Lab 2 Performance Debugging

Table of Contents

---

0 Repository Structure

The typical explanation for the repo structure. Lab specific instructions can be found further below.

0.1 HW

The HW folder should contain your schematic and board files for your PCB or circuits. In labs 1-5 and 10, you will be creating schematics for your circuit in EAGLE. A setup tutorial can be found here.

0.2 SW

The SW folder should contain your application firmware and software written for the lab. The SW/inc folder contains firmware drivers written for you by Professor Valvano. Feel free to write your own (in fact, in some labs, you may be required to write your own).

You can place any other source files in the SW/ folder. TAs will look at the files you create and/or modify for software quality and for running your project.

0.3 Resources

A couple files are provided in the Resources folder so you don't have to keep searching for that one TI document. Some of them are immediately useful, like the TM4C datasheet. Others may be useful for your final project, like the TM4C_System_Design_Guidelines page.

0.4 Git and Github

We will extensively use Git and Github for managing lab projects. This makes it easier for TAs to grade and help debug the project by allowing us to see commit histories, maintain a common project structure, and Likewise, it makes it easier for students to collaborate with partners, merge different codebases, and to debug their work by having a history of commits.

Two common ways of using Git and Github are Github Desktop and the command line. Tutorials are also abundant on the net for you to peruse. We've provided a cheatsheet for git in the Resources folder.

It is highly recommended to make the most out of Git, even if you've never used it before. Version control will save you a lot of suffering, and tools like Git or SVN are ubiquitous in the industry.

A gitignore file is added to the root of this repo that may prevent specific files from being tagged to the repo. This are typically autogenerated output files we don't care about, but sometimes other stuff (like .lib files) falls through that we want. Feel free to modify if necessary.

---

1 Summary

1.1 Goal

In this lab we will introduce various debugging techniques. In particular we will look at how to use the oscilloscope, spectrum analyzer, and logic analyzer, and then learn how to profile code using dumps to measure intrusiveness and noise.

You should understand the following concepts by the end of the lab:

- real-time systems
- time jitter
- critical sections
- bit banding and shared resources
- probability mass functions (PMF)
- central limit theorem (CLT)

1.2 Team Size

The team size for this lab is 2.

> Two shall be the number thou shalt count, and the number of the counting shall be two. Three shalt thou not count, neither count thou one, excepting that thou then proceed to two. Four is right out.

2 Preparation

1. Clone the GitHub repository:
    - If a GitHub classroom has been created, accept the assignment and clone the repository onto your local device. Push your edits to your private repository.
    - If you're cloning from the public organization, clone the template. Submit a zip file containing the entire repository to Canvas.
2. Implement the functions defined in `Dump.h`. Feel free to change the API as you see fit.
3. Debugging with the logic analyzer and oscilloscope:
    1. If you have access to a real logic analyzer and oscilloscope, you will use main programs `main1` and `main3` (which do not activate TExaS).
    2. If you **do not** have access to a real logic analyzer and oscilloscope, you will use main programs `main0`, `main2`, and `main4` (which will activate TExaS).
    3. Reading `TIMER1_TAR_R` will return the 32-bit current time in 12.5ns units. The timer counts down. To measure elapsed time, we read `TIMER1_TAR_R` at the start of the elapsed time measurement and read it again at the end of the elapsed time measurement. Next, we subtract the second measurement from the first. 12.5ns * 232 is 53 seconds. So, this approach will be valid for measuring elapsed times less than 53 seconds. The time measurement resolution is 12.5 ns.
4. The microcontroller is executing at 80 MHz. The following shows a small section of the C code and resulting assembly code generated by the compiler for the while loop in `main1` and `main2`.

```
0x00000D98 481F      LDR     r0,[pc,#124] ; @0x00000E18
0x00000D9A 6880      LDR     r0,[r0,#0x08]
0x00000D9C F0800002  EOR     r0,r0,#0x02
0x00000DA0 491D      LDR     r1,[pc,#116] ; @0x00000E18
0x00000DA2 6088      STR     r0,[r1,#0x08]
```

```
while (RealTimeCount < 3000) {
    PF1 ^= 0x02;
```

```
0x00000DA4 4812      LDR     r0,[pc,#72]  ; @0x00000DF0
0x00000DA6 6800      LDR     r0,[r0,#0x00]
0x00000DA8 491C      LDR     r1,[pc,#112] ; @0x00000E1C
0x00000DAA 4348      MULS    r0,r1,r0
0x00000DAC 491C      LDR     r1,[pc,#112] ; @0x00000E20
0x00000DAE FBB0F0F1  UDIV    r0,r0,r1
0x00000DB2 490F      LDR     r1,[pc,#60]  ; @0x00000DF0
0x00000DB4 6008      STR     r0,[r1,#0x00]
```

```
jitterVariable = (jitterVariable * 12345678) / 1234567;
```

```
0x00000DB6 4817      LDR     r0,[pc,#92]  ; @0x00000E14
0x00000DB8 6800      LDR     r0,[r0,#0x00]
0x00000DBA F64031B8  MOVW    r1,#0xBB8
0x00000DBE 4288      CMP     r0,r1
0x00000DC0 D3EA      BCC     0x00000D98
```

```
}
```

```
0x00000DF0 0014      DCW     0x0014
0x00000DF2 2000      DCW     0x2000
0x00000E14 0000      DCW     0x0000
0x00000E16 2000      DCW     0x2000
0x00000E18 5000      DCW     0x5000
0x00000E1A 4002      DCW     0x4002
0x00000E20 D687      DCW     0xD687
0x00000E22 0012      DCW     0x0012
```

Listing 1. Assembly Table.

This assembly code was obtained by observing the assembly listing in the debugger. You may see different assembly on your machine because of differences in the compiler version or optimization settings. You are allowed to solve the preparation with either this assembly or the assembly you see on your computer.

2.1 Lab Prep Questions

1. What are the purposes of the `DCW` statements? More specifically, what do these three constants mean: `0x20000014`, `0x40025000`, and `0x0012D687`?
2. Look at Section 3.3.1 (page 32) of the data sheet CortexM4_TRM_r0p1.pdf and find which instructions in the above while loop take more than 3 cycles to execute. Assume P=3 for the `BCC` instruction because it must refill the pipeline if it branches.
3. This while loop toggles PF1. Neglecting interrupts for this preparation question. Assuming assembly instructions take about 25 ns to execute, estimate how fast one cycle of the while loop would execute.

---

3 Procedure

3.1 Setup

1. Connect a constant analog voltage to an ADC input on PD3, PD2, PE2 or PB5. One option is to use a potentiometer, like Lab 8 in EE319K, Figure 2.1. Another option is to create 1.65V using two 10k resistors.

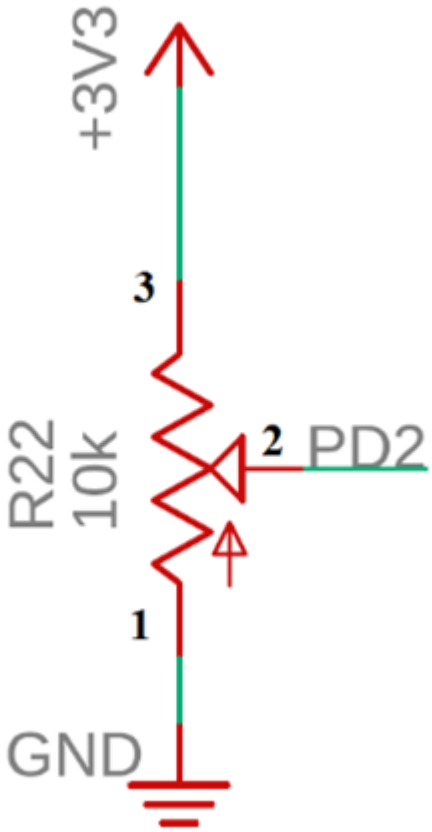

Figure 1. Possible hardware connection to create an analog input.

2. If using TExaS (and therefore mains `main/main0`, `main2`, and `main4`), edit the parameter for the call to `TExaS_Init` to specify your choice of channel.

```
// Parameters that can be passed into TExaS_Init based on HW configuration.

// TExaS.h
enum TExaSmode{
  SCOPE, // PD3
  SCOPE_PD2,
  SCOPE_PE2,
  SCOPE_PB5,
  LOGICANALYZERA,
  LOGICANALYZERB,
  LOGICANALYZERC,
  LOGICANALYZERE,
  LOGICANALYZERF,
  NONE
};
```

3. If not using TExaS, hook up an oscilloscope and/or logic analyzer to the ADC input. Make sure hook up the ground reference probe as well.

! ! ! Deliverable 1 ! ! !

Draw the electrical circuit you used to create the analog input.

3.2 Using the Oscilloscope, Spectrum Analyzer, and Logic Analyzer

You are expected to learn how to use these instruments in this class, so please ask your TA for a demonstration in lab if you are unfamiliar with them. You are provided the option to use the TM4C to emulate its own oscilloscope, spectrum analyzer, and logic analyzer through TExaS, with significant constraints on functionality and resolution.

Provided are two tutorials on using TExaS:

- TExaS Oscilloscope and Spectrum Analyzer
- TExaS Logic Analyzer

! ! ! **Deliverable 2** ! ! !

Use the oscilloscope to visualize and characterize the analog input of your circuit. In particular, capture and measure the noise of the signal. This can be done by measuring the AC RMS or peak-to-peak. Take a picture of the scope trace (screenshot or USB capture or phone picture) and add to the lab report.

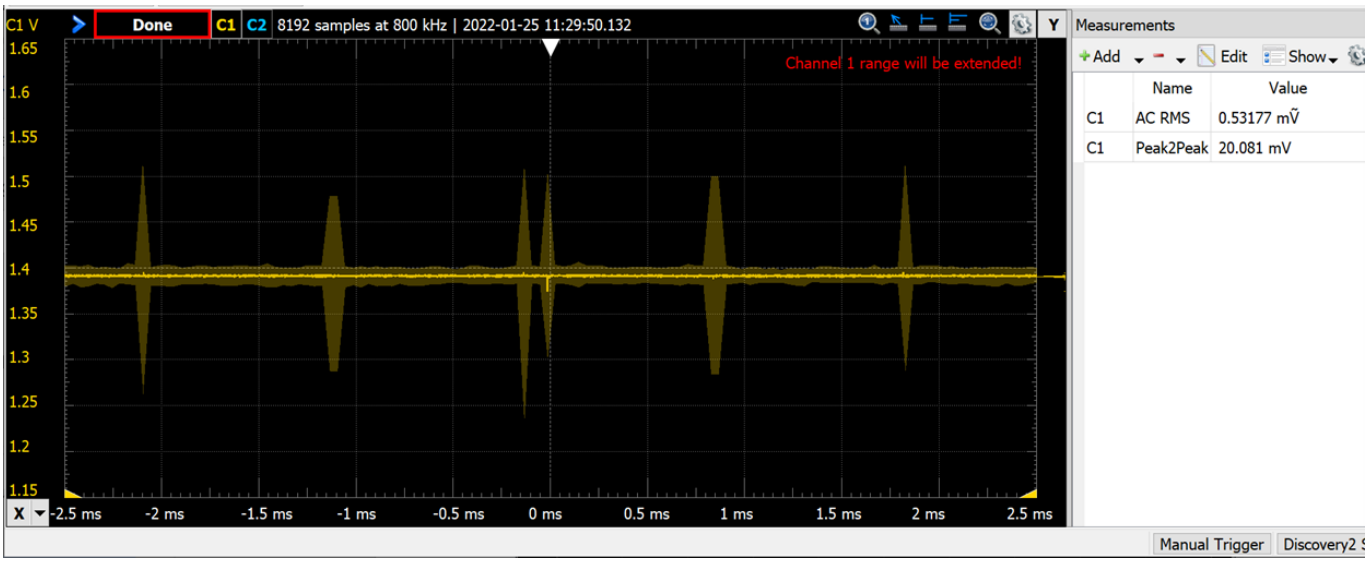

Figure 2. Analog voltage versus time measured with a real oscilloscope.

If using TExaS, an 8-bit analog signal on PD3 is sampled at 10 kHz and sent to the PC for plotting. To use the scope, connect the analog input to PD3. Be careful to limit the voltage between 0 and 3.3V, because PD3 is an unbuffered TM4C123 analog input. Run `main0`, which activates `TExaS_Init(SCOPE)`.

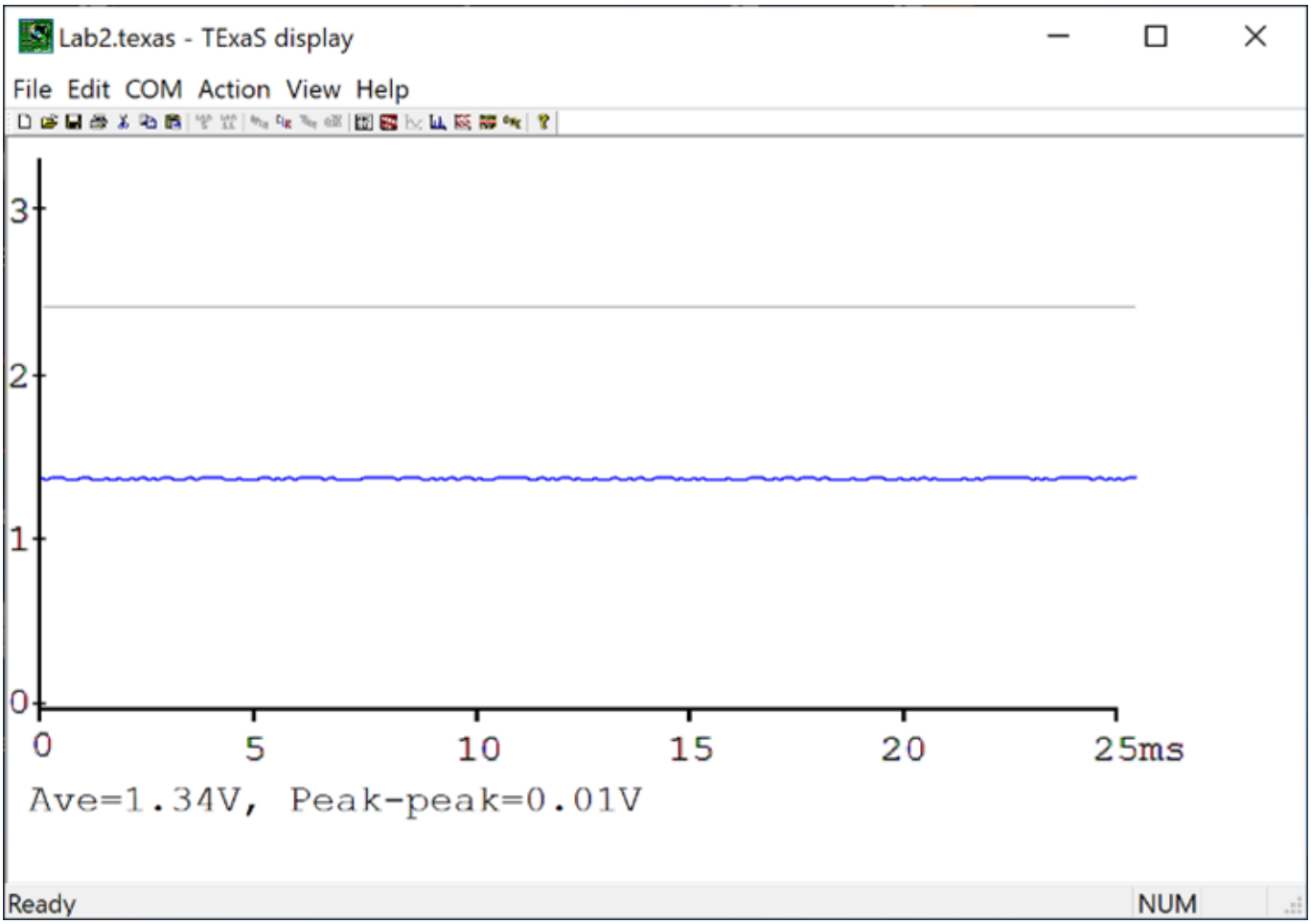

Figure 2b. Analog voltage versus time measured with the TExaS oscilloscope.

! ! ! Deliverable 3 ! ! !

Use the spectrum analyzer to measure amplitude vs frequency of the analog input of your circuit. Take a picture of the scope trace (screenshot or USB capture or phone picture) and add to the lab report.

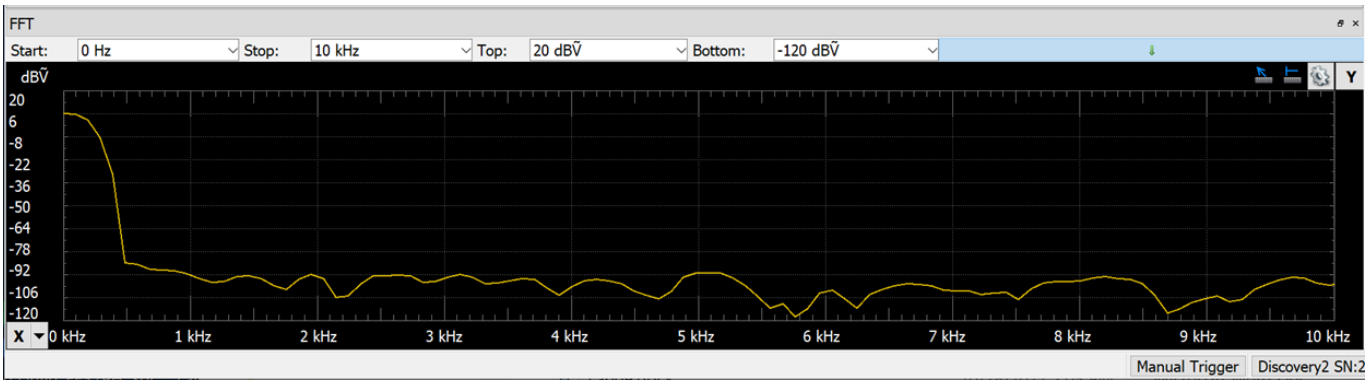

Figure 3. Analog voltage versus frequency measured with a real spectrum analyzer.

If using TExaS, follow the instructions in the TExaS Oscilloscope and Spectrum Analyzer video to select the spectrum analyzer from the view menu.

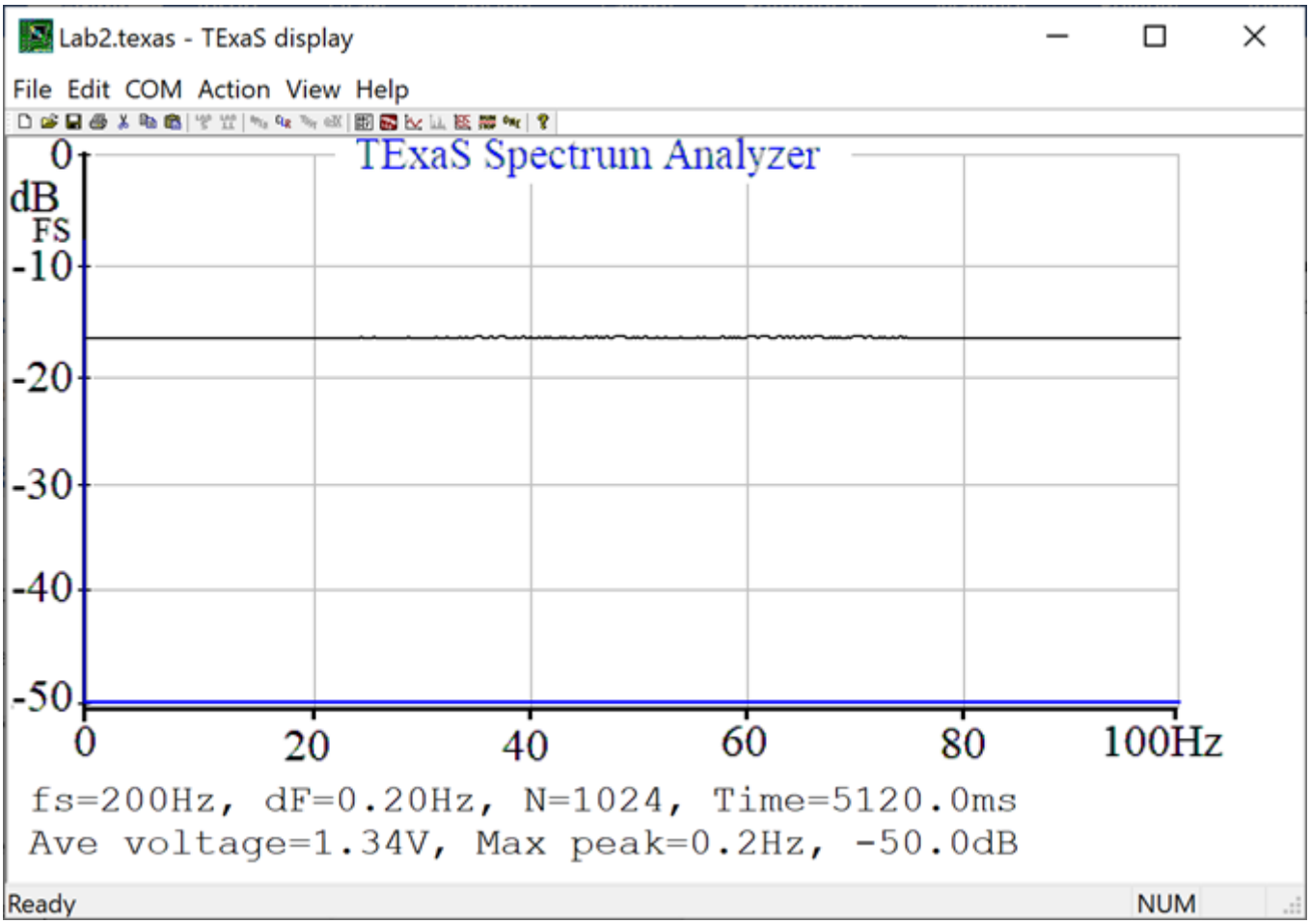

Figure 3b. Analog voltage versus frequency measured with the TExaS spectrum analyzer.

! ! ! Deliverable 4 ! ! !

Run `main3` (or `main4` if using TExaS) and observe PF3 (Timer2A ISR), PF2 (Timer0A ISR) and PF1 (main). See further below for more details on TExaS.

- Measure P0, the interrupt period for the Timer0A (should be 1/125Hz).
- Measure T0, the time to complete the Timer0A ISR (should be about 10us with `ADC0_SAC_R=0`).
- Calculate the Timer0A ISR utilization percentage. This is T0/P0.
- Measure P2, the interrupt period for the Timer2A (should be 1/1024Hz).
- Measure T2, the time to complete the Timer2A ISR (should be about 1us, depending on your `Jitter_Measure`).
- Calculate the Timer2A ISr utilization percentage. This is T2/P2.
- Calculate the total utilization percentage of the program. This is about 1-T0/P0-T2/P2.
- Also put some logic analyzer captures into the lab report.

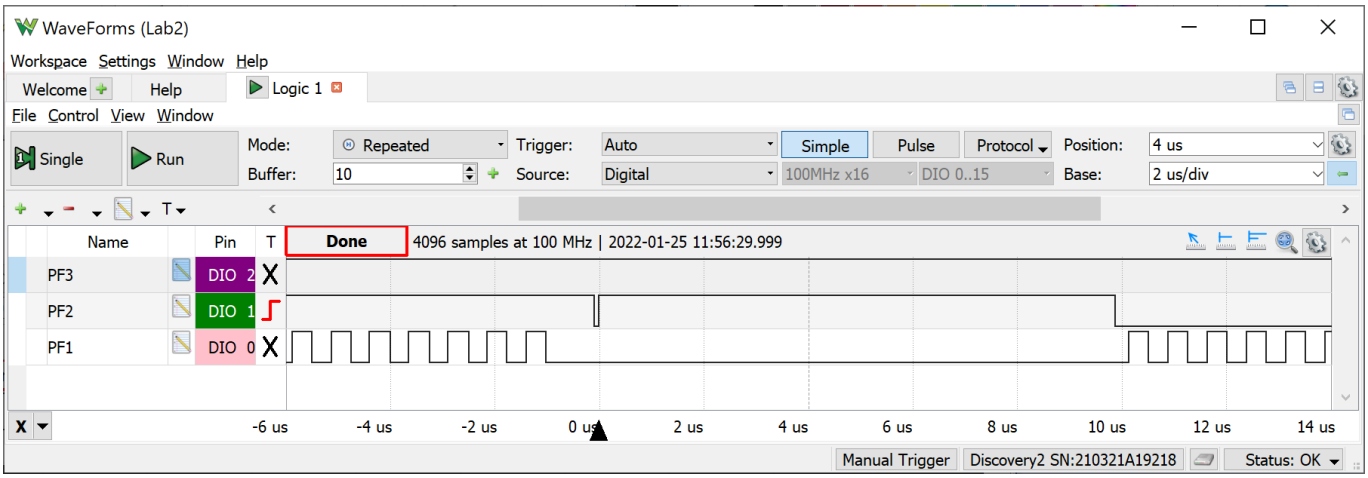

Figure 4. Zoomed in view of the PF1 PF2 PF3 recording to see a) the main program does not run while the Timer0A ISR is running and b) the time to execute the Timer0A ISR is about 10us (most of this 10us occurs converting the ADC) This recording was taken with ADC0_SAC_R=0.

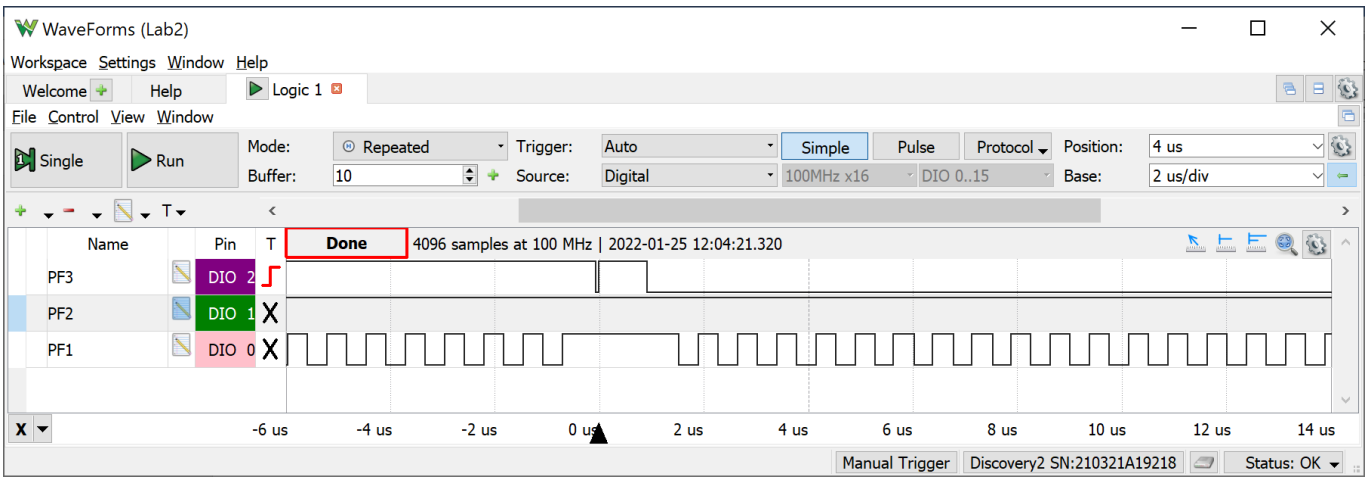

Figure 4b. Zoomed in view of the PF1 PF2 PF3 recording to see a) the main program does not run while the Timer2A ISR is running and b) the time to execute the Timer2A ISR is about 1us.

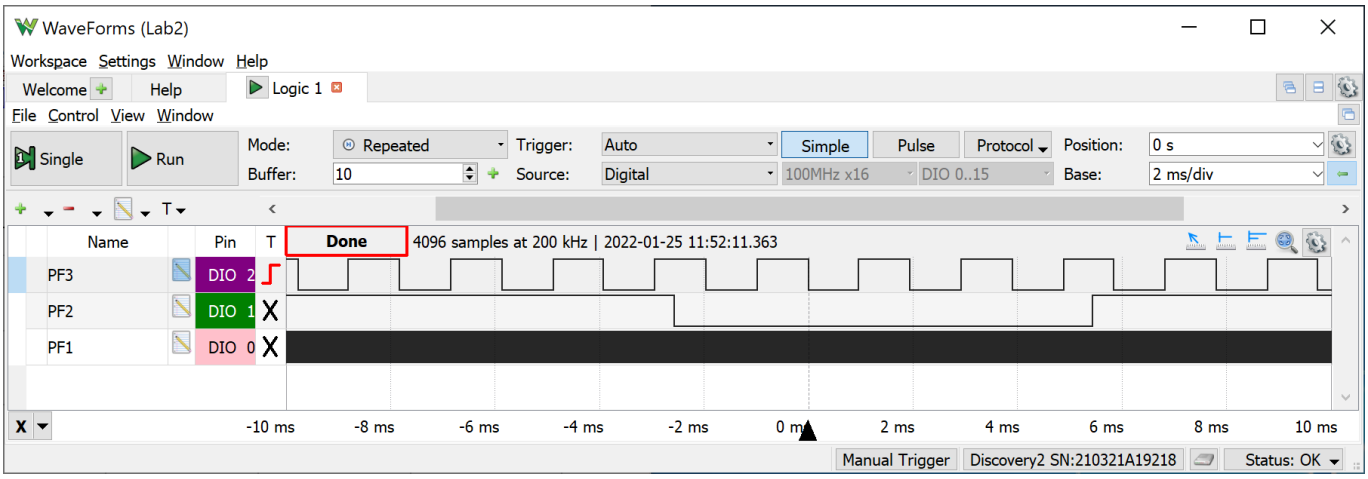

Figure 4c. Zoomed out view of the PF1 PF2 PF3 recording to see a) the Timer0A runs at 125 Hz, b) Timer2A runs at 1024 Hz, and c) most of the processor time is allocated to running the main program.

If using TExaS, the TExaS logic analyzer sends 7-bit data at 10 kHz to the PC for plotting. Run `main4`, which selects the logic analyzer on Port F. Notice the call to `TExaS_Init(LOGICANALYZERF)`. You do not have to make any hardware connections to utilize the logic analyzer. Since the priority of the TExaS interrupt is 5

(lower priority than the two ISRs in Lab 2), the triple toggles will always be seen as a single toggle. Observe PF3 (Timer2A ISR), PF2 (Timer0A ISR) and PF1 (main). Measure P0, the interrupt period for the Timer0A (should be 1/125Hz). The most accurate measurement of P0 is achieved by deriving it from F2, the frequency of channel 2 (PF2). P0 = 0.5/F2 (0.5/62.5 Hz=8ms in this figure). Assume T0, the time to complete the Timer0A ISR, is about 10us with `ADC0_SAC_R=0`. The percentage time in Timer0A ISR is T0/P0. Measure P2, the interrupt period for the Timer2A (should be 1/1024Hz). Similar, the most accurate measurement of P2 is achieved by deriving it from F3, the frequency of channel 3 (PF3). P2 = 0.5/F3 (0.5/511.6 Hz=0.977ms in this figure). The 0.5 in this equation results from the fact that each ISR toggles the output pin. Assume T2, the time to complete the Timer2A ISR, is about 1us. The percentage time in Timer2A ISR is T2/P2. The percentage time in the main program is therefore about 1 T0/P0-T2/P2. Notice the 10 kHz sampling rate of the TExaS logic analyzer cannot correctly capture the behavior of PF1.

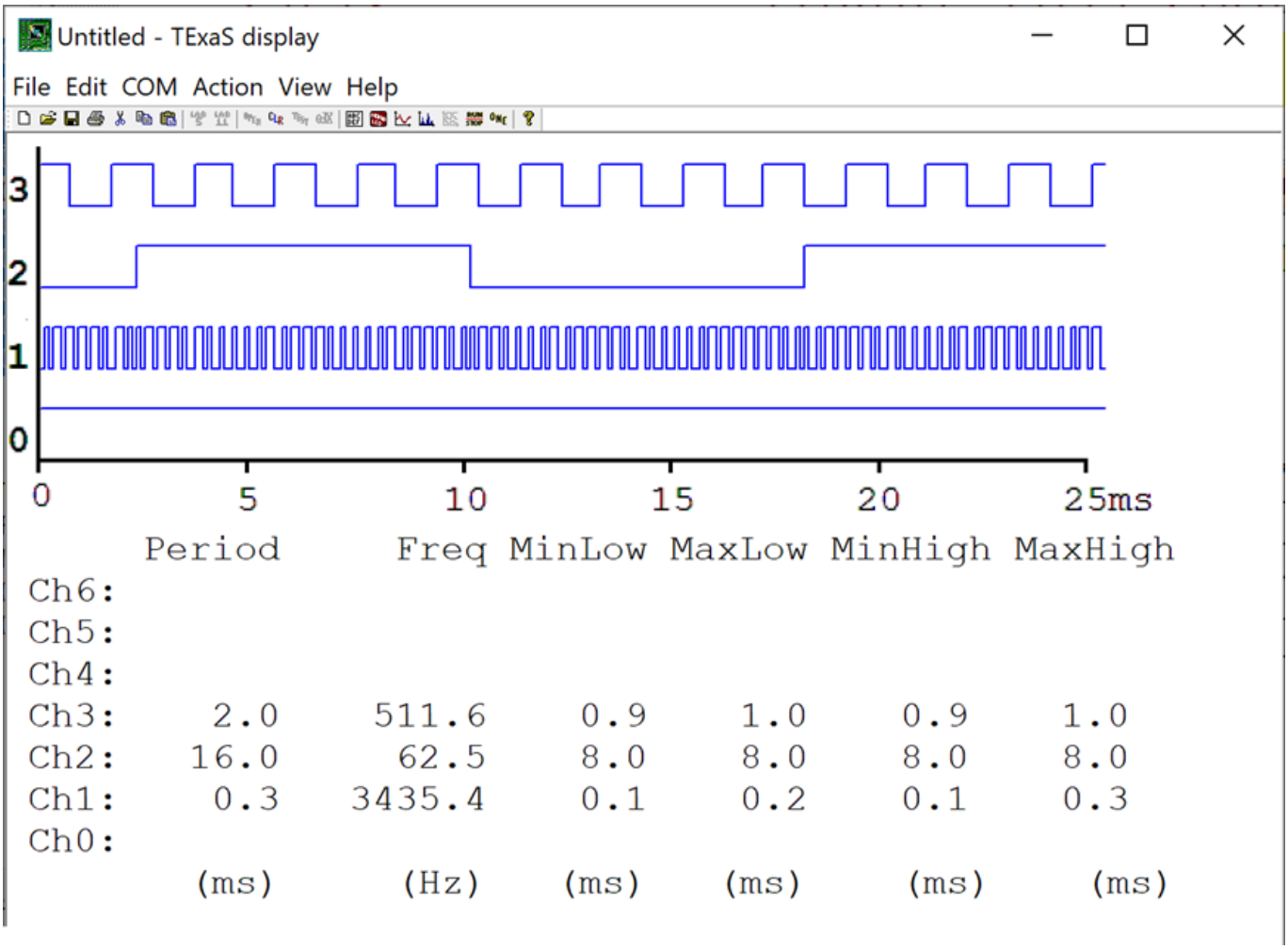

Figure 4d. Zoomed out view of the PF1 PF2 PF3 recording using the TExaS logic analyzer to see a) the Timer0A runs at 125 Hz, b) Timer2A runs at 1024 Hz, and c) most of the processor time is allocated to running the main program.

3.3 Debug `Dump.c` Functions and Prove the ADC Sampling is Real Time

! ! ! Deliverable 5 ! ! !

Measure the time jitter with just Timer2A (`main1` or `main2`). Explain what caused the small but non-zero jitter. Why would you classify Timer2A by itself as real time? Measure the time jitter with two ISRs (`main3` or `main4`).

Explain why Timer2A has a time jitter proportional to `2 * SAC`. Explain why the Timer0A jitter is close to zero. Why would you classify Timer0A as real time, but Timer2A is no longer real time?

Note: when we get to Lab 9, we will use timer-triggered ADC sampling, so that even with hardware averaging, all ISRs will be real time.

3.4 Evaluate Critical Sections

All three threads perform a read-modify-write access to Port F. Because of bit-specific addressing, these accesses are not critical. Change the accesses to use `GPIO_PORTF_DATA_R` instead of `PF1 PF2 PF3`, creating one or more critical sections. Critical sections create weird and unexpected behavior.

! ! ! Deliverable 6 ! ! !

Use any debugging technique to observe one instance of a critical section. Place the observation into your lab manual and explain the mistake the critical section created.

3.5 ADC Noise Measurements Using the Central Limit Theorem

To apply the Central Limit Theorem, we must assume the noise is random, the noise in each sample is independent from the noise in the other samples, and the noise has zero mean. Look up the ADC Sample Averaging Control (`ADC0_SAC_R`) register in the Chapter 13 of the data sheet. The Central Limit Theorem (CLT) states: as the number of samples increase, the calculated average (your data) will approach the theoretical mean (true signal). The CLT also states that regardless of the original probability density function (PDF) of the noise, the PDF of the averaged signal will become Gaussian.

Connect the constant voltage to the ADC input and run `main3` or `main4`. Since the input voltage is constant, the expected result would be all ADC data to be the same. Noise causes the variability. Observe the PMF of the noise as the program varies `ADC0_SAC_R` from 0 to 6. If you debug your software in the simulator, you should see all ADC data values the same. So, debug this part on the real board. You are allowed to adjust DUMPBUFSIZE to vary the number of points collected. *If you compare two PMFs with the same SAC value, you will not get the same result because the noise is not stationary.*

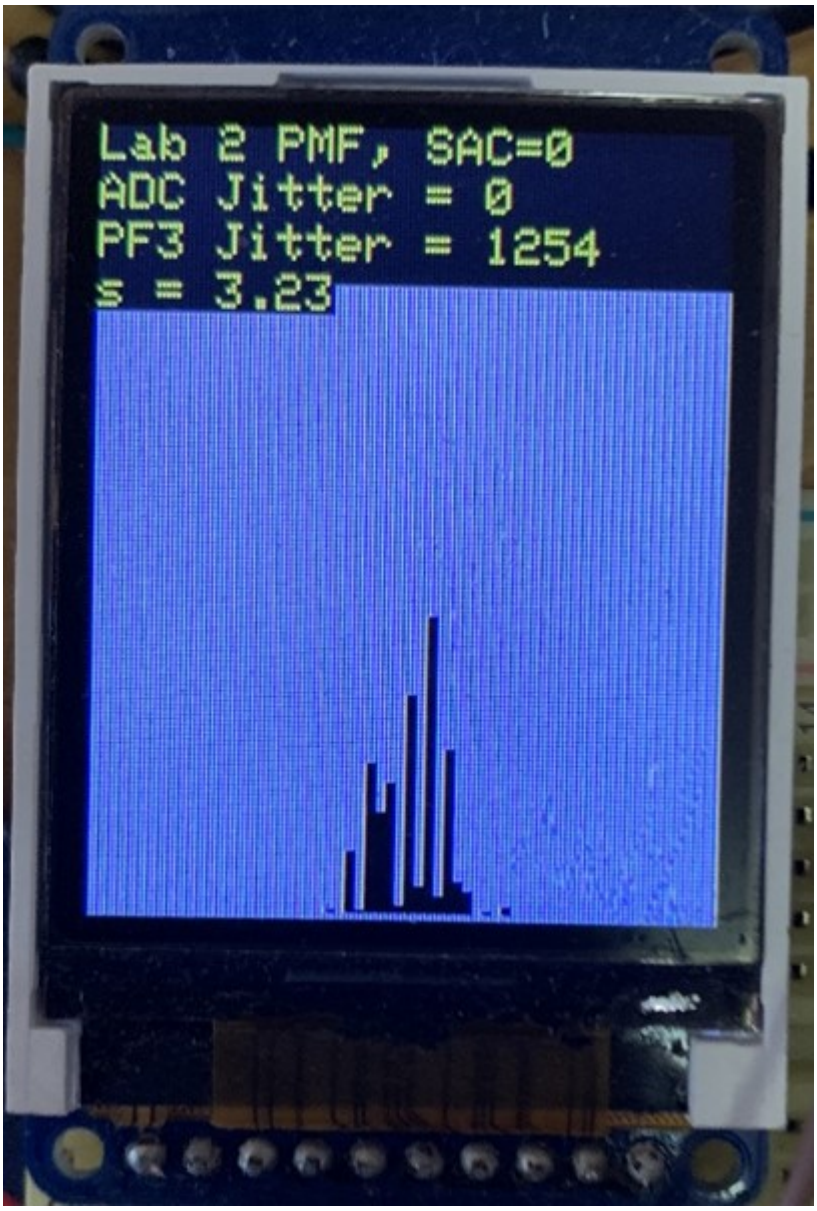

Figure 5. Photo of main3 output with a constant voltage applied to the analog input (SAC=0).

! ! ! Deliverable 7 ! ! !

Take four photos of the LCD screen PMF, like Figure 5, for hardware averaging of none, 4x, 16x, and 64x. In each case the sampling rate is fixed and there are DUMPBUFSIZE data points used to plot the PMF function. Describe qualitatively the effect of hardware averaging on the noise process. Consider two issues 1) the shape of the PMF and 2) the signal to noise ratio. *Hint: CLT*.

Fun activity: noise can vary, so before you generalize from the data you collected in this lab, go around the lab room, and look at the data from other groups.

3.6 Estimate the ADC Resolution

One simple estimate of the ADC resolution is standard deviation. Place a constant input on the ADC, sample the data multiple times and then calculate the standard deviation of the results. The data collected in Figure 6

shows the standard deviation of this data is about 3.23 samples. 3.23 samples are equivalent to 3.23*3.3/4096 ≈ 2.6mV. So, for SAC=0, we claim the ADC resolution is about 2.6mV.

The data in Figure 6 were collected with SAC=6. Conversely if the input were increased by only 0.5mV, the PMF distributions are not statistically different. For this data at SAC=6, we claim the ADC resolution is about 1mV. ECE445L does not expect you collect data like Figure 6.

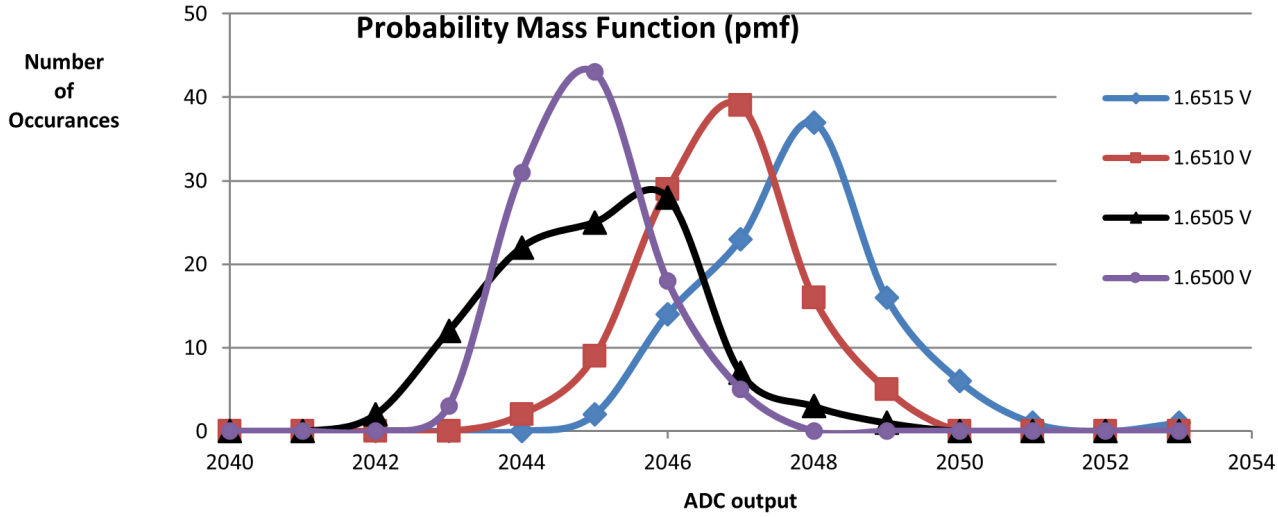

Figure 6. Probability mass function measured on the TM4C123 ADC with 64-point averaging.

! ! ! Deliverable 8 ! ! !

Estimate your ADC resolution with SAC=4 (16-point averaging).

---

4 Report

4.1 Deliverables

1. Objectives (1/2 page maximum). Simply repeat the items shown in the Goals section
2. Hardware Design (Deliverable 1)
3. Software Design (Dump.c and Dump.h)
4. Measurement Data (Deliverables 2,4-8) (3 is optional)

4.2 Analysis and Discussion (give short 1 or two sentence answers to these questions)

1. The ISR toggles PF2 three times. Is this debugging intrusive, nonintrusive or minimally intrusive? Justify your answer.
2. In this lab we dumped strategic information into arrays and processed the arrays later. Notice this approach gives us similar information we could have generated with a printf statement. In what ways are printf statements better than dumps? In what ways are dumps better than printf statements?
3. What are the necessary conditions for a critical section to occur? In other words, what type of software activities might result in a critical section?
4. Define “minimally intrusive”.

5. The PMF results should show hardware averaging is less noisy than not averaging. If it is so good, why don’t we always use it?