EE445L – Lab 2: Performance Debugging

Michael Park and Jack Zhao
02/12/16

1.0 OBJECTIVE

The objective of this lab is to experience the various methods of debugging on the TM4C123 board. The way we tested the debugging methods is the implementation of various interrupts for several purposes. Our primary objective was to obtain ADC and timer values so that we could store them into data dumps. We could then process that data using graphic PMFs and jitter calculations. In addition, we toggled the LED from PF1 and PF2 to aid in the debugging process.

2.0 MEASUREMENT DATA

(Prep Part 2)

1. What is the purpose of all the DCW statements?

DCW places the 16-bit half-word into memory. This is useful because it allows us to assist and control the assembly process. In this case this is helpful in storing large expressions into memory for us to call and use later, even if a certain execution requires more space to run than is available in the registers.

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

The main program executes at a clock rate of 80Mhz frequency. This means that the program will execute a line every 2 clock cycles or 2* 12.5 ns. If there are 6 asm lines to execute the while loop, the program will toggle PF1 at approximately every 150 ns (2 Instructions per 1 cycle so 12.5*2*6 = 25*6 = 150ns).

3. What is in R0 after the first LDR is executed? What is in R0 after the second LDR is executed?

After the first LDR is executed 4002500 will be stored, after the second LDR is executed, whatever is in the address 40025008 will be stored into R0 because the second LDR indexes the previous R0 by #0x08.

4. How would you have written the compiler to remove an instruction?

Remove the line “0x00000686 4904 LDR r1,[pc,#16] ; @0x0000068C” and chnage the first regiter from r0 to r1.

5. 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?

Yes the non-reentrant read-modify-write sequence creates a critical section. We can eliminate the critical section by either removing the access to the global variables (PF2, ADCvalue) or implementing mutual exclusion. Since we need the global variables for the desired function, we can create a mutual exclusion by disabling the interrupt before measuring the ADC value (critical section) and then enable it right after.

(Part A)

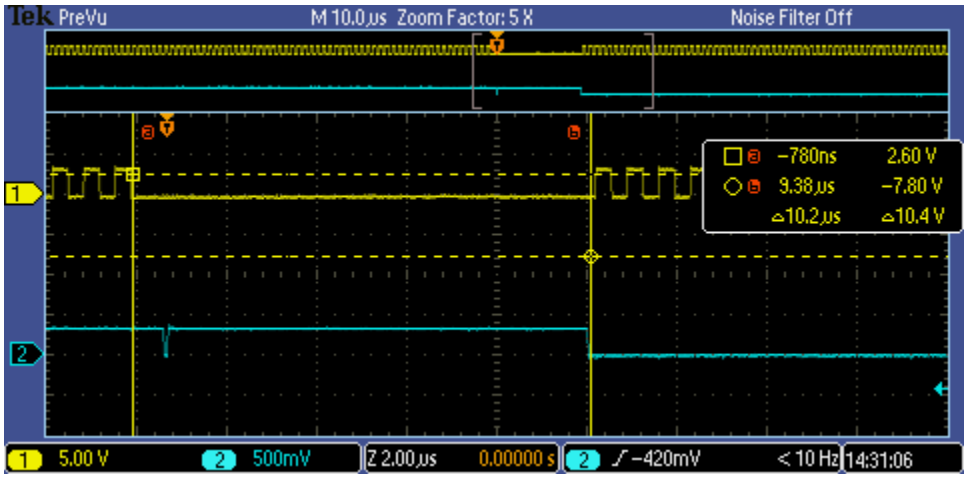

Figure 1: Debugging feature using oscilloscope

(Part B)

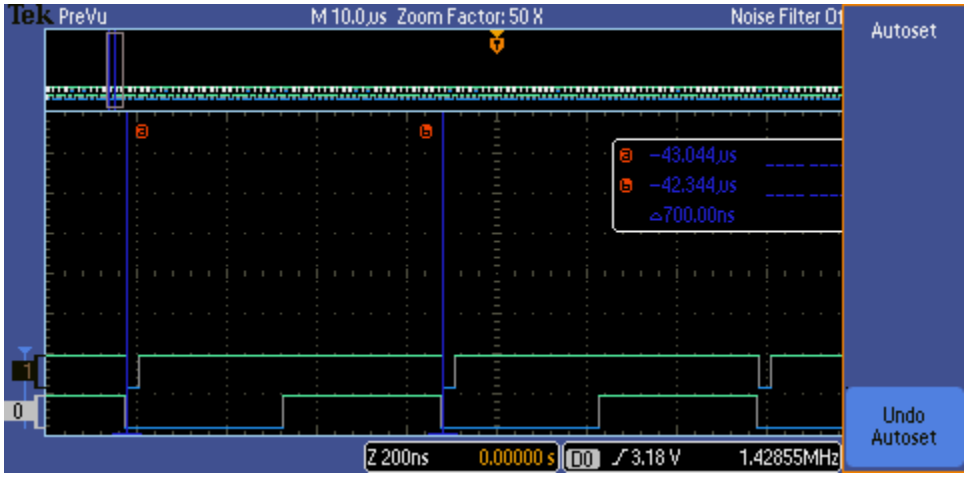

Figure 1.2: Debugging feature using logic analyzer

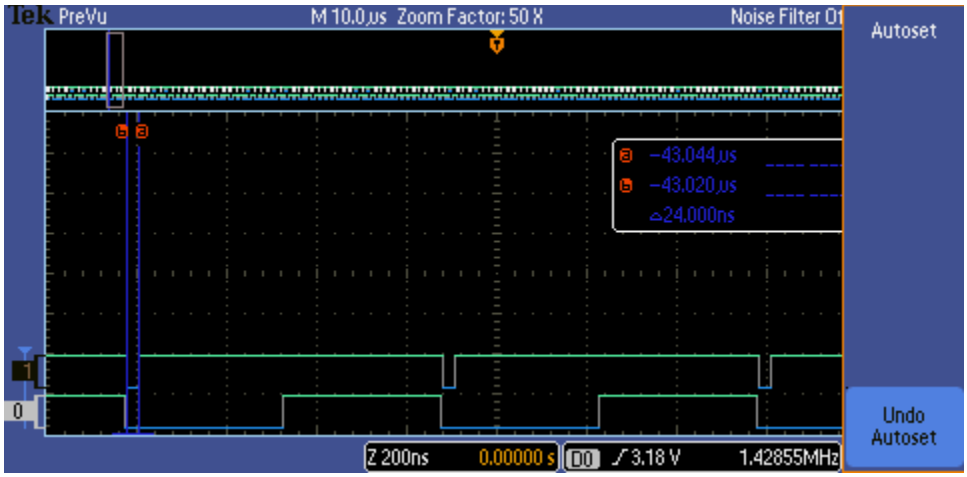

Figure 1.3: Debugging feature using logic analyzer

Result: 3.4% of a clock cycle is dedicated for the ISR.

96.6% of a clock cycle is dedicated for the main.

(Part C)

PF 2 is incorrect because this line of code indicates a causal dependency. Originally only PF1 would be toggled, but now the entire PortF is being toggled, which may or may not overwrite the interrupts toggle of PF2. Therefore, PF2 is incorrect.

In the original code, the program will toggle r0 by continuously loading the previous value from memory into r0, and will EOR that value with 0x02. Then it will use R1 as a placeholder to store this value back into the memory location that R0 accesses in the next cycle of the program

```
0x0000134C 480E      LDR        r0,[pc,#56]  ; @0x00001388
0x0000134E 6800      LDR        r0,[r0,#0x00]
0x00001350 F0800002  EOR        r0,r0,#0x02
0x00001354 490B      LDR        r1,[pc,#44]  ; @0x00001384
0x00001356 F8C103FC  STR        r0,[r1,#0x3FC]
```

In the new code, the value of R0 is constantly incrementing, and this is due to the retrieval process not retrieving the correct new R0, which should always be 0x00 or 0x02. In this case, the value of R0 increments till it is 0x08. This is not the proper way to toggle an led because the other bits, may have some significance elsewhere.

Other than bit specific addressing, another method to prevent this causal dependency would be to initiate an atomic portion of this chunk of code, so that only one thread may access this portion on the read-write phase.

(Part D)

Before Adding (PF1 = (PF1*12345678)/1234567+0x02;)

-Jitter 1: 4 cyles = 4*12.5ns = 50ns

-Jitter 2: 4 cyles = 4*12.5ns = 50ns

-Jitter 3: 4 cyles = 4*12.5ns = 50ns

After Adding (PF1 = (PF1*12345678)/1234567+0x02;)

-Jitter 1: 0x14 cyles = 20*12.5ns = 250ns

Activated Timer1 and Timer2 (99us)

-Jitter: 0x0C cyles = 12*12.5ns = 150ns

Deliverable: Measure time jitter with two or more interrupts active. Try to generalize the results deriving a theoretical estimate of the time jitter of the periodic ADC sampling using software triggering.

Theoretically, since we added an extra timer with higher priority, we can generalize the jitter to be the following: result = ADC sampling jitter + Timer2 Interrupt

(Part E and F)

How would you describe the shape of the noise process?

The shape of the noise graph represents a Gaussian Shape. The most frequent value is most likely the real value.

When you run it over and over again, do you get the same shape of the PMF?

The shape of the PMF changes every time the program is run.

Describe qualitatively the effect of hardware averaging on the noise process. Consider two issues 1) the shape of the PMF and 2) the signal to noise ratio.

The more samples we take for hardware averaging, the less standard deviation becomes. In PMF, with higher hardware averaging, the bell curve of the Gaussian shape becomes narrower, which means the central limit becomes more apparent. This also means that we get far more values that are closer to the real value. Thus higher signal to noise ratio (more signal and less noise).

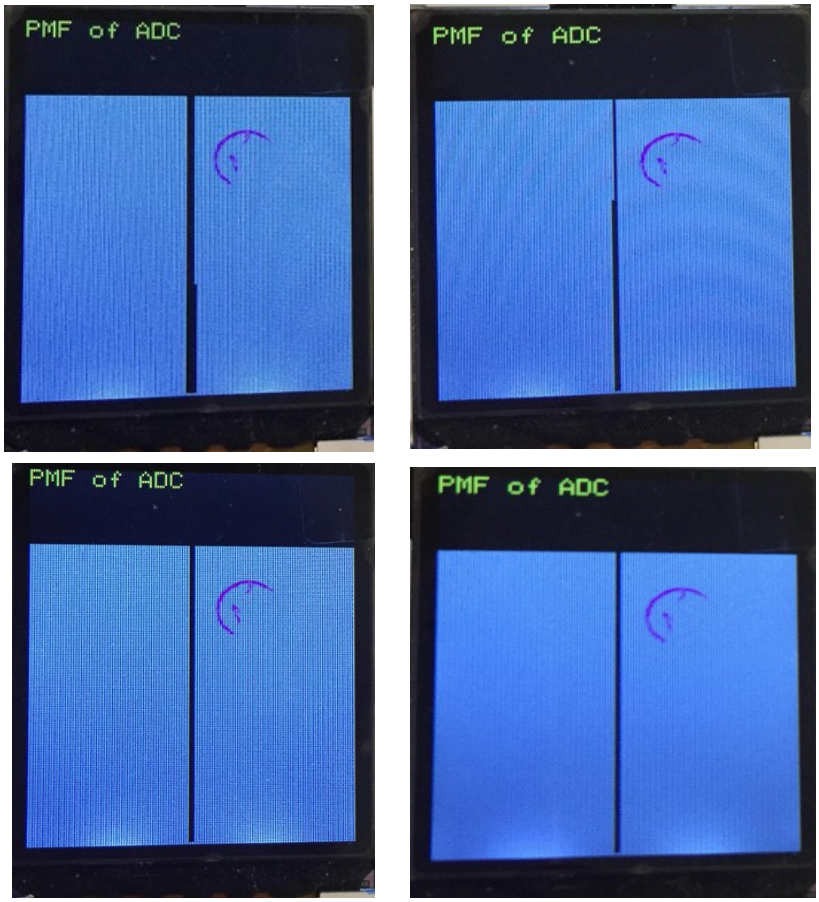

Figure 1.4: PMF for hardware averaging in order of x0, x4, x16, x64

(Part F)

Why is the thread profile like Figure 2.1 very different with hardware averaging?

With hardware averaging, it takes much longer to collect data. Therefore longer time is spent in the ISR.

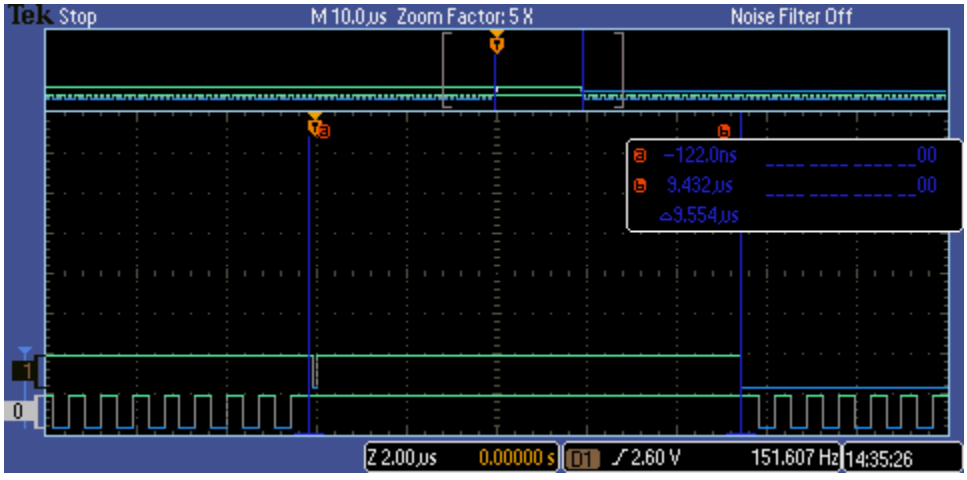

Figure 1.5: Debugging profile with no hardware averaging

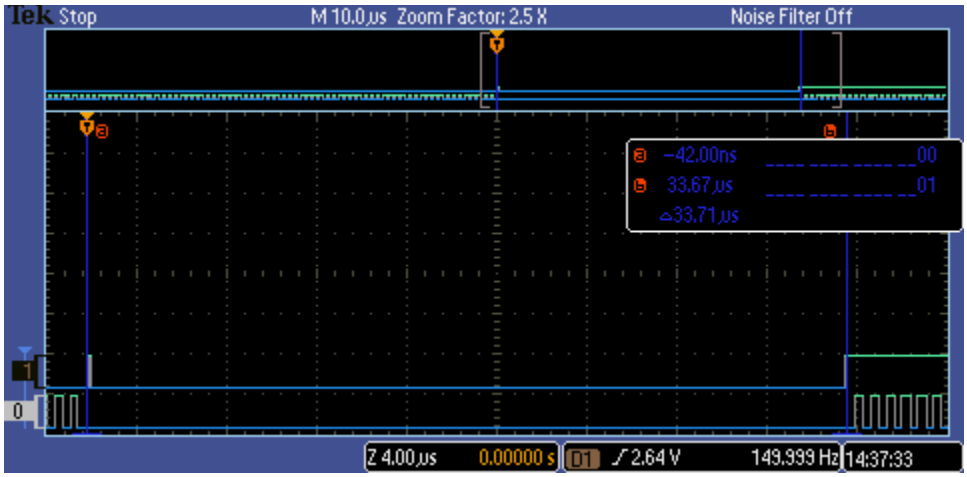

Figure 1.6: Debugging profile with x4 hardware averaging

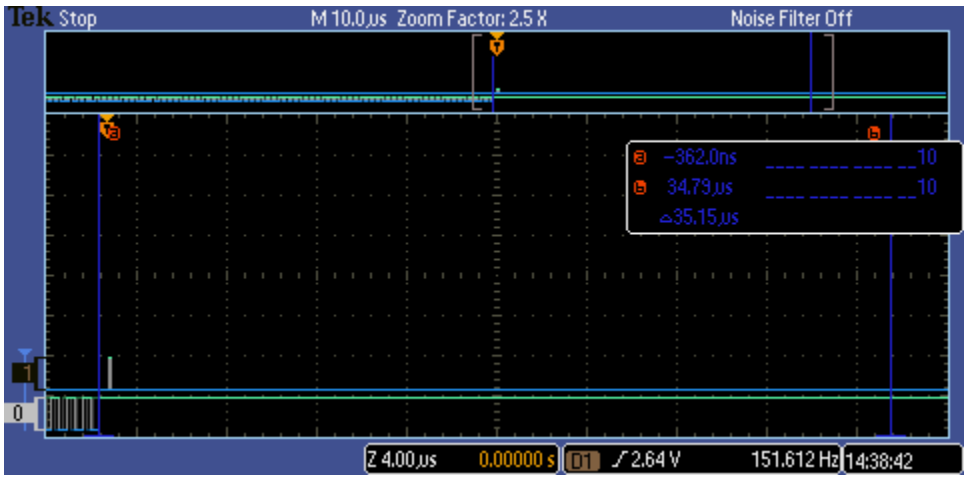

Figure 1.7: Debugging profile with x16 hardware averaging

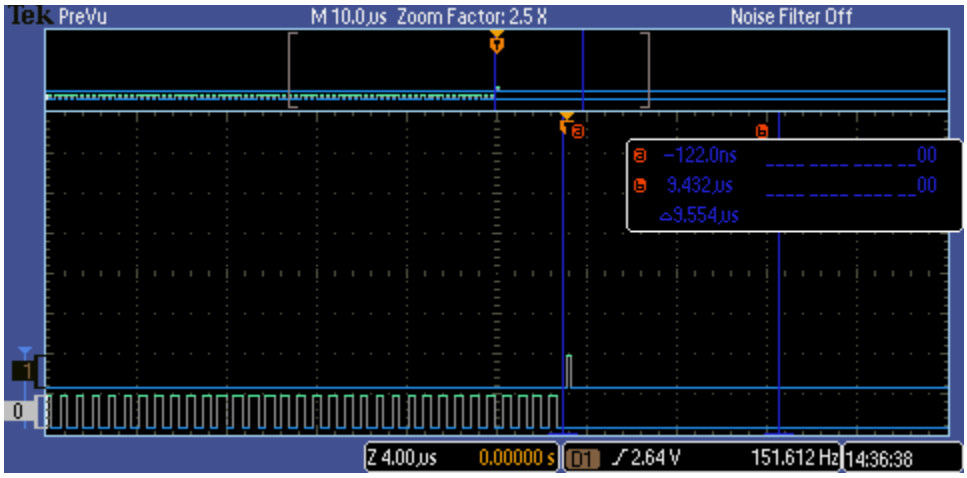

Figure 1.8: Debugging profile with x64 hardware averaging

3.0 ANALYSIS AND DISCUSSION

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

This debugging method is minimally intrusive because in implementing it, there will be a running time of a few cycles to run the code, which MAY impact results but is not probably. The effect on the system is almost negligible in this case.

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?

In some cases, data dumps would be better than implementing printf statements because data dumps allow us to implement various data processing methods into our program so that we could extract key pieces of information such as the jitter and maximum frequency of appearance. On the other hand, printf statements may be better in some cases where we need to check the specific values being outputted, in which case we could just view the outputs of the program vs having to go in the debugger and look through the array. A third case could be implemented such that a data dump would be created and this data dump is printed out with a series of printf statements in which case, this would contain the advantages of both types of debugging but would alternatively require too much processing time.

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?

The necessary conditions for a critical section to occur is that a multi-process program to not allow concurrent execution by more than one of the programs processes. Some software activities that may result in a critical section are pieces of programs that require exclusion of access to a particular part, such as using a shared resource in the program, whether it be a shared data structure or shared memory.

4. Define “minimally intrusive”.

Minimally intrusiveness is a debugging categorization that measures how efficient a debugging process is. Specifically, a minimally intrusive debugging method is one that the process does have an impact on the system but this effect is negligible. Generally, the existence of a minimally intrusive process will have minimal effect on the parameters being measured (ie. Execution time of the debugging portion is small compared to the execution time of the process being measured). Another classification of minimally intrusiveness is if the debugging time ratio is 1/1000 to the original program.

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?

The reason that we do not always use hardware averaging is as it states; it implements an average. Generally hardware averaging is to obtain a mass number of ADC values more efficiently and with less noise than generally can be done. The problem is that in some cases, we will need to view the noise and individual values of ADC outputs to properly process some data. Therefore, we cannot use hardware averaging options in these cases. In addition, the use of Hardware averaging leads to much more time consumption in the interrupt call.

4.0 Code Written for this Lab

```
// 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

// ADCTestMain.c
// Michael Park, Jack Zhao
// Date Modified: 02/2/16
// Main file. Includes hardware averaging options, jitter calculation function, adc plot function, and draw line function.
// Lab Number: 16340
// TA: Mahesh Srinivasan
// Last Revised: 02/8/16

/* 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 "ADCSWTrigger.h"
#include "../Shared/tm4c123gh6pm.h"
#include "PLL.h"
#include "Timer1.h"
#include "Timer2.h"
#include <stdlib.h>
#include <stdio.h>
#include "ST7735.h"
#include "tm4c123gh6pm.h"

#define PF2             (*((volatile uint32_t *)0x40025010))
#define PF1             (*((volatile uint32_t *)0x40025008))
#define DEBUG    //Debugging Feature for collecting 1000 samples. Uncomment to Enable the debugging feature

typedef int bool;
#define true 1
#define false 0

//********function prototypes*************************************************************//
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 calcTimeDif(void);
void adcPMF(void);
void partG(void);

//*******global variables*************************************************************************//
volatile uint32_t ADCvalue;
uint32_t time_dump[1000], adc_dump[1000], sorted_ADC[1000], yBuffer[4096];
static uint32_t count=0;
volatile bool flag=false;

//**cmpfunc for qsort************************//
int cmpfunc (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

// 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
  PF2 ^= 0x04;                  // profile
  ADCvalue = ADC0_InSeq3();

#ifdef DEBUG                                          //debugging feature for collecting 1000 samples
            if(count<1000){
                        PF1 = (PF1*12345678)/1234567+0x02;  // this line causes jitter
                        adc_dump[count]=ADCvalue;
                        time_dump[count] = TIMER1_TAR_R;
                        count++;
            }
            else{
                        TIMER1_IMR_R &= 0x11111110;
                        TIMER0_IMR_R &= 0x11111110;
                        flag=true;
            }
#endif
  PF2 ^= 0x04;                  // profile
}
int main(void)
{
  PLL_Init(Bus80MHz);                 // 80 MHz
  SYSCTL_RCGCGPIO_R |= 0x20;          // activate port
            ST7735_InitR(INITR_REDTAB);
  ADC0_InitSWTriggerSeq3_Ch9();       // allow time to finish activating
  Timer0A_Init100HzInt();             // set up Timer0A for 100 Hz interrupts
            Timer1_Init(0,12);
```

```
//          Timer2_Init(0,99000);
  GPIO_PORTF_DIR_R |= 0x06;          // make PF2, PF1 out (built-in LED)
  GPIO_PORTF_AFSEL_R &= ~0x06;       // disable alt funct on PF2, PF1
  GPIO_PORTF_DEN_R |= 0x06;          // 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

//*****Hardware Averaging Option*********************//
          ADC0_SAC_R &= 0xFFFFFFF8;        //no hardware oversample
          //ADC0_SAC_R &= 0xFFFFFFF8; //4x hardware oversample
          //ADC0_SAC_R |= 0x2;
          //ADC0_SAC_R &= 0xFFFFFFF8;                  //16x hardware oversample
          //ADC0_SAC_R |= 0x4;
//          ADC0_SAC_R &= 0xFFFFFFF8; //64x hardware oversample
          //ADC0_SAC_R |= 0x6;

  EnableInterrupts();

  while(1)
          {
    PF1 ^= 0x02;  // toggles when running in main
                  //GPIO_PORTF_DATA_R ^= 0x02;    //toggles when running in main
                  if(flag == true)
                  {
                          calcTimeDif();
                          adcPMF();
                          flag=false;
                          partG();
                  }
  }
}
```

```
//*********calcTimeDif function*********************
//This is used along with the debugging feature implemented inside Timer0A_Handler.
//This function calculates the jitter of collected ADC value by maxtime - mintime.
void calcTimeDif(void)
{
	uint32_t timeDif[999];
	uint32_t min=0, max=0;
	uint32_t jitter;
		uint32_t hi=0;

	for(int i =1;i<1000;i++){
		timeDif[i-1]=time_dump[i-1]-time_dump[i];
		if(timeDif[i-1]==0){
			hi++;
		}
	}
	min=timeDif[0];
	max=timeDif[0];
	for(int j=1;j<999;j++){
		if(timeDif[j]>max){
			max=timeDif[j];
		}
		if(timeDif[j]<min){
			min =timeDif[j];
		}
	}
	jitter=max-min;
	hi++;
}


//*********adcPMF function*********************
//This is used along with the debugging feature implemented inside Timer0A_Handler.
```

```
//This function calculates the frequency of each ADC value and outputs a PMF to ST7735 LCD screen.
void adcPMF(void){
//Function for building the PMF on the LCD

uint32_t counter=0;                     //counter for the yBuffer
uint32_t range=0;
uint32_t maxfreq=0;
uint32_t maxfreqpos=0;

//sort ADC values
        for(int i =0;i<1000;i++){
                sorted_ADC[i]=adc_dump[i];
        }
        qsort(sorted_ADC, 1000, sizeof(uint32_t), cmpfunc);
        range=sorted_ADC[999]-sorted_ADC[0];

//create xBuffer and yBuffer
        for(int j=0; j<4096; j++)
        {
                while( (sorted_ADC[counter]==j) && (counter<1000) )
                {
                        yBuffer[j]++;
                        counter++;
                        if(yBuffer[j]>=maxfreq)
                        {
                                maxfreq=yBuffer[j];
                                maxfreqpos=j;
                        }
                }
        }

        //"PMF of ADC"
        ST7735_SetCursor(0,0);
        ST7735_OutString("PMF of ADC");
```

```
        ST7735_SetCursor(0,159);
        ST7735_PlotClear(32,159);

        if(maxfreqpos<63)
        {
                for(int i =0; i<128; i++)
                {
                        ST7735_PlotBar(yBuffer[i]);
                        ST7735_PlotNext();
                }
                count++;
        }
        else if(maxfreqpos>4033)
        {
                for(int i=4033-128;i<4033;i++)
                {
                        ST7735_PlotBar(yBuffer[i]);
                        ST7735_PlotNext();
                }
        }
        else
        {
                for(int i =maxfreqpos-63; i<maxfreqpos+63; i++)
                {
                        ST7735_PlotBar(yBuffer[i]);
                        ST7735_PlotNext();
                }
        }
        count++;
}

//*********partG function*********************
//This function allows to draw lines to ST7735 LCD screen in any direction and any length the user wants.
void partG(void)
```

```
{
        ST7735_SetCursor(0,0);
        ST7735_OutString("Part G          ");
        ST7735_SetCursor(63,95);
        ST7735_PlotClear(32,159);

        ST7735_Line(0, 32, 127, 159, ST7735_BLUE);
        ST7735_Line(0, 159, 127, 32, ST7735_BLUE);
}

//*************ST7735_Line*****************************************
//  Draws one line on the ST7735 color LCD
//  Inputs: (x1,y1) is the start point
//          (x2,y2) is the end point
// x1,x2 are horizontal positions, columns from the left edge
//              must be less than 128
//              0 is y the left, 126 is near the right
// y1,y2 are vertical positions, rows from the top edge
//              must be less than 160
//              159 is near the wires, 0 is the side opposite the wires
//        color 16-bit color, which can be produced by ST7735_Color565()
// Output: none
void ST7735_Line(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t color){

        if(y1<32)
                y1=32;
        if(y2<32)
                y2=32;
        int16_t num, denom;
        uint16_t currentX, currentY;
        int32_t inc = 0;
        num = y2 - y1;
        denom = x2 - x1;
        currentX = x1;
```

```
currentY = y1;
int32_t store=0;

if ( abs(denom) > abs(num) )
{
	int numSteps = 1;
	if (denom > 0){
		inc = POSITIVE;
	} else {
		inc = NEGATIVE;
		denom=-denom;
	}
	//num=abs(num);
	uint32_t baseY = y1 * denom;
	while (currentX != x2){
		ST7735_DrawPixel(currentX, currentY, color);
		currentX += inc;
		currentY = (baseY + numSteps * num) / (denom);
		++numSteps;
	}
}

else {
	int numSteps = 1;
	if (num > 0){
		inc = POSITIVE;
	} else {
		inc = NEGATIVE;
		num=-num;
	}
	uint32_t baseX = x1 * num;

	while (currentY != y2){
		ST7735_DrawPixel(currentX, currentY, color);
```

```
                    currentY += inc;
                    currentX = (baseX + numSteps * denom) / (num);
                    ++numSteps;

                }
            }
        }
    }
```