A Design of Complex Impedance Meter

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

by

Yi Zhang

Project Advisor: Bruce Land

Degree Date: January 2007

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: A Design of Complex Impedance Meter

Author: Yi Zhang

Abstract:

The Complex Impedance Meter presented in this report is a high precision, low power consumption impedance measurement system which provides programmable frequency sweep and tuning capability for impedance measurement from 5kΩ to 1.1MΩ with system accuracy of 2%. The system is built with an AVR microcontroller ATmega32, an Analog Devices network analyzer AD5933, two analog multiplexers ADG706, a CTS low jitter clock oscillator MXO45HS, two accurate resistor networks with designed resistance values and a liquid crystal display RCM2034R. It allows an unknown external impedance to be excited with a known frequency. The response signal from the impedance is sampled by the on board ADC, and the DFT is processed by a DSP engine which returns a real and imaginary data word at each excitation frequency. The magnitude of these data words is further scaled by calibrated Gain Factor in order to return the actual impedance value. The prototype of the system is implemented and system calibration is done to improve the overall accuracy.

Report Approved by
Project Advisor: Date:

Executive Summary

A prototype of a Complex Impedance Meter is designed and implemented. The system is a microcontroller based, high precision and low power device. Due to the simple structure, it can be made as a portable impedance measurement device, which is used in a lot of scientific and industrial fields such as electrochemical analysis, bioelectrical impedance analysis and material property analysis. The accuracy of 2% is realized through the appropriate design and system calibration. This accuracy is good enough for most of the biomedical impedance measurement applications.

The system has been designed as an intelligent and friendly device, which does not need any adjustment or configuration before a measure. The maximum system response time is 0.5 second, which means in the worst case users can read the measured impedance on the LCD within half a second. If the impedance being measured is out of range, it will also give a message on the display indicating the error. The hardware design was implemented on several prototype boards. The values of the precision resistance were carefully calculated and chosen so that good linearity and measurable range (5kΩ to 1.1MΩ) can be achieved. Operating software was designed and implemented to realize the intelligence and easy-to-use features. The design trades off the maximum system response time with the intelligence and keeps a good balance between these two specs. Finally calibration was carried out and two methods were used to further improve the linearity of the system.

Introduction

From biological cell analysis to fuel cell tests, from coatings to cement paste quality control, Complex Impedance Measurement (CIM) has become a powerful tool in the vast environment of those applications. The basic principle of CIM is modeling the unit under test (UUT) in a combination of electrical components, applying small amplitude of AC voltage or current to the ends of UUT, over the frequency band of interest. For each frequency in the range, the measured impedance is a complex ratio between the input and output signal.

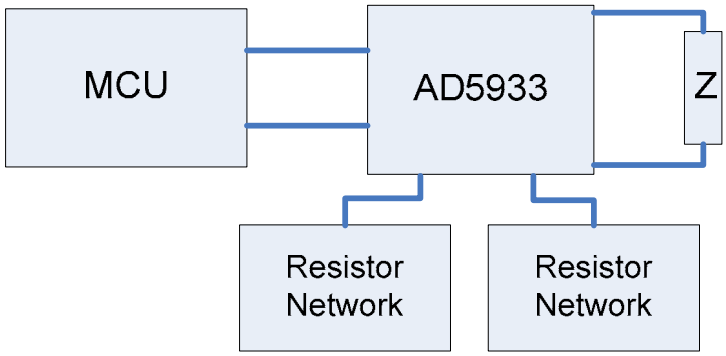

Fig.1 Top Level Diagram

In the design presented in this report, this principle is adopted into the architecture of the system. As Figure 2 shows, the on board frequency generator generates stimulus signal with known frequencies. The external unknown impedance is connected between the input and output ports. The response current from the impedance is converted into voltage by a trans-impedance amplifier. The output voltage of this amplifier is sampled by an on board ADC and the DFT is processed by a DSP engine at each excitation frequency. The real and imaginary results from the DSP are further processed by the microcontroller. They are compared with the results obtained from a known precise resistance using the same configuration on the signal path. The actual impedance value can be calculated from the known resistance together with the above measured results.

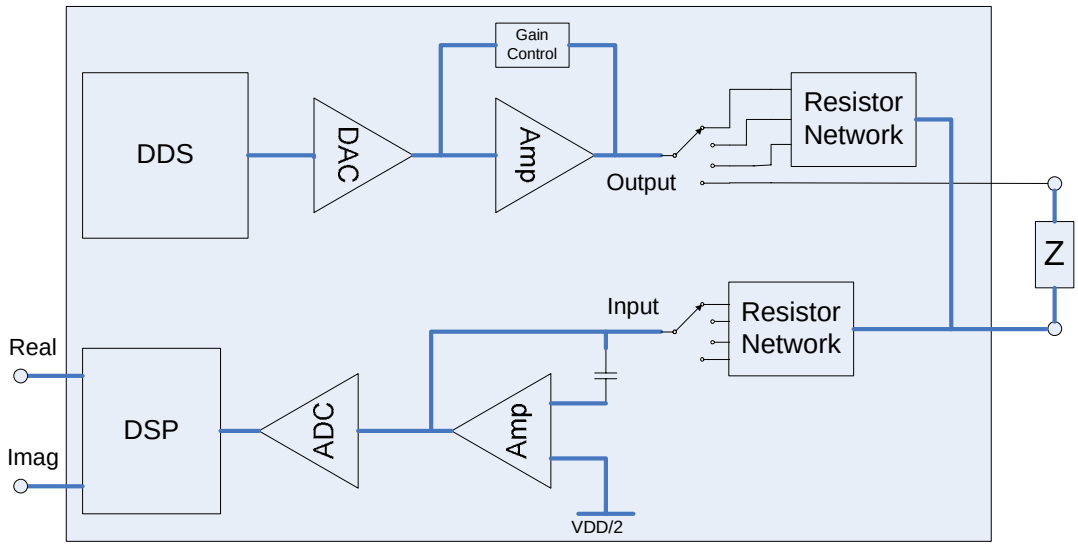

Fig.2 System Block Diagram

The Principle of Measurement

The basic measurement method of AD5933 is comparing the two measured results from the unknown impedance and the calibrated (known) resistance. And then the value of the unknown impedance is obtained through calculation. As the signal path shown in Figure 3, V_{OUT}, the output of the signal generator, can be modeled as a voltage source. The unknown impedance is connected between V_{OUT} and V_{IN}. V_{IN} is biased at VDD/2 causing the AC current through $Z_{unknown}$ and R_{FB} to be equal to $V_{OUT}/ Z_{unknown}$. If V_{OUT}, R_{FB} and the gain along the signal path is known, the unknown impedance can be calculated by sampling and processing the voltage at the output of low pass filter.

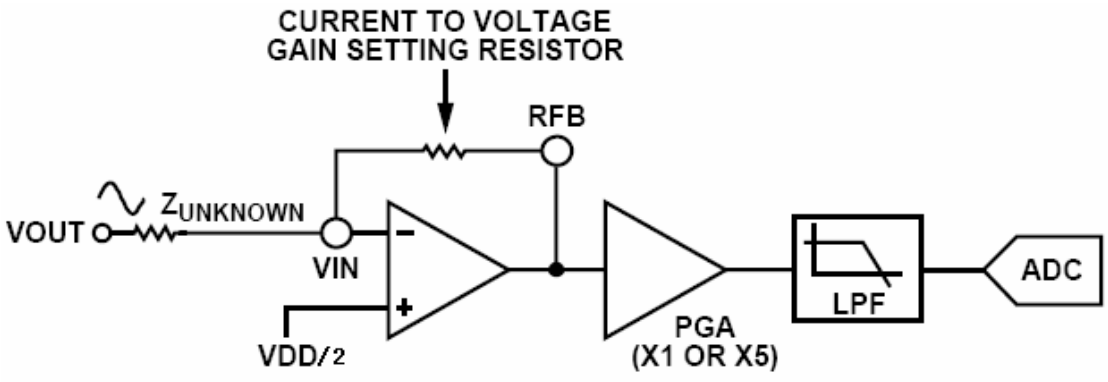

Fig. 3 AD5933 Signal Path

For accurate measure of the impedance, it is important that the receive stage is operating in its linear region. This requires careful selection of the excitation signal range, current-to-voltage gain resistor, and PGA gain. The gain through the system shown in Figure 3 is given by

$$SystemGain = OutputExcitationVoltageRange \times \frac{GainSetting\ \mathrm{Re}\,sistor}{Z_{Unknown}} \times PGAGain$$

---------- (1)

There are two issues related to the signal saturation problem. First, current from the external unknown impedance flows through the V_{IN} pin and into a trans-impedance amplifier which has a user determined external resistor across its feedback path. The output voltage of the trans-impedance amplifier is determined by the closed loop gain $-R_{FB}/Z_{unknown}$ and V_{OUT}. The positive node of the amplifier is biased at a fixed value of VDD/2, large difference between the positive and negative nodes can saturate the output the amplifier. The R_{FB} and the V_{OUT} should be chosen so that this voltage remains in the linear region. Second, the gain of the PGA should be properly set to make best use of the dynamic range of the ADC but not to saturate the following stages. Both of these issues rely on an approximation of the $Z_{unknown}$ and proper selection on the feedback resistor.

As the datasheet of AD5933 suggests, to get best accuracy, the $R_{FB}/Z_{unknown}$ ratio should be within 0.066~0.2. The resistance values of the feedback network are determined as 4.7k, 10k, 27k, 57k, 200k, 470k, 1M. The measurable impedance range is then divided into 7 segments. One of the values is picked after the range of the unknown impedance is estimated. The way to estimate the unknown impedance is discussed in the software design part.

System Design

The microcontroller is the control unit and data processor in the system. It controls the AD5933 and two analog switches during the measures. After calculation and calibration based on the data from AD5933, the microcontroller sends the results to the LCD, where users can read the impedance values.

The hardware connections between the microcontroller and the functional chips are shown below.

Name	Connection Type	Line numbers	Description
AD5933	I^2C	2	AD5933 control and data transfer
ADG706 (1)	Parallel	4	Feedback resistor selection
ADG706 (2)	Parallel	4	Calibration resistor selection
RCM2034R	Parallel	7	LCD control and data transfer

Table.1 Intrasystem Connection

The microcontroller used in this project has four 8-bit I/O ports. All I/O pins are shared with alternate function pins. By setting the I/O ports to a proper status, they can be used as either general digital I/O or I/O pins of functional blocks in ATmaga32.

I^2C bus

I^2C bus is a Serial Interface Protocol. The AD5933 is connected to this bus as a slave device, under the control of a master device, which is the ATmega32 here. The AD5933 has a 7-bit serial bus slave address. When the device is powered up, it will do so with a default serial bus address, 0001101.

Figure 4 shows the timing diagram for general read and write operations using the I^2C interface.

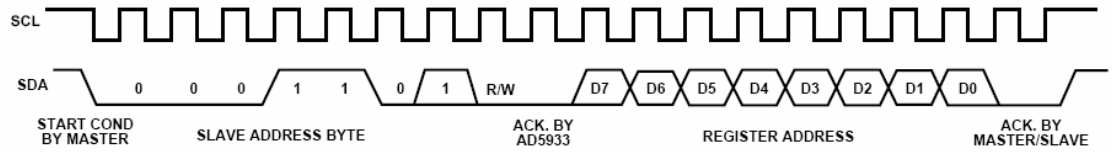

Fig. 4 General I^2C protocol

The master initiates data transfer by establishing a start condition, defined as a high to low transition on the serial data line (SDA) while the serial clock line (SCL) remains high. This indicates that a data stream follows. The slave responds to the start condition and shifts in the next 8 bits, consisting of a 7-bit slave address (MSB first) plus an R/W bit, which determines the direction of the data transfer. That is, whether data is written to or read from the slave device (0 = write, 1 = read).

The slave responds by pulling the data line low during the low period before the ninth clock pulse, known as the acknowledge bit, and holding it low during the high period of this clock pulse. All other devices on the bus remain idle while the selected device waits for data to be read from or written to it. If the R/W bit is 0, then the master writes to the slave device. If the R/W bit is 1, the master reads from the slave device.

Data is sent over the serial bus in sequences of nine clock pulses, 8 bits of data followed by an acknowledge bit, which can be from the master or slave device. If the operation is a write operation, the first data byte after the slave address is a command byte. This tells the slave device what to expect next. It may be an instruction telling the slave device to expect a block write, or it may be a register address that tells the slave where subsequent data is to be written. Data can flow in only one direction as defined by the R/W bit.

When all data bytes have been read or written, stop conditions are established. In write mode, the master pulls the data line high during the 10th clock pulse to assert a stop condition. In read mode, the master device releases the SDA line during the low

period before the ninth clock pulse, but the slave device does not pull it low. This is known as a no acknowledge (NACK). The master then takes the data line low during the low period before the 10th clock pulse, then high during the 10th clock pulse to assert a stop condition.

LCD connection and control

The RCM2034R is a reflective TN type liquid crystal module with a built-in controller / driver LSI and a display capacity of 16 characters with 2 lines. It supports both 4-bit and 8-bit operations. That is, data transfer with two transmissions of 4 bits at a time or one transmission of 8 bits at once. When using 4-bit operation mode, data is transferred along DB4 through DB7 buses and DB0 through DB3 buses are not used. (DB0-DB7 is the bi-directional data bus on RCM2034R.) Data transfer is completed after two transfers of 4 bit data. First the upper nibble (contents of DB4 through DB7 during 8-bit interfacing) is transferred and then the lower nibble (contents of DB0 through DB3 during 8-bit interfacing) is transferred.

The 4-bit operation mode is used in this system so that a single 8-bit port connection is enough for this operation mode. Besides 4-bit data bus, three control lines need to be connected to complete the hardware connection between the microcontroller and the LCD. Table-2 shows the connection details and their functions.

Symbol	Input/output	Function	MCU Connection
RS	Input	Register selection signal.	PD0
R/W	Input	Reading and writing selection signal	PD1
E	Input	Data reading and writing start signal	PD2
DB4-DB7	Input/output	4-bit operation data bus	PD4-PD7

Table.2 LCD Connection

Analog Switches control

The ADG706 is a low-voltage, CMOS analog multiplexers comprising 16 single channels. The ADG706 switches one of 16 inputs (S1–S16) to a common output, D, as determined by the 4-bit binary address lines A0, A1, A2, and A3. An EN input is used to enable or disable the device. When disabled, all channels are switched off. An 8-bit parallel port is just good for two switches with both of the EN inputs are tied to VCC. They are used to switch the resistor networks between V_{OUT}, V_{IN} and R_{FB}. The common terminals of these two switches are connected to V_{OUT} and V_{IN} respectively. The reference resistors are connected together to R_{FB} on one end while the other ends are connected to different inputs on the switch. Similarly, the calibration resistors are connected together to V_{IN} on one side. Figure 5 shows the connection of the resistor

networks and switches.

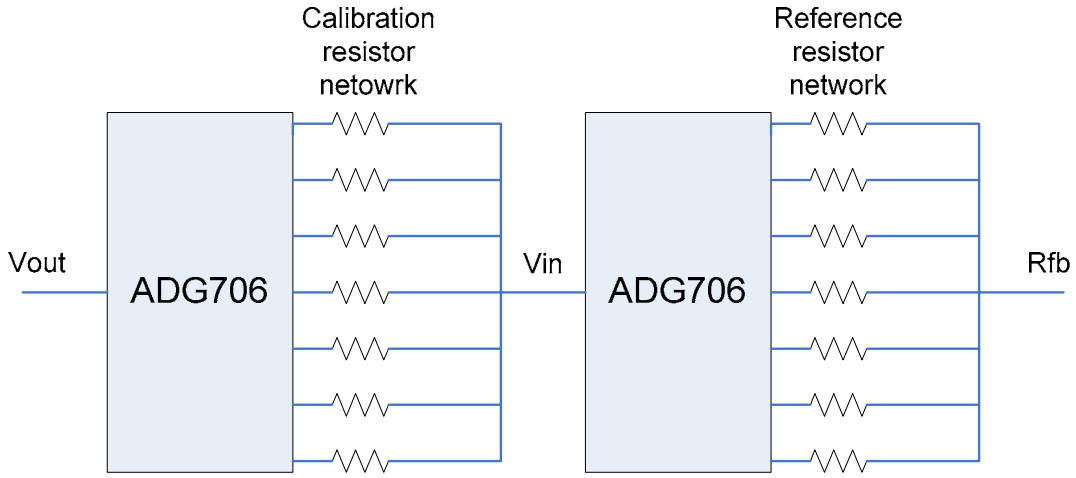

Fig. 5 Precision resistor networks

Crystal Oscillator

The frequency generator in AD5933 is based on DDS technique. There are two choices to get the reference frequency. The AD5933 has an on chip RC oscillator, which has an output frequency of 16.7MHz±0.2 MHz with a 330ppm jitter spec. Relatively high phase noise of this internal oscillator can seriously affect the output stimulus signal, thus the accuracy of the measured impedance because all the following calculation is based on the nominal output frequency. The other choice is using an external clock to feed the 16.6MHz reference. A crystal oscillator with high stability of 50ppm is used. Though it consumes extra power, the frequency accuracy of the output signal can be greatly improved.

Software design

I^2C bus

Since the commands and data bytes between microcontroller and AD5933 are transferred through I^2C bus, it is necessary to establish a function library containing all the read and write functions for the communication before applying high level application commands. In this design, the function library is divided into two levels. As shown in Figure 6, the low level functions complete the basic operations including register setting, reading and writing. The high level functions are based on the low level one. They use these basic register operations, as well as appropriate timing and protocol of AD5933, to complete all the communication operations AD5933 supports. Those functions include byte writing/reading, block writing/reading, address point setting. They are the routines called by the application level.

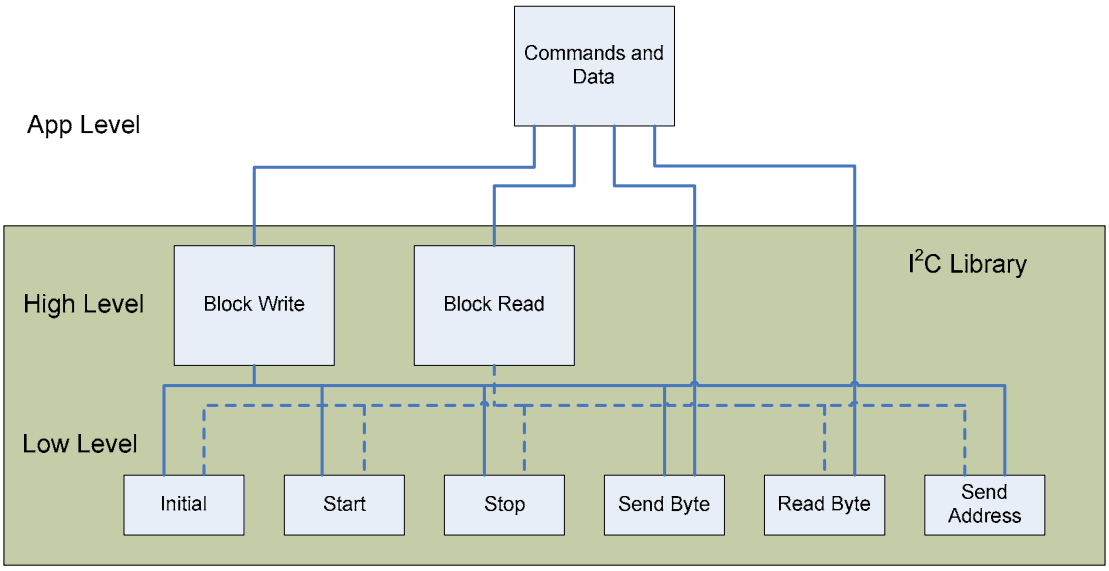

Fig. 6 I²C bus function library

LCD display

There are existing functions for LCD control in ATmage32 function library. But most of them are low level functions which can only complete such tasks as putting one character on the screen, move the cursor to certain point or clear the screen. Higher level functions are needed to bridge the low level to the application level. Since the only information that needs to be shown on the screen are the current testing frequency and the measured impedance, the format of the displayed words are similar. A function with two inputs, the frequency and impedance, is written as the interface to the application level. After every measurement, the application program calls this function to output the measured results.

Estimate the unknown impedance

The AD5933 is capable of measuring impedance values by providing the real (R) and imaginary (I) code. The magnitude of the real and imaginary data contents is given by $Magnitude(f) = \sqrt{R^2 + I^2}$. This magnitude value is equal to a scaled value of the actual impedance under test at the frequency point f. In order to determine actual impedance value users must multiply the magnitude by a number called Gain Factor GF(freq, Vdd, temp), which is a value representing the accumulative gain through the signal path of the system for known calibration impedance for a specified value of output gain voltage/pre ADC gain and feedback resistor settings. Therefore the actual impedance at any sample instance is given by the following,

$$\text{Im } pedance(f) = GainFactor \times Magnitude(f)$$

The Gain Factor is measured using a known external impedance, e.g., a precision resistor, connected between Vout and Vin as close as possible to the pins. Calculating the GF in this way calibrates out the parasitic impedance between Vout and Vin at a given frequency. The parasitic impedance is made up of a parallel capacitance

between Vout and Vin as well as a series resistance and series inductance mainly due to the bond wires and solder joints.

As there are several gain settings on the signal path, any adjustment to the supply voltage, calibration frequency, output excitation level, external feedback resistance value, and pre-ADC voltage gain will require a recalculation of the GF. If the GF is not recalculated after any system gain parameter adjustment then the impedance value returned by the AD5933 will have an error associated with it. Therefore the accuracy of final results depends upon the value of the GF.

The gain factor is dependant upon the ratio of the trans-impedance feedback resistance value R_{FB} to the impedance under test, $Z_{unknown}$. In order for the AD5933 to return accurate values, it is necessary to ensure that the largest signal is returned to the ADC while ensuring that gain factor will not vary significantly over the unknown impedance range. Minimising the gain factor variation is achieved by placing the AD5933 operating point in the flat region the variation of the gain factor. The ratio of feedback resistance to calibration impedance should lie in the range of 0.2~0.066, recommended in the data sheet.

These two values are chosen based on the knowledge of the range of unknown impedance. A straightforward and good way to estimate the unknown impedance is trial and error. Figure 7 shows the impedance estimation flow diagram. The estimation starts from the middle of the measurable impedance range, e.g., 27kΩ-57kΩ. The feedback resistor corresponding to this impedance range is chosen. Then the magnitude code of this unknown impedance under this feedback situation can be obtained. Compare this value with the upper and lower limit of the magnitude codes, which are the boundary values for the input signal staying in the linear range. If the measured value is not within the limits, the estimation process will stay in the loop and go through the steps discussed above again with a changed impedance range/feedback resistor. When the magnitude is larger than the upper limit, the next larger resistor will be chosen. When smaller, the next smaller resistor will be chosen. The loop will keep running until it finds a proper feedback resistor which makes the magnitude code within the limits or the unknown impedance is out of range. In either way, the loop stops. The magnitude code is recorded as one of the outputs of the loop when the impedance range is found while an error routine is called when it is out of range.

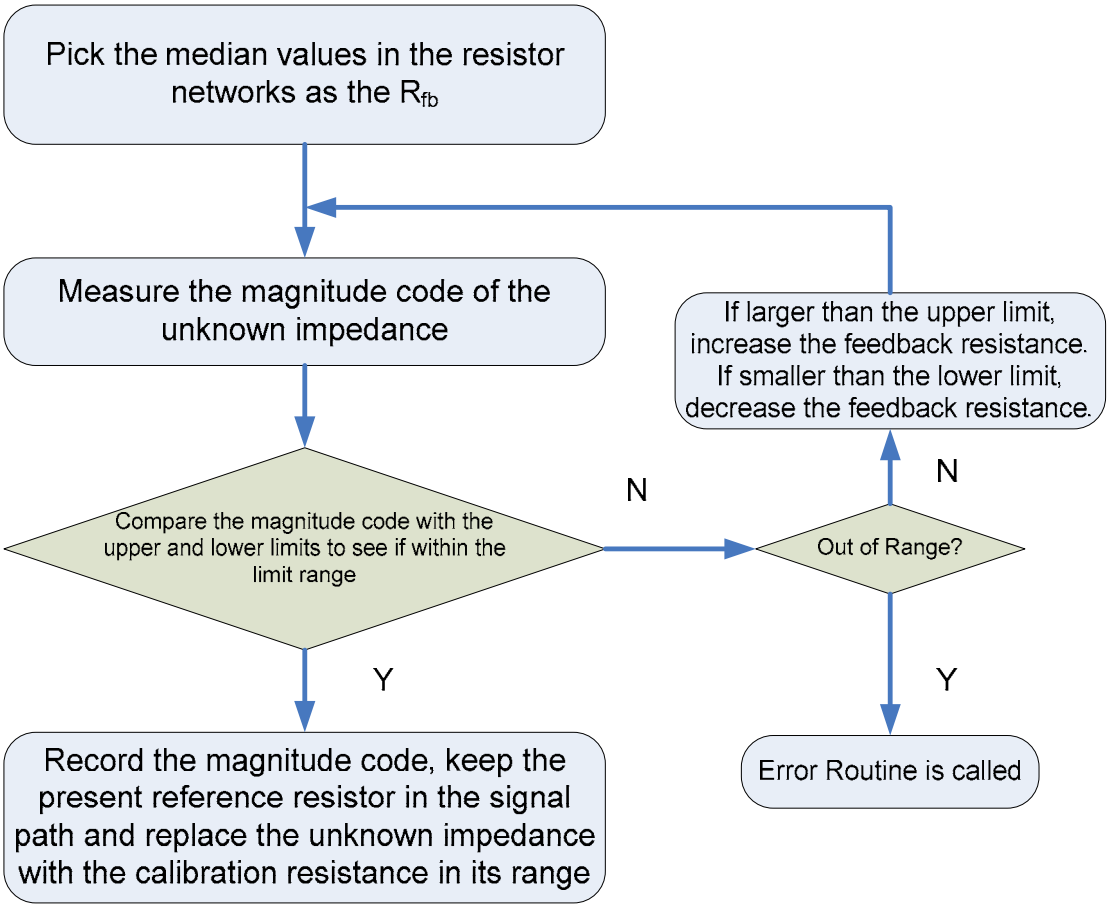

Fig. 7 Impedance Estimation Flow Diagram

System test and calibration

The prototype of system is implemented with resistor networks consisting of high precision resistor (0.5%). Since these resistors are only accurate under low frequency, the test and calibration is conducted within its precision range. According to the discussion above, a proper feedback and calibration resistor should be chosen so as to get the best measurement accuracy. However, even if the case, the measurement results may not be within the accuracy range of 2%. An important assumption of the measurement is that the gain factor within one of those defined impedance ranges is constant. That is, the magnitude code is simply a linear function of the impedance. But the real situation is that the magnitude code becomes non-linear gradually as the unknown impedance approaches the boundaries of the defined impedance range. If the magnitude code of the calibration impedance is still used as a reference for such cases, results are not accurate. Figure 8 shows the linearity of magnitude code from 33K Ω to 72K Ω. It can be seen that if the unknown impedance is located in the lower part of this impedance range, the results may have large errors due to the non-linearity.

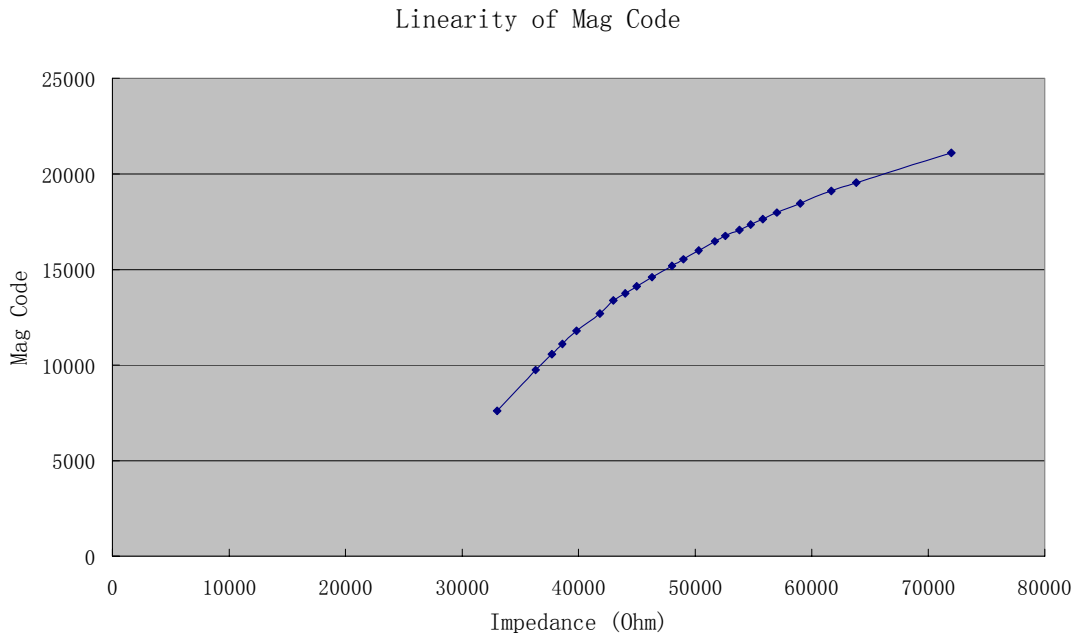

Fig. 8 Linearity of magnitude code

There are two ways to decrease the effect of this phenomenon. The first is to divide the impedance range into more sections. It is easy to see that if we can divide the impedance range shown in Figure 8 into 3 sections, 33K~45K~56K~72K, each of which is much more linear inside the range. However the cost of this way is to complicate the resistor networks, thus the whole system and more important, the measurement speed will be greatly lowered because of the nature of the impedance estimation method. In the worst case, the measurement time increases by three times as one impedance range is divided into three. The second way is to take advantage of the feature of signal path gain control on AD5933 to linearize the system. Since the magnitude code has linear relation to the gain through the system and the gain can be expressed as (1), changing the magnitude of output signal and/or the gain of the preamplifier has the same effect of changing the value of feedback resistor (gain setting resistor) on the magnitude code. So we can use two adjacent values of the feedback resistors (one in the impedance range where the unknown impedance located and the other in the adjacent range) to cover the same impedance range. Since the feedback resistors used in these two measurements are different, two different linearity curves can be obtained, which can be used to linearize the magnitude code to the desirable linearity. This needs to completely plot the linearity curves for all the impedance ranges and finely tune the parameters to get the best combination of these two linearity curves. But all these work can be done in the design stage. So it will not affect the measurement time during the use. The final design of this project adopted both of the two ways and the slowest measurement takes 0.5 second, which occurs in very low frequency range. Figure 9 shows the system linearity of the final design for the whole measurement range. The linearity can be calculated as

$$Linearity = \frac{Largest DeviationFromTheIdealCurve}{TheFullMeasurementRange} = \frac{21K\Omega}{1100 - 5K\Omega} = 1.9\%$$

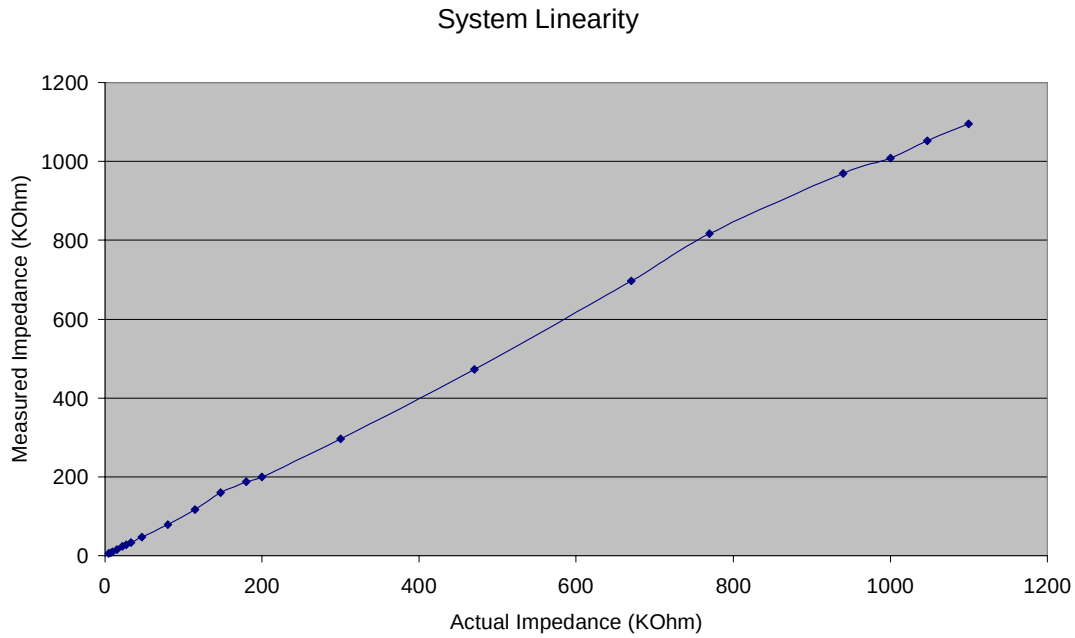

Fig. 9 System Linearity

Conclusions and Future work

A prototype of complex impedance meter is designed and implemented. The system is a microcontroller based, high precision and low power device. Due to the simple structure, it can be made as a portable impedance measurement device, which is used in a lot of scientific and industrial fields such as electrochemical analysis, bioelectrical impedance analysis and material property analysis. The accuracy of 2% is realized through the appropriate design and system calibration. This accuracy is good enough for most of the biomedical impedance measurement applications.

Due to the limitation of the precision resistor networks used in the system, the accuracy is only proved in relatively low frequency. For the next step, frequency independent high precision resistors can be adopted and the system needs recalibrated for the best results. The second thing for the future work is the prototype can be further implemented on a PCB board to achieve less parasitic parameters and low noise. This may further increase the accuracy of the measurement.

Acknowledge

Designing a measurement device begins with a great deal of excitement. However, during the system design, chip ordering and implementation many unexpected issues need to be studied and solved. I got the initial idea and successfully completed this

project under the guidance of Mr. Bruce Land. It is only appropriate that I express my best appreciation to him here. During the design work, I obtained quite a few valuable advice and suggestion from Mr. Land. What's more, he provided me lots of electrical components which were not easy to get by myself. This greatly facilitated the design progress and finally helped me complete a good job.

Reference

[1] ATmega32 data sheet, Atmel Corporation 2002.
[2] AD5933 data sheet, Analog Devices 2005.
[3] AD5933 evaluation board data sheet, Analog Devices 2005.
[4] ADG706 data sheet, Analog Devices 2002.
[5] MXO45 data sheet, CTS Communications Components Inc. 2002.
[6] RCM2034R data sheet, NOHM
[7] AVR SKT500 User Guide, Atmel Corporation 2001
[8] Cornell ECE476 website, Bruce Land

Appendix

Words define

Since both the operation of I^2C bus and control of AD5933 have lots of command and status words, it is convenient for use by defining those hex numbers to more meaningful constants. Similarly, all the register addresses in AD5933 are defined in the same way.

```
/*******************************************
This program was written by Yi Zhang in the
design of a Complex Impedance Meter
email: yz226@cornell.edu

Project : A Design of Complex Impedance Meter
Version : 1.0

Chip type               : ATmega32
Program type            : Application
Clock frequency         : 16.000000 MHz
Memory model            : Small
External SRAM size      : 0
Data Stack size         : 512
*******************************************/

/* Use an 1x16 alphanumeric LCD connected
   to PORTA as follows:
```

```
   [LCD]      [Mega32 pin]
     1 GND-    GND
     2 +5V-    VCC
     3 VLC 10k trimpot wiper (trimpot ends go to +5 and gnd)
     4 RS -    PC0
     5 RD - PC1
     6 EN - PC2
    11 D4 - PC4
    12 D5 - PC5
    13 D6 - PC6
    14 D7 - PC7
*/

#include <mega32.h>
#include <math.h>
#include <Delay.h>
#include <stdio.h>

#asm
       .equ __lcd_port=0x1b
#endasm
#include <lcd.h> //LCD driver routines

//Reference Code range
#define CodeMax 25000
#define CodeMin 10000
#define CodeMid       21000


#define LCDwidth 16       //LCD characters


#define startfreq 5      //Start Frequency


#define incfreq 100    //Incremental Frequency


#define SUCCESS 0xff    //Flag

//Command
#define Start 0xa4
#define Stop 0x94
```

```
#define Trans 0x84
#define ACK 0xc4

//I2C bus Status
#define START 0x08
#define ReSTART 0x10
#define SLA_W 0x1a
#define SLA_R 0x1b
#define MT_SLA_ACK 0x18
#define MT_SLA_NACK 0x20
#define MR_SLA_ACK 0x40
#define MR_SLA_NACK 0x48
#define MT_DATA_ACK 0x28
#define MT_DATA_NACK 0x30
#define MR_DATA_ACK 0x50
#define MR_DATA_NACK 0x58
#define TWINT 0x80

//AD5933 control codes
#define Init 0x10//Initialize with start Freq
#define Sweep 0x20  //Start Frequency Sweep
#define IncFreq 0x30//Increment Frequency
#define RepFreq 0x40      //Repeat Frequency
#define MeaTemp 0x90    //Measure Temperature
#define PowerDown 0xa0//Power down mode
#define Standby 0xb0       //Standby mode
#define Range2V 0x00     //Output Voltage range 2V
#define Range1V 0x06     //Output Voltage range 1V
#define Range400mV 0x04 //Output Voltage range 400mV
#define Range200mV 0x02 //Output Voltage range 200mV
#define gainx5 0x00  //PGA gain x5
#define gainx1 0x01  //PGA gain x1

//AD5933 Register addresses
#define Control_high 0x80
#define Control_low 0x81
#define Freq_high 0x82
#define Freq_mid 0x83
#define Freq_low 0x84
#define FreqInc_high 0x85
#define FreqInc_mid 0x86
#define FreqInc_low 0x87
#define NumInc_high 0x88
#define NumInc_low 0x89
```

```
#define NumSettle_high 0x8a
#define NumSettle_low 0x8b
#define Status 0x8f
#define Temp_high 0x92
#define Temp_low 0x93
#define Real_high 0x94
#define Real_low 0x95
#define Imag_high 0x96
#define Imag_low 0x97



/*********************************************************************
********
    Function : char Init_TWI(void)
    Setup the TWI module
    Baudrate : 250kHz @ 16MHz system clock
    Own address : OWN_ADR (Defined in TWI_driver.h)
*********************************************************************
*******/
unsigned char Init_TWI(void)
{
    //TWAR = OWN_ADR;  //Set own slave address
    TWBR = 0x18;            //Set baud-rate to 250 KHz at 16 MHz xtal
    TWCR = 0x04;        //Enable TWI-interface
    return 1;
}

/*********************************************************************
********
    Function : void Wait_TWI_int(void)
    Loop until TWI interrupt flag is set
*********************************************************************
*******/
void Wait_TWI_int(void)
{
    while(!(TWCR & TWINT));
}

/*********************************************************************
********
    Function :unsigned char   Send_start(void)
    Send a START condition to the bus and wait for the TWINT to be set and
    see the result. If it failed, return the TWSR value, if succeeded, return
```

```
    SUCCESS.
*********************************************************************
*******/
unsigned char     Send_start(void)
{
    TWCR=Start;                 //Send START

    Wait_TWI_int();       //Wait for TWI interrupt flag to be set

     if((TWSR & 0xF8)!=START || (TWSR & 0xF8)!=ReSTART)
         return TWSR;           //If it failed, return the TWSR value
     return SUCCESS;     //If succeeded, return SUCCESS
}

/*********************************************************************
********
    Function : void Send_stop(void)
    Send a STOP condition to the bus
*********************************************************************
*******/
void Send_stop(void)
{
    TWCR = Stop;          //Send a STOP condition
}

/*********************************************************************
********
    Function : unsigned char Send_adr(unsigned char adr)

    Send a SLA+W/R to the bus
*********************************************************************
*******/
unsigned char Send_adr(unsigned char adr)
{
    Wait_TWI_int();       //Wait for TWI interrupt flag set

    TWDR = adr;
    TWCR = Trans;               //Clear int flag to send byte

    Wait_TWI_int();       //Wait for TWI interrupt flag set

    if((TWSR & 0xF8)!= MT_SLA_ACK || (TWSR & 0xF8)!= MR_SLA_ACK)
         return TWSR;           //If NACK received return TWSR
```

```
    return SUCCESS;          //Else return SUCCESS
}

/*****************************************************************
********
    Function : unsigned char Send_byte(unsigned char data)
    Send one byte to the bus.
*****************************************************************
*******/
unsigned char Send_byte(unsigned char data)
{
    Wait_TWI_int();      //Wait for TWI interrupt flag set

    TWDR = data;
    TWCR = Trans;            //Clear int flag to send byte

    Wait_TWI_int();      //Wait for TWI interrupt flag set

    if((TWSR & 0xF8)!= MT_DATA_ACK)
        return TWSR;         //If NACK received return TWSR

    return SUCCESS;          //Else return SUCCESS
}

/*****************************************************************
********
    Function : unsigned char Set_pointer(unsigned char reg_loc)
    Set the pointer to a register location.
*****************************************************************
*******/
unsigned char Set_pointer(unsigned char reg_loc)
{
    Send_start();
    Send_adr(SLA_W);
     Send_byte(0xb0);          //Pointer command code '1011 0000'
     Send_byte(reg_loc); //a register location at which the pointer points
     return 1;
}

/*****************************************************************
********
    Function : unsigned char Byte_write(unsigned char reg_addr, unsigned char data)
    Write a byte to AD5933.
```

```
*************************************************************************
*******/
unsigned char Byte_write(unsigned char reg_addr, unsigned char data)
{
    Send_start();
    Send_adr(SLA_W);
     Send_byte(reg_addr);
    Send_byte(data);
    Send_stop();
    return 1;
}




/*************************************************************************
********
    Function : unsigned char Block_write(unsigned char reg_loc, unsigned char
byte_num, unsigned char* data_p)
    Write a block of data to AD5933.
*************************************************************************
*******/
unsigned char Block_write(unsigned char reg_loc, unsigned char byte_num, unsigned
char* data_p)
{
    unsigned char i;

    Set_pointer(reg_loc); //set the pointer location
     Send_start();                //write the data block
    Send_adr(SLA_W);
     Send_byte(0xa0);           //Block write command code '1010 0000'
     Send_byte(byte_num);    //Num of data to be sent

     for(i = 0;i < byte_num;i++)    //Send the data bytes
     {
         Send_byte(*(data_p+i));
     }
     Send_stop();
    return 1;
}

/*************************************************************************
********
    Function : unsigned char Byte_read(unsigned char reg_loc)
```

```
    Read a byte from AD5933.
*************************************************************************
*******/
unsigned char Byte_read(unsigned char reg_loc)
{
    Set_pointer(reg_loc); //set the pointer location

    //Receive a byte
    Send_start();
    Send_adr(SLA_R);
    TWCR = Trans;
    Wait_TWI_int();          //Wait for TWI interrupt flag set
    return TWDR;
}

/*************************************************************************
********
    Function : unsigned char Block_read(unsigned char reg_loc, unsigned char
byte_num)
    Read a block of data from AD5933.
*************************************************************************
*******/
unsigned char Block_read(unsigned char reg_loc, unsigned char byte_num)
{
    unsigned char i;
    unsigned char* data_p;

    Set_pointer(reg_loc); //set the pointer location

     //write the data block
     Send_start();
    Send_adr(SLA_W);
    Send_byte(0xa1);         //Block read command code '1010 0001'
    Send_byte(byte_num);     //Num of data to be received

    Send_start();
    Send_adr(SLA_R);
    for(i = 0;i < byte_num;i++)     //Receive all the bytes
    {

        TWCR = ACK;         //Clear int flag and enable acknowledge to receive data.
        Wait_TWI_int();     //Wait for TWI interrupt flag set
        *(data_p+i)=TWDR;
    }
```

```
    TWCR = Trans;
    Wait_TWI_int();             //Wait for TWI interrupt flag set

    *(data_p+i)=TWDR;       //Save Last byte

    Send_stop();
    return data_p;
}

/*****************************************************************
********
    Function : void Hit_botton7()
    A function used in testing.
*****************************************************************
*******/
void Hit_botton7()
{
    unsigned char hit=1;
    while(hit)
    {
        if(PIND==0x7f)
        {
            delay_ms(30);
            if(PIND==0x7f)
            hit=0;
        }
    }
}

/*****************************************************************
********
    Function : unsigned long int Data_proc(unsigned char data_high, unsigned char
data_low)
    Calculate the code value from AD5933.
*****************************************************************
*******/
unsigned long int Data_proc(unsigned char data_high, unsigned char data_low)
{
    unsigned long int data;
    data=(unsigned long int)data_high*256+data_low;
    if(data > 0x7fff)
    {
        data=0x10000-data;
```

```
    }
    return data;
}

/*****************************************************************
********
    Function : void error(unsigned char mode)
    A function handles error situations.
*****************************************************************
*******/
void error(unsigned char mode)
{
    char lcd_buffer[17];        //LCD display buffer
    lcd_init(LCDwidth);             //initialize the display
     lcd_clear();                   //clear the display
     lcd_gotoxy(0,0);               //position to upper left on display

     switch (mode)
     {
     case 1:
          lcd_putsf("INF");
          lcd_gotoxy(0,1);          //position to bottom left on display
          lcd_putsf("Range:4.7K-1.1M");
          break;
     case 2:
          lcd_putsf("Error"); break;
     }
}

/*****************************************************************
********
    Function : void Display(unsigned long int frequency, unsigned long int
impedance)
    A function used to display the measured results.
*****************************************************************
*******/
void Display(unsigned long int frequency, unsigned long int impedance)
{
    char lcd_buffer[17];        // LCD display buffer
    lcd_init(LCDwidth);             //initialize the display
     lcd_clear();                   //clear the display
     lcd_gotoxy(0,0);               //position to upper left on display
     lcd_putsf("Freq=");            //constant string from flash
     lcd_gotoxy(0,1);               //position to bottom left on display
```

```
        lcd_putsf("Imped=");            //constant string from flash

        //display present freq
        sprintf(lcd_buffer,"%-li",frequency);
        lcd_gotoxy(5,0);
        lcd_puts(lcd_buffer);

        //display measured impedance at present freq
        sprintf(lcd_buffer,"%-li",impedance);
        lcd_gotoxy(6,1);
        lcd_puts(lcd_buffer);
}



void main(void)
{
      unsigned char i,a,b,cnt,f;
      unsigned long int R,I;       //Real and Imag numbers
      unsigned char* data_s, data_r;
      unsigned long int freqcode,frequency,impedance;
      //Refence Impedance values used in the system
      unsigned long int
ReImpedance[]={4700,10000,27000,57000,200000,470000,1000000};
      //Analog switches control code
      unsigned char init_port[]={0x0f,0x1f,0x2f,0x3f,0x4f,0x5f,0x6f};
      unsigned char port[]={0x01,0x13,0x25,0x37,0x49,0x5b,0x6d};
      float GF,code,Calcode,tempcode;

// Port B initialization
PORTB=0x00;
DDRB=0x00;

// Port C initialization
PORTC=0xff;
DDRC=0x00;

// Port D initialization
PORTD=0x00;
DDRD=0x00;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: Timer 0 Stopped
```

```
// Mode: Normal top=FFh
// OC0 output: Disconnected
TCCR0=0x00;
TCNT0=0x00;
OCR0=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: Timer 1 Stopped
// Mode: Normal top=FFFFh
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer 2 Stopped
// Mode: Normal top=FFh
// OC2 output: Disconnected
ASSR=0x00;
TCCR2=0x00;
TCNT2=0x00;
OCR2=0x00;

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
// INT2: Off
MCUCR=0x00;
MCUCSR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x00;

// Analog Comparator initialization
```

```
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer/Counter 1: Off
// Analog Comparator Output: Off
ACSR=0x80;
SFIOR=0x00;

// 2 Wire Bus initialization
// Generate Acknowledge Pulse: On
// 2 Wire Bus Slave Address: 0dh
// General Call Recognition: Off
// Bit Rate: 250.000 kHz

DDRB=0xff;
Init_TWI();

while(1)
      {
                //set start frequency
                freqcode=startfreq*33.554432;           //calculate the hex frequency
code
                *data_s=0x000000ff & (freqcode>>16); *(data_s+1)=0x000000ff &
(freqcode>>8); *(data_s+2)=0x000000ff & freqcode;
                Block_write(Freq_high, 3, data_s);


                    //set Incremental frequency
                freqcode=incfreq*33.554432;       //calculate the hex inc frequency code
                *data_s=0x000000ff & (freqcode>>16); *(data_s+1)=0x000000ff &
(freqcode>>8); *(data_s+2)=0x000000ff & freqcode;
                Block_write(FreqInc_high, 3, data_s);

                  //Set Num of Inc
                  Byte_write(NumInc_high, 0x01);
                  Byte_write(NumInc_low, 0xf4);

                  //Set Num of Settling Time Cycles
                  Byte_write(NumSettle_high, 0x00);
                  Byte_write(NumSettle_low, 0x24);

                  //Working mode
                     Byte_write(Control_low, 0x00);
                  Byte_write(Control_high, 0xb0);

restart:   code=0;
```

```
        cnt=1;
          f=0;

        while (code<CodeMin || code>CodeMax)
        {
                if(cnt==1)                              //Estimate the External
Impedance
                {
                    i=3;                                //The channel for External
Impedance
                    PORTB=init_port[i];
                }

                else
                {
                    if(i>6 || i<0)        //External Impedance out of measurement
range
                    {
                        error(1);
                        goto restart;
                    }

                    if(code>CodeMax)
                        PORTB=init_port[++i];
                    else
                            PORTB=init_port[--i];
                  }
              //Initialize
                Byte_write(Control_high, 0x10);

                //Start measurement
                if(cnt==1)
                  Byte_write(Control_high, 0x26); //nominal output 1V, gain of
PGA=5

                else
                  Byte_write(Control_high, 0x46);

                while(!(Byte_read(Status) & 0x02));

                a=Byte_read(Real_high);
                b=Byte_read(Real_low);
                R=Data_proc(a, b);
```

```
  a=Byte_read(Imag_high);
  b=Byte_read(Imag_low);
  I=Data_proc(a, b);

  code=sqrt(R*R+I*I);
  cnt++;

  if(cnt>=20)
  {
     error(2);
     goto restart;
  }

}

  if(code>CodeMid)
  {
     tempcode=0;
     if(i<=5)
     {
              PORTB=init_port[i+1];
              Byte_write(Control_high, 0x46);
              while(!(Byte_read(Status) & 0x02));

              a=Byte_read(Real_high);
              b=Byte_read(Real_low);
              R=Data_proc(a, b);

              a=Byte_read(Imag_high);
              b=Byte_read(Imag_low);
              I=Data_proc(a, b);

              tempcode=sqrt(R*R+I*I);
     }

     if(tempcode<CodeMin)
     {
              PORTB=init_port[i];
              Byte_write(Control_high, 0x44);
              while(!(Byte_read(Status) & 0x02));

              a=Byte_read(Real_high);
              b=Byte_read(Real_low);
              R=Data_proc(a, b);
```

```
            a=Byte_read(Imag_high);
            b=Byte_read(Imag_low);
            I=Data_proc(a, b);

            code=0.5*(code+sqrt(R*R+I*I));
            f=1;
    }
    else
    {
            code=tempcode;
            i++;
    }
}

    PORTB=port[i];          //change channel to external impendance
    delay_ms(1);

    Byte_write(Control_high, 0x46);
    while(!(Byte_read(Status) & 0x02));

    a=Byte_read(Real_high);
    b=Byte_read(Real_low);
    R=Data_proc(a, b);

    a=Byte_read(Imag_high);
    b=Byte_read(Imag_low);
    I=Data_proc(a, b);

    Calcode=sqrt(R*R+I*I);

    if(f==1)
    {
        Byte_write(Control_high, 0x44);
        while(!(Byte_read(Status) & 0x02));
        a=Byte_read(Real_high);
        b=Byte_read(Real_low);
        R=Data_proc(a, b);

        a=Byte_read(Imag_high);
        b=Byte_read(Imag_low);
        I=Data_proc(a, b);

        Calcode=0.5*(Calcode+sqrt(R*R+I*I));
```

```
            }

                    impedance=(unsigned long int)(ReImpedance[i]*code/Calcode);
                       Display(startfreq,impedance);
            }
}
```