**CSE398 Junior Design Report**

**Multi Functional Control**

Team Members:

Dingfeng Shao

Zilong Li

Junfeng Que

**Introduction:**

This report presents a design of a multi functional control device that includes Beaglebone black, Wi-Fi adapter, LSM303 accelerometers and L3G20D inertial measurement unit, which includes accelerometer and gyroscope. We also used C# to modify the data for computer. This design was a valuable experience because we combined the data from sensors with hand gestures to control the computer curser and customized actions, such as screen capture, through wireless connection. Additionally, we created another game mode for this device, which users could play Fruit Ninja as a motion sensing game by naturally waving their hands in the air.

This design included three main objectives. The first objective was analyzing the sensors’ data and combining with hand gestures. We should make sure that hand gestures are easily for users to remember, and different gestures would not affect each other. The second objective was the communication between hardware. We used I2C interface to connect the sensors and Beaglebone, UDP to connect Beaglebone and C# program. The last objective of the design was how to combine the hardware together. We used a glove to hold the device, so users could wear the device and make hand gestures.

**Method:**

**Part1: The Basic Design**

The big first part of the junior design was come up a very good design prototype for the project. We had come up a design of motion sensing with a wearable glove. The fundamental idea of this project is to control the windows system by using the hand gesture and the finger commands. This design was a really practical idea because we used the LSM303 Sensor, which we used in the previous lab for motion detection. The following relation diagrams explain a brief idea of the interaction between user and the motion console.

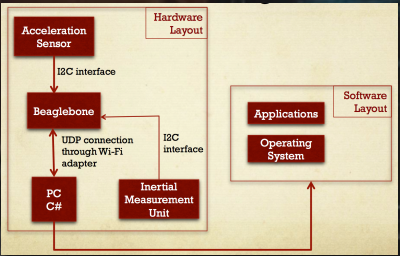
Beaglebone

bonebone operation

Hand motion

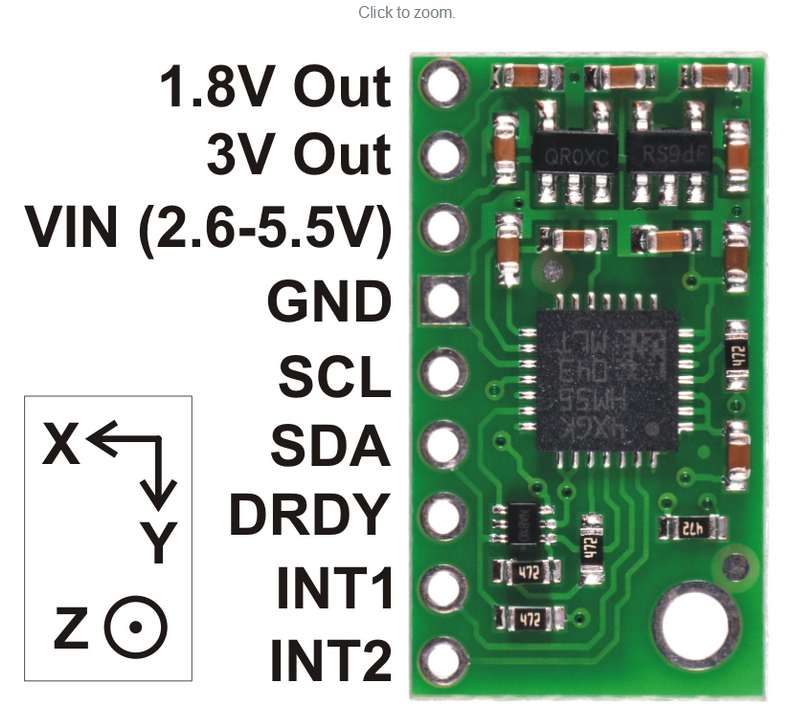
Sensor filter

In the previous Lab, we used the accelerometer to catch and filter out the hand motion. Then we sent captured raw data to beagle bone through SPI. The sensor was able to give us some useful data such “-1”, “1” and “0” flags for the highest bit. However, we might lack of hand gesture by only using the flag. For the beagle bone, we could only have two ports for the SPI at most, which limited the usage of the sensor. (We did change the SPI interface to I2C interface for the future IMU use) Then we thought about what could actually do with motion data. We could converted these data into a cursor type coordinates especially for the accelerometer. We could wave or swing the glove for tracking the coordinates and its motion direction. In this case, we were able to control both hardware layer and software layer by using the wife connection or cable connection.



**Part 2: SPI and LSM303 for testing (Accelerometer and Magnetometer)**

In this part of the design, we planned to get more understanding of the sensor. Therefore we could write re-useable sensor libraries via SPI. The LSM303 sensor’s complete name is “LSM303D 3D compass and accelerometer carrier with voltage regulator”. This chip were six independent reading and available through I2C and SPI interface.



Before we get our hands dirty for hard coding the libraries, we needed to have a pre made SPI libraries. We used the pre – configured SPI interface before it enables its communication abilities between the SPI and the sensor itself. The following initialization has to make before reading the data from the sensor. The SPI object opens the bus terminal for one SPI DEV from the beagle bone so the beagle could interact with the sensor. We also set the SPI mode to zero and set the frequency speed to 50000 for default reading setting. For communication purpose, we could easily invoke the “XFREL” function from the SPI class. (See Appendix)

Before we wrote our libraries for the sensor, we should read the online data reference for more specification. This LSM303D sensor had many configurable options, including dynamically selectable sensitive and the form of the output data rate. After we had read the data reference, we noticed that we needed to configure all the control pointes for the ideal data from both accelerometer and magnetometer. For the accelerometer, we set the control point one to 0x57 which 01010111 in binary form. The ctrl one register enables the accelerometer data rate to 50 Hz. It also enables the x-axis, y-axis and z-axis. For the control two, we set that tot 0x00 for 50 Hz acceleration anti-alias filter bandwidth and +- 2 gauss. For the magnetometer, we did the same enabling process for ctrl 5, 6 and 7, which set the resolution mode to high and the data rate to 6.25 Hz.

Lastly, we read all 6 channels of the LSM303 Sensor and stores in the vectored type variables. We combine the high byte and the low bytes into an int\_16 signed variables for each axis.

**Part3: I2C and IMU (Gyroscope, Accelerometer and Magnetometer)**

The object for this part of the junior lab was to build an I2C connection between IMU and beagle bone. The Beaglebone black has three I2C buses:I2C0, I2C1, and I2C2. However, not all three buses are export by the default and more confusingly under Linux, the buses were named in the order they were enumerated. So the best way was to check the memory address of differed sensor devices.

Before using the I2C, update the Linux i2c tool for commands control over the terminal. Enabled the third i2c bus by echoing the BB-i2c to slots. Then used the i2cdetect –l to check the OMAP i2c adapter that had the i2c terminals 0 to 1. The next step was to check the i2c device address, we used the command # i2cdetect –y –r 1. All the addresses were shown with 16 hexadecimal numbers. Since we were able to keep tracking with the device address and the sensor register from the data references. We were built the i2c libraries using these two major read and write method: i2cset and i2cget.

Right After we configured the i2c interface communication libraries, we initialized the gyroscope (L3G20D) with enabled value for both sensitivities and rotation axis. Since this was the IMU sensor that had the 9 degree of freedom. We could also obtain the raw data for both accelerometer and magnetometer.

**Part 4: UDP connection for both wireless and cable connection.**

After we got the sensor from the sensor, we need to send the data to the computer by UDP. We copied the same code that we used from the previous lab for the UDP connection and keep sending data from the beagle bone to the PC local host. Then in the window environment, we developed a C# server to receive all the data from the beagle bone client. We basically developed a very simply UI to show the data received from the beagle bone and used the data binding. Since we would use the integer to locate the cursor, we converted the string type to the integer type and used an infinite while loop keep capturing the input sending data. We also had to set the IP address of the C# server to 192.168.7.1. (The beagle‘s IP is 192.168.7.2), the send port to 1234 and the receive port to be 54321.

After the C# server received the data, we used the WIN32 API for locating the cursor. We developed an algorithm that calculates distance using two timers in the mainframe window. One is to detect the button press from the hand. Once the “S” Key was pressed from the keyboard, the application would set the initiated point for the current user’s X and Y coordinates. The two major formulas were just two simply linear functions: TempX = Xp – Ixp and TempY = Yp – Iyp. If the TempY and TempX were lower than 8, which is a stable range, the cursor would not move. Otherwise, the cursor would increase or decrease depends on the amount of TempX and TempY. If the difference is larger, the cursor will speed up faster. To control the mouse location on the screen, we used the WIN32 API call SetCursorPositon() function.

Next, we needed to deal with the click function base on the accelerometer from the index finger. Once the finger raw motion data were over at the certain limits, we called the function mouse right click and left click from the WIN32 API. After that, we developed a screen capture function once we force the hand down in very high speed. We would called the Get Screen shot function that does the graphic copy form the screen. Then we automatically save that to the desktop.

In order to make multifunction, we try not only used the accelerometer but also the gyroscope. We analyses the data from the gyroscope and developed another algorithm for game mode to move the mouse since we only want us that that when we play the game.

**Part 5: Algorithm**

We had to setup some mathematical formulas and definition for sensor filter. The first definition of the sensor was the sensitivity. We defined the gravity to 256, which is equivalent to 1G in the raw data coming from the accelerometer. Then, we also define the radian and degree converting unit function for math calculation. Next, we used the 2000dps full scale for gyroscope and set the gyro gain to 0.7 for 1 DPS. This action makes some scaled ADC raw data for gyro.

We also set the delay function and millis function. The delay function did the sleep for didference situations. The millis function returns the current system time with the precision of millis second. The point of creating these two functions was to have very precise system utilization and stabilized the sensor with their output rate.

The Filter was also able to correct the offset for both gyro and accelerometer for accuracy. The setup session of this filter took a 32 size of sample and accumulated all the data for offset calculation. This Setup section also regulated the gravity, gyro gran, timer and event trigger counter for reading.

**Part 6: Design of the glove**

After finishing the coding for both c# and Beaglebone, we started to design the combination of glove and hardware device. We put the battery cape right on top of the Beaglebone, which could supply power to the beagle bone using AAA batteries. The USB port connected the Wi-Fi adapter for network connection. The location of the wife adapter was point to the same direction of the fighter so we could have a better connection performance without interfering the adapter. In order to use the hand gestures to control the computer, we decided to use a ring to carry the sensor. We purchase the leather glove for the electric non-conduction purpose. In order to make the while device wirelessly control by hand, the battery cape must be charge by the 5V supply or the AAA batteries. Lastly, we soldered all the wires directly with the sensor therefore we could put the sensor on the ring and connect the wire s to the beagle bone.

**Part 7: Final Testing**

At the last day of the junior lab, we deiced to stop advancing the glove’s feature and started testing that for stable performance. There are few different technical problems or testing point that we had found for the glove and we tried to improve or solve each of them. At first, we found out that the cotton thread is very easy to break and that could cause the connection problem, which the Beaglebone may drop on the ground. Therefore, we used some metal wire to replace the cotton thread and we found it strongly bind to the glove. Next, we wanted to add a more stabilized click function to the device for better performance. After improving the click performance, we decided to add more if statements for determining for testing and interrupt trigger. Whenever a bug occurs, the application would goes down and gave back control to the PC. We also found out that the glove was heavily rely on the stable network connection, so we decided to use the cable connection for the open house. Finally, Instead of using the soldered wire sensor connection, we just used the ribbon cable for better contact with the bone.

**Result:**

**SPI and LSM303 for testing:**

After we connecting the LSM303 sensor to Beaglebone, by changing the sensor’s horizontal angles through different directions, we could receive six groups of changing data, and we could use three groups of data to determine the position of the sensor. However, we found that data kept changing even when we hold the sensor as stable as we could. After changing the gravity level of the sensor from +8 to +16, we were able to control the deviations at +-10.

**I2C and IMU:**

After we connecting the gyroscope to Beaglebone, by changing the sensor’s horizontal angles through different directions, we could receive nine groups of data. WE analyzed the data and found that these nine groups of data could tell us the position of the sensor.

**UDP connection for both wireless and cable connection:**

We connected all the hardware on the glove. After closing the computer’s firewall and pressing the “S” button while the C# program was running. When we changing the horizontal angles of the hand left or right, the cursor on the computer also moves to left or right. When we changing the horizontal angles of the hand front and back, the cursor also moves up and down. If we moved the finger with the LSM303 sensor up or down, the cursor would do a click or right click. We could also save a computer screen saver if we forcing our hand down in high speed.

**Algorithm:**

After modifying the C++ program in Beaglebone and the C# program, when we do hand gestures, the cursor moves very smooth, and we could be able to hold the cursor not moving when we comfortably stable our hand.

**Conclusion：**

In this junior design lab, we created a multifunctional remote control. We used sensors to detect hand motion and sent data to Beaglebone. Then, Beaglebone transferred data to a C# program in PC using UDP. After that, C# analyzed data and controlled the cursor in windows. Finally, we could use the remote control to do many operations, such as moving mouse, clicking, double clicking, right clicking, screen shot. Also, we developed a game mode to play fruit ninja using gyroscope.

In the first part, we had trouble in designing our project. Since we did not have any implementation, we had to design our own project. We listed many options, such as small cars, smart traffic. However, it was hard to finish these stuffs in three weeks. Thus, we decided to make a wearable device, which was quite popular. After that, we decided to use LSM303 sensors we used in previous lab. Also, we figured out other requirements: Beaglebone black and C#.

In the second part, we found that accelerometer had a disadvantage. We only had three data: 0, positive and negative. If we want to move the mouse, we need more accurate data to allocate the cursor. Thus, we did a lot of search on the Internet and figured out how to get more data from accelerometer. These data were based on accelerated velocities. Faster we moved, bigger the corresponding data were. Using these data, we were able to move the mouse stably.

Although accelerometer made mouse move steadily, it was not flexible. We tried to replace accelerometer with gyroscope, which would be smoother. However, we found that we could only I2C ports to transfer data for gyroscope. Previously, we were using SPI (Motorola). I2C (Intel) was a new transfer method similar to SPI. We did some search on the Internet. Then we wrote our own header file and cpp files to get access to data using I2C. However, we found that gyroscope had a characteristic. It always went back to the normal status (0) after we changed the data. It meant that it would be hard to analyze the data and could not be as stable as accelerometer. Therefore, we decided to use both sensors to move the cursor. When we manipulated the operation system (windows 7), we used accelerometer to guarantee the stability. When we played games, like fruit ninja, we used gyroscope to make the cursor more flexible.

After we got the data from sensor, we had another trouble. We could not send data from Beaglebone to the lab computer. At first, we thought we needed to bind an IP address with UDP socket. However, we found that only the receive end should bind with an IP instead of the send end. After that, we figured that it was because campus network turned on the firewall. It blocked data sending to the computer. Our data were from unsecure source, so they were blocked. Then we used our own PC and turned off the firewall. It worked well.

In the next part, we wanted the wearable device to be wireless. Therefore, we purchased a battery cape and a wireless adapter. However, since there were more than 5 groups sharing one router, the wireless network was so bad that we could not even connect to Beaglebone. We thought that a better adapter may improve the performance. After we changed to a new one, we could connect to Beaglebone, but sometimes the data transfer was delayed. The cursor was always stuck. Also, a battery cape could only last about one hour. If we switched to another power supply, Beaglebone would reset. We had to open the interface and ran the program again. Thus, we decided to use USB cable instead to guarantee the beast performance.

When we tested the device, we noticed that sometimes sensor did not work well and we kept receiving the exact same data. After we restarted the program, it worked again. Then we found it may be a connection problem. At first, we soldered wires directly to the sensor. It may cause bad contact. Therefore, we used ribbon cable to connect sensors and Beaglebone. Although it was hard to install the cable on the glove, this situation did not happen anymore.

During testing, we also found that right click and left click were triggered by mistake sometimes. Since the data seemed similar when we moved the cursor, our algorithm invoked the click functions at the same time. In order to avoid this problem, we decided to add an “if” statement. Click functions could be invoked only if the cursor was in stable status. After that, click functions worked well.

In the last few days, we kept testing the device and adjusted the limits to make the functions more accurate. The performance was pretty good in the end.

**Appendix：**

#ifndef \_I2C\_H\_

#define \_I2C\_H\_

#define MAX\_BUFFER\_SIZE 64

#include<string>

using namespace std;

class I2C

{

private:

int File;

public:

I2C();

~I2C();

//public buffers for read and write

unsigned char rbuff[MAX\_BUFFER\_SIZE];

unsigned char wbuff[MAX\_BUFFER\_SIZE];

//Initialize Functions

void Open(string path);

void Close();

void SetAddress(unsigned char address);

//Read Data from Device on the register

unsigned char Read(unsigned char address, unsigned char reg);

unsigned char Read(unsigned char address, unsigned char reg, int n);

//Write Data to Device on the register

void Write(unsigned char address, unsigned char reg, unsigned char data);

};

#endif

#include "I2C.h"

#include <linux/i2c-dev.h>

#include <sys/ioctl.h>

#include <fcntl.h>

#include <unistd.h>

#include <cstdlib>

#include <cstdio>

#include <iostream>

using namespace std;

I2C::I2C()

{

}

I2C::~I2C()

{

this->Close();

}

void I2C::Open(string path)

{

File = open(path.c\_str(), O\_RDWR);

if(File < 0)

{

cerr << "The file is unable to open !" << endl;

exit(1);

}

else

{

cout << "The file has loaded !" << endl;

}

}

void I2C::Close()

{

close(File);

}

void I2C::SetAddress(unsigned char address)

{

if( ioctl(File, I2C\_SLAVE, address) < 0 )

{

cerr << "No device is founded !" << endl;

exit(1);

}

}

unsigned char I2C::Read(unsigned char address, unsigned char reg)

{

wbuff[0] = reg;

SetAddress(address);

if( write(File, wbuff, 1) != 1 )

{

cerr << " Write Error !" << endl;

}

SetAddress(address);

if( read(File, rbuff, 1) != 1)

{

cerr << "Read Error !" << endl;

}

return rbuff[0];

}

void I2C::Write(unsigned char address, unsigned char reg, unsigned char data)

{

SetAddress(address);

wbuff[0] = reg;

wbuff[1] = data;

if( write(File, wbuff, 2) != 2)

{

cerr << "Write Error !" << endl;

}

}

unsigned char I2C::Read(unsigned char address, unsigned char reg, int n)

{

wbuff[0] = reg;

SetAddress(address);

if( write(File, wbuff, 1) != 1 )

{

cerr << " Write Error !" << endl;

}

SetAddress(address);

if( read(File, rbuff, n) != n )

{

cerr << "Read Error !" << endl;

}

return rbuff[0];

}

#ifndef \_IMU\_H\_

#define \_IMU\_H\_

#include "SENSOR.h"

using namespace std;

// LSM303 accelerometer: 8 g sensitivity

// 3.9 mg/digit; 1 g = 256

#define GRAVITY 256 //this equivalent to 1G in the raw data coming from the accelerometer

#define ToRad(x) ((x)\*0.01745329252) // \*pi/180

#define ToDeg(x) ((x)\*57.2957795131) // \*180/pi

// L3G4200D gyro: 2000 dps full scale

// 70 mdps/digit; 1 dps = 0.07

#define Gyro\_Gain\_X 0.07 //X axis Gyro gain

#define Gyro\_Gain\_Y 0.07 //Y axis Gyro gain

#define Gyro\_Gain\_Z 0.07 //Z axis Gyro gain

#define Gyro\_Scaled\_X(x) ((x)\*ToRad(Gyro\_Gain\_X)) //Return the scaled ADC raw data of the gyro in radians for second

#define Gyro\_Scaled\_Y(x) ((x)\*ToRad(Gyro\_Gain\_Y)) //Return the scaled ADC raw data of the gyro in radians for second

#define Gyro\_Scaled\_Z(x) ((x)\*ToRad(Gyro\_Gain\_Z)) //Return the scaled ADC raw data of the gyro in radians for second

class IMU

{

public:

//class members

SENSOR sensor;

SENSOR acc\_sensor;

//sensor sign

int SENSOR\_SIGN[9] = {

1,1,1, //gyro

-1,-1,-1, //accelerometer

1,1,1 //magnetometer

};

float Accel\_Vector[3]= {0,0,0}; //Store the acceleration in a vector

float Gyro\_Vector[3]= {0,0,0};//Store the gyros turn rate in a vector

int AN[6]; //sores the gyro and accelerometer data

int AN\_OFFSET[6] = {0, 0, 0, 0, 0, 0};

//gryo, acc, and mag data

int gyro\_x, gyro\_y, gyro\_z;

int acc\_x, acc\_y, acc\_z;

int mag\_x, mag\_y, mag\_z;

// Integration time (DCM algorithm) run the integration loop at 50Hz

double G\_Dt = 0.02;

//General timer

long long timer = 0;

long long timer\_old;

long long timer24 = 0;

//base number

int base\_z ;

int base\_y ;

//counter

unsigned int counter = 0;

//constructor and destructor

IMU();

~IMU();

void Setup();

//main process

void Run();

//Read data from the sensor

void Read\_Gyro();

void Read\_Acc();

void Read\_Mag();

//Other utilities

void Delay(int ms);

long long millis();

};

#endif

#include "IMU.h"

#include <unistd.h>

#include <iostream>

#include <sys/time.h>

#include <cmath>

#include <cstdio>

#include <string>

#include "udpclient.h"

using namespace std;

IMU::IMU()

{

//Gyroscope + Accelerometer + Magnetometer

sensor.EnableI2C\_1();

sensor.AccInit();

//sensor.MagInit();

//sensor.GyroInit();

//sensor.testacc();//replace gyro

//Accelerometer

acc\_sensor.EnableI2C\_2();

acc\_sensor.newacc();

}

IMU::~IMU()

{

}

void IMU::Read\_Gyro()

{

sensor.ReadGyro();

AN[0] = sensor.g.x;

AN[1] = sensor.g.y;

AN[2] = sensor.g.z;

gyro\_x = SENSOR\_SIGN[0] \* (AN[0] - AN\_OFFSET[0]);

gyro\_y = SENSOR\_SIGN[1] \* (AN[1] - AN\_OFFSET[1]);

gyro\_z = SENSOR\_SIGN[2] \* (AN[2] - AN\_OFFSET[2]);

}

void IMU::Read\_Acc()

{

sensor.ReadAcc();

AN[3] = sensor.a.x >> 4;

AN[4] = sensor.a.y >> 4;

AN[5] = sensor.a.z >> 4;

acc\_x = SENSOR\_SIGN[3] \* (AN[3] - AN\_OFFSET[3]);

acc\_y = SENSOR\_SIGN[4] \* (AN[4] - AN\_OFFSET[4]);

acc\_z = SENSOR\_SIGN[5] \* (AN[5] - AN\_OFFSET[5]);

}

void IMU::Read\_Mag()

{

sensor.ReadMag();

mag\_x = SENSOR\_SIGN[6] \* sensor.m.x;

mag\_y = SENSOR\_SIGN[7] \* sensor.m.y;

mag\_z = SENSOR\_SIGN[8] \* sensor.m.z;

}

void IMU::Delay(int ms)

{

usleep( (ms\*1000) );

}

long long IMU::millis()

{

struct timeval te;

gettimeofday(&te, NULL); // get current time

long long milliseconds = te.tv\_sec\*1000LL + te.tv\_usec/1000; // caculate

return milliseconds;

}

void IMU::Setup()

{

Delay(20);

//Take reading

for(int i = 0; i < 32; i++)

{

Read\_Gyro();

Read\_Acc();

//Accumulate all the datas and add them to the offset

for(int j = 0; j < 6; j++)

{

AN\_OFFSET[j] += AN[j];

}

Delay(20);

}

//Get the Average

for(int i = 0; i < 6; i++)

{

AN\_OFFSET[i] = AN\_OFFSET[i] / 32;

}

AN\_OFFSET[5] -= GRAVITY\*SENSOR\_SIGN[5];

//Print Offset

cout << "Offset:" << endl;

for(int i = 0; i < 6; i++)

{

cout << AN\_OFFSET[i] << " ";

}

cout << endl;

//Regulating the timer and counter

Delay(2000);

timer = millis();

Delay(20);

base\_z = 2;

base\_y = 7;

}

void IMU::Run()

{

/\*

\* How to use the IMU.cpp

\* Your new sensor is "acc\_sensor"

\* you can read the acceleration from the new sensor

\* by calling the "acc\_sensor.ReadAcc()"

\* then you can obtain the acceleration from acc\_sensor.a.x

\* , acc\_sensor.a.y and acc\_sensor.a.z

\*

\*

\* variables screen\_x and screen\_y are the centrer value of

\* Your screen. Make-sure you change that to your local screen

\* size.

\*

\* Look down below, you will notice that a "Temp" variable is

\* comparing with a number 1280 and 1024. This is my local

\* maximum screen width and height.

\*

\*

\*/

timer = millis();

int global\_x = 0, global\_y = 0;

int left\_click = 0;

string sdata;

udpclient client=udpclient();

while(true)

{

//runs at 50Hz

if( (millis() - timer) >= 0 )

{

sdata.clear();

timer = millis();

acc\_sensor.ReadNewAcc();//new sensor

//Read\_Gyro();

Read\_Acc();

//Read\_Mag();

//global\_x = gyro\_z - base\_z;

//global\_y = gyro\_y - base\_z;

//sdata.append(to\_string(global\_x));

//sdata.append(" ");

//sdata.append(to\_string(global\_y));

//sdata.append(" ");

sdata.append(to\_string( sensor.a.x ));

sdata.append(" ");

sdata.append(to\_string( sensor.a.y));

sdata.append(" ");

sdata.append(to\_string(sensor.a.z));

sdata.append(" ");

sdata.append(to\_string(acc\_sensor.a.x));

sdata.append(" ");

sdata.append(to\_string(acc\_sensor.a.y));

sdata.append(" ");

sdata.append(to\_string(acc\_sensor.a.z));

cout<<sdata<<endl;

client.udpsend(sdata);

}

}

}

#ifndef SENSOR\_H\_

#define SENSOR\_H\_

#include "I2C.h"

#include <cstdint>

//I2C Device

#define LSM303 0x1D

#define L3GD20H 0x6B

#define ACCELERATE 0x1E

//Magnetic

#define LSM\_STATUS\_M 0x07

#define LSM\_OUT\_X\_L\_M 0x08

#define LSM\_OUT\_X\_H\_M 0x09

#define LSM\_OUT\_Y\_L\_M 0x0A

#define LSM\_OUT\_Y\_H\_M 0x0B

#define LSM\_OUT\_Z\_L\_M 0x0C

#define LSM\_OUT\_Z\_H\_M 0x0D

//Acceleration

#define LSM\_STATUS\_A 0x27

#define LSM\_OUT\_X\_L\_A 0x28

#define LSM\_OUT\_X\_H\_A 0x29

#define LSM\_OUT\_Y\_L\_A 0x2A

#define LSM\_OUT\_Y\_H\_A 0x2B

#define LSM\_OUT\_Z\_L\_A 0x2C

#define LSM\_OUT\_Z\_H\_A 0x2D

//Controls

#define LSM\_CTRL0 0x1F

#define LSM\_CTRL1 0x20

#define LSM\_CTRL2 0x21

#define LSM\_CTRL3 0x22

#define LSM\_CTRL4 0x23

#define LSM\_CTRL5 0x24

#define LSM\_CTRL6 0x25

#define LSM\_CTRL7 0x26

//Controls

#define L3G\_CTRL\_REG1 0x20

#define L3G\_CTRL\_REG2 0x21

#define L3G\_CTRL\_REG3 0x22

#define L3G\_CTRL\_REG4 0x23

#define L3G\_CTRL\_REG5 0x24

#define L3G\_REFERENCE 0x25

#define L3G\_OUT\_TEMP 0x26

#define L3G\_STATUS\_REG 0x27

//Gyroscope

#define L3G\_OUT\_X\_L 0x28

#define L3G\_OUT\_X\_H 0x29

#define L3G\_OUT\_Y\_L 0x2A

#define L3G\_OUT\_Y\_H 0x2B

#define L3G\_OUT\_Z\_L 0x2C

#define L3G\_OUT\_Z\_H 0x2D

using namespace std;

template <typename T> struct vector3D

{

T x, y, z;

};

class SENSOR

{

private:

I2C \*i2c;

public:

vector3D<int16\_t> a;

vector3D<int16\_t> m;

vector3D<float>g;

SENSOR();

~SENSOR();

void EnableI2C\_2();

void EnableI2C\_1();

void newacc();

void AccInit();

void MagInit();

void GyroInit();

void ReadAcc();

void ReadMag();

void ReadGyro();

void ReadNewAcc();

};

#endif

#include "SENSOR.h"

#include <iostream>

#include <string>

using namespace std;

SENSOR::SENSOR()

{

i2c = new I2C();

}

void SENSOR::EnableI2C\_2()

{

i2c->Open("/dev/i2c-2");

}

void SENSOR::EnableI2C\_1()

{

i2c->Open("/dev/i2c-1");

}

SENSOR::~SENSOR()

{

delete i2c;

i2c = NULL;

}

void SENSOR::AccInit()

{

//Accelerometer initilization

i2c->Write(LSM303, LSM\_CTRL1, 0x57);

i2c->Write(LSM303, LSM\_CTRL2, 0x18);

}

void SENSOR::newacc()

{

i2c->Write(ACCELERATE, LSM\_CTRL1, 0x57);

i2c->Write(ACCELERATE, LSM\_CTRL2, 0x18);

}

void SENSOR::MagInit()

{

//Magnetometer initilization

i2c->Write(LSM303, LSM\_CTRL5, 0x64);

i2c->Write(LSM303, LSM\_CTRL6, 0x20);

i2c->Write(LSM303, LSM\_CTRL7, 0x00);

}

void SENSOR::GyroInit()

{

//Gyroscope initilization

i2c->Write(L3GD20H, L3G\_CTRL\_REG1, 0x0F);

i2c->Write(L3GD20H, L3G\_CTRL\_REG4, 0x20);

}

void SENSOR::ReadAcc()

{

a.x = (int16\_t)( i2c->Read(LSM303, LSM\_OUT\_X\_H\_A) << 8 | i2c->Read(LSM303, LSM\_OUT\_X\_L\_A) );

a.y = (int16\_t)( i2c->Read(LSM303, LSM\_OUT\_Y\_H\_A) << 8 | i2c->Read(LSM303, LSM\_OUT\_Y\_L\_A) );

a.z = (int16\_t)( i2c->Read(LSM303, LSM\_OUT\_Z\_H\_A) << 8 | i2c->Read(LSM303, LSM\_OUT\_Z\_L\_A) );

}

void SENSOR::ReadNewAcc()

{

a.x = (int16\_t)( i2c->Read(ACCELERATE, LSM\_OUT\_X\_H\_A) << 8 | i2c->Read(ACCELERATE, LSM\_OUT\_X\_L\_A) );

a.y = (int16\_t)( i2c->Read(ACCELERATE, LSM\_OUT\_Y\_H\_A) << 8 | i2c->Read(ACCELERATE, LSM\_OUT\_Y\_L\_A) );

a.z = (int16\_t)( i2c->Read(ACCELERATE, LSM\_OUT\_Z\_H\_A) << 8 | i2c->Read(ACCELERATE, LSM\_OUT\_Z\_L\_A) );

}

void SENSOR::ReadMag()

{

m.x = (int16\_t)( i2c->Read(LSM303, LSM\_OUT\_X\_H\_M) << 8 | i2c->Read(LSM303, LSM\_OUT\_X\_L\_M) );

m.y = (int16\_t)( i2c->Read(LSM303, LSM\_OUT\_Y\_H\_M) << 8 | i2c->Read(LSM303, LSM\_OUT\_Y\_L\_M) );

m.z = (int16\_t)( i2c->Read(LSM303, LSM\_OUT\_Z\_H\_M) << 8 | i2c->Read(LSM303, LSM\_OUT\_Z\_L\_M) );

}

void SENSOR::ReadGyro()

{

g.x = (int16\_t)( i2c->Read(L3GD20H, L3G\_OUT\_X\_H) << 8 | i2c->Read(L3GD20H, L3G\_OUT\_X\_L) );

g.y = (int16\_t)( i2c->Read(L3GD20H, L3G\_OUT\_Y\_H) << 8 | i2c->Read(L3GD20H, L3G\_OUT\_Y\_L) );

g.z = (int16\_t)( i2c->Read(L3GD20H, L3G\_OUT\_Z\_H) << 8 | i2c->Read(L3GD20H, L3G\_OUT\_Z\_L) );

}

#ifndef \_UDPCLIENT\_H\_

#define \_UDPCLIENT\_H\_

#include <sys/socket.h>

#include <netinet/in.h> /\* needed for sockaddr\_in \*/

#include <arpa/inet.h>

#include <unistd.h>

#include <string>

#include <string.h>

using namespace std;

#define SEND\_IP "192.168.7.1"

#define BUFLEN 100

#define SEND\_PORT 54321

class udpclient

{

private:

int fd;

struct sockaddr\_in myaddr,youraddr;

char buf[BUFLEN];

public:

udpclient();

void udpsend(string data);

};

#endif

#include "udpclient.h"

#include <iostream>

#include <stdlib.h> /\* defines system calls \*/

#include <stdio.h> /\* needed for printf \*/

using namespace std;

udpclient::udpclient()

{

if ((fd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("cannot create receive socket");

}

memset((void \*)&myaddr, 0, sizeof(myaddr));

memset((void \*)&youraddr,0,sizeof(youraddr));

myaddr.sin\_family = AF\_INET;

youraddr.sin\_family = AF\_INET;

myaddr.sin\_addr.s\_addr = inet\_addr("192.168.7.2");

if(inet\_aton(SEND\_IP,&youraddr.sin\_addr)==0)

{cout<<"address wrong"<<endl;exit(1);}

myaddr.sin\_port = htons(12345);

youraddr.sin\_port=htons(SEND\_PORT);

if (bind(fd, (struct sockaddr \*)&myaddr, sizeof(myaddr)) < 0) {

perror("receive bind failed");

}

}

void udpclient::udpsend(string data)

{

int len;

int slen=sizeof(youraddr);

strcpy(buf,data.c\_str());

len=strlen(buf);

buf[len]='\0';

if (sendto(fd, buf, len, 0, (struct sockaddr\*)&youraddr, slen)==-1)

{printf("sendto() fail\n");}

}

#include "IMU.h"

//#include "stdmath.h"

#include <iostream>

using namespace std;

int main(int argc, char\* argv[])

{

cout << "Running.." << endl;

IMU imu9;

imu9.Setup();

imu9.Run();

return 0;

}