Ronald Macmaster and Parth Adhia

EID: Rpm953 and Pda375

09/05/2016

EE 445L – Lab 2: Performance Debugging

1.0: Objectives

We will practice various performance debugging techniques for real-time systems. Also, we will practice profiling our software with time and data dumps into arrays. Then, we will learn to use an oscilloscope and logic analyzer. We will also experience the concepts of real-time, probability mass functions, and the Central Limit Theorem. Finally, we will experiment with critical sections and gain a head-start on lab 3 by writing a line drawing function.

Our analysis of various performance debugging techniques will reinforce the importance of minimally intrusive debugging in real-time system design. We will study the impacts of intrusive, minimally intrusive, and nonintrusive debugging instruments on the real-time constraints of our system (print statements, data dumps, and hardware probes).

2.0: Software Design

We have uploaded our main program (lab2.c), ADCSWTrigger.c, Timer.h, and Timer.c files to canvas as software modules we worked on throughout this lab. Lab2.c contains all of our driver logic as well as the PMF, time jitter, and line drawing functions. We will collect data and time measurements from the ADC that will be stored in a shared pair of global arrays placed in RAM. The time jitter and PMF data will be calculated from our sample data. Our PMF plots and time jitter analysis can be found in the sections below.

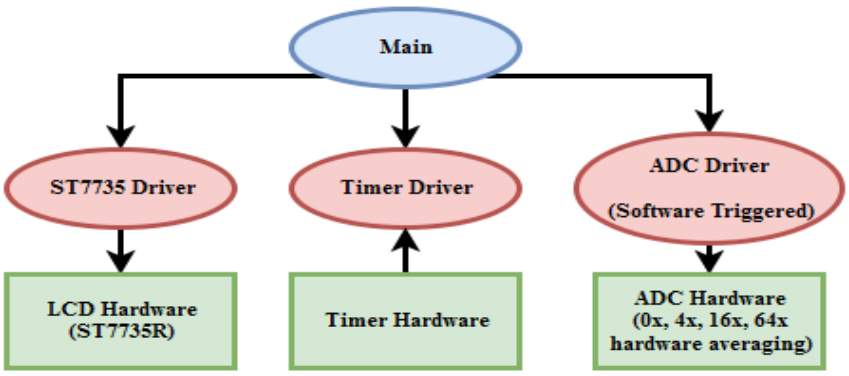


Figure 2.1: Call graph for our system. Most of the driver logic is in the main module.

3.0: Measurement Data:

3.1: Preparation Questions

a) What is the purpose of all the DCW statements?

DCW declares a 16 bit half word global variable in memory. These specific DCW statements define GPIO port addresses for PORT F access (system control, digital enable, data base, and directional).

b) The main program toggles PF1. Neglecting interrupts for this part, estimate how fast PF1 will toggle.

It takes 6 instructions to toggle the LED, and each instruction takes about 25 ns to execute. Thus, it takes about 6 * 25ns = 150ns to toggle the LED. The LED toggles at 6.7 MHz.

c) What is in R0 after the first LDR is executed? What is in R0 after the second LDR is executed?

After the first LDR, R0 contains the base address of GPIO Port F.
After the second LDR, R0 contains the current data value of PF1.

d) How would you have written the compiler to remove an instruction?

Instead of overwriting the Port F address (in R0) with the data value, load the data value of PF1 into R1 instead. Then, you can avoid reloading the Port F address into R1 (instruction 4).

e) 100-Hz ADC sampling occurs in the Timer0 ISR. The ISR toggles PF2 three times. Toggling three times in the ISR allows you to measure both the time to execute the ISR and the time between interrupts. See Figure 2.1. Do these two read-modify write sequences to Port F create a critical section? If yes, describe how to remove the critical section? If no, justify your answer?

No, these read-modify write sequences **do not** create a critical section.
PF2 is accessed through bit-specific addressing, and the ISR does not share PF2 with any other program threads. The value read into PF1 and modified in the main thread will always be accurate because it is unaffected by updates to PF2 in the ISR.

3.2: Debugging Profile (Oscilloscope)

Estimated sample time (interrupt, ADC, return from interrupt): 9.84 us

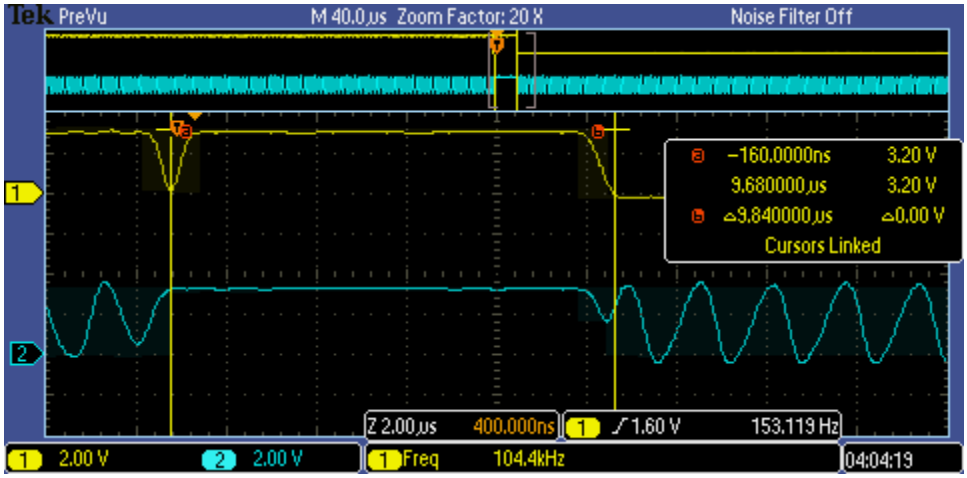


Figure 3.1): Oscilloscope profile of Timer ISR (ADC sampling).

3.3: Debugging Profile (Logic Analyzer)

Figure 3.2): Zoomed out logic profile of PF1 (channel 1) and PF2 (channel 2). The ISR runs every 10 ms

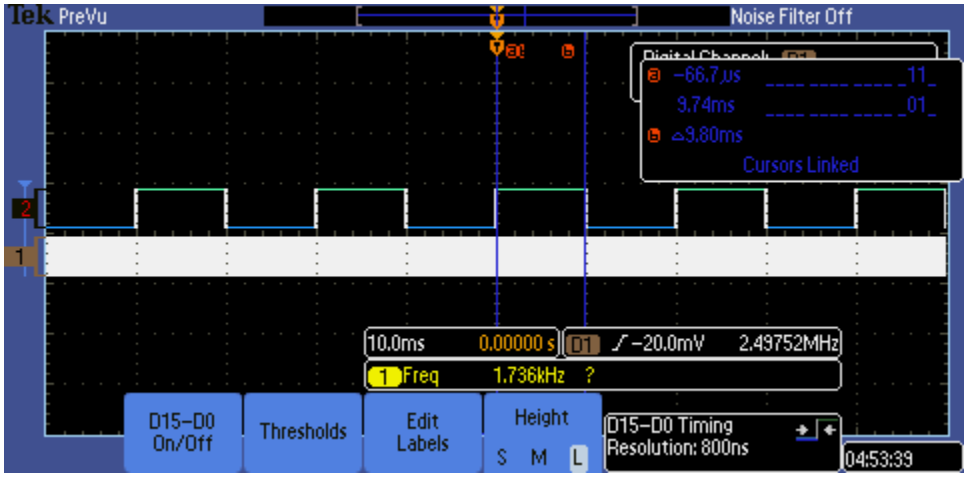
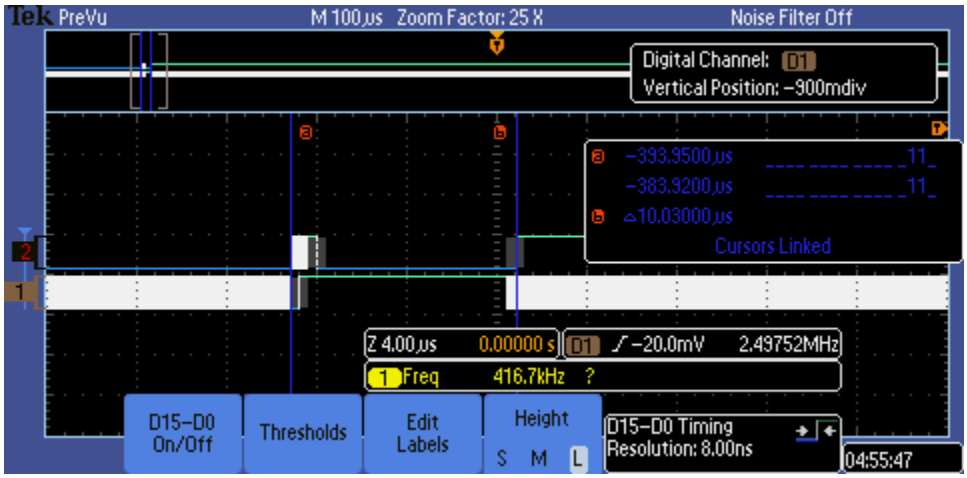


Figure 3.3): Zoomed in logic profile of PF1 (channel 1) and PF2 (channel 2).
Sampling the ADC and completing the ISR takes about 10us to run.



Estimate percentage of time running in the main versus running in the ISR.

% Time running in main = (time in main) / (time in ISR + time in main) * 100%

= (9.80 ms / (9.80ms +10.03 us)) * 100% = 99.89%

% Time running in ISR = (time in ISR) / (time in ISR + time in main) * 100%

= (10.03us / (9.80ms +10.03 us)) * 100% = 0.11%

3.4: Critical Section Analysis

Figure 3.4): Logic analyzer profile on our critical section.

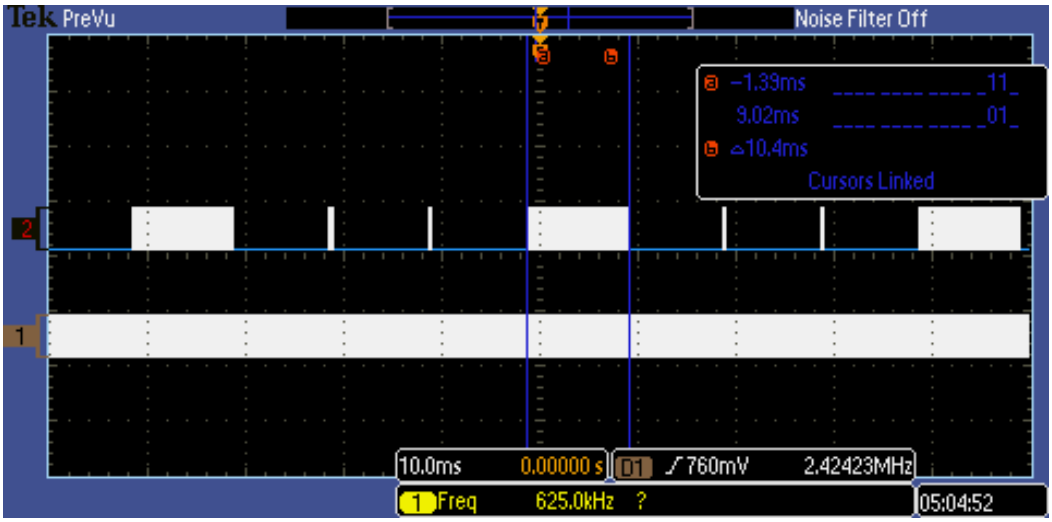


Figure 3.5): Logic analyzer profile on our critical section.

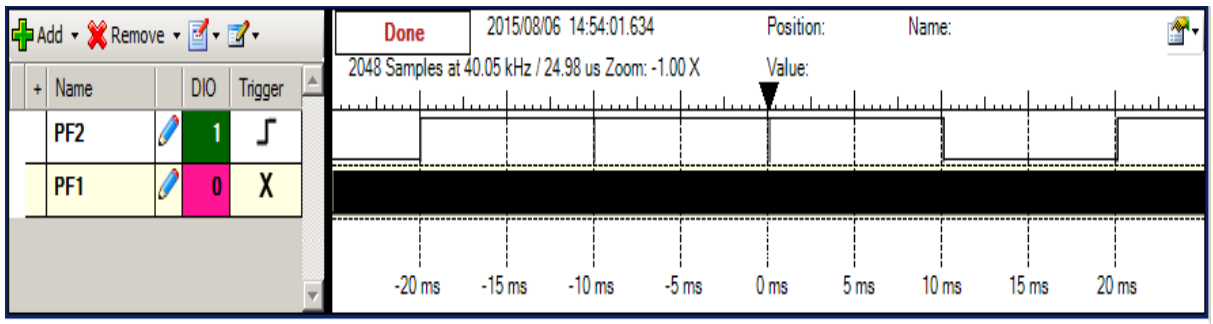


Figure 3.6): Assembly Language for Critical Section.

```
    75:                 GPIO_PORTF_DATA_R ^= 0x02;  // toggles wh
0x00000C30 480D       LDR             r0,[pc,#52]  ; @0x00000C68
0x00000C32 6800       LDR             r0,[r0,#0x00]
0x00000C34 F0800002   EOR             r0,r0,#0x02
0x00000C38 4904       LDR             r1,[pc,#16]  ; @0x00000C4C
0x00000C3A F8C103FC   STR             r0,[r1,#0x3FC]
0x00000C3E E7F7       B               0x00000C30
0x00000C40 387F       DCW             0x387F
0x00000C42 0001       DCW             0x0001
0x00000C44 34FF       DCW             0x34FF
0x00000C46 000C       DCW             0x000C
0x00000C48 0030       DCW             0x0030
0x00000C4A 2000       DCW             0x2000
```

Which pin is incorrect PF1 or PF2? Look at the assembly language of the main program and explain the sequence of steps that results in the corruption of the debugging profile. Other than using bit specific addressing (PF1 PF2) what other solutions could you have used to have two debugging profile pins without a critical section?

In our critical section diagram, **PF2 is the incorrect pin**. The vulnerable sequence occurs when the PC is between 0x0C32 and 0x0C3A (LDR r0, [r0, #0x00]; and STR r0, [r1, #0x3FC];). First, the main thread reads the data in PORTF. Then, the timer interrupts and the ISR is called, updating the LED profile on PF2. When execution returns to the main thread, R0 still contains the old value of PF2 in its PORTF data. Finally, the store instruction corrupts the current value of PF2.

There are many other things we could do besides bit-specific addressing to remove the critical section. Instead of using both profile pins on PORTF, we could place one on a different port and link it to an LED on a breadboard. We could also disable interrupts to make the sequence of events atomic. However, we would prefer that as a last resort to avoid design issues later on in development.

In General (Extra):

When two threads have shared access to global data or an I/O port, and one of the accesses is a write, then there exists a critical section in the lower priority thread. This usually occurs when the write access is a multistep, non-atomic sequence of read-modify-write, write-write, or write-read.

We can remove critical sections by removing shared access between threads or making the vulnerable sequences atomic. That involves utilizing bit-specific addressing, adjusting thread priority, packing global data, or disabling interrupts.

3.5: Time-Jitter Measurements

Measurements of jitter with 1 or 2 ISRs active and generalization of contributing factors to jitter.

Figure 3.7): Time Jitter measurement with only 1 ISR. (**Jitter = 0**).

Figure 3.8): Time Jitter measurement with 2 ISRs at a similar interrupt period. (**Jitter = 180**).

Try to generalize the results deriving a theoretical estimate of the time jitter of the periodic ADC sampling using software triggering.

The theoretical time jitter of the periodic ADC is equal to the time taken by all of the higher priority ISRs. In our case, we have one ISR that takes about 180 cycles to execute (for loop with 20 iterations, 9 cycles each).

3.6: PMF data and Analysis

Input voltage

DC Voltage: **1.66V** AC Voltage: **0.006V**

How would you describe the shape of the noise process?
When you run it over and over do you get the same shape of the PMF?
The noise process is bimodal in nature. As we plot the PMF repeatedly, we do not get the same shape of the PMF. The ADC noise it not stationary and varies from sample session to sample session.

Does your data support the CLT? If not, why?
Yes, as we increase the Hardware Averaging of the ADC, the data distribution approaches a Gaussian distribution. This is really evident when examining the 4X and no averaging figures in comparison to the 64X averaging figure.

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
As we increase hardware averaging, the ADC sampling noise decreases. We can infer this from the position of PMF, which shifts to lower values as hardware averaging increases. Also, the shape of the PMF approaches a Gaussian distribution as we increase hardware averaging. Also, the signal to noise ratio increases as hardware averaging increases.

Figure 3.8): ADC PMF with 0x hardware averaging. Figure is not Gaussian.

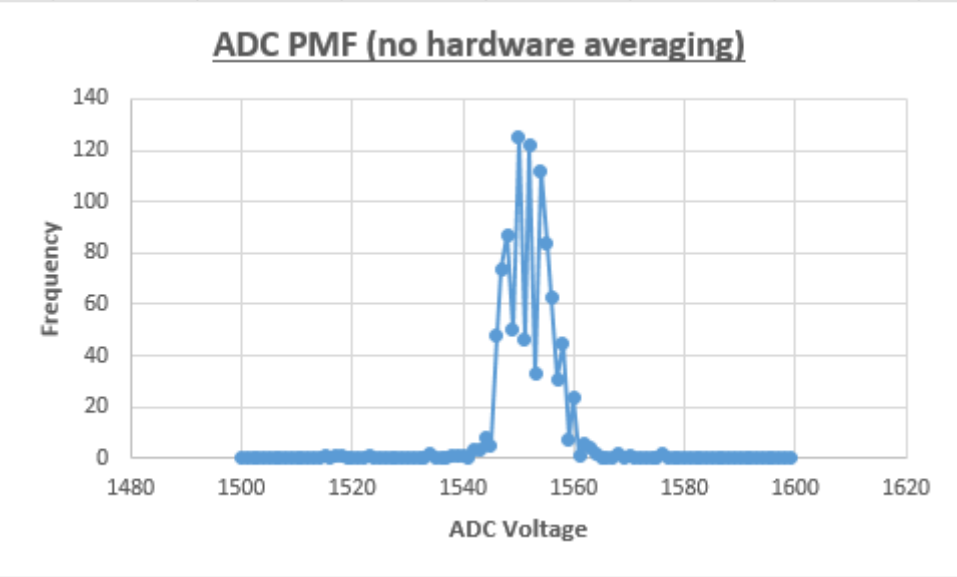


Figure 3.9): ADC PMF with 4x hardware averaging. Figure is still not Gaussian.

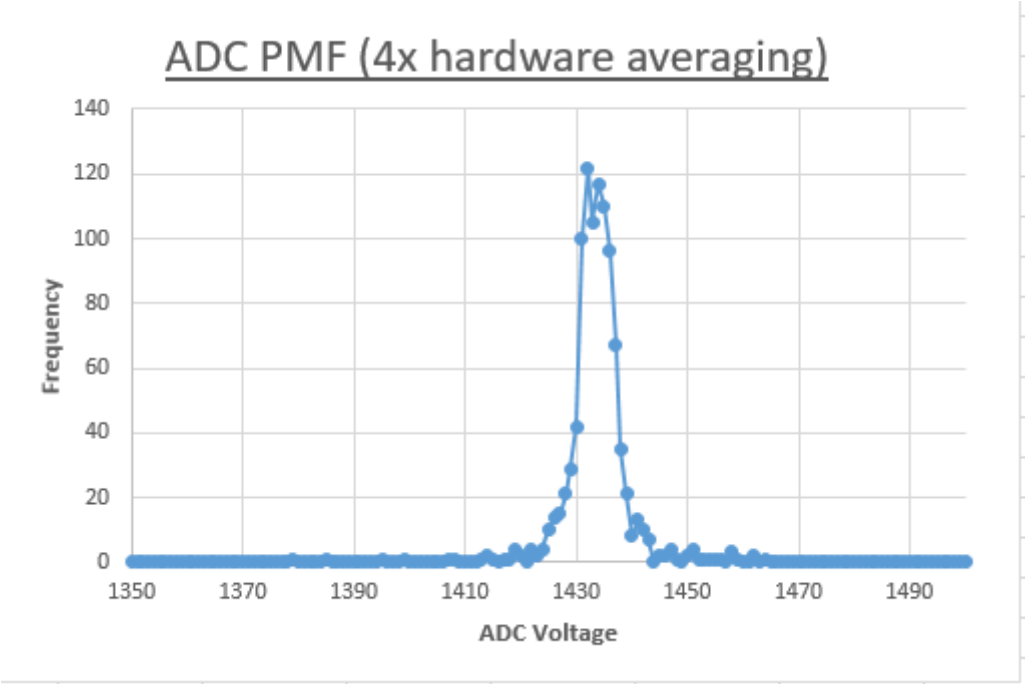


Figure 3.10): ADC PMF with 16x hardware averaging. Figure is a little more Gaussian.

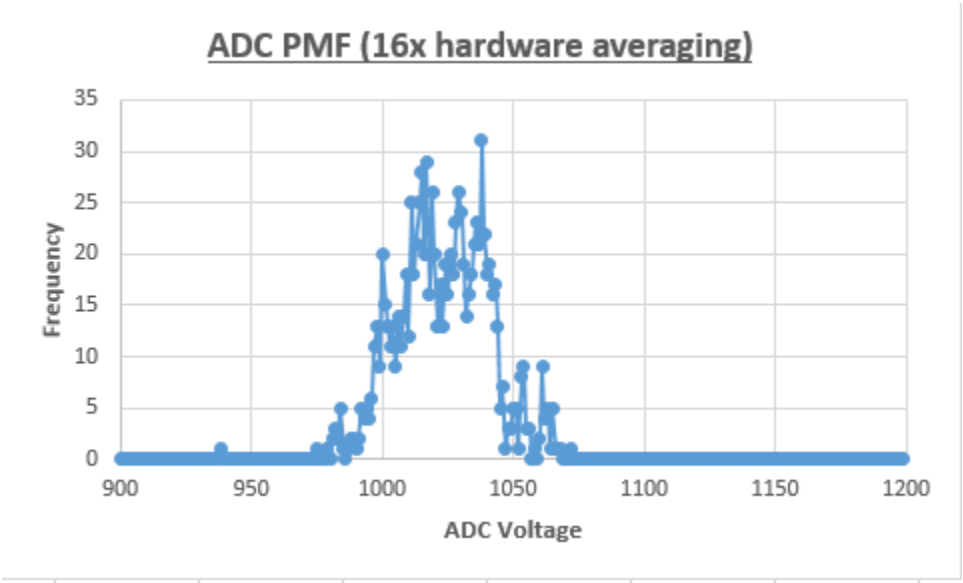
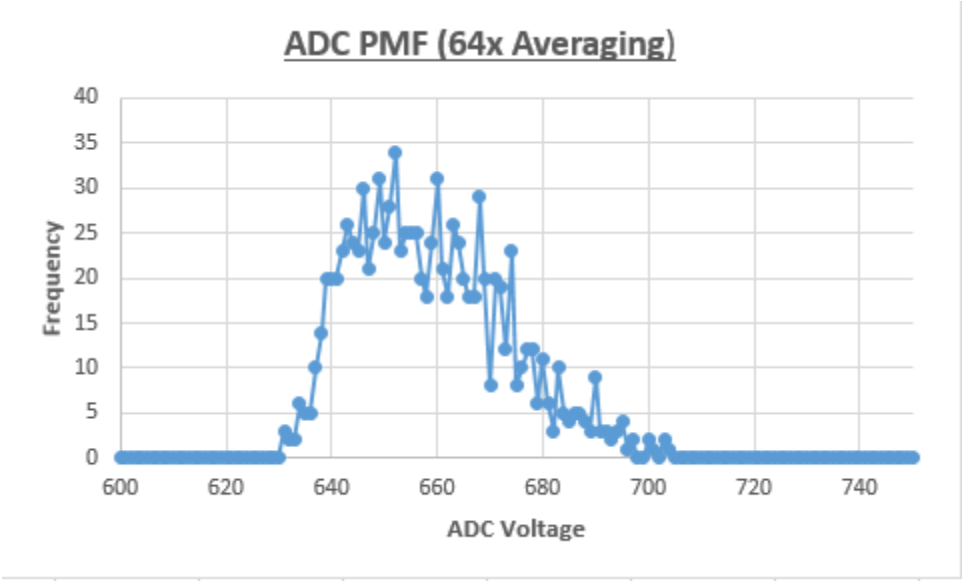


Figure 3.11): ADC PMF with 64x hardware averaging. Figure is a much more Gaussian.



3.6: ISR Execution with Hardware Averaging

Use the logic analyzer or scope to determine the effect of hardware averaging on the time to execute the ISR. Why is the thread profile like Figure 2.1 very different with hardware averaging?

Utilizing higher levels of hardware averaging will linearly increase the time to execute the ISR. Hardware averaging requires multiple samples from the ADC. Thus, a hardware averaging rate of 64X will require us to sample the ADC 64 times in the ISR. That means we can expect the ISR to run 64X slower.

Figure 3.12): ISR profile with 4x hardware averaging. Time in ISR = **34.6 us**

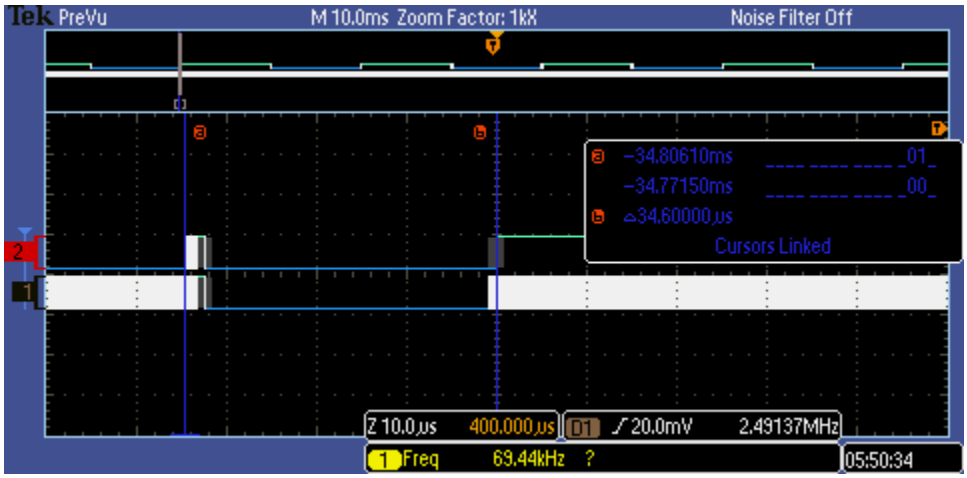


Figure 3.13): ISR profile with 8x hardware averaging. Time in ISR = **64.6 us**

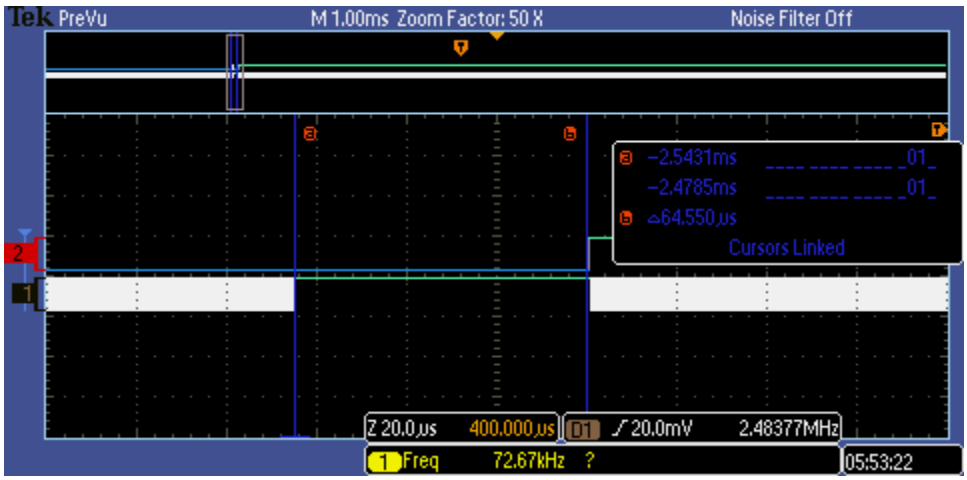


Figure 3.14): ISR profile with 16x hardware averaging. Time in ISR = **128.6 us**

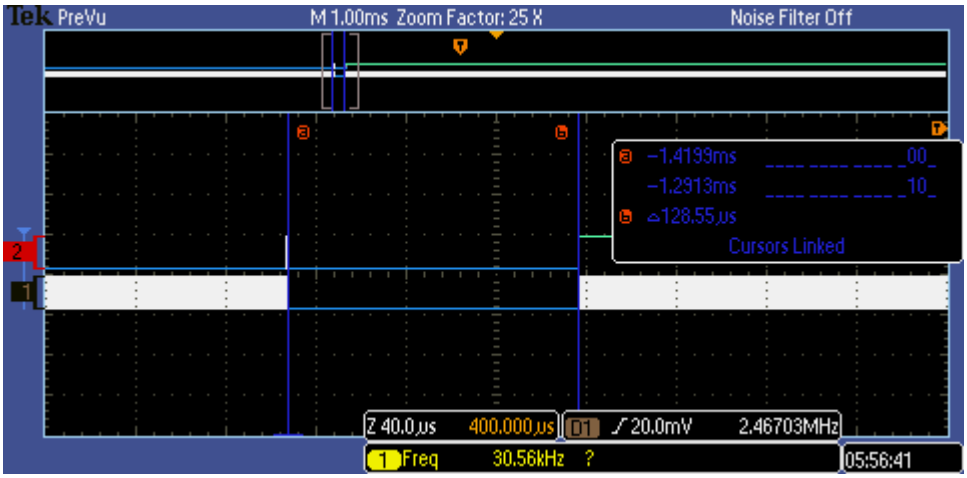


Figure 3.15): ISR profile with 32x hardware averaging. Time in ISR = **256.6 us**

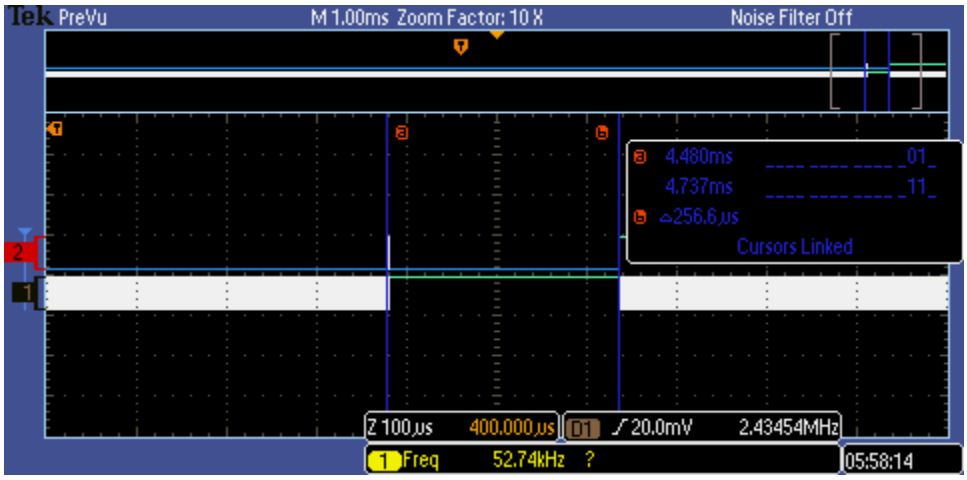
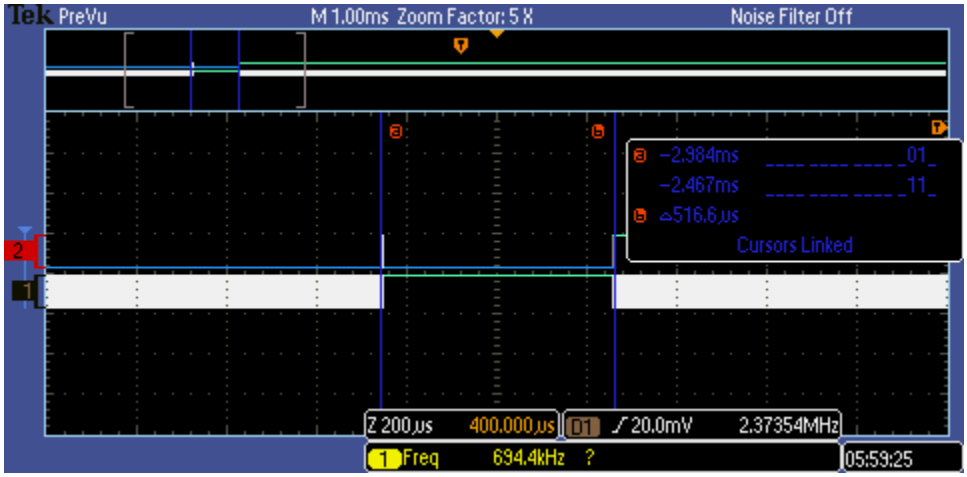


Figure 3.16): ISR profile with 64x hardware averaging. Time in ISR = **516.6 us**



4.0: Analysis and Discussion:

1) The ISR toggles PF2 three times. Is this debugging intrusive, nonintrusive or minimally intrusive? Justify your answer.

Toggling PF2 is a minimally intrusive debugging technique. The effect on the real-time interaction of the system is negligible.

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 ways are printf statements better than dumps? In what ways are dumps better than printf statements?

Printf statements require almost no extra memory to operate. They are also extremely convenient and easy to use while debugging traditionally. Dumps require a lot of extra memory overhead, but they are much less intrusive than print statements. Writing to a dump requires nanoseconds, while calling a printf statement requires about a millisecond. Dumps are overall much better for debugging real-time systems (minimally intrusive).

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?

When two threads have shared access to global data or an I/O port, and one of the accesses is a write, then there exists a critical section in the lower priority thread. This usually occurs when the write access is a multistep, non-atomic sequence of read-modify-write, write-write, or write-read.

4) Define "minimally intrusive".

We classify a debugging instrument as *minimally intrusive* if it has a neglibile effect on the system being debugged. With respect to real-time systems, it cannot disturb the real-time interaction or interfere with the bounded latency.

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?*

Hardware averaging increases the time it takes to run the sampling interrupt service routine (ISR). Thus, it decreases the throughput, and we could possibly miss some samples. Also, this will increase the latency of our system / decrease our ability to service other interrupt requests.