: D (587.330 Hz)	PA3	PB1	PE1
Piano key 0: C (523.251 Hz)	PA2	PB0	PE0

Table 13.2. Possible ports to interface the piano key inputs.

Figure 13.2 shows some screen shots of Lab 13 in simulation mode. The first figure shows the option tab for configuring Keil for Lab 13 simulation. The second figure shows the dialog window for interacting with the Lab 13 input/output. Specify which pins you plan to use for input and which pins for output. Set the number of DAC pins by setting the resistance values. A blank field means that pin is not connected to the DAC. The third figure shows you how to configure the logic analyzer to display voltage versus time. The symbol DACOUT will be generated by the simulator from the microcontroller digital outputs and the resistance network you specify. The last figure shows the DACOUT as a sine wave is being generated.


Options for Target 'Lab13'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Use Simulator
Settings
Limit Speed to Real-Time
Use:
Stellaris ICDI
Settings
Load Application at Startup
Run to main()
Initialization File:
Edit...
Restore Debug Session Settings
Breakpoints
Toolbox
Watch Windows & Performance Analyzer
Memory Display
Watch Windows
CPU DLL:
Parameter:
SARMCM3.DLL
-MPU
Driver DLL:
Dialog DLL:
DCM.DLL
-pCM4 -dedXLab13
TCM.DLL
-pCM4
OK
Cancel
Defaults
Help


TExaS edX Lab 13
DAC Hardware
TM4C123
Binary-weighted DAC
80 MHz
Key3
Key2
Key1
Key0
+3.3
PE3
PE2
PE1
PE0
Inputs
PE3-0
Outputs
PB5-0
PB5
PB4
PB3
PB2
PB1
PB0
1.50
3.00
6.00
12.00
0.000 volts
0.00 mA
DACOUT
Headphone
32 ohms
Resistance in Kohms
Clock disabled
Clock disabled
Grading Controls
Grade
Score:
0
Number from edX:
Copy this to edX:

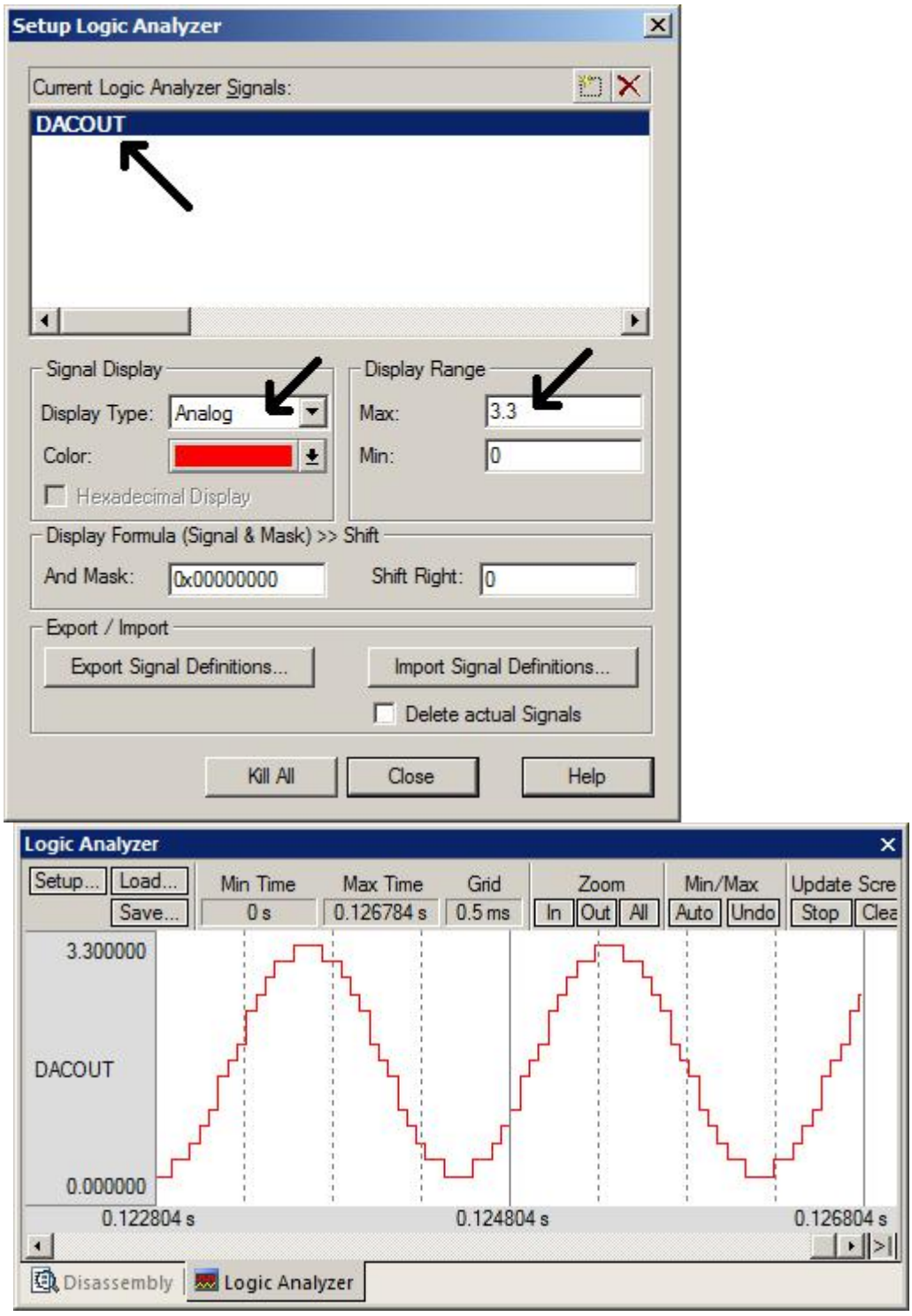


Figure 13.2. The screens needed to run Lab 13 in simulation.

Draw the circuit required to interface the binary-weighted DAC and four switches to the TM4C123. Design the DAC 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. Lab 12 presented three ways to connect the headphones to your circuit. The best solution is to insert the audio jack directly into the breadboard. You could solder 24 gauge solid wires to the audio jack. You could strip some 24 gauge solid wire and wrap it tightly around the headphone plug.

Part b) 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 the **DAC.c** code file. The **DAC.h** header file contains 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 should write a simple main program to test the DAC, similar in style as Program 13.2. 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 13.2). You can single step this program, comparing digital data to the analog voltage at the DACOUT without the speaker attached (i.e., left side of Figure 13.1).

```
#include "DAC.h"
// Inputs: Number of msec to delay
// Outputs: None
void delayms(unsigned long msec){
  unsigned long count;
  while(msec > 0 ) {  // repeat while there are still delay
    count = 16000;    // about 1ms
    while (count > 0) {
      count--;  // each loop takes 5 cycles in simulation
    }
    msec--;
  }
}
int main(void){ // this main is to debug the DAC
      // you must connect PD3 to your DAC output
  TExaS_Init(SW_PIN_PE3210, DAC_PIN_PB3210,ScopeOn);
  DAC_Init(); // initialize SysTick timer and DAC
  EnableInterrupts();  // enable after all initialization are done
   while(1){unsigned long i; // static debugging
     for(i=0;i<16;i++){
       DAC_Out(i);
       delayms(10); // connect PD3 to DAC output
     }
   }
}
```

Program 13.2. A simple program that outputs all DAC values in sequence.

Figure 13.3 shows the static testing using program 13.2 running in the simulator.

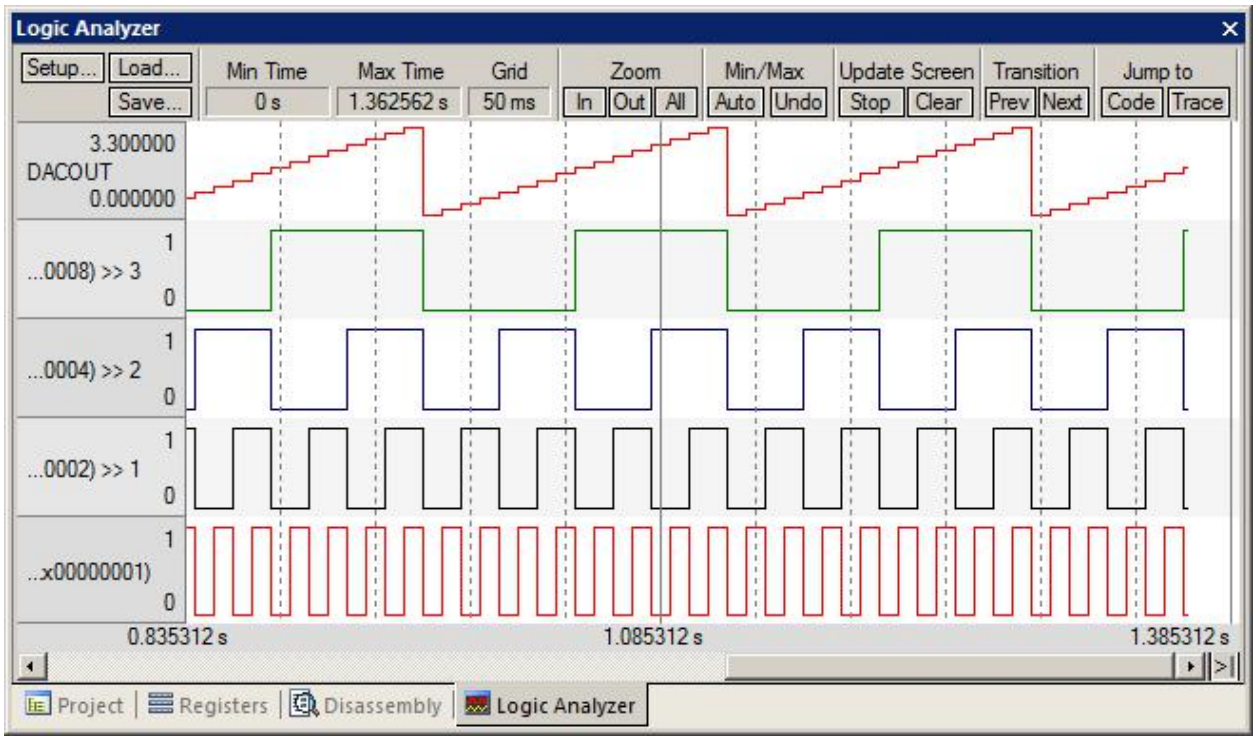


Figure 13.3. A screenshot in simulation mode showing the static testing of the DAC.

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 four switches in the **Piano.c** code file. The **Piano.h** header file contains 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. Once the software calls play, the sound continuous until the software explicitly calls the driver again to change pitch or turn off the sound. In order to turn the sound off, you could implement a third function, **Sound_Off()** or simply call the play function with a special parameter, e.g., **Sound_Play(0)**. Place all code that implements the waveform generation in the **Sound.c** code file. The **Sound.h** header file contains the prototypes for public functions. This driver contains the SysTick ISR, and since the ISR is private to the driver do NOT add a prototype for it in the header file. Add comments that describe what it does in the **Sound.h** file and how it works in the **Sound.c** file. To set the pitch, the play function will write to the **RELOAD** register, without completely initializing SysTick.

Part f) Write a main program to implement the four-key piano. Make heartbeats on PF3, PF2, PF1 so you can see your program is running. Debug the system first in the simulator then on the real board using the TExaS oscilloscope. In the simulator you can add the heartbeats to the logic analyzer. When debugging on the real board you can observe the DAC and heartbeats by looking at the LED, or connecting PD3 and using the TExaS scope. When no buttons are pressed, the output will be quiet and the DAC output 0. Each of the four switches is associated with a separate pitch (C, D, E, G) as listed in Table 13.2. Only one button should be pressed at a time. The sound lasts until the button is released. Figure 13.4 shows a data flow graph of the Lab 13 system.

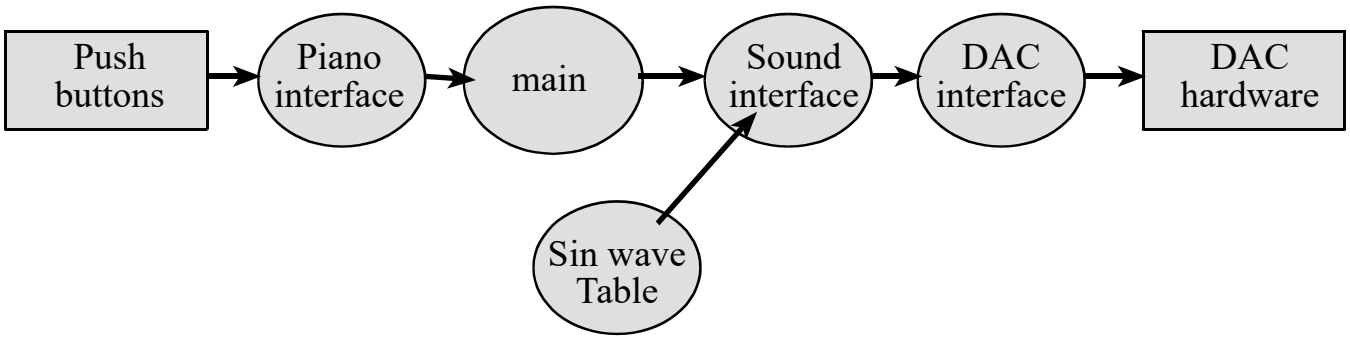


Figure 13.4. Data flows from the memory and the switches to the speaker.

Figure 13.5 shows a possible call graph of the Lab 13 system. Dividing the system into modules allows for concurrent development and eases the reuse of code.

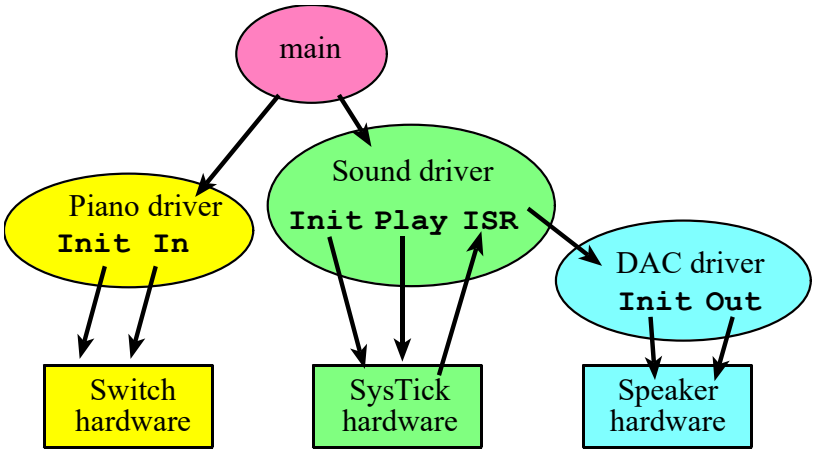


Figure 13.5. A call graph showing the three modules used by the regular part of Lab 6. Notice the SysTick hardware calls the SysTick ISR.

This lab is much too complex to type it all in and expect the system to run. Since there are four modules (DAC, piano, sound, and main) it makes sense to debug each module separately. Start with the DAC and then the piano, because they are lowest level. As with the other labs, you should first complete Lab 13 in simulation. During the simulation grading I will automatically set the inputs and check your outputs. Simulation grading checks for the approximate frequency but not the shape of the wave. As always, watch the command window to see what the grader is looking for.

Part f) After you get a good score in simulation, you should debug on the real board. Because of the complexity of the system and the possibility of hardware errors, we strongly suggest you test first the DAC module and then the piano module. Once you are sure the DAC and piano modules are correct, your can then debug the sound module. The TExaS oscilloscope can be used for debugging on the real board.

Do not connect headphones during grading. Lastly, you will run your system with the real board grader. While the power is off, you will need to connect a wire from PD3 to the output of your DAC. PD3 will become an analog input and the grader will use it to record the DAC output verses time. During the real board grading you will have to push the external switches so that all five cases are tested (pressing a switch causes sound and releasing the switch does not create sound). If the "Signal is too small" then either the headphones are attached or the PD3 wire is not connected to the DAC output. If the grading test sometimes passes and sometimes fails, then your timing may be beyond the +/- 2% expected tolerance. The real board grader checks for both the frequency and the shape (sinusoidal) of the DAC output. If the grading always fails make sure you are outputting sine waves at the proper frequency.

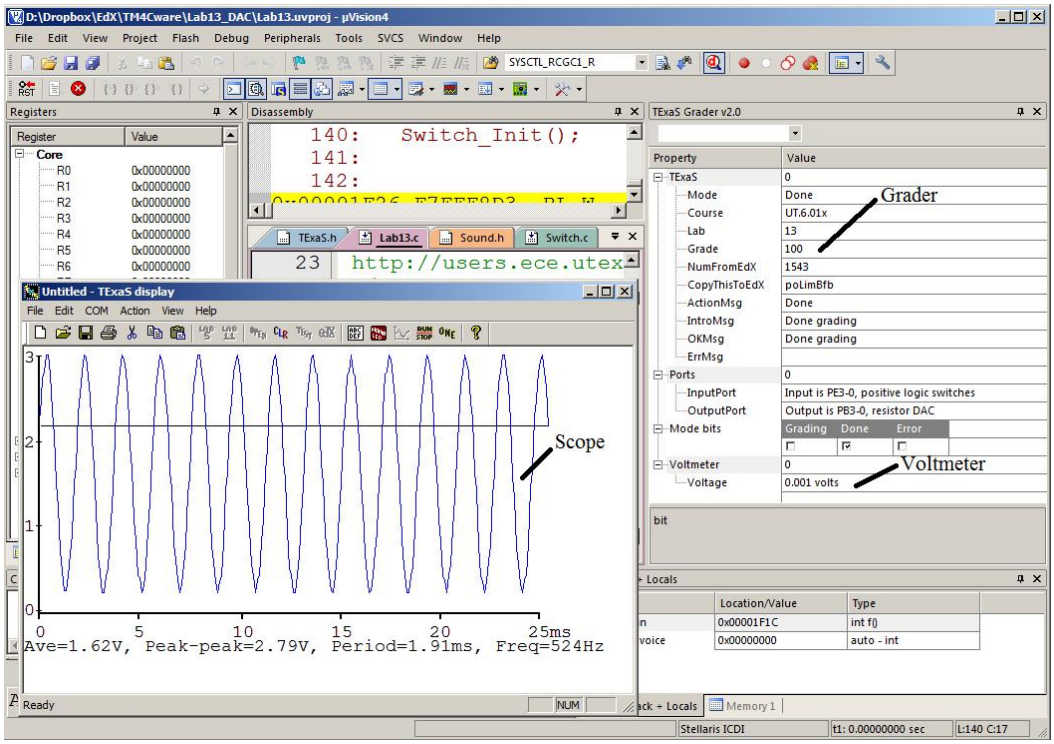


Figure 13.5. Learning environment on the real board.

Interesting questions (things to think about but NOT implement in your lab)

How does the Nyquist Theorem apply to this lab?

If we were to add more bits to the DAC, would we need to increase the table size?

Would it have been possible to vary the frequency of the sine wave while having the interrupt frequency remain the same? If so, how could we have solved it?

How could we have made it sound like a bassoon, guitar or flute? Open the files dac_basson.xls, dac_flute.xls, dac_Guitar.xls, dac_horn.xls, dac_trumpet.xls. How could you have used these files. (Warning, the real board grader expects a sine wave, so these data cannot be used during grading.)

To play a song, we can use another interrupt that calls **Sound_Play**(), feeding the system with the notes to play. Consider the brief song in Figure 13.6. To play this song you will have to call **Sound_Play**() six times. Think about what the parameter will be and when the calls need to be made.

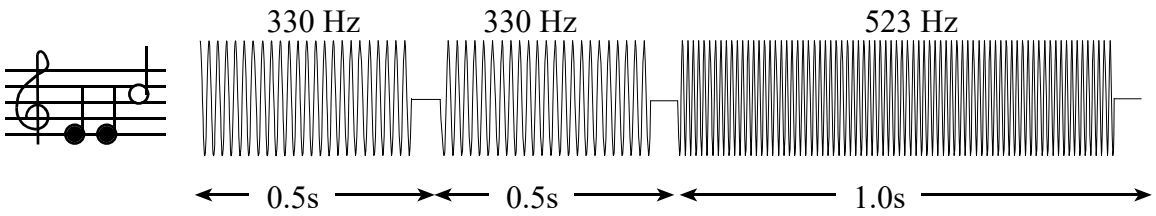


Figure 13.6. A song with three notes.

Lab 14. Measurement of Distance

Preparation

You will need a LaunchPad, a slide potentiometer, a ruler, and an optional 1kΩ resistor. If you have the Nokia 5110 you may use it. The Nokia 5110 LCD is optional.

Book Reading Sections 8.5, 8.6, 9.6, 10.1, 10.4, and 10.5 in Volume 1 Embedded Systems

Starter project Lab14_MeasurementOfDistance

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, which are integers used to represent non-integer values; 5) the development of an interrupt-driven real-time sampling device driver; 6) the development of a software system involving multiple files; and 7) learn how to debug one module at a time.

System Requirements

In this lab you will design a distance meter. A linear slide potentiometer converts distance into resistance ($0 \le R \le 10$ k). Your software will use the 12-bit ADC built into the microcontroller. The ADC will be sampled at 40 Hz using SysTick interrupts. You will write a C function that converts the ADC sample into distance, with units of 0.001 cm. That data stream will be passed from the ISR into the main program using a mailbox, and the main program will output the data on a display. The display is optional.

Some suggested slide pots are listed on the kit buying page, http://edx-org-utaustinx.s3.amazonaws.com/UT601x/worldwide.html. Luckily, any potentiometer that converts distance to resistance can be used. The pots on the buying page can be plugged into the breadboard, whereas others may require you to solder or wrap wires onto the pins. Depending on the size of your pot and how you attach the wires, the full scale range of distance measurement may be anywhere from 1.5 to 10 cm. The pot used in the photos and videos measures distance from 0 to 2 cm. You will use an electrical circuit to convert resistance into voltage (**Vin**). Since the potentiometer has three leads, one possible electrical circuit is shown in Figure 14.1. The default ADC channel is AIN1, which is on PE2. 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 usec. Your software will calculate distance with a resolution of 0.001 cm. The position measurements could be displayed to the computer screen via UART0 using the **TExaSdisplay** interface. If you have a Nokia display, you can instead output the position measurements to it. A periodic interrupt will be used to establish the real-time sampling.

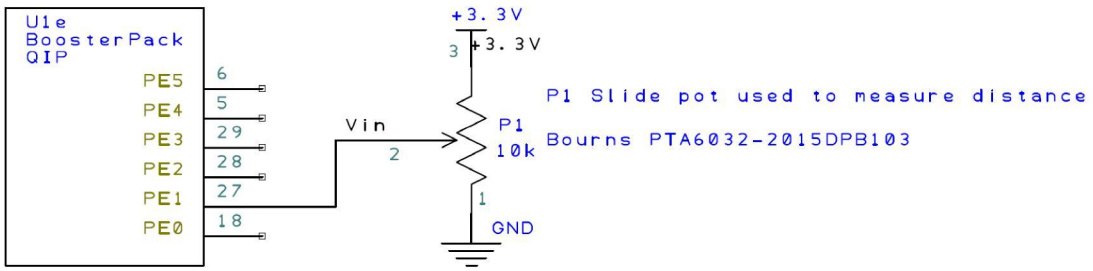


Figure 14.1. Possible electrical circuit to interface the distance transducer to the microcontroller.

The left of Figure 14.2 shows a possible data flow graph of this system. Dividing the system into modules allows for concurrent development and eases the reuse of code. Each module will have a code file and a corresponding header file. The code file contains the actual implementation, and the header file has the prototypes for public functions. The SysTick initialization, SysTick ISR, and the main program will be in the **main.c** file. The ADC module will consist of the **ADC.c** and **ADC.h** files. The LCD module will consist of the **Nokia5110.c** and **Nokia5110.h** files.

Figure 14.3 shows a possible call graph. A call grap

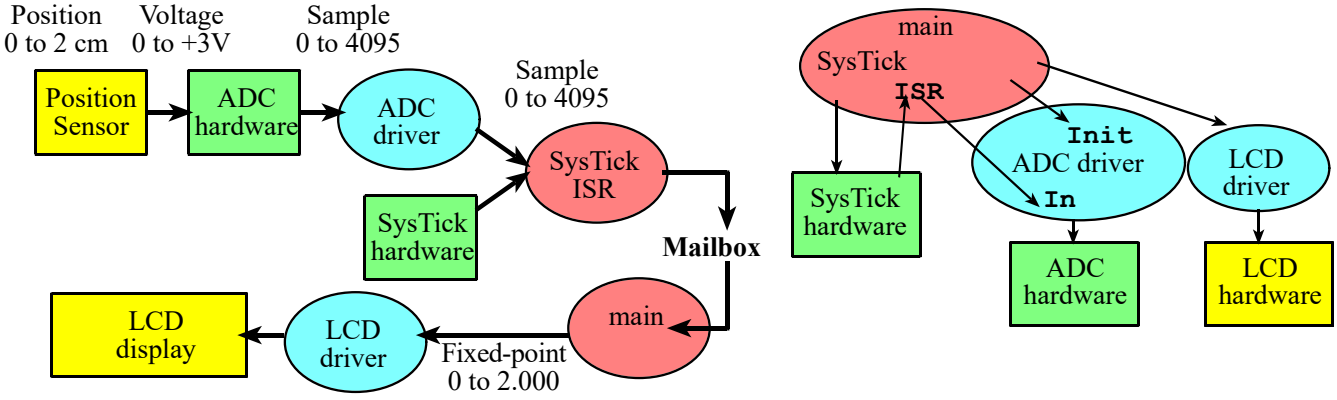


Figure 14.2. Data flow graph and call graph of the position meter system. Notice the hardware calls the ISR. If you do not have a Nokia 5110 LCD, you can use the UART and TExaSdisplay.

You should make the distance resolution and accuracy as good as possible using the 12-bit ADC. The distance resolution is the smallest change in distance that your system can reliably detect. In other words, if the resolution were 0.01 cm, and the distance 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 2 cm/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.

The **armature** is defined as the part that moves. In this lab, you will be measuring the position of the armature (the movable part) on the slide potentiometer, see Figure 14.3. Figure 14.4 shows a clear definition of "true" distance by using a ruler and a cursor.

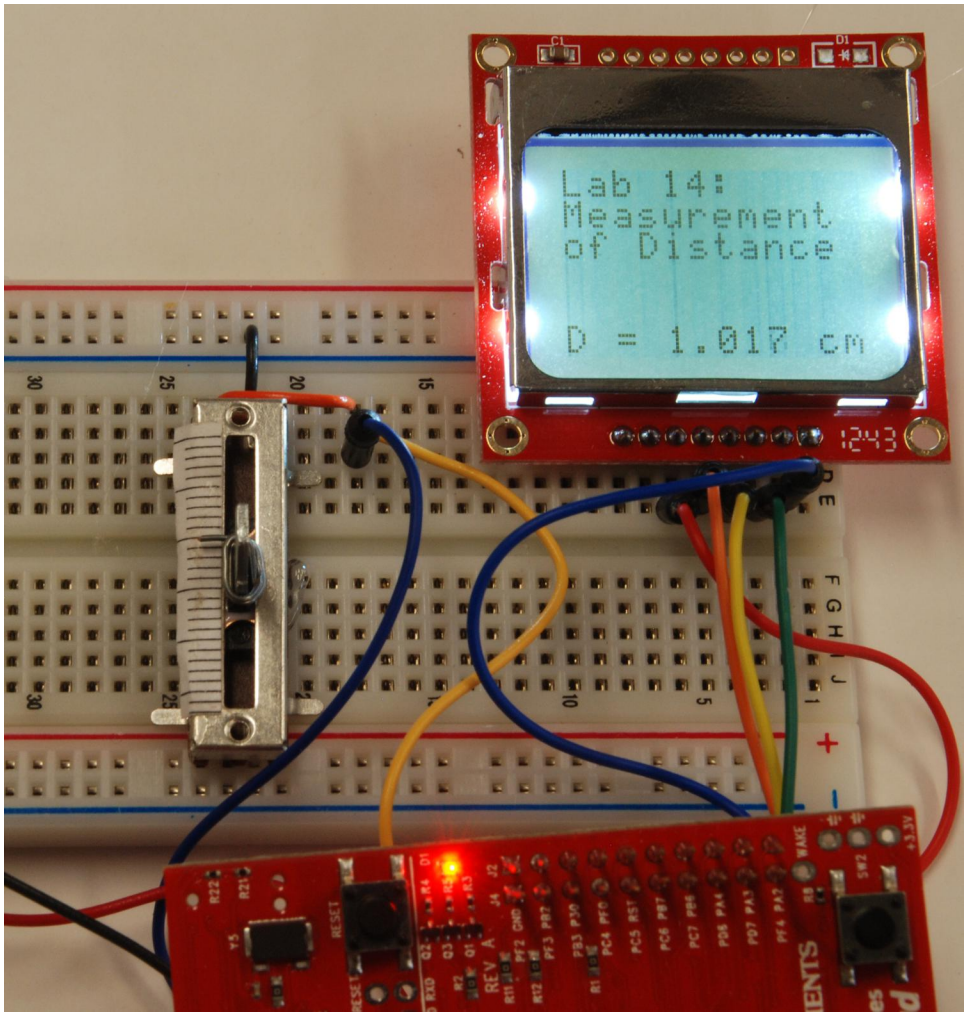


Figure 14.3. Hardware setup for Lab 14, showing the slide pot and optional Nokia. The slide pot is used to measure distance.

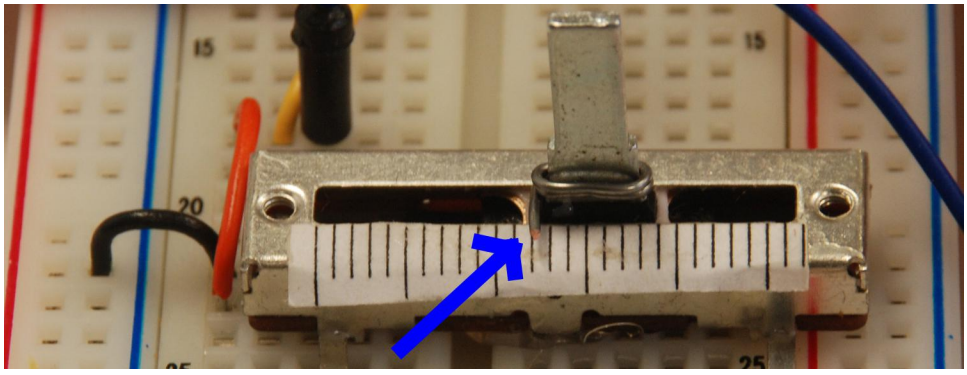


Figure 14.4. Close-up photo of the transducer. The tip of the cursor defines "truth".

Due to the mass of the armature and the friction between the armature and the frame, the distance signal has a very low frequency response. One way to estimate the bandwidth of the distance 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 distance as a signal sine wave $x(t)=1cm*\sin(2\pi ft)$, we calculate the maximum velocity of this sine wave to be $1cm*2\pi f$. Therefore we estimate the maximum frequency using $20cm/sec = 1cm*2\pi f$, 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 were able to oscillate it 10 times a second, then we would 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. A SysTick interrupt will be used to establish the real-time periodic sampling.

You will sample the ADC exactly every 0.025 sec and calculate distance using decimal fixed-point with Δ of 0.001 cm. This means if the distance is 1.234 cm, we will store the integer 1234 in the computer. Conversely, if the integer in the computer is 567, it will mean the distance is 0.567 cm. If you do display the results, we suggest you include 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. Did you notice the least significant digit flickering in the video? You should expect your least significant digit to flicker as well. We made you display the thousandth digit just you can see that the ADC is not the limiting factor for resolution. In most data acquisition systems the noise and transducers are significant sources of error.

When a transducer is not linear, you could use a piece-wise linear interpolation to convert the ADC sample to distance. 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. This is a very general approach and can be used for most applications.

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 14.5, is use a linear equation to convert the ADC sample to position (Δ of 0.001 cm.) Since the transducer is linear, we will use this simple approach. Let **ADCdata** be the 12-bit ADC sample and let **Distance** be the distance in 0.001-cm units. You will find a linear equation, with slope and offset, to convert the ADC sample into distance.

Distance = (A***ADCdata**)>>10 + B where A and B are calibration constants

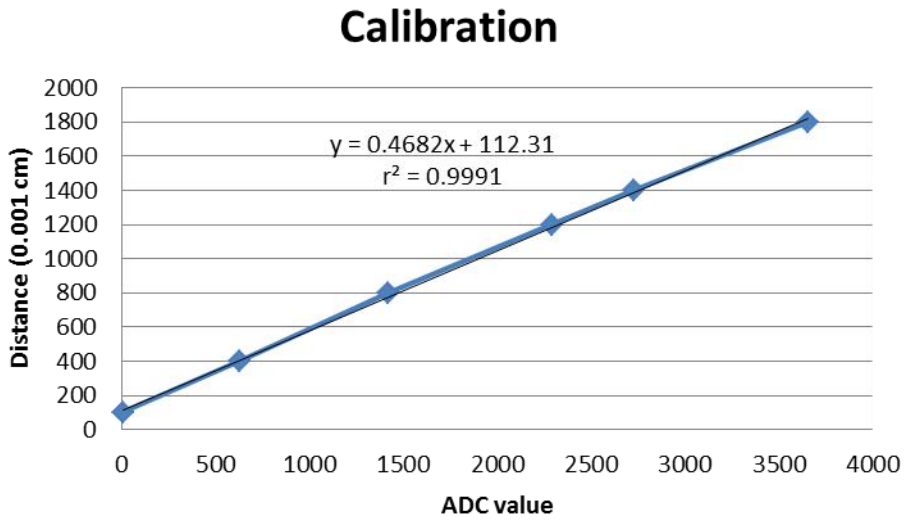


Figure 14.5. A linear equation used to convert ADC value to distance.

One measure of linearity is first fit a straight line to the Distance versus ADC data using linear regression. For more information on linear regression see http://en.wikipedia.org/wiki/Simple_linear_regression. The linearity then is defined how well the points in figure match the fitted straight line and given as the r^2 value. An r^2 value of 1 means the data is perfectly linear. The data in Figure 14.5 is pretty close to linear (r^2=0.9991). The real board grader will measure linearity and require an r^2 value above 0.96 to pass.

Furthermore we expect you to notice that **calibration drift** will be the limiting factor for accuracy. Calibration drift occurs when the constants A and B change over time.

Specific Requirements

1) You must connect a slide pot to ADC channel 0, 1, or 8. You will need to define some place on the slide pot as "true" distance.

2) You must use ADC0, sequencer 3, and software start (because these are the only modes simulated by TExaS).

3) You must use SysTick interrupts to sample the ADC at 40 Hz. In particular, the ADC should be started exactly every 25 msec.

4) You must convert the ADC to distance, storing your measurement into the global variable **Distance**. For example, if the "true" distance is 1.2 cm, then your software should set **Distance** to 1200. The SysTick ISR will store the 32-bit calibrated distance in the variable **Distance** (called a mailbox) and set a flag. Reread Section 12.2 in the ebook to see how a mailbox can be used to pass data from the background into the foreground.
5) It is optional to output the results to the UART or to the Nokia LCD. However, if you do output you must perform the output in the main program and use a mailbox to pass the data from the ISR to the main. The main program will collect data from the mailbox and convert the distance with a Δ of 0.001 cm into a string, using your **UART_ConvertDistance()** function that you developed as part of Lab 11. The string can be display on the UART with **UART_OutString** or on the Nokia with **Nokia_OutString**. Include units on your display.
6) We expect the accuracy to be better than 0.02 cm in simulation, and better than 0.1 cm on the real board. Accuracy is defined as the difference between truth and measured. Only the simulation grader will check your accuracy.
7) We expect the linearity to be better r^2=0.96. Linearity is defined as the r^2 value calculated when the distance versus ADC sample is fit to a straight line. Only the real board grader will check your linearity.

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.

Part a) Decide which port pin you will use for the ADC input. If you plan to do the game in Lab 15, we suggest you place the ADC on PE2, which is ADC channel 1. However, the graders will allow you to use channel 0 (PE3), 1 (PE2), or 8 (PE5), as listed in Table 14.1. The game in Lab 15 requires you to use channel 1 (PE2) to sample a potentiometer in the similar way as this lab. Therefore, we recommend PE2/Ain1.

ADC Distance Measuring Pot.	AIN0 (PE3)	AIN1 (PE2)	AIN8 (PE5)

Table 14.1. Possible ports to interface the ADC inputs. The highlighted column is the recommended pin.

Part b) Write the ADC software that initializes and samples the ADC. Notice there is an **ADC.h** file containing the prototypes for the public functions and an **ADC.c** file containing the implementations. For this part, you are to write **ADC0_Init()** and **ADC0_In**. We suggest you look at the ADC code in section 14.2 and convert it to sample the pin you selected in Part a). You should debug this in the simulator, using a main program like the following

```
unsigned long ADCdata;
int main(void){
  TExaS_Init(ADC0_AIN1_PIN_PE2, SSI0_Real_Nokia5110_NoScope);
  ADC0_Init();    // initialize ADC0, sequencer 3, software start
  EnableInterrupts();
  while(1){
    ADCdata = ADC0_In(); // read ADC, return 12-bit
  }
}
```

Remember, if you use the UART for the scope, you will not be able to use the UART for displaying data. Figure 14.6 shows the simulator screen. During this test you should see a linear relationship between the voltage on PE2 and the number in the **ADCdata** variable. In the simulator, this relationship is exact, but on the real board the relationship will be similar but not exactly like this.

$$ADCdata = 4095*Vin/3.3$$

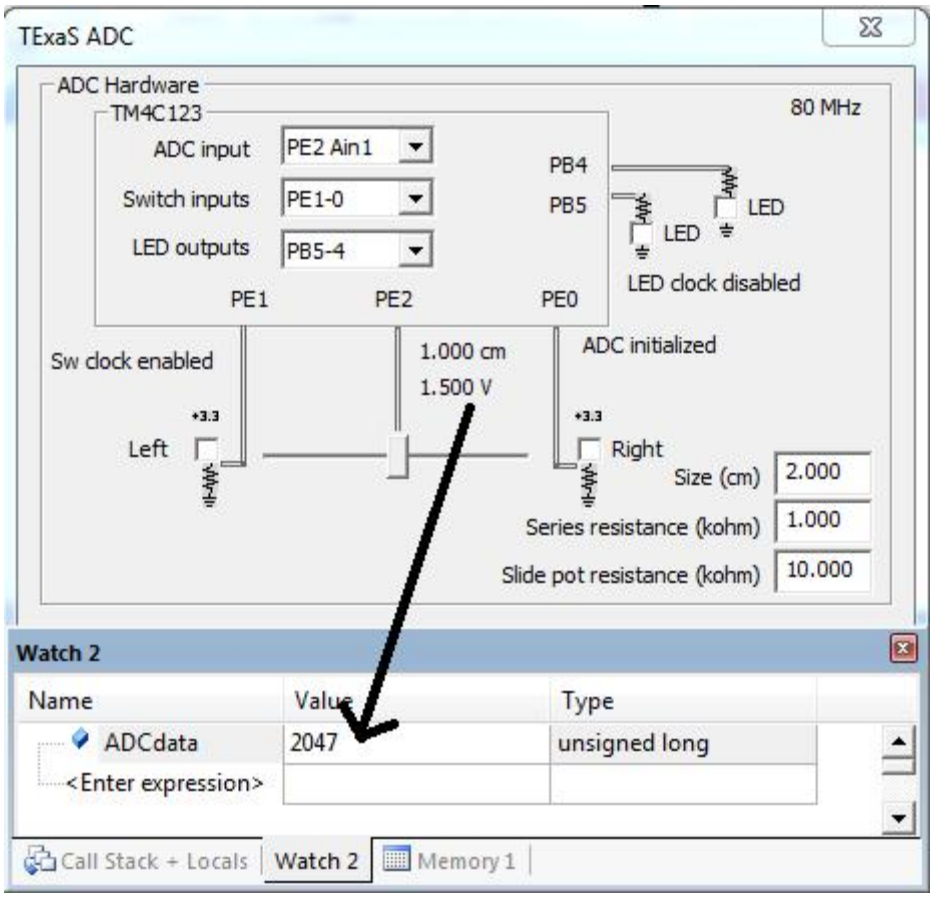


Figure 14.6. Debugging screen showing `ADC0_In()` *properly converts voltage to a digital number.*

Part c) We suggest you observe the full scale range of your potentiometer. The Bourns potentiometer in the figures has a full range of 2 cm, so I set my Size to 2.0 cm. You should set your Size field to match your actual potentiometer. Also set your resistance values to match what you expect to build once you get to real-board testing. Next, you will calibrate the system in the simulator. In the simulator all responds are linear and ideal. Run your test program in part b) for a couple of distances, and then determine the A and B constants that convert the ADC sample into distance with 0.001 cm units.

Distance = ((A***ADCdata**)>>10)+B where A and B are calibration constants

Write a C function that performs this conversion, called **Convert()**. You should debug this function and the optional output to the display in the simulator, using a main program like the following. This example uses the optional Nokia to display results. Figure 14.7 shows the simulator screen the Nokia output matches the true distance. Don't expect it to be perfect, but it should be close.

```
unsigned long ADCdata;
unsigned long Distance;
int main(void){
  TExaS_Init(ADC0_AIN1_PIN_PE2, SSI0_Real_Nokia5110_NoScope);
  ADC0_Init();    // initialize ADC0, channel 1, sequencer 3,
software start
  Nokia5110_Init();        // optional: initialize Nokia5110 LCD
  EnableInterrupts();
  while(1){
    ADCdata = ADC0_In();
    Nokia5110_SetCursor(0, 0);   // optional: move cursor to top
    Distance = Convert(ADCdata); // required step, you write this
    UART_ConvertDistance(Distance); // optional: from Lab 11
    Nokia5110_OutString(String);    // optional: output to LCD
  }
}
```

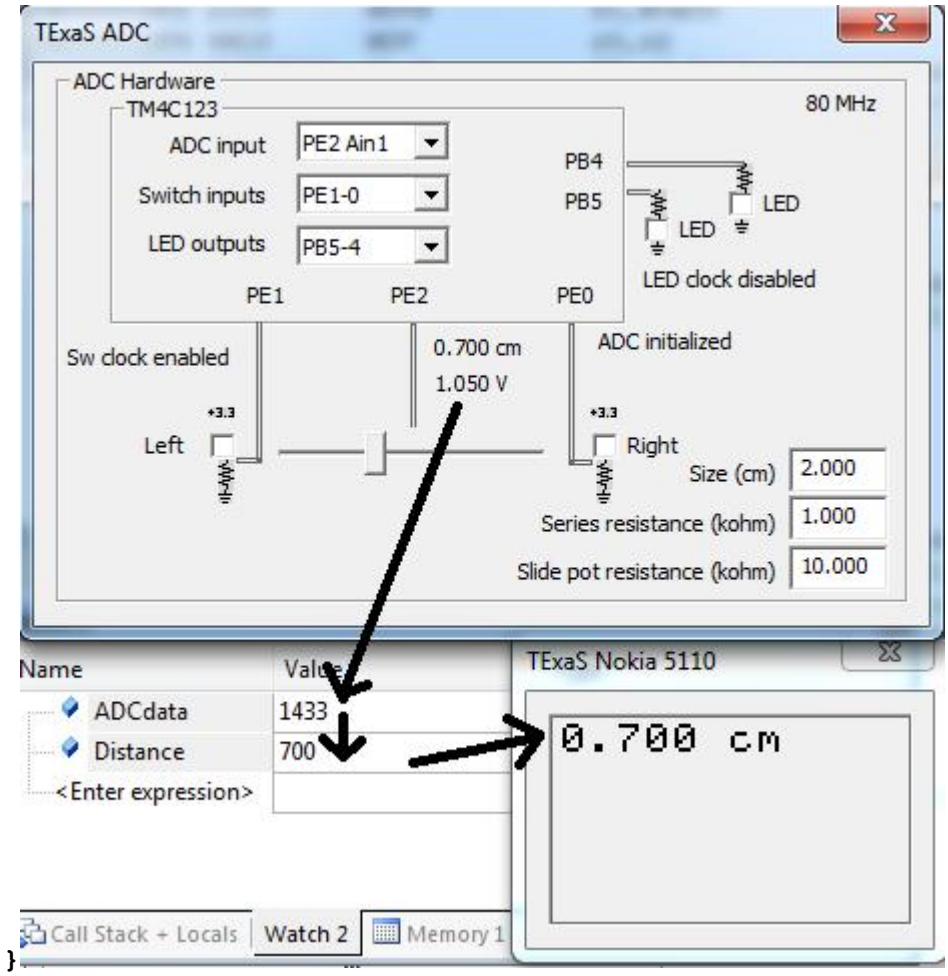


Figure 14.7. Debugging screen showing Convert properly converts the ADCdata into Distance.

Part d) Next, you should configure SysTick interrupts to sample at 40 Hz, so that you can perform the ADC sampling at a regular rate. Sampling at a known and regular rate is essential for real-time embedded systems. It is not required, but I suggest you add a debugging monitor on an extra pin. I used PF1 (the red LED), but feel free to use any digital output. If we add a global variable called **Flag**, then the **Distance** and **Flag** variables together constitute a **mailbox**. The pseudo code for SysTick ISR is

1) Toggle PF1
2) Toggle PF1 again
3) Sample the ADC, calling your **ADC0_In()**
4) Convert the sample to **Distance**, calling your **Convert()**, and storing the result into the global
5) Set the Flag, signifying new data is ready
6) Toggle PF1 a third time

The pseudo code for main program is

1) Activate **TExaS_Init**
2) Enable the ADC, calling your **ADC0_Init()**
3) Optional: enable the display, either
   a) Call **Nokia5110_Init()**
   b) Call **UART0_Init()**
4) Configure SysTick for 40 Hz interrupts
5) Configure PF1 as a regular GPIO output
6) Enable interrupts
7) Perform these steps in an infinite loop
   Clear the **Flag** to 0
   Wait for the **Flag** to be set
   Convert **Distance** to **String**, calling your **UART_ConvertDistance()**
   Optional: output the string
      a) **Nokia5110_SetCursor(0, 0); Nokia5110_OutString(String);**
      b) **UART_OutString(String); UART_OutChar('\n');**

The simulation screen should look similar to Figure 14.7, showing the distance on the dialog matches the **Distance** in the watch window and the distance on the display. Figures 14.8 and 14.9 illustrate proper deployment of interrupts. Figure 14.8 shows the ISR running exactly at 40 Hz (25 ms), and Figure 14.9 shows the time to execute the ISR is small compared to the time between interrupt triggers.

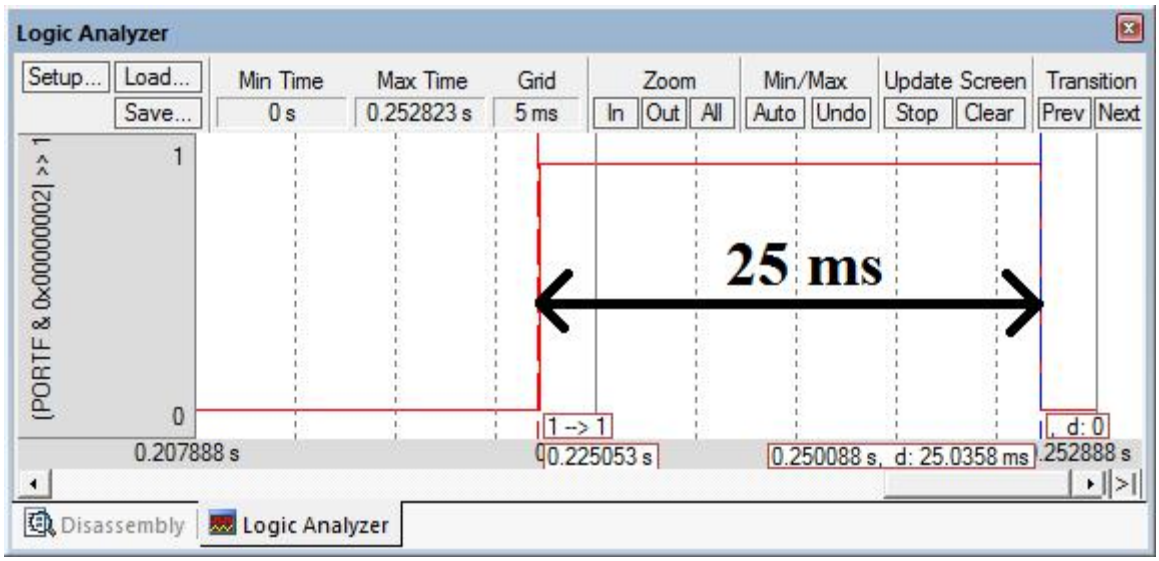


Figure 14.6. Zoomed out debugging screen shows SysTick interrupting every 25 ms.

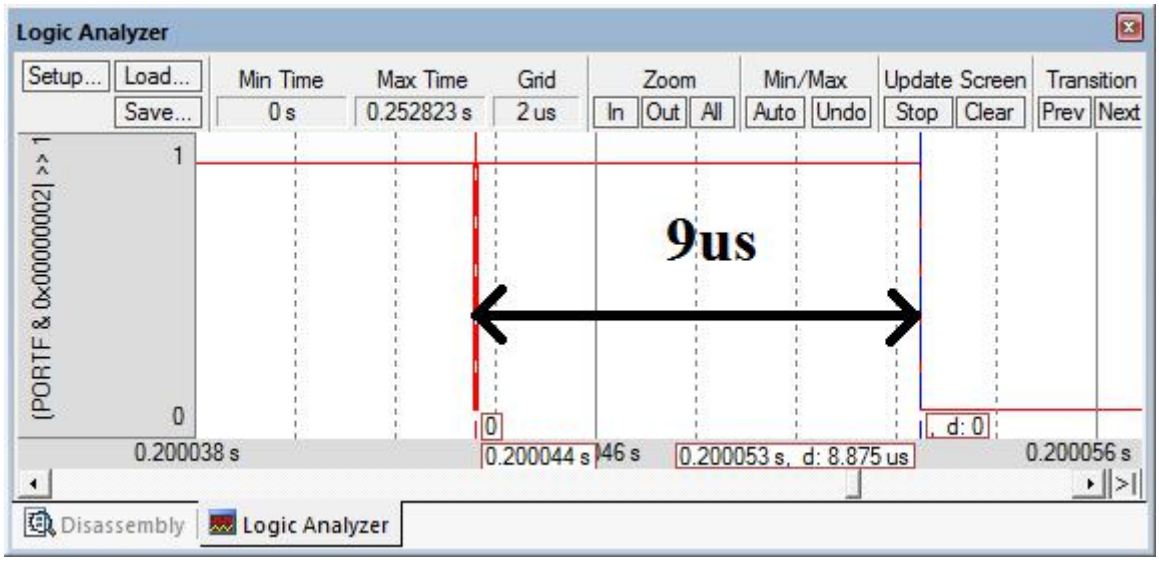


Figure 14.7. Zoomed in debugging screen shows SysTick ISR executes in about 9us.

You do not need a display to pass the simulation grader. There are three steps to the simulation grader:

0) Initialization tests will look specifically to make sure SysTick is interrupting at 40Hz. The system is running at 80 MHz, and there is only one value the grader will accept for the SysTick RELOAD register that will produce an interrupt every 25 ms;
1) The grader will check to see if the ADC is initialized properly and the channel you have selected in the dialog window matches the channel you configured the ADC. You must use ADC0. However, you may use channel 0, 1, or 8. You must use sequencer 3, and software start;

2) You must store your position measurements into the global **Distance** with units of 0.001 cm. It doesn't matter what settings you select for Size, Series Resistance or Slide Pot Resistance. Therefore, I suggest you make these settings match the hardware you will build on the real board. The grader will move the slide pot to five different positions, and calculate the average accuracy comparing truth to your measurements. After each move to a new position, the grader will wait at least 35ms and observe your global variable **Distance**. Let n be the number of data points, let t_i be the true distance, and let m_i be your measured distance. Both distances have units of 0.001 cm, and the index i varies from 0 to n-1. The grader determines **accuracy** by calculating the average difference between truth and measurement,

$$\text{Average error (with units in 0.001 cm)} = (\text{sum}(|t_i - m_i|))/n$$

Average error must be less than 0.02 cm. The simulation grader does not check to see if you output the results to the LCD or the UART.

After you get a good score in simulation, you should debug on the real board. Because of the complexity of the system and the possibility of hardware errors, we strongly suggest you perform the same three tests: 1) just the ADC; 2) ADC and display; and 3) ADC, display and SysTick.

Part e) There are many ways to build a transducer. If your potentiometer has tabs, then you will need to gently bend them so the pot will plug into the breadboard. Be careful not to bend the pins. I suggest you consult a data sheet for your potentiometer to verify which pins refer to the three pins shown in Figure 14.1. The data sheet for the Bourns PTA series of slide pots can be found at BournsPTASlidePotentiometer.pdf. The next video shows how I bent the tabs on my slide pot.

If your pot does not plug into the breadboard, attach three wires to the pins by soldering or twisting solid wire very tightly on the three pins.

You will need a mechanical definition for "true" distance. A cursor in Figures 14.8 and 14.9 was created by twisting a solid wire around the armature. I glued the metric ruler onto the slide pot positioning the ruler so the zero-position of the slide pot (closest to pin 1) lined up with the zero-position of the ruler. I defined truth as the position of the cursor over the ruler. Feel free to define truth as however you wish. For example, you could skip the cursor and define truth as one edge of the armature itself. If you are gluing, be careful not to get glue into the potentiometer. The potentiometer may have places where it is nonlinear. So you may have to run the grader until your find a set of points that are linear enough. The full scale range may be any value from 1.5 to 10.0 cm, depending on your potentiometer. The full scale range does not matter.

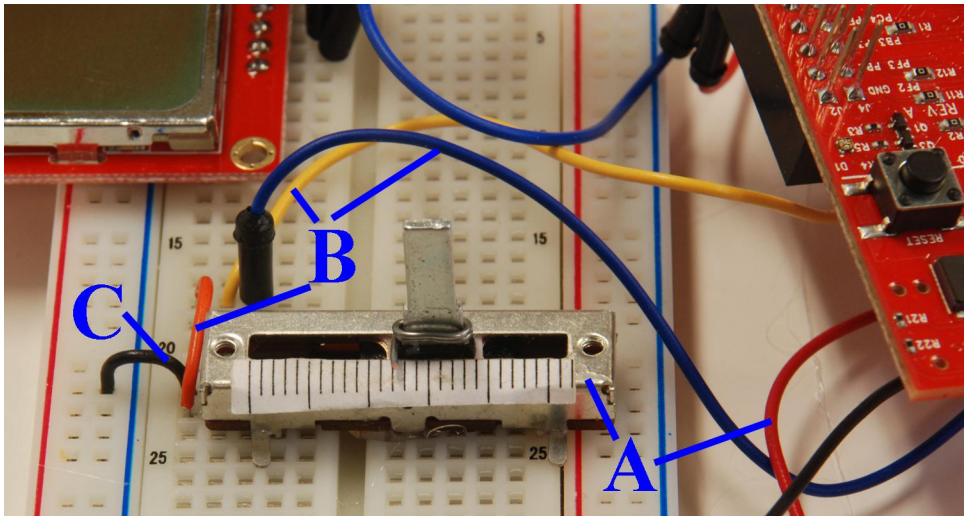


Figure 14.8. Hardware setup for Lab 14, showing the slide pot. In this system the label A is +3.3V, which is also connected to pin 3 of the potentiometer. The label B is pin 2 of the potentiometer, which is also connected both to PE1 and PD3. Label C is ground, which is also connected to pin 1 of the potentiometer.

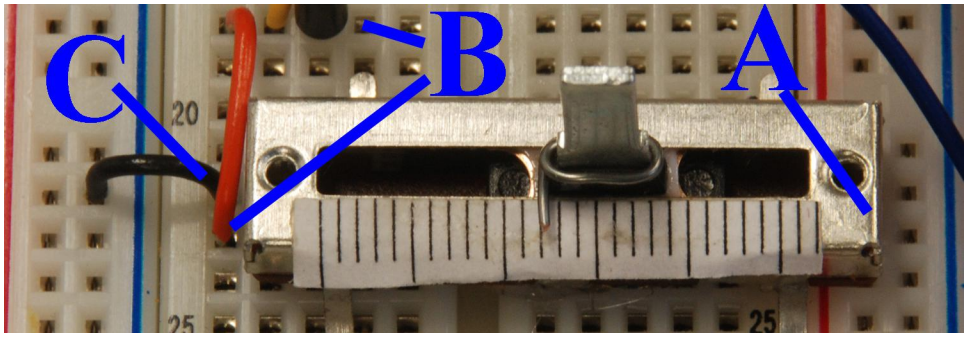


Figure 14.9. Top view of the slide pot. Notice the wire twisted around the armature creating a cursor at which the "true" distance is defined.

Part f) Because of imperfections in the ADC and the slide pot, you will have to calibrate your system. I ran one of the test programs and collected this data, where Truth has units 0.001 cm

Truth	ADCdata
500	784
1000	1878

1500	2980

Next I calculated the two slopes. If the slopes are not about the same, I suggest you retake the data

```
Slope = (1000-500)/(1878-784) = 0.457
Slope = (1500-1000)/(2980-1878) = 0.453
```

The average slope is about (0.457+0.453)/2 = 0.455. Next I calculated three offsets

```
Offset = 500-0.455*784 = 143
Offset = 1000-0.455*1878 = 145
Offset = 1500-0.455*2980 = 143
```

The average offset is about (143+145+143)/3 = 144. This means I should calculate

```
Distance = 0.445*ADCdata +144
```

I calculate the constant A as 0.445*1024 = 466. The equation the software will calculate is

```
Distance = ((466*ADCdata)>>10) +144
```

Part g) One by one repeat the testing of the modules on the real board. Just like the simulator, the goal is to get the value of Distance, as seen in a Watch window, to match the true distance on the slide pot. The real boar grader looks at linearity rather than accuracy, so you do not need to spend a lot of time making the real system accurate.

The TExaS voltmeter, PD3, can be used for debugging on the real board. However, the UART cannot be used for both the oscilloscope and as a character display at the same time. Remember the microcontroller must be running for the voltmeter and the oscilloscope to be operational. You will be able to use the TExaS oscilloscope to see zoomed-out execution profile like Figure 14.8, but not zoomed in like Figure 14.9 because this oscilloscope takes data only at 10 kHz. Figure 14.12 shows the SysTick ISR is indeed running at 40 Hz on the real board.

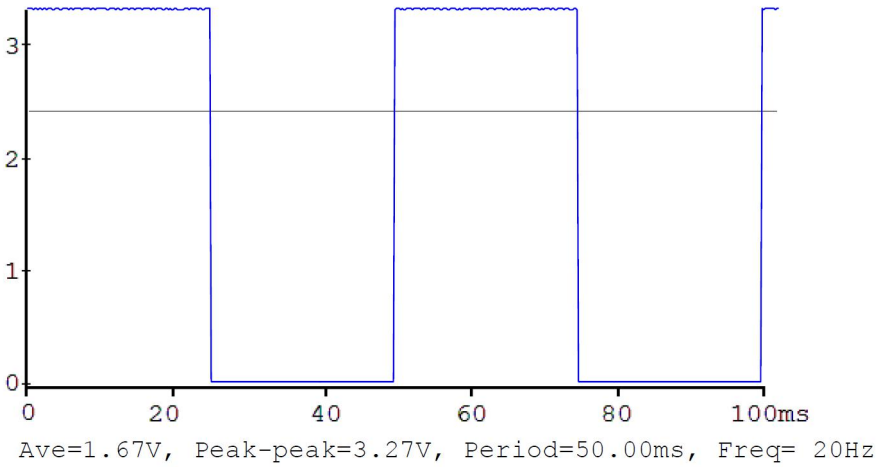


Figure 14.10. Zoomed out debugging screen shows SysTick interrupting every 25 ms on the real board.

Part h) This step is educational but will not be graded. Use the system to collect another five data points, creating a table showing the true position (t_i as determined by reading the position of the hair-line on the ruler), and measured position (m_i using your device). Calculate average error by calculating the average difference between truth and measurement. Because of the difficulties in calibration we expect the average error to be around 0.1 cm on the real board.

True distance t_i	Measured distance m_i	Error $\lvert t_i - m_i \rvert$

Table 14.2. Accuracy results of the distance measurement system.

You do not need a display to pass the real-board grader. However, you must connect PD3 to the analog input, potentiometer pin 2. There are three steps to the real board grader: 0) Initialization tests will look specifically to make sure SysTick is interrupting at 40Hz. In particular, there is only one value the grader will accept for the SysTick RELOAD register; 1) The grader will check to see if the ADC is initialized properly and the channel you have selected in the dialog window matches the channel you configured the ADC. You must use ADC0. However, you may use channel 0, 1, or 8. You must use sequencer 3, and software start;

2) The grader will measure the linearity of the system, which is the measured position (the value you put into Distance) as a function of the voltage on PD3 (measured by the grader). You must store your position measurements into the global **Distance**. You will be asked to move the slide pot to five different positions. It is important that these five positions be difference, but they do not need to vary by a constant distance. Every time the grader asks for a new point, move the slide pot either up or down. When you have moved the slide pot to a new position you will push the SW1 button on the LaunchPad and the grader will collect a (voltage, Distance) point. After five points are collected, the grader will calculate the correlation coefficient of the linear regression on these points, r^2. To pass your system must be $r^2>0.96$. Program 14.2 shows the function the real board grader used to calculate r2

```
//  http://en.wikipedia.org/wiki/Simple_linear_regression
unsigned long checkLinearity( long x0,  long y0,
                              long x1,  long y1,
                              long x2,  long y2,
                              long x3,  long y3,
                              long x4,  long y4){
  long r2, sumx, sumy, sumxy, sumx2, sumy2, n;
  long numerator25, denominator0_25;
  // divide data by 2^n until the numerator is less than sqrt(2^31-1) = 46,340
  n = 0;
  sumx = (x0 + x1 + x2 + x3 + x4);
  sumy = (y0 + y1 + y2 + y3 + y4);
  sumxy = ((x0*y0) + (x1*y1) + (x2*y2) + (x3*y3) + (x4*y4)); // sum of x*y
  sumx2 = ((x0*x0) + (x1*x1) + (x2*x2) + (x3*x3) + (x4*x4)); // sum of x^2
  sumy2 = ((y0*y0) + (y1*y1) + (y2*y2) + (y3*y3) + (y4*y4)); // sum of y^2
  numerator25 = 5*sumxy - sumx*sumy;
  while(((numerator25 > 46340) || (numerator25 < -46340)) && (n < 8)){
    x0 = x0>>1; x1 = x1>>1; x2 = x2>>1; x3 = x3>>1; x4 = x4>>1;
    y0 = y0>>1; y1 = y1>>1; y2 = y2>>1; y3 = y3>>1; y4 = y4>>1;
    n = n + 1;
    sumx = (x0 + x1 + x2 + x3 + x4);
    sumy = (y0 + y1 + y2 + y3 + y4);
    sumxy = ((x0*y0) + (x1*y1) + (x2*y2) + (x3*y3) + (x4*y4)); // sum of x*y
    sumx2 = ((x0*x0) + (x1*x1) + (x2*x2) + (x3*x3) + (x4*x4)); // sum of x^2
    sumy2 = ((y0*y0) + (y1*y1) + (y2*y2) + (y3*y3) + (y4*y4)); // sum of y^2
    numerator25 = 5*sumxy - sumx*sumy;
  }
  denominator0_25 = (5*sumx2 - sumx*sumx)*(5*sumy2 - sumy*sumy)/100;
  r2 = numerator25*numerator25/denominator0_25;
  return r2;
}
```

Interesting questions (things to think about but NOT implement in your lab)

How does the Nyquist Theorem apply to this lab?

What factors affect accuracy?

What will happen if you increase your sampling rate 100 times faster?

It is bad style to perform display output in the SysTick ISR because performing output wastes time and prevents other interrupts from being serviced.

Lab 15. Systems-Level Approach to Embedded Systems

Preparation

You will need a LaunchPad, a slide potentiometer, two switches, two LEDs, and the appropriate resistors to interface the switches and LEDs. If you have the Nokia 5110 you may use it. The Nokia 5110 LCD is optional.

Book Reading Sections 2.1-2.5, 6.1-6.4, 8.4, 9.1-9.9, and 10.1-10.5 in Volume 1 Embedded Systems

Starter project

Lab15_SpaceInvaders

Purpose

This lab has these major objectives: 1) learn modular design by putting all the components of the class into one system; 2) the development of a second ISR using one of the timers, creating three threads; 3) learn how to represent and maniputate images as two-dimensional arrays; 4) design, test, and debug a large C program; and 5) review I/O interfacing techniques used in this class. An addition component of this lab will be code review. After May 7, you will be able to view the YouTube videos. After May 14, you will be able to download and run code written by other students. This will allow you to observe the good and not so good practices of your fellow students. You can add compliments and constructive feedback to their YouTube video.

System Requirements

Refer back to Section 15.1, Requirements Document, for a general description of the game. In this section we will list detailed specifications needed to be able to share your game with others.

- It must run on a real LaunchPad, or run in the simulator with LaunchPadDLL version 1.0.0.6 (or higher)
- It must compile in Keil using the either the **Lab15_SpaceInvader** or **Lab15_VirtualSpaceInvaders** project simply by copy-pasting your code into the SpaceInvaders.c file, without changing any of the other files in the project, and without adding any additional files to the project. Your project may use 1) the simulator, 2) the real Nokia or 3) the virtual Nokia (UART+TExaSdisplay).
- The compilation must occur with the 32k-limit free version of Keil
- The slide pot must be attached to PE2/AIN1
- Two buttons must be attached to PE1 and PE0
- Two LEDs must be attached to PB5 and PB4
- The 4-bit DAC must be attached to PB3-0
- If you use the real Nokia5110, it is interfaced to PA7, PA6,PA5,PA3,PA2

- *If you use the virtual Nokia5110, then TExaSdisplay version 1.08 (or higher) must be running*

Please write a little code, and then test it. Then write a little bit more code and test it again. If you write a lot of code, and then try to test it, you could get hopelessly lost.

Also remember the free version of Keil limits the memory image (RAM+ROM) to be less than 32k. So, you will have to keep your images and sounds small. You can see the percentage of the 32k limit when you start the debugger. For example, the starter project itself uses 28% of the available space.

```
*** Currently used: 9336 Bytes (28%)
```

There is an option for those students who do not have a real Nokia. Start with the **Lab15_VirtualSpaceInvaders** project which is included as step 8 at

http://edx-org-utaustinx.s3.amazonaws.com/UT601x/download.html
For more details on the virtual display, see the section on debugging on the real board.

There is a **psychology** for developing games that are fun to play. Fun games are easy to learn but exciting to play. Successful games give the illusion of being difficult or dangerous, but at the same time the game is actually not as hard to play as it seems. If you want users to like your game, you need to let them "win" without letting them know you are letting the win. Successful games also invoke positive reinforcements in the player. One type of positive feedback is called **consistent reinforcement**, which are good things that always happen when one plays the game. A second type of positive feedback is called **inconsistent reinforcement**, meaning occasionally something wonderful happens.

Part a) The entire game can be debugged in simulation using the LaunchPadDLL version 1.0.0.6 or higher. Unfortunately the speed of the game will be much slower than running on the real board. However, just like the other labs, debugging in simulation first allows you to debug software without worrying if the hardware works. The next video shows me running my game in the simulator. This program runs about 10 times slower than it does on the real board. So once I got it to run in the simulator, I had to re-tune all the timings for the game to run on the real board.

In a manner similar to Lab 14, we suggest you develop and test Lab 15 in a modular fashion. For examples, each of the modules could be independently developed and tested, before you begin combining modules into the system.

- 11kHz interrupts with DAC to play sounds
- 30 Hz interrupt with ADC input and semaphore link to main program
- Switch input
- LED output
- Low level Nokia LCD
- Drawing sprites

One approach is to develop and test each module on the simulator. Once the modules are tested you can put them together. After the game runs completely on the simulator, you switch over and debug on the real board.

A second approach is to develop and test a module in the simulator. Once the module works on the simulator, test that module on the real board. Either way we suggest you add only 5 to 10 lines of code and then test.

Part b) The hardware for Lab 15 is essentially a combination of Labs 13 and 14, plus two switches on PE1 and PE0 and two LEDs on PB5 and PB4. Once you have the hardware built, it makes sense to rerun the Lab 13 real board grader and Lab 14 real board grader to make sure the DAC and ADC both still work. The new hardware is the Nokia 5110 display. There are three options

If you have a real Nokia, then you can interface it to PA7, PA6, PA5, PA3 and PA2. See the comments in the Nokia5110.c for how to wire it up. Notice there are multiple versions of the display (blue and red). Make sure you are connecting signals by their name on not by pin number.

You can of course develop and run the game on the simulator. Since Lab 15 is completely optional, feel free to debug it on the simulator and stop. You will need a screen capture application to create a YouTube video to upload for the competition.

There is a version of the Nokia5110.c driver that creates a virtual display. If you choose this option, first debug the game in the simulator, and then when you are ready to switch over to the real board and use the virtual display. All the function calls are the same, so you can use the same **Nokia5110.h** header file. For the virtual display, the graphics commands are streamed out the UART to your PC. Because of optimization, the virtual display actually runs faster than the real display. You must be running TExaSdisplay version 1.08 or higher to use the virtual display. Open the COM port and select the Nokia option to observe the graphics generated by your program. The cool part of this option is games developed on the real Nokia can be played on this virtual Nokia, and games developed on this virtual Nokia can be played on the real Nokia.

LaunchPad	Nokia 5110
PA7 Reset	RST
PA3 SSI0Fss	CE
PA6 Data/Command	DC
PA5 SSI0Tx	Din
PA2 SSI0Clk	Clk
3.3V power	Vcc
not connected	back light BL
Ground	Gnd

If you wish to use the back light, you could connect 3.3V through a 200-470 Ω resistor to the back light.

If you have a Nokia 5100 LCD you can skip the sections on the Virtual Nokia. However, if you do not have a Nokia 5110, you can use the virtual display. Download the starter project for the virtual Nokia as step 8 at

http://edx-org-utaustinx.s3.amazonaws.com/UT601x/download.html

The virtual Nokia system does not run in the simulator, so the proper approach is to use regular Nokia 5110 software and debug in the simulator. After it runs in the simulator switch over and debug in the the project on the real board with the virtual Nokia. Look inside the VirtualNokia5110.c file on line 37 you will find this code

```
#define VIRTUAL_NOKIA 0
```

This line defines the hardware configuration. Specify 0 to simulate or to use a real Nokia 5110 LCD. Specify 1 in line 37 to activate the virtual Nokia. In particular, to activate the virtual Nokia feature change line 37 to

#define VIRTUAL_NOKIA 1

In virtual mode, the LaunchPad is connected to PC through debugging USB cable and TExaSdisplay is running on PC with COM port open. The following video demonstrates the usage of the virtual display.