

EXPERIMENT #8

MOTION SENSING SYSTEM IMPLEMENTATION USING RASPBERRY PI

1.0 Purpose

In this lab, students are required to build 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. By using UDP connection between a Raspberry Pi and a server, encrypted data from the Raspberry Pi are sent to the server, where the data are decrypted and displayed in a graphical format.

The purpose of this experiment is to introduce the following items:

- Raspberry Pi single-board computer
- Adafruit ADXL345 triple-axis accelerometer
- Linux development environment
- I²C (Inter-Integrated Circuit) bus
- SPI (Serial Peripheral Interface) bus
- Client-server internet communication architecture

The purpose of this exercise is to introduce students with different methods of communication between a microcomputer and I/O devices using Raspberry Pi, a simple data encryption-decryption method using a symmetric key, and a client-server UDP connection.

2.0 Component Requirements

- 1 x Raspberry Pi 3 single-board computer
- 1 x Adafruit ADXL345 triple-axis accelerometer
- 1 x Desktop (as a data storage server)

3.0 Background

A. Description of Raspberry Pi Single-Board Computer

Raspberry Pi is a series of small single-board computers developed by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries. In the lab, the third generation of Raspberry Pi, known as Raspberry Pi 3, will be used. Note that other types of Raspberry Pi can be used to accomplish this experiment.

A Raspberry Pi 3 is equipped with the following components:

- CPU: Broadcom BCM2837 1.2 GHz 64-bit Quad-core (ARMv8 Cortex-A53)
- GPU: Broadcom VideoCore IV
- RAM: 1GB LPDDR2 (900 MHz)

- Network: 10/100 Ethernet, 2.4 GHz 802.11n Wi-Fi, Bluetooth 4.1
- Storage: microSD card
- GPIO: 40-pin header, populated
- Ports: HDMI, 3.5mm audio jack, 4xUSB 2.0 ports, Camera Serial Interface (CSI), Display Serial Interface (DSI)

All Raspberry Pi 3 in this lab are installed with Raspbian, a variation of Linux operating system tailored for Raspberry Pi. Advantages of using Raspberry Pi in this lab as following. First, Raspberry Pi is a full-fledged microcomputer, like SANPER, Raspberry Pi also equipped with CPU, RAM and ability to interface a variety of peripheral devices. Second, Raspberry Pi is equipped with modern technologies, such as USB ports, networking via Ethernet, Wi-Fi, Bluetooth, and running an operating system, etc. Moreover, it is easily reconfigured by re-programming, and has extensibility of adding many peripheral devices.

B. Description of Adafruit ADXL345 Triple-Axis Accelerometer

The ADXL345 is a digital-output, 3-axis accelerometer whose low power consumption and built-in features make it ideal for use in a wide variety of applications. In this lab, ADXL345 serve as the sensor by providing the acceleration measurements to Raspberry Pi. Due to its small size, the ADXL345 has minimal effect on performance of the system and of the accelerometer, and thus it is ideal for evaluation of the ADXL345 in an existing system. Note that the ADXL345 must be communicated with digitally via SPI or I²C. With SPI and I²C equipped on both ADXL345 and Raspberry Pi, the two devices can communicate easily without the help of any other devices.

Technical Specification of ADXL345:

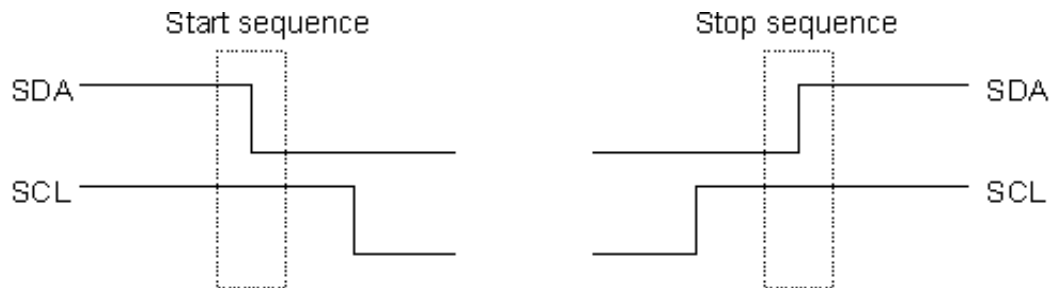
- 4 sensitivity levels: $\pm 2g$, $\pm 4g$, $\pm 8g$ or $\pm 16g$
- occupies I²C 7-bit address 0x53
- VCC takes up to 5V input and regulates it to 3.3V

C. I²C (Inter-integrated Circuit) Bus

I²C bus can be used to transfer data between master device and slave devices. I²C uses just two wires, SCL and SDA. SCL is the clock line. It is used to synchronize all data transfers over the I²C bus. SDA is the data line. It is used to specify addresses and transfer data. The SCL & SDA lines are connected to all devices on the I²C bus. Both SCL and SDA lines are "open drain" drivers. Devices on the I²C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I²C bus, only a master can do that. There can be, and usually are, multiple slaves on the I²C bus, however there is normally only one

master. It is possible to have multiple masters, but it is unusual and not covered here. Slaves will never initiate a transfer. Both master and slave can transfer data over the I²C bus, but that transfer is always controlled by the master.

When the master wishes to talk to a slave it begins by issuing a start sequence on the I²C bus. A start sequence is one of two special sequences defined for the I²C bus, the other being the stop sequence. The start sequence and stop sequence are special in that these are the only places where the SDA (data line) is allowed to change while the SCL (clock line) is high. When data is being transferred, SDA must remain stable and not change whilst SCL is high. The start and stop sequences mark the beginning and end of a transaction with the slave device, shown as following:



Data is transferred in sequences of 8 bits. The bits are placed on the SDA line starting with the MSB (Most Significant Bit). The SCL line is then pulsed high, then low. For every 8 bits transferred, the device receiving the data sends back an acknowledge bit, so there are actually 9 SCL clock pulses to transfer 8 bits of data. If the receiving device sends back a low ACK bit, then it has received the data and is ready to accept another byte. If it sends back ACK as logic high, then no further data will be sent, and the data transfer will be terminated by master.

All I²C addresses are either 7 bits or 10 bits. The use of 10-bit addresses is rare and is not covered in this experiment. Even though we are transmitting the 7-bit address over the bus, it is required to send 8 bits. The extra bit is used to inform the slave if the master is writing to it or reading from it. If the bit is 0, the master is writing to the slave. If the bit is 1 the master is reading from the slave. The 7-bit address occupies from bit 7 and bit 1 of the byte, and the Read/Write (R/W) bit is in the LSB (Least Significant Bit).

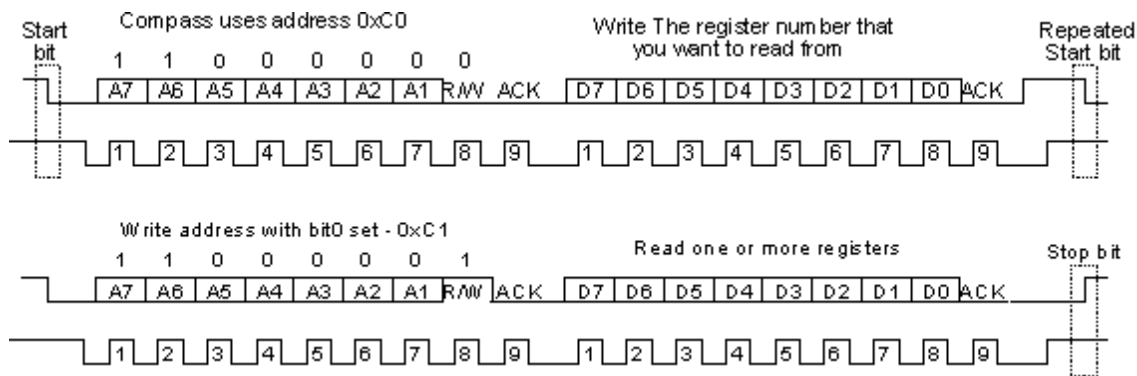
Example of I²C write process:

1. Send a start sequence
2. Send the I²C address of the slave with the R/W bit low (even address)
3. Send the internal register number of slave device you want to write to
4. Send the data byte
5. [Optionally, send any further data bytes]
6. Send the stop sequence

Example of I²C read from register 0x01 on device 0xC0 process:

1. Send a start sequence
2. Send 0xC0 (I²C address of the device with the R/W bit low (even address))
3. Send 0x01 (Internal address of the bearing register)
4. Send a start sequence again (repeated start)
5. Send 0xC1 (I²C address of the device with the R/W bit high (odd address))
6. Read data byte from device
7. Send the stop sequence

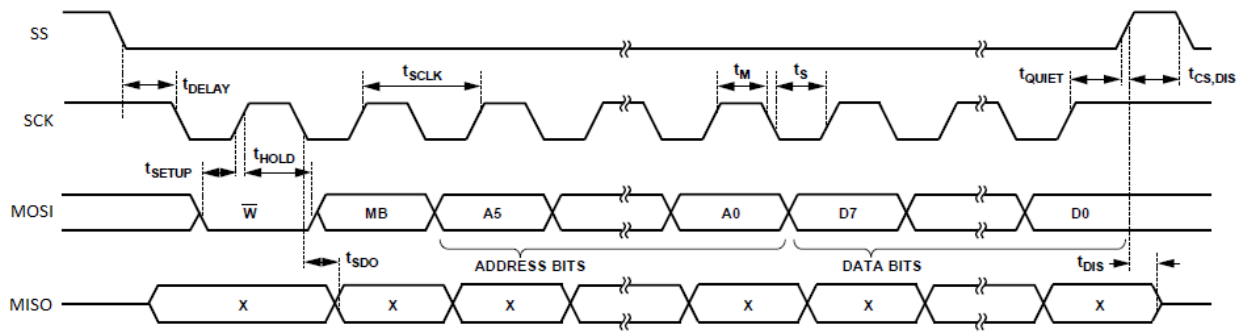
Bit sequence of reading is shown as following:



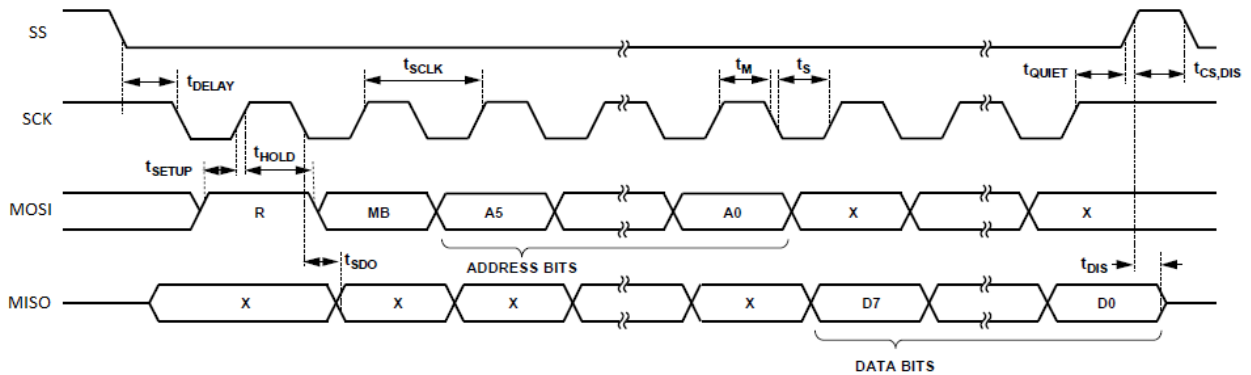
D. SPI (Serial Peripheral Interface) Bus

Similar as I²C, SPI bus can be used to transfer data between master device and slave devices. Unlike I²C, SPI uses just 3 wires or 4 wires to connect between master and slaves. In this lab, 4 wire mode is used. These 4 wires are SCK, MOSI, MISO and SS. SCK is the clock line. It is used to synchronize all data transfers over the SPI bus. MOSI and MISO are the data lines. MOSI stands for master out slave in. It is used to transfer data (information) from master to slave. MISO stands for master in slave out. It is used to transfer data (information) from slave to master. SS stands for slave select. It is a signal generated by master device, slave device with logic 0 showing on its SS pin will be enabled.

In 4-wire mode, each transaction on SPI will take 2 bytes. When writing, master device initiates the communication by driving SS of slave to low. In the first byte, the master will clear MSB for writing, and put the internal address of the register to be written on bit 6 to bit 0. Unlike I²C bus, SPI relies on SS signal to select slave device, therefore, the address showing on the bus is always the internal address on selected slave device. In the second byte, data to be written will be sent by master device.

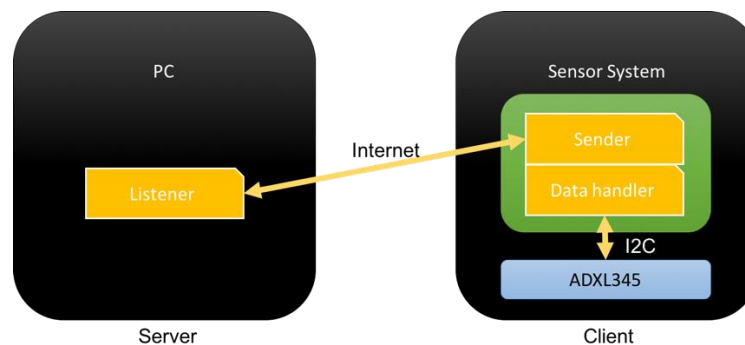


When reading, master device initiates the communication by driving SS of slave to low. In the first byte, the master will set MSB for writing, and put the internal address of the register to be read on bit 6 to bit 0. In the second byte, slave device will response by sending the data from the register on to master. Shown as following:



E. Server-Client Architecture

Client-server architecture (client/server) is a network architecture in which each computer or process on the network is either a client or a server. Servers are powerful computers or processes dedicated to managing disk drives (file servers), printers (print servers), or network traffic (network servers). Clients are PCs or workstations on which users run applications. Clients rely on servers for resources, such as files, devices, and even processing power. In this lab, a server-client architecture is adopted for communication between Raspberry Pi(client) and PC(server). The system architecture is shown as following.



F. Encryption of Electronic Data

The purpose of data encryption is to protect digital data confidentiality as it is stored on computer systems and transmitted using the internet or other computer networks. As mentioned in Server-Client Architecture, data collected by the device will be transmitted to the server over the internet. Data transmitted over the internet is prone to eavesdropping, therefore, they are often encrypted.

In this lab we use AES (Advanced Encryption Standard) algorithm to encrypt data transmitted over the internet on application layer. To be more specific, once data been collected by the client device, it will be encrypted then transmitted. The server will decrypt received data before using it. AES is a symmetric encryption algorithm, therefore, the keys used for encryption and decryption are the same. To avoid brutal force deciphering, AES algorithms are usually further enforced by IV (Initialization Vector).

In the example code, both AES key and IV are specified. Please review the example code carefully, and understand the processes regarding encryption, decryption and transmission.

4.0 Statement of Problem

In this experiment you will implement a real-time 3 axis motion sensing system using Raspberry Pi and ADXL345. The system should be able to retrieve data from sensor and send them to a server.

5.0 Preliminary Assignment

1. Review course materials on I²C and SPI, understand how Raspberry Pi and ADXL345 communicate, and how such process is implemented in software.
2. Use information provided in *Appendix D* and *Reference 8*, draw a schematic of wiring between Raspberry Pi 3 and Adafruit ADXL345 using I²C. Show your work to your TA in the beginning of the lab.
3. Use information provided in *Appendix D* and *Reference 8*, draw a schematic of wiring between Raspberry Pi 3 and Adafruit ADXL345 using SPI. Show your work to your TA in the beginning of the lab.
4. Preview instruction on Raspberry Pi in procedure A.1, understand how to compile a project on Raspberry Pi.

5. Implement the program as required in procedure A.2. Write the program with proper comments, explain how you implemented such program to your TA in the beginning of the lab.
6. Implement the program as required in procedure B.2. Write the program with proper comments, explain how you implemented such program to your TA in the beginning of the lab.
7. Familiar yourself with the client-server program architecture as explained in procedure B, implement the program that sending data to server. Explain how you implemented such program to your TA in the beginning of the lab.

6.0 Procedure

A. Implement Real-Time Motion Sensing Device Using I²C

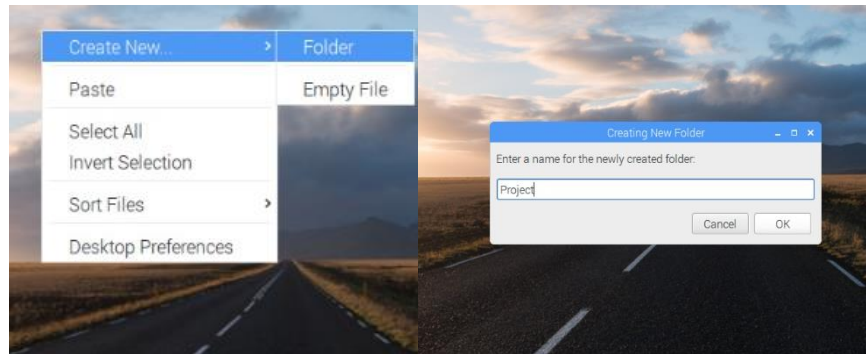
In this part of the lab, you are required to implement a program on Raspberry Pi that retrieve and log real-time readings from ADXL345 using I²C bus.

1. Remove the cover of Raspberry Pi, wire the Raspberry Pi and ADXL345 properly. So that these two devices can communicate using I²C bus.
2. Compile Example Code Provided in Appendix A on Raspberry Pi. Follow the instructions in this section, compile the example code on Raspberry Pi. Record the result if necessary.

To compile this code in Raspberry Pi, there are a few steps needs to take:

- 1) Connect the mouse, keyboard via USB. You can use the touch screen on mounted on the device or connect to a monitor through HDMI cable. Once powered on, the video output device cannot be changed, thus you must choose your output before powering on.
- 2) Turn on the Raspberry Pi, by plug in the micro USB cable into Raspberry Pi. Wait for the system to boot up.

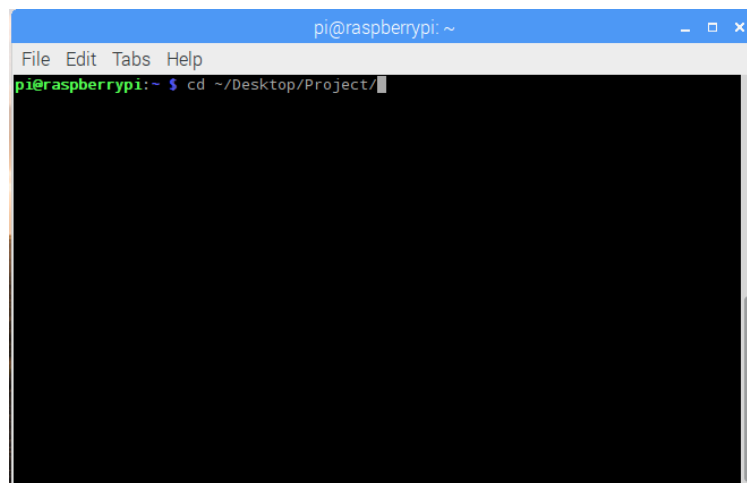
Once the system is boot up, you can create a working directory on desktop and put all source code including header files in it. To create a folder, simply right click on the desktop, then select "*Create New >> Folder*" then type folder name of your desire (in this instruction let's use "*Project*" as name for working directory), as shown in following screenshots.



- 3) Now you are ready to compile the code and generate an executable file. To do so, you will have to open a “*terminal*” by pressing combination of *ctrl* + *alt* + *t*. Then navigate to your working directory, in this case “*Project*” on desktop, by typing in following command in the terminal:

```
cd ~/Desktop/Project/
```

Your terminal should look like this:



Then hit *enter* to enter the directory.

- 4) After entering the working directory, you can compile the source code using “*g++*”. To do so simply typing the command in the terminal:

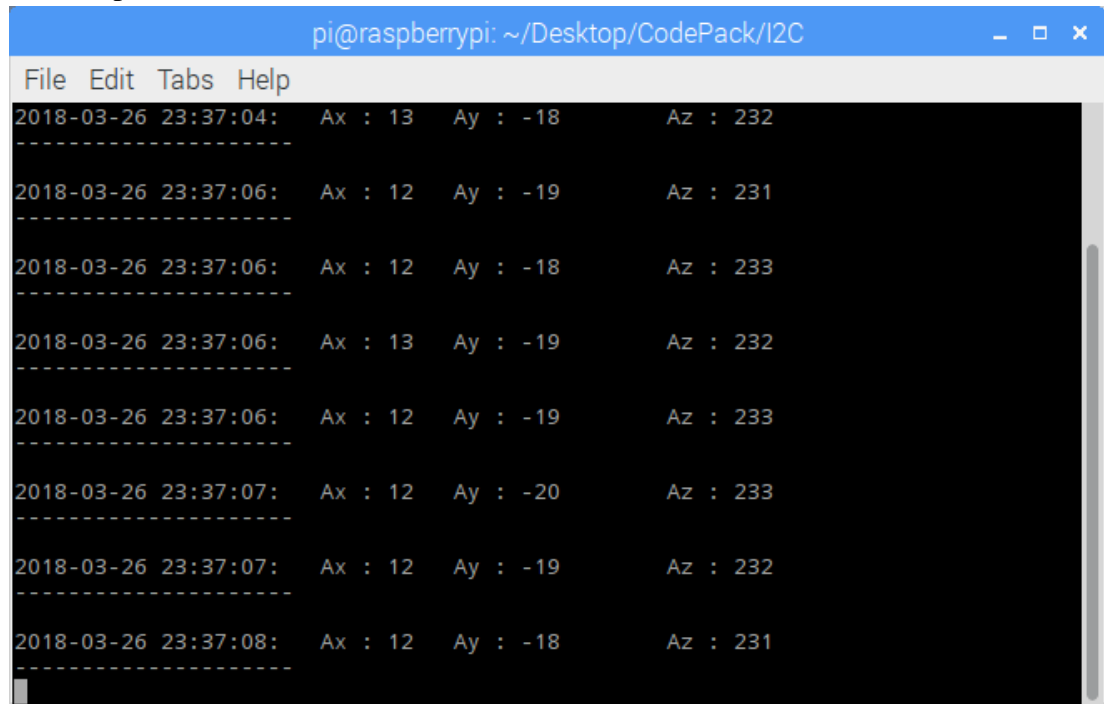
```
g++ -g main.cpp ADXL345.cpp -o exe.out
```

This command will use “*g++*” as the compiler, compile “*main.cpp*”. “*-o exe.out*” specifying that the compiler should generate an executable output named as “*exe.out*”. “*-g*” set the flag for debug symbols, without such flag, the program will not be debugged. You can use GDB to debug your program, please refer *Reference 6*.

5) Now execute the file by typing:

`./exe.out`

Your output should look like this:



The screenshot shows a terminal window titled "pi@raspberrypi: ~/Desktop/CodePack/I2C". The window contains a menu bar with "File", "Edit", "Tabs", and "Help". The output is as follows:

Timestamp	Ax	Ay	Az
2018-03-26 23:37:04:	13	-18	232
2018-03-26 23:37:06:	12	-19	231
2018-03-26 23:37:06:	12	-18	233
2018-03-26 23:37:06:	13	-19	232
2018-03-26 23:37:06:	12	-19	233
2018-03-26 23:37:07:	12	-20	233
2018-03-26 23:37:07:	12	-19	232
2018-03-26 23:37:08:	12	-18	231

- 6) To stop the program simple press and hold “q” until the program exits and return to bash(terminal). Finally, check the output on the screen as well as in the “*output.txt*”. A copy of your output should be stored in the text file. You can terminate the process forcefully by pressing *ctrl+c* at any time, however this method should only be used when your program stops responding.
3. In this step, please modify the example code and make it store a time stamp in the output file as well. To be more specific, you program should store the sensor reading into an output text file, moreover, for each entry, you need to store the time as well. Show your result to your TA.

B. Client Server Communication

To build a server client application, two major parts need to be implemented. One is a data sender on the client. The other part is a listener running on the server. In this part of the lab, you are required to implement a data sender on Raspberry Pi that retrieve and log real-time data from ADXL345 sensor, then send them to a server (PC in the lab).

1. Regarding implementing the data sender, you are required to change the wiring on Raspberry Pi so it can communicate with ADXL345 using SPI instead of I²C. An example program of client sender been provided in Appendix B. This example program meant to be compiled and run on Raspberry Pi. In the example, the program encrypts and send a predefined information to the server after encryption. Run the example code and check the predefined data shown on the server.
 - 1) Remove the cover of Raspberry Pi, and re-wire the Raspberry Pi and ADXL345. So that these two devices can communicate using SPI bus. (Use your design from pre-lab assignment)
 - 2) Compile example code provided in Appendix B on Raspberry Pi (review procedure A.2.3-A.2.6 if necessary). This example first retrieves data from ADXL345, then encrypt and send them to the server.
 - 3) In this step, please modify the example code and make it store a time stamp in the output file as well. To be more specific, you program should store the sensor reading into the output text file, moreover, for each entry, you need to store data as well as the time. Show your result to your TA. (Without the server listener ready, you may only see your data logged locally.)
 - 4) Please make necessary change(s) to the example program provided in Appendix B so the client can collect encrypt and send data collected from the sensor in real-time (review instruction in Procedure A.2.1-A.2.4 if necessary). Please use following command to compile your code:

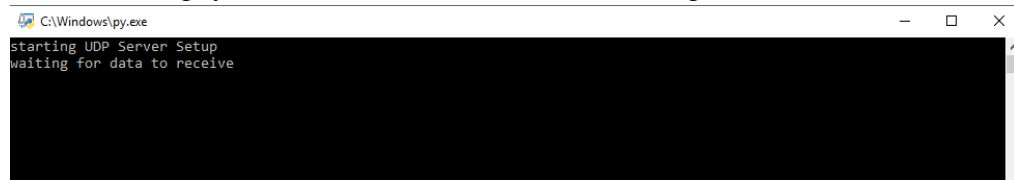
```
g++ -g -lwiringPi -lcrypto client.cpp encryption.cpp -o exe.out
```

While “*-lwiringPi*” and “*-lcrypto*” are the flags for the linker. In this case linker will link the project with *wiringPi* and *crypto* libraries locate in system environment paths.

2. Regarding the server listener, it will run on the server while waiting for any incoming connections and response accordingly. Once connection is established, data from sensor

will be received, decrypted (using [Crypto package](#) in Python) and displayed on the server, a live plot will be generated (using [Matplotlib package](#) in Python) on the server as well. The server listener software is provided in *Appendix B*, the server listener software needs no change. By taking following steps, you should be able start the server listener.

- 1) With the data sender ready, the connection can be established once the listener on the server is able to process received information properly. The listener on server is implemented in Python. Python is a script language; hence it can be executed by invoking a Python interpreter. The advantage of this script language is, the environment will be easier to be configured. In this lab, all computers have configured with [anaconda](#) tool. In short, anaconda is a tool that manage Python runtime as well as packages.
- 2) To execute the Python script, first copy “*server.py*” to any desired directory on server computer. Then open “*anaconda prompt*” and navigate to the folder contain your Python script. Finally type “python *server.py*” in anaconda prompt. Once the server listener is running, you should see a window as following:



- 3) As explained earlier, the server listener software has been implemented, therefore, you don't need to modify the code. However, you need to modify IP address; port number; AES key and AES IV to receive incoming connections properly. To find the IP address of your server, you need to open CMD (command prompt), a counterpart of terminal on windows. You can open it by press “*Windows_Key+r*”, then type “*cmd*” and press “*run*”. In the CMD window type:

ipconfig

You should get the network information about your computer, shown as following:

```

Select Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Student>ls
'ls' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Student>ipconfig

Windows IP Configuration

Ethernet adapter Ethernet:

    Connection-specific DNS Suffix  . : iit.edu
    IPv6 Address. . . . . : 2620:f3:8000:5009::de31
    Link-local IPv6 Address . . . . . : fe80:2598:87b5:b21a:ea9%5
    IPv4 Address. . . . . : 216.47.158.246
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : fe80:d50::11%5
                                216.47.158.2

Wireless LAN adapter Wi-Fi:

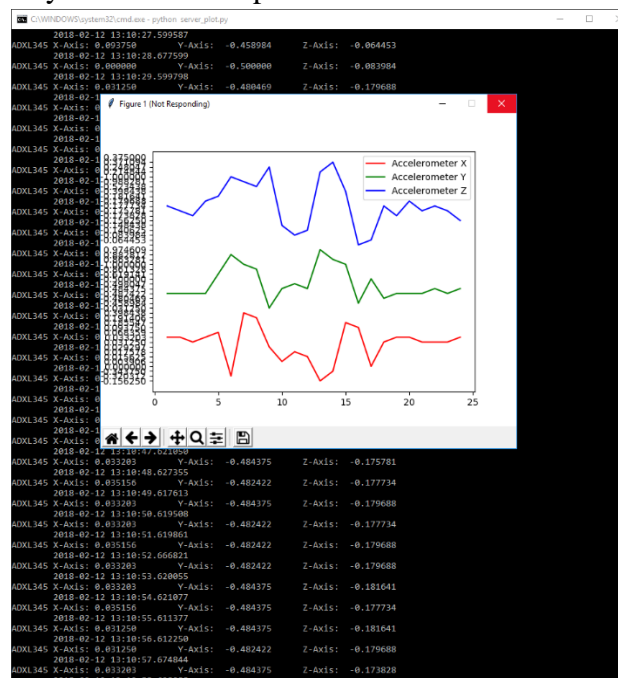
    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 11:

    Media State . . . . . : Media disconnected

```

- 5) Find the entry shown as “IPv4 Address”, that would be the address of your computer. As to the port for connection, due to the network policy of Illinois Tech, only Port 6000 is allowed for UDP connection. Update your “server.py” accordingly. Once your client and server are able to run properly, you should see the server showing following content on your server computer:



3. With both server listener and client sender explained, now you are required implement a program on client that collect motion sensor reading and send them to the server. Show your result to your TA.

7.0 Discussion

1. A fully commented code that you implemented in Procedure A.2 (10 pts)
2. Draw a flow chart and explain how the program you implemented in Procedure A.2 works, please elaborate on how Raspberry Pi get the reading from sensors. (10 pts)
3. How does a master device differentiate slave devices using I²C and SPI bus (in terms of protocol hardware implementation)? (10 pts)
4. If value 0x0A were found in register 0x31 on ADXL345, what is the behavior of the sensor regarding the register? (10 pts)
5. What's the difference between terminal on Raspberry Pi and TUTOR on SANPER? What are the advantages and disadvantages each of them has? (10 pts)

8.0 References

Some references you may find useful:

- [1] A more general tutorial on C programming on Raspberry Pi
<https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/rpii/cintro.html>
- [2] More insight on socket programming
<https://www.geeksforgeeks.org/socket-programming-cc>
- [3] Raspberry Pi hardware documents
<https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md>
- [4] Common Linux commands
<https://www.raspberrypi.org/documentation/linux/usage/commands.md>
- [5] Adafruit ADXL345 accelerometer program guide
<https://learn.adafruit.com/adxl345-digital-accelerometer/programming>
- [6] GDB debugger quick manual
<http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>
- [7] Python 3.6 tutorial
<https://docs.python.org/3/tutorial/>
- [8] ADXL345 datasheet
<http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf>

Appendix A

Example for sensor data retrieval and logging using I²C (dependent header files are not included but can be found in CD package)

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <sys/time.h>
4
5  #include "i2c-dev.h"
6  #include "ADXL345.h"
7
8  #define I2C_FILE_NAME "/dev/i2c-1" // for Rpi B+
9  void INThandler(int sig);
10
11 int main(int argc, char **argv)
12 {
13     // Open a connection to the I2C userspace control file.
14     int i2c_fd = open(I2C_FILE_NAME,O_RDWR);
15     if (i2c_fd < 0)
16     {
17         printf("Unable to open i2c control file, err=%d\n", i2c_fd);
18         exit(1);
19     }
20
21     printf("Opened i2c control file, id=%d\n", i2c_fd);
22     ADXL345 myAcc(i2c_fd);
23     int ret = myAcc.init();
24     if (ret)
25     {
26         printf("failed init ADXL345, ret=%d\n", ret);
27         exit(1);
28     }
29     usleep(100 * 1000);
30
31     signal(SIGINT, INThandler);
32     short ax,ay,az;
33     // create file IO
34     FILE *fp;
35     fp = fopen("./output.txt","w+");
36
37     char TimeString[128];
38     timeval curTime;
39     while(1)
40     {
41         //get the current time
42         gettimeofday(&curTime, NULL);
43         strftime(TimeString, 80, "%Y-%m-%d %H:%M:%S", localtime(&curTime.tv_sec));
44         printf(TimeString);
45         printf(": ");
46         // now, fetch data from sensor
47         myAcc.readXYZ(ax,ay,az);
48         //print to screen
49         printf("Ax : %hi \t Ay : %hi \t Az : %hi \n",ax,ay,az);
50         printf("-----\n");
51         //print to file
52         fprintf(fp,TimeString);
53         fprintf(fp,": ");
54         fprintf(fp, "Ax : %hi \t Ay : %hi \t Az : %hi \n",ax,ay,az);
55         fprintf(fp, "-----\n");
56         if (getchar() == 'q') break;
57     }
58
59     fclose(fp);
60
61     return 0;
62 }
63
64 void INThandler(int sig)
65 {
66     signal(sig, SIG_IGN);
67     exit(0);
68 }
69
```

Appendix B

Example of the sensor data retrieval using SPI protocol and client module to send data to server

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <string.h>
6  #include <termios.h>
7  #include <fcntl.h>
8
9  #include <wiringPiSPI.h>
10
11 #include <netdb.h>
12 #include <netinet/in.h>
13 #include <sys/types.h>
14 #include <sys/socket.h>
15
16 #include "encryption.hpp"
17
18 //specify the size of buffer and port for transmit
19 #define BUFFERSIZE 64
20 #define PORTNUMBER 50123
21
22 //specify key and IV
23 const char KEY[] = "3874460957140850";
24 const char IV[] = "9331626268227018";
25
26 //specify the address for server
27 const char hostname[] = "0.0.0.0";
28
29 //specify the channel used for SPI
30 const int spiChannel(0);
31
32 void initialSPI()
33 {
34     wiringPiSPISetupMode(spiChannel, 1000000, 3);
35     usleep(10);
36     unsigned char buf[2];
37     //buf[0] = 0x80;
38     //wiringPiSPIDataRW(spiChannel, (unsigned char *)&buf, 2);
39     //std::cout << std::hex << (short)buf[1] << std::endl;
40     //std::cout << "finished testing" << std::endl;
41
42     //configure power
43     buf[0] = 0x2D;
44     buf[1] = 0x18;
45     wiringPiSPIDataRW(spiChannel, (unsigned char *)&buf, 2);
46     //std::cout << "finished setting up powerctl" << std::endl;
47
48     //configure data format (Full_res, left justify, +-2g)
49     buf[0] = 0x31;
50     buf[1] = 0x00;
51     wiringPiSPIDataRW(spiChannel, (unsigned char *)&buf, 2);
52     //std::cout << "finished setting up dataformat" << std::endl;
53     return;
54 }
55
56 void readRawXYZ(short &X, short &Y, short &Z)
57 {
58     unsigned char txRxData[2];
59     unsigned char buf[6];
60     //read data
61     for (unsigned short i(0); i < 6; i++)
62     {
63         txRxData[0] = (unsigned char) ((unsigned short)0xB2 + i);
64         wiringPiSPIDataRW(spiChannel, (unsigned char *)&txRxData, 2);
65         buf[i] = txRxData[1];
66     }
67
68     X = (buf[1] << 8) | buf[0];
69     Y = (buf[3] << 8) | buf[2];
```

```

70     Z = (buf[5] << 8) | buf[4];
71
72     return;
73 }
74
75 void readXYZ(float &X, float &Y, float &Z, const short &scale = 2)
76 {
77     short x_raw, y_raw, z_raw;
78     readRawXYZ(x_raw, y_raw, z_raw);
79     X = (float) x_raw / 1024 * scale;
80     Y = (float) y_raw / 1024 * scale;
81     Z = (float) z_raw / 1024 * scale;
82
83     return;
84 }
85
86 // return true when keyboard been pressed, false other wise
87 bool kbhit(void)
88 {
89     struct termios oldt, newt;
90     int ch;
91     int oldf;
92
93     tcgetattr(STDIN_FILENO, &oldt);
94     newt = oldt;
95     newt.c_lflag &= ~(ICANON | ECHO);
96     tcsetattr(STDIN_FILENO, TCSANOW, &newt);
97     oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
98     fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);
99
100    ch = getchar();
101
102    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
103    fcntl(STDIN_FILENO, F_SETFL, oldf);
104
105    if(ch != EOF)
106    {
107        ungetc(ch, stdin);
108        return true;
109    }
110
111    return false;
112 }
113
114 int main()
115 {
116     initialSPI();
117     float x, y, z;
118
119     //verify the socket
120     int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
121     if (sockfd < 0)
122     {
123         printf("ERROR opening socket\n");
124         exit(1);
125     }
126
127     //verify host name/address
128     struct hostent *server = gethostbyname(hostname);
129     if (server == NULL)
130     {
131         printf("ERROR, no such host as %s\n", hostname);
132         exit(1);
133     }
134
135     //Build the server internet address
136     struct sockaddr_in serveraddr;
137     bzero((char *) &serveraddr, sizeof(serveraddr));
138     serveraddr.sin_family = AF_INET;

```



```

139 bcopy((char *)server->h_addr,(char *)&serveraddr.sin_addr.s_addr, server->h_length);
140 serveraddr.sin_port = htons(PORTNUMBER);
141
142 char bufRaw[BUFFERSIZE], bufCiphared[BUFFERSIZE * 2];
143
144 while(!kbhit())
145 {
146     readXYZ(x,y,z);
147
148     //format the buffere with output
149     //float with 2 and 6 digits before and after decimal point
150     snprintf(bufRaw,BUFFERSIZE,"%2.6f, %2.6f, %2.6f,\n", x, y, z);
151     printf(bufRaw);
152
153     int length = encrypt((const char *)&KEY, (const char *)&IV, &bufRaw[0], (char
    *)&bufCiphared);
154
155     for (uint i = 0; i < length; i++)
156     printf("%02x", bufCiphared[i]);
157     printf("\n");
158
159
160     int sendStatus = sendto(sockfd, bufCiphared, length, 0, (struct sockaddr
    *)&serveraddr, sizeof(serveraddr));
161     if ( sendStatus<0 )
162     {
163         printf("sent failed with status %d\n", sendStatus);
164         exit(1);
165     }
166
167
168     usleep(1000000);
169
170 }
171 return 0;
172 }
173
174

```

Appendix C

Example of the listener (Server)

```
1  #!/usr/bin/env python
2  import socket
3  import sys
4  import datetime
5  import matplotlib.pyplot as plot
6  from matplotlib import animation
7  from Crypto.Cipher import AES
8
9
10 # server network configurations
11 SERVER_IP_ADDRESS = "0.0.0.0"
12 PORT = 50123
13
14 time = [0]*50
15 for i in range(0,50):
16     time[i] = i
17
18 ax_points = [float(0)]*50
19 ay_points = [float(0)]*50
20 az_points = [float(0)]*50
21
22 print("starting UDP Server Setup")
23 sys.stdout.flush()
24
25 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
26 sock.bind( (SERVER_IP_ADDRESS, PORT) )
27
28 print("waiting for data to receive")
29 sys.stdout.flush()
30
31 KEY = b'3874460957140850'
32 iv = b'9331626268227018'
33
34 fig = plot.figure()
35 ax = plot.axes(xlim=(0, 50), ylim=(-2, 2))
36 lineX, lineY, lineZ, = ax.plot([], [], [], [], [], lw=2)
37
38
39 def init():
40     lineX.set_data([], [])
41     lineY.set_data([], [])
42     lineZ.set_data([], [])
43     return lineX, lineY, lineZ,
44
45 def updateData(i):
46     decryption_suite = AES.new(KEY, AES.MODE_CBC, IV=iv)
47     data, addr = sock.recvfrom(64)
48     print (''.join('{:02x}'.format(x) for x in data))
49     plain_text = decryption_suite.decrypt(data)
50
51     data = plain_text.decode('utf-8')
52     ax,ay,az,dump = data.split(",")
53     print("ADXL345 X-Axis: " + ax + "\tY-Axis: " + ay + "\tZ-Axis: " + az + "\t" +
54           datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f"))
55     sys.stdout.flush()
56
57     del ax_points[0]
58     del ay_points[0]
59     del az_points[0]
60     ax_points.append(float(ax))
61     ay_points.append(float(ay))
62     az_points.append(float(az))
63     lineX.set_data(time,ax_points)
64     lineY.set_data(time,ay_points)
65     lineZ.set_data(time,az_points)
66     return lineX, lineY, lineZ,
```

```
67  anim = animation.FuncAnimation(fig, updateData, init_func=init,  
68                                frames=200, interval=20, blit=True)  
69  plot.show()  
70
```

Appendix D

Pin map of Raspberry Pi with I2C and SPI highlighted:

