LAB 04 – PART 1/3 INTERFACING AN ACCELEROMETER WITH LINUX ON RPI BOARD

GOAL

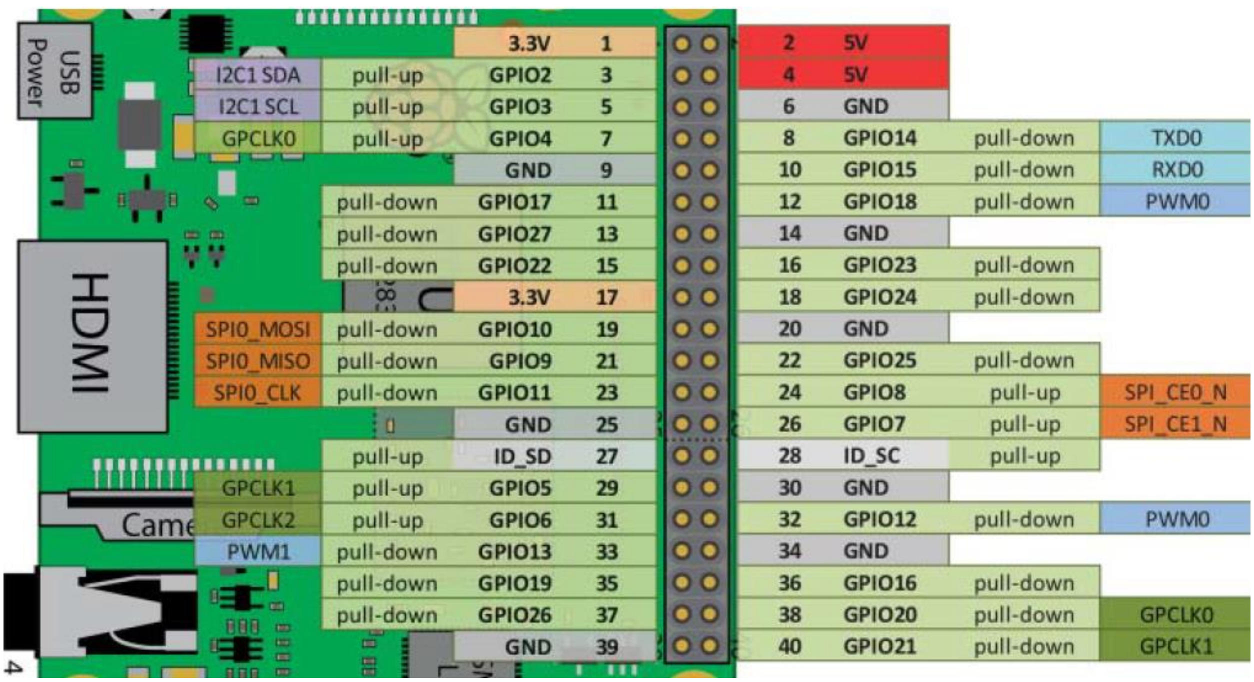
The goal of this Lab (part 1/3) is to learn how you use I2C bus on Raspberry Pi (RPi) to interface with ADXL345 accelerometer

OVERVIEW

The ADXL-345 digital accelerometer measures acceleration in three axes, and it internally sample and filter its data according to settings that are placed in its registers. The ADXL-345 is set to measure values with a fixed 13-bit resolution at up to +/- 16 g with the sensitivity of measuring an acceleration of 1g (9.81m/s2). The ADXL-345 can be interfaced to I2C or SPI bus. The datasheet for ADXL-345 is an important document that should be read along with this lab exercise: www.analog.com/ADXL345

The C++ programming example in this lab simply reads in registers and displays the device ID (DEVID) of an accelerometer on a shared I2C bus in RPi, and thereby confirming that the C++ program is working correctly with a specific accelerometer device and reading correct orientation (x, y, and z) data.

PIN LAYOUT - THE RASPBERRY PI BOARD



CONNECT THE ADXL345 TO I2C BUS ON THE RASPBERRY PI

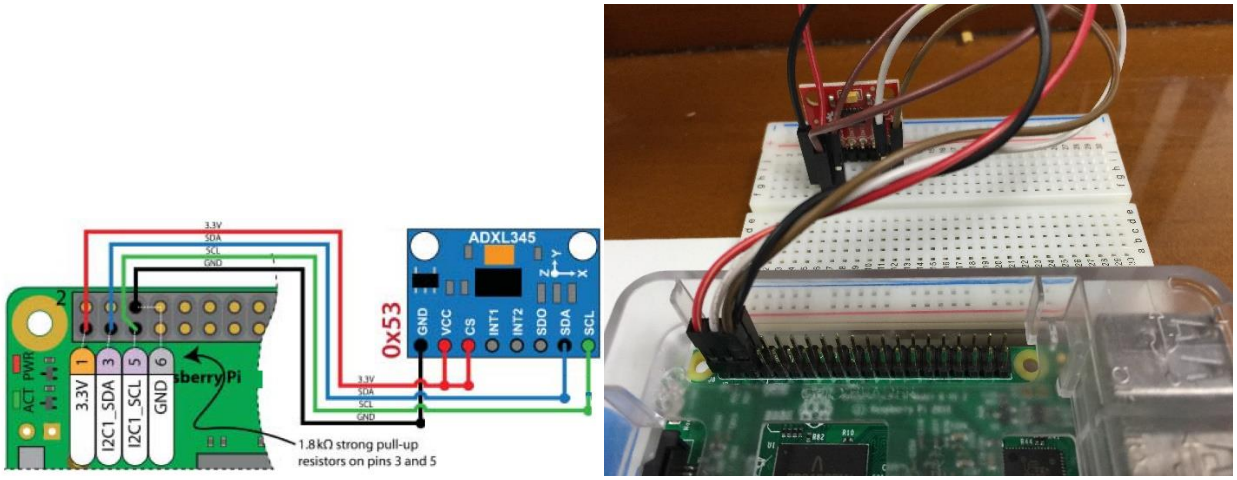


Fig. 1 ADXL345 wiring with Raspberry Pi

The next is to construct a circuit as shown in Figure 1 above.

DEVICE ID AND ADDRESS ON I2C BUS

The I2C bus allows users to connect more than one devices. Therefore, each device connected to the same I2C bus needs to have a unique ID to communicate with the processor. We can view the device ID for the ADXL345 accelerometer.

First, we use the Linux command line as follows.

When both I2C buses are enabled, the i2cdetect command display the available i2c buses.

```
pi@raspberrypi:~ $ i2cdetect -l
i2c-1   i2c             bcm2835 I2C adapter                     I2C adapter
```

If the ADXL345 is wired as in Fig. 1, the ADXL345 should attach to /dev/i2c-1 bus, then it can be probed for connected devices with a command prompt as shown below, which will result in the following output:

Linux command: i2cdetect -y -r -1

```
pi@raspberrypi:~ $ i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- 53 -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Fig. 2 address block form

Hexadecimal address at 0x53 is displayed and this address is where the ADXL345 occupies on the I2C bus of a RPi. Each device has its addressed defined but sometimes a problem will arise if two slave devices use the same address. However, many I2C devices have options of selecting different addresses to avoid an address conflict.

I2C DUMP

The i2c dump command as shown below can be used to read the values of the registers of the device attached to the I2C bus and display them in a hexadecimal block form as shown in figure 2.

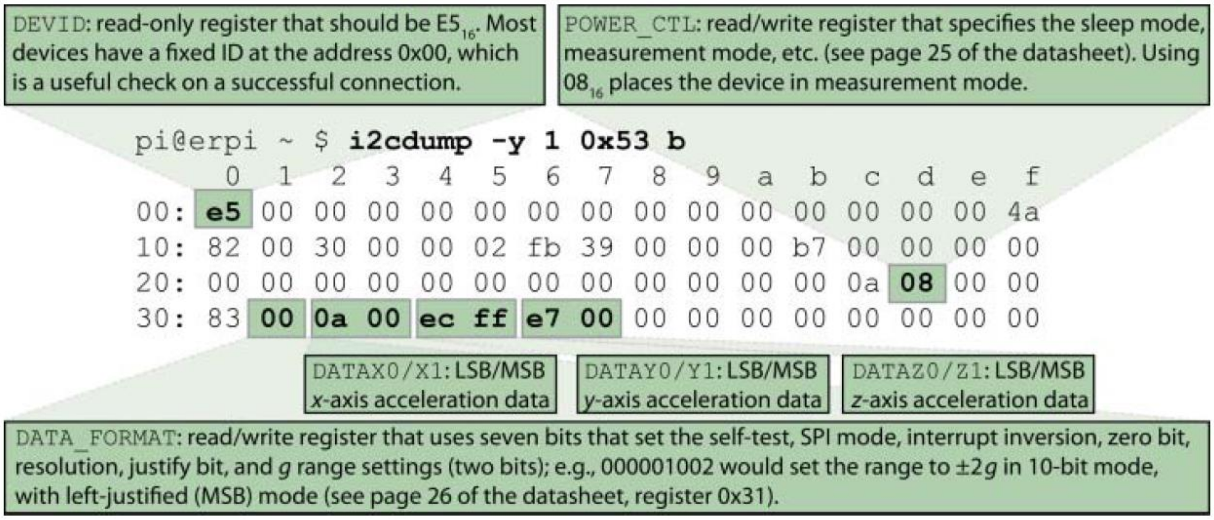
```
pi@raspberrypi:~ $ i2cdump -y 1 0x53
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: e5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4a    ?..............J
10: 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00    ..?.............
20: 00 00 00 00 00 00 00 00 00 00 00 00 0a 00 00 00    ............?...
30: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ?...............
40: e5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4a    ?..............J
50: 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00    ..?.............
60: 00 00 00 00 00 00 00 00 00 00 00 00 0a 00 00 00    ............?...
70: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ?...............
80: e5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4a    ?..............J
90: 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00    ..?.............
a0: 00 00 00 00 00 00 00 00 00 00 00 00 0a 00 00 00    ............?...
b0: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ?...............
c0: e5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4a    ?..............J
d0: 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00    ..?.............
e0: 00 00 00 00 00 00 00 00 00 00 00 00 0a 00 00 00    ............?...
f0: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ?...............
```

The address at 0x53 shows the device ID of the ADXL345 connected to the raspberry Pi.

The use of i2cget command allows to print the device ID (0xE5) on the screen.

```
pi@raspberrypi:~ $ i2cget -y 1 0x53 0x00
0xe5
```

The datasheet for ADXL345 can be found here: www.analog.com/ADXL345. You can find the similar information as shown below in the datasheet.



Most of the *x*-, *y*-, and *z*-axis acceleration values in accelerometers are stored using a 10-bit or 13-bit resolution; therefore, two bytes are required for each reading. Also, the data is in 16-bit two's complement form. To sample at 13 bits, the ADXL345 is set to ±16 *g* range. The Figure above (based on the ADXL345 datasheet) describes the signal sequences required to read and write to the device. For example, to write a single byte to a device register, the master/slave access pattern in the first row is used as follows:

- The master sends a *start bit* (i.e., it pulls SDA low, while SCL is high).
- While the clock toggles, the 7-bit slave address is transmitted one bit at a time.
- A read bit (1) or write bit (0) is sent, depending on whether the master wants to read or write to/from a slave register.
- The slave responds with an *acknowledge bit* (ACK = 0).
- In write mode, the master sends a byte of data one bit at a time, after which the slave sends back an ACK bit. To write to a register, the register address is sent, followed by the data value to be written.
- Finally, to conclude communication, the master sends a *stop bit* (i.e., it allows SDA to float high, while SCL is high).

I2C COMMUNICATION IN C

To display the ADXL345 device ID, the following code is created.

```
int readRegisters(int file){    // read all 64(0x40) registers to a buffer
   writeRegister(file, 0x00, 0x00);  // set address to 0x00 for block read
   if(read(file, dataBuffer, BUFFER_SIZE)!=BUFFER_SIZE){
      cout << "Failed to read in the full buffer." << endl;
      return 1;
   }
   if(dataBuffer[DEVID]!=0xE5){
      cout << "Problem detected! Device ID is wrong" << endl;
      return 1;
   }
   return 0;
}
```

To process the two raw 8-bit acceleration registers, code to combine two bytes into a single 16-bit value is written as follows:

```
short combineValues(unsigned char upper, unsigned char lower){
   //shift the MSB left by 8 bits and OR with the LSB
   return ((short)upper<<8)|(short)lower;
}
```

COMPLETE C++ CODE IMPLEMENTATION TO INTERFACE THE ADXL345 ACCELEROMETER

```
/** Sample I2C ADXL345 Code that outputs the x,y and z accelerometer values
 on a single line for sixty seconds.  */

#include<iostream>
#include<stdio.h>
#include<fcntl.h>
#include<sys/ioctl.h>
#include<linux/i2c.h>
#include<linux/i2c-dev.h>
#include<iomanip>
#include<unistd.h>
using namespace std;

// Small macro to display value in hexadecimal with 2 places
#define HEX(x) setw(2) << setfill('0') << hex << (int)(x)

// The ADXL345 Resisters required for this example
#define DEVID       0x00
#define POWER_CTL   0x2D
#define DATA_FORMAT 0x31
#define DATAX0      0x32
#define DATAX1      0x33
#define DATAY0      0x34
#define DATAY1      0x35
#define DATAZ0      0x36
#define DATAZ1      0x37
#define BUFFER_SIZE 0x40
unsigned char dataBuffer[BUFFER_SIZE];

// Write a single value to a single register
int writeRegister(int file, unsigned char address, char value){
   unsigned char buffer[2];
   buffer[0] = address;
   buffer[1] = value;
   if (write(file, buffer, 2)!=2){
      cout << "Failed write to the device" << endl;
      return 1;
   }
   return 0;
```

```
   }
   return 0;
}

// Read the entire 40 registers into the buffer (10 reserved)
int readRegisters(int file){
   // Writing a 0x00 to the device sets the address back to
   //  0x00 for the coming block read
   writeRegister(file, 0x00, 0x00);
   if(read(file, dataBuffer, BUFFER_SIZE)!=BUFFER_SIZE){
      cout << "Failed to read in the full buffer." << endl;
      return 1;
   }
   if(dataBuffer[DEVID]!=0xE5){
      cout << "Problem detected! Device ID is wrong" << endl;
      return 1;
   }
   return 0;
}

// short is 16-bits in size on the Raspberry Pi
short combineValues(unsigned char msb, unsigned char lsb){
   //shift the msb right by 8 bits and OR with lsb
   return ((short)msb<<8)|(short)lsb;
}

int main(){
   int file;
   cout << "Starting the ADXL345 sensor application" << endl;
   if((file=open("/dev/i2c-1", O_RDWR)) < 0){
      cout << "failed to open the bus" << endl;
      return 1;
   }
   if(ioctl(file, I2C_SLAVE, 0x53) < 0){
      cout << "Failed to connect to the sensor" << endl;
      return 1;
   }
   writeRegister(file, POWER_CTL, 0x08);
   //Setting mode to 00000000=0x00 for +/-2g 10-bit
```

```
    writeRegister(file, POWER_CTL, 0x08);
    //Setting mode to 00000000=0x00 for +/-2g 10-bit
    //Setting mode to 00001011=0x0B for +/-16g 13-bit
    writeRegister(file, DATA_FORMAT, 0x00);
    readRegisters(file);
    cout << "The Device ID is: " << HEX(dataBuffer[DEVID]) << endl;
    cout << "The POWER_CTL mode is: " << HEX(dataBuffer[POWER_CTL]) << endl;
    cout << "The DATA_FORMAT is: " << HEX(dataBuffer[DATA_FORMAT]) << endl;
    cout << dec << endl;   //reset back to decimal

    // Now loop a display the x, y, z accelerometer for 60 seconds
    int count=0;
    while(count < 60){
       short x = combineValues(dataBuffer[DATAX1], dataBuffer[DATAX0]);
       short y = combineValues(dataBuffer[DATAY1], dataBuffer[DATAY0]);
       short z = combineValues(dataBuffer[DATAZ1], dataBuffer[DATAZ0]);
       //Use \r and flush to write the output on the same line
       cout << "X="<<x<<" Y="<<y<<" Z="<<z<<" sample="<<count<<"     \r"<<flush;
       usleep(1000000);
       readRegisters(file);  //read the sensor again
       count++;
    }
    close(file);
    return 0;
}
```

YOUR TURN

Your task in this lab is to use the raw data for x, y, and z from the ADXL345 and convert these data to pitch and roll angle, and then print these data on the screen.

Add code to convert these values (x, y, and z data) into pitch and roll form. When done, have the instructor check off your work.

CONVERSION TO PITCH AND ROLL

The algorithm used to convert to pitch and roll can be found online resources with a google search. In summary, you may use the model in the code below to calculate for pitch and roll angles for most accelerometers.

```
// now loop and display the x, y, z, accelerometer for 60 seconds
float COUNTS_PER_G = 4096.0;
float roll = 0.0;
float pitch = 0.0;
int count = 0;

    short x = combineValues(dataBuffer[DATAX1], dataBuffer[DATAX0]);
    short y = combineValues(dataBuffer[DATAY1], dataBuffer[DATAY0]);
    short z = combineValues(dataBuffer[DATAZ1], dataBuffer[DATAZ0]);
    //use \r and flush to write the output on the same line
    float ax = x/COUNTS_PER_G;
    float ay = y/COUNTS_PER_G;
    float az = z/COUNTS_PER_G;

    roll = atan2(ay, az)*180/M_PI;
    pitch = atan2(ax, sqrt(ay*ay + az*az))*180/M_PI;
```

BUILD THE PROJECT

To build the simple.cpp project, use the following:

g++ ADXL345.cpp -o ADXL345

After the project is built, execute the file using the following:

./ADXL345

-End of the lab 4 Part 1

-Proceed to lab 4 Part 2 Accelerometer programming exercise using the Freedom Board