EE345L – Lab 2: Performance Debugging

Sean Tremblay and Dhruv Sandesara

02/09/17

A) OBJECTIVE

The lab aimed to familiarize students with dynamic and real-time performance debugging techniques with various degrees of intrusiveness. Oscilloscopes, logic analyzers, and software dumps were used to observe data.

Profiling also presented the detection and visualization of program activity. We were also familiarized with the concept of jitter when using software interrupts

C) // ADCTestMain.c

// Runs on TM4C123

// This program periodically samples ADC channel 0 and stores the

// result to a global variable that can be accessed with the JTAG

// debugger and viewed with the variable watch feature.

// Daniel Valvano

// September 5, 2015

/* This example accompanies the book

"Embedded Systems: Real Time Interfacing to Arm Cortex M Microcontrollers",

ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2015

```
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */

// center of X-ohm potentiometer connected to PE3/AIN0
// bottom of X-ohm potentiometer connected to ground
// top of X-ohm potentiometer connected to +3.3V
#include <stdint.h>
#include <stdlib.h>
#include "ADCSWTrigger.h"
#include "../inc/tm4c123gh6pm.h"
#include "PLL.h"
//#include "Timer1.c"

#define PF2             (*((volatile uint32_t *)0x40025010))
#define PF1             (*((volatile uint32_t *)0x40025008))


void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value
void WaitForInterrupt(void);  // low power mode
//the 1000length arrays and 2 indexes of that array
//if indecies are 999 then we're done
int32_t Required_Debug_array1[1000];
int32_t Required_Debug_array2[1000];
int array1_pointer;
int array2_pointer;
int complete_flag;
```

```
volatile uint32_t ADCvalue;
// This debug function initializes Timer0A to request interrupts
// at a 100 Hz frequency.  It is similar to FreqMeasure.c.
void (*PeriodicTask)(void);   // user function

// ***************** TIMER1_Init ****************
// Activate TIMER1 interrupts to run user task periodically
// Inputs:  task is a pointer to a user function
//          period in units (1/clockfreq)
// Outputs: none
void Timer1_Init(void(*task)(void), uint32_t period){
  SYSCTL_RCGCTIMER_R |= 0x02;   // 0) activate TIMER1
  PeriodicTask = task;          // user function
  TIMER1_CTL_R = 0x00000000;    // 1) disable TIMER1A during setup
  TIMER1_CFG_R = 0x00000000;    // 2) configure for 32-bit mode
  TIMER1_TAMR_R = 0x00000002;   // 3) configure for periodic mode, default down-count settings
  TIMER1_TAILR_R = 0xFFFFF-1;    // 4) reload value
  TIMER1_TAPR_R = 0;            // 5) bus clock resolution
  //TIMER1_ICR_R = 0x00000001;    // 6) clear TIMER1A timeout flag
  //TIMER1_IMR_R = 0x00000001;    // 7) arm timeout interrupt
  NVIC_PRI5_R = (NVIC_PRI5_R&0xFFFF00FF)|0x00008000; // 8) priority 4
// interrupts enabled in the main program after all devices initialized
// vector number 37, interrupt number 21
  NVIC_EN0_R = 1<<21;           // 9) enable IRQ 21 in NVIC
  TIMER1_CTL_R = 0x00000001;    // 10) enable TIMER1A
}
```

```
void Timer1A_Handler(void){
        /*if(array1_pointer <= 999){
                Required_Debug_array1[array1_pointer] = TIMER1_TAILR_R;
                Required_Debug_array2[array1_pointer] = TIMER1_TAILR_R;
        }*/
  //int saved_value;
        //saved_value = PF1;
        //PF1 ^= 0x02;
        //PF1 ^= 0x02;
        TIMER1_ICR_R = TIMER_ICR_TATOCINT;// acknowledge TIMER1A timeout
  //(*PeriodicTask)();                // execute user task
        //PF1 ^= 0x02;
        //PF1 = saved_value;
}



void Timer0A_Init100HzInt(void){
  volatile uint32_t delay;
  DisableInterrupts();
  // **** general initialization ****
  SYSCTL_RCGCTIMER_R |= 0x01;      // activate timer0
  delay = SYSCTL_RCGCTIMER_R;      // allow time to finish activating
  TIMER0_CTL_R &= ~TIMER_CTL_TAEN; // disable timer0A during setup
  TIMER0_CFG_R = 0;                // configure for 32-bit timer mode
  // **** timer0A initialization ****
                                   // configure for periodic mode
  TIMER0_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
  TIMER0_TAILR_R = 799999;         // start value for 100 Hz interrupts
```

```
  TIMER0_IMR_R |= TIMER_IMR_TATOIM;// enable timeout (rollover) interrupt
  TIMER0_ICR_R = TIMER_ICR_TATOCINT;// clear timer0A timeout flag
  TIMER0_CTL_R |= TIMER_CTL_TAEN;  // enable timer0A 32-b, periodic, interrupts
  // **** interrupt initialization ****
                                  // Timer0A=priority 2
  NVIC_PRI4_R = (NVIC_PRI4_R&0x00FFFFFF)|0x40000000; // top 3 bits
  NVIC_EN0_R = 1<<19;              // enable interrupt 19 in NVIC
}
void Timer0A_Handler(void){



  PF2 ^= 0x04;                  // profile
  PF2 ^= 0x04;                  // profile
  ADCvalue = ADC0_InSeq3();
          if(array1_pointer <= 999){
                  Required_Debug_array1[array1_pointer] = TIMER1_TAR_R;//This one is obviously the
timer
                  Required_Debug_array2[array1_pointer]  = ADCvalue;//This one is obviously the ADC
                  array1_pointer += 1;
          }else{
                  complete_flag = 1;
          }


  PF2 ^= 0x04;                  // profile
          TIMER0_ICR_R = TIMER_ICR_TATOCINT;    // acknowledge timer0A timeout




}
```

```
void Jitter_Calculation(void){//int *,
	int i;
	int timer_max;
	int timer_min;
	int ADC_min;
	int ADC_max;
	int time_difference;
	int ADC_difference;

	timer_max =  abs(Required_Debug_array1[1] - Required_Debug_array1[0]);;
	ADC_max =  abs(Required_Debug_array2[1] - Required_Debug_array2[0]);;
	time_difference = 0;
	ADC_min =  abs(Required_Debug_array2[i] - Required_Debug_array2[0]);;
	timer_min =  abs(Required_Debug_array1[i] - Required_Debug_array1[0]);;

	for(i =2; i< 999; i++){
		if(abs(Required_Debug_array1[i] - Required_Debug_array1[i-1]) < timer_min){
			timer_min  = abs(Required_Debug_array1[i] - Required_Debug_array1[i-1]);
		}
		if(abs(Required_Debug_array1[i] - Required_Debug_array1[i-1]) > timer_max){
			timer_max  = abs(Required_Debug_array1[i] - Required_Debug_array1[i-1]);
		}
		//This is the part for the ADC
		if(abs(Required_Debug_array2[i] - Required_Debug_array2[i-1]) < ADC_min){
			ADC_min  = abs(Required_Debug_array2[i] - Required_Debug_array2[i-1]);
		}
		if(abs(Required_Debug_array2[i] - Required_Debug_array2[i-1]) > ADC_max){
			ADC_max  = abs(Required_Debug_array2[i] - Required_Debug_array2[i-1]);
```

```
                }

        }
        time_difference = timer_max - timer_min;
        ADC_difference = ADC_max - ADC_min;


}


int main(void){
        int sanity_check_value;
  PLL_Init(Bus80MHz);                 // 80 MHz
  SYSCTL_RCGCGPIO_R |= 0x20;          // activate port F
  ADC0_InitSWTriggerSeq3_Ch9();       // allow time to finish activating
  Timer0A_Init100HzInt();             // set up Timer0A for 100 Hz interrupts
        Timer1_Init(Timer1A_Handler, 100);
  GPIO_PORTF_DIR_R |= 0x16;           // make PF2, PF1 out (built-in LED)
  GPIO_PORTF_AFSEL_R &= ~0x16;        // disable alt funct on PF2, PF1
  GPIO_PORTF_DEN_R |= 0x16;           // enable digital I/O on PF2, PF1
                                      // configure PF2 as GPIO
  GPIO_PORTF_PCTL_R = (GPIO_PORTF_PCTL_R&0xFFFFF00F)+0x00000000;
  GPIO_PORTF_AMSEL_R = 0;             // disable analog functionality on PF
  PF2 = 0;                    // turn off LED
  EnableInterrupts();
        complete_flag = 0;
  while(1){
    PF1 ^= 0x02;  // toggles when running in main
                sanity_check_value = Required_Debug_array1[array1_pointer];
                if(complete_flag == 1){
```

```
                Jitter_Calculation();
                complete_flag = 0;
            }

  }
}
```

D) Measurement Data (you may take photographs of the scope/logic analyzer/LCD or download the data)

Prep part 2) Show your answers to the five questions a – e.

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

Ans: It is to store the addresses of ports

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

Ans: 1 time every 6 instructions. Therefore 1 time every 80/6 MHz = 13.333 MHz

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

Address of PF1 after 1st LDR, Data in PF1 after 2nd LDR.

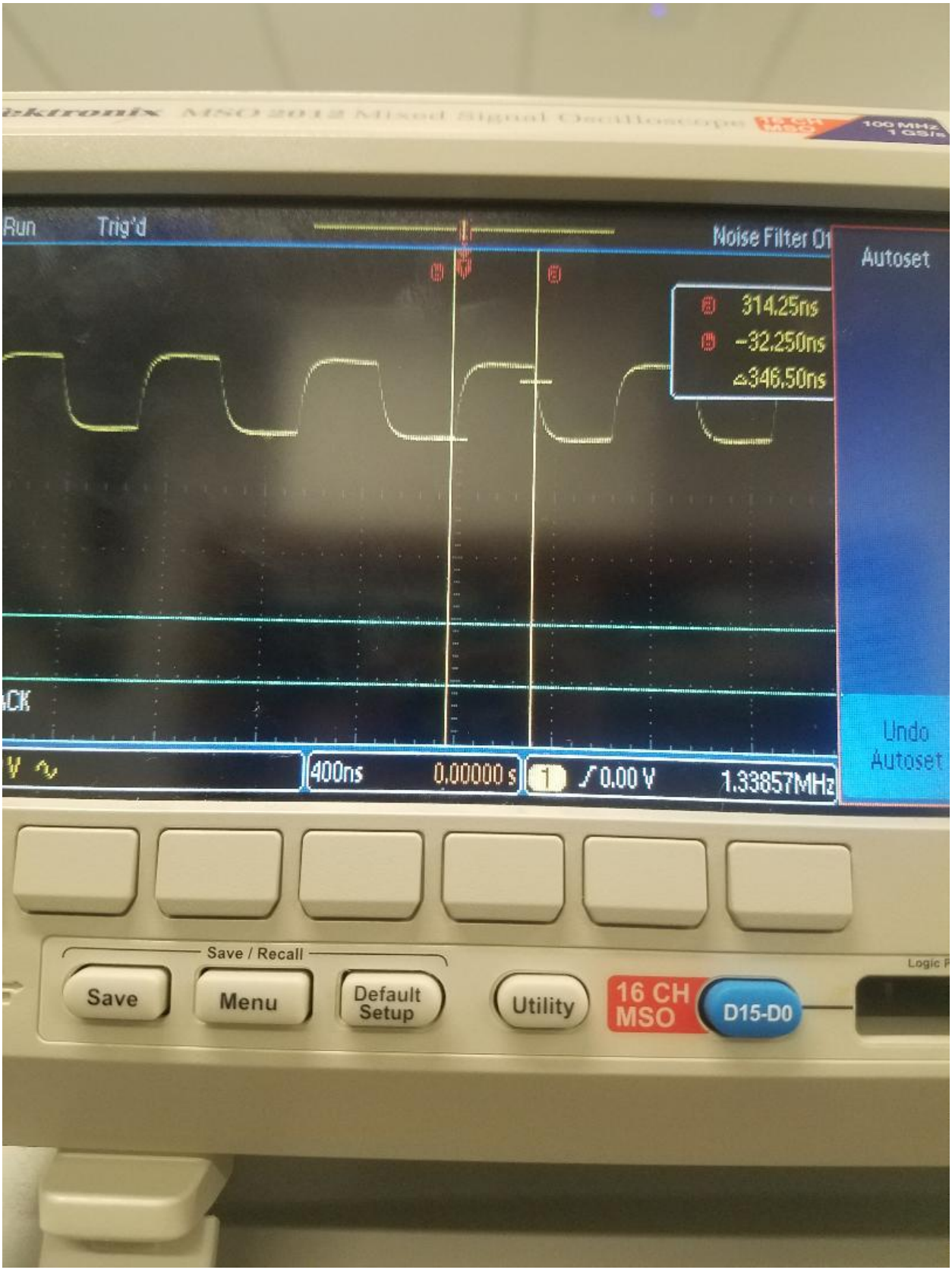
d) How would you have written the compiler to remove an instruction?

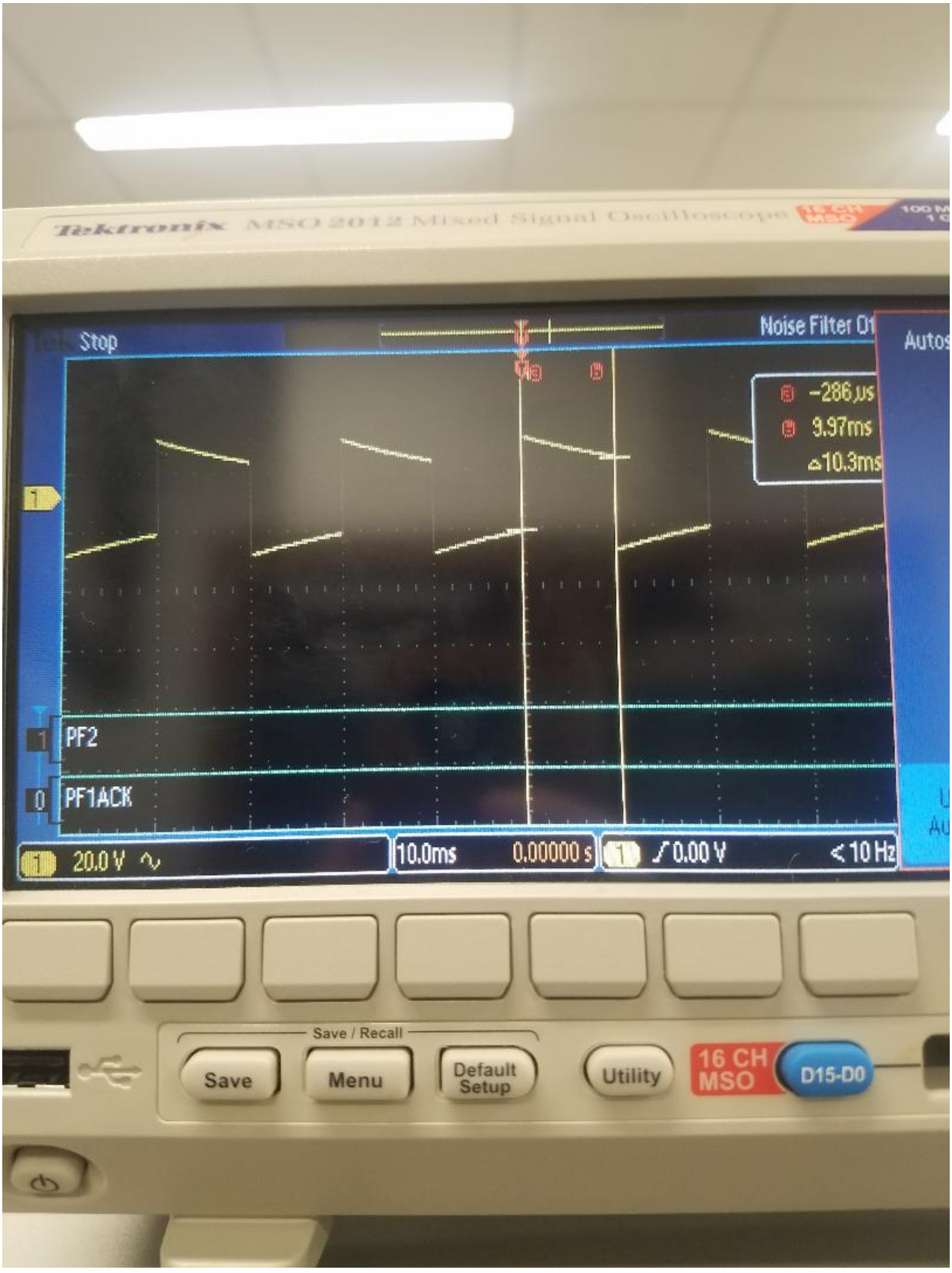
Ans: Change 1st LDR's r0 to r1 and get rid of 3rd LDR

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?

Ans: Yes this is a critical section since if the ISR is triggered after the 2nd LDR and before the STR there is a chance that PF2 has been switched and thus the date that we might output through the main fuction will still contain the old value of PF2 and thus will be incorrect. To mitigate this. Simply save the port data when we jump into the ISR, and then begin the bit toggle. At the end of the interrupt restore the original state of the port.

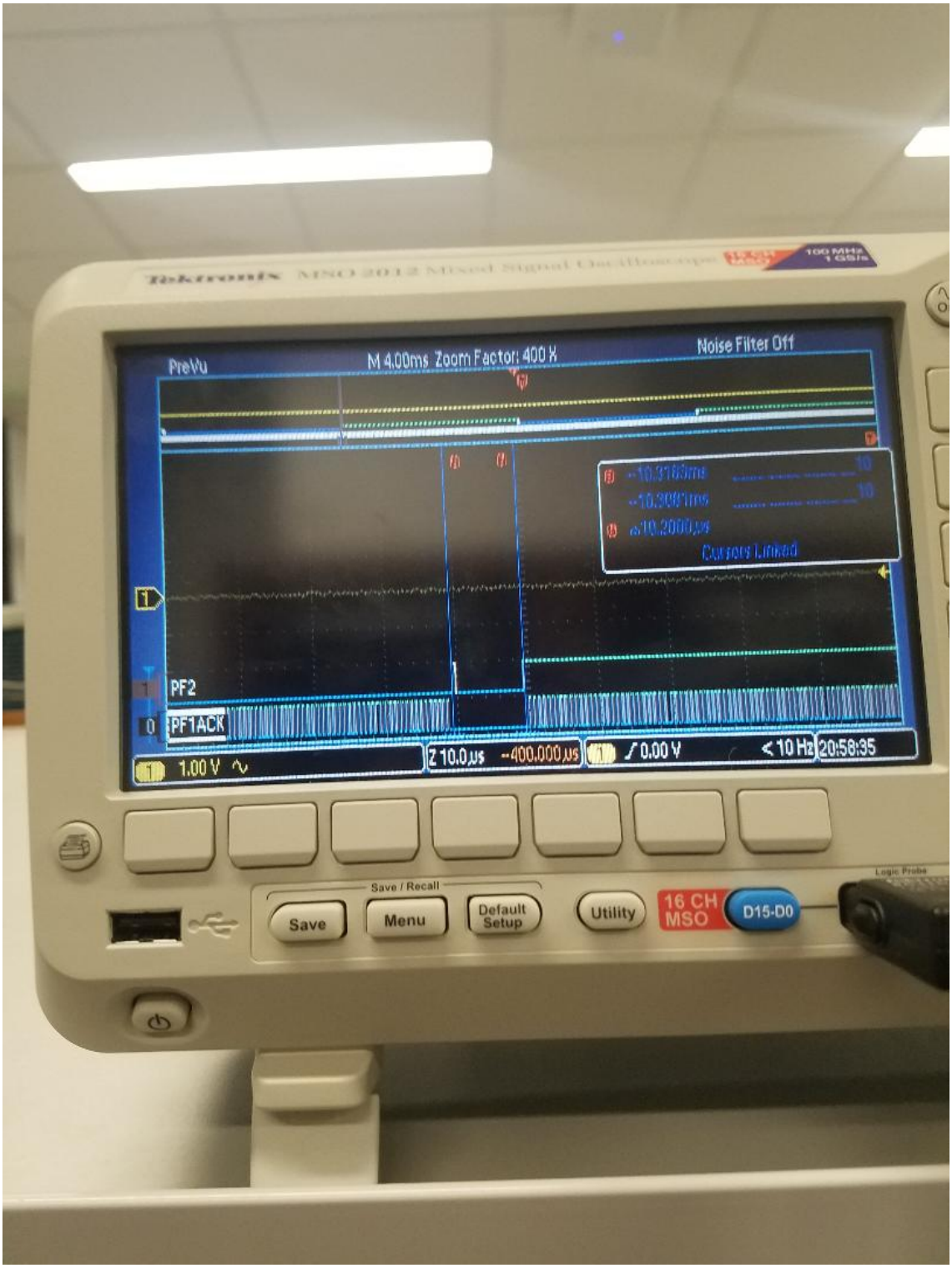
Part A) Debugging profile with scope


Run
Trig'd
Noise Filter
Autoset
314.25ns
-32.250ns
346.50ns
400ns
0.00000 s
0.00 V
1.33857MHz
Undo
Autoset
Save / Recall
Save
Menu
Default Setup
Utility
16 CH MSO
D15-D0


Tektronix MSO 2012 Mixed Signal Oscilloscope
Stop
Noise Filter
-286 µs
9.97ms
Δ10.3ms
PF2
PF1ACK
20.0 V
10.0ms
0.00000 s
0.00 V
< 10 Hz
Save / Recall
Save
Menu
Default Setup
Utility
16 CH MSO
D15-D0

Part B) Debugging profile with logic analyzer, estimation of percentage time in main/ISR

Estimation: 10 microseconds / 10 ms = .1%



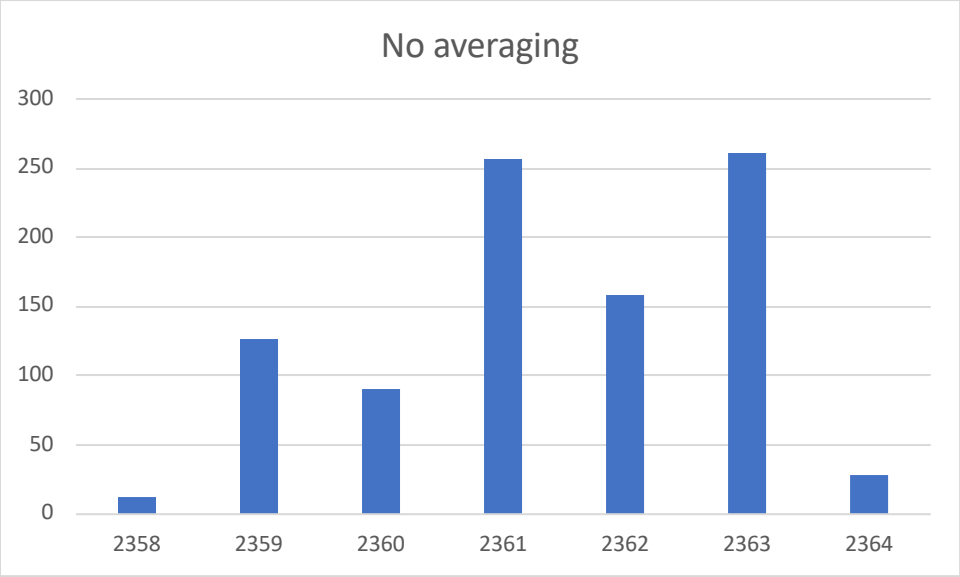
Part C) Explain the critical section and present alternate solutions to removing it

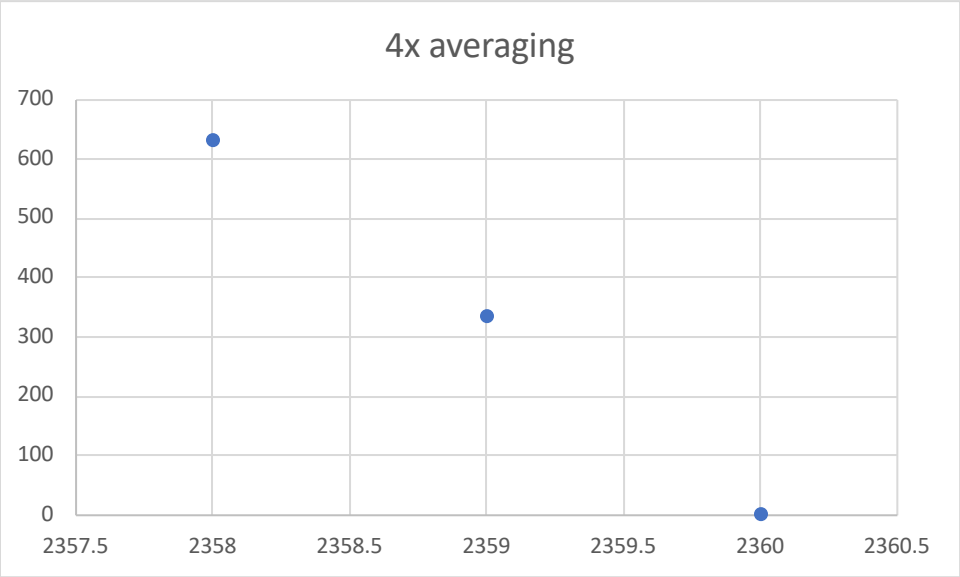
The Critical section occurs when R0 is returned from the interrupt. Should the interrupt occur after the R0 in the main is has already loaded the GPIO value, the interrupt will go toggle, toggle, toggle resulting in a true-> false and vice versa. However, the critical section will cause the main to revert PF2 to it's pre-interrupt value.

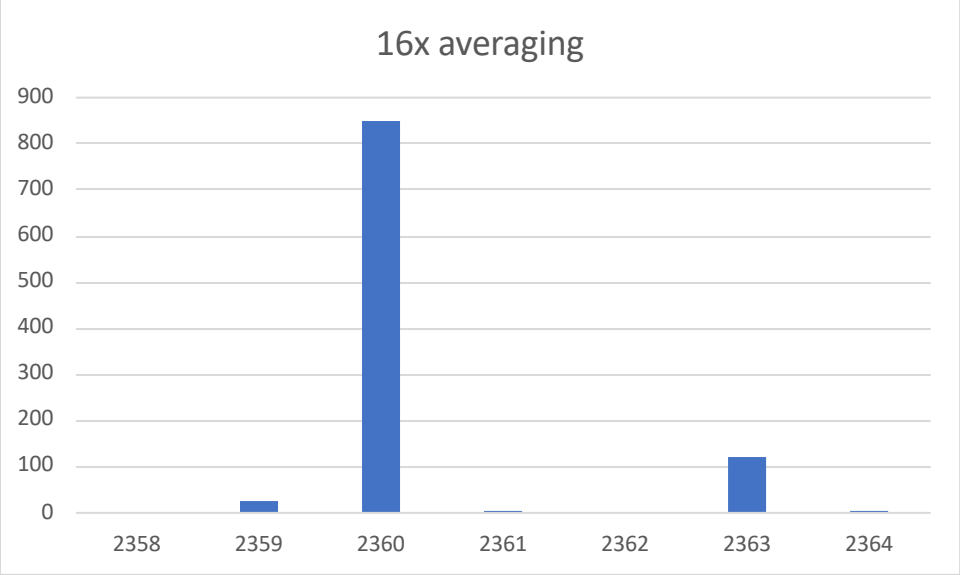
Part D) Time-jitter measurements and generalization of factors that contribute to jitter

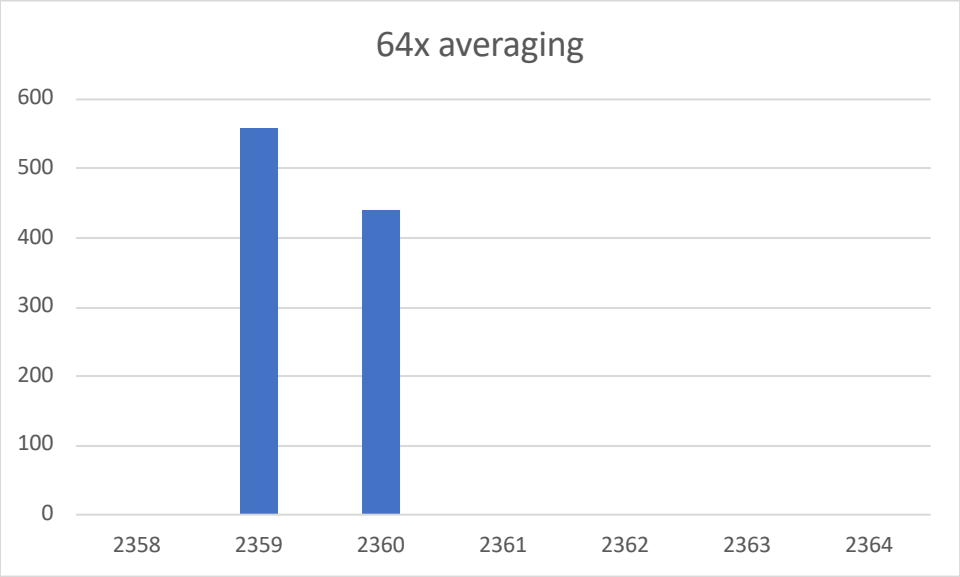
Time jitter was 4 by the code we ran. Time jitters are caused because the ISR waits for the current instruction to complete. This causes a delay in the time calculation which is corrected in the next trigger when it is lower than normal to correct for the first delay. Thus there is an even time jitter number usually.

Part E and F) PMF data and discussion of results. Does your data support CLT? If not why?




4x averaging
700
600
500
400
300
200
100
0
2357.5
2358
2358.5
2359
2359.5
2360
2360.5

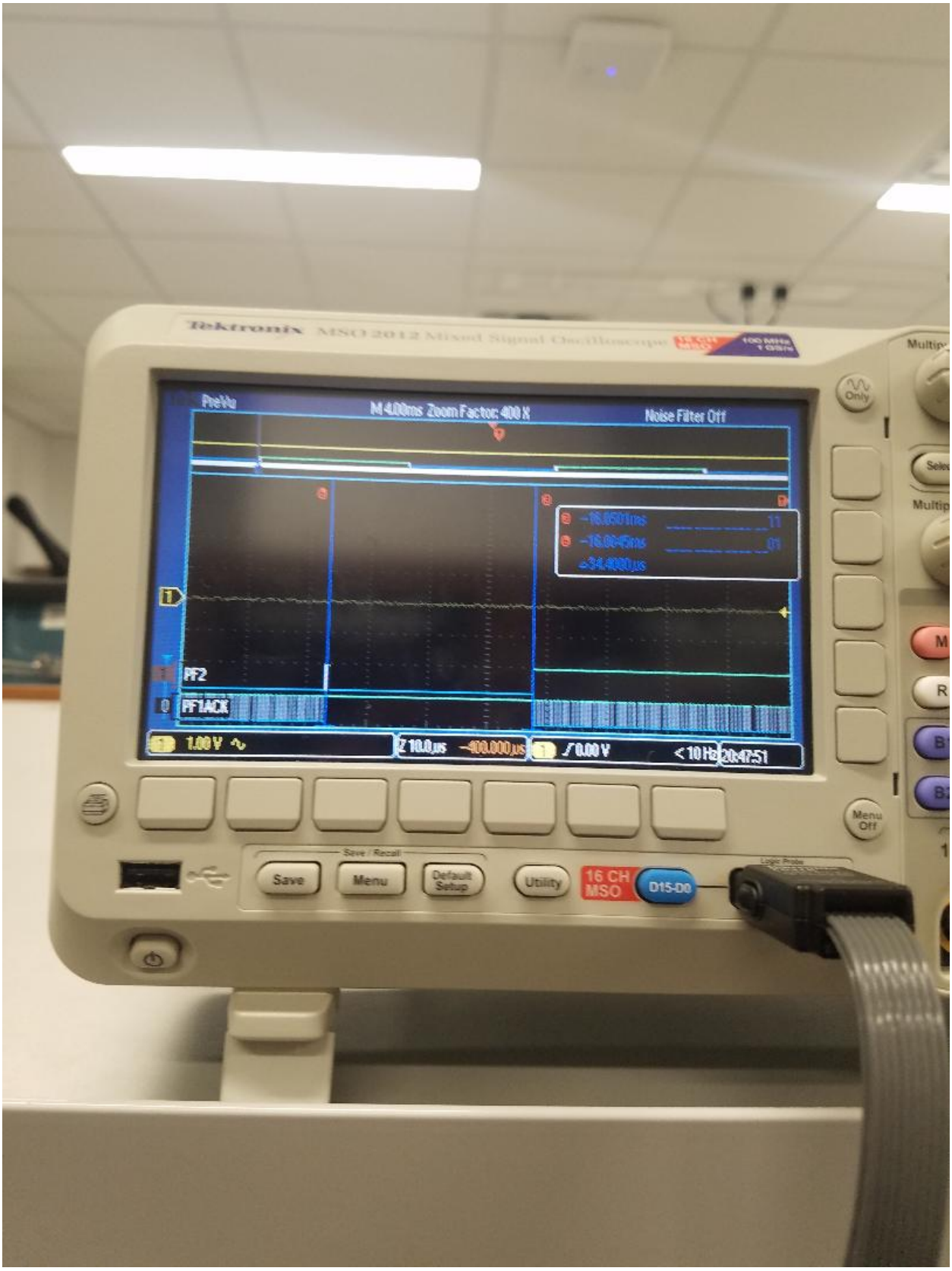

16x averaging
900
800
700
600
500
400
300
200
100
0
2358
2359
2360
2361
2362
2363
2364



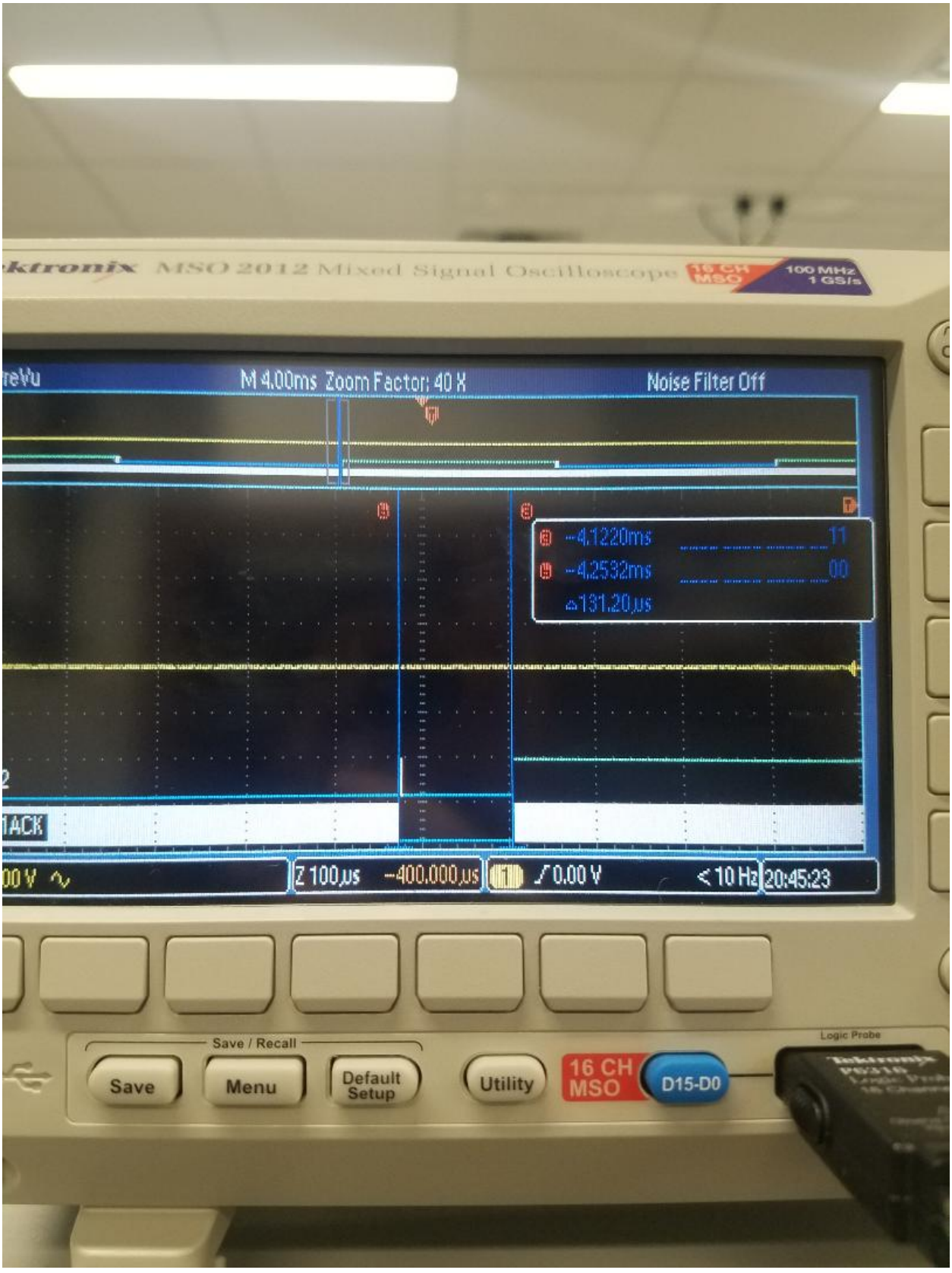
The graphs do support our hypothesis with the CLT as initially without the averaging the variance was large but with the averaging the variance collapses onto the real value that it is supposed to be. This is exactly what CLT says would happen

Part F) Debugging profile of execution time in ISR with hardware averaging. Why is it different?

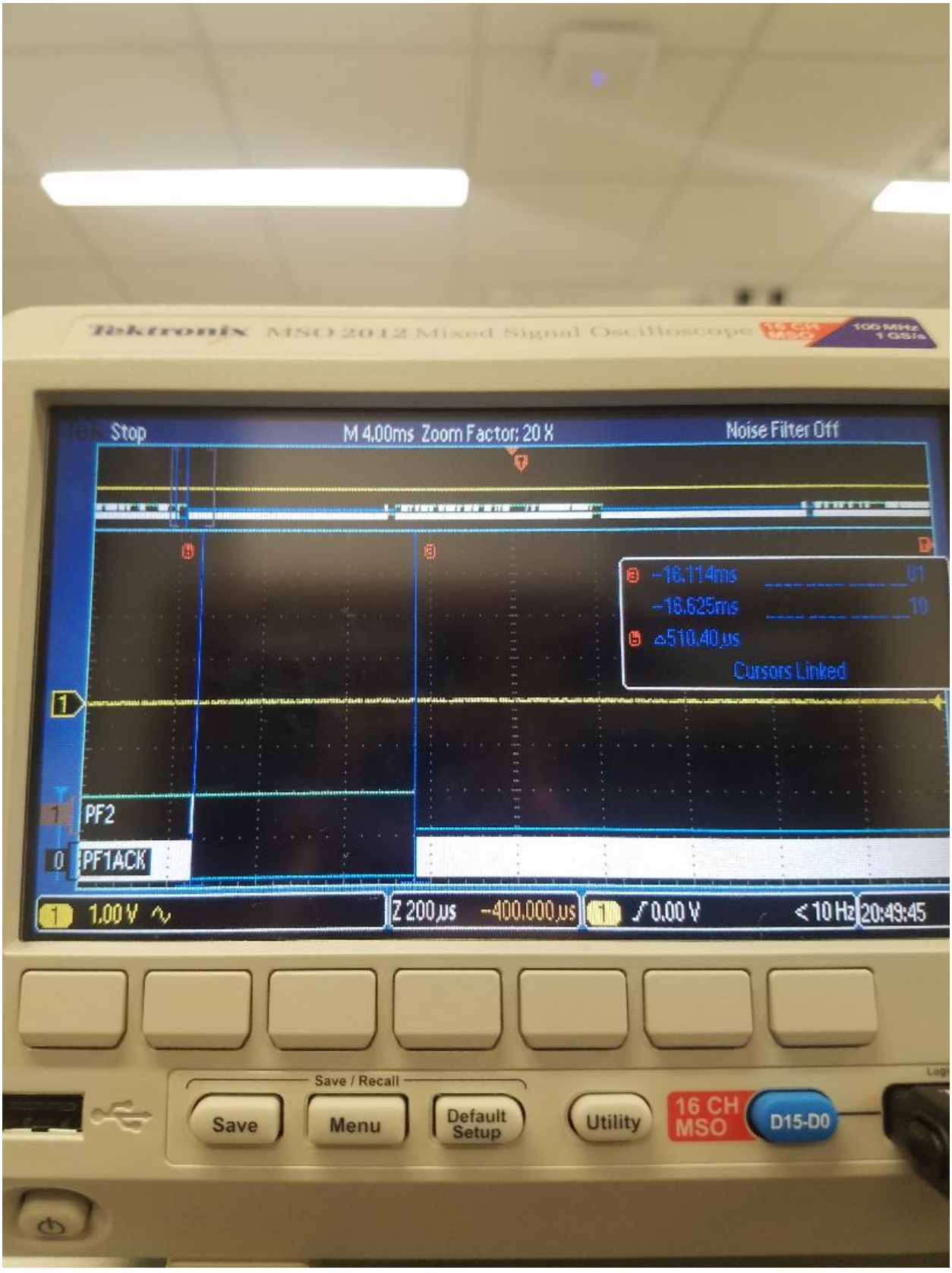
4X Averaging:


Tektronix MSO 2012 Mixed Signal Oscilloscope
PreVu
M 4.00ms Zoom Factor: 400 X
Noise Filter Off
PF2
PF1ACK
1.00 V
10.0µs
0.00 V
Save
Menu
Default Setup
Utility
16 CH MSO
D15-D0
Save / Recall
Menu Off

16X:


MSO 2012 Mixed Signal Oscilloscope
16 CH MSO
100 MHz 1 GS/s
M 4.00ms Zoom Factor: 40 X
Noise Filter Off
-4.1220ms
-4.2532ms
Δ131.20µs
Z 100µs -400.000µs
0.00 V
< 10 Hz
20:45:23
Save / Recall
Save
Menu
Default Setup
Utility
16 CH MSO
D15-D0
Logic Probe

64X:


Tektronix MSO 2012 Mixed Signal Oscilloscope
Stop
M 4.00ms Zoom Factor: 20 X
Noise Filter Off
-16.114ms
-16.625ms
Δ510.40µs
Cursors Linked
PF2
PF1ACK
1.00 V
Z 200µs -400.000µs
0.00 V
< 10 Hz
20:49:45
Save / Recall
Save
Menu
Default Setup
Utility
16 CH MSO
D15-D0

In this case hardware averaging makes it so that the interrupt takes more time to complete. This is because the ADC requires 4, 16 and 64 samples respectively to output a value. This leads to time profiles that are approximately 3, 10, and 50 away from the execution time of the system that does not incorporate hardware averaging.

2.0 ANALYSIS AND DISCUSSION

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

This is minimally intrusive as the ports being toggled have little to negligible effect on the overarching system.

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?

Dumps are better than printfs because the individual elements are easy to manipulate should requirements fluctuate or new data is needed for debugging. Printfs are better than dumps, because they abstract away the low-level processes inherent in dumps, and allow a user to print a data stream with relative ease

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?

A necessary condition for a critical to section occur is that two different processes must access and write to the same memory address. For there to be a critical section, interrupts must occur, where the interrupt handler causes a process to write to data that was to be used later in the pre-interrupt function

4) Define “minimally intrusive”.

Minimally intrusive describes process that has a negligible effect on a system. In debugging theory, this would impy a protocol that has a negligible effect on the running of the rest of the system.

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?

We do not always use hardware averaging, because it slows down our processes. For instance, if we need a system to run in a fixed interval, using hardware averaging can slow our system to the point where we are crossing our maximum threshold