Tissue Impedance and Biomechanical Measurement Device Modification

Mingyuan Huang
Advisor: Dr. Bruce Land, Dr. Jonathan Butcher

Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

Project Title: Tissue Impedance and Biomechanical Measurement Device Modification
Author: Mingyuan Huang

Abstract:

This Meng project was focused on the modification of a tissue impendence measurement device for Dr. Jonathan Butcher in the Biomedical Engineering department. The goal for this project was to modify the electrical components of system in order to improve its performance such as data measuring speed, device stability and portability. First goal of our project was to build a miniaturized fast frequency function generator using PIC32 Microcontroller and integrated it with the current system to increase data collection speed as well as miniaturize the device size. Second goal was to replace the current DAQ component with a PIC32 MCU based design to further increase the data collection speed.

Executive Summary

Through two semesters' working on this project, I succeeded in building the new PIC32 microcontroller based function generator, and integrated it with the current system. The new function generator was able to generate desired sine wave signals as the original one, and at the same time improved the overall system performance.

Firstly, the new function generator increased the communication speed with the laptop, hence reduced the test duration drastically. This achievement was critical to the current system, since the ultimate goal for this impedance and mechanical measurement device was to measure impedance and mechanical property of soft tissue on woman who may have risk of preterm birth. Therefore, it is important to minimize the data collection time to lower patient discomfort.

Secondly, the new function generator was less than half of the volume of the original function generator. This would greatly improve the whole device's portability. Portability is also a key design factor for the whole device, since doctors may need to use the device to out of the clinics. Therefore, it is important that the whole device can be easily carried on.

Thirdly, I also wrote the control algorithm on the laptop, so that the function generator can change its frequency automatically once it received the commands from the laptop. The original function generator can only set a single frequency during the test, which is inefficient in terms of collecting tissue impedance.

In conclusion, the new function generator I built did play an important role in terms of improving the system performance as well as narrow the gap between lab testing and real human testing. I believe this system would be able to analyze real human tissue properties very soon.

Besides the function generator, I am also working on modifying the data acquisition component (DAQ) of the system with another graduate student Zhenchen. We decided to use another PIC32 microcontroller to replace the current DAQ. We believed this would reduce the test duration further as well as improve its portability. Currently, we were able to record a pair of voltage and current signals for a single frequency and send it back to the microcontroller. Next step would be synchronizing this DAQ with the function generator, so that the DAQ can record pairs of voltage and current for multi frequency test.

Table of Contents

Introduction

This PIC32 microcontroller based function generator is designed to replace the original function generator (Velleman PCGU1000) used in the soft tissue impedance and mechanical measurement device in Dr. Butcher's lab. The original function generator has the following limitations, which affect the performance the whole system.

Limitation 1: Slow response to the commands sent from the laptop. The original function generator took around 1 second to change a single frequency. Therefore, for a multi-frequency test, the current function generator will slow down the data collection process.

Limitation 2: Bulky size. The size of the original function generator is (5.9 x 8.07 x 1.3 inches), which does not satisfy the size requirement for the whole system.

In order to overcome these two limitations, with the help of Dr. Bruce Land, I built this pic32 microcontroller based function generator (figure 1). This new function generator responds to laptop commands in microseconds. Additionally, its volume is much smaller than the original one. Therefore, this new function generator does improve the performance of the entire system.

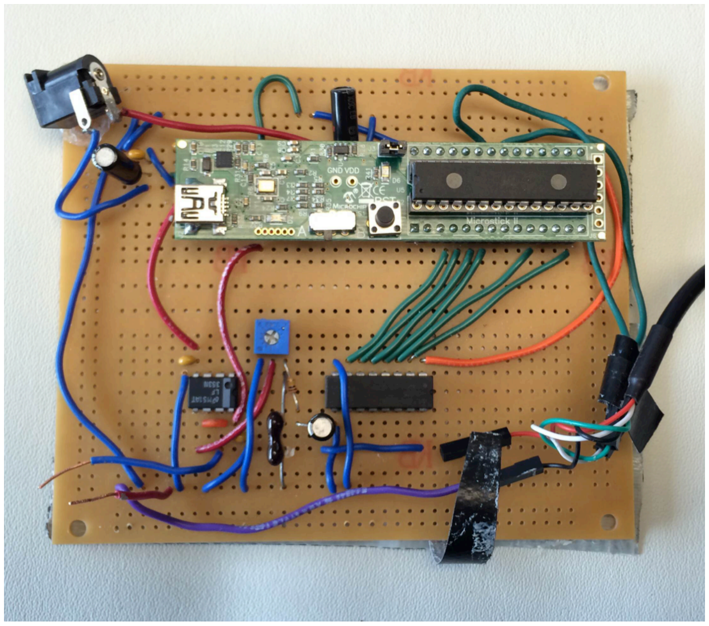
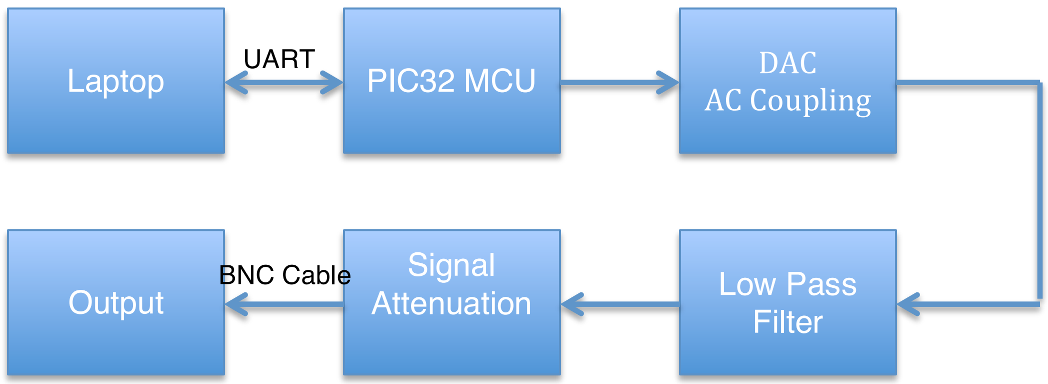


Figure 1: Circuit board layout

High Level Design

- **System Structure**



The structure of the function generator contains six major components. First component is the laptop used to send commands and read commands to/from the PIC32 microcontroller. Second component is a PIC32 microcontroller, which generates a sine function based on different frequency request, and bursts the signal out from B.0 to B.5 and B.7 using DMA (direct memory access). The third component, a R2R resistor ladder is used to convert the digital signals coming out of the MCU pins to an analog signal, in this case, a sine wave. Because of the specification of the sine wave, a signal attenuation circuit is applied to reduce the amplitude of the sine wave to an optimal level. After that, a low pass filter is applied to reduce the high frequency noises generated by the MCU. I used a 9V split circuit (designed by Dr.Land) to split a 9V DC supply to power the op-amp used in the signal attenuation circuit. Then the output of the sine wave will be fed into the tissue.

Hardware

- **Overall circuit (Figure 2)**

The overall circuit of the function generator as shown in figure 2 contains five major components: a microcontroller, a low pass filter, a signal attenuator, a 9V split circuit and a serial to USB connector. Pin 22 and Pin 21 of the microcontroller connect to the laptop through the serial to USB connector. Universal Asynchronized Receiver and Transmitter (UART) is implemented to read the commands sending from the laptop to change the frequency. The microcontroller calculates the sine table based on the required frequency, and then outputs the digital signals from pin 4, 5, 6, 7, 11 and 14(RB0-5 and RB7). A resistor network (4116R-R2R-253) then takes the digital outputs from the microcontroller and converts them to an analog signal at pin 16. Then the output will be AC coupled, attenuated and filtered to the optimal level.

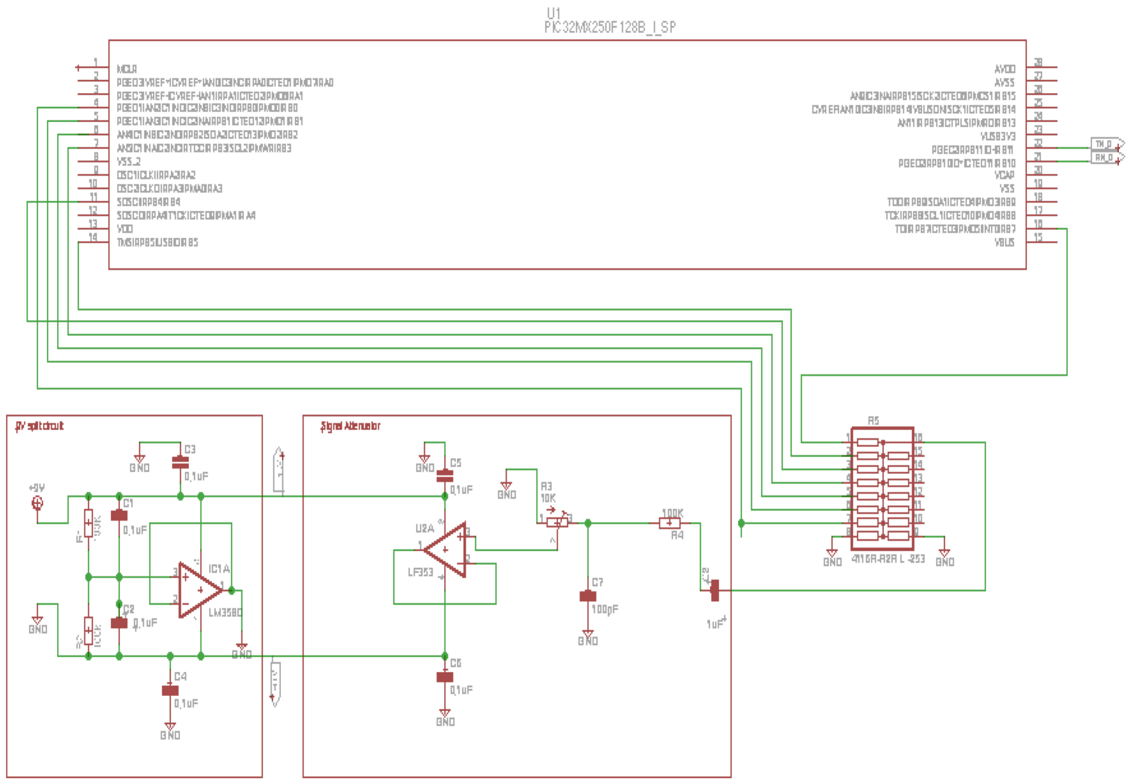


Figure 2: Overall circuitry

- **Signal attenuation and low pass filter (Figure 3)**

A low pass filter is implemented to reduce the high frequency noise (higher than 100K Hz). Based on the calculation shown below, we picked 100K and 100 pF, so that the signal attenuates about -13dB at 80K Hz. Then we used a 10K trimpot and the 100K resistor together as a voltage divider to lower the signal amplitude to desired level. After that, a LF353 op-amp is applied as a buffer before the signal being fed to the tissue. We also added 0.1 uF decoupling capacitors between ground and 4.5V and -4.5V to reduce the change in power supply voltage.

Calculation of cutoff frequency:

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2*3.14*10^5*10^{-10}} \approx 15.9K\ Hz\ (\,signal\ attenuates - 3dB)$$

$$When\ f\ = 80K\ Hz, the\ signal\ attenutes{:}\ \frac{10}{\frac{80}{15.9}} = \frac{-20-3}{x} \Rightarrow x =\ -11.6\ dB$$

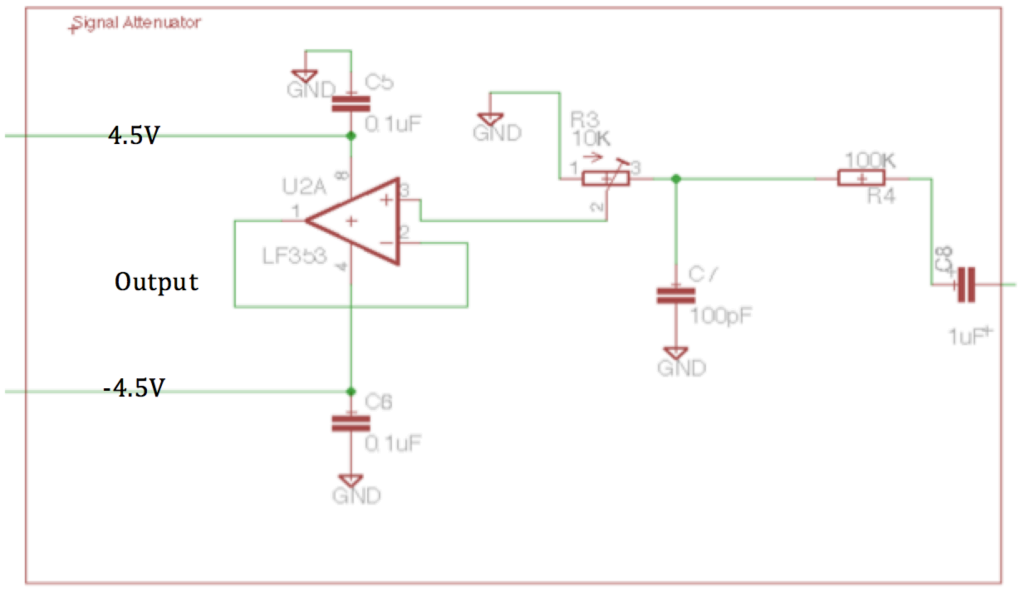


Figure 3: Signal attenuation and low pass filter

- **9 V split circuit (Figure 4)**

  The circuit shown in figure 3 is used to split a 9V DC source to one 4.5V, one -4.5 and a virtual ground to power the op-amp in the signal attenuator circuits.

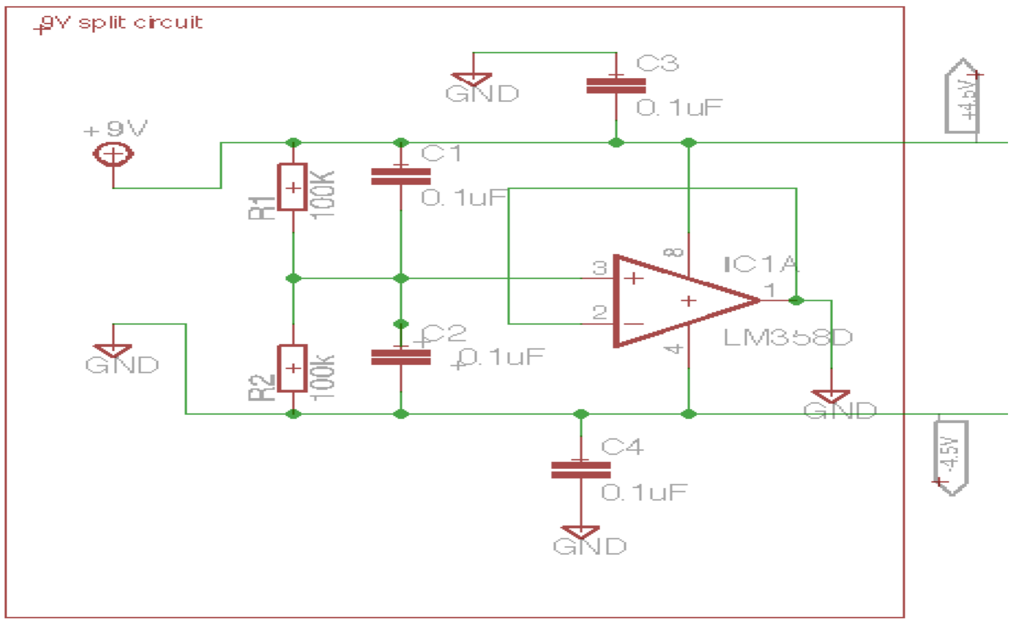


Figure 4: 9V split circuit

- **USB to serial Connector (figure 5)**

We used a USB to TTL serial cable (figure 5) for communication between the laptop. The green wire connects to the RX pin of the microcontroller, and the white wire connects to the TX pin of the microcontroller. The black wire is the ground. Using this cable allows us to easily implement UART between the microcontroller and the laptop, so that the laptop can send command to the microcontroller at anytime at baud rate 57600. However the major problem with this connector is the soft wires at the microcontroller side. Although wires are soldered to the circuit board, the soldered area may lose connection when users accidentally pull wires very hard. Therefore, a better alternative would be the USB to Serial connector with a micro USB adapter on it (Figure 6). This module can be purchased from sparkfun.com.

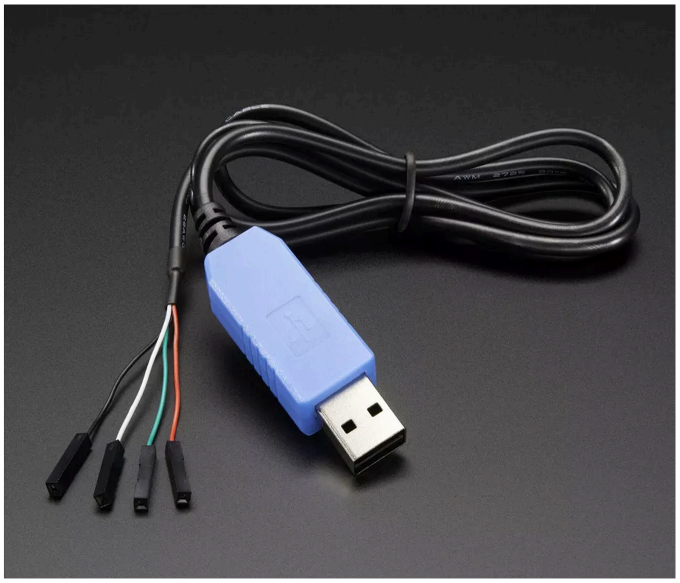


Figure 5: USB to serial connector

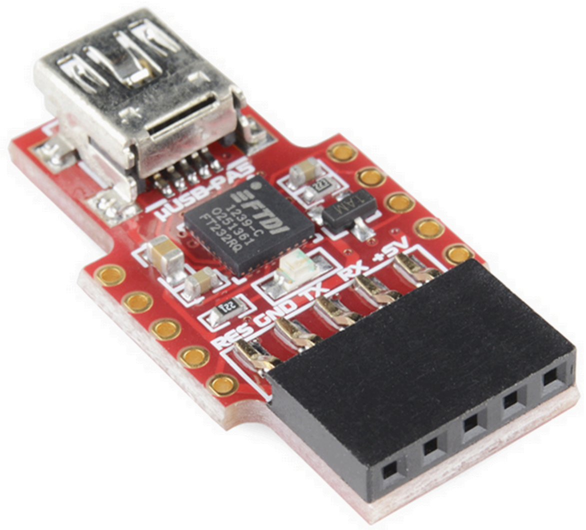


Figure 6: USB to serial with micro USB adapter

Software

- **Logic**

First, the microcontroller will wait until it receives frequency change command sent from the laptop using GetDataBuffer() function. Once it receives the command, it chooses the proper sine table size and timer period. We calculated the timer interrupt period using the following equation.

$$\text{Timer Period} = \frac{\text{System Frequency}}{SineTableSize * F_{out}} = \frac{60{,}000{,}000}{Sine\ Table * F_{out}}$$

F_out is the expected frequency output. System Frequency is the internal clock that the PIC32 is running at. Sine table is the size of the points we wants to output for one cycle of the sine wave. If sine table size is 256, it means there are 256 points in one cycle of the output sine wave. Since the system frequency is fixed, when the output frequency increases, we had to reduce the sine table size to keep the timer period at certain ticks. Otherwise, the precision of the sine wave would be affected. In our case we pick the following sine table size corresponding to different frequency range.

$$Sine\ Table\ Size = \begin{cases} 256 \quad Fout \le 5000 \\ 128 \quad Fout \le 10000 \\ 64 \quad Fout \le 20000 \\ 32 \quad Fout \le 100000 \\ 16 \quad Fout > 100000 \end{cases}$$

After that, we calculate the sine table value based on the sine function. We then output the values in the sine table using DMA (Direct Memory Access). The microcontroller will then continuously output the sine wave while waiting for the next command from the laptop.

- **Microcontroller Functions**

Void GetDataBuffer(char* buffer, int max_size):

This function builds a string from the bytes sent from the laptop. It continuously stores the character received from the UART buffer unless the characters stored in the buffer is larger than the max_size or a '\r' is received. We use this function to receive the frequency request from the laptop.

PPSInput (2, U2RX, RPB11) and PPSoutput(4, RPB10,U2TX):

These two macros set the pins for the UART functionality. PPSinput sets portB bit 11 as the UART receiver pin. Its physical pin number is 22. PPS output sets portB bit 10 as the UART transmitter pin. Its physical pin number is 21.

DmaChnSetEventControl(dmaChn,DMA_EV_START_IRQ(_TIMER_2_IRQ)) :

This function sets the event that starts the DMA transfer. In our case, the DMA starts transferring the data when there is a timer2 interrupt.

mPortBSetPinsDigitalOut(BIT_0lBIT_1lBIT_2lBIT_3lBIT_4lBIT_5lBIT_7):

This macro sets certain pins of Port B as digital output. In our case, we use B0 to B5 and B7 as our digital ourput.

OpenTimer2(T2_ONlT2_SOURCE_INTlT2_PS_1_1, timer_limit):

This macro sets up timer2. In our case, we set timer module on after we calculate the sine table, and we set internal clock as our timer 2clock source. We also set pre scalar to 1, and its period as the value we calculated based on different frequency. We pick the time_period calculated before for the timer_limit variable.

DmaChnSetTxfer(0,sine_table, (void*)&LATB,sine_table, 1,1):
This function set up how DMA transfers data. In our case, we use Dma channel 0, and transfer data from the sine table we calculated to Port B.

- **Console (laptop)**

In the console side, we defined a new serial port, and assigned it with the serial port, which connected microcontroller and laptop. After that we changed frequency every eight measurements since there were eight pairs of electrodes on the Flex circuit. In order to increase transfer speed, we only sent one character representing one certain frequency from the frequency table we need each time.

Test

Three major components would be tested to verify the performance of the function generator. 1.Signal quality. 2. Communication speed. 3. Portability and stability.

- **Signal quality**

Based on the specification, the desired output signal should be a sine wave with amplitude level around 100mV. Its frequency should able to vary between 1K to 100K Hz with acceptable noise level. Based on these two specifications, we sent command from the laptop to vary the output signal frequency, and recorded the corresponding frequency, amplitude and noise level. The frequency sets we tested are 1K, 10K, 50K and 80K.

- **Communication speed**

Besides signal quality, we also need to test whether the communication speed between the microcontroller and the laptop is faster than the original function generator. In order to test the communication speed, we recorded the overall test duration, and compared it with the original test duration.

- **Portability and stability**

Portability and stability are also important factors affect the performance of the entire system. Therefore, we need to check if the new function generator improves the portability as well as is stable enough for long-term use. In order to verify these two properties, we weighted and measured two function generators, and tested its stability by taking the entire system out of lab and carried it around.

Results

- **Signal Quality**

We captured sine wave signals 1k, 10k, 50k and 80k at two locations: one was before attenuation and the other one was after actuation. Figure 7-figure 15 show signals before attenuation and signals after attenuation. After the signal coming out of the attenuation circuits, slightly noises were observed with the signal. We ran the Fast Fourier Transform and recorded the dB differences between the fundamental frequency and higher hamonics. Results are shown in figure 15 to 17. From the results we can see the fundamental frequency equals to the acceptable frequency (with about $\pm 1Hz\ error$). The dB differences between the fundamental and higher hamonics are ranging between 28 dB and 32 dB. This noise level was expectable for the application. Therefore, we proved that the signal quality satisfied specifications.

Figure 18 -21 show the real data collected by the data acquisition device with the new function generator integrated in the system. The difference signal patterns can be used to indirectly measure the tissue strain. Additionally, we are able to measure the tissue impedance using the voltage and current data in the activated data (tissue in contact with the electrode pair) figure. The strain versus impedance data is shown in figure 22. These figures prove that the new function generator works properly with the current system.

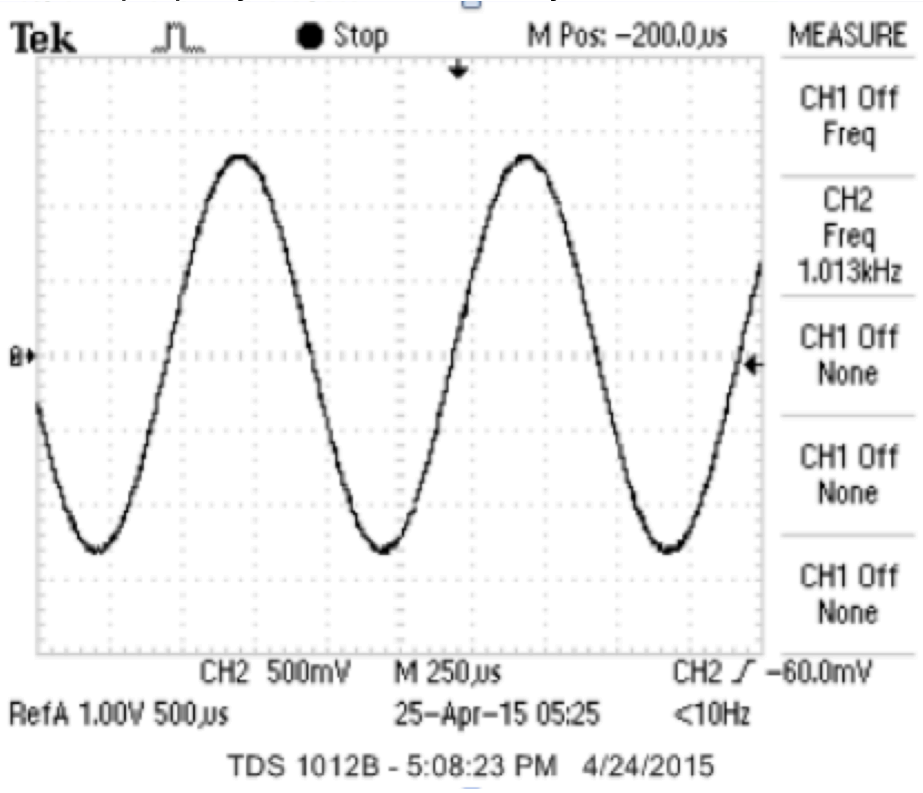


Figure 7: 1K sine wave coming out of the DAC

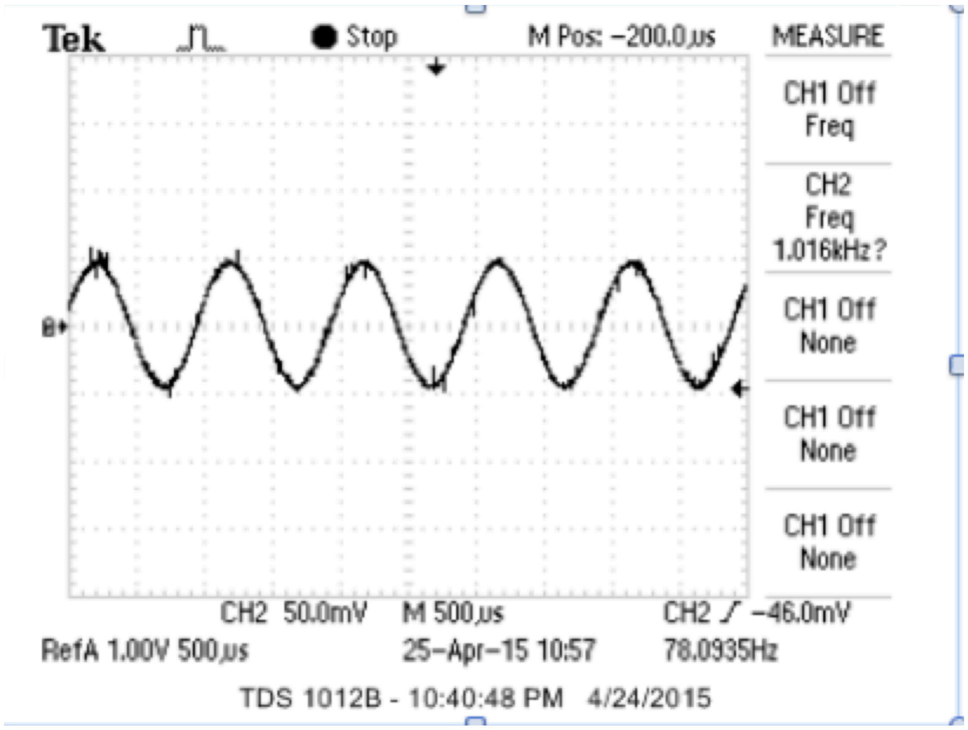


Figure 8: 1K sine wave coming out the attenuation circuit

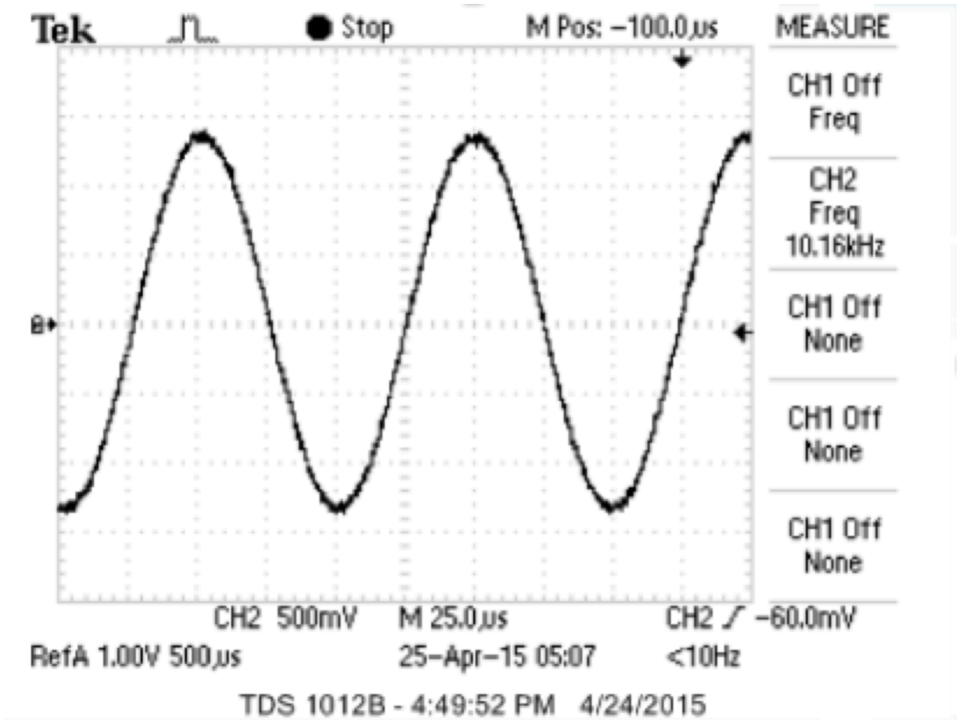


Figure 9: 10K sine wave coming out of the DAC

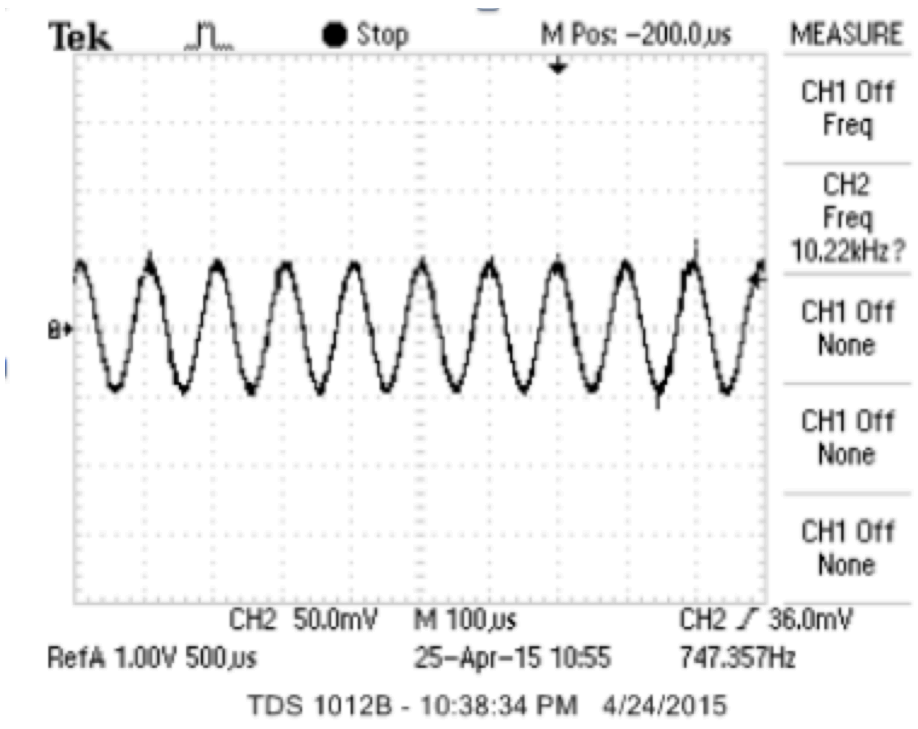


Figure 10: 10K sine wave coming out of the attenuation circuit

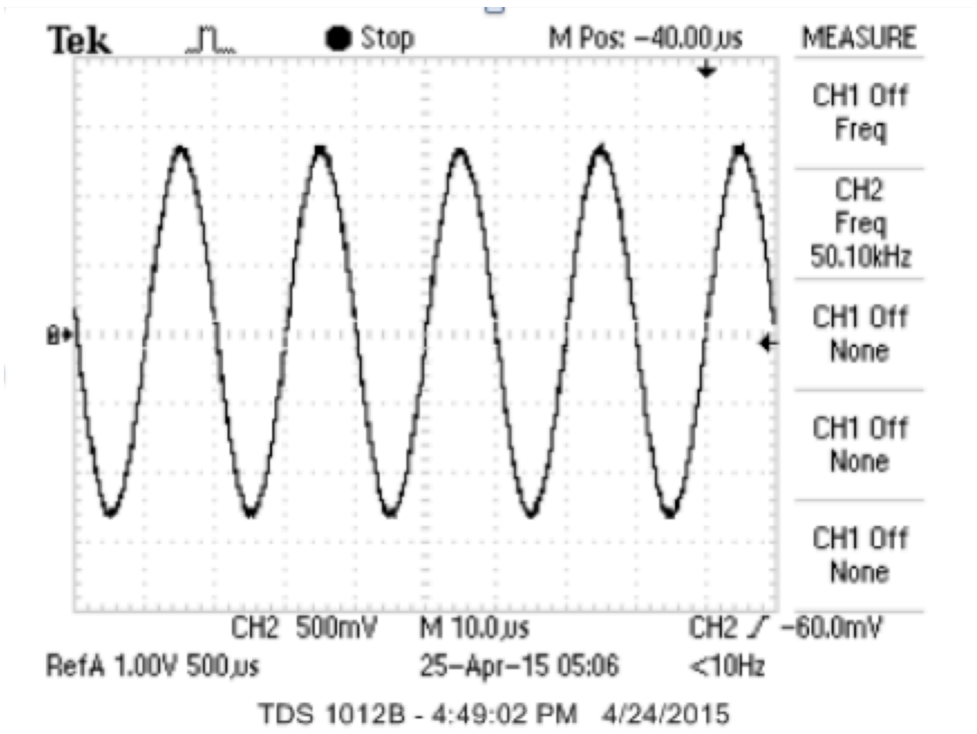


Figure 11: 50K sine wave coming out of the DAC

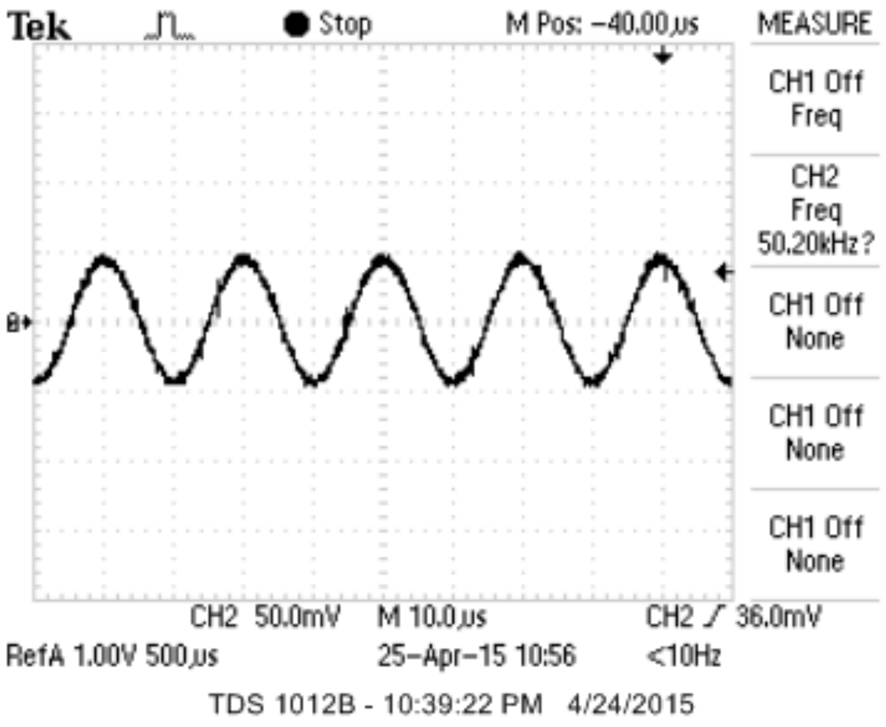


Figure 12: 50K sine wave coming out of the attenuation circuit

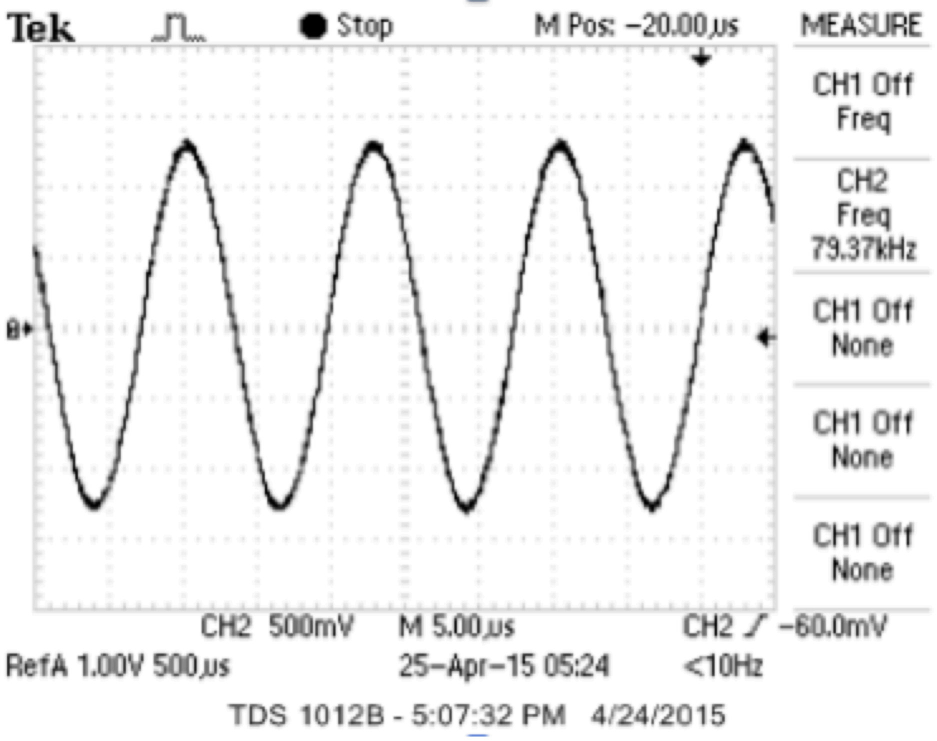


Figure 13: 80K sine wave coming out of the DAC

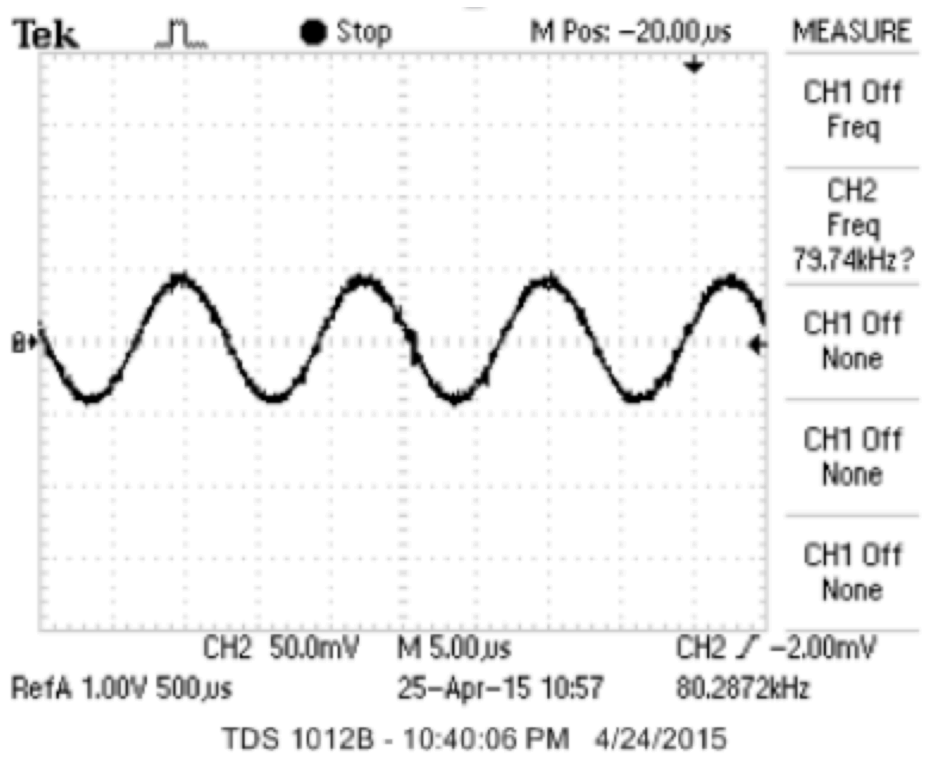


Figure 14: 80K sine wave coming out of the attenuation circuit

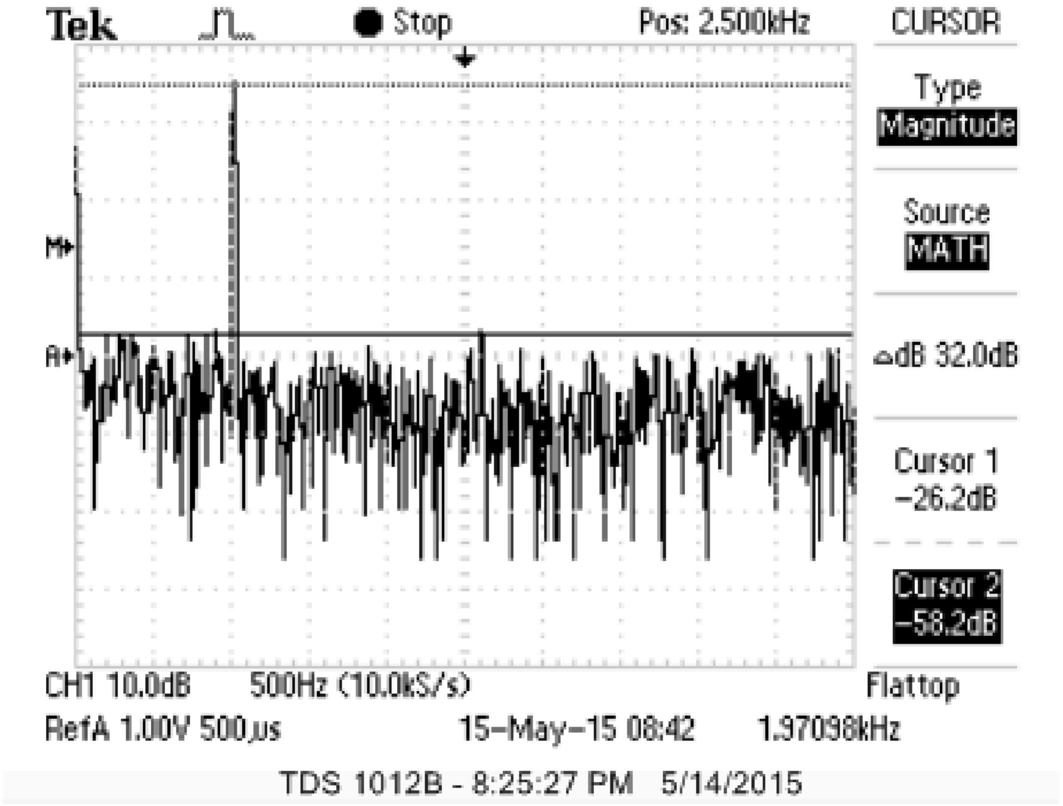


Figure15: dB differences at 1K Hz

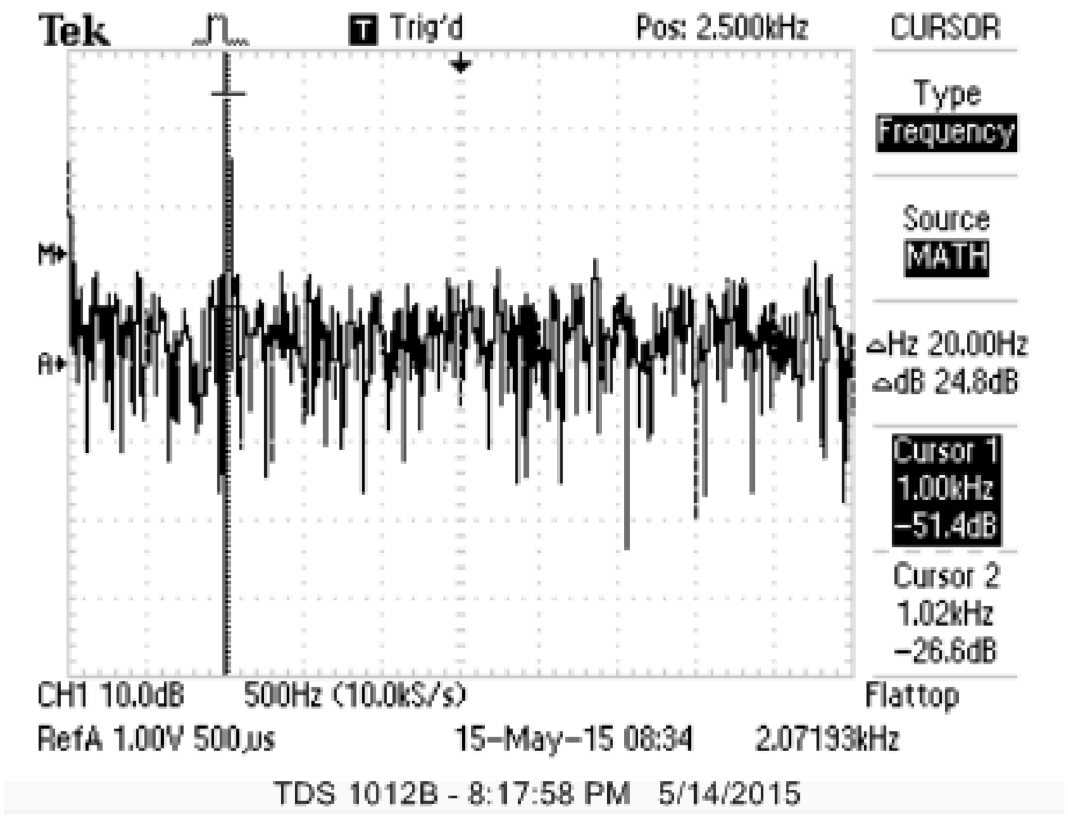


Figure16: dB differences at 10K

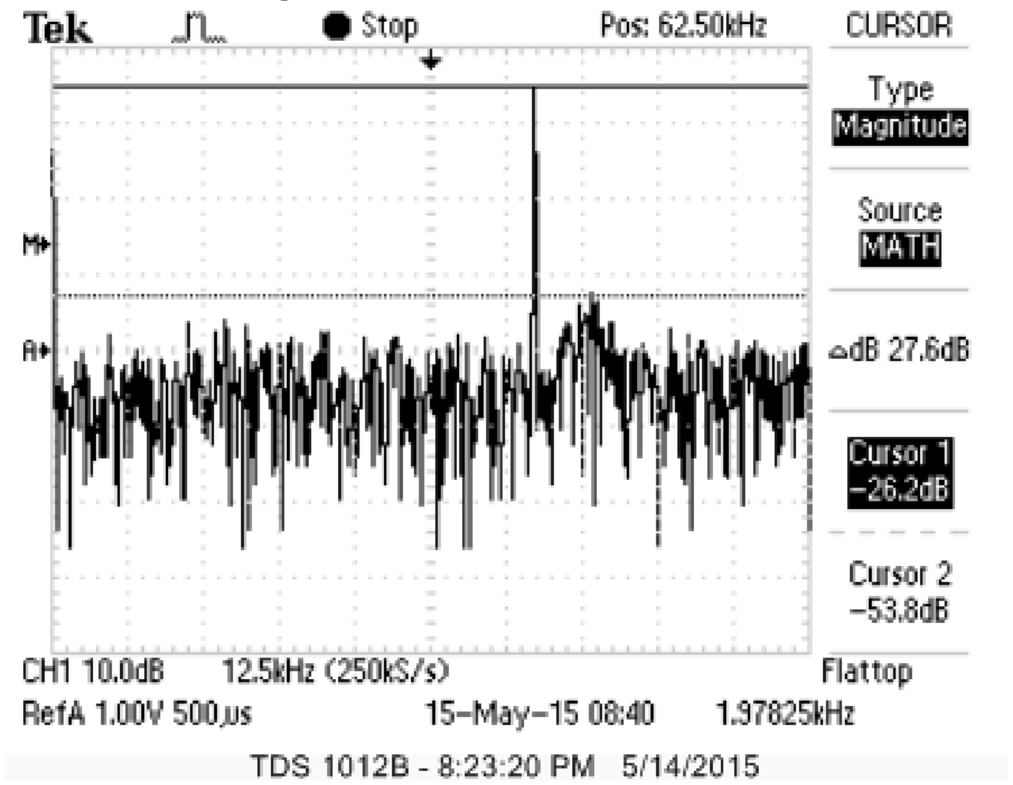


Figure 17: dB differences at 80K

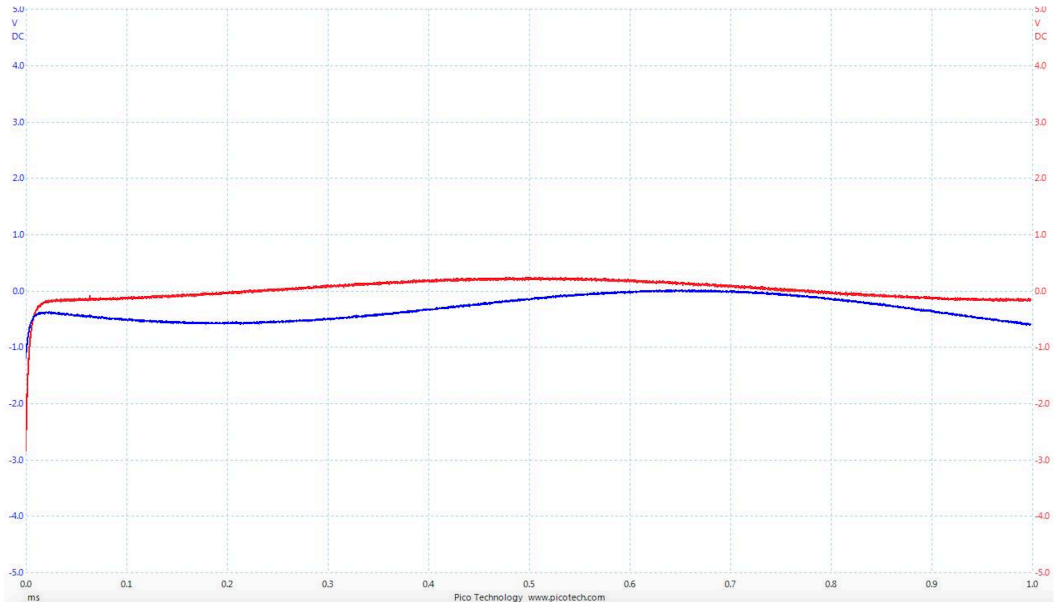


Figure 18: In tissue voltage and current at 1K

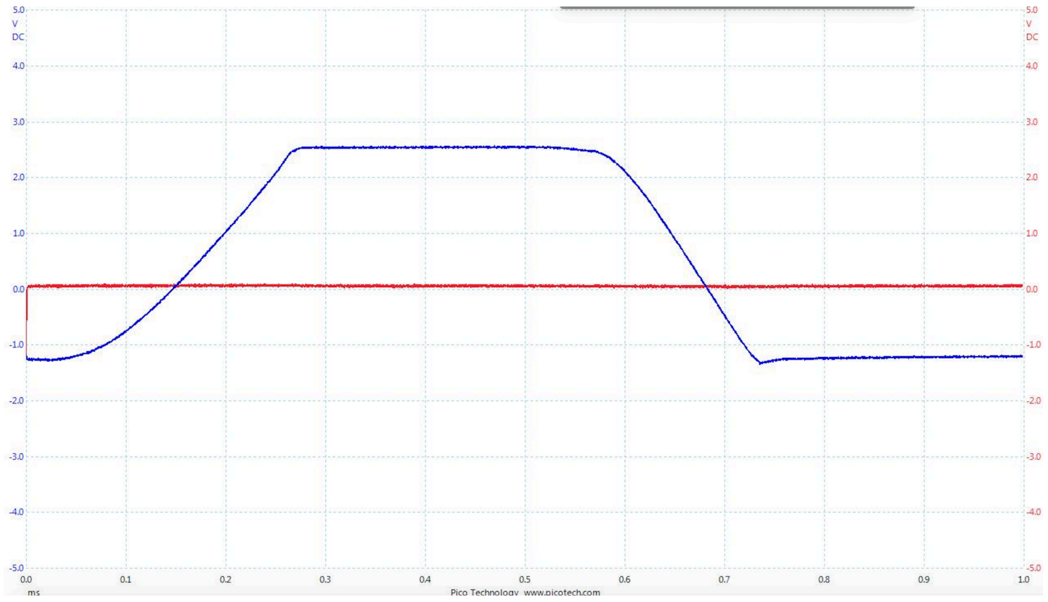


Figure 19: No contact voltage and current at 1K

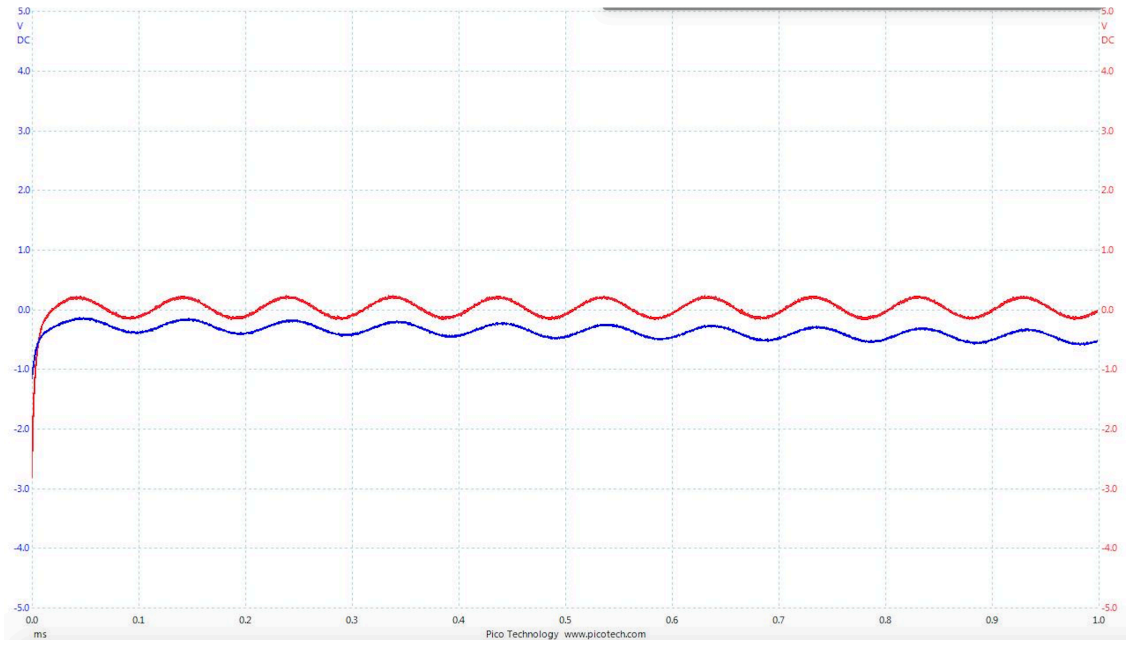


Figure 20: In tissue voltage and current at 10K

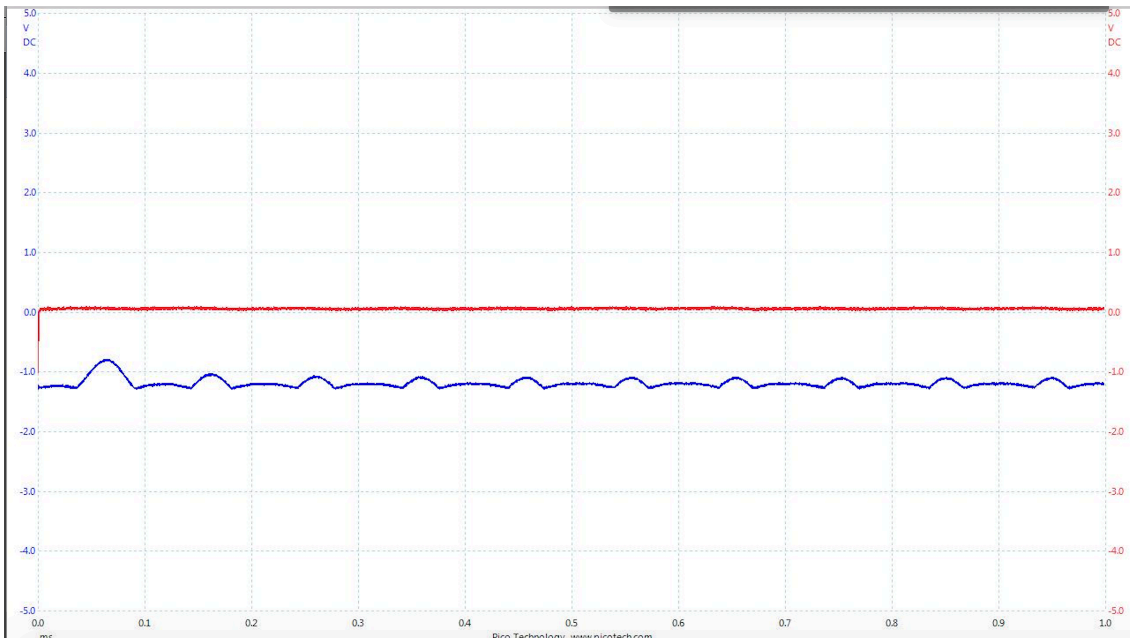


Figure 21: No contact voltage and current at 10K

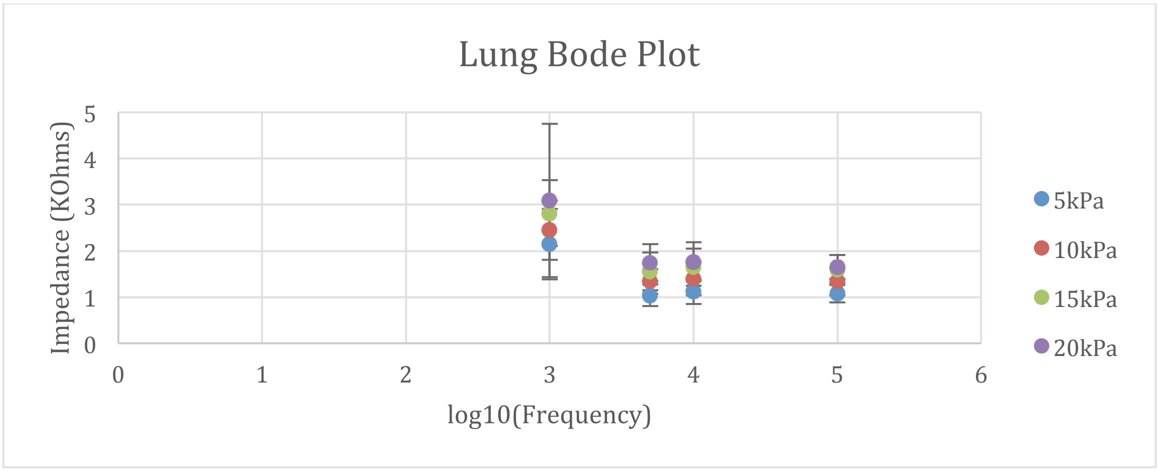


Figure 22: Pig lung impedance versus frequency (1k, 10k, 50k,100k) at different pressure point (different color)

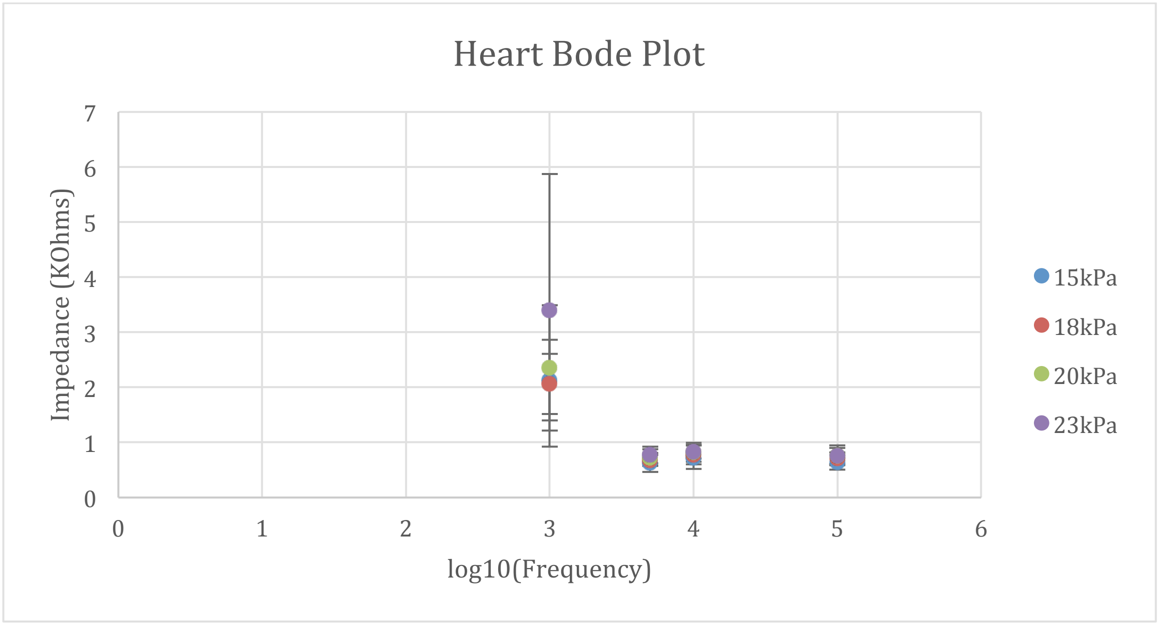


Figure 23: Pig Heart impedance versus frequency (1k, 10k, 50k, 100k) at different pressure point (different color)

- **Communication speed**

Based on the test results, we found for sweep 4 different frequencies (1K, 10K, 50K, and 80K) with 36 pressure points, it took the new function generator 6mintes to finish the test. However, for the original function generator, it took more than 20 minutes to complete the test. Additionally, it took the new function generator around 16ms (in average) to sweep frequency, which was much faster than the original function generator. Therefore, we proved that the communication speed was greatly improved.

- **Portability and stability**
  Current function generator is built on a solder board, which is about (4.7 x 4.7x 1.1) and weighted around 3.5 ounces. The original function generator is (5.9 x 8.07 x1.3 inches), and weighted 10 ounces. We concluded that the new function generator is much smaller and lighter than the original one. Hence, the new function generator improved the portability of the system. For the stability test, we first kept the function generator running for an entire day, and it worked fine.
  Besides that, we brought the whole system out of the lab, putting it inside a car and carried it around. After that, we brought it back and checked its functionality. We found the USB to serial cable didn't work because of the soldering part was losing contact with the circuit board. In order to improve the stability, a print PCB board with surface mounted electrical components would be necessary.

Acknowledgement

I would like to thank Dr. Land for his expert advice through the project as well as his kind help on implementing the microcontroller code and designing circuitry. I would also like to thank Dr. Butcher for his help as well as for giving me this valuable opportunity to broaden my knowledge on microcontroller applications. I would also like to thank Zhenchen for his help through the project.

Appendix 1: Microcontroller code and Console code

- Microcontroller Code:

```
/*********************************************************************
 *
 *  PIC32 Function Generator Code
 *  Bruce Land
 ********************************************************************/
// all peripheral library includes
#include <plib.h>
#include <math.h>

// Configuration Bit settings
// SYSCLK = 40 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
//
#pragma config FNOSC = FRCPLL, POSCMOD = HS, FPLLIDIV = DIV_2, FPLLMUL =
MUL_15, FPBDIV = DIV_1, FPLLODIV = DIV_1
#pragma config FWDTEN = OFF
// turn off alternative functions for port pins B.4 and B.5
#pragma config JTAGEN = OFF, DEBUG = OFF
#pragma config FSOSCEN = OFF

// frequency we're running at
#define         SYS_FREQ 60000000

// UART parameters
#define BAUDRATE 57600// must match PC end
#define PB_DIVISOR (1 << OSCCONbits.PBDIV) // read the peripheral bus divider,
FPBDIV
#define PB_FREQ SYS_FREQ/PB_DIVISOR // periperhal bus frequency

// useful ASCII/VT100 macros for PuTTY
#define clrscr() printf( "\x1b[2J")
#define home()   printf( "\x1b[H")
#define pcr()    printf( '\r')
#define crlf     putchar(0x0a); putchar(0x0d);
#define backspace 0x08
#define max_chars 10 // for input buffer

// receive function prototype (see below for code)
//int GetDataBuffer(char *buffer, int max_size);
void GetDataBuffer(char *buffer, int max_size);
//**************************************************
#define str_buffer_size 20
char str_buffer[str_buffer_size];

// the sine lookup table
volatile unsigned char sine_table[256];

// main ////////////////////////////
int main(void)
{
    int sine_table_size ;
    int timer_limit ;
    unsigned char s;
    //#define F_OUT 80000
    int F_OUT = 10000 ;
    // sine table index
    int i;
```

```
        __XC_UART = 2;

    /*     Initialize PPS */
    // Data sheet:
    //
http://people.ece.cornell.edu/land/courses/ece4760/PIC32/Microchip_stuff/2xx_da
tasheet.pdf
    // data sheet table 11-1 gives input pin mapping
    // data sheet table 11.2 gives output pin mapping
    // Also Section 12 of PIC32MX Family Reference Manual
    //
http://hades.mech.northwestern.edu/images/2/21/61132B_PIC32ReferenceManual
.pdf
    // macro defs
    // http://quadcopter-miami-ece.googlecode.com/svn/trunk/mpide-0023-
windows-20120903/hardware/pic32/compiler/pic32-
tools/pic32mx/include/peripheral/pps.h
    //
    // specify PPS group, signal, logical pin name
    PPSInput (2, U2RX, RPB11); //Assign U2RX to pin RPB11 -- Physical pin 22 on 28
PDIP
    PPSOutput(4, RPB10, U2TX); //Assign U2TX to pin RPB10 -- Physical pin 21 on 28
PDIP

    ANSELA =0; //make sure analog is cleared
    ANSELB =0;

    // init the uart2
    UARTConfigure(UART2, UART_ENABLE_PINS_TX_RX_ONLY);
    UARTSetLineControl(UART2, UART_DATA_SIZE_8_BITS | UART_PARITY_NONE |
UART_STOP_BITS_1);
    UARTSetDataRate(UART2, PB_FREQ, BAUDRATE);
    UARTEnable(UART2, UART_ENABLE_FLAGS(UART_PERIPHERAL | UART_RX |
UART_TX));

        // calibrate the internal clock
            SYSKEY = 0x0; // ensure OSCCON is locked
        SYSKEY = 0xAA996655; // Write Key1 to SYSKEY
        SYSKEY = 0x556699AA; // Write Key2 to SYSKEY
        // OSCCON is now unlocked
        // make the desired change
        OSCTUN = 31; // tune the rc oscillator
        // Relock the SYSKEY
        SYSKEY = 0x0; // Write any value other than Key1 or Key2
        // OSCCON is relocked
```

```
	// Configure the device for maximum performance but do not change the PBDIV
	    // Given the options, this function will change the flash wait states, RAM
	    // wait state and enable prefetch cache but will not change the PBDIV.
	    // The PBDIV value is already set via the pragma FPBDIV option above..
	    SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

	// frequency settable to better than 2% relative accuracy at low frequency
	// frequency settable to 5% accuracy at highest frequency
	// Useful frequency range 10 Hz to 100KHz
	// >40 db attenuation of  next highest amplitude frequency component below 
100 kHz
	// >35 db above 100 KHz



	// Set up timer2 on,  interrupts, internal clock, prescalar 1, toggle rate
	// Uses macro to set timer tick to 2 microSec = 120 cycles 500 kHz DDS
	// peripheral at 60 MHz
	//--------------------
	//------------------
	// 64 LEVEL samples thru external DAC
	// interval=60, 32 samples, 31200 Hz, next harmonic 40 db down
	// interval=20, 32 samples, 90.8 KHz, next harmonic 45 db down
	// interval=600, 32 samples, 3170 Hz, next harmonic 32 db down
	// interval=300, 64 samples, 3170 Hz, next harmonic 38 db down
	// interval=150, 128 samples, 3170 Hz, next harmonic 43  db down


	// Open the desired DMA channel.
	#define dmaChn 0
	    // We enable the AUTO option, we'll keep repeating the sam transfer over and 
over.
	    DmaChnOpen(dmaChn, 0, DMA_OPEN_AUTO);

	    // set the transfer event control: what event is to start the DMA transfer
	// In this case, timer2
	    DmaChnSetEventControl(dmaChn, DMA_EV_START_IRQ(_TIMER_2_IRQ));

	    // once we configured the DMA channel we can enable it
	    // now it's ready and waiting for an event to occur...
	    DmaChnEnable(dmaChn);

	// set up i/o port pin
	ANSELA =0; // turn off analog on A
```

```
    mPORTAClearBits(BIT_0);                    //Clear A bits to ensure light is off.
    mPORTASetPinsDigitalOut(BIT_0);    //Set A port as output
    PORTSetBits(IOPORT_A, BIT_0);
    // set up DAC pins
    ANSELB =0; // turn off analog on B
    // See also pragmas to turn off alternative functions
    mPORTBClearBits(BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5);
    mPORTBSetPinsDigitalOut(BIT_0 | BIT_1 | BIT_2 | BIT_3 | BIT_4 | BIT_5 | BIT_7 );
//Set port B as output
    i=0;
    clrscr();  //clear PuTTY screen

       while(1)
       {
      // toggle a bit for perfromance measure
       //mPORTBToggleBits(BIT_7);
       //printf(">");
       //printf("Please type a number for the frequncy\n\r");
       //sscanf("%d", &F_OUT);
       GetDataBuffer(str_buffer, max_chars);
       // check to see if a valid number was typed and use it
      // float temp;
       if (sscanf(str_buffer, "%d", &F_OUT)) {
         // printf("The frequnecy you entered is %f \n\r", temp);
         F_OUT=1000*F_OUT;
        printf("The frequnecy you entered is %d \n\r", F_OUT);
         if (F_OUT <= 5000 ){
           sine_table_size = 256 ;
           timer_limit = SYS_FREQ/(sine_table_size*F_OUT) ;
         }
         else if (F_OUT <= 10000 ){
           sine_table_size = 128 ;
           timer_limit = SYS_FREQ/(sine_table_size*F_OUT) ;
         }
         else if (F_OUT <= 20000 ){
           sine_table_size = 64 ;
           timer_limit = SYS_FREQ/(sine_table_size*F_OUT) ;
         }
         else if (F_OUT <= 100000 ){
           sine_table_size = 32 ;
           timer_limit = SYS_FREQ/(sine_table_size*F_OUT) ;
         }
         else {
           sine_table_size = 16 ;
           timer_limit = SYS_FREQ/(sine_table_size*F_OUT) ;
         }
```

```
            // build the sine lookup table
            // scaled to produce values between 0 and 63 (six bit)

            for (i = 0; i < sine_table_size; i++){
            //sine_table[i] =(unsigned char) (31.5 *
sin((float)i*6.283/(float)sine_table_size) + 32); //
              // s =(unsigned char) (63.5 * sin((float)i*6.283/(float)sine_table_size) +
64); //7 bit
              s =(unsigned char) (63.5 * sin((float)i*6.283/(float)sine_table_size) + 64);
//7 bit
              sine_table[i] = (s & 0x3f) | ((s & 0x40)<<1) ;
             }
            // set up timer
             OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, timer_limit);
             // set up DMA
             // set the transfer parameters: source & destination address, source &
destination size, number of bytes per event
            // Setting the last parameter to one makes the DMA output one
byte/interrupt
            DmaChnSetTxfer(dmaChn, sine_table, (void*)&LATB, sine_table_size, 1, 1);


         }

          }
          return 0;
}
////////////////////////////////////////////////////////////////////
// build a string from the UART2 /////////////
///////////////////////////////////////////////
//int GetDataBuffer(char *buffer, int max_size)
void GetDataBuffer(char *buffer, int max_size)

{
  int num_char;
  num_char = 0;
  memset(buffer, 0, max_size);

  while(num_char < max_size)
  {
    char character;

    // get the character
    while(!UARTReceivedDataIsAvailable(UART2)){};
    character = UARTGetDataByte(UART2);
    UARTSendDataByte(UART2, character);
```

```
        // end line
        if(character == '\r'){
           *buffer = 0; // zero terminate the string
           //crlf; // send a new line
           break;
        }
        // backspace
        if (character == backspace){
           putchar(' '); // write a blank over the previous character
           putchar(backspace); // go back one position
           buffer--;
           num_char--;
        }
        else { // save it all
           *buffer = character;
           buffer++;
           num_char++;
        } //if (character == backspace)

     } //while(num_char < max_size)
     //return num_char;
  }
```

- Console code:

```
//Uart Port Definition:
    p = new SerialPort("COM11");
                p.BaudRate =57600;
                p.Parity = Parity.None;
                p.StopBits = StopBits.One;
                p.DataBits = 8;
                p.Handshake = Handshake.None;
                p.DataReceived += new
SerialDataReceivedEventHandler(p_DataReceived);
                p.Open();
//Write frequency every 8 cycles:
                   if (testCount >= 8)
                   {
                      portMsg = Convert.ToString((1.0f * (100 * 1000000) /
nextSet.hold_us)/1000);

                      if (portMsg.Length != 0)

                      {
```

```
    p.Write(portMsg + "\r\n");
    Thread.Sleep(30);

}
```