Ryan Comer – rnc525
Connor White – bcw824

Lab 2 Report

Performance Debugging

Objectives

The objective of this lab were to teach us different techniques for debugging, demonstrate the risk of critical sections, and measure different pins on the board with an oscilloscope and logic analyzer. The primary debugging technique that we used in this lab was to dump a large amount of data into arrays, and to do some calculations on the data to determine if the system functions properly. Specifically, we periodically took samples from the on-board ADC sampler and dumped the data readings and their corresponding time into 2 different arrays. We then took processed the data in two different ways: finding the time jitter for the sampling, and finding the PMF of the value of the sample. We were shown the risk of critical sections when 2 different threads of execution were accessing the same global variable. In this case, it was the data register for port f. The starter code initially used bit-specific addressing to remove the risk, but changing the code to write to the entire data register showed that the value might be overwritten and invalidated. Finally, we measured PF1 and PF2 on the board using the oscilloscope and logic analyzer. This allowed us to see exactly how long the ISR ran and how often by looking at the values of the pins.

Software Design

The software for this lab included a timer interrupt handler that read data from the on-board sequencer on PE4 and dumped the time/data measurements into 2 arrays, and a main program that waited until 1000 measurements were taken, measured the time jitter, and generated a plot of the PMF of the data on the Stelaris LCD display. The source code for this lab was submitted through canvas along with this report.

Measurement Data

We took various measurements throughout this lab, and this section will show those measurements and describe their significance.

Oscilloscope

We measured the value of PF1 and PF2 using an oscilloscope. The main method of our program toggled PF1 while the timer interrupt toggled PF2. With these 2 measurements, we were able to see exactly when the interrupts happened and how long they lasted. The following figure shows our measurements. The top signal (probe 1) measured PF1 while the bottom signal (probe 2) measured PF2.

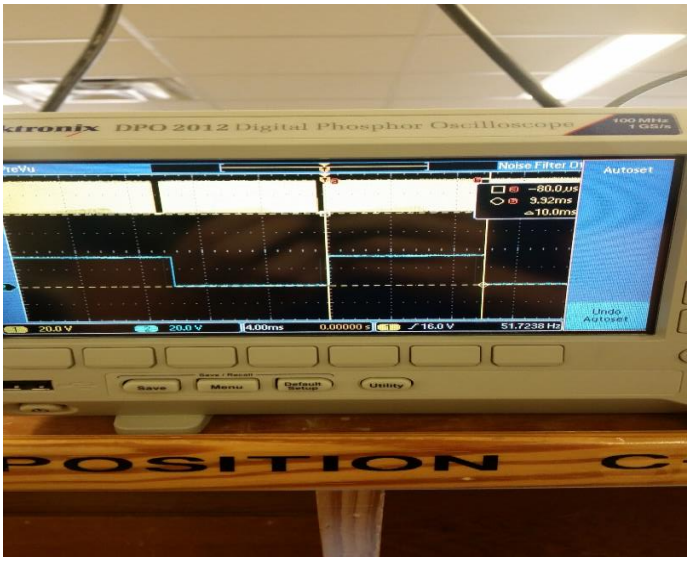

Figure 1: Oscilloscope

Logic Analyzer

We made similar measurements with the logic analyzer as we did with the oscilloscope. The difference here is that the logic analyzer sets a threshold for the voltage and only displays a 1 or 0 depending on where the voltage is relative to the threshold. The following figure shows the our logic analyzer measurements.

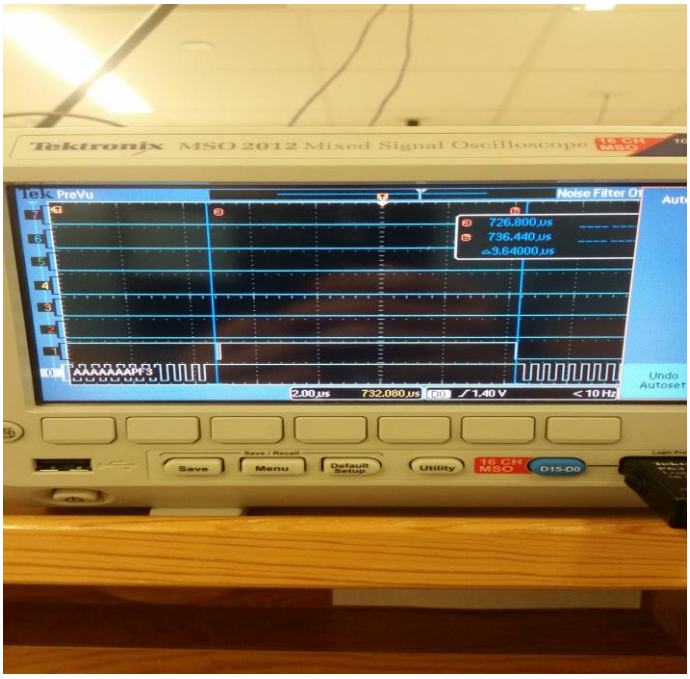

Figure 2: Logic Analyzer

The interrupt service routine is estimated to finish in 9.6 microseconds as shown in the figure.

Critical Section

There is a potential critical section in this lab regarding port F. Both the main function and the timer interrupt toggle bits in the port F data register. This read-modify-write sequence for the main function is shown with the following assembly code:

```
LDR     r0,[pc,#56]  ; @0x00001010
LDR     r0,[r0,#0x00]
EOR     r0,r0,#0x02
LDR     r1,[pc,#40]  ; @0x00001008
STR     r0,[r1,#0x3FC]
```

After R0 is loaded with the data in the port F data register, the bit for PF1 is toggled and the value is stored back into the data register. The ISR has similar code, which can be a problem if the interrupt happens in the middle of this read-modify-write sequence. Essentially, one of the threads could be modifying 'old' data. The starter code uses bit-specific addressing, which removes the critical section by removing the data register as a shared global variable, and instead uses two separate memory locations for PF1 and PF2. Another option for removing the critical section would be to disable interrupts when a read-modify-write sequence occur on a global variable. This isn't ideal however because interrupts could have important tasks that need to happen (i.e. air bags).

Time Jitter

In this lab, we measured the time jitter for 1000 different samples of the on-board ADC sequencer. Time jitter is defined as the difference between the largest and smallest time differences of consecutive measurements. In a real system, there would be requirements for the time jitter of the system to be under a certain value. If the jitter is too high, then your system would be unreliable as to when certain tasks would be performed. We took 4 time jitter measurements under different conditions, and the following table shows our measurements:

Condition	Measurements
1 ISR and no divide instruction	[0x14, 0x0, 0x14, 0x0]
1 ISR and a divide instruction	[0x18, 0x10, 0x18, 0x10]
3 ISRs and no divide instruction	[0xB4, 0xBE, 0xAE, 0xBE]
3 ISRs and a divide instruction	[0xAE, 0xB4, 0xB4, 0xB8]

Table 1: Time Jitter Calculations

Time jitter occurs when something is keeping a task from occurring on time. As Table 1 shows, more interrupts of a higher priority in a system will increase the time jitter for a lower priority interrupt because the higher priority interrupt has to finish first. If you need to bound the time jitter for a particular part of a system, then you need to reduce the number of things that can stop the task from starting and completing on time.

PMF of the Samples

After sampling the sequencer 1000 times, a PMF of the values was generated. This PMF shows the probability of the sequencer returning a certain reading. One would expect the sequencer to always

return the same reading for the same circuit, but real circuits have noise. The noise is what we are graphing here. We sampled the sequencer with and without hardware averaging, which we would expect to reduce the noise in the measurements. In the figure, the graphs for hardware averaging are read as follows: 0x (top left), 4x (top right), 16x (bottom left), 64x (bottom right).

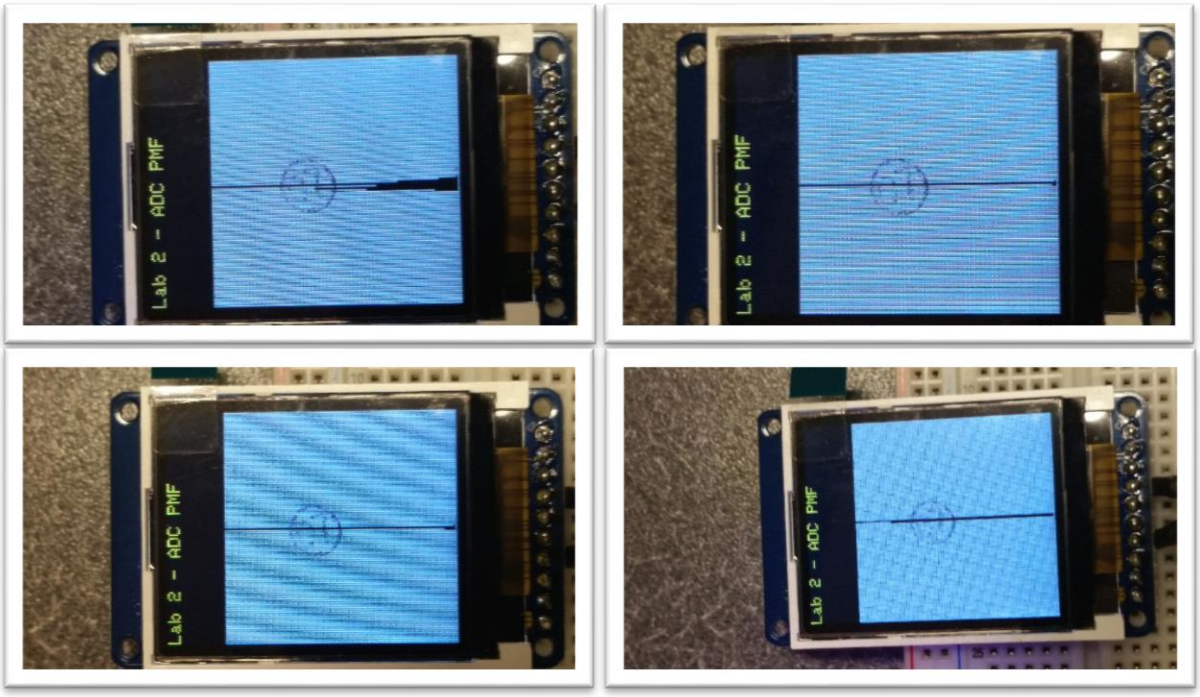

Figure 3: Sampling PMF

The Central Limit Theorem states that the distribution of the averages of N samples should be approximately normal as N goes to infinity. Therefore, higher values of hardware averaging should produce PMFs that are closer and closer to a normal distribution. This is hard to see in our graphs because not much noise was measured. You can however see a difference between no averaging (top left) and 64x averaging (bottom right). There is some noise visible with no averaging, but the 64x averaging is almost a vertical line. This is supported by the Central Limit Theorem because more of the values are concentrated around the mean for the 64x averaging.

Hardware Averaging Effect on ISR Time

Although hardware averaging reduces the noise in the PMF, it increases the execution time for the ISR. The following figure shows the debugging profile for the execution time of the ISR on the logic analyzer:

Figure 4: ISR Time with Hardware Averaging

As you can see, the ISR takes 128 microseconds to execute, which is much longer than the 9 microseconds without hardware averaging.

Analysis/Discussion

The ISR toggles PF2 three times. Is this debugging intrusive, nonintrusive or minimally intrusive?

The debugging is minimally intrusive because the time the debugging takes is negligible with respect to the ISR where the toggling happens. The ISR is executed every 10 milliseconds, while toggling a pin 3 times takes (25 * 5 * 3) = 375 nanoseconds. The time 375 nanoseconds is very small compared to 10 milliseconds.

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?

Printf and data dumping are both useful debugging techniques, but they should be used in different situations. If you want to view and compare different variable at a specific point in time, then a printf statement is useful because you can print all of them to the screen for evaluation. Data dumping is more useful if you want to run calculations on a large amount of data. For example we could have used printf and printed out each data/time pair and compared them visually, but storing the data into arrays and calculating the time jitter from those arrays saved us hours of doing it by hand. If we wanted to see the min and max value that the sequencer sampled, then we could have printed them both to the screen after the sampler was done.

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?

The software would need to access shared variables in different threads and it has to actually write to the variable. This could cause threads to interrupt each other in the middle of a read-modify-write sequence and invalidate the data that another thread is using.

Define “minimally intrusive”

A debugging instrument is minimally intrusive if it has a negligible effect on the instrument being tested. This is measured by the time the debugging takes divided by the time between executions of the thing being tested.

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

Even though hardware averaging is more accurate, it takes more power. It could also depend on the problem you are trying to solve, and capturing the noise of a system might be part of the system's requirements.