Design Lab Experiment No. 2:

Motion Sensing System Implementation using Raspberry Pi

By: Alan Palayil

Lab Partner: Individual

Instructor: Prof. Jafar Saniie, Prof. Won-Jae Yi

TA: Mikhail Gromov

ECE 442 Internet of Things and Cyber Physical Systems

Date: 06-03-2022

Acknowledgment: I acknowledge all works including figures, codes and writings belong to me and/or persons who are referenced. I understand if any similarity in the code, comments, customized program behavior, report writings and/or figures are found, both the helper (original work) and the requestor (duplicated/modified work) will be called for academic disciplinary action.

Signature: ___Alan Biju Palayil_____

I. Introduction

A. Purpose

In this lab, we are building a motion sensing system using a Raspberry Pi single-board computer, where the sensing data are transmitted to a server through a client-server communication. Using a UDP connection between the server and Raspberry Pi, encrypt data from the Pi and decrypt the data in the server and have it displayed graphically.

The purpose of this exercise is to introduce different methods of communications between a microcomputer and I/O devices using Raspberry Pi, AES symmetric key encryption, and a client-server UDP connection.

B. Background

This experiment is divided into two parts:

1. Implementation of real-time motion sensing device using I2C protocol:
   I2C is used to transfer data between master and slave devices. It uses two pins SDA and SCL. With the SCL, clock line to synchronize all data transfer over the bus and SDA, data line used to specify the addresses and transfer data. The Pi acts as the master and accelerometer as slave. The slave responds to the request of the master and the Pi requests the readings of the axes as an input data to be stored and displayed on the Pi.
2. Implementation of real-time motion sensing device using SPI protocol and establishing server-client communication:
   The concept of SPI transfers data between master and slave devices. It uses 3-4 wires for the connections. SCK, MOSI, MISO, and SS are used. The Master initiates the communication by SS of slave to low. 1st byte is sent by the master with MSB for writing. In 2nd byte, slave will respond to master by sending the data from register on to master. The Pi will initiate the communication and act as master while the slave accelerometer will send the data to the master.
   This data is sent using the server-client communication to the server. The PC acts as the server and Pi acts as the client. Using AES, the data is encrypted and

transmitted over the internet. The server decrypts the data using the same key as encryption and uses it.

II. Lab Procedure and Equipment List

A. Equipment

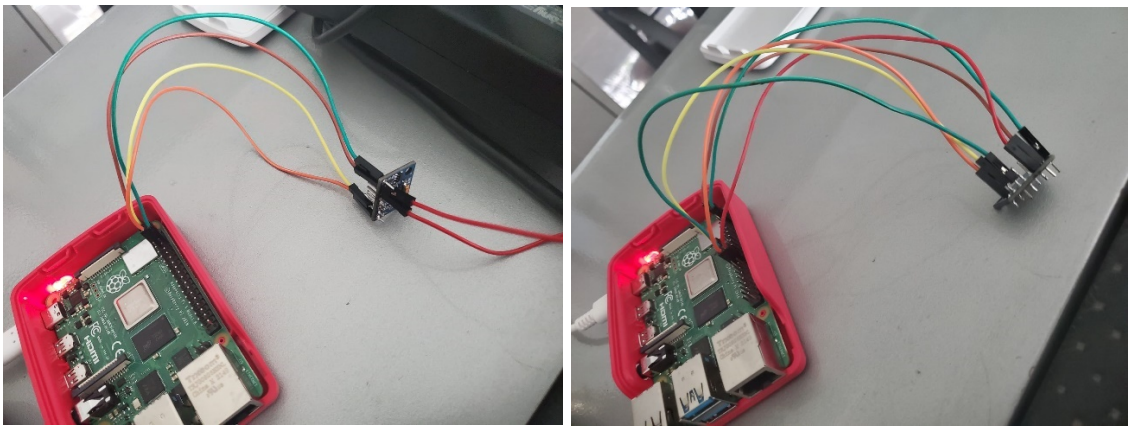
- 1 x Raspberry Pi 4 single-board computer
- 1 x Adafruit ADXL345 triple-axis accelerometer
- 1 x Desktop (as a data storage server)
- Wires

B. Procedure

Refer to Lab Manual for detailed Procedure

III. Results and Analysis

A. Results



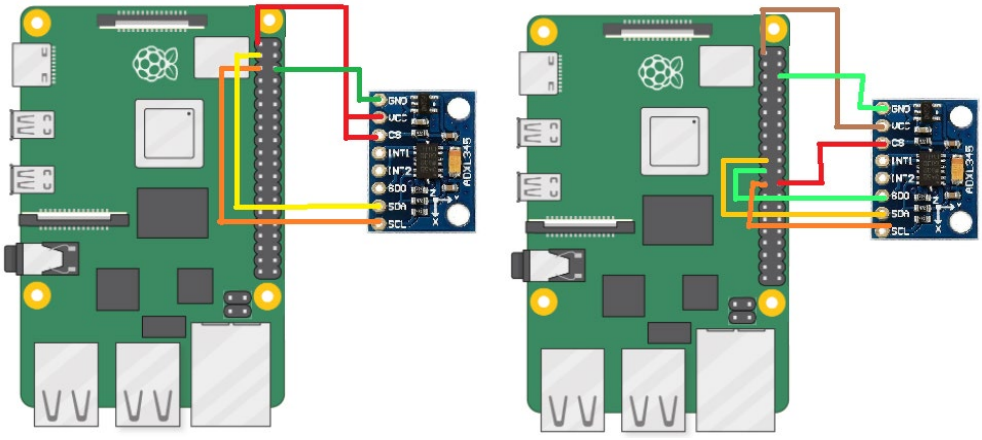
Picture of the wiring for Part A and Part B.

From the experiment, the output received to the server are as expected with no errors found while performing the experiment. The accelerometer was sending the axes data to the Pi and then to the serve. The server received the values in real time and displayed the values in a graph format with some delay.

B. Answers to Discussion Questions

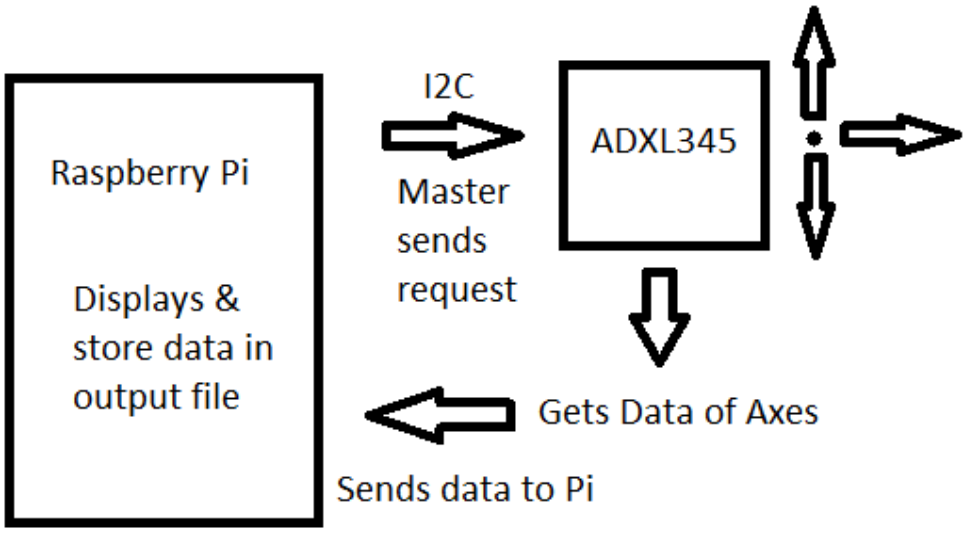
1. Refer to appendix A for the fully commented codes.

   Schematics for Part A and Part B:



2. The Raspberry Pi acts as the master using the I2C connection send the request to slave accelerometer to transmit data. The accelerometer SCL clock line takes care of the synchronization between Pi and accelerometer. The SDA data line sends the data to the Pi. During the request of the master, it sends the coordinates axes data to Pi. The Pi displays it on the screen and stores the values

   Flowchart for I2C operation.



3. For an I2C type of connection. The SDA pin of the slave device is used to transmit the data. If more than one slave devices are connected than SDO pin of the other accelerometer acts as the alternate pin for the I2C protocol. The Raspberry Pi chooses the slave devices using these pins that send data to it.

   For a SPI type of connection, all other connections can work cohesively between two or more slaves with the Raspberry Pi. The CS is different for different slave sensors and is connected to different pin on the Pi. Thus, making the Pi easily differentiate between two slave devices using which CS pin is connected.

   To differentiate between the two connections, the Raspberry Pi can determine it because the pins used for either of the connections are different. SPI uses the Pi's MISO, MOSI pins while I2C uses SCL and SDA pins. The protocol being different for the two connections, the Pi can identify the slaves whether the slaves are connected to I2C or SPI.

4. The register 0x31 is for data format control, to read or write. The 0x2F sends the Interrupt mapping control. Any bits set to 0 in 0x2F sends their respective interrupts from the 0x31 to the INT1 pin, while bits set to 1 in 0x2F sends their respective interrupts from the 0x31 to the INT2 pin.

5. There is a delay in the data received from the physical accelerometer between the output of the Raspberry Pi and the server side. This is because of two main reasons: first is because the data is received from the accelerometer to the Raspberry Pi unencrypted and the second being that the data is processed to deliver the data in graph format. The data received to the Pi direct and sorted into the file while the data is encrypted before it is sent to the master PC server side. The server side decrypts the data and processes the data to display it into a graph using MatLab. I believe to improve the real-time data on the server side, clean up the existing code a decrease the delay time (sleep function).

6. There are different sampling rates according to different peripheral for the Pi from 100kHz to 400kHz. Since the maximum sampling rate in theory is 400kHz, the I2C bus bandwidth would be 400000/27=14814 samples per second. To transfer 2 bytes through bus needs at least 27-bits (address + 2 data bytes). For the maximum clock rate is 3.6 MHz when running at 5V. The device requires 18 clocks per data samples, so 3.6 MHz/18=200k samples per second. Sampling Rate for SPI=3.6 MHz/24=150ksps.

   To achieve the maximum theoretical sampling rate the complexity of the code needs to take into consideration, check over the timings, look over idle gaps between pollings, and check for unexpected communications or activity on the bus with the accelerometer.

IV. Conclusion

The experiment proves that the method mentioned above is good for sending sensor data and dependable too. It can be accessed from the server and then the data can be used for manipulation and controlling other sensors and devices. This serves the purpose of internet of things.

References

[1] Experiment 2 Lab Manual

[2] Lecture Note 3 of ECE 442/510

Appendix

Appendix A

1. *Main.cpp*

   /**********

   Use example and information in Appendix A in lab instruction manual to finish this file

   **********/

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include "i2c-dev.h"
#include "ADXL345.h"
#define I2C_FILE_NAME "/dev/i2c-1"
void INThandler (int sig);
int main(int argc, char **argv)
{
	int i2c_fd=open(I2C_FILE_NAME, O_RDWR);
	if (i2c_fd<0)
	{
		printf("Unable to open i2c control file, err=%d\n",i2c_fd);
		exit(1);
	}
	printf("Opened i2c control file, id=%d\n",i2c_fd);
	ADXL345 myAcc(i2c_fd);
	int ret=myAcc.init();
	if (ret)
	{
		printf("fialed init ADXL345, ret=%d\n", ret);
		exit(1);
	}
	usleep(100 * 1000);
	signal(SIGINT, INThandler);
	short ax, ay, az;
	//create file IO
	FILE *fp;
	fp = fopen("./output.txt","w+"); //creating an output file to store the data
	char TimeString[128];
	timeval curTime;
```

```
        while(1)
        {
                // get the current time
                gettimeofday(&curTime, NULL);
                strftime(TimeString,80,"%Y-%m-%d
%H:%M:%S",localtime(&curTime.tv_sec));
                printf(TimeString);
                printf(": ");
                // now, fetch data from sensor
                myAcc.readXYZ(ax, ay, az);
                //print to screen
                printf("Ax : %hi \t Ay : %hi \t Az : %hi \n", ax,ay,az);
                printf("----------------------\n");
                //print to file
                fprintf(fp,TimeString);
                fprintf(fp, ": ");
                fprintf(fp, "Ax : %hi \t Ay : %hi \t Az : %hi \n", ax,ay,az);
                fprintf(fp, "----------------------\n");
                if (getchar() == 'q') break;
        }
        fclose(fp);
        return 0;
}
void INThandler(int sig)
{
        signal(sig, SIG_IGN);
        exit(0);
}
```

2. *ADXL345.cpp*

```
/*
```

```
Basic readout of ADXL345 accelerometer via I2C
Original code taken from the very bottom of this page:
http://www.raspberrypi.org/forums/viewtopic.php?t=55834
Updated by Jan Balewski, August 2014
*/
#include <assert.h>
#include "ADXL345.h"
//=============================================
//=============================================
bool ADXL345::selectDevice(){
 if (ioctl(fd, I2C_SLAVE, myAddr) < 0) {
      fprintf(stderr, "device ADXL345 not present\n");
      return false;
   }
   return true;
}
//=============================================
//=============================================
bool ADXL345::writeToDevice(char * buf, int len){
   if (write(fd, buf, len) != len)   {
     fprintf(stderr, "Can't write to device ADXL345 buf=%s len=%d\n",buf,len);
     return false;
   }
  return true;
}
//===============
bool  ADXL345::readXYZ( short &x , short &y, short &z) {
  assert(fd>0); // crash if port was not opened earlier
  if(!selectDevice())   return false;
  //   printf("selectDevice(fd,ADXL345...)  passed\n");
  char buf[7];
```

```
  buf[0] = 0x32;     // This is the register we wish to read from
  if(!writeToDevice(buf,2))     return false;
  if (read(fd, buf, 6) != 6) {  // Read back data into buf[]
    printf("Unable to read from slave for ADXL345\n");
    return false;
  }  else {
    x = (buf[1]<<8) |  buf[0];
    y = (buf[3]<<8) |  buf[2];
    z = (buf[5]<<8) |  buf[4];
  }
  return true;
}
//=============================================
//=============================================
//=============================================
int ADXL345::init()  {
  assert(fd>0); // crash if port was not opened earlier
  char buf[6];       // Buffer for data being read/ written on the i2c bus
  if(!selectDevice()) return -1;
  buf[0] = 0x2d;                  // Commands for performing a ranging
  buf[1] = 0x18;
  if(!writeToDevice(buf,2))  return -2;
   buf[0] = 0x31;              // Commands for performing a ranging
   buf[1] = 0x0A; //09 4g , A 8g
   if(!writeToDevice(buf,2))  return -3;
   printf("ADXL345::init() OK\n");
   return 0;
}
```

3. *ADXL345.h*

```
/*
```

```
Basic readout of ADXL345 accelerometer via I2C bus
Oryginal code taken from the very bottom of this page:
http://www.raspberrypi.org/forums/viewtopic.php?t=55834
Updated by Jan Balewski, August 2014
*/
#ifndef ADXL345_HH
#define ADXL345_HH
#include <stdio.h>
#include <stdlib.h>
#include "i2c-dev.h"
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <iostream>
class ADXL345 {
 public:
  ADXL345 (int fdx, unsigned char addx=0x53) { fd=fdx; myAddr=addx;}
  int init();
  bool readXYZ(short &ax , short &ay, short &az);
 private:
  bool selectDevice();
  bool writeToDevice(char * buf, int len);
  unsigned char myAddr;
  int fd;// File descriptor
};
#endif
```

4. *I2c-dev.h*

```
/*
```

```
   i2c-dev.h - i2c-bus driver, char device interface
   Copyright (C) 1995-97 Simon G. Vogl
   Copyright (C) 1998-99 Frodo Looijaard <frodol@dds.nl>
   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.
   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
   GNU General Public License for more details.
   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
   MA 02110-1301 USA.
*/
/* $Id: i2c-dev.h 5894 2010-12-12 13:22:29Z khali $ */
#ifndef LIB_I2CDEV_H
#define LIB_I2CDEV_H
#include <linux/types.h>
#include <sys/ioctl.h>
//#include <linux/i2c-dev.h>
/* -- i2c.h -- */
/*
 * I2C Message - used for pure i2c transaction, also from /dev interface
 */
struct i2c_msg {
        __u16 addr;     /* slave address                        */
        unsigned short flags;
#define I2C_M_TEN  0x10    /* we have a ten-bit chip address       */
#define I2C_M_RD    0x01
```

```
#define I2C_M_NOSTART   0x4000
#define I2C_M_REV_DIR_ADDR   0x2000
#define I2C_M_IGNORE_NAK       0x1000
#define I2C_M_NO_RD_ACK                0x0800
        short len;              /* msg length                   */
        char *buf;              /* pointer to msg data          */
};
/* To determine what functionality is present */
#define I2C_FUNC_I2C                    0x00000001
#define I2C_FUNC_10BIT_ADDR             0x00000002
#define I2C_FUNC_PROTOCOL_MANGLING      0x00000004                      /*
I2C_M_{REV_DIR_ADDR,NOSTART,..} */
#define I2C_FUNC_SMBUS_PEC              0x00000008
#define I2C_FUNC_SMBUS_BLOCK_PROC_CALL      0x00008000 /* SMBus 2.0 */
#define I2C_FUNC_SMBUS_QUICK                0x00010000
#define I2C_FUNC_SMBUS_READ_BYTE            0x00020000
#define I2C_FUNC_SMBUS_WRITE_BYTE           0x00040000
#define I2C_FUNC_SMBUS_READ_BYTE_DATA       0x00080000
#define I2C_FUNC_SMBUS_WRITE_BYTE_DATA      0x00100000
#define I2C_FUNC_SMBUS_READ_WORD_DATA       0x00200000
#define I2C_FUNC_SMBUS_WRITE_WORD_DATA      0x00400000
#define I2C_FUNC_SMBUS_PROC_CALL            0x00800000
#define I2C_FUNC_SMBUS_READ_BLOCK_DATA      0x01000000
#define I2C_FUNC_SMBUS_WRITE_BLOCK_DATA 0x02000000
#define I2C_FUNC_SMBUS_READ_I2C_BLOCK0x04000000 /* I2C-like block xfer  */
#define I2C_FUNC_SMBUS_WRITE_I2C_BLOCK      0x08000000 /* w/ 1-byte reg.
addr. */
#define I2C_FUNC_SMBUS_BYTE (I2C_FUNC_SMBUS_READ_BYTE | \
                     I2C_FUNC_SMBUS_WRITE_BYTE)
#define                                         I2C_FUNC_SMBUS_BYTE_DATA
(I2C_FUNC_SMBUS_READ_BYTE_DATA | \
```

```
                        I2C_FUNC_SMBUS_WRITE_BYTE_DATA)
#define                                         I2C_FUNC_SMBUS_WORD_DATA
(I2C_FUNC_SMBUS_READ_WORD_DATA | \
                        I2C_FUNC_SMBUS_WRITE_WORD_DATA)
#define                                         I2C_FUNC_SMBUS_BLOCK_DATA
(I2C_FUNC_SMBUS_READ_BLOCK_DATA | \
                         I2C_FUNC_SMBUS_WRITE_BLOCK_DATA)
#define I2C_FUNC_SMBUS_I2C_BLOCK (I2C_FUNC_SMBUS_READ_I2C_BLOCK
| \
                        I2C_FUNC_SMBUS_WRITE_I2C_BLOCK)
/* Old name, for compatibility */
#define I2C_FUNC_SMBUS_HWPEC_CALC       I2C_FUNC_SMBUS_PEC
/*
 * Data for SMBus Messages
 */
#define I2C_SMBUS_BLOCK_MAX         32      /* As specified in SMBus standard */
#define I2C_SMBUS_I2C_BLOCK_MAX  32      /* Not specified but we use same
structure */
union i2c_smbus_data {
        __u8 byte;
        __u16 word;
        __u8 block[I2C_SMBUS_BLOCK_MAX + 2]; /* block[0] is used for length */
                                        /* and one more for PEC */
};
/* smbus_access read or write markers */
#define I2C_SMBUS_READ 1
#define I2C_SMBUS_WRITE         0
/* SMBus transaction types (size parameter in the above functions)
  Note: these no longer correspond to the (arbitrary) PIIX4 internal codes! */
#define I2C_SMBUS_QUICK                     0
#define I2C_SMBUS_BYTE              1
```

```
#define I2C_SMBUS_BYTE_DATA             2
#define I2C_SMBUS_WORD_DATA             3
#define I2C_SMBUS_PROC_CALL    4
#define I2C_SMBUS_BLOCK_DATA            5
#define I2C_SMBUS_I2C_BLOCK_BROKEN  6
#define I2C_SMBUS_BLOCK_PROC_CALL   7           /* SMBus 2.0 */
#define I2C_SMBUS_I2C_BLOCK_DATA    8
/* ----- commands for the ioctl like i2c_command call:
 * note that additional calls are defined in the algorithm and hw
 *      dependent layers - these can be listed here, or see the
 *      corresponding header files.
 */
                                /* -> bit-adapter specific ioctls       */
#define I2C_RETRIES             0x0701          /* number of times a device address      */
                                /* should be polled when not            */
                        /* acknowledging                                */
#define I2C_TIMEOUT             0x0702          /* set timeout - call with int           */
/* this is for i2c-dev.c */
#define I2C_SLAVE  0x0703               /* Change slave address                 */
                                /* Attn.: Slave address is 7 or 10 bits */
#define I2C_SLAVE_FORCE         0x0706          /* Change slave address
        */
                                /* Attn.: Slave address is 7 or 10 bits */
                                /* This changes the address, even if it */
                                /* is already taken!                    */
#define I2C_TENBIT 0x0704               /* 0 for 7-bit addrs, != 0 for 10 bits    */
#define I2C_FUNCS  0x0705               /* Get the adapter functionality */
#define I2C_RDWR   0x0707               /* Combined R/W transfer (one stop only)*/
#define I2C_PEC                 0x0708          /* != 0 for SMBus PEC                */
#define I2C_SMBUS  0x0720               /* SMBus-level access */
/* -- i2c.h -- */
```

```
/* Note: 10-bit addresses are NOT supported! */
/* This is the structure as used in the I2C_SMBUS ioctl call */
struct i2c_smbus_ioctl_data {
        char read_write;
        __u8 command;
        int size;
        union i2c_smbus_data *data;
};
/* This is the structure as used in the I2C_RDWR ioctl call */
struct i2c_rdwr_ioctl_data {
        struct i2c_msg *msgs;/* pointers to i2c_msgs */
        int nmsgs;                  /* number of i2c_msgs */
};
static inline __s32 i2c_smbus_access(int file, char read_write, __u8 command,
                                     int size, union i2c_smbus_data *data)
{
        struct i2c_smbus_ioctl_data args;
        args.read_write = read_write;
        args.command = command;
        args.size = size;
        args.data = data;
        return ioctl(file,I2C_SMBUS,&args);
}
static inline __s32 i2c_smbus_write_quick(int file, __u8 value)
{
        return i2c_smbus_access(file,value,0,I2C_SMBUS_QUICK,NULL);
}
static inline __s32 i2c_smbus_read_byte(int file)
{
        union i2c_smbus_data data;
        if (i2c_smbus_access(file,I2C_SMBUS_READ,0,I2C_SMBUS_BYTE,&data))
```

```
                return -1;
        else
                return 0x0FF & data.byte;
}
static inline __s32 i2c_smbus_write_byte(int file, __u8 value)
{
return i2c_smbus_access(file,I2C_SMBUS_WRITE,value,
I2C_SMBUS_BYTE,NULL);
}
static inline __s32 i2c_smbus_read_byte_data(int file, __u8 command)
{
union i2c_smbus_data data;
if (i2c_smbus_access(file,I2C_SMBUS_READ,command,
I2C_SMBUS_BYTE_DATA,&data))
return -1;
else
return 0x0FF & data.byte;
}
static inline __s32 i2c_smbus_write_byte_data(int file, __u8 command,
                                __u8 value)
{
union i2c_smbus_data data;
data.byte = value;
return i2c_smbus_access(file,I2C_SMBUS_WRITE,command,
I2C_SMBUS_BYTE_DATA, &data);
}
static inline __s32 i2c_smbus_read_word_data(int file, __u8 command)
{
union i2c_smbus_data data;
if (i2c_smbus_access(file,I2C_SMBUS_READ,command,
I2C_SMBUS_WORD_DATA,&data))
```

```
return -1;
else
return 0x0FFFF & data.word;
}
static inline __s32 i2c_smbus_write_word_data(int file, __u8 command,
                                              __u16 value)
{
union i2c_smbus_data data;
data.word = value;
return i2c_smbus_access(file,I2C_SMBUS_WRITE,command,
I2C_SMBUS_WORD_DATA, &data);
}
static inline __s32 i2c_smbus_process_call(int file, __u8 command, __u16 value)
{
union i2c_smbus_data data;
data.word = value;
if (i2c_smbus_access(file,I2C_SMBUS_WRITE,command,
I2C_SMBUS_PROC_CALL,&data))
return -1;
else
return 0x0FFFF & data.word;
}
/* Returns the number of read bytes */
static inline __s32 i2c_smbus_read_block_data(int file, __u8 command,
                                              __u8 *values)
{
union i2c_smbus_data data;
int i;
if (i2c_smbus_access(file,I2C_SMBUS_READ,command,
I2C_SMBUS_BLOCK_DATA,&data))
return -1;
```

```
else {
for (i = 1; i <= data.block[0]; i++)
values[i-1] = data.block[i];
return data.block[0];
}
}
static inline __s32 i2c_smbus_write_block_data(int file, __u8 command,
                                               __u8 length, const __u8 *values)
{
union i2c_smbus_data data;
int i;
if (length > 32)
length = 32;
for (i = 1; i <= length; i++)
data.block[i] = values[i-1];
data.block[0] = length;
return i2c_smbus_access(file,I2C_SMBUS_WRITE,command,
I2C_SMBUS_BLOCK_DATA, &data);
}
/* Returns the number of read bytes */
/* Until kernel 2.6.22, the length is hardcoded to 32 bytes. If you ask for less than 32 bytes,
your code will only work with kernels
2.6.23 and later. */
static inline __s32 i2c_smbus_read_i2c_block_data(int file, __u8 command,
                                                  __u8 length, __u8 *values)
{
union i2c_smbus_data data;
int i;
if (length > 32)
length = 32;
data.block[0] = length;
```

```
if (i2c_smbus_access(file,I2C_SMBUS_READ,command,
length == 32 ? I2C_SMBUS_I2C_BLOCK_BROKEN :
I2C_SMBUS_I2C_BLOCK_DATA,&data))
return -1;
else {
for (i = 1; i <= data.block[0]; i++)
values[i-1] = data.block[i];
return data.block[0];
}
}
static inline __s32 i2c_smbus_write_i2c_block_data(int file, __u8 command,
                                                   __u8 length,
                                                   const __u8 *values)
{
union i2c_smbus_data data;
int i;
if (length > 32)
length = 32;
for (i = 1; i <= length; i++)
data.block[i] = values[i-1];
data.block[0] = length;
return i2c_smbus_access(file,I2C_SMBUS_WRITE,command,
I2C_SMBUS_I2C_BLOCK_BROKEN, &data);
}
/* Returns the number of read bytes */
static inline __s32 i2c_smbus_block_process_call(int file, __u8 command,
                                                 __u8 length, __u8 *values)
{
union i2c_smbus_data data;
int i;
if (length > 32)
```

```
length = 32;
for (i = 1; i <= length; i++)
data.block[i] = values[i-1];
data.block[0] = length;
if (i2c_smbus_access(file,I2C_SMBUS_WRITE,command,
I2C_SMBUS_BLOCK_PROC_CALL,&data))
return -1;
else {
for (i = 1; i <= data.block[0]; i++)
values[i-1] = data.block[i];
return data.block[0];
}
}
#endif /* LIB_I2CDEV_H */
```

5. *Output.txt*

```
2022-06-03 04:01:31: Ax : -106      Ay : -220      Az : -71
----------------------
2022-06-03 04:01:45: Ax : -40       Ay : -237      Az : 92
----------------------
2022-06-03 04:01:46: Ax : -38       Ay : -234      Az : 121
----------------------
2022-06-03 04:01:46: Ax : -39       Ay : -243      Az : 114
----------------------
2022-06-03 04:01:47: Ax : -43       Ay : -230      Az : 126
----------------------
2022-06-03 04:01:47: Ax : -38       Ay : -60       Az : 262
----------------------
2022-06-03 04:01:47: Ax : 2   Ay : 30          Az : 254
----------------------
2022-06-03 04:01:48: Ax : 60        Ay : -143      Az : 195
```

2022-06-03 04:01:48: Ax : 49 Ay : 1 Az : 276

2022-06-03 04:01:48: Ax : 35 Ay : 17 Az : 266

2022-06-03 04:01:48: Ax : 33 Ay : 12 Az : 268

2022-06-03 04:01:48: Ax : 22 Ay : 9 Az : 255

2022-06-03 04:01:48: Ax : 22 Ay : 18 Az : 261

2022-06-03 04:01:49: Ax : 33 Ay : 36 Az : 273

2022-06-03 04:01:49: Ax : 32 Ay : 27 Az : 274

2022-06-03 04:01:49: Ax : 30 Ay : 23 Az : 264

2022-06-03 04:01:49: Ax : 28 Ay : 32 Az : 268

2022-06-03 04:01:49: Ax : 18 Ay : 44 Az : 270

2022-06-03 04:01:50: Ax : 24 Ay : 38 Az : 274

2022-06-03 04:01:50: Ax : 17 Ay : 1 Az : 263

2022-06-03 04:01:50: Ax : 68 Ay : -112 Az : 256

2022-06-03 04:01:51: Ax : 40 Ay : -104 Az : 268

2022-06-03 04:01:51: Ax : 7 Ay : -64 Az : 257

2022-06-03 04:01:52: Ax : -26 Ay : -121 Az : 195

2022-06-03 04:01:53: Ax : -54 Ay : -271 Az : 71

Appendix B

1. *Encryption.cpp*

```
#include "encryption.hpp"
int encrypt(const char * key, const char * iv, char * msg, char * msgCiphered)
{
 int key_length, iv_length, msg_length;
 key_length = strlen(key);
 iv_length = strlen(iv);
 msg_length = strlen(msg);
 const EVP_CIPHER *cipher;
 int cipher_key_length, cipher_iv_length;
 cipher = EVP_aes_128_cbc();
 cipher_key_length = EVP_CIPHER_key_length(cipher);
 cipher_iv_length = EVP_CIPHER_iv_length(cipher);
 if (key_length != cipher_key_length) {
  fprintf(stderr, "Error: key length must be %d, %d found\n", cipher_key_length, key_length);
  exit(EXIT_FAILURE);
 }
 if (iv_length != cipher_iv_length) {
  fprintf(stderr, "Error: iv length must be %d, %d found\n", cipher_iv_length, iv_length);
  exit(EXIT_FAILURE);
 }
 EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
 int i, cipher_length, final_length;
 EVP_CIPHER_CTX_init(ctx);
```

```
 EVP_EncryptInit_ex(ctx, cipher, NULL, (unsigned char *)key, (unsigned char *)iv);
 cipher_length = msg_length + EVP_MAX_BLOCK_LENGTH;
 EVP_EncryptUpdate(ctx, (unsigned char *)msgCiphered, &cipher_length, (unsigned char *)msg, msg_length);
 EVP_EncryptFinal_ex(ctx, (unsigned char *)msgCiphered + cipher_length, &final_length);
 EVP_CIPHER_CTX_free(ctx);
 return cipher_length + final_length;
}
```

2. *Encryption.hpp*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/evp.h>
int encrypt(const char * key, const char * iv, char * msg, char * msgCiphered);
```

3. *Client.cpp*

```
/*******************
Use example and information in Appendix B in Lab instruction manual to finish this file
*******************/
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <termios.h>
#include <fcntl.h>
#include <wiringPiSPI.h>
#include <netdb.h>
#include <netinet/in.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include "encryption.hpp"
//specify the size of buffer and port for transmit
#define BUFFERSIZE 64
#define PORTNUMBER 6000
//specify key and IV
const char KEY[]= "3874460957140850";
const char IV[]= "9331626268227018";
//specify the address for server
const char hostname[] = "192.168.1.166";
//specify the channel used for SPI
const int spichannel(0);
void initialSPI()
{
wiringPiSPISetupMode(spichannel, 1000000, 3);
usleep(10);
unsigned char buf[2];
buf[0] = 0x2D;
buf[1]=0x18;
wiringPiSPIDataRW(spichannel, (unsigned char *)&buf, 2);
//std::cout << std::hex << (short)buf[1] << std::end1;
//std::cout << "finished testing" << std::end1;
//configure power
//std::cout << "finished setting up powerctl" << std::end1;
//configure data format (Full_res, left justify, +-2g)
buf[0]= 0x31;
buf[1]= 0x00;
wiringPiSPIDataRW(spichannel, (unsigned char *)&buf, 2);
//std::cout << "finished setting up dataformat" << std::end1;
return;
```

```
}
void readRawXYZ(short &X, short &Y, short &Z)
{
unsigned char txRxData[2];
unsigned char buf[6];
//read data
for (unsigned short i(0); i<6; i++)
{
txRxData[0] = (unsigned char) ((unsigned short)0xB2 +i);
wiringPiSPIDataRW(spichannel, (unsigned char *)&txRxData, 2);
buf[i] = txRxData[i];
}
X = (buf[1] << 8) | buf[0];
Y = (buf[3] << 8) | buf[2];
Z = (buf[5] << 8) | buf[4];
return;
}
void readXYZ(float &X, float &Y,float &Z, const short &scale = 2)
{
short x_raw, y_raw, z_raw;
readRawXYZ(x_raw, y_raw, z_raw);
X = (float) x_raw / 1024 * scale;
Y = (float) y_raw / 1024 * scale;
Z = (float) z_raw / 1024 * scale;
return;
}
//return true when keyboard been pressed, false other wise
bool kbhit(void)
{
struct termios oldt, newt;
int ch;
```

```
int oldf;
tcgetattr(STDIN_FILENO, &oldt);
newt = oldt;
newt.c_lflag &= ~(ICANON | ECHO);
tcsetattr(STDIN_FILENO, TCSANOW, &newt);
oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);
ch = getchar();
tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
fcntl(STDIN_FILENO, F_SETFL, oldf);
if(ch != EOF)
{
ungetc(ch, stdin);
return true;
}
return false;
}
int main()
{
initialSPI();
float x, y, z;
//verify the socket;
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if(sockfd <0)
{
printf("ERROR opening socket\n");
exit(1);
}
//verify host name/address
struct hostent *server = gethostbyname(hostname);
if(server == NULL)
```

```
{
printf("ERROR, no such host as %s\n",hostname);
exit(1);
}
//Build the server internet address
struct sockaddr_in serveraddr;
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
bcopy((char *)server->h_addr,(char *)&serveraddr.sin_addr.s_addr, server->h_length);
serveraddr.sin_port = htons(PORTNUMBER);
char bufRaw[BUFFERSIZE], bufCiphered[BUFFERSIZE *2];
while(!kbhit())
{
readXYZ(x,y,z);
//format the buffers with output
//float with 2 and 6 digits before and after decimal point
snprintf(bufRaw, BUFFERSIZE, "%2.6f, %2.6f, %2.6f,\n", x,y,z);
printf(bufRaw);
int length = encrypt((const char *)&KEY, (const char *)&IV, &bufRaw[0], (char
*)&bufCiphered);
for(uint i = 0; i< length; i++)
printf("%02x", bufCiphered[i]);
printf("\n");
int sendStatus = sendto(sockfd, bufCiphered, length, 0, (struct sockaddr *)&serveraddr,
sizeof(serveraddr));
if(sendStatus<0)
{
printf("sent failed with status %d\n", sendStatus);
exit(1);
}
usleep(1000000);
```

```
}
return 0;
}
```

4. *Server.py*

```
#!/usr/bin/env python
import socket
import sys
import datetime
import matplotlib.pyplot as plot
from matplotlib import animation
from Cryptodome.Cipher import AES
# server network configurations
SERVER_IP_ADDRESS = "192.168.1.166" #to be determined
PORT = 6000 #to be determined
time = [0]*50
for i in range(0,50):
                time[i] = i
ax_points = [float(0)]*50
ay_points = [float(0)]*50
az_points = [float(0)]*50
print("starting UDP Server Setup")
sys.stdout.flush()
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind( (SERVER_IP_ADDRESS, PORT) )
print("waiting for data to receive")
sys.stdout.flush()
KEY = b'3874460957140850'
iv = b'9331626268227018'
fig = plot.figure()
ax = plot.axes(xlim=(0, 50), ylim=(-2, 2))
```

```
lineX, lineY, lineZ, = ax.plot([], [], [], [], [], [], lw=2)
def init():
    lineX.set_data([], [])
    lineY.set_data([], [])
    lineZ.set_data([], [])
    return lineX, lineY, lineZ,
def updateData(i):
        decryption_suite = AES.new(KEY, AES.MODE_CBC, IV=iv)
        data, addr = sock.recvfrom(64)
        print (''.join('{:02x}'.format(x) for x in data))
        plain_text = decryption_suite.decrypt(data)
        data = plain_text.decode('utf-8')
        ax,ay,az,dump = data.split(",")
        print("ADXL345 X-Axis: " + ax + "\tY-Axis: " + ay + "\tZ-Axis: " + az + "\t" +
datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f"))
        sys.stdout.flush()
        del ax_points[0]
        del ay_points[0]
        del az_points[0]
        ax_points.append(float(ax))
        ay_points.append(float(ay))
        az_points.append(float(az))
        lineX.set_data(time,ax_points)
        lineY.set_data(time,ay_points)
        lineZ.set_data(time,az_points)
        return lineX, lineY, lineZ,
anim = animation.FuncAnimation(fig, updateData, init_func=init,
                        frames=200, interval=20, blit=True)
plot.show()
```