Accelerometer Angle Acquisition and Subsequent Display in MATLAB®

By: Caleb M. Gannon

Abstract The essence of this paper is the exploration of three key components in an experimental system. The measurement device, the data processor, and the subsequent display of data. The focus of this system is the acquisition of angle data by means of an accelerometer (*adxl337*). The processor is an *EsduinoXtreme* microcontroller, and the display of data is performed in MATLAB®. This process requires the use of serial communications, integer mathematics, and button interrupts. The accelerometer measures the strength of gravity along an axis, this data is fed to the microcontroller as an analog voltage. The analog-to-digital conversion is sampled at a rate of 4kHz. The microcontroller uses a low-pass (rolling average) filter of four data points. As a result, filtered data is produced at a rate of 1kHz. The data is mapped to a predetermined *arcsin* input space of range 800, by centering it around zero. This data is then run through an *arcsin* approximation, which uses a mix of a third order Taylor series approximation, and a look-up-table. The angle returned by *arcsin* is converted to signed Binary Coded Decimal and displayed on a series of nine LEDs. This angle is serially transmitted to MATLAB®, where it is dynamically plotted using a queue system. Button interrupts are implemented to switch the current axis (*x* / *y*), and to start/stop serial communication. The *z* axis is used to implement 360-degree measurement.

***Index Terms*— Angle Acquisition, Analog-to-Digital, Electrical & Computer Engineering, McMaster University, Serial Communication**

TABLE OF CONTENTS

I. INTRODUCTION

GRAVITY and mathematics are the fundamental tools needed to measure the angled orientation of an accelerometer. In combination with a microcontroller, a programming environment, and physical hardware, a system can be constructed that allows for dynamic visual plotting of the accelerometer orientation. Moreover, the applications of such a system extend far beyond the visualization of sampled data. The data can be used as part of a more intricate system. Be it having a smartphone replace the bubble-gauge in measuring a perfect slope, or in designing the flight system of a modern Boeing 747, sampling an accelerometer to measure orientation is a foundational system in modern electronics. It highlights a collective of many key aspects of computer system design. The sampling of accelerometer data invokes concepts such as bus speeds, frequency filters, and Nyquist rates. Sampling of real-time data requires the use of analog-to-digital conversion of a signal. The processing of this data requires knowledge of microcontroller design, as well as integer mathematics and functional approximation. Cross-system communication is invoked with the use of serial communication, as the collected and processed data can be digitally transmitted with a standard protocol to be used in any other means. The data is dynamically plotted following its serial transmission using another system, MATLAB®. The system involves the user, with active-low momentary switches, and thus invokes the use of interrupts and program flow design. These core concepts apply to a range of technologies in computer design. A cars ABS computer must read the users brake pedal and wheel angular speed against acceleration data, to determine if the car is slipping. The processed data is then transmitted as a result to the braking system, and if the wheels are locked up the braking system will compensate. There are many situations in which the measurement, processing, and transmission of data work out to be the purpose of a systems design, as will be the basis of this report today.

II. BACKGROUND

There are six core concepts that are required to create this angle acquisition system. Data acquisition, sampling, processing, transmission, display, and user input. Firstly, the accelerometer is used as the direct analog tool of measurement. The accelerometer package chosen for this system is the *adxl337*. This package has a three-axis sensor, with a range of ±3.6g (typical). It has a low volume of $13mm^3$, is low power, and operates from 1.8 V to 3.6 V. These conditions make it an excellent low-cost choice for this system.

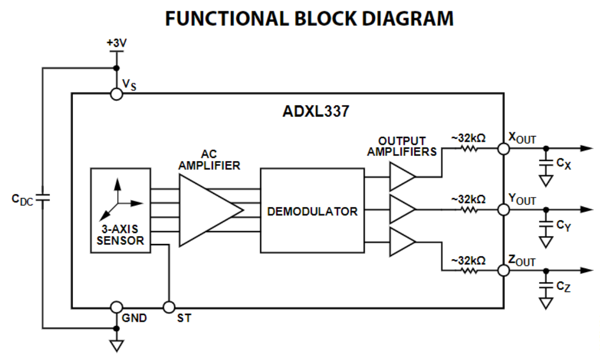

Figure 1.

The sensor data is processed using the *EsduinoXtreme* (esduino) microcontroller system using the 9S12GA240 processor from NXP. This microcontroller has sub-systems such as analog-to-digital converters, timer units, power width modulation, functional interrupts, and serial communications. These systems allow the microcontroller to handle data sampling, processing, transmission, and user input.

Data processing begins with the analog-to-digital conversion (ADC) system. This system represents the continuous accelerometer signal using discrete values. The system does this by mapping the analog signal to a discrete voltage, determined by the bit-size of the conversion, and the range of V_{RH} and V_{RL}, using a successive approximation method.

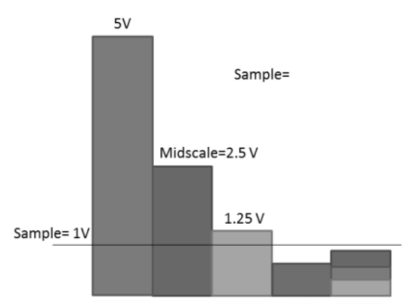

Figure 2.

This plot (Figure 2, p2) demostrates an ADC sequence with an analog sample of 1V, V_{RH} = 5V, V_{RL} = 0V, and 4-bits for conversion. The max error will be (5V-0V)/2^4, or 0.3125V.

The esduino ADC module is using a 12-bit conversion method as based on the second LSD of the student number 400137271, i.e. "7". The V_{RH} has been configured on breaker JB8 of the esduino to be 3.3V. This means there is a maximum error of (3.3V-0V)/2^{12}, or 806µV. This error is minimal compared to the range of outputs from the ADC, which span 800 values. Thus, the accelerometer readings range by 800*806µV, or 645mV. The ADC is configured to sample data at a rate of 4MHz, allowing a theoretical Nyquist frequency of 2MHz.

Data is sampled by the microcontroller at a much lower rate, with the ADC being read at a rate of 4kHz. A low-pass rolling average filter is applied to the data across four data points, to remove unwanted high frequency accelerometer vibrations from the environment or users' hands. This renders the final filtered sampling rate to be 1kHz, with a Nyquist frequency of 500Hz. Further processing is required as the measured acceleration value must be converted into an angle. The relationship between the measured strength of gravity and the angle of an accelerometer is represented by this model:

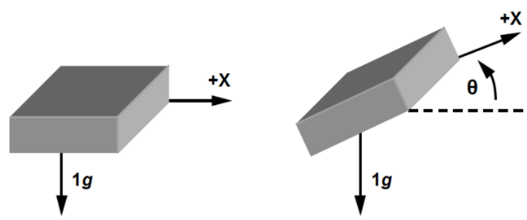

Figure 3.

$$\theta = \frac{arcsin(A_{X,OUT}[g])}{1g} \times \frac{180}{\pi}$$

Equation 1.

This model demonstrates the relationship between gravity and the angle of an accelerometer. *Theta* can be calculated using the mathematical routines of the esduino microcontroller, given the sampled & filtered data point, A_X, and the function *arcsin*. The sampled data was captured on the AN5 pin for the *x* axis, in accordance with project specifications based on the LSD of the student number 400137271, i.e. '1'. The esduino lacks floating-point arithmatic routines, and thus the *arcsin* function is fully implimented using integer mathematics (Section III, G).

The data is displayed directly from the device in signed Binary Coded Decimal via nine LEDs. The respective pin of each LED is mapped using a *setLEDx(n , v)* function, which sets the LED in binary position *n* to state *v*. Where the variable v = 1 (LED on) or v = 0 (LED off). This functional allocation allows for good modulatiry of design, as the rest of the code calls *setLEDx()* rather than accessing the pin registers directly. The *displayBCD(n)* function calls the *setLEDx()* function and uses bit shifting and masking to correcty display *n* in BCD.

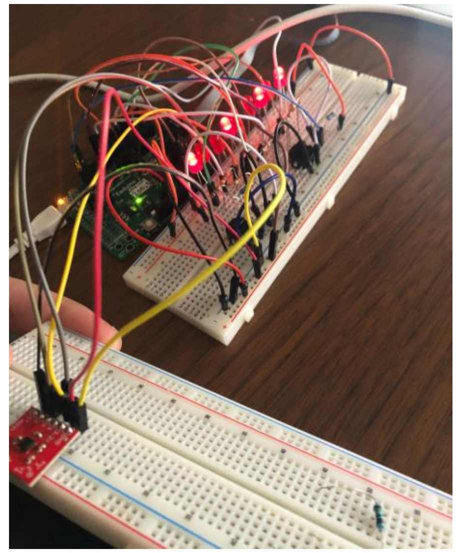

Figure 4.

Bits 0,2,5,8 are illuminated in figure 4. This indicates in BCD a measured angle of -26 degrees.

The measured *theta* values are transmitted serially to MATLAB®, where they are plotted in real time using a dynamic array-queue system. The array-queue system initializes and empty array, first filling it with sampled serial values, then once full the array left-shifts the values and places the next sample at the last index. This results in live plotting.

The range is shown as the x-axis is moved from -180 degrees, to 180 degrees (Figure 6). The jump at ~5.5s demonstrates the reading passing over 180 degrees to -180 degrees, thus a full 360 degrees of measurement. The bumps appear as a result of the hand motions required to fully rotate the device. The full 360 degrees of measurement was achieved through measurement of the *z*-axis. The z-axis was found to read 2100 at the point that both *x* and *y* would reach the ±90 degree bounds. If *z* read less than 2100, the *x*/*y* reading would move past ±90 degree in accordance with the mapping:

$$\theta = \theta + 2 \cdot (90 - \theta)$$

Equation 2.

Where the 180 degree method would reach 90 degrees, then begin to return to zero as the angle increases, the 360 degree method reaches 90 degrees, then 91, 92, etc.

User input was configured to allow for two momentary switches. The first switch would change the readings between the *y* and *x* axis, while the second would start/stop serial transmission. Both switches are implimented using interrupts on the microcontroller timer module. The *y* / *x* swap was achieved by the button tied active-low to PT4. Upon receiving a falling edge, the interrupt would trigger, and switch the *ATDCTL5* register, changing the active analog port between AN5 (*x*) and AN4 (*y*). The second momentary switch starts and stops serial transmission by means of a *serial_en* conditional. Upon falling edge from the second momentary switch, located on PT0, *serial_en* switches between 0 and 1. If it reads 1, serial communications is enabled via if-statement in the main loop. If it reads 0, the if statement fails, and no transmission occurs.

III. Design Methodology

The design of this project began with a system-wide structural design and is followed by the sub-system design methods. This includes the pin assignment, signal property analysis, transducer design, Precondition /Amplification / Buffer design, ADC register configuration, LED BCD configuration, data processing, serial communication, and the system-wide block diagram and circuit schematic.

A. Final Pin Assignment Map

Esduino Pin	***Port***	***Use***
A5	AN5	x-axis
A4	AN4	y-axis
A3	AN3	z-axis
D5	PP1	LED0
PT3	PT3	LED1
D12	AN11	LED2
D10	AN10	LED3
D9	AN9	LED4
D8	AN8	LED5
D7	PT2	LED6
D6	AN7	LED7
D4	AN6	LED8
PT4	PT4	x/y switch
D2	PT0	Start/stop serial
D13	PJ0	Mode Indicator
GND	GND	GND
3.3V	3.3V	3.3V supply
5V	5V	Switch Active High

B. Quantify Signal Properties

Resolution of ADC is 806µV.

x-axis:
V_{Lx} = 1675*806µV = 1.35V
V_{Hx} = 2475*806µV = 1.99V

y-axis:
V_{Ly} = 1620*806µV = 1.31V
V_{Hy} = 2420*806µV = 1.95V

C. Transducer

A transducer is a device capable of converting physical phenomena into electrical signals. In this case, the *adxl337* accelerometer is used. This accelerometer outputs the acceleration reading along its *x*, *y*, and *z* axis as a voltage.

D. Precondition/Amplification/Buffer

The *adxl337* handles preconditioning, amplification, and buffering. The output of its internal 3-axis sensor is pre-amplified, demodulated using phase-sensitive demodulation techniques that determine the magnitude and direction of the acceleration, then the resultant signal is amplified again. This signal is transmitted out across 32Ω current limiting resistors.

E. Analog to Digital Conversion

The conversion was performed in 12-bit resolution, right-justified, with one sample per sequence. The prescaler was left to its default of 2, and thus the sampling frequency was 4MHz, half of the 8MHz bus clock.

V_{RH} is set directly to 3.3V on the breaker pin in JB8. V_{RL} is set to 0V. This leads to a resolution of 806µV. This allows a range of 800 possible data values to be read between the analog input max/min.

F. LED display in MODE 0 & MODE 1

The LED display was configured using a modular, multifunctional design. A function *setLEDx(n , v)* was implemented using bit-masking, that would selectively set the LED representing bit *n* of the BCD display to the state of *v*, where v = 1 or 0.

This function was called by a second function, *displayBCD(n)*, which used a combination of bit-shifting and masking to enable the correct LEDs based on the value of *n*. By passing *theta* into this function, the current angle of the accelerometer is displayed.

Mode switching was accomplished using a momentary switch in an active-low configuration. With an interrupt enabled on PT4 using the TIE, TSCR, and TCTL registers, a marker variable *dir_en* would be swapped between 1 and 0 to indicate the current mode. The PTJ LED on D13 glows to indicate *y* is enabled, and is turned off to indicate *x*.

G. Data Processing (Algorithm)

The use of integer mathematics requires ceratin approximations to be made in the calculation of *arcsin*. Due to the nature of integer arithmatic and sampling, a psudo-floor function is applied to each input value. Thus, within the confines of analog-to-digital conversion, if the arcsin approximation is within one degree of error, it is achiving the maximum possible performance. With this condition in mind, an adequate *arcsin* approximation was designed. A third order Taylor series approximation is used initialy, so long as the resultant output is within one degree of the true *arcsin* function. This range was determined using the graphical calculating software *desmos*™. Due to the symmetrical nature of the *arcsin* function, and the linear nature of the input range, a symmetrical approximation was designed considering the *absolute value*, using only 400 units. The conversion multiplier of 180 degrees / pi was approximated by the integer ratio 573/10, and 57:

$$g(x) = \text{floor}\left(\frac{573x}{4000}\right) + \text{floor}\left(\frac{57x^3}{6 \cdot 400^3}\right)$$

Equation 3.

This equation (Equation 2.) is derived from the third order taylor series approximation of arcsin:

$$\arcsin(x) = x + \frac{x^3}{6}$$

Equation 4.

The taylor series approximation is scaled for 400 units of input, and premultiplied by 573/10 in the linear term, and by 57 in the cubic term, to convert radians to degrees. The cubic approximation is used due to the maximum range of the long integer format, being 2^{32} -1 or 4,294,967,295. The max possible integer required for this approximation is $57*400^3$, or 3,648,000,000, which is within the long integer range. The input was left-shifted by +4 values, to simulate a round method rather than floor:

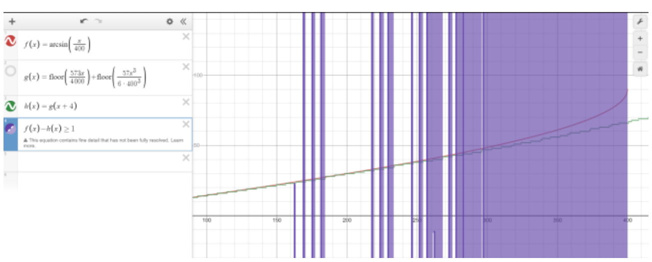

Figure 5.

The purple shaded area above in (Figure 5) respresents the range of inputs for which the taylor approximation is incorrect by greater than one degree, and thus fails. The area intensifies at around the 250 unit mark, and considering look-up-table memory constraints this was selected as the breaking point. Following the 250 unit mark, and precalculated array of look-up-table values are to be used to maximise accuracy.

The look-up-table values were calcualted using a simple MATLAB® program which would map *arcsin* on a range from 250-400 to an array of 151 values. These values are perfectly mapped, and remove the need for calculations on the microcontroller. The value sampled from the accelerometer directly indexes the correct theta value from the look-up-table. Ideally, this method would have been used to store all of the *arcsin* values and eliminate the need for approximation, however, the memory on the device was the limiting factor, as the maximum allowed size of look-up-table was found to be circa. 160 elements. However, as the look-up-table is used once the Taylor series fails by one degree, the continous approximation is as accurate as possible given floored integer values:

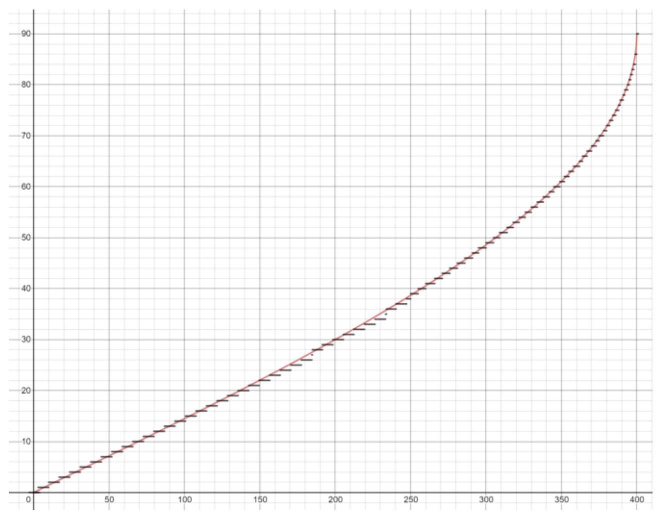

Figure 6.

The black lines indicate the system's *arcsin* integer approximation, with the red line being a continuous arcsin function. Due to the symmetrical nature of *arcsin*, this method was used to map the positive and negative acceleration readings of both the *y* and *x* axis to a respective angle; as the sampled data was mapped to a 800 data point range centered around 0, with an external variable to keep track of the sign, and reapply it via multiplication before display/transmission.

H. Control/Communicate

The serial transmission was enabled/disabled via a momentary switch on port PT0. This port took advantage of the timer module interrupts, and upon detecting a falling edge, it would trigger a *serial_en* control variable to flip between 0 and 1. A respective conditional statement would allow/disable the serial transmission and display of data.

The serial transmission was configured at a baud rate of 38400 bits/s, which was the maximum possible given this microcontroller, and the bus clock of 8MHz. The 8MHz bus clock meant that the error in baudrate is 0.16%, as the actual baudrate is:

$$\frac{\left(8 \cdot 10^6\right)}{16 \cdot 13} = 38\,462$$

Equation 5.

The *13* above (Equation 5.) is the current SCI0BDL register for a baud rate of 38400bits/s.

The error of 0.16% is acceptable in maintaining good serial communication, and thus given the bus speed provided in the project specification of 8MHz, 38400bits/s was chosen.

This data was serially received and processed in MATLAB®, using its serial library. A serial port is opened with a matching baud rate of 38400bits/s, on the devices active serial port (in the author's case, this was '*COM3*'). The serial port is read using the *readline()* command, which grabs the current transmission and stores it in a variable.

Using an array-queue teqnique, the current transmission is stored in an array, with new values being placed at the front of the array, and old values being left-shifted. Using the *refreshdata* and *drawnow* commands, the array is dynamically plotted (Figure 9, p6).

I. Full System Block Diagram

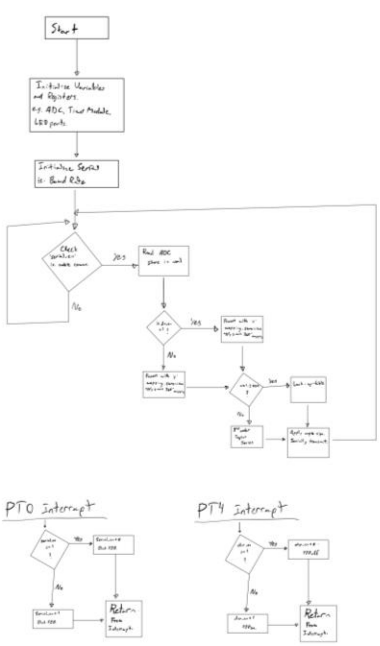

Figure 7.

J. Full System Circuit Schematic

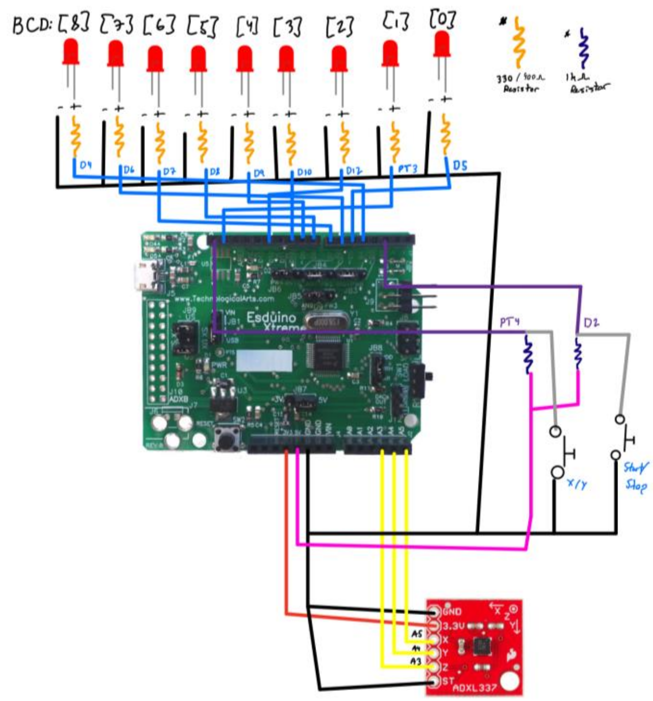

Figure 8.

IV. Data and Results

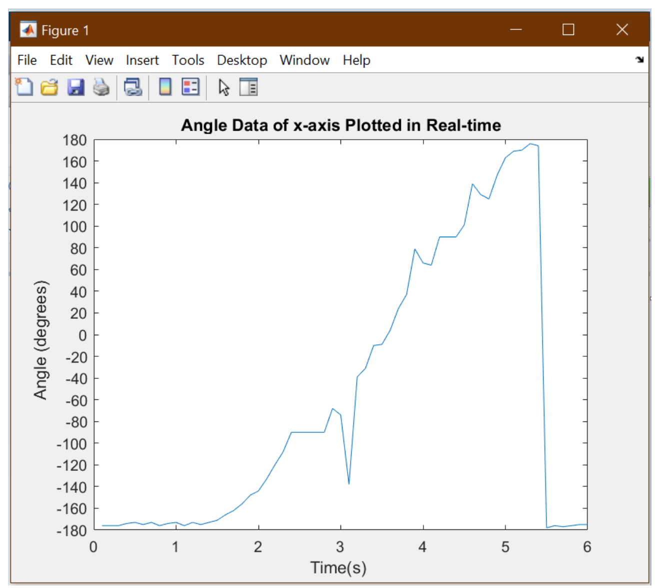

Figure 9.

The data was collected through serial transmission in MATLAB® and promptly displayed using the array-queue technique. This is a test of the serial transmission, data sampling, filtering, and processing systems. The resultant waveform in Figure 9. demonstrates a success.

The signal smapling was tested in Realterm™, as the value measured by the ADC was transmitted. Realterm™ was also used to test the correct processing of *theta*. These two values can be seen in this transmission:

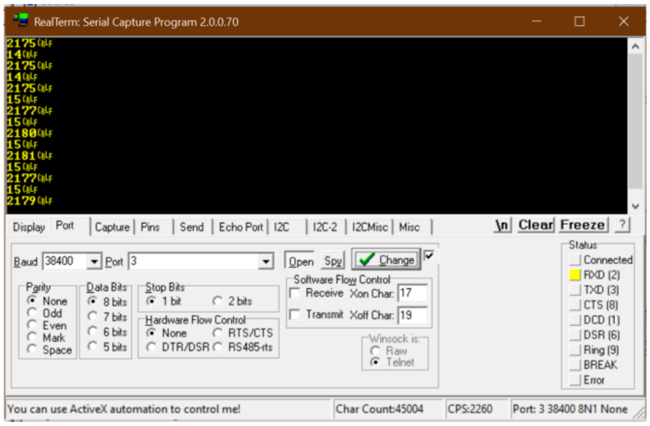

Figure 10.

The start/stop serial transmision button was tested in Realterm™. When initially pressed, the transmission would appear (Figure 10). When pressed again, the transmission would stop and Realterm™ would freeze in place. Realterm™ would resume upon a third press.

The LED system was tested using a counter from 1-99 that would display in binary, and once confirmed operational *theta* in BCD was displayed.

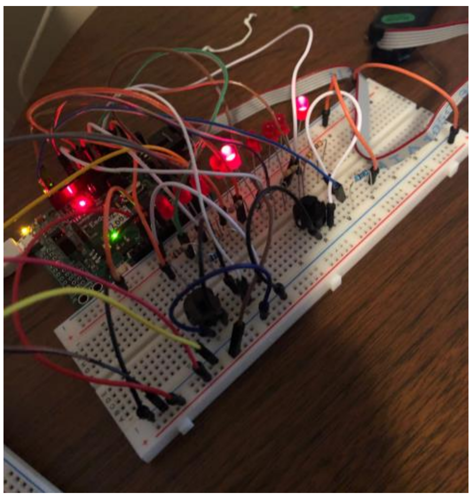

Figure 11.

The button interrupts were tested seperately. The *x*/*y* switch was tested using the LEDs. Upon button press the PJ0 LED would glow, and the BCD LEDs would display the *y*-axis readings in BCD (Figure 11). Upon another press, PJ0 would turn off, and the BCD LEDs would display the *x*-axis (Figure 4, p3).

V. Discussion

1) Summarize how you were able to overcome the ESDX not having floating point capability and not having trigonometric functions.

The lack of floating-point capabilities on the esduino board means integer arithmetic had to be used. Pre-multiplying and post-division were two techniques that would replace the equation 10*0.5 with 100*5/10.

These techniques meant that careful consideration of the range of possible integer values was necessary, to ensure that at no point in an operation there would be overflow.

To ensure this, the Taylor series approximation was limited to a third order cubic form, with the remainder of the *theta* values being accessed via a look-up-table.

2) Calculate your maximum quantization error.

The following formula was used to calculate the maximum quantization error.

$$Q = \frac{max(x) - min(x)}{2^{N+1}}$$

Equation 6.

N is 12-bit, V_{RH} =3.3V, and V_{RL} =0V. Thus, the max quantization error is the same as the resolution, 806µV.

3) Based upon your assigned bus speed, what is the maximum standard serial communication rate you can implement. How did you verify?

A few key factors were considered in determining the baud rate. The first was percent error in esduino representation. The SCI0BDL regeister controlls the clock divisor that sets the baud rate (Equation 5, p6). 115200bits/s was found to have 8% error with an 8MHz bus clock, and 57600bits/s was found to have 4%. These are unaceptable levels of error. 38400bits/s had only 0.16% error, and was left as the highest viable commercially standard baud rate.

This baud rate was verified in both MATLAB® and Realterm™, upon the succesfull transmission of theta values.

4) Reviewing the entire system, which element is the primary limitation on speed? How did you test this?

The limiting factor regarding speed was determined to be the *adxl337*. Its bandwidth is capped at 1600Hz as per the specification. An Analog Discovery 2 was used to measure the response factor to movement of the accelerometer, and it was found to be consistently below 1200Hz.

This is compensated for by the rolling average low-pass filter, which produces data at 1000Hz.

5) Based upon the Nyquist Rate, what is the maximum frequency of the analog signal you can effectively reproduce? What happens when your input signal exceeds this frequency?

The Nyquist frequency is half that of the sampling rate. As the ADC frequency is 4MHz, the maximum analog frequency that can be accurately sampled is 2MHz.

In the case that the analog signal is greater than 2MHz, aliasing will occur, and all fourier components of the periodic signal will not be recovered.

6) In general, are analog input signals with sharp transitions accurately reproduced digitally? Justify your answer.

In general, analog signals with sharp transitions can not be accurately represented digitally. This can be explained simply when considering the fourier series representation of such waveforms. The sharp transitions correspond to infinitely high frequencies, and thus no sampling rate will be able to correctly capture them.

Additionally, due to the Gibbs Phenomenon some sharply transitioned waves such as square waves are impossible to be accurately represented due to a ringing found at the jump discontinuity.

VI. Conclusion

The device proves to be functional in measuring the angle of an accelerometer. More so, as a complex combination of subsystems, it demonstrates modularity and data processing with respect to all computer systems. The system involves sampling techniques, filtering, processing, data mapping. It makes use of complex integer mathematics, serial transmission, and user input techniques.

The many tools, timers and registers configured in the implementation of such a system are representative of the role of computer systems in modern technology: collecting, processing, and utilizing data. The accelerometer is not the only transducer, and the techniques implemented in this system will extend to system design regarding a multitude of other data driven designs. From airplanes, to smartphones, to cars, transducers and their implementations are limitless.

This system was a success. In combination with MATLAB®, Realterm™, and Freescale CodeWarrior™, the *EsduinoXtreme*™ is able to accurately process and display angle readings from an *adxl337* accelerometer.

VII. Appendix

Alongside main.c and the matlab code, an esduino serial library was provided, and is included in the project package.

Codewarrior Esduino Main.c

```c
// filename ******** Main.C **************

//*********************************************************************
// Simple ADC example for the Technological Arts EsduinoXtreme board
// by Carl Barnes, 12/03/2014
//*********************************************************************

#include <hidef.h>      /* common defines and macros */
#include "derivative.h"  /* derivative information */
#include "SCI.h"
#include "stdlib.h"
#include "stdio.h"
//#include "math.h"
//#include "LUT.h"

char string[4];
unsigned short val;
short theta;
long g;
//short LUT[1001];
int LUT_i;
int index_m;
unsigned short val1;
unsigned short val2;
unsigned short val3;
unsigned short val4;
short sign_theta=0;
unsigned long x;

int max=0;
int min=0;
short z_dir=0;

short x_sign=0;

int loop_cnt=0;

short press_cnt=0;
short dir_en=0;
short serial_en=1;

short temp_atdctl;


void QuartermsDelay(unsigned long time);
short getLUTdata(int i);
char atan(int n, long* x_p);
short getSign(int n);
void busCLK(void);
void setLEDx(int n, int v);
void displayBinary(short s,unsigned short n);
void displayBCD(short s,unsigned short n);


//---------------------OutCRLF---------------------
// Output a CR,LF to SCI to move cursor to a new line
// Input: none
// Output: none
// Toggle LED each time through the loop

void OutCRLF(void){
  SCI_OutChar(CR);
  SCI_OutChar(LF);
  PTJ ^= 0x20;          // toggle LED D2
}

void main(void) {
const char LUT[]={39,39,39,39,39,40,40,40,40,40,41,41,41,41,41,41,42,42,42,42,42,43,43,43,43,43,44,44,44,44,4


  int g_index=0;

   press_cnt=0;
   dir_en=0;
   serial_en=0;
   loop_cnt=0;
   z_dir=0;
busCLK();

DDRP=0xFF;
DDRT=0xFC;
ATDDIEN=0xFFC0;
DDR1AD=0xC0;
DDR0AD=0xFF;

// Setup and enable ADC channel 0
// Refer to Chapter 14 in S12G Reference Manual for ADC subsystem details

    ATDCTL1 = 0x4F;     // set for 12-bit resolution
    ATDCTL3 = 0x88;     // right justified, one sample per sequence
    ATDCTL4 = 0x00;     // prescaler = 2; ATD clock = 8MHz / (2 * (0 + 1)) == 4MHz
    ATDCTL5 = 0x25;     // continuous conversion on channel 0


// Setup LED and SCI
  DDRJ |= 0x01;     // PortJ bit 0 is output to LED D2 on DIG13


  //LUT=getLUT();


  //enable timer interrupts

  TSCR1 = 0x90;    //Timer System Control Register 1
                    // TSCR1[7] = TEN:  Timer Enable (0-disable, 1-enable)
                    // TSCR1[6] = TSWAI:  Timer runs during WAI (0-enable, 1-disable)
                    // TSCR1[5] = TSFRZ:  Timer runs during WAI (0-enable, 1-disable)
                    // TSCR1[4] = TFFCA:  Timer Fast Flag Clear All (0-normal 1-read/write clears interrupt flags)
                    // TSCR1[3] = PRT:  Precision Timer (0-legacy, 1-precision)
                    // TSCR1[2:0] not used

  TSCR2 = 0x00;    //Timer System Control Register 2
                    // TSCR2[7] = TOI: Timer Overflow Interrupt Enable (0-inhibited, 1-hardware irq when TOF=1)
                    // TSCR2[6:3] not used
                    // TSCR2[2:0] = Timer Prescaler Select: See Table22-12 of MC9S12G Family Reference Manual r1.25 (set for bus/1)


  TIOS = 0xEE;     //Timer Input Capture or Output capture
                    //set TIC[0] and input (similar to DDR)
  PERT = 0x11;     //Enable Pull-Up resistor on TIC[0]

  TCTL3 = 0x02;    //TCTL3 & TCTL4 configure which edge(s) to capture
  TCTL4 = 0x02;    //Configured for falling edge on TIC[0]

/*
 * The next one assignment statement configures the Timer Interrupt Enable
 */

  TIE = 0x11;      //Timer Interrupt Enable
  EnableInterrupts;
  PTP=0x02;
  for(;;) {
    //PTJ ^= 0x01;          // toggle LED

    if(serial_en==1){
    if(loop_cnt==0){
     SCI_Init(38400);
     loop_cnt=1;
    }
    temp_atdctl=ATDCTL5;

    val1=ATDDR0;

    QuartermsDelay(1);
    val2=ATDDR0;

    QuartermsDelay(1);
    val3=ATDDR0;

    QuartermsDelay(1);
    val4=ATDDR0;
    ATDCTL5=0x23;
    QuartermsDelay(1);

    val=(val1+val2+val3+val4)/4;

    z_dir=ATDDR0;
    if(z_dir>2100){

      z_dir=0;
    }else{
      z_dir=1;
    }

```

```c
        ATDCTL5=temp_atdctl;

        if(dir_en==0){


          g_index=abs(val-(1475+400));
          x_sign=getSign(val-(1475+400));
          if(g_index<260){
          theta=atan(g_index,&x);
          }else if(g_index<400){
           theta=LUT[g_index-260];
          }else{
           theta=LUT[150];
          }

          }else{
           g_index=abs(val-(1620+400));
           x_sign=getSign(val-(1620+400));
           if(g_index<260){
           theta=atan(g_index,&x);
           }else if(g_index<400){
            theta=LUT[g_index-260];
           }else{
            theta=LUT[150];
           }

          }

        if(z_dir==1){
         theta=theta+2*(90-theta);

        }


        //PT1AD=0xC0;
        //PT0AD=0xFF;
        //PTT=0xFF;
        ///press_cnt=0;
        //while(1){
         ///press_cnt=press_cnt+1;
         //if(press_cnt>255)press_cnt=0;
         //displayBinary(0,press_cnt);
        // QuartermsDelay(750);

      // }



        displayBCD(x_sign,theta);
        sign_theta=theta*x_sign;
        sprintf(&string, "%d",sign_theta);
        SCI_OutString(string); OutCRLF();
        SCI_OutUDec(val);OutCRLF();
        }

      }
           // 1 sec delay, you can validate using method outlined in Lecture W8-2_W8-3

}

interrupt  VectorNumber_Vtimch4 void ISR_Vtimch4(void){
unsigned short temp;
PTJ^=0x01;

  dir_en =(dir_en+1)%2;
  if (dir_en==0)
  ATDCTL5 = 0x25;
  if (dir_en==1)
  ATDCTL5 = 0x24;
  QuartermsDelay(2000);
  temp_atdctl=ATDCTL5;

  temp = TC4;       //Refer back to TFFCA, we enabled FastFlagClear, thus by reading the Timer Capture input we automatically clear the flag, allowing another TIC interrupt
}
  interrupt  VectorNumber_Vtimch0 void ISR_Vtimch0(void){
unsigned short temp1;
unsigned short temp2=PTJ;
 serial_en=1-serial_en;

 if(serial_en==1){
  PTJ=0x00;
  QuartermsDelay(1000);
  PTJ=0x01;
  QuartermsDelay(1000);
  PTJ=0x00;
  QuartermsDelay(1000);
  PTJ=0x01;
  QuartermsDelay(1000);
  PTJ=0x00;
 }

 if(serial_en<0 || serial_en>1)serial_en=0;

 PTJ=temp2;
 QuartermsDelay(1000);

 temp1 = TC0;       //Refer back to TFFCA, we enabled FastFlagClear, thus by reading the Timer Capture input we automatically clear the flag, allowing another TIC interrupt
}

void QuartermsDelay(unsigned long time)
{
    long i;
    TSCR1=0x90;
    TSCR2=0x00;
    TIOS=0x02;
    TC0=TCNT+2000;
    for(i=0;i<time;i++){
     while(!TFLG1_C1F);
     TC1 +=2000;

    }
}

char atan(int n,unsigned long* x_p){
 unsigned long cubic=0;
 unsigned long linear=0;

 unsigned long z=(unsigned long)(n+4);
 if(n<=4){
  return 0;
 }


 *x_p=(z*573)/4000;
 linear=*x_p;
 *x_p= (( 57*z*z*z)/(384000000));
 cubic=*x_p;

 *x_p=(linear +cubic);





 //*x_p=( ((v*20*v*20)*(v*(*x_p)/13)) + (( (v*(*x_p))*(v*(*x_p))*(v*(*x_p)) )/(6*13*13*13)) )/(v*20*v*20*v*20);
 //*x_p=1800000*(*x_p)/31415;

 return (char)*x_p;
}

int abs(int n){
 if(n>=0){
  return n;
 } else{
  return -n;
 }
}

short getSign(int n){
  if(n>=0){
  return 1;
 }else{
  return -1;
 }
}

void busCLK(void){

 CPMUCLKS=0x80;
 CPMUOSC=0x00;
 CPMUPROT=0x26;
 CPMUSYNR=0x1F;
 CPMUFLG=0x00;

}

void setLEDx(int n, int v){
 if(n==0){
  if(v==1){
   PT1AD=PT1AD|0b01000000;
  }
  if(v==0){
   PT1AD=PT1AD&0b10111111;
  }
 }
 if(n==1){
  if(v==1){
   PT1AD=PT1AD|0b10000000;
  }
  if(v==0){
   PT1AD=PT1AD&0b01111111;
  }
 }
 if(n==2){
  if(v==1){
   PTT=PTT|0b00000100;
  }
  if(v==0){
   PTT=PTT&0b11111011;
  }
 }
 if(n==3){
  if(v==1){
   PT0AD=PT0AD|0b00000001;
  }
  if(v==0){
   PT0AD=PT0AD&0b11111110;
  }
 }
 if(n==4){
  if(v==1){
   PT0AD=PT0AD|0b00000010;
  }
  if(v==0){
   PT0AD=PT0AD&0b11111101;
  }
 }
 if(n==5){
  if(v==1){
   PT0AD=PT0AD|0b00000100;
  }
  if(v==0){
   PT0AD=PT0AD&0b11111011;
  }
 }
 if(n==6){
  if(v==1){
   PT0AD=PT0AD|0b00001000;
  }
  if(v==0){
   PT0AD=PT0AD&0b11110111;
  }
 }
 if(n==7){
  if(v==1){
   PTT=PTT|0b00001000;
  }
  if(v==0){
   PTT=PTT&0b11110111;
  }
 }
 if(n==8){
  if(v==1){
   PTP=PTP|0b00000010;
  }
  if(v==0){
   PTP=PTP&0b11111101;
  }
 }

}

void displayBinary(short x,unsigned short n){
  setLEDx(7,n&0b00000001);
  setLEDx(6,n>>1&0b0000001);
  setLEDx(5,n>>2&0b000001);
  setLEDx(4,n>>3&0b00001);
  setLEDx(3,n>>4&0b0001);
  setLEDx(2,n>>5&0b001);
  setLEDx(1,n>>6&0b01);
  setLEDx(0,n>>7&0b1);

  if(x==-1){
    setLEDx(8,1);
  }else{
  setLEDx(8,0);
  }

}
void displayBCD(short x,unsigned short n){
  short dig1=n%10;
  short dig2=n/10;
  setLEDx(7,dig1&0b00000001);
  setLEDx(6,dig1>>1&0b0000001);
  setLEDx(5,dig1>>2&0b000001);
  setLEDx(4,dig1>>3&0b00001);
  setLEDx(3,dig2&0b00000001);
  setLEDx(2,dig2>>1&0b0000001);
  setLEDx(1,dig2>>2&0b000001);
  setLEDx(0,dig2>>3&0b00001);

  if(x==-1){
    setLEDx(8,1);
  }else{
  setLEDx(8,0);
  }

}
```

Matlab Serial Processing

```matlab
serialportlist()
s=serialport("COM3",38400)
angle_arr=zeros(60,1);
i=0
hf=figure
h = plot(angle_arr);
axis([0 60 -180 180])
yticks([-180:20:180])
title('Angle Data of x-axis Plotted in Real-time');
xlabel('Time(s)')
ylabel('Angle (degrees)')
xticks([0 10 20 30 40 50 60])
xticklabels({'0','1','2','3','4','5','6'})
p=0;
while true
    val=readline(s)
    p=p+1;
    if mod(p,10)==0
        p=0;

        if i<60
        i=i+1;

        angle_arr(i)=val;
        else
            for j=1:59
                angle_arr(j)=angle_arr(j+1);
            end
            angle_arr(60)=val;
        end
        set(h,'YData', angle_arr);
        refreshdata(hf,'caller')
        drawnow
    end


end
```