EE345L – Lab 2: Performance Debugging

Sean Tremblay and Dhruv Sandesara

02/09/17

1. 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

Copyright 2015 by Jonathan W. Valvano, valvano@mail.utexas.edu

You may use, edit, run or distribute this file

as long as the above copyright notice remains

THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS, IMPLIED

OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.

VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,

OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

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.

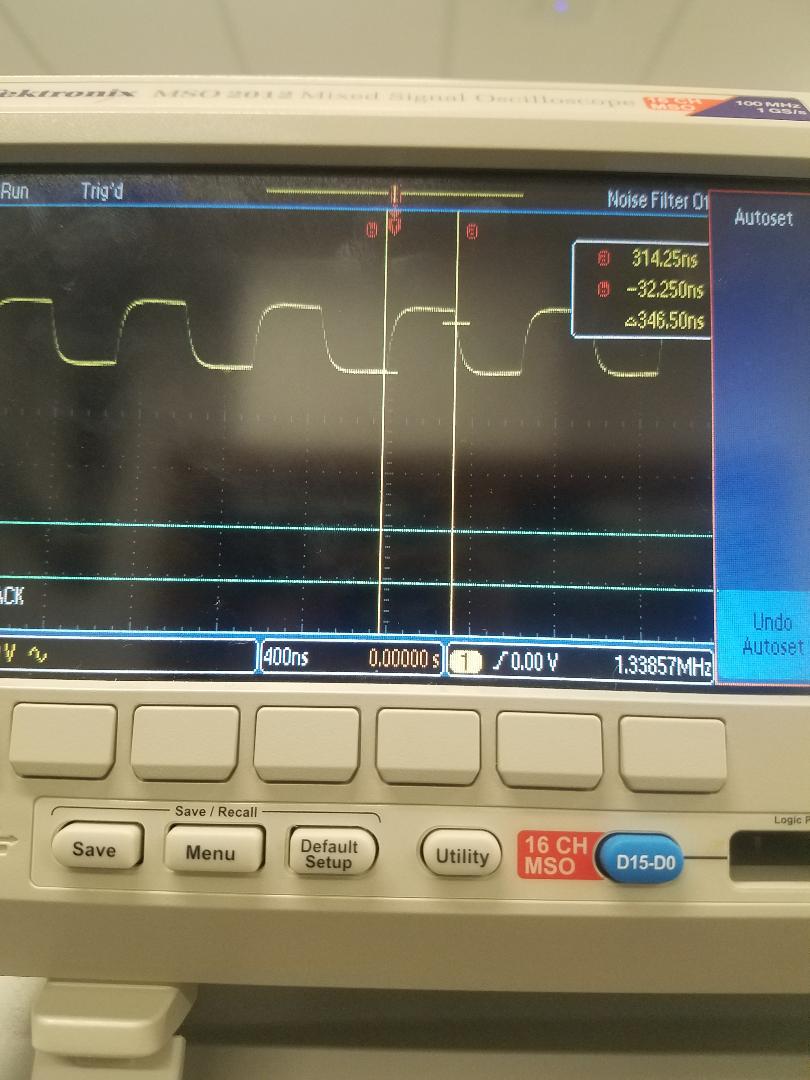
d) How would you have written the compiler to remove an instruction?

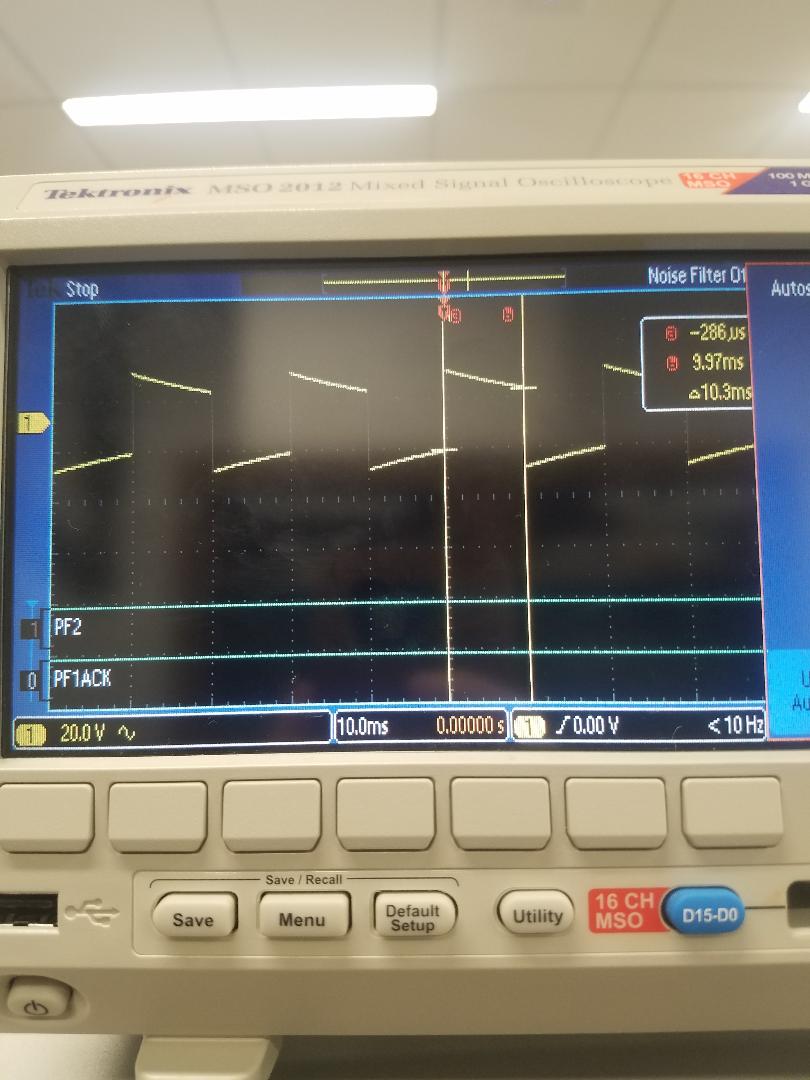
Ans: Change 1st LDR’s r0 to r1 and get rid of 3rd LDR

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.

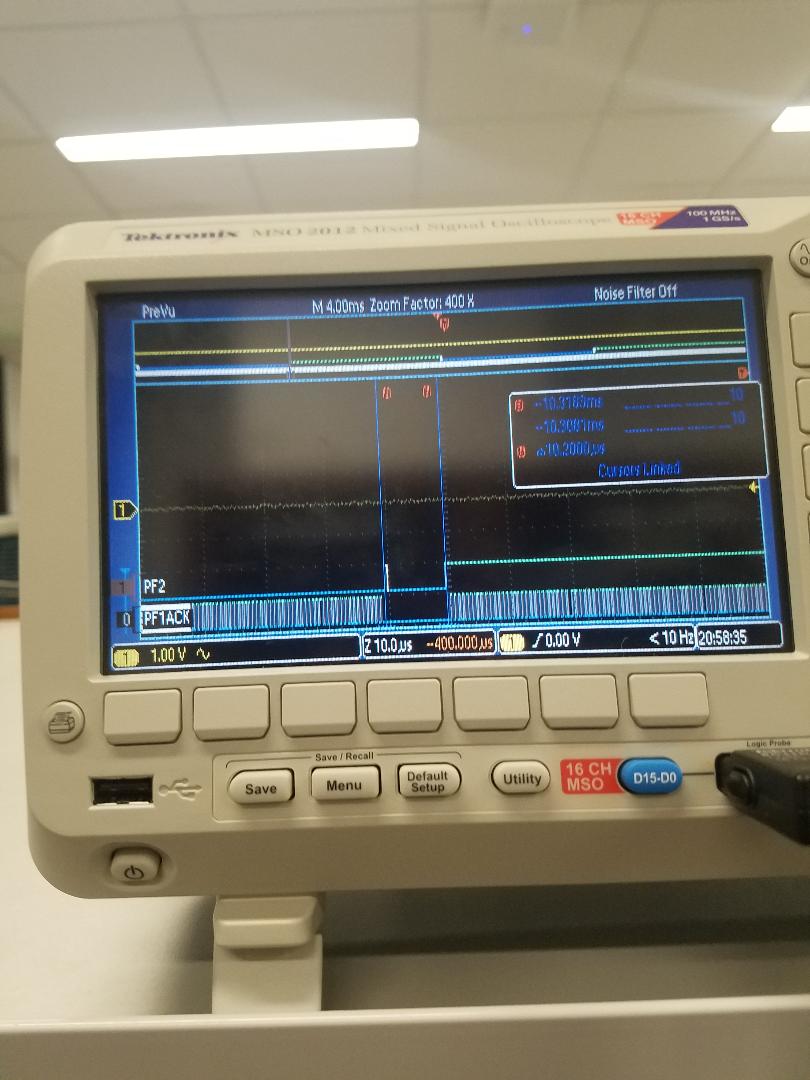
**Part A)** Debugging profile with scope





**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

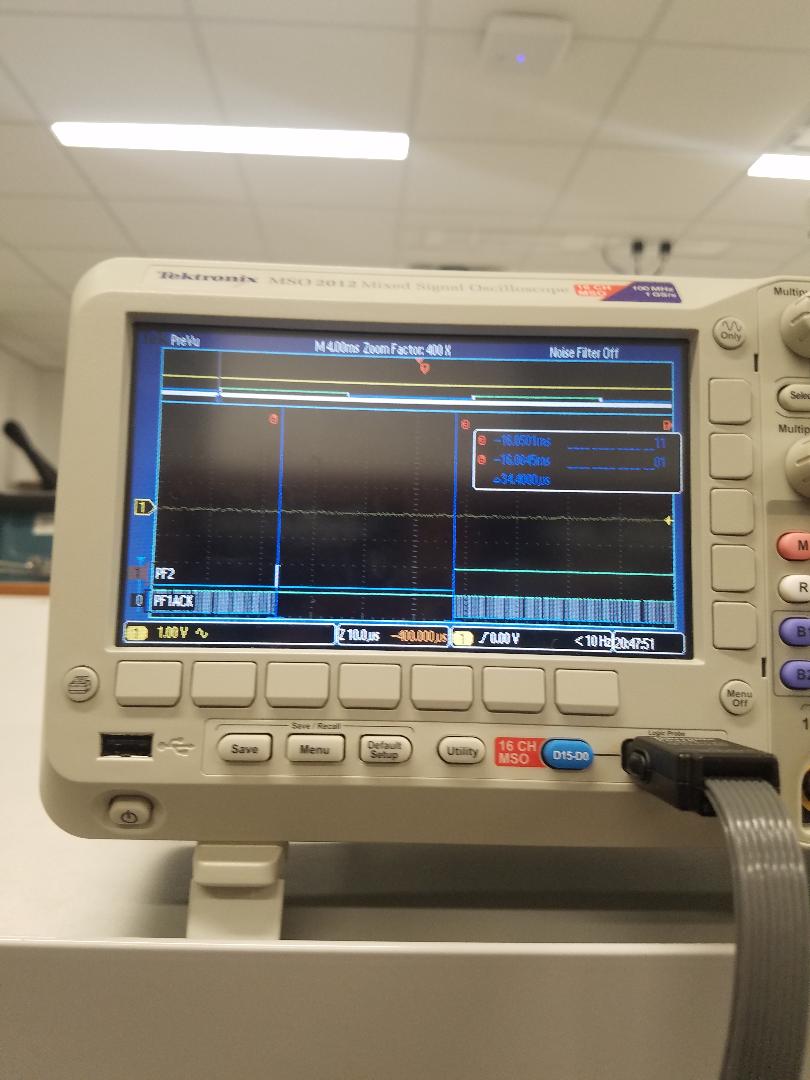
Time jitter was 4 by the code we ran. Time jitters are caused because the ISR waits for the current instruction to complete. This causes a delay in the time calculation which is corrected in the next trigger when it is lower than normal to correct for the first delay. Thus there is an even time jitter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

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

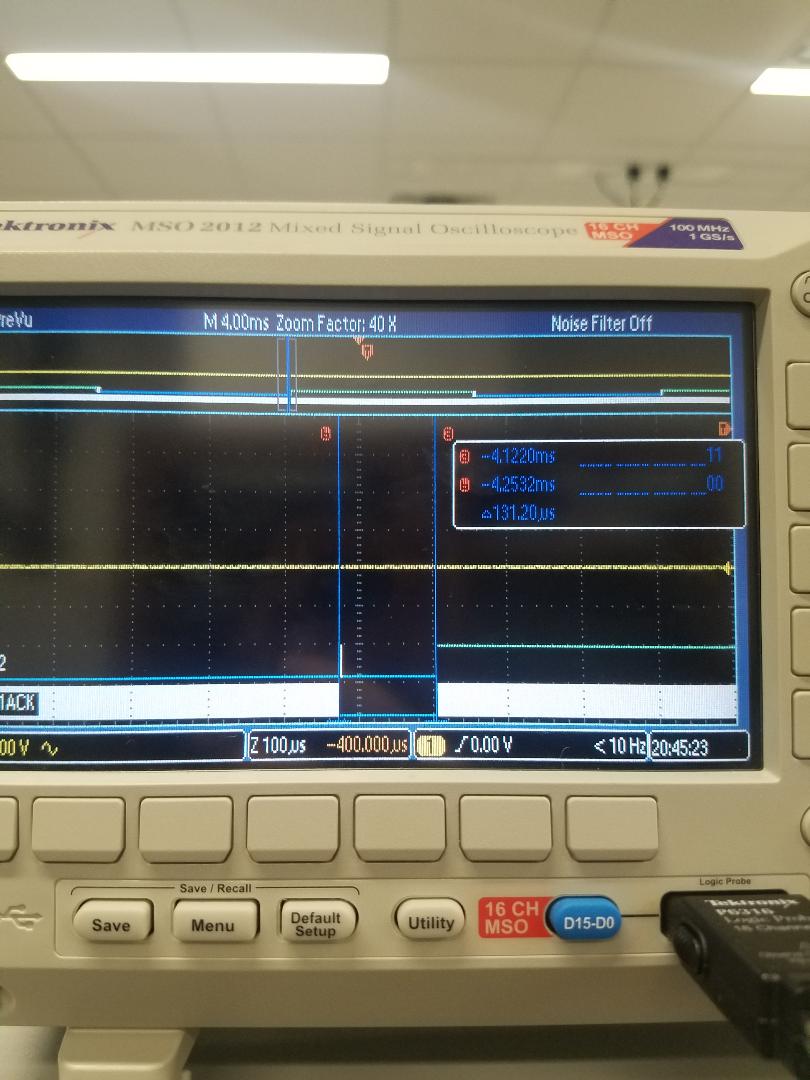
**The graphs do support our hypothesis with the CLT as initially without the averaging the variance was large but with the averaging the variance collapses onto the real value that it is supposed to be. This is exactly what CLT says would happen**

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

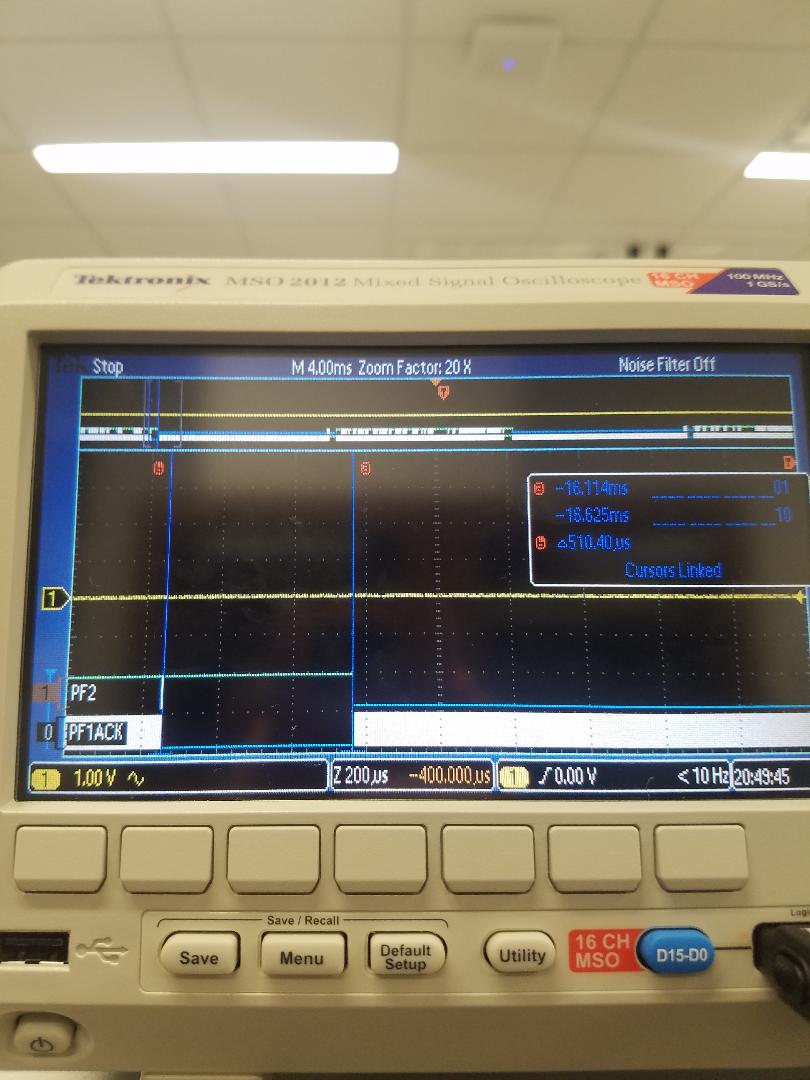
4X Averaging:



16X:



64X:



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