Daniel Diamont

Robert Noe

Lab 2 Report

Goals:

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

DCD statements are there to provide access to the following memory-mapped I/O: RCGCGPIO (0x400F.E608), GPIODIR (0x4002.5400), PORTF\_BASE (0x4002.5000)

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

PF1 will toggle every 150 nanoseconds

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

R0 has the address of the Port F GPIO Data register after the first LDR instruction. After the second LDR instruction, R0 has the state of PF1 with the state of all the other pins in port F shown as cleared.

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

I would have written the compiler to fetch the state of PF1 in one instruction:

0x0000068C DCW 0x4002

DCW 0x5008

LDR R0, [pc, #24];

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?

Since the main program is only interested in PF1 by using bit-specific addressing, any changes made by the ISR to PF2, which also uses bit specific addressing, will not affect the main program’s read and write to PF1. In other words, there is no critical section between the two threads.

**Software Design (shown below is main; all files are in zip file)**

// main.c

// Name: Daniel Diamont

// TA: Saadallah Kassir

// Created: 1/30/2018

// Last Modified: 1/30/2018

// 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 PE2/AIN1

// bottom of X-ohm potentiometer connected to ground

// top of X-ohm potentiometer connected to +3.3V

#include <stdint.h>

#include <stdbool.h>

#include "ADCSWTrigger.h"

#include "tm4c123gh6pm.h"

#include "PLL.h"

#include "ST7735.h"

#include <math.h> //absolute value

#include "fixed.h"

#include "SysTickInts.h"

#define PF4 (\*((volatile uint32\_t \*)0x40025040))

#define PF3 (\*((volatile uint32\_t \*)0x40025020))

#define PF2 (\*((volatile uint32\_t \*)0x40025010))

#define PF1 (\*((volatile uint32\_t \*)0x40025008))

#define PF0 (\*((volatile uint32\_t \*)0x40025004))

//function declarations

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

void Get\_Time\_Differences(void); //calculate the time differences from the Debug\_Buffer

uint32\_t Get\_Time\_Jitter(void); //calculate the time jitter from the Time\_Difference buffer

uint32\_t Get\_Average(void); //calculate the mean of the ADC data so that we can create the plot

void Plot\_PMF(void);

void Pause(void);

void DelayWait10ms(uint32\_t n);

//Debug Buffer Set up

#define BUFFER\_SIZE 1001

#define BUFFER\_LEN 2

#define time 0

#define data 1

#define MAX\_ALLOWABLE\_JITTER 64 // 800 ns / 12.5 ns intervals = 64.

uint32\_t Debug\_Buffer[BUFFER\_SIZE][BUFFER\_LEN]; //where [t][i][0] is the time and [t][i][1] is the data

uint16\_t Debug\_Cnt = 0;

bool Data\_Collection\_Complete = false;

bool shouldRead = false;

//global for passing information between threads

volatile uint32\_t ADCvalue;

uint32\_t Time\_Difference[BUFFER\_SIZE-1];

uint32\_t Time\_Jitter;

void Debug\_Dump(uint8\_t n){

if(Debug\_Cnt < BUFFER\_SIZE){

switch (n){

case (time):

Debug\_Buffer[Debug\_Cnt][time] = TIMER1\_TAR\_R;

break;

case (data):

Debug\_Buffer[Debug\_Cnt][data] = ADCvalue;

Debug\_Cnt++;

break;

}

}

else

Data\_Collection\_Complete = true;

}

// This debug function initializes Timer0A to request interrupts

// at a 100 Hz frequency. It is similar to FreqMeasure.c.

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){

TIMER0\_ICR\_R = TIMER\_ICR\_TATOCINT; // acknowledge timer0A timeout

PF2 ^= 0x04; // profile

Debug\_Dump(time);

PF2 ^= 0x04; // profile

ADCvalue = ADC0\_InSeq3(); //read new value from the ADC sequencer

Debug\_Dump(data);

PF2 ^= 0x04; // profile

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 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(){

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 = 0xFFFFFFFF; // 4) reload value

TIMER1\_TAPR\_R = 0; // 5) bus clock resolution

TIMER1\_ICR\_R = 0x00000001; // 6) clear TIMER1A timeout flag

TIMER1\_CTL\_R = 0x00000001; // 10) enable TIMER1A

}

int main(void){

PLL\_Init(Bus80MHz); // 80 MHz

SYSCTL\_RCGCGPIO\_R |= 0x20; // activate port F

while((SYSCTL\_PRGPIO\_R&0x20)==0){}; // allow time for clock to start

ST7735\_InitR(INITR\_REDTAB); //initialzie LCD screen

ST7735\_FillScreen(ST7735\_BLACK);

//initialize Port F

GPIO\_PORTF\_DIR\_R |= 0x0D; // make PF2, PF1, PF3 out (built-in LED)

GPIO\_PORTF\_PUR\_R |= 0x10; // 5) pullup for PF4

GPIO\_PORTF\_AFSEL\_R &= ~0x1D; // disable alt funct on PF1 PF2, PF3 and PF4

GPIO\_PORTF\_DEN\_R |= 0x1D; // enable digital I/O on PF1-PF4

// configure PF1-PF4 as GPIO

GPIO\_PORTF\_PCTL\_R = (GPIO\_PORTF\_PCTL\_R&0xFFFFF00F)+0x00000000;

GPIO\_PORTF\_AMSEL\_R = 0; // disable analog functionality on PF

ST7735\_OutString("Test Lines\nDraw a Star!\nDraw an X!");

ST7735\_PlotClear(0,BUFFER\_SIZE/2); // range from 0 to BUFFER\_SIZE

//Star Drawing

ST7735\_Line(50,50,100,100,ST7735\_BLUE); //negative slope

ST7735\_Line(50,100,100,50,ST7735\_BLUE); //positive slope

//ST7735\_Line(0,100,50,120,ST7735\_BLUE); //random

ST7735\_Line(75,50,75,100,ST7735\_BLUE);//vertical line

ST7735\_Line(50,75,100,75,ST7735\_BLUE); //horizontal line

//X drawing

ST7735\_Line(30,90,10,70,ST7735\_BLUE);

ST7735\_Line(30,70,10,90,ST7735\_BLUE);

Pause();

ST7735\_FillScreen(ST7735\_BLACK);//clear screen

ST7735\_SetCursor(0,0);

ST7735\_OutString("Acquiring data...");

ADC0\_InitSWTriggerSeq3\_Ch9(); // allow time to finish activating

PF2 = 0; // turn off LED

//SysTick\_Init(7920); //Systick interrupts every 99 us

Timer0A\_Init100HzInt(); // set up Timer0A for 100 Hz interrupts

Timer1\_Init(); // set up Timer1A to return the current time up to 53 s in 12.5 ns units

EnableInterrupts();

while(1){

PF1 ^= 0x02; // toggles when running in main

//GPIO\_PORTF\_DATA\_R ^= 0x02;

//PF1 = ((PF1\*12345678)/1234567+0x02)\*(PF1/(1234567+0x02)); // this line causes jitter

if(Data\_Collection\_Complete){ //begin processing

long sr = StartCritical(); //disable interrupts

ST7735\_FillScreen(ST7735\_BLACK);

Get\_Time\_Differences();

Time\_Jitter = Get\_Time\_Jitter();

ST7735\_SetCursor(0,0);

ST7735\_OutString("Time Jitter: ");

ST7735\_OutUDec(Time\_Jitter);

if(Time\_Jitter > MAX\_ALLOWABLE\_JITTER){

ST7735\_OutString("\ntoo much jitter...");

}

ST7735\_OutString("\nPMF w/ HW avg: ");

ST7735\_OutUDec(ADC0\_SAC\_R);

ST7735\_OutString("\nVoltage: ");

int averageVoltage = (Get\_Average()\*3300+2048)/4096; //fixed point

ST7735\_sDecOut2(averageVoltage/10);

ST7735\_OutString(" V");

DelayWait10ms(20); //wait two seconds

//plot PMF on LCD Screen

ST7735\_PlotClear(0,BUFFER\_SIZE/2); // range from 0 to BUFFER\_SIZE

ST7735\_SetCursor(0,0);

Plot\_PMF(); //plot PMF on the screen

//HW averaging factor increment request by the user... (switch at PF4)

Pause();

ST7735\_FillScreen(ST7735\_BLACK);

//increase HW averaging factor

ADC0\_ACTSS\_R &= ~0x0008; // disable sample sequencer 3

ADC0\_SAC\_R = (ADC0\_SAC\_R + 2) % 8; //wrap around

ADC0\_ACTSS\_R |= 0x0008; // enable sample sequencer 3

Debug\_Cnt = 0;

Data\_Collection\_Complete = false;

ST7735\_OutString("Acquiring data...");

EndCritical(sr);

}

}

}

void Get\_Time\_Differences(void){

//calculate time differences

for(int i = 0; i < BUFFER\_SIZE - 1; i++){ //total of 999 computations

Time\_Difference[i] = Debug\_Buffer[i][time] - Debug\_Buffer[i+1][time]; //compute the difference between time dump items

}

}

uint32\_t Get\_Time\_Jitter(void){

uint32\_t max;

uint32\_t min;

if(Time\_Difference[2] >= Time\_Difference[1]) {

max = Time\_Difference[2];

min = Time\_Difference[1];

}

else

{

min = Time\_Difference[2];

max = Time\_Difference[1];

}

//find min and max

for(int i= 3; i < BUFFER\_SIZE-1; i++){

if(Time\_Difference[i] < min) min = Time\_Difference[i];

else if(Time\_Difference[i] > max) max = Time\_Difference[i];

}

return max-min;

}

uint32\_t Get\_Average(void){

uint32\_t avg = 0;

uint32\_t sum = 0;

for(int i=1; i < BUFFER\_SIZE; i++){

sum += Debug\_Buffer[i][data];

}

avg = sum/(BUFFER\_SIZE-1);

return avg;

}

void Plot\_PMF(void){

{

//make an array of all the data points to plot

int freq[127];

int offset = 64; //128/2 = 64

int count = 0;

int avg = Get\_Average();

for(int i=0; i < 127;i++){

count = 0;

for(int j=1;j< BUFFER\_SIZE; j++){

if(Debug\_Buffer[j][data] == avg-offset+i)

count++;

}

freq[i] = count;

count = 0;

}

for(int j=0;j<127; j++){

ST7735\_PlotBar(freq[j]); // called BUFFER\_SIZE times

ST7735\_PlotNext();

}

}

}

void Pause(void){

while(PF4==0x00){

DelayWait10ms(10);

}

while(PF4==0x10){

DelayWait10ms(10);

}

}

void DelayWait10ms(uint32\_t n){

uint32\_t volatile timer;

while(n){

timer = 727240\*2/91; // 10msec

while(timer){

timer--;

}

n--;

}

}

void SysTick\_Handler(){

PF3 ^= 0x04;

PF3 ^= 0x04;

// int a = 123098;

// a = (a\*6808080)/2440404040;

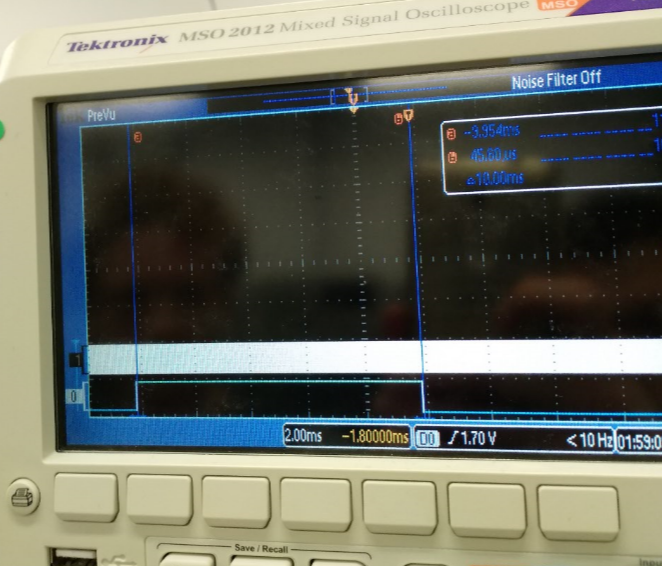
PF3 ^= 0x04;

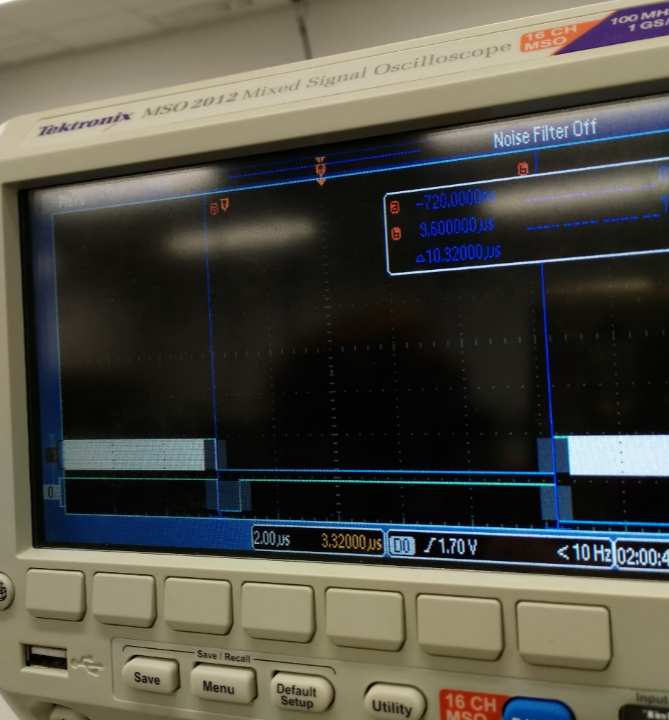
}

**Part A)** Debugging profile with scope



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





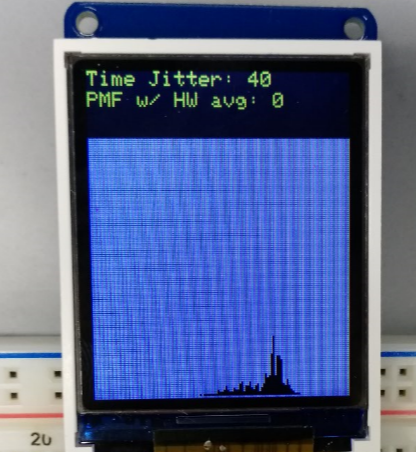
Let *t,* be the percentage time spent in ISR and away from main thread.

*%t =* (10.32 us / 10.00 ms)\*100 = 0.1032 %

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

The critical section occurs because even though the main thread is using bit-specific addressing to toggle PF1, the ISR reads and writes to the entire Port F in order to toggle PF2. Thus, anywhere in between the read-modify-write cycle to the port during the ISR, there can develop a race condition between the threads. Two ways to remove this problem are: A) ensure that the most important thread is atomic, thereby ensuring the same outcome in every situation. B) Use bit-specific addressing to completely avoid a critical section by reducing each threads access to a specific pin, rather than reading and writing to the entire port.

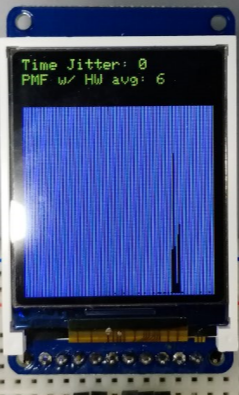
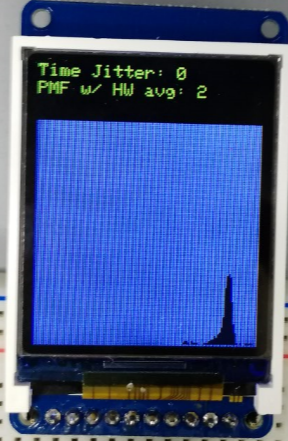
**Part D)** Time-jitter measurements and generalization of factors that contribute to jitter



The above photos are examples of two trials runs in which our software detected noticeable differences between the maximum and minimum time between threads. In the first case, there is a jitter of 8 bus cycles (100 nanoseconds), and in the second case there is time jitter of 40 bus cycles (500 nanoseconds).

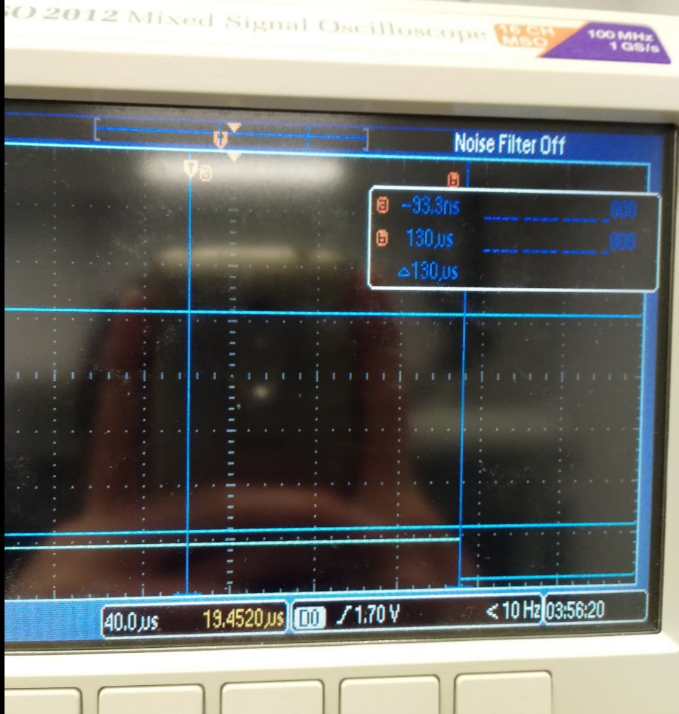
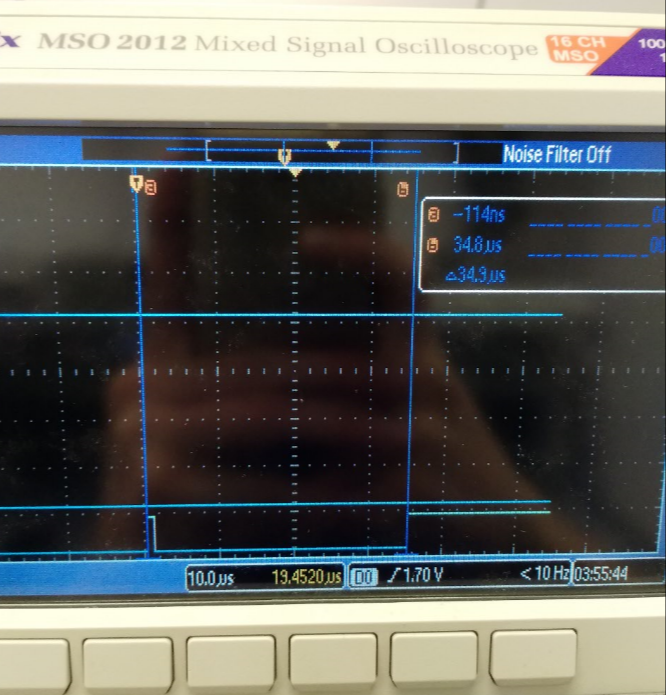
Some factors that can contribute to jitter include programming techniques for interfacing with data acquisition hardware that include busy waiting or having interrupt services routines which take up a noticeable percentage of time away from the main program. Additionally, adding more interrupts increases the changes that the time critical interrupt will not be serviced under a hard-real-time constraint, as the processor may be occupied servicing higher priority tasks. Lastly, jitter may be caused by a processor executing instructions with a high average quantity of bus cycles to execute, such as integer division which takes about 2 to 12 bus cycles. This can delay the interrupt from being serviced with a hard deadline.

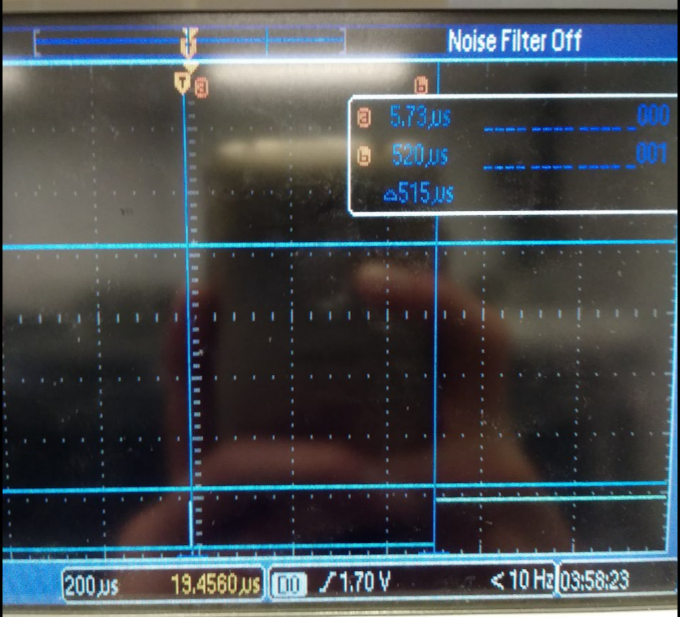
**Part E and F)** PMF data and discussion of results. Does your data support CLT? If not why?



Yes, our data supports the Central Limit Theorem. By increasing the averaging factor of our data acquisition hardware, the standard deviation of our Gaussian distribution decreased. In other words, when we took an increasing number of sample points, the error increased, and the data points converged closer towards the mean of the distribution. This supports central limit theorem which argues that as the number of random variables, or independent samples, that are added together increases to infinity, the probability mass function will generate a plot that approaches that of a normal distribution.

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





The debugging profiles are different because the increased factor for hardware averaging causes the hardware to take longer to sample the signal. Thus, the interrupt service routine must busy wait the hardware for information for larger periods of time, given a larger hardware averaging factor to take more sample points.