Testing of I2C on ARM-11 Development Board

Sagar Shah
Electrical Engineering Department, Charles W. Davidson College of Engineering
San Jose State University, San Jose, CA 95112
Email: sagar.shah@sjsu.edu

Abstract

This paper describes design and development details of hardware and software components required for testing and verification of I2C using Samsung S3C6410 ARM-11 board. To achieve this, we connect prototype board to the development board. The prototype board has a voltage divider and an op-amp stage soldered to it. An I2C sensor will be mounted on a PCB. The output will be available at the laptop screen. The aim of this lab is to interface the Samsung ARM-11 board with that of the LSM303 sensor using I2C protocol. The LSM303 mounted on PCB will give output based on the motion of the PCB.

Introduction

Samsung TINY6410 is an ARM-11 based development board. It is powered with Samsung S3C6410 SoC. It is capable of running full fledge operating system. it has got different types of I/O ports like IR, USB, Audio, Ethernet, Camera, TVout, 40 pin system bus, 30 pin GPIO, 20 pin SDIO, etc. It runs Linux operating system. We are going to use GPIO pins to interface with the external prototype board. Software program running on ARM board has two components. Its main driver that actually interfaces with I2C pin runs in kernel mode. User level application program continuously reads the digital value of the voltage connected to ARM pin. Basic components involved in the design are computer, ARM-11 board, prototype board and power adapter. Computer is connected to ARM-11 board using USB to serial connecter. ARM-11 board is connected to LSM303 board.

Technical Challenges

The technical challenges involved in this project are both software and hardware in nature. On the hardware side we have to establish a connection between the development board and prototype board. This was quite demanding as we had to ensure that both the ground of the prototype board and development board were connected together, to avoid the CON1 pin from being damaged. On the software side the main challenge was to configure the data and control registers properly to ensure smooth connection and communication. The application program and driver program has to be linked. There were a few issues involved in this. We also had to make sure that the modules are properly compiled.

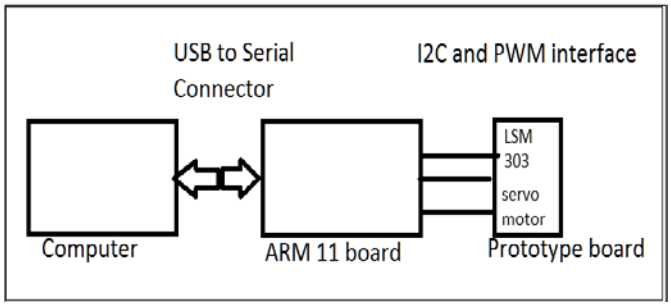


Figure 1: - SYSTEM ARCHITECTURE BLOCK DIAGRAM

Objectives

1.To write a driver program and an application program for the ARM-11,CPU.

2.To obtain the result of LSM303 on Computer screen..

Requirements

Considering the design objectives in the previous section the requirement for this experiment are divided in into both hardware and software.

The hardware requirements are as follows.

1. Samsung ARM 11 Development board.
2. USB to Serial Converter and cable.
3. Wire wrapping prototype board.

For the software requirement, we need the following:

1. Linux Ubuntu.
2. ARM Linux toolchain.
3. Putty.

SAMSUNG ARM-11 Board

Samsung ARM-11 Board is powered by Samsung S3C6410 SoC. This board has 30 pin GPIO port. Out of which we are going to use 2 pins viz., I2C SDA (Pin 20) and SCL (Pin 19) and GND (Pin 2 on JTAG). GND is connected to common ground. This board runs Linux operating system on it. So to interface with I2C pins, we need to write a device driver module. In the further sections the report explains no how to write device driver software for the same and user level application program to read and validate I2C data. However, in this section, hardware details of Samsung S3C6410 ARM-11 board are explained. Figure 4 shows the picture of that board. And Figure 5 shows picture of CPU module on that board. As you can see in Figure 5, it has two different types of connectors mounted on it, viz., CON1 and PORT2. We are going to use PORT2 as it has I2C pins of our interest. CPU module on this board is detachable, and detailed picture of CPU module is shown in Figure 3.

Figure 2. Samsung S3C6410 ARM-11 Board

CON1.1	VDD_IO(3.3V)	CON1.2	GND
CON1.3	GPE1	CON1.4	GPE2
CON1.5	GPE3	CON1.6	GPE4
CON1.7	GPM0	CON1.8	GPM1
CON1.9	GPM2	CON1.10	GPM3
CON1.11	GPM4	CON1.12	GPM5
CON1.13	GPQ1	CON1.14	GPQ2
CON1.15	GPQ3	CON1.16	GPQ4
CON1.17	GPQ5	CON1.18	GPQ6
CON1.19	SPICLK0	CON1.20	SPIMISO0
CON1.21	SPICS0	CON1.22	SPIMOSI0
CON1.23	EINT6	CON1.24	EINT9
CON1.25	EINT11	CON1.26	EINT16
CON1.27	EINT17	CON1.28	AIN2
CON1.29	AIN3	CON1.30	DACOUT1

Table 1. Pin Configuration for CON1 Port on CPU Board

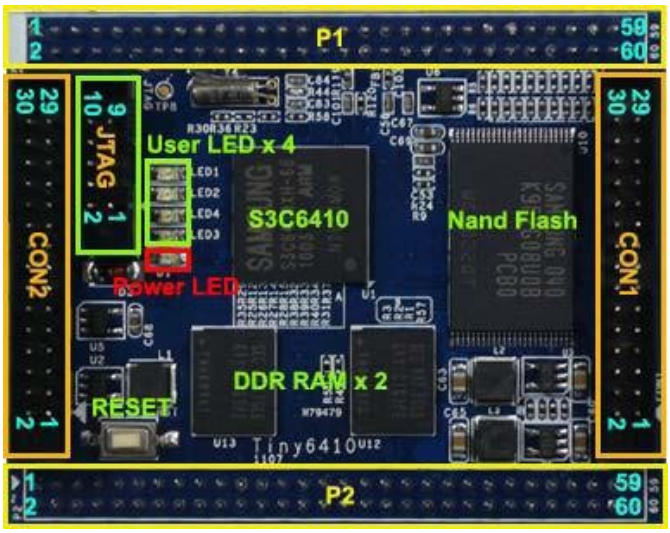


Figure 3. S3C6410 CPU Module

P2	Pin define	P2	Pin define
P2.1	OM3	P2.2	OM4
P2.3	M_nRESET	P2.4	VDD_RTC
P2.5	RTSn1	P2.6	CTSn1
P2.7	TXD0	P2.8	RXD0
P2.9	TXD1	P2.10	RXD1
P2.11	TXD2	P2.12	RXD2
P2.13	TXD3	P2.14	RXD3
P2.15	SPIMOSI	P2.16	SPIMISO
P2.17	SPICLK	P2.18	SPICS
P2.19	I2CSCL	P2.20	I2CSDA
P2.21	SD0_CLK	P2.22	SD0_CMD
P2.23	SD0_nCD	P2.24	SD0_nWP
P2.25	SD0_DAT0	P2.26	SD0_DAT1
P2.27	SD0_DAT2	P2.28	SD0_DAT3
P2.29	AC97_BITCLK	P2.30	AC97_RSTn
P2.31	AC97_SYNC	P2.32	AC97_SDO
P2.33	AC97_SDI	P2.34	XEINT12
P2.35	ADDR0	P2.36	ADDR1
P2.37	ADDR2	P2.38	ADDR3
P2.39	nCS1	P2.40	XEINT7
P2.41	nWAIT	P2.42	nESET
P2.43	LnWE	P2.44	LnOE
P2.45	DATA0	P2.46	DATA1
P2.47	DATA2	P2.48	DATA3

P2.49	DATA4	P2.50	DATA5
P2.51	DATA6	P2.52	DATA7
P2.53	DATA8	P2.54	DATA9
P2.55	DATA10	P2.56	DATA11
P2.57	DATA12	P2.58	DATA13
P2.59	DATA14	P2.60	DATA15

Table 2. Pin Configuration for P2 Port on CPU Board

H/W DESIGN

Hardware design has three major components, viz., LSM303, PCB and ARM-11 Board. This section explains implementation details of prototype board and interfacing I2C pin on ARM-11 Board.

Component	Description	Count
Power Adapter and Connector J1	Input: 110V AC Output: 5V DC Current: 1200 mA	1
LSM303	Magnetometer/Accelero meter	1
Samsung ARM-11	Platform of work	1

Table 3. Bill of Materials

KERNEL MODULE ALGORITHM FOR I2C

Kernel module has four different routines. One routine initializes the driver module. Second routine is invoked whenever I2C device is opened through a user application. Third routine is invoked whenever read () system call is issued by user application on that device. And last routine is interrupt handler which is invoked when analog to digital conversion is complete. This init () module is invoked when this kernel module is inserted using insmod. Building and running of these software modules is explained in later section.

Algorithm for init () routine of kernel module:

A. Initialize I2C

B. Enable I2C clock

C. Register interrupt handler

D. Register driver and open (), read () callbacks

E. Return

Algorithm for open () handler routine of kernel module:

A. Initialize the wait queue for the device

B. Select channel 2 as I2C input

C. Select Prescale

D. Return

Algorithm for read () routine of kernel module:

A. Acquire I2C I/O

B. Start I2C conversion

C. Wait for conversion complete interrupt to return

D. Read I2C value

E. Release I2C I/O

F. Send I2C value to user program

G. Return

Algorithm for int_handler () routine of kernel module:

A. Read I2CCON register

B. Store converted I2C value to local variable

C. Clear interrupt

D. Return

KERNEL MODULE PSEUDO CODE FOR I2C

```
int main()
{
   struct lsm303 e;
   int recvData[8]={0}, ret=0, i=0;

while(1){
        magread();
        sleep(2);
}
return 0;
}
```

OPERATION OF SLAVE RECEIVER MODE OF I2C WITH SAMSUNG ARM-11

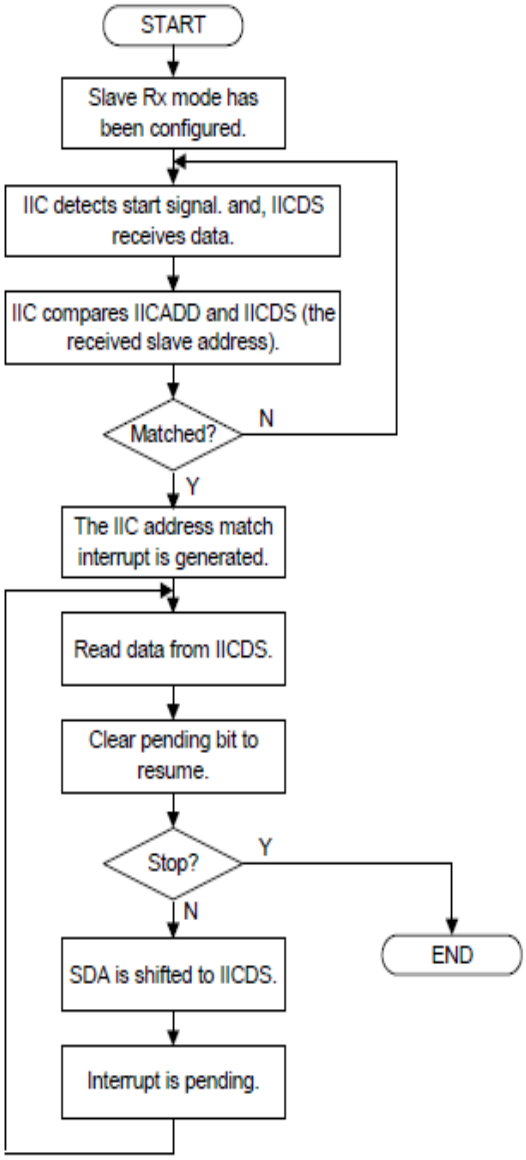


Figure 8. Operation of I2C in Slave Receiver mode

MULTIMASTER I2CCON REGISTER

30.11.1 MULTI-MASTER IIC-BUS CONTROL (IICCON) REGISTER

Register	Address	R/W	Description	Reset Value
IICCON	0x7F004000	R/W	IIC Channel 0 Bus control register	0x0X
	0x7F00F000	R/W	IIC Channel 1 Bus control register	0x0X

IICCON	Bit	Description	Initial State
Acknowledge generation (1)	[7]	IIC-bus acknowledge (ACK) enable bit. 0: Disable 1: Enable In Tx mode, the IICSDA is free in the ACK time. In Rx mode, the IICSDA is L in the ACK time.	0
Tx clock source selection	[6]	Source clock of IIC-bus transmit clock prescaler selection bit. 0: IICCLK = PCLK /16 1: IICCLK = PCLK /512	0
Tx/Rx Interrupt (5)	[5]	IIC-Bus Tx/Rx interrupt enable/disable bit. 0: Disable, 1: Enable	0
Interrupt pending flag (2) (3)	[4]	IIC-bus Tx/Rx interrupt pending flag. This bit cannot be written to 1. When this bit is read as 1, the IICSCL is tied to L and the IIC is stopped. To resume the operation, clear this bit as 0. 0: 1) No interrupt pending (when read). 2) Clear pending condition & Resume the operation (when write). 1: 1) Interrupt is pending (when read) 2) N/A (when write)	0
Transmit clock value (4)	[3:0]	IIC-Bus transmit clock prescaler. IIC-Bus transmit clock frequency is determined by this 4-bit prescaler value, according to the following formula: Tx clock = IICCLK/(IICCON[3:0]+1).	Undefined

SOFTWARE DESIGN

Software implementation required for this experiment is divided into two parts, viz., Kernel Driver Module and User Application Program, ARM-11 board runs Linux Operating System on it. So user level applications cannot access hardware directly. Hardware access is allowed to programs running in kernel mode only. So to gain the hardware access, we need to write a kernel device driver module for interfacing with I2C pins. And user level application communicates with this kernel driver via system calls like open, read, etc. There are two different I2C pins SDA pin 20 and SCL pin 19. Pin 2 which is GND pin, is connected to GND on prototype board. I2C protocol establishes communication between the LSM303 and the ARM-11 board using these two pins. The movement of the PCB board ensures varying values of accelerometer and magnetometer which are displayed on the terminal and with help of print command we are displaying the output.

BUILDING AND RUNNING SOFTWARE MODULE

Device driver program are generally located inside kernel source code. For our distribution, source for this driver module is written inside "drivers/char" directory of kernel source code. After kernel module source code is saved inside the specified directory, we need to configure kernel configuration so as to build our module. For this add following lines in the file "drivers/char/Kconfig":

Config.

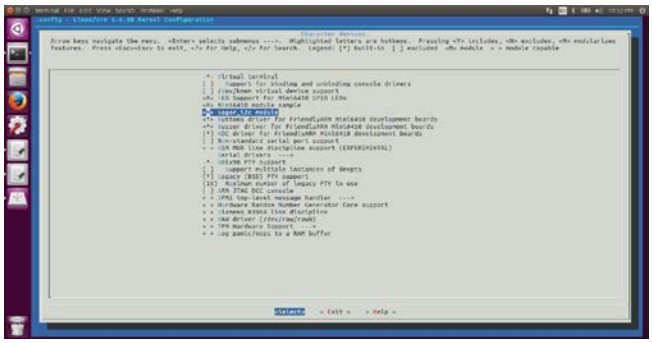
MINI6410_i2c_sagar

tristate "sagar_i2c module"

depends on CPU_S3C6410
help

EE 242 MINI6410 LAB1 Module

Figure 9. Kernel Configuration to Build Module



After this run "make menuconfig" inside kernel source directory. This will open a kernel configuration menu as shown in Figure 9. In this menu, select "Device Drivers" -> "Character Devices" option. After selecting this option, in the next screen you will be able to see name of your device driver module i.e. "sagar_i2c module". Select this module by pressing "Space" button on your keyboard. Press "Space" until you see "M" infront for device name. This will tell kernel to create a kernel module for your device driver program. After module is configured, we need to update the "Makefile" for that module. For this, open "drivers/char/Makefile" and add following line to it:

obj -$(CONFIG_MINI6410_LAB2_MODULE) += mini6410_i2c_sag.o

Save above Makefile and go to kernel source directory. Then run command "make modules". This will build the kernel module for your driver inside "drivers/char" directory. The compiled file "mini6410_i2c_sag.ko" will be generated in the same directory. Figure 10 shows how to compile module.

To build the application program, no special command is needed. Cross compile that application program using gcc and arm toolchain. Copy generated binary and mini6410_i2c_sag. ko file to ARM-11 board.

On the board, install the driver module using following command:

insmod mini6410_i2c_sag.ko

This will install driver module inside the linux running on ARM board. You can verify whether module is installed or not using "lsmod" command. This command will list all the modules installed in the system. Note that you have to do every time after reboot.

LSM303 Source code

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <linux/fs.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <assert.h>
#include <string.h>
#include <getopt.h>
#include <errno.h>
#include <sys/stat.h>
#include "24cXX.h"
#define CTRL_REG1_A
        (0x20)
#define CTRL_REG2_A
        (0x21)
#define CTRL_REG3_A
        (0x22)
#define CTRL_REG4_A
        (0x23)
#define CTRL_REG5_A
        (0x24)
#define CTRL_REG6_A
        (0x25)
#define OUT_X_L_A
        (0x28)
#define OUT_X_H_A
        (0x29)
```

```
#define OUT_Y_L_A                       (0x2A)
#define OUT_Y_H_A                       (0x2B)
#define OUT_Z_L_A                       (0x2C)
#define OUT_Z_H_A                       (0x2D)
#define CRA_REG_M               (0x00)
#define CRB_REG_M               (0x01)
#define MR_REG_M                (0x02)
#define OUT_X_H_M               (0x03)
#define OUT_X_L_M               (0x04)
#define OUT_Z_H_M               (0x05)
#define OUT_Z_L_M               (0x06)
#define OUT_Y_H_M               (0x07)
#define OUT_Y_L_M               (0x08)
#define SR_REG_M                (0x09)
#define TEMP_OUT_H_M                    (0x31)
#define TEMP_OUT_L_M                    (0x32)
#define LSM_303_ACCEL_ADDR              (0x19)
#define LSM_303_MAGNET_ADDR  (0x1E)
#define usage_if(a) do { do_usage_if( a , __LINE__); } while(0);
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,comma
nd,
                        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,comman
d,

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,comma
nd,
                        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,co
mmand,

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,com
mand,

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, __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,comma
nd,
 I2C_SMBUS_BLOCK_DATA, &data);
}
#define CHECK_I2C_FUNC( var, label ) \
        do {    if(0 == (var & label)) { \
                fprintf(stderr, "\nError: " \
                        #label " function is required.
Program halted.\n\n"); \
exit(1); } \
        } while(0);
int lsm303_open(char *dev_fqn, int addr, struct
lsm303* e)
{
        int funcs, fd, r;
        e->fd = e->addr = 0;
        e->dev = 0;

        fd = open(dev_fqn, O_RDWR);
        if(fd <= 0)
        {
fprintf(stderr, "Error lsm303_open: %s\n",
strerror(errno));
                return -1;
        }
// get funcs list
        if((r = ioctl(fd, I2C_FUNCS, &funcs) <
0))
        {
fprintf(stderr, "Error ioctl I2C_FUNCS: %s\n",
strerror(errno));
                return -1;
        }
// check for req funcs
CHECK_I2C_FUNC( funcs,
I2C_FUNC_SMBUS_READ_BYTE );
CHECK_I2C_FUNC( funcs,
I2C_FUNC_SMBUS_WRITE_BYTE );
CHECK_I2C_FUNC( funcs,
I2C_FUNC_SMBUS_READ_BYTE_DATA );
CHECK_I2C_FUNC( funcs,
I2C_FUNC_SMBUS_WRITE_BYTE_DATA );
CHECK_I2C_FUNC( funcs,
I2C_FUNC_SMBUS_READ_WORD_DATA );
CHECK_I2C_FUNC( funcs,
I2C_FUNC_SMBUS_WRITE_WORD_DATA );
// set working device
        if( ( r = ioctl(fd, I2C_SLAVE, addr)) < 0)
        {
fprintf(stderr, "Error eeprom_open: %s\n",
strerror(errno));
                return -1;
        }
        e->fd = fd;
        e->addr = addr;
        e->dev = dev_fqn;
        return 0;
}
int lsm303_close(struct lsm303 *e)
{
```

```
close(e->fd);
e->fd = -1;
e->dev = 0;
	return 0;
}


#if 0
int eeprom_24c32_write_byte(struct lsm303 *e,
__u16 mem_addr, __u8 data)
{
	__u8 buf[3] = { (mem_addr >> 8) & 0x00ff,
mem_addr & 0x00ff, data };
	return i2c_write_3b(e, buf);
}
int eeprom_24c32_read_current_byte(struct lsm303*
e)
{
	ioctl(e->fd, BLKFLSBUF); // clear kernel
read buffer
	return i2c_smbus_read_byte(e->fd);
}
int eeprom_24c32_read_byte(struct lsm303* e, __u16
mem_addr)
{
	int r;
	ioctl(e->fd, BLKFLSBUF); // clear kernel
read buffer
	__u8 buf[2] = { (mem_addr >> 8) & 0x0ff,
mem_addr & 0x0ff };
	r = i2c_write_2b(e, buf);
	if (r < 0)
		return r;
	r = i2c_smbus_read_byte(e->fd);
	return r;
}
#endif

#if 0
int lsm303_read_current_byte(struct lsm303* e)
{
	ioctl(e->fd, BLKFLSBUF); // clear kernel
read buffer
	return i2c_smbus_read_byte(e->fd);
}


int lsm303_read_byte(struct lsm303* e, __u16
mem_addr)
{
	int r;
	ioctl(e->fd, BLKFLSBUF); // clear kernel
read buffer
	if(e->type ==
EEPROM_TYPE_8BIT_ADDR)
	{
		__u8 buf =  mem_addr & 0x0ff;
		r = i2c_write_1b(e, buf);
	} else if(e->type ==
EEPROM_TYPE_16BIT_ADDR) {
		__u8 buf[2] = { (mem_addr >> 8)
& 0x0ff, mem_addr & 0x0ff };
		r = i2c_write_2b(e, buf);
	} else {
		fprintf(stderr, "ERR: unknown
eeprom type\n");
		return -1;
	}
	if (r < 0)
		return r;
	r = i2c_smbus_read_byte(e->fd);
	return r;
}
```

```
int lsm303_write_byte(struct lsm303 *e, __u16
mem_addr, __u8 data)
{
        if(e->type ==
EEPROM_TYPE_8BIT_ADDR) {
                __u8 buf[2] = { mem_addr & 0x00ff,
data };
                return i2c_write_2b(e, buf);
        } else if(e->type ==
EEPROM_TYPE_16BIT_ADDR) {
                __u8 buf[3] =
                        { (mem_addr >> 8) & 0x00ff,
mem_addr & 0x00ff, data };
                return i2c_write_3b(e, buf);
        }
        fprintf(stderr, "ERR: unknown eeprom
type\n");
        return -1;
}


#endif


void do_usage_if(int b, int line)
{
        const static char *lsm303_usage =
                "I2C-LSM303 Program, ONLY FOR
TEST!\n";


        if(!b)
                return;


        fprintf(stderr, "%s\n[line %d]\n",
lsm303_usage, line);
        exit(1);
}
#define die_if(a, msg) do { do_die_if( a , msg,
__LINE__); } while(0);

void do_die_if(int b, char* msg, int line)
{
        if(!b)
                return;
        fprintf(stderr, "Error at line %d: %s\n",
line, msg);
        fprintf(stderr, "          sysmsg: %s\n",
strerror(errno));
        exit(1);
}
#if 0
static int read_from_lsm303(struct lsm303 *e, int
addr, int size)
{
        int ch, i;
        for(i = 0; i < size; ++i, ++addr)
        {
                die_if((ch = lsm303_read_byte(e,
addr)) < 0, "read error");
                if( (i % 16) == 0 )
                {
                        printf("\n %.4x|  ",
addr);
                }
                else if( (i % 8) == 0 )
                {
                        printf("  ");
                }

                printf("%.2x ", ch);
                fflush(stdout);
        }

        fprintf(stderr, "\n\n");
```

```
        return 0;
}
static int write_to_lsm303(struct lsm303 *e, int addr)
{
        int i;
        for(i=0, addr=0; i<256; i++, addr++)
        {
                if( (i % 16) == 0 )
                        printf("\n %.4x|  ", addr);
                else if( (i % 8) == 0 )
                        printf("   ");
                printf("%.2x ", i);
                fflush(stdout);
                die_if(lsm303_write_byte(e, addr, i),
"write error");
        }
        fprintf(stderr, "\n\n");
        return 0;
}

#endif
/*****************************************************
**********************************
 Main entry point

*****************************************************
**********************************/
int accread()
{
   struct lsm303 e;
   int recvData[8]={0}, ret=0, i=0;

        ret = lsm303_open("/dev/i2c/0",
LSM_303_ACCEL_ADDR, &e);
        if(ret < 0)
        {
                printf("Unable to open
Accelerometer device file \n");
                return -1;
        }
// Enable Accelerometer
ret = i2c_smbus_write_byte_data(e.fd ,
CTRL_REG1_A, 0x27);
        if(ret<0)
        {
                printf("Error writing data\n");
                return -1;
        }
ret = i2c_smbus_write_byte_data(e.fd,
CTRL_REG4_A, 0x28);
        if(ret<0)
        {
        printf("Error writing data\n");
                return -1;
        }
// Read data from Acceleromemter
recvData[0] = i2c_smbus_read_byte_data(e.fd,
OUT_X_L_A);
recvData[1] = i2c_smbus_read_byte_data(e.fd,
OUT_X_H_A);
recvData[2] = i2c_smbus_read_byte_data(e.fd,
OUT_Y_L_A);
recvData[3] = i2c_smbus_read_byte_data(e.fd,
OUT_Y_H_A);
recvData[4] = i2c_smbus_read_byte_data(e.fd,
OUT_Z_L_A);
recvData[5] = i2c_smbus_read_byte_data(e.fd,
OUT_Z_H_A);
for (i = 0 ; i < 6 ; i++)
        {
printf("Received Accelerometer data is
%d\n",recvData[i]);
```

```
        }
 lsm303_close(&e);
        return (0);
}
int magread()
{
   struct lsm303 e;
   int  ret=0, i=0;
   signed int recvData[8]={0},
xaxis=0,yaxis=0,zaxis=0,temper=0;

/**********************Magnetometer*********
*****************************/
ret = lsm303_open("/dev/i2c/0",
LSM_303_MAGNET_ADDR, &e);
   if(ret < 0)
   {
      printf("Unable to open Magnetometer device file
\n");
      return;
   }
ret = i2c_smbus_write_byte_data(e.fd,
CRA_REG_M, 0x94);
   if(ret<0)
   {
      printf("Error writing data %d\n",__LINE__);
      return;
   }

   ret = i2c_smbus_write_byte_data(e.fd,
MR_REG_M, 0x00);
   if(ret<0)
   {
      printf("Error writing data %d\n",__LINE__);
      return;
}
   // Read data from Magnetometer
recvData[0] = i2c_smbus_read_byte_data(e.fd,
OUT_X_H_M);
recvData[1] = i2c_smbus_read_byte_data(e.fd,
OUT_X_L_M);
recvData[2] = i2c_smbus_read_byte_data(e.fd,
OUT_Y_H_M);
recvData[3] = i2c_smbus_read_byte_data(e.fd,
OUT_Y_L_M);
recvData[4] = i2c_smbus_read_byte_data(e.fd,
OUT_Z_H_M);
recvData[5] = i2c_smbus_read_byte_data(e.fd,
OUT_Z_L_M);
recvData[6] = i2c_smbus_read_byte_data(e.fd,
TEMP_OUT_H_M);
recvData[7] = i2c_smbus_read_byte_data(e.fd,
TEMP_OUT_L_M);

xaxis = recvData[0] * 256 + recvData[1];
yaxis = recvData[2] * 256 + recvData[3];
zaxis = recvData[4] * 256 + recvData[5];
temper = recvData[6] * 256 + recvData[7];

    printf(" Received Magnetometer Reading For
X-axis: %d \n Received Magnetometer Reading
For Y-axis: %d \n Received Magnetometer
Reading For Z-axis: %d \n Received
Magnetometer Reading For Temperature: %d
\n",xaxis,yaxis,zaxis,temper);
        lsm303_close(&e);
        return 0;
}
```

TESTING AND VERIFICATION

System setup for carrying out the experiment is shown in Figure below shows the development board and LSM303 mounted on PCB. Figure 10 and Figure 11 shows screenshot of program output. program reads many different values of LSM303 based on the output from the prototype board.

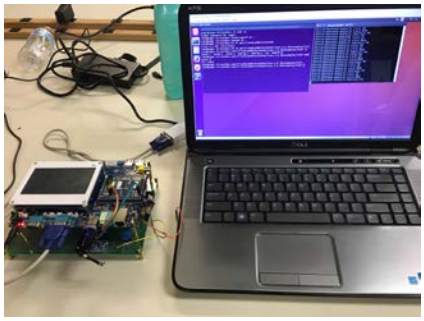


Figure 10. Setup of the System

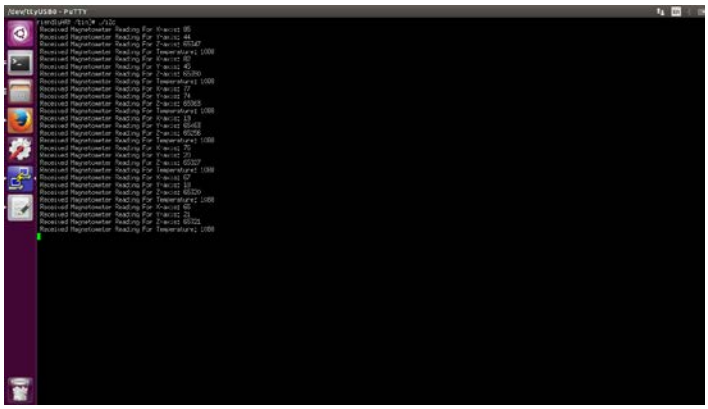


Figure11.Screenshot of Readings

CONCLUSION

The integration of hardware and software of the system done successfully. The interfacing of LSM303 with the Samsung ARM-11 development board successfully completed. The LSM303 mounted on the PCB is done and readings from the LSM303 taken successfully. The observations made are recorded successfully.

ACKNOWLEDGMENT

The work described in this paper was made possible and achievable by the contribution of Dr. Harry Li. The electrical and electronic components were made available by Adafruit and Anchor Electronics

REFERENCES

1.Datasheet for S3C6410
2.Datasheet for LSM303
3.CMPE 242- Embedded Hardware System Design Lecture Notes of Dr. Harry Li.