Development of a Novel Wearable Posture Correction Apparatus Using Advanced Accelerometry Techniques: Signal Analysis Unit

Nanxi Zha

Electrical and Biomedical Engineering
Faculty Advisor: Prof. de Bruin

Electrical and Biomedical Engineering Project Report
Submitted in partial fulfillment of the degree of
Bachelor of Engineering

McMaster University
Hamilton, Ontario, Canada
April 23, 2010

ABSTRACT

Proper posture is an integral part of a healthy lifestyle. For individuals suffering from unilateral body neglect, proper posture cannot always be guaranteed. The wearable posture correction apparatus uses measurements from accelerometers. Based on gravitational acceleration, calculations are performed to measure posture tilt. This action is modeled on a software program. Improper posture is also noted, and corrective instructions are given. The theory behind our device, hardware design, the experimental results, and correctness of the system are presented.

Keywords: Posture correction, accelerometer, tilt, unilateral body neglect

ACKNOWLEDGEMENTS

The author would like to thank David Zhitomirsky of McMaster University, who worked on the signal acquisition component of this design project. Without his contribution, this project would not have been possible. In addition, the author would also like to thank Prof. deBruin of McMaster University for his invaluable advice during the design project. Lastly, the author would like to thank Larissa Schudlo for providing hardware advice to the project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Development of a Novel Wearable Posture Correction Apparatus Using Advanced Accelerometry Techniques: Signal Analysis Unit

1. INTRODUCTION

Biomedical engineering involves using conventional engineering practices to improve healthcare-related problems. In this project, the overall goal is to design a posture correction system for physiotherapy and rehabilitation purposes. Upon sensing the patient's posture, the device will display the information graphically, and offer corrective feedback.

1.1 Objectives

The signal analysis portion of the project includes a hardware objective and a software objective. In the hardware component, the signal generated by the accelerometer is processed by the microcontroller, and then transmitted wirelessly to a second microcontroller, which is then connected to a computer. The software component consists of utilizing this data and displaying a graphically model of the patient's movement on a computer, as well as calculating any posture changes the patient needs to make.

1.2 Methodology

It was determined that accelerometers would be used to implement a posture sensing algorithm. Through testing procedures, it was determined that one accelerometer would be adequate in sensing the thoracic posture of the patient. The methodology begins with the accelerometer (ADXL335) sends 3-axis acceleration signals to a microcontroller (Adruino ATmega 328 on an Adruino Duemilanove USB board), hereby known as Microcontroller #1. The data is then processed on Microcontroller #1 to determine the tilt angles with respect to the x, y, and z – axis. Afterwards, the data is then sent from Microcontroller #1 to a second microcontroller (Adruino ATmega 328 on an Adruino Duemilanove USB board), hereby known as Microcontroller #2. The link between Microcontroller #1 and Microcontroller #2 is wireless. This is accomplished by using two nRF24L01 tranceiver chips. After the data arrives at Microcontroller #2, it is then sent to a computer, via a USB connection. The data is then used to plot the patient’s position on a screen (in LabVIEW), as well as offering posture correction feedback.

1.3 Scope

Since the signal analysis unit must be connected with the signal acquirement unit seamlessly, a scope for this portion must be defined. It is determined that the signal analysis unit will include data hand-off from Microcontroller #1 to the LabVIEW program on the computer.

2. LITERATURE REVIEW

Patients who are suffering from unilateral body neglect (e.g., as a result of a stroke) have a decreased awareness of a particular side of their body. Although they have trouble maintaining proper posture, corrections to their postures are possible when given encouragement (Punt, Riddoch 2006).

Currently, there are posture monitoring systems available involving accelerometers and gyroscopes (Hyde et al. 2008). This project will focus on the use of accelerometers due to their small size, light weight, and low cost (Mizuike, Ohgi & Morita 2009). Past accelerometer studies have been applied to posture maintenance and gait analysis. A series of accelerometers attached at the joints can model movements of the entire human body (Giansanti et al. 2003). By using frequency filtering and analysis techniques (e.g., Fourier Transform), the signals can be classified based on different posture (DC signals) and movement (AC signals) (Fahrenberg et al. 1997). Furthermore, incorrect gait patterns can be detected by accelerometers using normalized root mean square and auto correlation analysis (Mizuike, Ohgi & Morita 2009).

In comparison, the posture correction system designed will focus solely on accelerometers. Posture maintenance will be reinforced through the usage of vibration motors, LED lights, as well as a graphical modeling tool. Analysis performed on the acceleration signals will be based on tilt calculations. This allows for much simpler calculations, which can save time and memory, while ensuring accuracy of the system. By using tilt angles to model posture, it also allows for a very simple posture correction algorithm to be implemented. Its simplicity also saves time and memory resources, allowing all this to be performed on Microcontroller #1.

Through this comparison, it is believed that this posture correction system is a novel design in the rehabilitation and physiotherapy sciences. The resources used to implement this system are inexpensive but efficient. It allows for a portable system that can be applied to rehabilitation and physiotherapy.

3. PROBLEM AND METHODOLOGY OF SOLUTION

In this project, there existed three main problems: a tilt calculation algorithm need to be designed, as well as a wireless software library need to be implemented for data transmission and receiving. These problems are the backbones for the project, and are all essential to its success.

3.1 Tilt Calculation

From the application notes of the ADXL 335 (Tuck 2007) it was determined that the angle of tilt can be found with the following equation, using one axial acceleration data:

$$\theta = sin^{-1}\left(\frac{V_x - V_{offset}}{Sensitivity}\right) \dots\dots\dots\dots (3.1)$$

In this equation, V_x is the voltage measured from the accelerometer in the x-axis direction, V_{offset} is the voltage offset recorded when the x-axis is perpendicular to the gravitational acceleration, the sensitivity is taken from the ADXL 335 datasheet, and θ is the angle of tilt as measured from the x-axis. Similarly, the tilt of the y and z-axis can be found using equation (3.1).

Unfortunately, the ADXL 335 has a limited region of linearity. To compensate for this problem, all tilt angles were calculated based on two axial acceleration data. The equations are as follows (Tuck 2007):

$$\phi = tan^{-1}\left(\frac{\sqrt{a_x^2 + a_y^2}}{a_z}\right) \dots\dots\dots\dots (3.2)$$

$$\rho = tan^{-1}\left(\frac{a_x}{\sqrt{a_z^2 + a_y^2}}\right) \dots\dots\dots\dots (3.3)$$

$$\theta = tan^{-1}\left(\frac{a_{yx}}{\sqrt{a_z^2 + a_x^2}}\right) \dots\dots\dots\dots (3.4)$$

The angles ϕ, ρ, and, θ correspond to the angles with respect to the x, y, and z-axis respectively.

3.2 Wireless Tranceiver

To implement a wireless connection between the two microcontrollers, controls must be implemented between the microcontrollers and the transceivers. The signals from the microcontroller are in integer form, and this must be translated into bytes in order for transmission to occur. This can be accomplished by typecasting an integer into its corresponding 1-dimensional byte array. Unfortunately, the same process could not be used when three integers were required to be transmitted (this is necessary as the accelerometer produces three integer acceleration values, which pertains to the x, y, and z-axis). If the data were to be sent one by one, the first integer can be received, but the two subsequent numbers are lost. Therefore, the three integers must be combined into one 1-dimensional byte array, which is then sent.

In order to transmit three integers as one 1-dimensional byte array, a byte array with the size of three integers must first be initialized. The three axial acceleration data are stored as integers, and then converted into three 1D byte arrays. Since the input of the data sending method is one 1D byte array, the three 1D byte arrays must be compressed into one 1D byte array. The three numbers are then stored into the array.

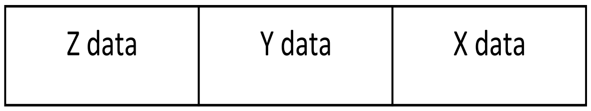


Figure 3.1: Data storage in 1-dimensional byte array

From Figure 3.1, it can be seen that the three integers are stored into one byte array. The right cell represents the first index of the array, which stores data in the x-axis direction. The middle cell refers to the second index of the array, which stores data in the y-axis direction. Lastly, the left cell contains the last index of the array, which stores data in the z-axis direction. The final result is a single 1-dimensional byte array, which is the compatible input for the Adruino-Nordic library’s send data function.

At the receiving terminal, a 1-dimensional byte array is then received by the nRF24L01 transceiver. Processing is done to extract the three numbers from the byte array. Since the x direction data occupies the lowest indices of the byte array, each element was multiplied by 256^n, where n is the index of the array (zero being the lowest). This concept was then applied to the y and z direction data, and the respective numbers were obtained.

4. DESIGN & EXPERIMENTATION PROCEDURES

The design procedures occurred in four steps. First, the accelerometer was connected to a simple LabVIEW software, where initial filtration techniques were tested. Next, the accelerometer was put through a series of planned trajectories, which were then compared to the measured trajectories. After this, real-time analysis of the accelerometer was computed. Lastly, a wireless link was established between the data acquisition unit and the data analysis unit.

4.1 Initial Testing

The initial testing design schematic can be seen on Figure 4.1. The "Vcc" pin of the ADXL 335 is connected to the supply voltage, which is 3.3V. The "GND" pin is connected to the ground. The "X", "Y", and "Z" connections at the laptop is a serial connection.

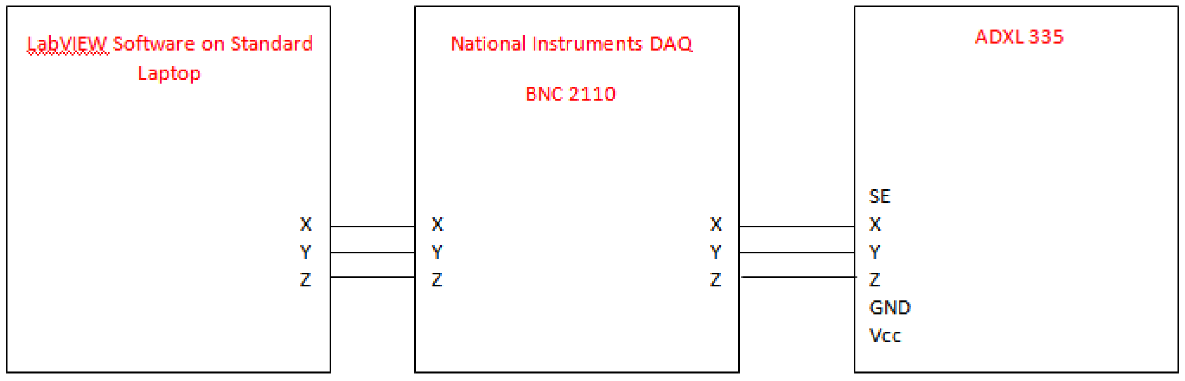


Figure 4.1: Initial testing setup

This set up allowed for simple trajectories to be performed with the accelerometer, and the results were recorded in LabVIEW. The results were then saved in a text file from LabVIEW, and then passed into MATLAB for filtration purposes.

Using one such trajectory, as seen in Figure 4.2, various filtration techniques were performed.

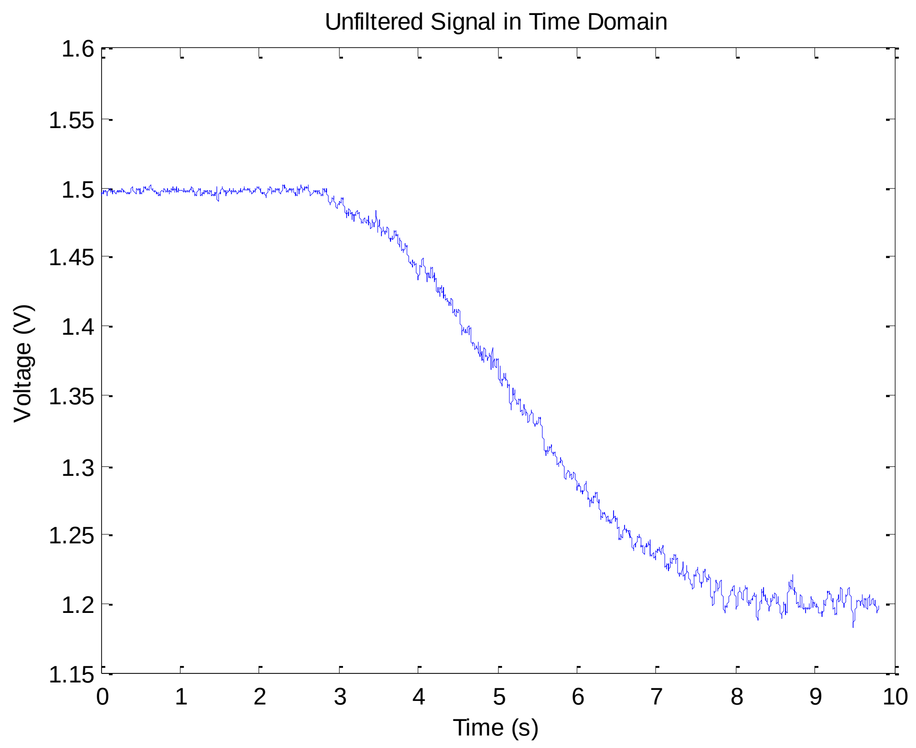


Figure 4.2: Initial accelerometer trajectory

First, a digital low pass filter was implemented in MATLAB, in order to minimize the noise that occurred in the higher frequencies. This resulted in the graph shown in Figure 4.3. It can be seen that the minimal changes resulted when the low pass filter was implemented. Furthermore, extra error was added to the plot near the beginning of the trajectory. This was due to the internal error of the filter function in MATLAB. Therefore, it was decided to forego the digital low pass filter, as the processing time costs outweighed the benefits.

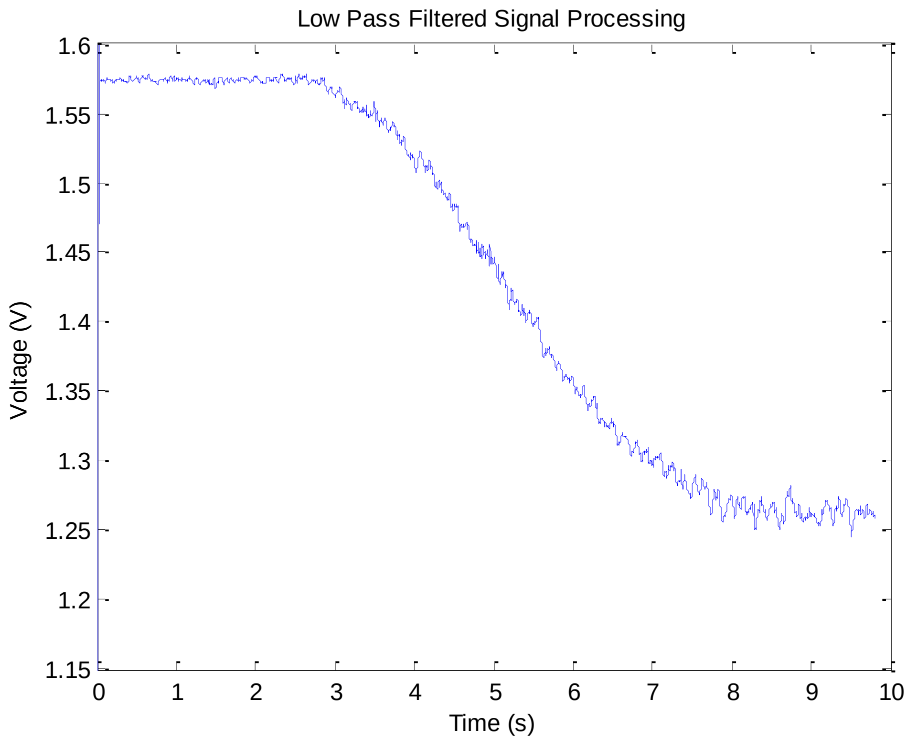


Figure 4.3: Low-pass filtered trajectory

Next, a moving average filter was implemented. In this case, ten samples of the accelerometer's data were taken, and the average value was determined. Based on the results in Figure 4.4, it can be noted that the moving average filter did not smooth out the results as much as expected. Therefore, the improvements seen in the moving average filter were deemed to be not worth the extra cost of processing time.

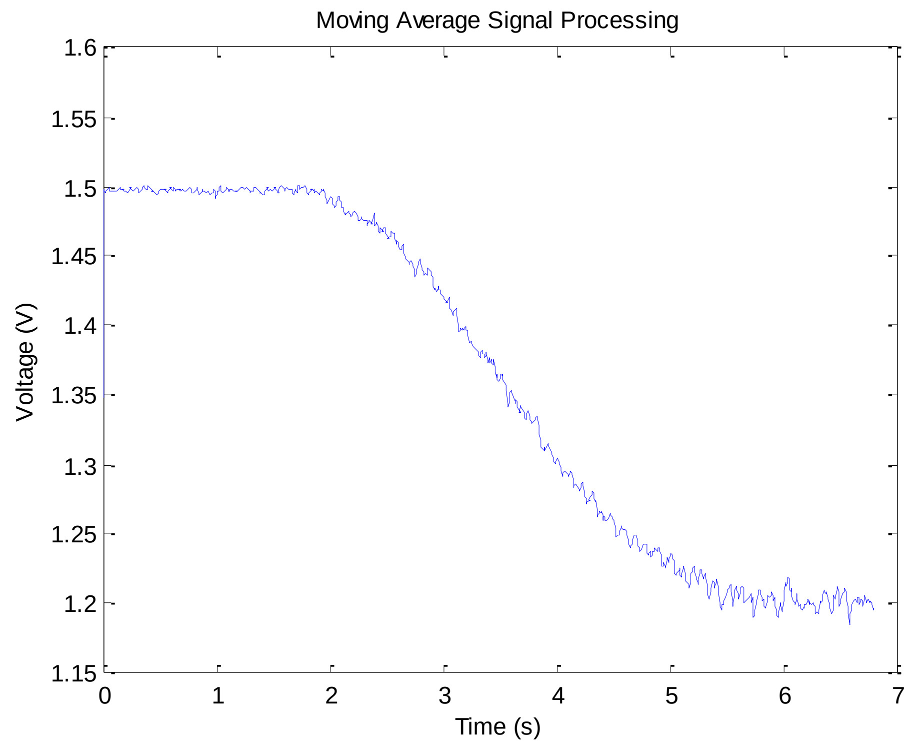


Figure 4.4: Moving average filtered trajectory

As seen in the two results, it was determined that the improved outcomes due to digital filtration were not worth the extra processing time required.

4.2 Trajectory Calculation

For the trajectory calculation, the accelerometer was first moved horizontally and then vertically along a horizontal plane, and the resulting accelerations were then recorded.

A computer-calculated model of the accelerometer's movements was first calculated based on the equations of motion:

$$v_f = a * \Delta t + v_i \ldots\ldots\ldots\ldots (4.1)$$

$$d = v_i + \frac{1}{2}a\Delta t^2 \text{ ………… } (4.2)$$

Where v_f is the final velocity, v_i is the initial velocity, a is the acceleration, Δt is the time interval, and d is the distance travelled.

For each time interval, the final velocity calculated becomes the initial velocity of the next time interval. The initial velocity at the start is assumed to be at zero. The distance travelled, as calculated from equation 4.2, is then plotted in MATLAB, as seen in Figure 4.5.

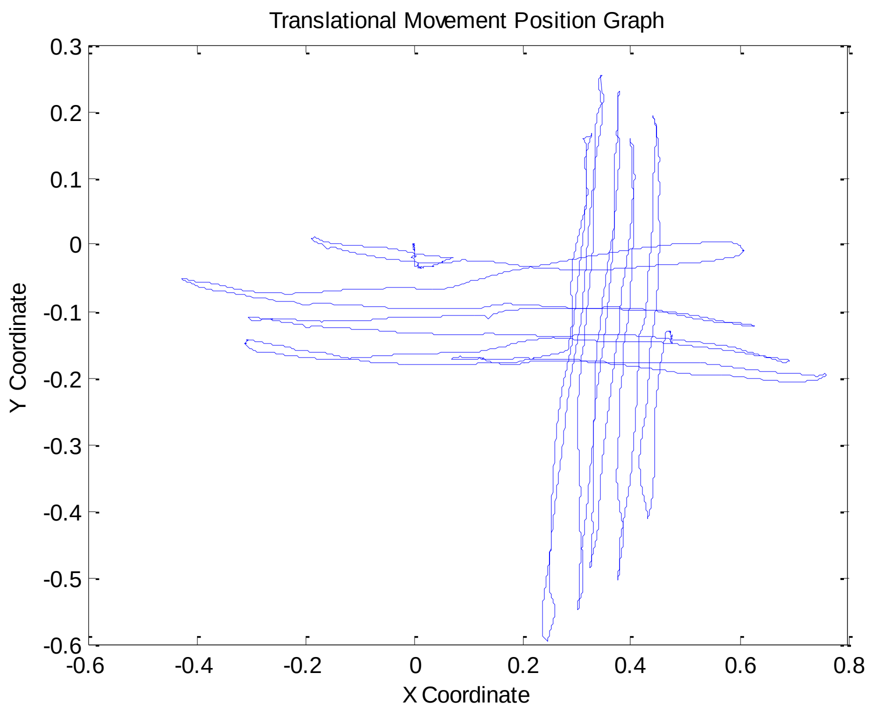


Figure 4.5: Translational movement of the accelerometer

Since this trajectory matched the actual trajectory of the accelerometer, it was concluded that the accelerometer is functioning correctly, and can be used to determine the patient’s positions in posture control.

After this, the accelerometer was spun around a wheel apparatus for tilt angle testing. The tilt algorithm was tested, using equation 3.1.

Figure 4.6 shows the tilt trajectory, as plotted in MATLAB. This trajectory matched the actual trajectory of the accelerometer.

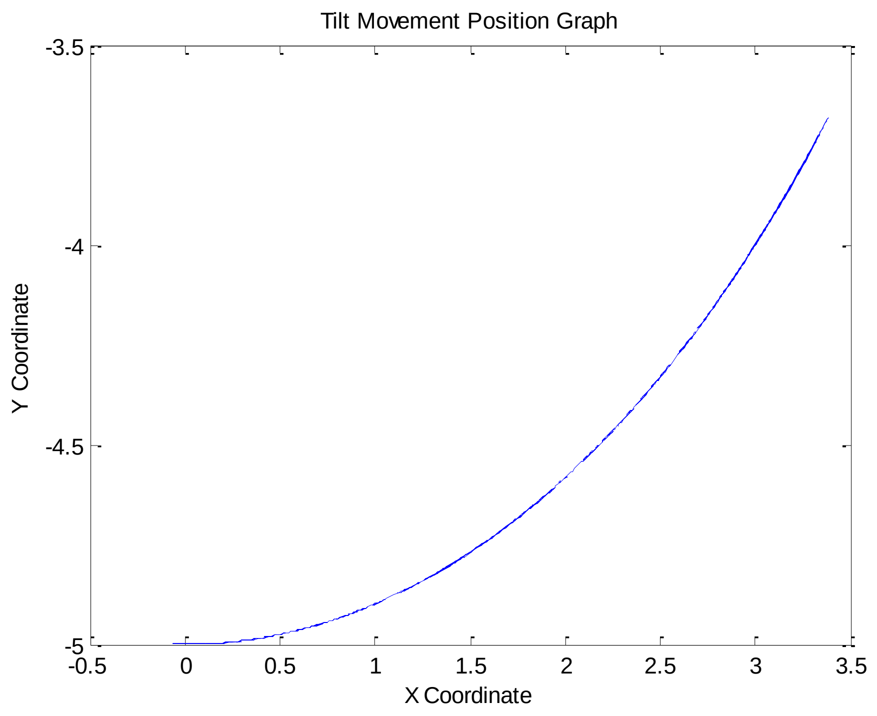


Figure 4.6: Tilt trajectory

The programming code for this tilt algorithm can be found in Appendix A.

4.3 Real Time Implementation

To implement real time analysis of data, the LabVIEW program needed to be modified such that all calculations would take place here, as opposed to in MATLAB. This was done with the usage of a "formula node" in LabVIEW. The two-axis acceleration equations (3.2 to 3.4) were also implemented in this step.

In order to make this project more portable, the National Instruments DAQ was replaced with an Adruino Dueomilanove USB microcontroller. This board was chosen for its fast processing power, onboard analog to digital converters and multiple analog read-in and digital read-out pins. It can be powered via a USB connection, or a 9V battery source, which makes the project portable. Figure 4.7 shows the new schematic for the project, where the pins on the right of the Adruino are analog read-in pins, and the pins on the left are digital write-out pins.

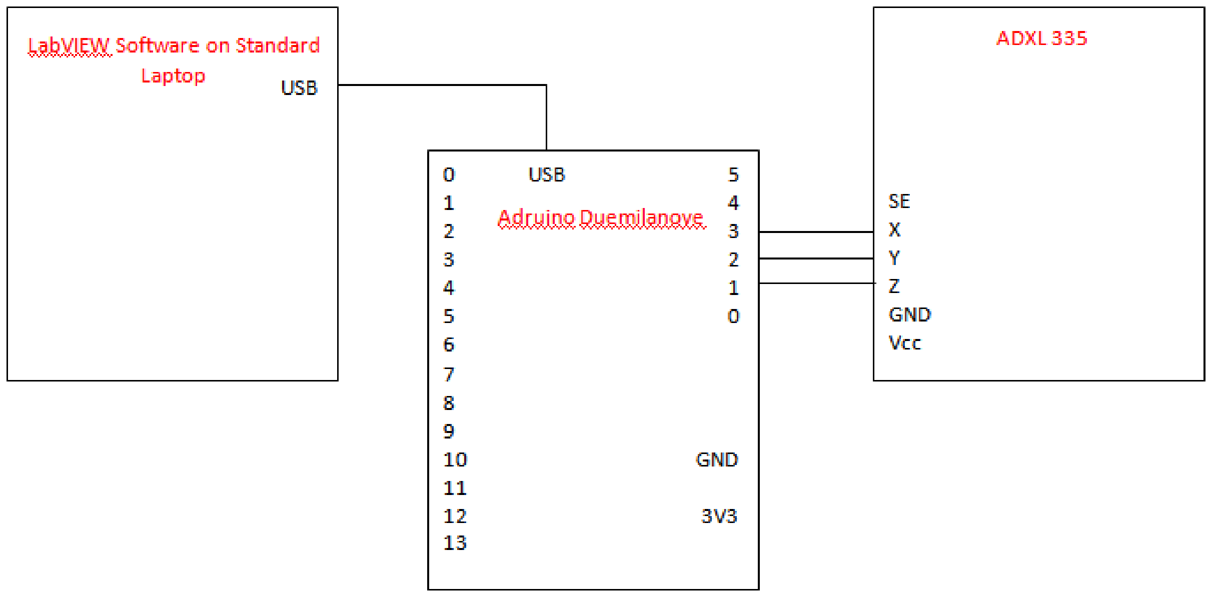


Figure 4.7: Project schematic

4.4 Wireless Implementation

The wireless module is based on the nRF24L01 transceiver. This chip is capable of acting both as the transmitter and the receiver.

The nRF24L01 transceiver was chosen especially for its low input voltage requirements (3.3V). An on-chip voltage regulator reduces all input voltages to a maximum of 3.3V. This was helpful, especially since the output pins of the Adruino Duemilanove are driven high to 5V.

Figure 4.8 shows the transmitter module's schematic, while figure 4.9 shows the receiver module.

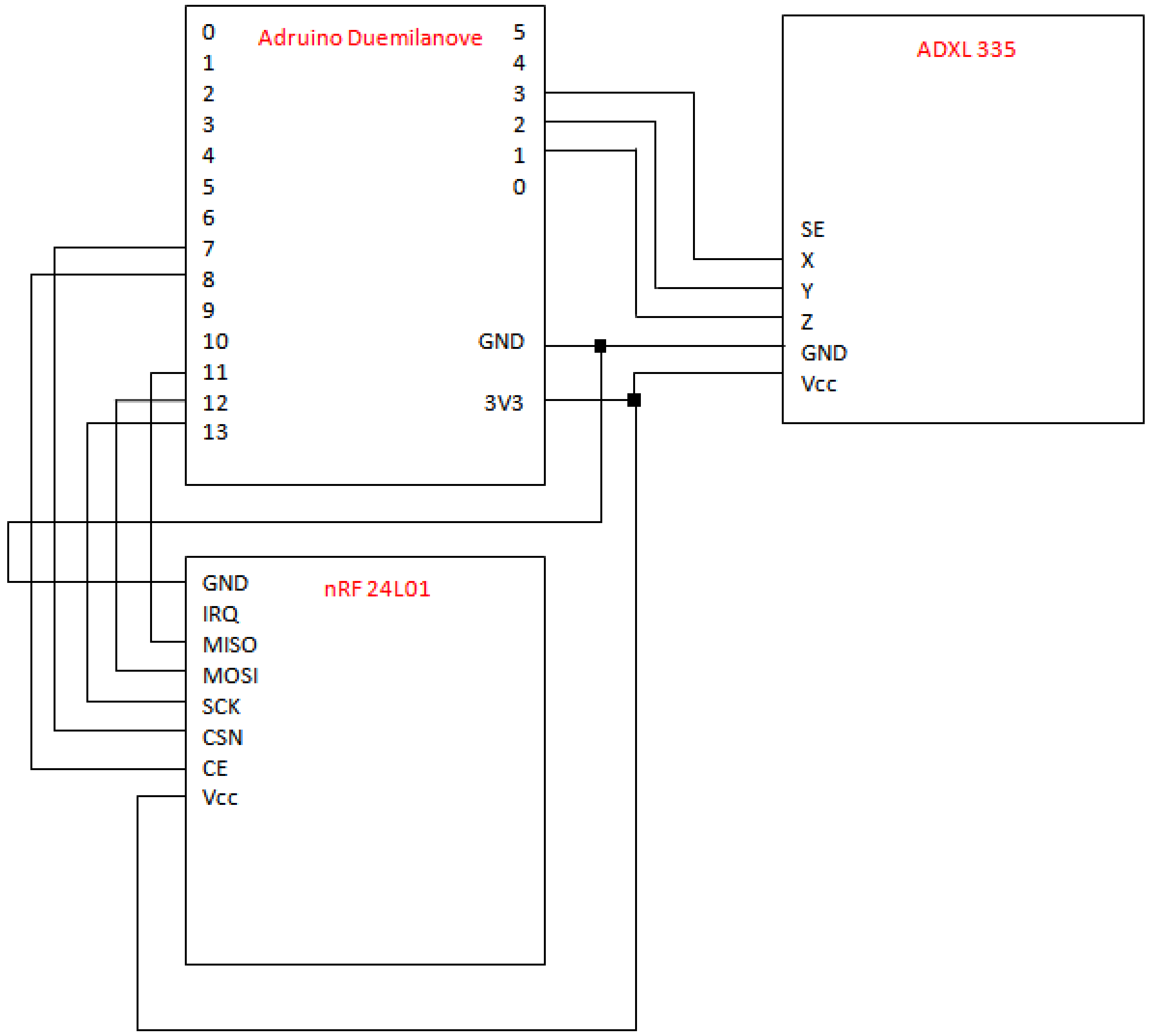


Figure 4.8: Transmitter module

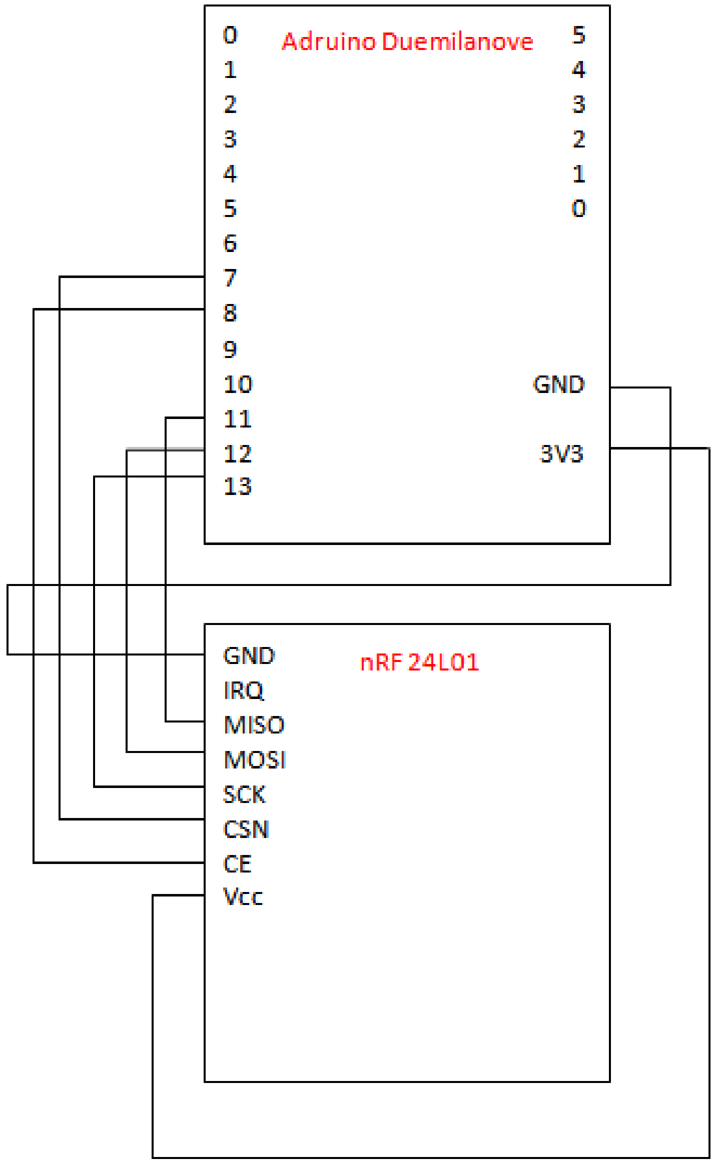


Figure 4.9: Receiver module

The power supply for the transceiver can be taken directly from the 3.3V supply source of the Adruino. The pins, MISO and MOSI are for output and input of data respectively. CE is the chip enable, which enables transmit or receive mode. CSN is the chip select pin. SCK is the digital clock for the SPI.

An Adruino-Nordic library was found specific for the nRF24L01 transceiver. The library works in conjunction with the SPI (Serial Peripheral Interface) library. The details of these functions are not explored; however, several key methods (that are essential to the project) are mentioned below.

void setRADDR(byte *addr) is responsible for setting the receiving address. The address mentioned in the transmitter module and the receiver module must match.
void setTADDR(byte *addr) is used to set the transmission address. The address mentioned in the transmitter module and the receiver module must match.
bool dataReady() determines if there is data ready to be received.
void getData(byte *data) gets the data that is sent.
void send(byte *data) sends the data.
bool isSending() determines if there is data that is currently being sent.

Initially, a test program was run to send one integer number over the wireless network (refer to Appendix B).

5. RESULTS AND DISCUSSIONS

The following table shows the angle calculation performed using equations 3.2 to 3.4, versus actual angles:

Table 5.1: Percent Error for Left/Right Tilt

Theoretical Angle (degrees)	**Actual Angle (degrees)**	**% Error**
-90	-89.04	1.06667
-60	-56.76	5.4
-30	-26.96	10.1333
0	4.27	N/A*
30	28.95	3.5
60	66.03	10.05
90	86.98	3.35556

*N/A due to division by the theoretical value, which is zero.

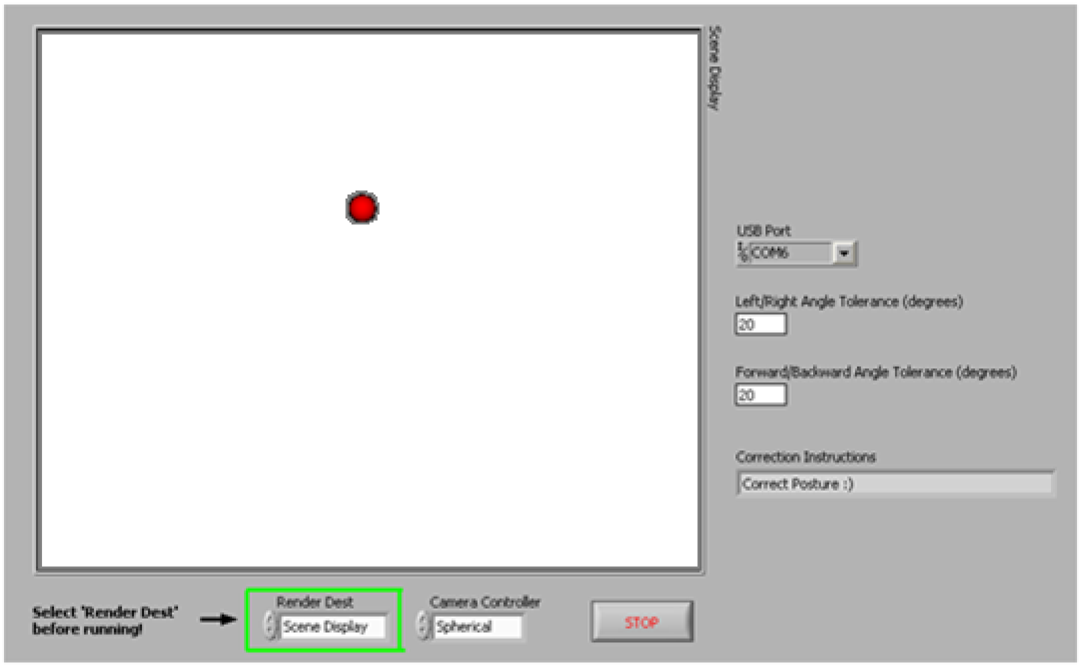
Table 5.2: Percent Error for Forward/Backward Tilt

Theoretical Angle (degrees)	**Actual Angle (degrees)**	**% Error**
-90	-86.72	3.64444
-60	-58.91	1.81667
-30	-27.54	8.2
0	1.787	N/A*
30	30.57	1.9
60	62.19	3.65
90	85.25	5.27778

*N/A due to division by the theoretical value, which is zero.

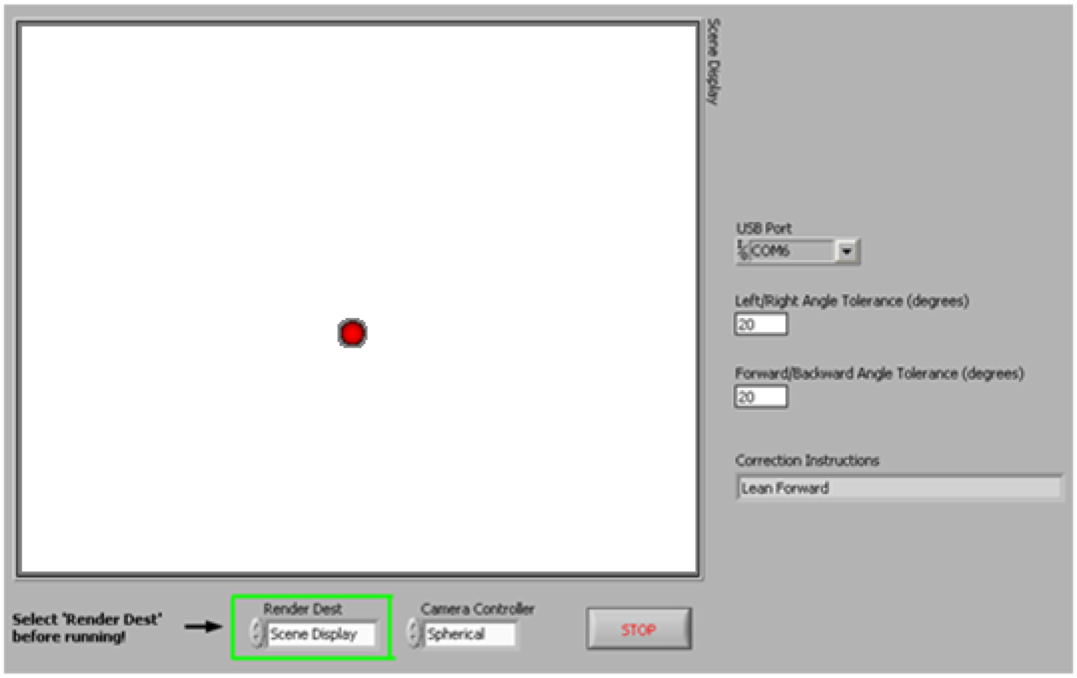
It can be seen that the angle calculations are fairly accurate. The biggest % error is 10.1333% (probably due to the nonlinearity behaviour of the accelerometer at boundary cases), with the majority of the % errors under 5%. Therefore, the angle calculation equations are deemed to be accurate for this tilt application. Better accuracy can be obtained by using a look-up table with voltages and their corresponding angles.

The following series of screen captures illustrates the graphical animation program used to model the movement of the accelerometer:



USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Correct Posture :)
Select 'Render Dest' before running!
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.1: Accelerometer at the home position (correct posture)



Left/Right Angle Tolerance (degrees): 20

Forward/Backward Angle Tolerance (degrees): 20

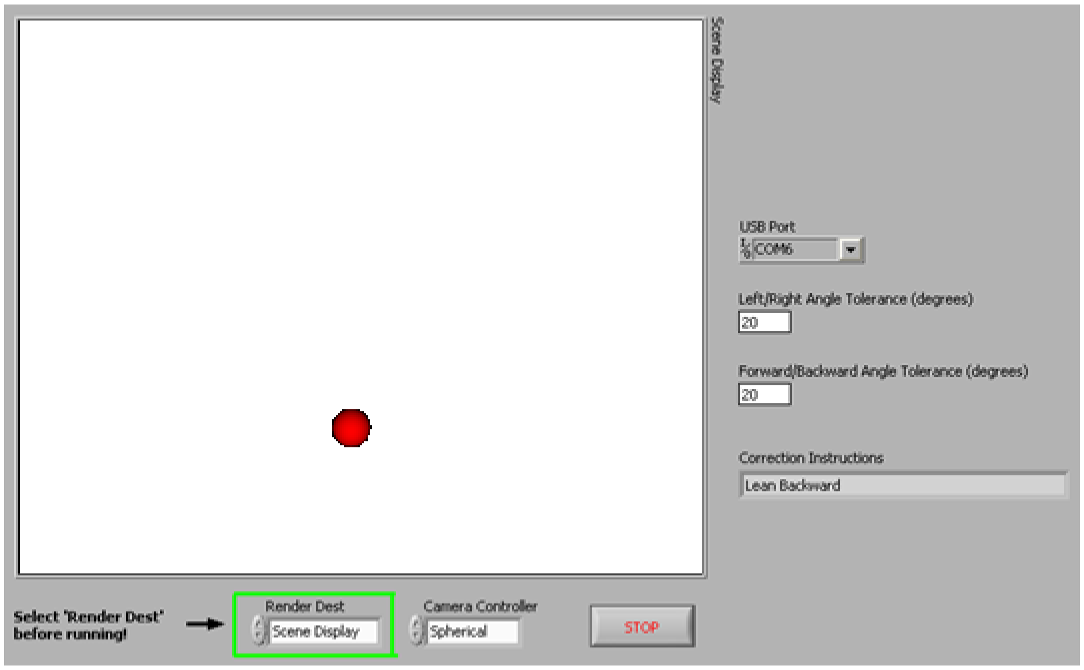
Correction Instructions
Lean Forward

Select 'Render Dest' before running!

Render Dest: Scene Display

Camera Controller: Spherical

Figure 5.2: Accelerometer tilted backwards



Left/Right Angle Tolerance (degrees): 20

Forward/Backward Angle Tolerance (degrees): 20

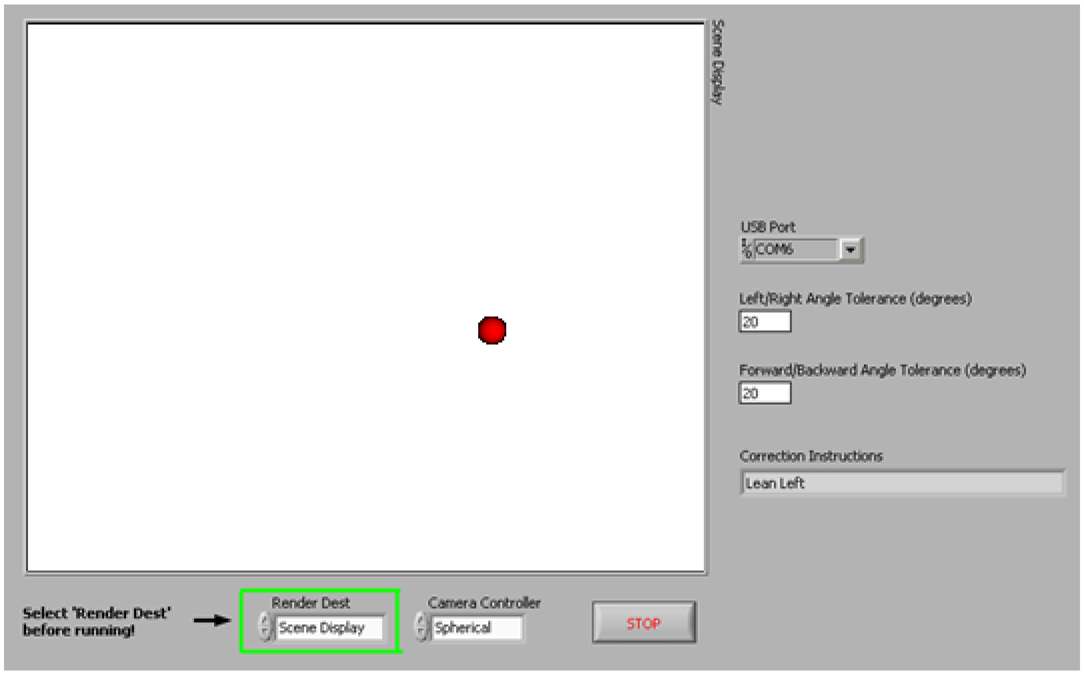
Correction Instructions
Lean Backward

Select 'Render Dest' before running!

Render Dest: Scene Display

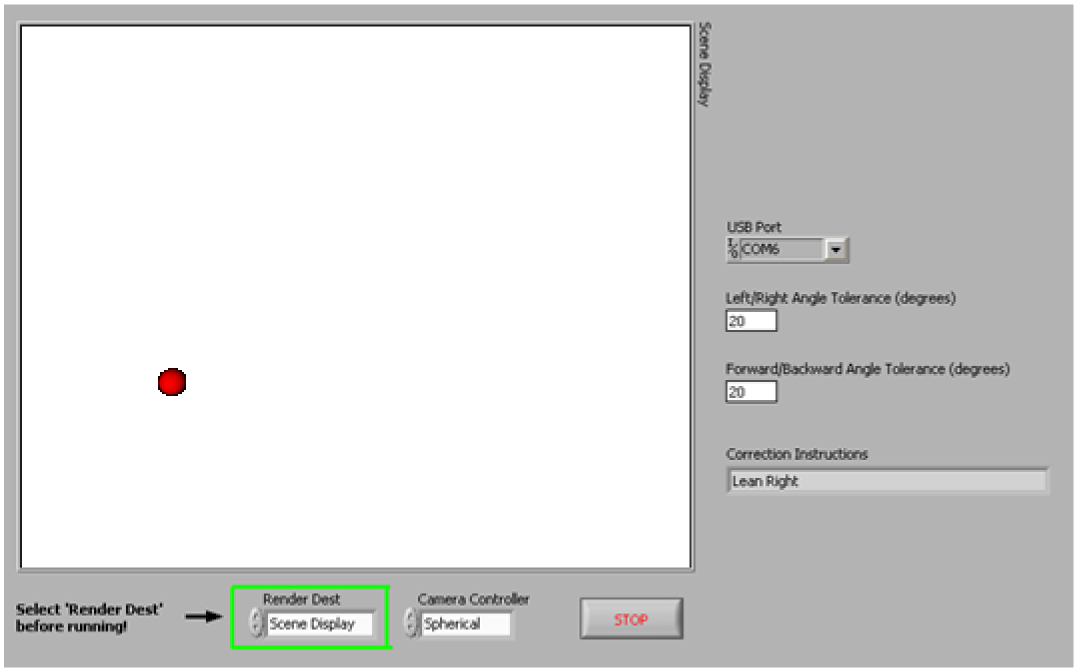
Camera Controller: Spherical

Figure 5.3: Figure 5.2: Accelerometer tilted forwards



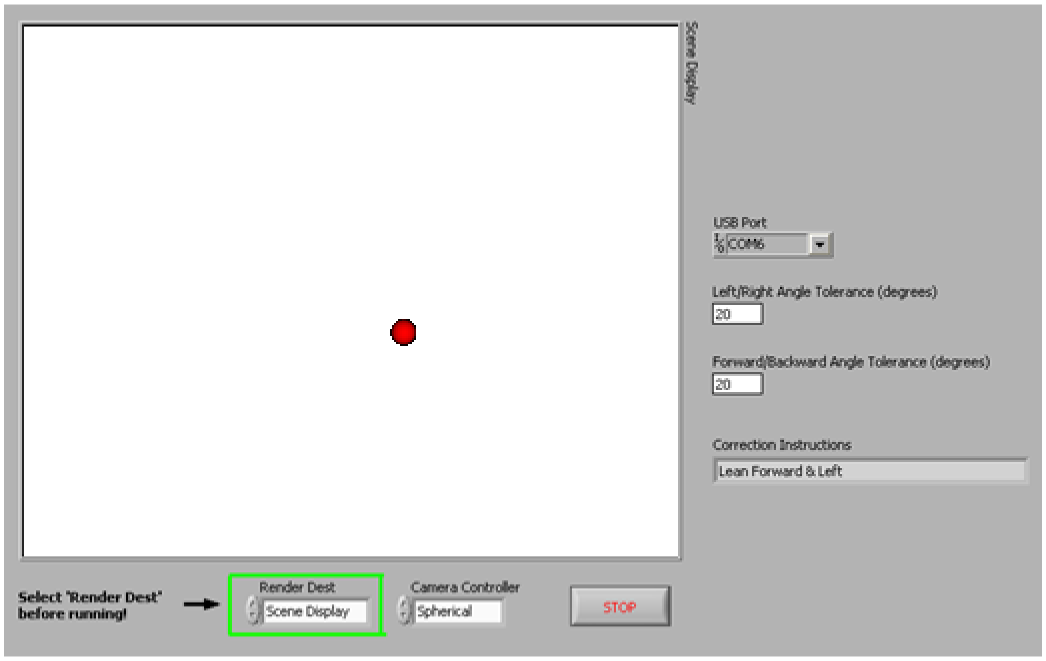
USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Lean Left
Select 'Render Dest' before running!
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.4: Accelerometer tilted to the right



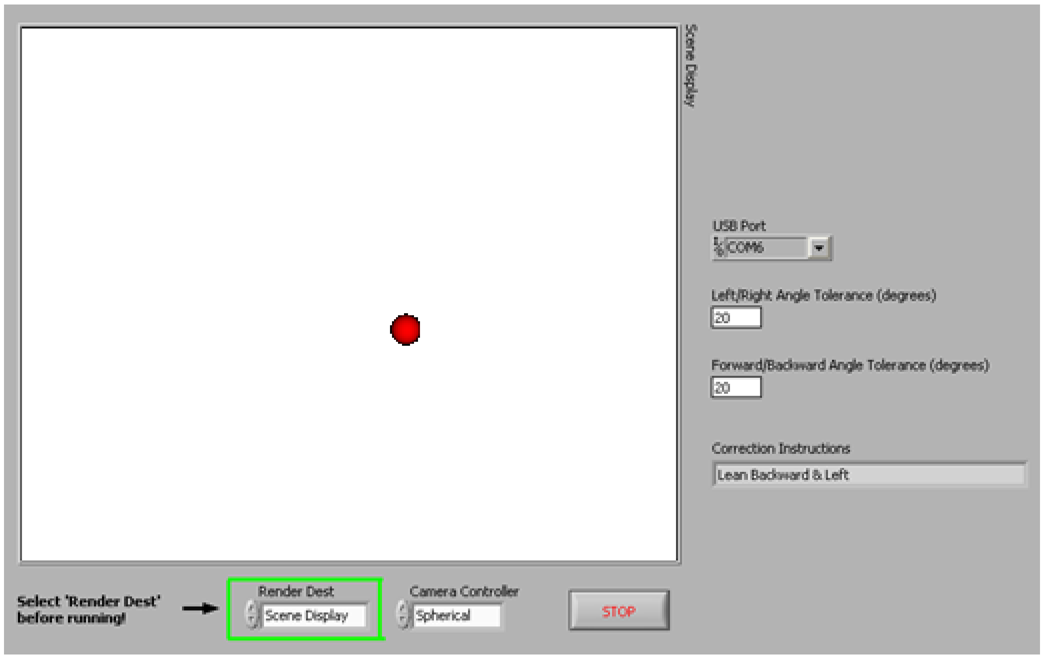
USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Lean Right
Select 'Render Dest' before running!
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.5: Accelerometer tilted to the left



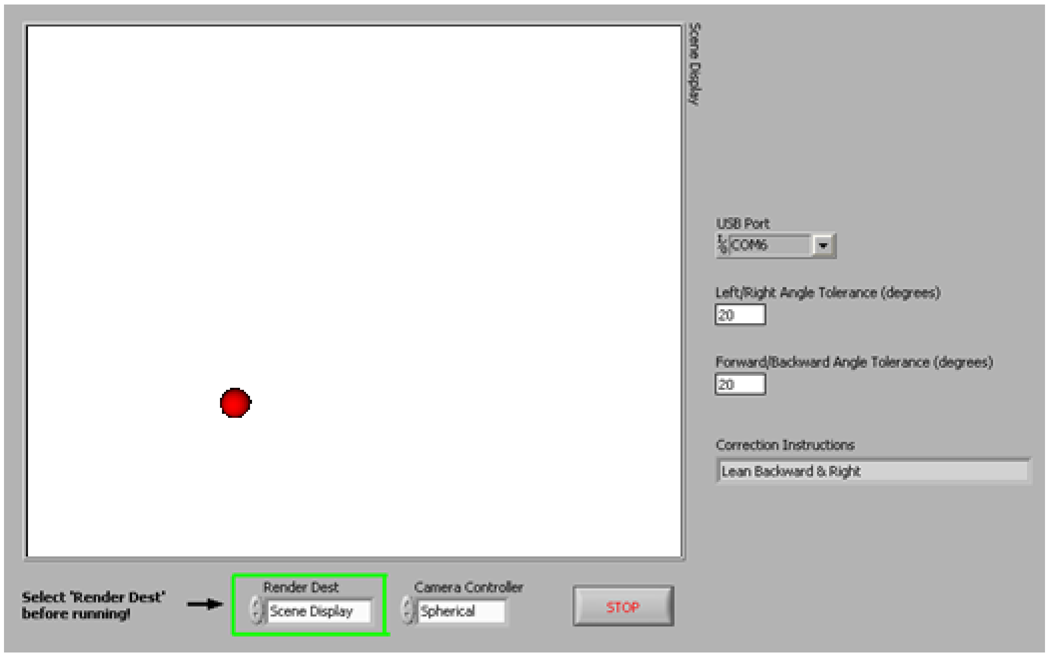
USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Lean Forward & Left
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.6: Accelerometer tilted backwards and to the right



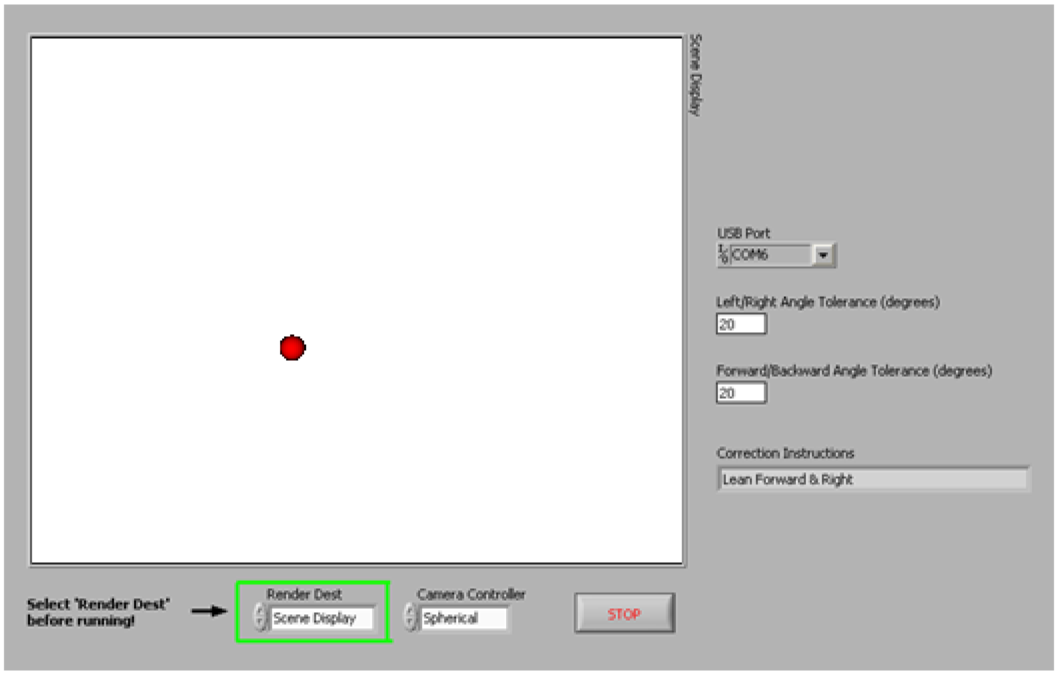
USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Lean Backward & Left
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.7: Accelerometer tilted forwards and to the left



USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Lean Backward & Right
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.8: Accelerometer tilted forwards and to the left



USB Port: COM6
Left/Right Angle Tolerance (degrees): 20
Forward/Backward Angle Tolerance (degrees): 20
Correction Instructions: Lean Forward & Right
Render Dest: Scene Display
Camera Controller: Spherical

Figure 5.9: Accelerometer tilted backwards and to the left

From Figures 5.1 to 5.9, it can be seen that the graphical model successfully mimicks patient’s posture movements, in three dimensions. The corrective instructions agree with the position of the accelerometer. Furthermore, when the corrective instructions are followed, the accelerometer is brought back to the home position (Figure 5.1).

6. CONCLUSIONS AND RECOMMENDATIONS

It can be seen that the novel, wearable posture sensing device is capable of measuring tilt posture in three dimensions. The hardware component senses improper posture, and delivers a vibration (via a vibration motor) to the patient. Meanwhile, the software component models the patient’s posture, as well as offering corrective feedback.

The portability of the system is convenient for the user, as the hardware component communicates wirelessly with the software module. Furthermore, the tilt angle tolerances can be adjusted by the user.

Upon reflection, it can be seen that both primary goals and secondary goals were completed successfully. The accelerometer data was converted into corresponding tilt angles (primary goal), which was then modelled graphically in LabVIEW (secondary goal). In addition, a wireless link was established between the hardware and software components (secondary goal). Overall, the design and implementation of the novel, wearable posture sensing device is successful.

In the future, the LabVIEW module should be updated to be more efficient. During the process of the project, the LabVIEW module was the preliminary source for tilt testing. As a result, a function was written in LabVIEW to calculate the tilt angles, using the accelerometry information. Later on, Microcontroller #1 was used to calculate tilt angles as well, as it was required for the hardware component. Unfortunately, this means that the tilt angles were calculated twice: once on the microcontroller, and once in LabVIEW. Although this extra calculation time is not noticeable by the user, it is not efficient. Future prototypes should transfer the calculated angles directly to the LabVIEW program. In addition, the LabVIEW program can be made to be more user-friendly. Currently, the program allows the user to input two fields:

left/right tilt angle tolerance, and forward/backward tilt angle tolerance. The fields can be expanded to four: left, right, forward, backward tilt angle tolerance.

The overall packaging of the system can be scaled down. Since the prototype created is the first prototype, aesthetics was not placed of higher importance. For future prototypes, a more compact packaging system should be used, as well as giving considerations for female and male body types.

7. APPENDIX A: MATLAB CODE FOR TILT ANALYSIS

```
function [x, y, time] = tilt_analy (filename, initialPos, axis)


% Finds the tilt position coordinates given acceleration in 1D
% filename: name of file, with time interval and acceleration recorded,
%   from Labview
% initialPos: coordinate of the initial position
% result: coordinate vector of the diplacement coordinates per time
% interval (same time interval as the input)


% Open file to read
fid = fopen(filename);
result= textscan(fid, '%f %f %f %f');
fclose(fid);


time = result{1,1};
Vx = result{1, axis+2}; %offset axis by 1 b/c 'time' is in the first column


% Adjust for acceleration
% From datasheet: 300mV/g sensitivity
g = 9.80665; %m/s^2
sens = .3;


% Calibration
zero_g = mean(voltage(1:100));
```

```
for i=1:length(voltage)
    accele(i) = (voltage(i) - zero_g)/sens*g;
end

% Tilt Calculations
theta = asin((Vx - V_offset)/sens);

Xo = 600;
Yo = 300;
Length_arm = 200;

[x y theta] = moving_avg(Vx);
x = zeros(1, length(theta));
y = zeros(1, length(theta));

for i=1:length(theta)
   if (theta<0)
       x(i) = Xo-Length_arm*cos(theta(i));
   else
       x(i) = Xo+Length_arm*cos(theta(i));
   end
   y(i) = Yo-Length_arm*sin(theta(i));
end

time = time(1:length(y));
```

8. APPENDIX B: TEST CODE FOR THE NRF24L01 TRANSCEIVERS

8.1 Transmission side

```
#include <Spi.h>
#include <mirf.h>
#include <nRF24L01.h>

void setup(){
   Serial.begin(9600);
   Mirf.init();
   Mirf.setRADDR((byte *)"clie1");
   Mirf.payload = sizeof(unsigned long);
   Mirf.config();
   Serial.println("Beginning ... ");
}

void loop(){
  //Variable sent = time
  unsigned long time = 4;
  Mirf.setTADDR((byte *)"serv1");

  //Send data
  Mirf.send((byte *)&time);

  //Waits until the program finishes sending
  while(Mirf.isSending()){
  }
  Serial.println("Finished sending");
  delay(1000);
}
```

8.2 Receiver Side

```
#include <Spi.h>
#include <mirf.h>
#include <nRF24L01.h>
void setup(){
  Serial.begin(9600);
  Mirf.init();
  Mirf.setRADDR((byte *)"clie1");
  Mirf.payload = sizeof(unsigned long);
  Mirf.config();
  Serial.println("Beginning ... ");
}

void loop(){
  unsigned long time ;
  Mirf.setTADDR((byte *)"serv1");

  //Loop until data is present
  while(!Mirf.dataReady()){
    Serial.println("Waiting");
  }

  //Receive the data
  Mirf.getData((byte *) &time);
  Serial.println((time));
   delay(1000);
}
```

9. APPENDIX C: WIRELESS LINK PROGRAM FOR TRANSMITTING ACCELEROMETER DATA

9.1 Transmission Side

```
#include <Spi.h>
#include <mirf.h>
#include <nRF24L01.h>


const int xpin = 1;
const int ypin = 2;
const int zpin = 3;


void setup(){
  // Configuration of the nRF24L01 transceiver
  Serial.begin(9600);
  // Initialization of chip
  Mirf.init();
  // Set receiving address
  Mirf.setRADDR((byte *)"clie1");
  // Set payload size to transmit three integers
  Mirf.payload = sizeof(int)*3;
  Mirf.config();
}


void loop(){
```

```
// Read in the accelerometer data
int t1 = analogRead(ypin);
int t2 = analogRead(zpin);
int t3 = analogRead(xpin);

// Set transmitting address
Mirf.setTADDR((byte *)"serv1");

// Initialize byte array to store data
byte transmit [sizeof(int)*3];

// Convert all integer data to bytes
byte* temp_x = (byte*)&t1;
byte* temp_y = (byte*)&t2;
byte* temp_z = (byte*)&t3;

// Convert the three byte arrays into one 1D array for transmitting
for (int i=0; i<sizeof(int); i++)
{
  // Store the first data in the first slot of the byte array
  transmit[i] = temp_x[i];
  // Store the second data in the second slot of the byte array
  transmit[i+sizeof(int)] = temp_y[i];
  // Store the third data in the third slot of the byte array
  transmit[i+2*sizeof(int)] = temp_z[i];
  }

  // Send the data
  Mirf.send(transmit);

  // Program pauses until sending is complete
  while(Mirf.isSending()){
  }

  Serial.println("Transmit Succeded.");

  delay(1000);
}
```

9.2 Receiving Side

```
#include <Spi.h>
#include <mirf.h>
#include <nRF24L01.h>

void setup(){
  // Configuration of the nRF24L01 transceiver
  Serial.begin(9600);
  // Initialization of chip
  Mirf.init();
  // Set receiving address
  Mirf.setRADDR((byte *)"clie1");
  // Set payload size to transmit three integers
  Mirf.payload = sizeof(int)*3;
  Mirf.config();
}

void loop(){
  // Initialize receving byte array
  byte receive[sizeof(int)*3];

  // Set transmitting address
  Mirf.setTADDR((byte *)"serv1");

  // Wait until there is data to be received
  while(!Mirf.dataReady()){
  }

  // Obtain the data
  Mirf.getData(receive);

  // Initialize the data variables
  float x_data=0;
  float y_data=0;
  float z_data=0;
  // Extract the data from the 1D byte array
  for (int i=0; i<sizeof(int); i++)
  {
    x_data = x_data + int(receive[i])*pow(256, i);
    y_data = y_data + int(receive[i+sizeof(int)])*pow(256, i);
    z_data = z_data + int(receive[i+2*sizeof(int)])*pow(256,i);
  }

  // Print the data
  Serial.print((int)x_data + "\t");
  Serial.print((int)y_data + "\t");
  Serial.print((int)z_data + "\t");

  delay(1000);
}
```

10. APPENDIX D: FINAL MICROCONTROLLER CODE

10.1 Microcontroller #1

```
#include <math.h>

#include <float.h>

#include <Spi.h>

#include <mirf.h>

#include <nRF24L01.h>

int ledPin =  6;

int pin9 = 9;

//set up vibration outputs

int leftVib = 2;

int rightVib =3;

int frontVib =4;

int backVib =5;


//set input analog channels

const int xpin = 3;

const int ypin = 2;

const int zpin = 1;


//set variables to hold analog inputs

float Xval=0;

float Yval=0;

float Zval=0;
```

```
//set offsets and calibration stuff
float offx=1.5;
float offy=1.5;
float offz=1.5;
int i;
volatile int CalibrateFlag=0; //variable to indicate whether interrupt has occured

//set accelerations
float ax;
float ay;
float az;

//angles
float alpha=0;
float beta = 0;
float delta = 0;

//accelerometer device sensitivities
float sensitivity=.22;

//flags to determine direction of tilt
int FlagLR=0; //flag to determine left and right tilt
int FlagFB=0; //flag to determine forward and backward tilt
```

```
//thresholds on angles that indicate tilt adjustment is required
float alphaThresh=20;
float betaThresh=20;
float deltaThresh=20;

void setup(){
  Serial.begin(9600);
  Mirf.init();
  Mirf.setRADDR((byte *)"clie1");
  Mirf.payload = sizeof(int)*3;
  Mirf.config();
  pinMode(leftVib, OUTPUT);
  pinMode(rightVib, OUTPUT);
  pinMode(frontVib, OUTPUT);
  pinMode(backVib, OUTPUT);
  pinMode(ledPin, OUTPUT);
  //initialize pin0 as input
  pinMode(pin9, INPUT);
}


void loop()
{
  if(digitalRead(pin9)==0){  //calibration switch is pressed
   offx = 0;
```

```
    offy = 0;

    offz = 0;

for(i=0;i<10;i++){ //find average value of offset for calibration

 offx = offx + analogRead(xpin);

offy = offy + analogRead(ypin);

offz = offz + analogRead(zpin);

delay(50);

         }

offx = offx/10 *3.3/1023 - sensitivity; //special case for X axis sensitivity

offy = offy/10 *3.3/1023;

offz= offz/10*3.3/1023;

        }


//read in values from accelerometer

  Xval = analogRead(xpin);

  Yval = analogRead(ypin);

  Zval = analogRead(zpin);


// find accelerations

ax = (Xval*3.3/1023-offx)/sensitivity;

ay = (Yval*3.3/1023-offy)/sensitivity;

az = (Zval*3.3/1023-offz)/sensitivity;

//find the angles and convert to degrees


  alpha =   atan(az/sqrt(pow(ay,2)+pow(ax,2)))*(180/3.14592);
  beta =   atan(ay/sqrt(pow(az,2)+pow(ax,2)))* (180/3.14592);
  delta = atan(sqrt(pow(ay,2)+pow(az,2))/ax)* (180/3.14592);

  if(delta>=deltaThresh){
            if(beta>betaThresh) FlagLR = 1; //tilting left
            else if (beta<-betaThresh) FlagLR=-1;//tilting right

            if(alpha>alphaThresh-10) FlagFB= -1; //tilting backward
        else if (alpha<-alphaThresh) FlagFB = 1; //tilting forward

            }
else{
  FlagLR=0;
  FlagFB=0;
}

if(FlagFB!=0 || FlagLR!=0)
digitalWrite(ledPin,HIGH);
else
digitalWrite(ledPin,LOW);
//set front and back vibration
 if(FlagFB==1)
 {
  digitalWrite(frontVib, HIGH);   // set actuator motor on
 }
 else if(FlagFB==-1)
 {
   digitalWrite(backVib, HIGH);


 }
 else
 {
            digitalWrite(frontVib, LOW);
         digitalWrite(backVib, LOW);
 }

//set left and right vibration
if(FlagLR==1)
{
  digitalWrite(leftVib, HIGH);   // set actuator motor on
}
else if(FlagLR==-1)
{
  digitalWrite(rightVib, HIGH);
}
else
{
  digitalWrite(leftVib, LOW);
  digitalWrite(rightVib, LOW);
```

```
}

//Transmit data wirelessly
  int t1 = analogRead(ypin);
  int t2 = analogRead(zpin);
  int t3 = analogRead(xpin);

  Mirf.setTADDR((byte *)"serv1");

  byte test [sizeof(int)*3];
  byte* temp1 = (byte*)&t1;
  byte* temp2 = (byte*)&t2;
  byte* temp3 = (byte*)&t3;

  for (int i=0; i<sizeof(int); i++)
  {
    test[i] = temp1[i];
    test[i+sizeof(int)] = temp2[i];
    test[i+2*sizeof(int)] = temp3[i];
  }
    Mirf.send(test);

  while(Mirf.isSending()){
  }
```

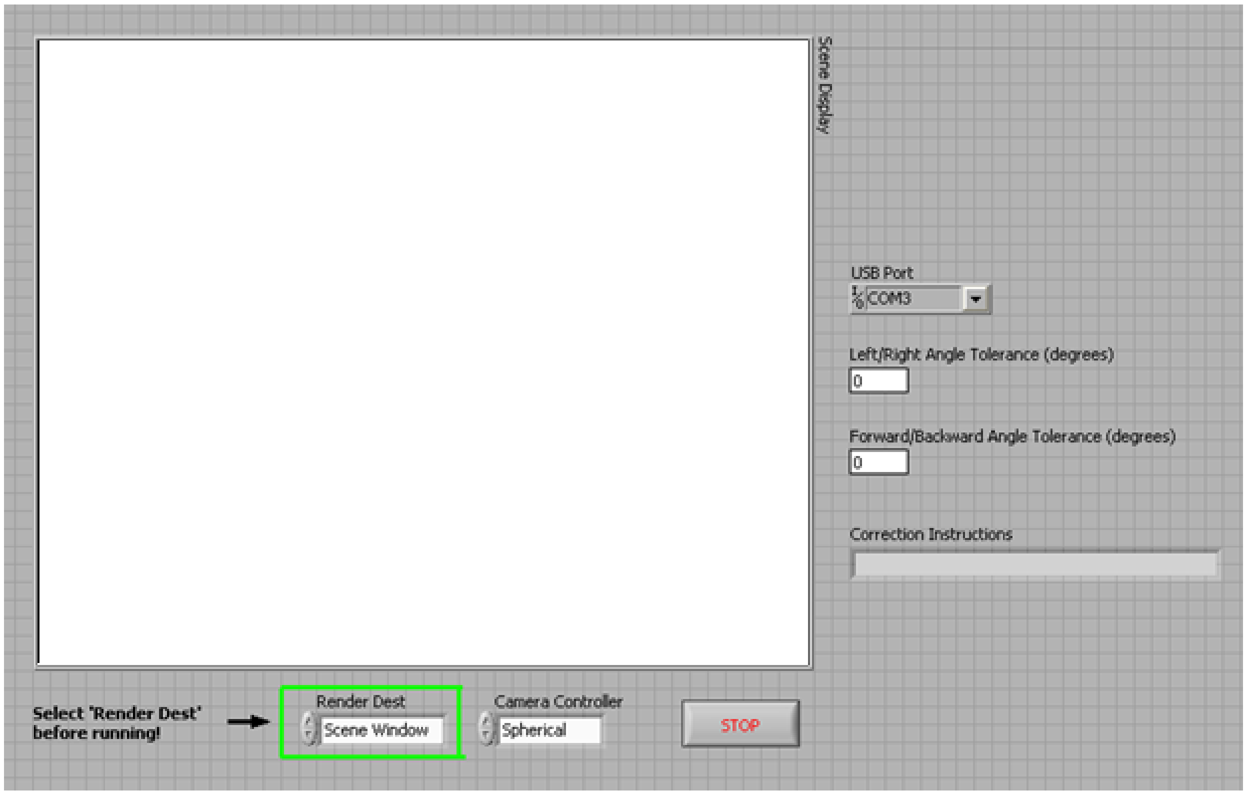
```
}
```

10.2 Microcontroller #2

```
#include <Spi.h>
#include <mirf.h>
#include <nRF24L01.h>

void setup(){
  Serial.begin(9600);
  Mirf.init();
  Mirf.setRADDR((byte *)"clie1");
  Mirf.payload = sizeof(int)*3;
  Mirf.config();

}
int check=0;
void loop(){
  byte test[sizeof(int)*3];
  int time;
  Mirf.setTADDR((byte *)"serv1");
  while(!Mirf.dataReady()){
  }
  Mirf.getData(test);
  float time1=0;

  float time2=0;

  float time3=0;


  for (int i=0; i<sizeof(int); i++)

  {

    time1 = time1+int(test[i])*pow(256, i);

    time2 = time2 + int(test[i+sizeof(int)])*pow(256, i);

    time3 = time3 + int(test[i+2*sizeof(int)])*pow(256,i);

  }


  Serial.print((int)time1);

  Serial.print("\t");

  Serial.print((int)time2);

  Serial.print("\t");

  Serial.print((int)time3);

  Serial.print("\n");

  check=1;


}
```

11. APPENDIX E: LABVIEW FINALIZED PROGRAM

11.1 Front Panel

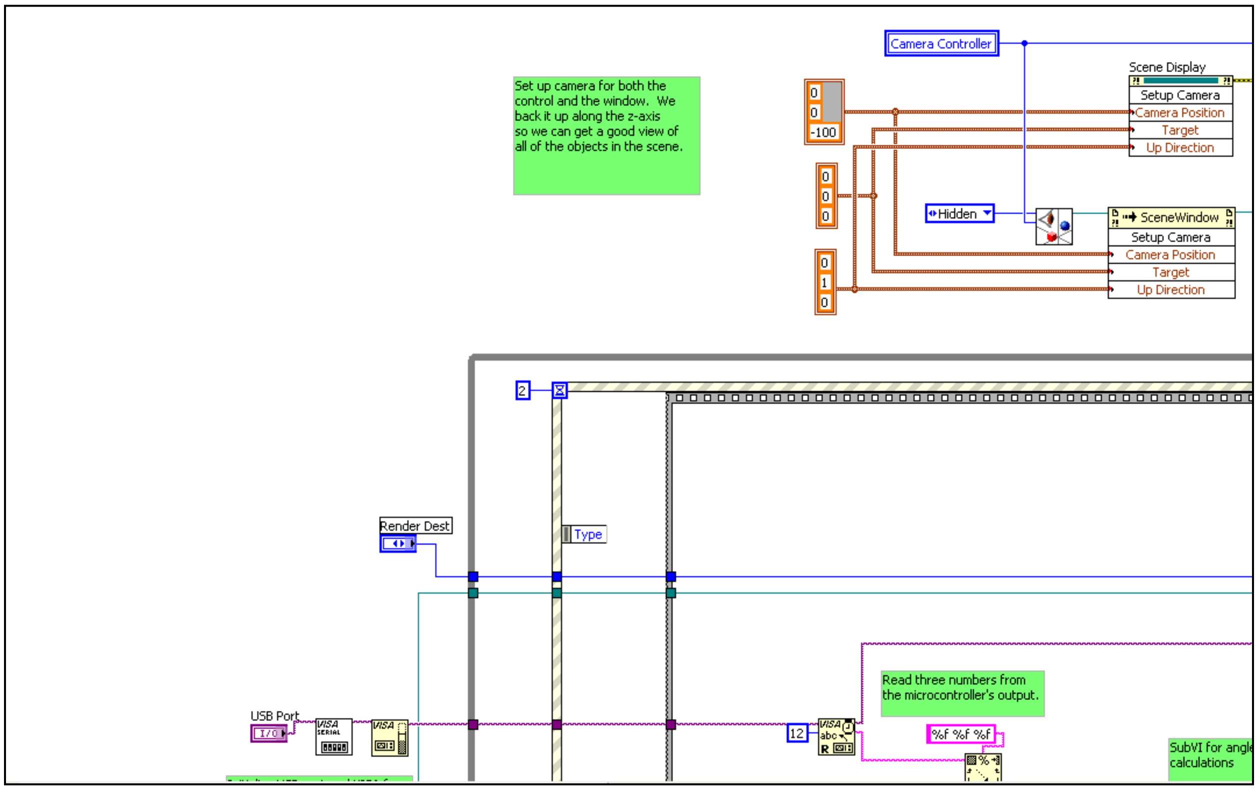

Scene Display
USB Port
COM3
Left/Right Angle Tolerance (degrees)
0
Forward/Backward Angle Tolerance (degrees)
0
Correction Instructions
Select 'Render Dest' before running!
Render Dest
Scene Window
Camera Controller
Spherical
STOP

The front panel is the user interface.

The right side bar allows the user to select the USB port where the wireless transceiver is connected to (via Microcontroller #2). He/she can also specify the tilt tolerance in degrees. The posture correction instructions also appear here.

On the bottom panel, the user can specify to render via the “Scene Window”, which opens up a pop-up window, or the “Scene Display”, which displays the graphic on the current screen. The user can also specify a camera controller type (used to rotate the graphical region). There is also a “Stop” button to end the program.

11.2 Top Left of Block Diagram

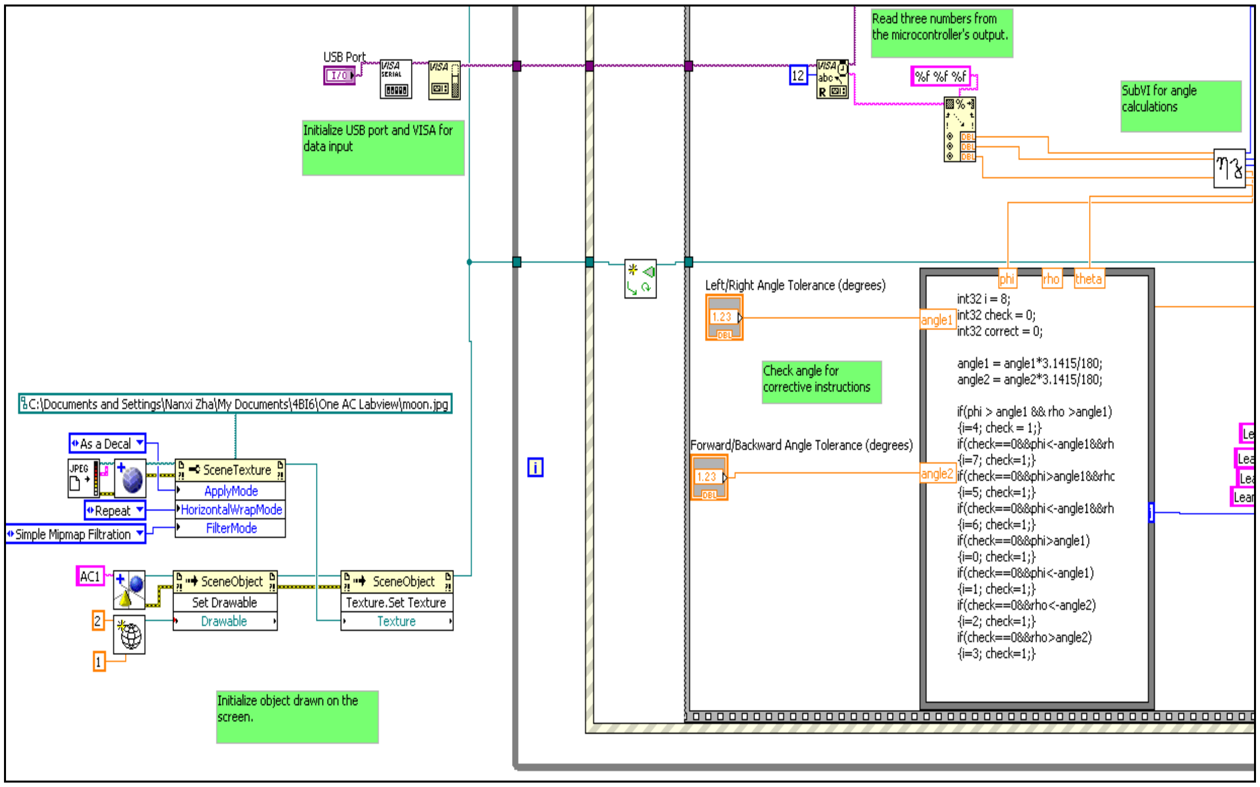


Set up camera for both the control and the window. We back it up along the z-axis so we can get a good view of all of the objects in the scene.

Read three numbers from the microcontroller's output.

SubVI for angle calculations

11.3 Bottom Left of Block Diagram



Read three numbers from the microcontroller's output.

Initialize USB port and VISA for data input

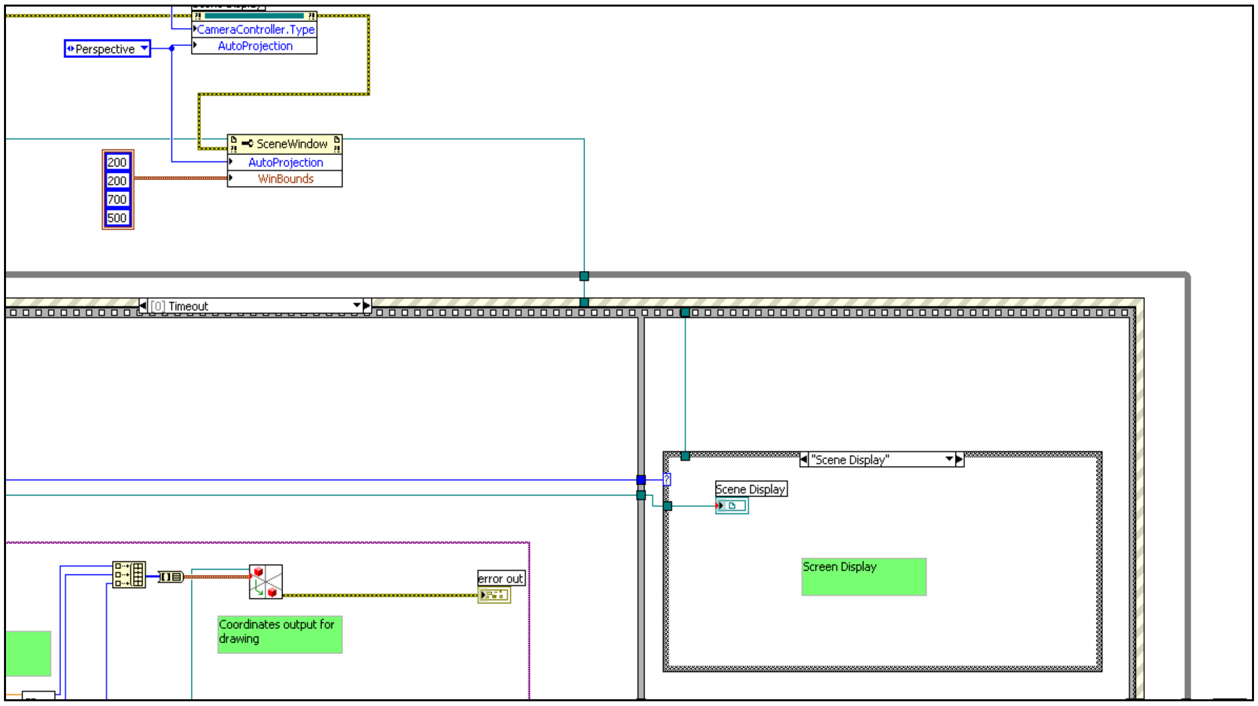
SubVI for angle calculations

Check angle for corrective instructions

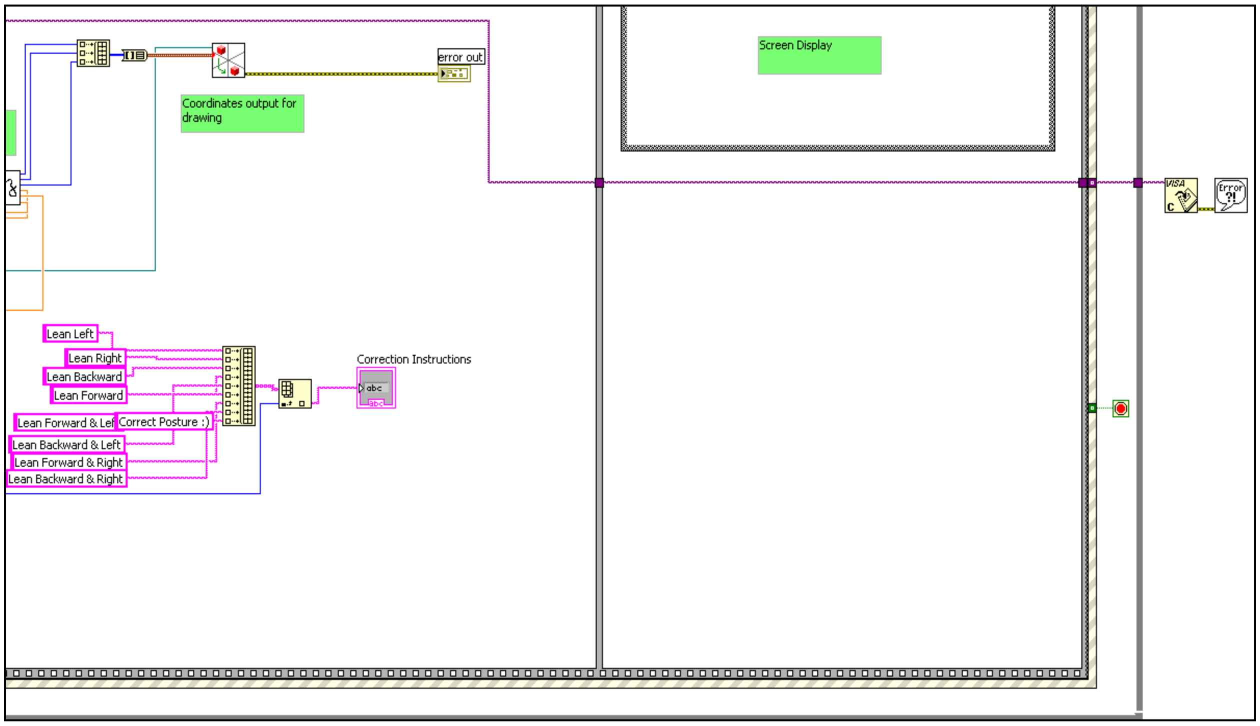
Initialize object drawn on the screen.

```
int32 i = 8;
int32 check = 0;
int32 correct = 0;

angle1 = angle1*3.1415/180;
angle2 = angle2*3.1415/180;

if(phi > angle1 && rho >angle1)
{i=4; check = 1;}
if(check==0&&phi<-angle1&&rh
{i=7; check=1;}
if(check==0&&phi>angle1&&rhc
{i=5; check=1;}
if(check==0&&phi<-angle1&&rh
{i=6; check=1;}
if(check==0&&phi>angle1)
{i=0; check=1;}
if(check==0&&phi<-angle1)
{i=1; check=1;}
if(check==0&&rho<-angle2)
{i=2; check=1;}
if(check==0&&rho>angle2)
{i=3; check=1;}
```

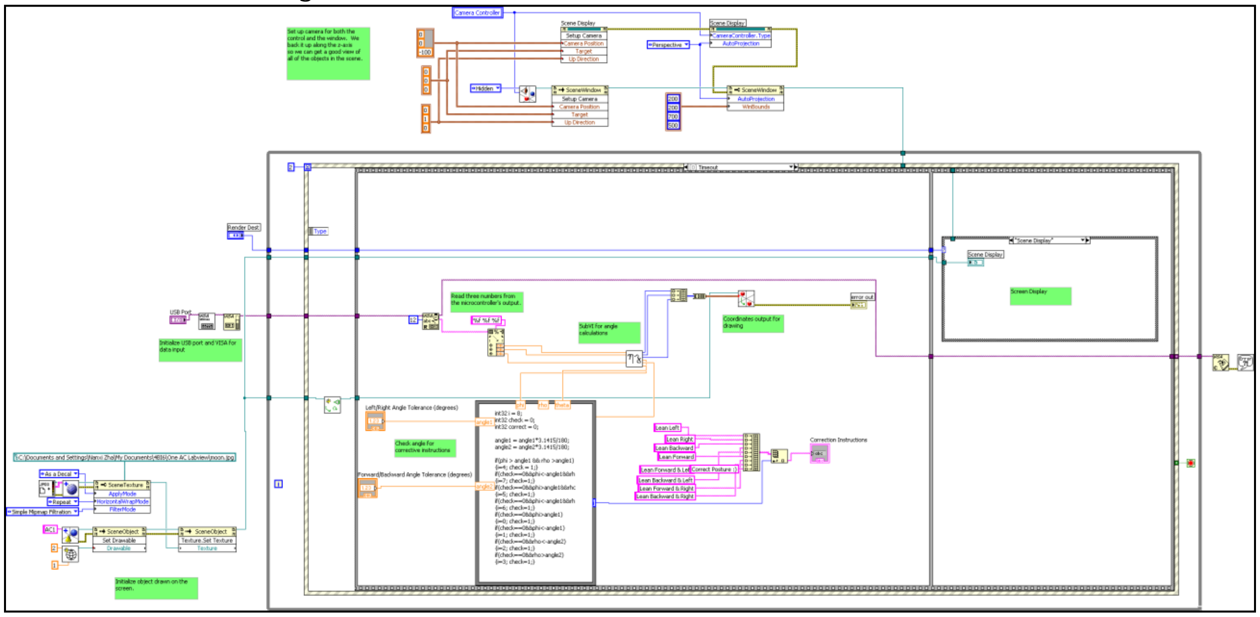
11.4 Top Right of Block Diagram



11.5 Bottom Right of Block Diagram

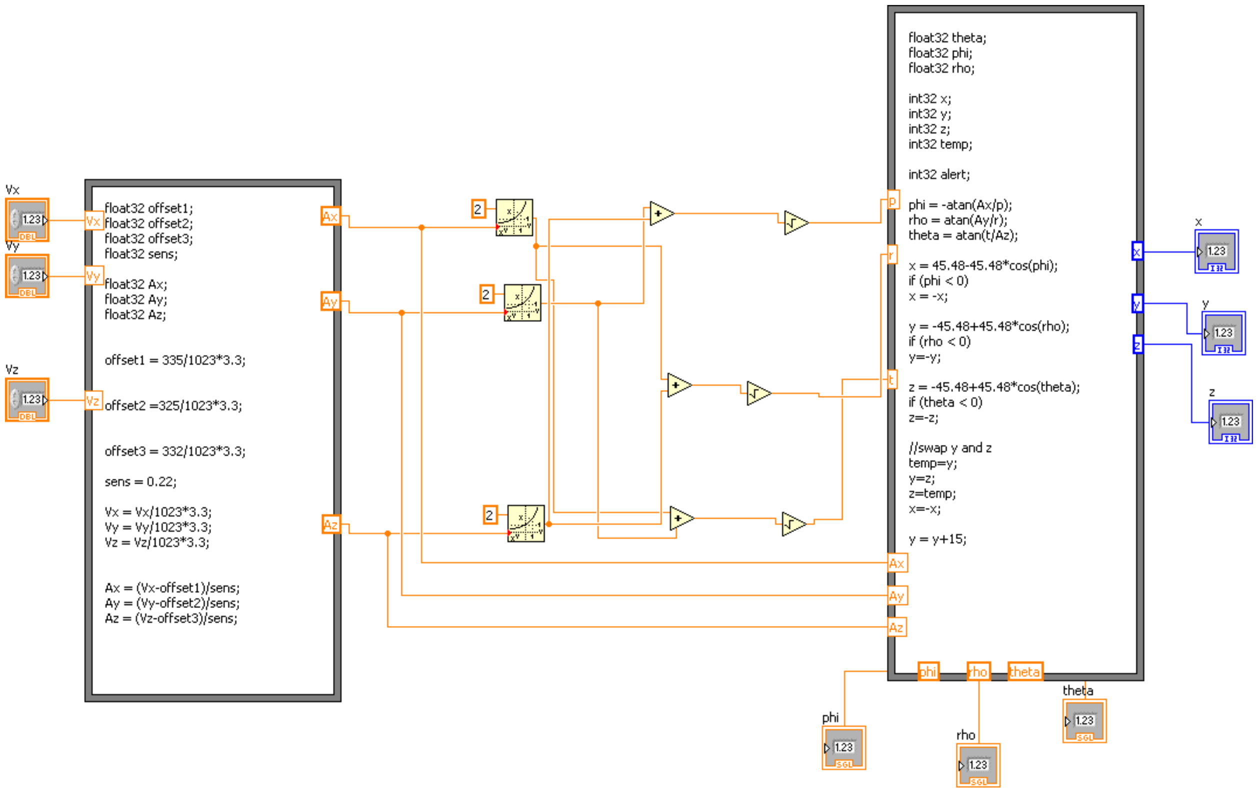


11.6 Overall Block Diagram



```
int32 i = 8;
int32 check = 0;
int32 correct = 0;

angle1 = angle1*3.1415/180;
angle2 = angle2*3.1415/180;

if(phi > angle1 && rho >angle1)
{i=4; check = 1;}
if(check==0&&phi<-angle1&&rh
{i=7; check=1;}
if(check==0&&phi>angle1&&rhc
{i=5; check=1;}
if(check==0&&phi<-angle1&&rh
{i=6; check=1;}
if(check==0&&phi>angle1)
{i=0; check=1;}
if(check==0&&phi<-angle1)
{i=1; check=1;}
if(check==0&&rho<-angle2)
{i=2; check=1;}
if(check==0&&rho>angle2)
{i=3; check=1;}
```

11.7 SubVI for Angle Calculation



```
float32 offset1;
float32 offset2;
float32 offset3;
float32 sens;

float32 Ax;
float32 Ay;
float32 Az;


offset1 = 335/1023*3.3;

offset2 =325/1023*3.3;

offset3 = 332/1023*3.3;

sens = 0.22;

Vx = Vx/1023*3.3;
Vy = Vy/1023*3.3;
Vz = Vz/1023*3.3;


Ax = (Vx-offset1)/sens;
Ay = (Vy-offset2)/sens;
Az = (Vz-offset3)/sens;
```

```
float32 theta;
float32 phi;
float32 rho;

int32 x;
int32 y;
int32 z;
int32 temp;

int32 alert;

phi = -atan(Ax/p);
rho = atan(Ay/r);
theta = atan(t/Az);

x = 45.48-45.48*cos(phi);
if (phi < 0)
x = -x;

y = -45.48+45.48*cos(rho);
if (rho < 0)
y=-y;

z = -45.48+45.48*cos(theta);
if (theta < 0)
z=-z;

//swap y and z
temp=y;
y=z;
z=temp;
x=-x;

y = y+15;
```

12. VITAE

NAME: Nanxi Zha

PLACE OF BIRTH: Chengdu, Sichuan, China

YEAR OF BIRTH: 1988

SECONDARY EDUCATION: Nelson High School (2002 – 2006)

HONOURS and AWARDS: Honour Roll Student (2002 – 2006)

Governor General's Bronze Medal (2006)

The Nortel Network scholarship (2006)

The Miller Thompson National Scholarship (2006)

The McMaster University President Entrance Scholarship (2006)

The University Senate Scholarship (2007, 2008)

The Ontario Professional Engineers Foundation for Education Undergraduate Scholarship (2009)

13. BIBLIOGRAPHY

Fahrenberg, J., Foerster, F., Smeja, M. & Muller, W. (1997), "Assessment of posture and motion by multichannel piezoresistive accelerometer recordings", *Psychophysiology*, vol. 34, no. 5, pp. 607-612.

Giansanti, D., Macellari, V., Maccioni, G. & Cappozzo, A. (2003), "Is it feasible to reconstruct body segment 3-D position and orientation using accelerometric data?", *IEEE transactions on bio-medical engineering*, vol. 50, no. 4, pp. 476-483.

Godfrey, A., Conway, R., Meagher, D. & OLaighin, G. (2008), "Direct measurement of human movement by accelerometry", *Medical engineering & physics*, vol. 30, no. 10, pp. 1364-1386.

Hyde, R.A., Ketteringham, L.P., Neild, S.A. & Jones, R.S. (2008), "Estimation of upper-limb orientation based on accelerometer and gyroscope measurements", *IEEE transactions on bio-medical engineering*, vol. 55, no. 2 Pt 1, pp. 746-754.

Kavanagh, J.J. & Menz, H.B. (2008), "Accelerometry: a technique for quantifying movement patterns during walking", *Gait & posture*, vol. 28, no. 1, pp. 1-15.

Mizuike, C., Ohgi, S. & Morita, S. (2009), "Analysis of stroke patient walking dynamics using a tri-axial accelerometer", *Gait & posture*, vol. 30, no. 1, pp. 60-64.

Morris, J.R. (1973), "Accelerometry--a technique for the measurement of human body movements", *Journal of Biomechanics*, vol. 6, no. 6, pp. 729-736.

Punt, T.D. & Riddoch, M.J. (2006), "Motor neglect: implications for movement and rehabilitation following stroke", *Disability and rehabilitation*, vol. 28, no. 13-14, pp. 857-864.

Tuck, K. (2007), "Tilt sensing using linear accelerometers", *Freescale semiconductor,* Application notes, pp. 1-8