1. **OBJECTIVES**

Lab 2 aimed to teach us software debugging techniques (such as performance debugging, data dumps, and profiling), experience concepts of real time (such as probability mass functions and the central limit theorem), observe critical sections, and use the oscilloscope and logic analyzer. Finally, we were able to get a head start on Lab 3 by writing a line drawing function.

1. **SOFTWARE DESIGN**

See TestPmf.c and TestTimeJitter.c

1. **MEASUREMENT DATA**
   1. **Answers to the prep questions**
      1. ***What is the purpose of all the DCW statements?***

DCW stands for “declare constant word” but really puts a 16 bit halfword into memory. LDR and STR load from/store to memory by offsetting from a register or the PC and the offset is not very large (relatively). To fix this, the compiler stores into memory the values it needs to access and stores these near the program counter. Since IO ports are memory mapped and PF1 is mapped to 0x40025008, the DCW command is used to store 0x5000 at 0x068C and 0x4002 at 0x68E.

* + 1. ***The main program toggles PF1. Estimate how fast PF1 will toggle.***

At 80 MHz, each assembly instruction takes about 25 nanoseconds to execute. Toggling PF1 takes 5 assembly instructions, so the execution will take 125 ns.

* + 1. ***What is in R0 after the first LDR is executed? And after the second LDR?***

The Cortex M4 processor uses little endian format so when the first LDR executes, the value 0x40025000 is loaded into R0. Next, the value at address 0x40025008 (i.e. the value of PF1) is then loaded into R0 (meaning R0 will be either 0x00 or 0x02 after the second LDR).

* + 1. ***How would you have written the compiler to remove an instruction?***

You could just LDR the value into a separate register other than R0 since that value (0x40025000) will be used twice.

LDR r1,[pc,#24] ; r1 = 0x40025000

LDR r0,[r1,#0x08] ; r0 = M[0x40025008]

EOR r0,r0,#0x02

~~LDR r1,[pc,#16] ; @0x0000068C~~

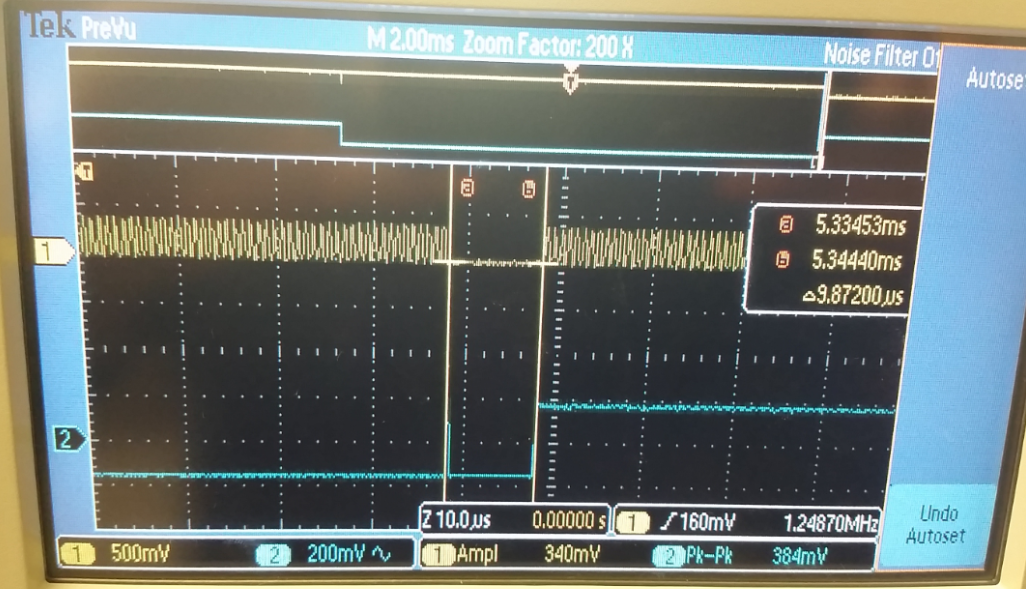
STR r0,[r1,#0x08]

* + 1. ***100-Hz ADC sampling occurs in the Timer0 ISR. The ISR toggles PF2 three times. Is there a critical section? If yes, describe how to remove the critical section. If no, then justify your answer.***

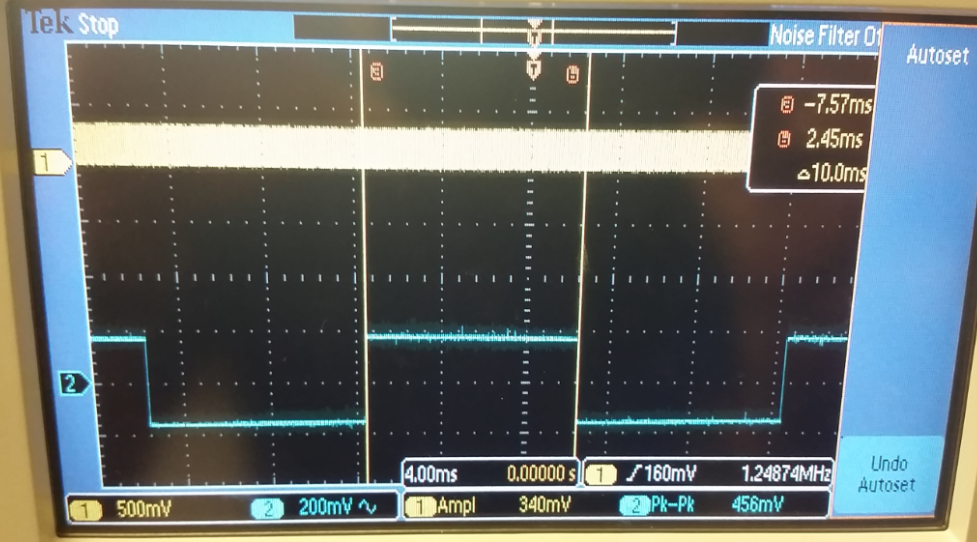
No critical region because both the interrupt toggle and main toggle use different pins due to bit-specific addressing. This means the code remains friendly.

* 1. **Debugging Profile with the Oscilloscope**

Channel one (the top yellow waveforms) shows the behavior of PF1 and indicates that the main program is running. Channel two (the bottom blue waveforms) shows the behavior of PF2 and shows that the interrupt service routine (ISR) is running.



*Figure 1: Zoomed in view shows that the ISR runs for about 10 microseconds and that the main is not running during the ISR (since PF1 is not toggling).*



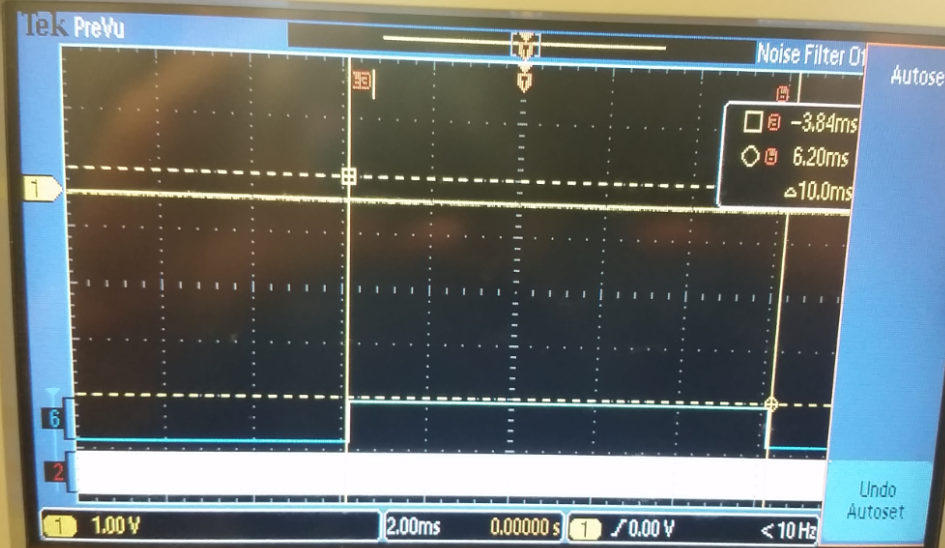
*Figure 2: Zoomed out view shows that the ISR is called about every 10 ms.*

* 1. **Debugging Profile with the Logic Analyzer**

Channel 6 (the upper waveform) shows the behavior of PF2 and shows that the ISR is running. Channel 2 (the lower waveform) shows the behavior of PF1 and indicates that the main program is running. The program is running in the main if the ISR is not running. Therefore, the percentage of time spent in the main versus the time spent in the ISR is shown by Equation 1.



*Figure 3: Zoomed in view shows that the ISR runs for about 10 microseconds and that the main is not running during the ISR (since PF1 is not toggling).*

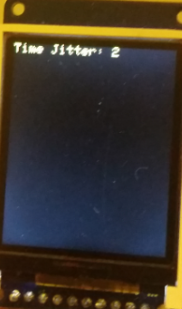


*Figure 4: Zoomed out view shows that the ISR is called about every 10 ms.*

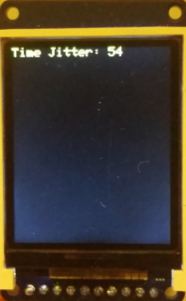
* 1. **Experiencing a Critical Section**

After the critical section, PF2 is incorrect. The assembly loads from GPIO\_PORTF\_R in preparation to perform the XOR operation inside the main. However, an interrupt is signaled and PF2 is toggled 3 times. The PC returns to main and operates on the previous value of the GPIO port, which essentially incorrectly toggles PF2 again.  To fix the critical section without bit-specific addressing, we could have two different ports for the debugging profile pins.

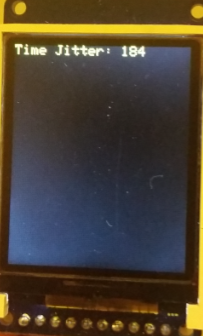
* 1. **Time Jitter Measurements**



*Figure 5: time jitter with just one sampling interrupt active.*



*Figure 6: time jitter with two interrupts but the 2nd ISR does not do anything*



*Figure 7: time jitter with two interrupts and the 2nd ISR loops for 10 iterations.*

Time jitter is a function of how many interrupts are active, how frequently the

interrupts occur, how long each interrupt takes, the random event of which

instruction in the main program was being executed at the time of the interrupt

trigger, and the priority of the interrupts. Say interrupt 2 is the one causing the

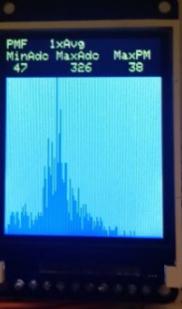
time jitter. If interrupt 2 has a lower precedence than interrupt 1, then the time

jitter will remain the same.

* 1. **ADC Noise Measurements**

The noise appeared to be bimodal. When we run it over and over, we generally get the same shape but varying values due to the noise. Sometimes it appears unimodal though.

* 1. **PMF Plots using Hardware Averaging**



*Figure 8: PMF plot for no hardware averaging*



*Figure 9: PMF plot with 4x hardware averaging*



*Figure 10: PMF plot with 16x hardware averaging*



*Figure 11: PMF plot with 64x hardware averaging*

The Central Limit Theorem states that the more samples you take of a noisy signal, the closer the values get to the population’s normal distribution.  This is exactly what we see with hardware averaging.  As we took more samples, the noise got damped out and the samples got put into two different humps (each of which was the population’s true average). Hardware averaging makes the ISR take longer to execute (because it takes more samples per data point). This makes the occurrence of an ISR much more noticeable, the channel with PF1 doesn’t look like a perfect square wave anymore.

* 1. **Line Drawing Function for Lab 3**



*Figure 12: simple line drawing function*

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

Toggling PF2 is minimally intrusive because it still affects the program’s functionality (it only requires approximately 10 clock cycles to execute and uses up one pin), but this disruption is rather negligible. It doesn’t affect the real-time interaction between hardware and software as much as breakpoints and single-stepping, but it still has a minor effect.

* 1. **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 are nice because they are very convenient (all you have to do is look at the value on the screen). Dumps may not be as convenient, but they are less intrusive in terms of execution time (they do require memory space though). Calling printf could make the ISR take too long and we could miss recording the next value at the appropriate time.

* 1. **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?**

Three things need to happen in order for a critical section to occur: there needs to be a shared global, a nonatomic access, and at least one write. An example of a critical section is when an ISR performs an operation that is nonatomic and another ISR interrupts before the operation is complete and reads a bogus value or modifies a value needed by the original ISR.

* 1. **Define “minimally intrusive”**

Intrusiveness is defined by the extent to which the software affects the functionality of the code (whether it be the program’s memory footprint, its execution time, etc.). Something is minimally intrusive if it still affects the program’s performance but in a way that is rather negligible.

* 1. **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?**

Averaging samples to get a single data point requires a longer execution time and uses up more power per data point. Sometimes we do not need the most accurate values anyway so hardware averaging wouldn’t be needed (sometimes the noise is tolerable). If the sampling is done in an ISR, then using hardware averaging could make the ISR run for too long.