I2C Interface Design and Implementation

Ifunanya K. Nnoka, 010754159 Author

***Abstract*—This Document covers the technical details following the implementation of the I2C protocol as a means of interfacing a LSM303 accelerometer/magnetometer sensor with a Linux ARM 11 CPU Development board.**

Appendix—

I. Introduction

THE objective of this lab is to learn and execute an I2C communication interface between a microcomputer and a sensor device. We learn how i2c supporting devices communicate with a master/slave mannerism. We also learn about the two essential wires (SDA/SCL) that make the communication possible, we also get to see why the I2C protocol is different from protocols like SPI and serial ports. The main point is that with I2C communication only needs two major wires, and these two wires can host several slaves. We also learn the importance of data/bit transmission with respect to clock, It is critical that data is being sent/received in a specific order and specific edge of the clock.

Hardware Resources:

- Samsung SC6410 ARM 11 CPU
- LSM303DLHC Accelerometer/Magnetometer Sensor
- Linux PC
- Serial port wire

Software Resources:

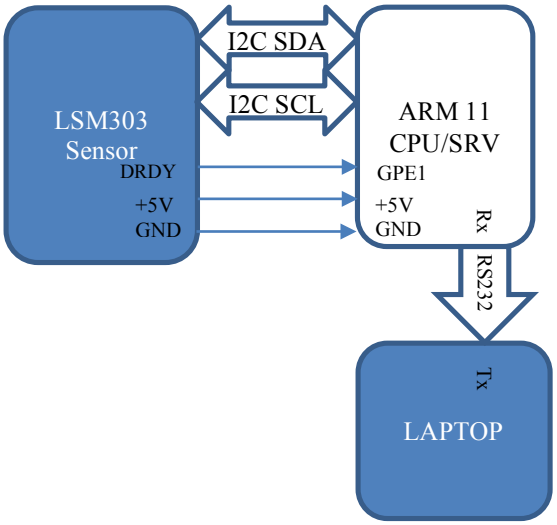
- Arm-linux-gcc cross compiler toolschain
- Linux 2.6.38 OS distribution

Deliverables:

- lab1b.c
- drdy_driver.c
- i2c (executable)
- drdy_driver.ko
- i2c-dev.ko
- 24cxx.c/.h
- Makefile

II. System and Hardware Design

A. System Block Diagram



B. Microprocessor System Block Diagram

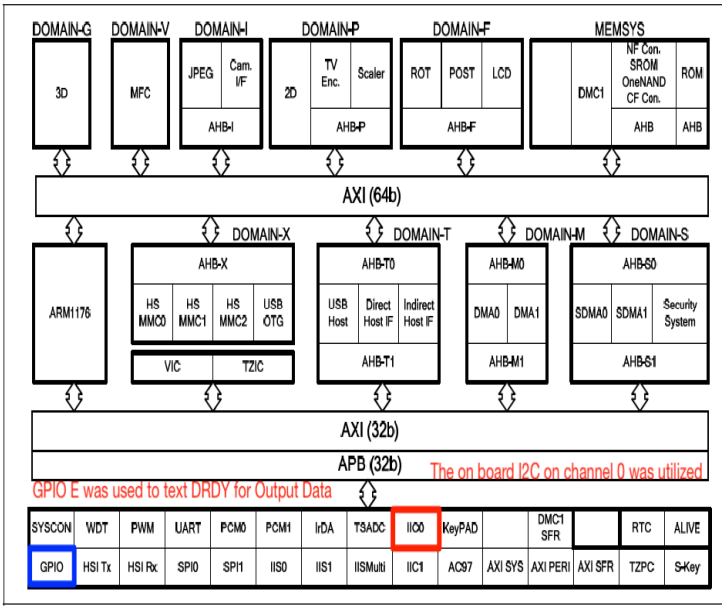
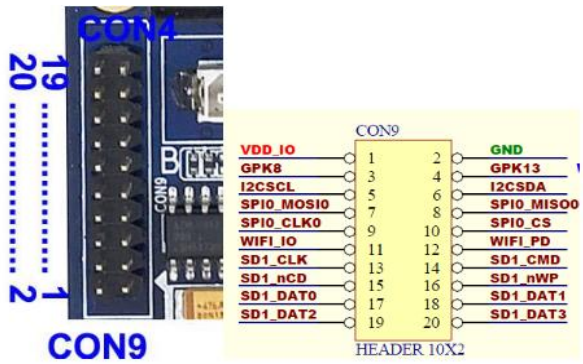
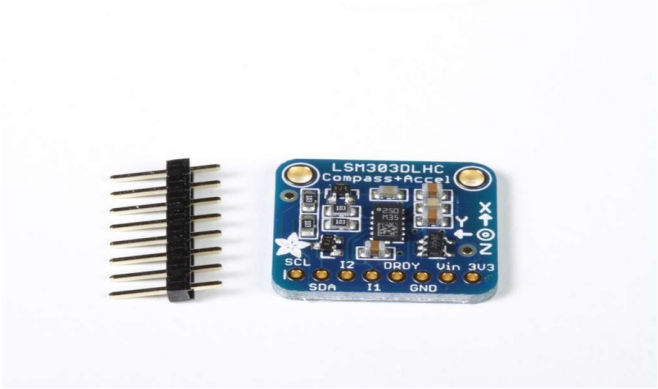


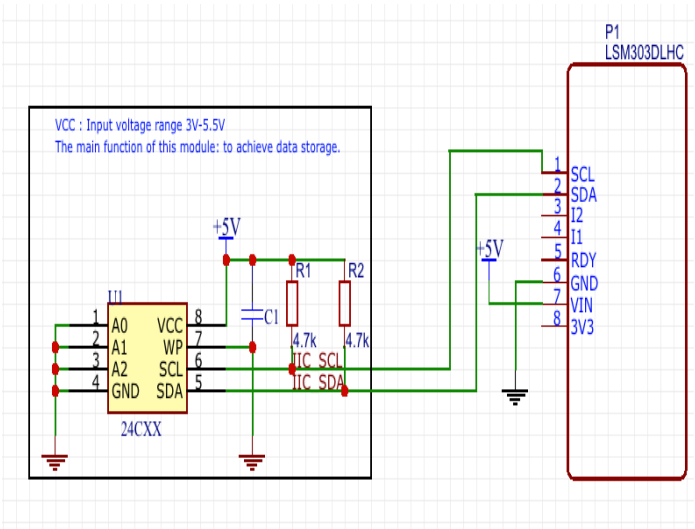
Figure 3-1. S3C6410X block diagram



I2C CPU Connector Pins.



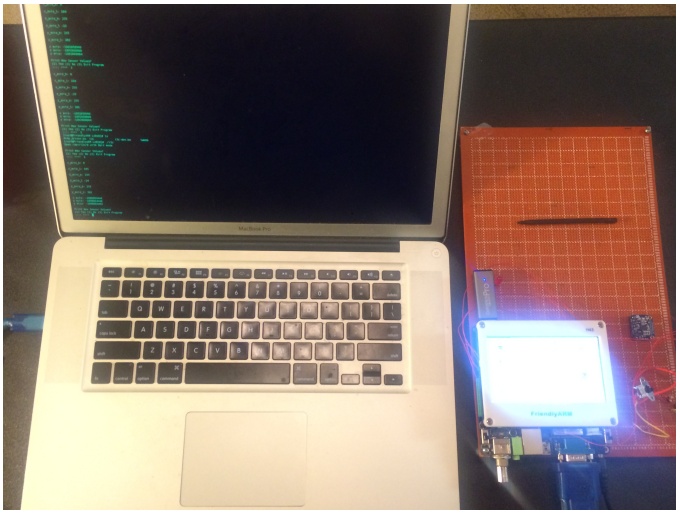
LSM303 Sensor /Pins.



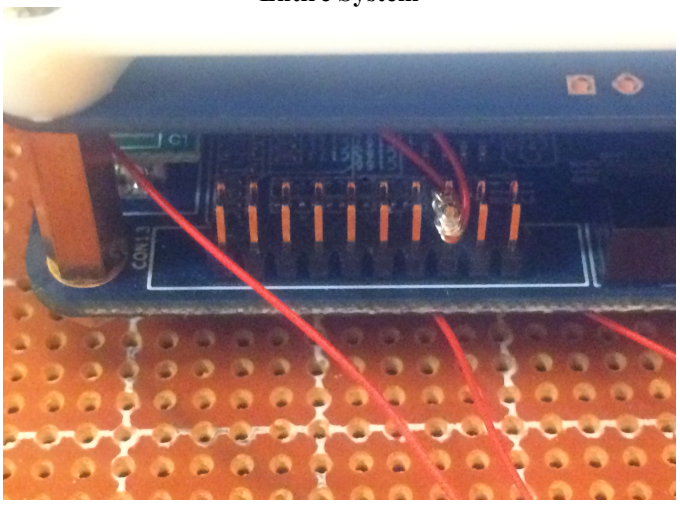
LSM303/EEPROM Electrical Connections

Master		Slave
Serial Clock (SCL)	Connects to	SCL
Serial Data (SDA)	Connects to	SDA
GPE1	Connects to	DRDY
5V	Connects to	Vin
GND	Connects to	GND

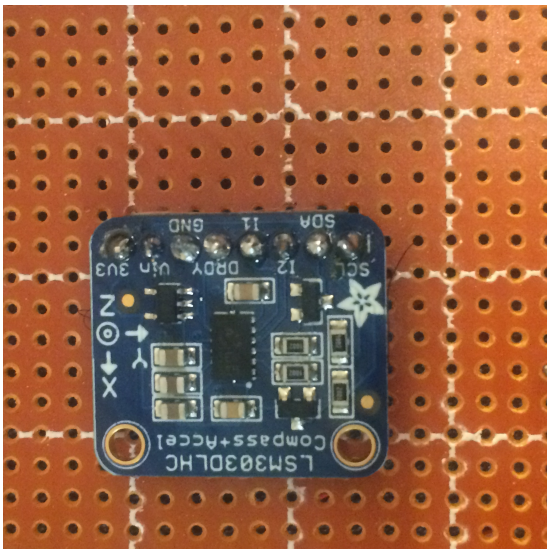
C. System Implementation



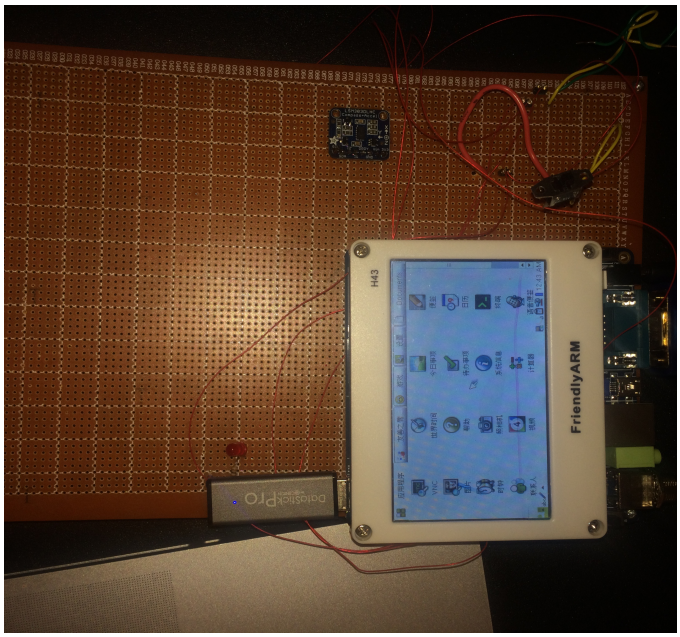
Entire System



I2C CPU Connectors (SDA/SCL)



LSM303 Sensor



Mini6410 (ARM 11 CPU)

III. SOFTWARE DESIGN

A. Software Development Environment

The software development environment required to build and implement this system is the arm Linux 2.6.38 OS. To build and compile source code on the kernel the arm-linux-gcc cross compiler is required.

B. Algorithm Description

- First thing required is the initialization of both the master and slave device. Master starts first by Setting up the right environment for the communication to occur. This includes:

  fd = open("/dev/i2c/0",O_RDWR); (Opening the i2c device driver file in RDWR mode. Next Master calls ioctl(fd,0x703,addr); in write mode to enable I2C_SLAVE read/write on the bus, the addr field is that of the desired sensor, when the sensor matches the address communication can begin. If master is using the smbus a call to iocntl is necessary to get required functions.

- Next, the master must write several bytes to the slave to initialize and configure the device to desired settings. The first byte sent to the slave by the master is the address of the sensor, this is only after a START bit has been transmitted. The start bit is a signal transition from high to low when the clock is held high.

- After the desired sensor address has been sent and the sensor sends an ACK. The master then sends the sub address it wishes to configure, followed by a byte of the value which corresponds to the desired configuration of that sub address. A stop bit is sent after each configuration data of the sub address has been sent.

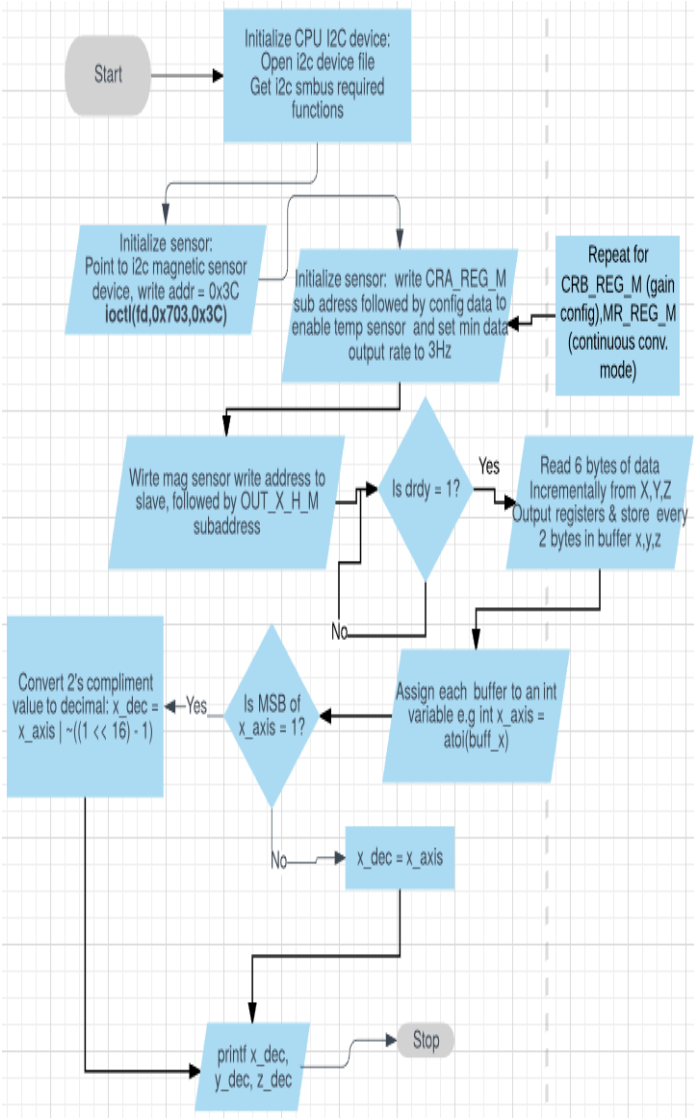
- If a user wished to configure multiple control registers without having to repeatedly go through the standard routine. The user must set the MSB of the 1st sub address to 1. This indicates to the sensor to auto increment the sub address after each data byte has been received. Thereby allowing a user to send multiple data bytes followed by a stop bit when all data bytes have been sent.

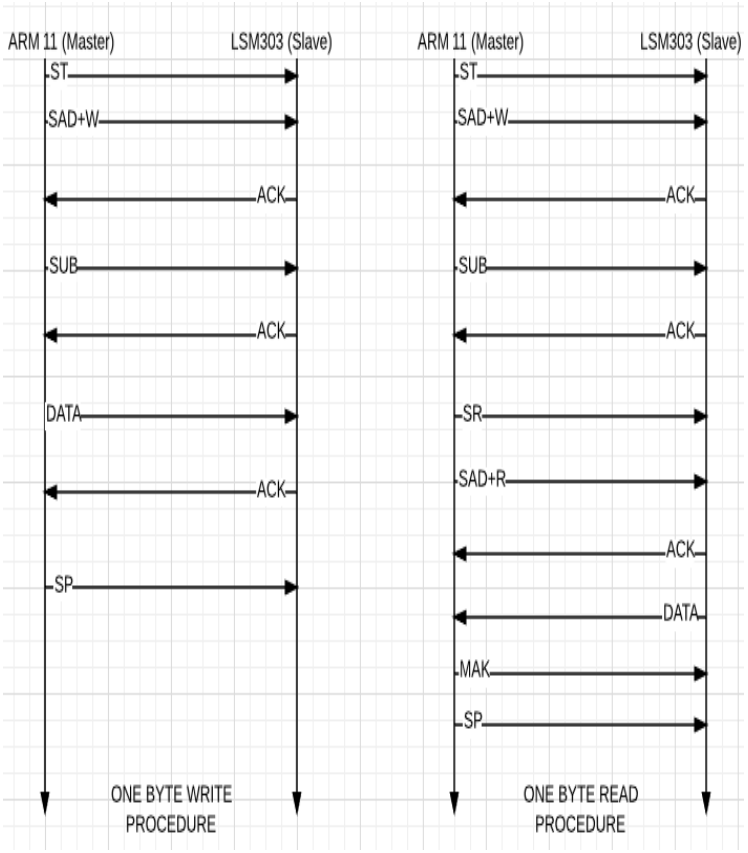
- To read a byte or multiple bytes from the slave, the master device must first call iocntl() in write mode in order to be able to send the sensor the desired control register address. After the slave has received the bytes of its write address, and sub address for one of its output registers, a repeat start bit is sent followed by the device read address. Only then can master begin to read from the slave, either a byte at a time or multiple bytes by incremental update of its output register address, this is done automatically by the magnetometer sensor. After the read is done a stop bit is sent.

- Six bytes are collected and stowed into a buffer for further processing. These 6 bytes represent the MSB and LSB of the magnetometer reading in the x,z,y axis. The first byte received is the MSB of the x-axis reading followed by the LSB of the x-axis, MSB of the z-axis, LSB of the z-axis, MSB of the y-axis and LSB of the y-axis.

- These 6 bytes are sent to the CPU by the sensor in 2's compliment form. So in other to display the reading in decimal, first, byte are shifted out of the buffer after every two readings and stowed into a separate buffer eg. x_h_l_m[] for x-axis etc. Next each of the two bytes are assigned to an integer variable e.g int x = atoi(x_h_l_m). Next each of these integers are converted from 2's compliment to decimal by first testing the MSB to see if it is '1' or '0', if it is a'0' then no further processing needs to be done, but if it's a '1', then 1 is subtracted and the bits are flipped. The resulting values are then printed to stdout.

C. Algorithm Flowchart





SPACE-TIME DIAGRAM

D. Pseudo-Code

//Initialize

- Point file descriptor to i2c device file: fd = open("/dev/i2c/0",O_RDWR);
- Check for required smbus functions : (io = ioctl(fd, 0x0705, &func); check_i2c_func();
- Point to i2c magnetic sensor address/device device io =ioctl(fd,0x703,0x1E); //I2C_SLAVE Address = 0x0703
- Define i2csmbus data structure. Structure.mode = write, structure.command = subaddress, structure.size = size of(control reg subaddress +data), structure.data = desired config data.
- Write to slave device: ioctl(fd,0x0720,&structure) //i2c_smbus_address = 0x0720
- Repeat bullet #4 and #5 for all required control registers.
- Clear the kernel before reading buffer: ioctl(fd, BLKFLSBUF);
- Write output register sub address to slave device before read: Structure.mode = write, structure.command = subaddress OUT_X_H_M, structure.size = size of(control reg subaddress +data), structure.data = NULL.

- Write to slave device:
  ioctl(fd,0x0720,&structure)
  //i2c_smbus_address = 0x0720
- Set up i2c smbus structure to read from output reg:
  Structure.mode = read, structure.command = x-axis output reg subaddress (0x00), structure.size = size of(output reg subaddress +data), structure.data = &data.
- Open drdy device driver: fd2 = open("/dev/drdy", 0);
- If (drdy = high){
  Count = 0;
  do all this while count < 6
  Clear the kernel: (ioctl(fd, BLKFLSBUF);
  read 12c bus: io = ioctl(fd,0x0720,&structure);
  store in buffer: char *point = buffer[100];
  point += sprintf(point,"%d",(0x0FF & data.byte));
  if (count == 1):
  copy the 2 bytes in buffer into x_axis buffer:
  strcpy(x_h_l_m,buffer);
  set pointer back to lowest index:point = &buffer[0];
  clear buffer: bzero(buff,sizeof(buff));
  if (count == 3):
  copy the 2 bytes in buffer into z_axis buffer:
  strcpy(z_h_l_m,buffer);
  set pointer back to lowest index:point = &buffer[0];
  clear buffer: bzero(buff,sizeof(buff));
  if (count == 5):
  copy the 2 bytes in buffer into y_axis buffer:
  strcpy(y_h_l_m,buffer);
  set pointer back to lowest index:point = &buffer[0];
  clear buffer: bzero(buff,sizeof(buff));
  count++
  }
- Convert string to integer value:
  int x_axis = atoi(x_h_l_m)
  int z_axis = atoi(z_h_l_m)
  int y_axis = atoi(y_h_l_m)
- If number is 2's complement convert before printf to stdout:
  neg_x = (x_axis AND ('1' shifted 15 bits to the left)) != 0;
  if (neg_x) (i.e if MSB == 1)
  x_dec = x_axis OR NOT(('1' shifted 16 bits to the left) - 1);
  else if MSB == 0
  x_dec = x_axis;
  (repeat for z_axis and y_axis)
- Printf(x_dec,z_dec,y_dec);
- Close file descriptors:
  close(fd2);
  close(fd);

E. *Source Code*

```
fprintf(stderr, "Open /dev/i2c/0 with 8bit mode\n");
die_if(eeprom_open("/dev/i2c/0", 0x1E, EEPROM_TYPE_8BIT_ADDR, &e) < 0,
       "unable to open eeprom device file "
       "(check that the file exists and that it's readable)"); //read/write bit is
          taken care of in driver, therefore addr is 1E
die_if(eeprom_write_byte(&e, 0x00, 0x88), "write error");//sub address CRA_REG_M
    0x00 //(1) Temperature sensor enables (2) 3Hz minimum data output rate config
die_if(eeprom_write_byte(&e, 0x01, 0x80), "write error");//sub address CRB_REG_M
    0x01 //gain config: +4,-4 input field range (gauss)
die_if(eeprom_write_byte(&e, 0x02, 0x00), "write error");//sub address MR_REG_M
    0x02 //continous conversion mode
```

I2C Init Config

```
fd2 = open("/dev/drdy", 0);
if (fd2 < 0) {
    perror("open device drdy");
    exit(1);
}
for(;;){
    printf("\nPrint New Sensor Values?\n(1) Yes (2) Exit Program\n");
    printf(":::: >>>>  ");
    scanf("%d",&p);
    if(p == 1){
    drdy = ioctl(fd2,0, NULL);
    if (drdy == 2){
        bzero(x_h_l_m,sizeof(x_h_l_m));
        bzero(z_h_l_m,sizeof(z_h_l_m));
        bzero(y_h_l_m,sizeof(y_h_l_m));
        bzero(buff,sizeof(buff));
        point = &buff[0];
        for (i=0, out_addr = 0x03; i<6; i++, out_addr++) {
            die_if((n = eeprom_read_byte(&e, out_addr)) < 0, "read error");
            buff[i] = n;
        }
        x_h_l_m[0] = buff[0];
        x_h_l_m[1] = buff[1];
        z_h_l_m[0] = buff[2];
        z_h_l_m[1] = buff[3];
        y_h_l_m[0] = buff[4];
        y_h_l_m[1] = buff[5];
    x_axis =  concatenate(x_h_l_m[0], x_h_l_m[1]);
    y_axis =  concatenate(y_h_l_m[0], y_h_l_m[1]);
    z_axis =  concatenate(z_h_l_m[0], z_h_l_m[1]);
```

Sensor Reading

```
        x_axis = atoi(x_h_l_m);
            const int neg_x = (x_axis & (1 << 15)) != 0;
        int x_dec;
        if (neg_x)
        x_dec = x_axis | ~((1 << 16) - 1);
        else
        x_dec = x_axis;

 z_axis = atoi(z_h_l_m);
            const int neg_z = (z_axis & (1 << 15)) != 0;
        int z_dec;
        if (neg_z)
        z_dec = z_axis | ~((1 << 16) - 1);
        else
        z_dec = z_axis;

 y_axis = atoi(y_h_l_m);
            const int neg_y = (y_axis & (1 << 15)) != 0;
        int y_dec;
        if (neg_y)
        y_dec = y_axis | ~((1 << 16) - 1);
        else
        y_dec = y_axis;

             printf("\nx axis 2's comp: %d\n",x_axis);
             printf("z axis 2's comp: %d\n",z_axis);
             printf("y axis 2's comp: %d\n",y_axis);
         printf("\n\nx axis magnetometer reading: %d\n",x_dec);
             printf("z axis magnetometer reading: %d\n",z_dec);
             printf("y axis magnetometer reading: %d\n",y_dec);
        }
        }
        if (p==2) {
            goto exit;
        }
    }
exit:
    eeprom_close(&e);
```

2's Compliment Conversion

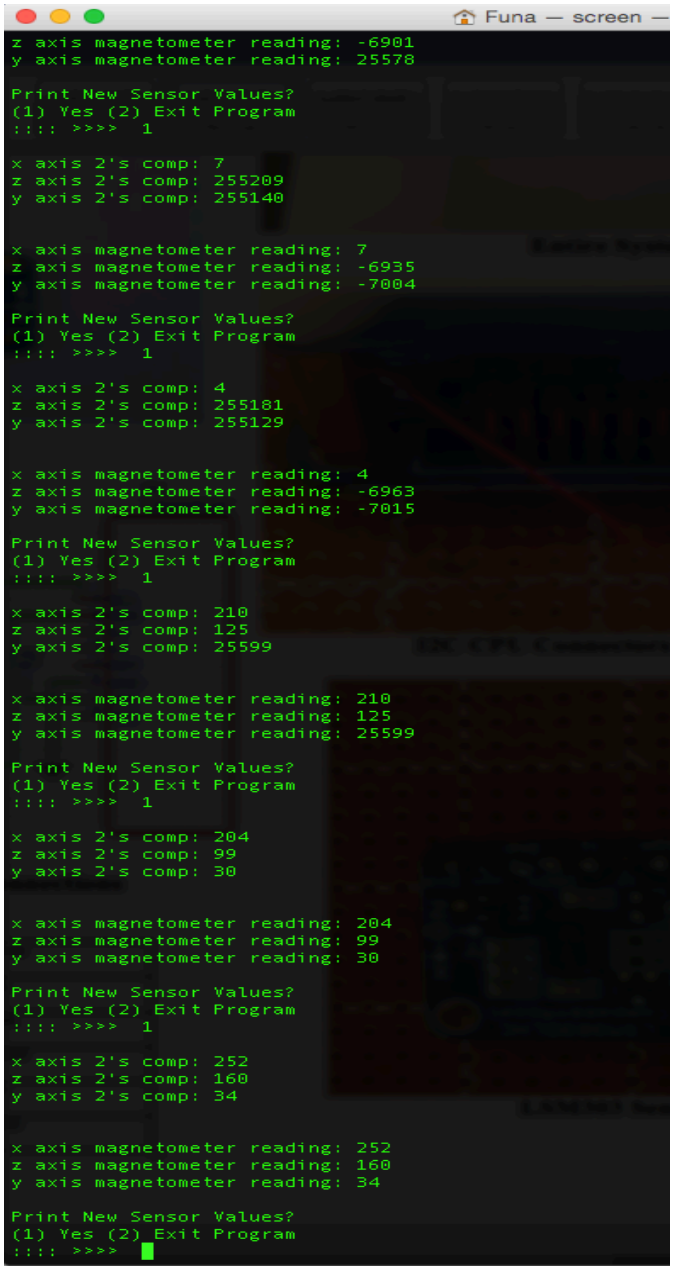
IV. Testing and Verification

A. Hardware Testing

	LOW	HIGH
SCL	3.61	3.64
SDA	0.01	3.64

When testing both pins on the sensor. SCL only toggled between 3.64V – 3.61V upon program execution, this might be due to the change in pin state occurring so fast that my voltmeter is unable to catch/detect it. SDA pin values ranged from 3.64 Volts to 0.01V upon program execution.

B. Software Testing



Program Execution Output

Program Execution Output

References

[1] W.J Kim Author, "Memory Map," in *S3C6410X User Manual RISC Microprocessor REV 1.10*. Republic of Korea: *Aug 22, 2008*, ch. *2*, sec. *2.2*, pp. *119*

[2] W.J Kim Author, "System Controller," in *S3C6410X User Manual RISC Microprocessor REV 1.10*. Republic of Korea: *Aug 22, 2008*, ch. *3*, sec. *3.3.3*, pp. *121*

[3] W.J Kim Author, "GPIO" in *S3C6410X User Manual RISC Microprocessor REV 1.10*. Republic of Korea: *Aug 22, 2008*, ch. *10*, sec. *10.1 – 10.5*, pp. *306 -322*

[4] W.J Kim Author, "IIC," in *S3C6410X User Manual RISC Microprocessor REV 1.10*. Republic of Korea: *Aug 22, 2008*, ch. *30*.

[5] LSM303DLHC DATASHEET

[6] Mini6410 Hardware manual

[7] "Mini6410 Hardware Specification", geeetech.com