

# **REAL TIME SELF BALANCING ROBOT**

## **Team Members:**

Aniket Kumar Lata

Ashwin Raman

Extended Lab/Final Project  
ECEN 5623 REAL TIME EMBEDDED SYSTEMS

August 08, 2015

## Table of Contents

<b>1. INTRODUCTION</b>	3
<b>2. DESIGN OVERVIEW</b>	4
<b>2.1. PRINCIPLES OF OPERATION</b>	4
<b>2.2 DATA FLOW DIAGRAM</b>	5
<b>3. DETAILED DESIGN</b>	6
<b>3.1 HARDWARE AND FIRMWARE</b>	6
<b>3.1.1 I2C</b>	7
<b>3.1.2 ACCELEROMETER</b>	9
<b>3.1.3 GYROSCOPE</b>	15
<b>3.1.4 DC MOTOR AND CONTROLLER</b>	17
<b>3.2 SOFTWARE</b>	18
<b>3.2.1 CODE INTEGRATION</b>	18
<b>3.2.2 REAL TIME SCHEDULER</b>	19
<b>4. TESTING PLANS AND RESULTS</b>	23
<b>4.1 FINAL RESULTS</b>	26
<b>5. CONCLUSIONS</b>	27
<b>6. REFERENCE</b>	28
<b>APPENDIX</b>	29

# 1. INTRODUCTION

The goal of this project is to make a robot that can balance itself, but has only two points of contact with the ground. Our aim is to accomplish this goal with the help of real time theory.

The basic idea is that you have a mass located above its pivot point. This causes the robot to be unstable, and without any help, it will quickly fall over. Sensors on the robot will take acceleration and gyroscope measurements, which are sent to a control algorithm. As the robot starts to fall, the control algorithm will send a signal to the motor, telling it which direction and how much to move in order to keep the robot upright.

The target platform for this project is the Beagleboard xM development board. We are running the Ubuntu distribution of the Linux kernel on this board.

## 2. DESIGN OVERVIEW

### 2.1 PRINCIPLE OF OPERATION

The system runs three services, one each for: accelerometer, gyroscope and DC motor. Each of the sensors and the motor is serviced periodically. The main program initializes threads for each of the services. The services are scheduled using a real time scheduler policy. The device drivers capture data from the two sensors. This data is accessed by the dc motor service and corrective acceleration is provided depending upon the angle of tilt.

The basics behind a balancing robot is based on the Inverted Pendulum concept. The goal is to have a control algorithm called Proportional Integral Derivative (PID) to keep the robot balanced by trying to keep the wheels under the center of gravity. Eg. If the robot leans forwards, the wheels spin forward trying correct the lean.

**PID** is a control loop feedback mechanism(controller) widely used in industrial control systems. A PID controller calculates an *error* value as the difference between a measured process variable and a desired setpoint. The controller attempts to minimize the *error* by adjusting the process through use of a manipulated variable.

Simply speaking

- If the robot is tilting forwards, move the wheels in the forward direction.
- Else, if the robot is tilting backwards, move the wheels in the backward direction.
- Else do nothing.
- Repeat.

The PID controller takes into account the *amount of tilt* and applies a proportionally large amount of force on the wheels. The algorithm ideally takes three arguments; present error, sum of accumulated errors and expected future error

**Proportional** – (present error) the present error is the amount of tilt that the robot has

**Integral** – sum of accumulated errors. Average of errors should be zero.

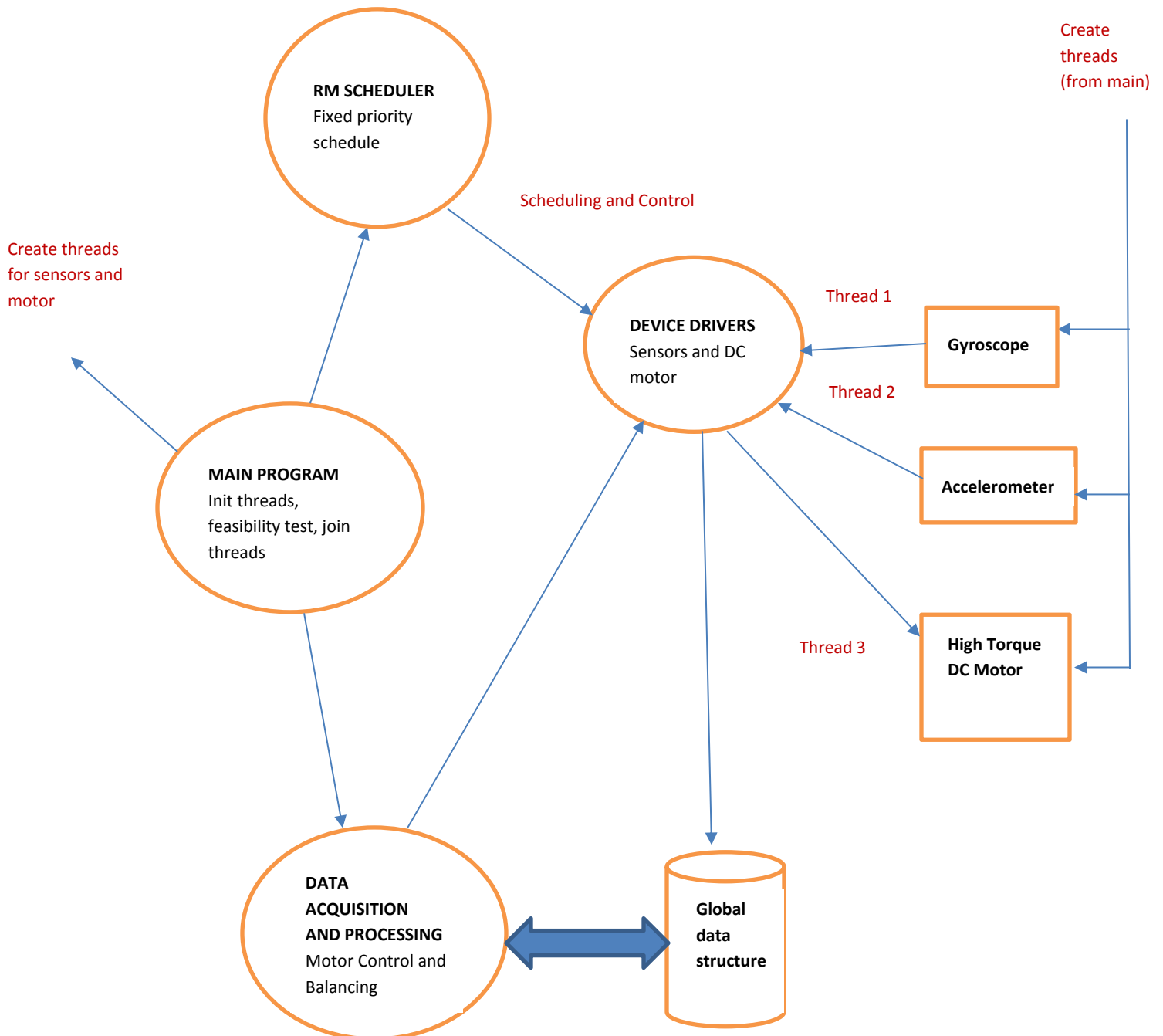
**Derivative** – expected future error

The algorithm takes this into account when it decides how much force to apply on the wheels.

The algorithm implement for this project is a simpler version of the PID control loop algorithm. It is a loop where we take a measure, and compare it with what we want. We then apply an operation then repeat.

A simple implementation of the PID controller is the complementary filter.

## 2.2 DATA FLOW DIAGRAM

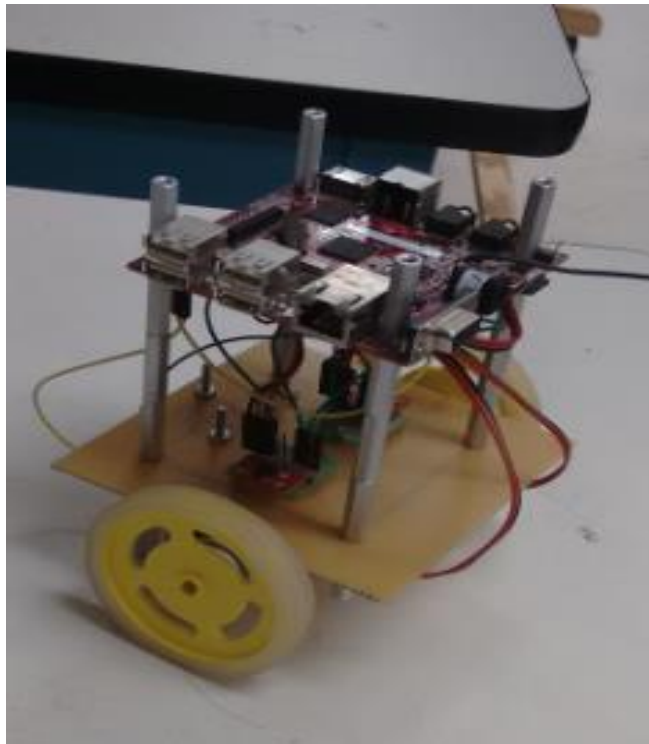


## 3.DETAILED DESIGN

### 3.1 HARDWARE AND FIRMWARE

- 1) Development Board: Beagleboard xM
- 2) I2C
- 3) Accelerometer - ADXL345
- 4) Gyroscope - ITG 3200
- 5) DC Motor driver – DRV 8830 Mini Moto breakout
- 6) DC Motor – 3V, 300mA, high torque

The hardware section is an assembly of the above mentioned components. We have mounted the sensors over a general purpose PCB.



### 3.1.1 I2C

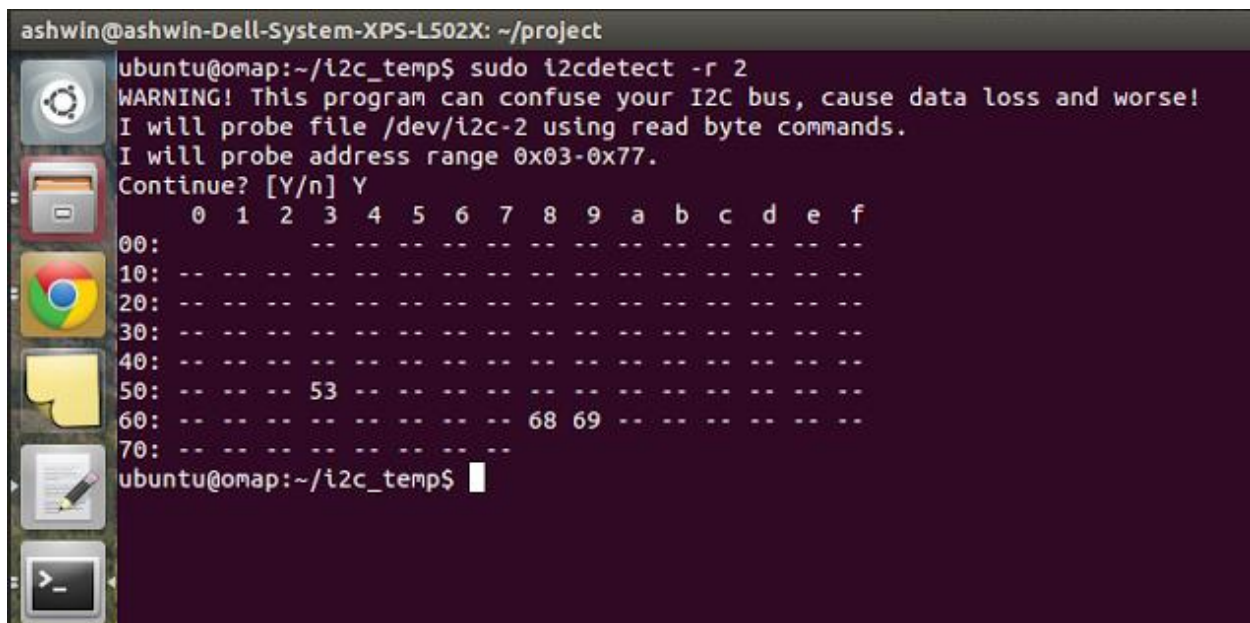
I<sup>2</sup>C (or without fancy typography, "I2C") is an acronym for the "Inter-IC" bus, a simple bus protocol which is widely used where low data rate communications suffice. Most I2C devices use seven bit addresses, and bus speeds of up to 400 kHz.

The Linux I2C programming interfaces support only the master side of bus interactions, not the slave side. The programming interface is structured around two kinds of driver, and two kinds of device. An I2C "Adapter Driver" abstracts the controller hardware; it binds to a physical device (perhaps a PCI device or platform\_device) and exposes a struct i2c\_adapter representing each I2C bus segment it manages. On each I2C bus segment will be I2C devices represented by a struct i2c\_client.

The kernel loaded onto the Beagleboard supports 3 I2C buses (i2c-1, i2c-2 and i2c-3). We have interfaced our devices (accelerometer, gyroscope and DC motor controller) to the i2c-2 bus. The expansion ports on the BeagleboardXM has dedicated SDA and SCL pins (PIN23 for SDA and PIN24 for SCL) for the i2c-2 bus. The slave devices connected to the microprocessor can be detected using the following command on the terminal

```
sudo i2cdetect -r 2
```

This will show all the slaves interfaced to the i2c bus no.2. Following is a snapshot of the command, showing the device address of the slave devices interfaced to the microcontroller



```
ashwin@ashwin-Dell-System-XPS-L502X: ~/project
ubuntu@omap:~/i2c_temp$ sudo i2cdetect -r 2
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-2 using read byte commands.
I will probe address range 0x03-0x77.
Continue? [Y/n] Y
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  53  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  68  69  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
ubuntu@omap:~/i2c_temp$
```

The System Management Bus (SMBus) is a sibling protocol. Most SMBus systems are also I2C conformant. The electrical constraints are tighter for SMBus, and it standardizes particular protocol messages and idioms. Controllers that support I2C can also support most SMBus operations, but SMBus controllers don't support all the protocol options that an I2C controller will.

I2C communication with the slave device was achieved using the i2c-tools library. Dedicated functions for I2C read and write help in communicating with the device.

The follow simple command writes the byte value 255 to the I2C device at address 20 hex on the i2c bus 2 (**/dev/i2c-0**).

```
~# i2cset -y 2 0x20 255
```

The follow simple command read a byte from an I2C device at address 20 hex on the i2c bus 2 (**/dev/i2c-2**).

```
~# i2cget -y 2 0x20
```

These commands were used extensively for debugging purposes and to establish basic communication and for checking responses with the devices.

The read and write functions can be performed using the i2c\_smbus functions.

- 1) Create and open file descriptor for the i2c bus (eg. I2C-2)

```
fd = "/dev/i2c-2"  
file = open(fd , O_RDWR);
```

- 2) Establish communication with the slave (slave address)

```
ioctl(file, I2C_SLAVE, addr) //eg : addr=slave address
```

- 3) Read and Write operations operation

```
s32 i2c_smbus_read_byte_data ( const struct i2c_client * client,  
                               u8 command);
```

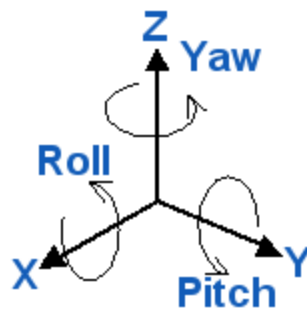
```
s32 i2c_smbus_write_byte_data (const struct i2c_client * client,  
                               u8 command,  
                               u8 value);
```



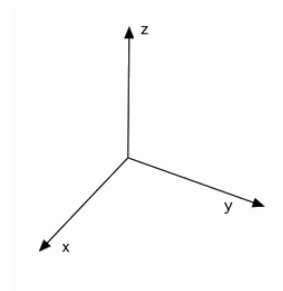
### 3.1.2 ACCELEROMETER

Accelerometer is a device that measures the acceleration in a specific direction from gravity and movement. When placed on a flat surface, it will always measure the earth's gravitational acceleration which is  $9.81\text{m/s}^2$ . This is measured in g-forces. In a state of rest we experience 1G force.

In most cases the acceleration is used as a vector quantity, which can be used to sense the orientation of the device, more precisely the pitch and roll. We can thus calculate the inclination angles of each of the axes using vector mathematics.



The robot has only 2 points of contact with the ground. Inherently, the robot should be unstable when no power is supplied. According to the orientation of the accelerometer and the gyroscopes as mounted on the robot, it is constrained to move in a straight line along the x-axis. This is achieved by maintaining equal rpm of both the motors that drive the wheels of the robot. Given its inherent instability, it can also rotate along the y axis (when it falls over). Hence the readings that we are concerned with for the PID controller are the linear acceleration ( $A_x$ ) and the angular velocity ( $\omega_y$ ). (However, for simplicity, I have positioned the gyroscope in such a way that the gyroscope sensor would measure the angular velocity about the x-axis when the robot falls over)

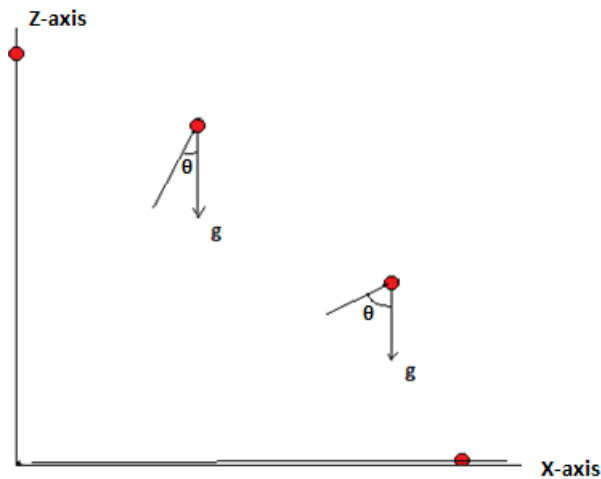


To measure the angle using an accelerometer we have to sense the gravity in each axis of the accelerometer, what it means, the projection of the gravity acceleration on each direction of

the sensor give us an idea about the angle. However, by restricting the robot's motion to linear motion along 1 axis (x-axis) and rotation about only 1 axis (x-axis), the inclination angle the Z-axis of the robot makes with the direction of gravity can simply be measured as.

$$\theta = \sin^{-1}\left(\frac{A_x}{g}\right)$$

The drawing below shows the inclination angle and how it changes as the robot falls. The inclination angle when standing perfectly upright will be 0



## ADXL345 FIRMWARE



The ADXL345 is a 3-axis accelerometer with high resolution (13-bit) measurement at up to  $\pm 16$  g. Its high resolution (4 mg/LSB) enables measurement of inclination changes less than  $1.0^\circ$ . Communication with the accelerometer sensor can be established by an I2C or SPI interface. We have used the I2C interfacing to communicate with the device

The CS pin has to be tied high to VDD I/O to enable I2C mode on the ADXL345. With the SDO/ALT ADDRESS pin high, the 7-bit I2 C address for the device is 0x1D, followed by the R/W bit. This translates to 0x3A for a write and 0x3B for a read. An alternate I2 C address of 0x53 (followed by the R/W bit) can be chosen by grounding the SDO/ALT ADDRESS pin. This

translates to 0xA6 for a write and 0xA7 for a read. *In this project the device address we configured the device address to 0x53.*

## INITIALIZATION

The initialization of ADXL345 consists of two primary things.

- 1) Enabling the measurement mode in register POWER\_CTL. This register is addressed at 0x2D in the device. To start the measurements we just need to set bit 3 in POWER\_CTL register, basically we just write 0x08 to it,
- 2) Specifying the data format in the DATA\_FORMAT register. This register is addressed at 0x31 in the device. The format of the DATA Register is as follows

**Register 0x31—DATA\_FORMAT (Read/Write)**

D7	D6	D5	D4	D3	D2	D1	D0
SELF_TEST	SPI	INT_INVERT	0	FULL_RES	Justify	Range	

In our case, we are just interested in 3 bits - D3, D1 and D0. When FULL\_RES bit is enabled, the device will run in full resolution mode (4mG/LSB). No matter what range is specified, one bit will represent 4mG of acceleration. If it is not enabled the ADXL345 will run in 10-bit mode, and the range bits will determine how many mg/LSB.

The Range bits basically sets the range of the measurements, see table below for possible configurations:

**Table 21. g Range Setting**

Setting		g Range
D1	D0	
0	0	$\pm 2 g$
0	1	$\pm 4 g$
1	0	$\pm 8 g$
1	1	$\pm 16 g$

The table below is the mg/LSB for different g-range configurations

Range	mG/LSB
All g-range when FULL_RES bit = 1	3.9
±2 g, 10 bit mode	3.9
±4 g, 10 bit mode	7.8
±8 g, 10 bit mode	15.6
±16 g, 10 bit mode	31.2

### Reading the raw acceleration and converting to Gs

The acceleration in raw format is stored in registers – DATA0, DATA1, DATA2, DATA3, DATA4, DATA5, DATA6, DATA7 (addressed at 0x32 – 0x37). The results are divided in two 8 bit registers forming 16 bit registers where the format is LSB is first then followed by MSB. Again the data are stored in two's complement format. With I2C protocol, it is possible to request multiple bytes in one reading session. The data can be continuously read from the aforementioned registers using the `i2c_smbus_read_` and `i2c_smbus_write_` commands.

After we have read raw data from the acceleration, we need to convert them to Gs. In this project, we are using 16 bit mode and full resolution, which corresponds to 3.9mG/LSB. Hence, we multiply the data with 0.0039 to convert the raw data to Gs.

$$G\_value = raw\_value\_component * 0.0039$$

Additionally, we have implemented a low pass filter. To negate the fluctuations in the readings This type of filter attenuates the higher frequencies of the signal, thus providing a smoother reading. The Low-Pass filter is easily implemented by using the following equation:

$$y_t = ax_t + (1-a) \cdot y_{t-1}$$

Where  $y_t$  is our filtered signal,  $y_{t-1}$  the previous filtered signal,  $x_t$  the accelerometer reading and 'a' the smoothing factor. Substituting `G_value` and `raw_value_component` in the filter equation we would get

$$G\_value = Gvalue*a + (1-a) \cdot raw\_value\_component$$

Using the sin inverse function we can then find the inclination angle or 'tilt' of the Z-axis from the gravitational acceleration vector using.

$$Angle = \sin^{-1}(G\_value)$$

Please refer to the appendix for the code for the accelerometer.

## ACCELEROMETER FIRMWARE CODE

```
/* ADXL345 Accelerometer functions */
void ADXL345_setup(int file)
{
    i2c_smbus_write_byte_data(file, DATA_FORMAT_REG, 0x0B);

    i2c_smbus_write_byte_data(file, POWER_CTRL_REG, 0x08);
}

void* ADXL345_i2c(void *threadid)
//*****
{
    int file,i;
    int adapter_no = 2;
    char filename[20];
    double stop = 0.0;

    int addr = 0x53; /* The I2C address */

    char reg; /*= 0x10; Device register to access */
    char res;
    char buf[10];

    int alpha = 0.5;

    snprintf(filename, 19, "/dev/i2c-%d", adapter_no);
    file = open(filename , O_RDWR);

    if (file < 0)
    {
        perror("NO FILE DESCRIPTOR \n");
        exit(1);
    }

    if (ioctl(file, I2C_SLAVE, addr) < 0)
    {
        perror("I2C COMMUNICATION ESTABLISHMENT FAILED");
        exit(1);
    }

    ADXL345_setup(file);

    /* WRITE TO FIFO_CTL REGISTER */
    i2c_smbus_write_word_data(file, FIFO_CTRL_REG, 0x83);

    while(1)
    //for(i=0;i<10;i++)
    {
        /* reading accelerometer data */
        x_a_val = i2c_smbus_read_byte_data(file, X_CO_H)<<8;
        x_a_val |= i2c_smbus_read_byte_data(file, X_CO_L);

        y_a_val = i2c_smbus_read_byte_data(file, Y_CO_H)<<8;
```

```

y_a_val |= i2c_smbus_read_byte_data(file, Y_CO_L);

z_a_val = i2c_smbus_read_byte_data(file, Z_CO_H)<<8;
z_a_val |= i2c_smbus_read_byte_data(file, Z_CO_L);
/*-----*/

if ((x_a_val < 0)|| (y_a_val<0)|| (z_a_val<0))
/* ERROR HANDLING: i2c transaction failed */
    printf("\n I2C TRANSACTION FAILED ");
else
    /* res contains the read word */
    {
        //get G-value from raw data
        x_acc = x_a_val * 0.0039;
        y_acc = y_a_val * 0.0039;
        z_acc = z_a_val * 0.0039;

        // low pass filter
        x_acc = x_a_val * alpha + (x_acc * (1.0 - alpha));
        y_acc = y_a_val * alpha + (y_acc * (1.0 - alpha));
        z_acc = z_a_val * alpha + (z_acc * (1.0 - alpha));

        //calculating pitch or angle
        if(x_acc>=0 && x_acc<=1)
            pitch = (asin(x_acc)*180.0)/PI;
        else
            pitch = -1*(asin(256.0-x_acc)*180.0)/PI;

        //printf("ACCELEROMETER : X co-ordinates : %lf\t Y co-
ordinates : %lf\t Z coordinates : %lf\t ANGLE :
%lf\n\n",x_acc,y_acc,z_acc,pitch);

    }
}
close(file);
}

```

### 3.1.3 GYROSCOPE

Gyroscopes are devices that measure rotational motion. MEMS (microelectromechanical system) gyros are sensors that measure angular velocity. The units of angular velocity are measured in degrees per second ( $^{\circ}/s$ ) or revolutions per second (RPS). Angular velocity is simply a measurement of speed of rotation. A triple axis MEMS gyroscope, can measure rotation around three axes: x, y, and z.

Because the gyroscope gives us the angular rate, we can calculate the angle using the formula as given below

$$\text{angle} = \text{initial\_angle} + \omega * dt$$

*where  $\omega$  is the measured angular velocity.*

A gyro measures the rate of rotation, which has to be tracked over time to calculate the current angle. To calculate the angle covered by the rotating object, we have to integrate its angular velocity over a period of time.

#### ITG-3200 GYROSCOPE FIRMARE



The ITG3200 communicates thorough an I2C interface. The device address for this gyroscope in the default shipped state is 0x69.

The gyroscope data values are stored in registers addressed from 0x1D- 0x22. XOUT\_MSB in 0x1D, XOUT\_LSB in 0x1E, YOUT\_MSB in 0x1F and so on. The communication and the order of data retrieval from the gyroscope is very similar to the accelerometer (ADXL345).

1. We initialized the sample rate to 100Hz and a maximum range of detection to +/- 2000 degrees per second by writing values 0x18 to the DLPF\_FS (address 0x16) register and 0x09 to the SMPLRT\_DIV (address 0x15) register.
2. We then start reading from the register that hold the value of the angular velocity. This is raw gyroscope data, and it has not been converted to degrees per second yet. Bigger numbers mean the device is rotating faster. Positive numbers indicate one direction of rotation while negative numbers indicate the opposite rotation direction. Since this is a triple-axis gyroscope, we can measure the rotational rate of the board no matter which way

the board is rotating. Rotation is usually measured in degrees per second. If the board spins around an axis exactly one time in a second, the gyroscope would measure 360 degrees per second.

3. We then apply the formula as described above to calculate the degree of rotation.

## GYROSCOPE FIRMWARE CODE

```
void ITG3200_setup(int file) //initialization
{
    char id;
    id = i2c_smbus_read_byte_data(file, 0x00); //read from WHO_AM_I Register;

    printf("WHO AM I : %d\n\n",id);

    //Set the gyroscope scale for the outputs to +/-2000 degrees per second
    i2c_smbus_write_byte_data(file, DLPF_FS, 0x18);

    //Set the sample rate to 100 hz
    i2c_smbus_write_byte_data(file, SMPLRT_DIV, 0x09);
}

void ITG3200_i2c() //GYROSCOPE
{
    int file,i;
    int adapter_no = 2;
    char filename[20];

    int addr = 0x69; /* The I2C address */

    snprintf(filename, 19, "/dev/i2c-%d", adapter_no);
    file = open(filename , O_RDWR);

    if (file < 0)
    {
        perror("NO FILE DESCRIPTOR \n");
        exit(1);
    }

    if (ioctl(file, I2C_SLAVE, addr) < 0) //establishing device communication
    {
        perror("I2C COMMUNICATION ESTABLISHMENT FAILED");
        exit(1);
    }

    ITG3200_setup(file);

    while(1)
    //for(i=0;i<10;i++)
    {
        x_gyro = i2c_smbus_read_byte_data(file, ITG_X_CO_H)<<8;
        x_gyro |= i2c_smbus_read_byte_data(file, ITG_X_CO_L);

        y_gyro = i2c_smbus_read_byte_data(file, ITG_Y_CO_H)<<8;
        y_gyro |= i2c_smbus_read_byte_data(file, ITG_Y_CO_L);

        z_gyro = i2c_smbus_read_byte_data(file, ITG_Z_CO_H)<<8;
        z_gyro |= i2c_smbus_read_byte_data(file, ITG_Z_CO_L);

        if ((x_gyro < 0)||(y_gyro<0)||(z_gyro<0))/*ERROR HANDLING:i2c transaction failed*/
            printf("\n I2C TRANSACTION FAILED ");
        else
            /* res contains the read word */
            printf("GYROSCOPE : X co-ordinates : %d\t Y co-ordinates : %d\t Z
coordinates : %d\n",x_gyro,y_gyro,z_gyro);
    }
    close(file);}
```



### 3.1.4 DC MOTOR AND CONTROLLER

#### MOTOR

We interfaced 2 DC motors to control the wheels of the robot. A 1.8-3.3V/600mA low torque motor was initially used as a part of this project. However, on testing with this motor, we found that it did not provide enough torque to rotate the wheels of the robot. This motor was replaced with a 3V/300mA high torque dc motor. It is a 300 rpm motor but provides a high enough torque to carry the weight of the robot.

#### CONTROLLER (DRV 8830 Mini-Moto break out board)



DRV 8830 motor control breakout board is used to control the dc motor. It has an I2C interface to the target board. The processor can communicate with the motor controller using the `i2c_smbus_read_` and `i2c_smbus_write_` commands. We can control the rpm and operate the dc motor either in forward or reverse directions by applying the correct voltages levels to the inputs of the DRV 8830.

#### MOTOR CONTROLLER FIRMWARE

The MiniMoto board has two jumpers for setting its I<sup>2</sup>C address. The jumpers can be in one of three states: open, 1, or 0. We used the controller in its default state in which the A0 and A1 jumpers were open (1). The corresponding device address for the default state is 0x68.

- 1) We first establish communication with the device address and receive acknowledgement from the slave device. This is done by using linux file descriptors as explained in the I2C section
- 2) The DRV8830 chip on the driver circuit has only two registers addressed at 0x00 and 0x01. We can control the PWM and direction of the motor by writing an 8 bit value to the 0x00 register. The magnitude needed to vary the PWM varies from 6 to 63. This is a 6-bit argument to which two more bits are appended corresponding to the direction of rotation i.e. D8-D2 bits of the 0x00 will contain values from 6-63 corresponding to desired PWM output and the last 2 bits will correspond to the direction of the rotation (D1 D0 can hold either 01 or 10 i.e. clockwise or anticlockwise).  
*Eg. The output voltage to the motor from the breakout board is Vcc for 63 PWM (maximum) with 01 in last two bits and -Vcc for 63 PWM and 10 in the last two bits of the 0x00 register.*
- 3) The motor can be stopped by writing 0x00 in 0x00 register .

## 3.2 SOFTWARE

### 3.2.1. CODE INTEGRATION

On obtaining angle values from both sensors we have to merge the values to get a final correct angle measurement. The gyroscope has a drift and in a few time the values returned are completely wrong. The accelerometer returns a true values of angles within a certain range of up to +/-45 degrees about the vertical z axis but due to high noise we may suffer from marginal errors

Usually a Kalman filter is used to mix and merge the two values, in order to have a correct value. However this algorithm is complex computationally intensive and it takes a lot of time to process a single data acquisition from the 2 sensors. Another approach to merge the readings is to apply the Complementary filter. The equation for the complementary filter is easier as compared to Kalman filter. It is given as

$$\text{Current angle} = 98\% \times (\text{current angle} + \text{gyro rotation rate}) + (2\% \times \text{Accelerometer angle})$$

However, the algorithm depends a lot on the accuracy of the gyro rotation rate as it attributes 98% to it. Since we are not integrating the angular velocity readings but rather summing it over discrete intervals there is a chance of errors creeping in.

Given the limitation on the degrees of freedom in our project we took average of both sensor reading. Within a small deviation the readings from either sensors did not deviate much and hence it was sufficient for the project. This approach may help in keeping the robot steady in the absence of any external force, but it will fail if an external force is applied as the reading from either sensor will vary a lot causing the PID control algorithm to take incorrect 'corrective' measures.

### 3.2.2 REAL TIME SCHEDULER

The rate monotonic (RM) scheduling policy has been used to schedule tasks for the self-balancing robot.

*The system runs three distinct services:*

- 1) Accelerometer service
- 2) Gyroscope service
- 3) Motor control service

*These services run periodically at the below intervals:*

T1 = 5ms (accelerometer)

T2 = 5ms (gyroscope)

T3 = 10ms (motor control)

*The execution time for these services has been calculated as:*

C1 = 2ms (accelerometer)

C2 = 2ms (gyroscope)

C3 = 5ms (motor control)

As per the rate monotonic policy, we assign priorities in the order of decreasing frequency:

Highest priority service: accelerometer

Middle priority service: gyroscope

Lowest priority service: motor control

The scheduler spawns three threads, one for each service. We use semaphores to synchronize these threads according to the schedule. The period of the services is decided based on the execution time and the number of samples. These are needed to determine the direction of tilt and corresponding action to be taken by the motor control service. The motor control service handles tilt in the forward and reverse directions. The accelerometer service takes readings for linear acceleration in the direction of x, y, z axes. The gyroscope takes readings for angular velocity in directions of x, y, z axes.

A steady state balanced position for the robot has been determined. The tilt is measured and corrective acceleration is provided by the motor to get the self-balancing robot to this steady state position.

The scheduler uses Native POSIX Thread Library (NPTL) from the Linux kernel to run services on the system. These NPTL threads are used with the FIFO scheduling policy. Processor affinity has been set to use one particular core.

The schedule has been implemented using semaphores. The semaphores are initialized to the locked state by default. The threads are made to wait in a `while(1)` loop servicing the accelerometer, gyroscope or motor driver. Semaphores are released one at a time in the main thread by doing a `sem_post`. After each `sem_post`, a pre-determined delay is introduced. After this delay, the semaphore is locked in the thread and subsequently the next semaphore is released. This goes on in cyclic fashion for each of the services. The accelerometer and gyroscope run for 5 ms and the dc motor driver runs for 10 ms.

### **SCHEDULER ANALYSIS:**

The scheduler operates in the below sequence:

- 1) Initialize semaphores in the locked state.
- 2) Create threads for each service. Execute all tasks for the first time and wait on the `sem_wait()` at the end of the task.
- 3) Release a semaphore for task 1 using `sem_post()`.

- 4) Execute task 1 for a duration of 5ms. Print the time elapsed for task 1.
- 5) Wait on semaphore 1. Release task 2. Execute task 2 for 5ms. Print the time elapsed for task2.
- 6) Wait on semaphore 2. Release task 3. Execute task 3 for 10ms. Print the time elapsed for task3.
- 7) Wait on semaphore 3. Repeat the same process for tasks 1, 2, 3.

On observing the below scheduler output, we can see that the accelerometer service which is the highest priority service has been dispatched first. Following this service, the gyroscope service is dispatched after an interval of 5ms. The lowest priority service which is the motor control service is dispatched after another 5ms and runs for 10ms. This can be observed from the below screenshots of the three services running in the self-balancing robot system.

```
Accelerometer service
Time Elapsed: 0.005585

Gyroscope service
Time Elapsed: 0.010803

Motor control service
Time Elapsed: 0.021149

Accelerometer service
Time Elapsed: 0.026398

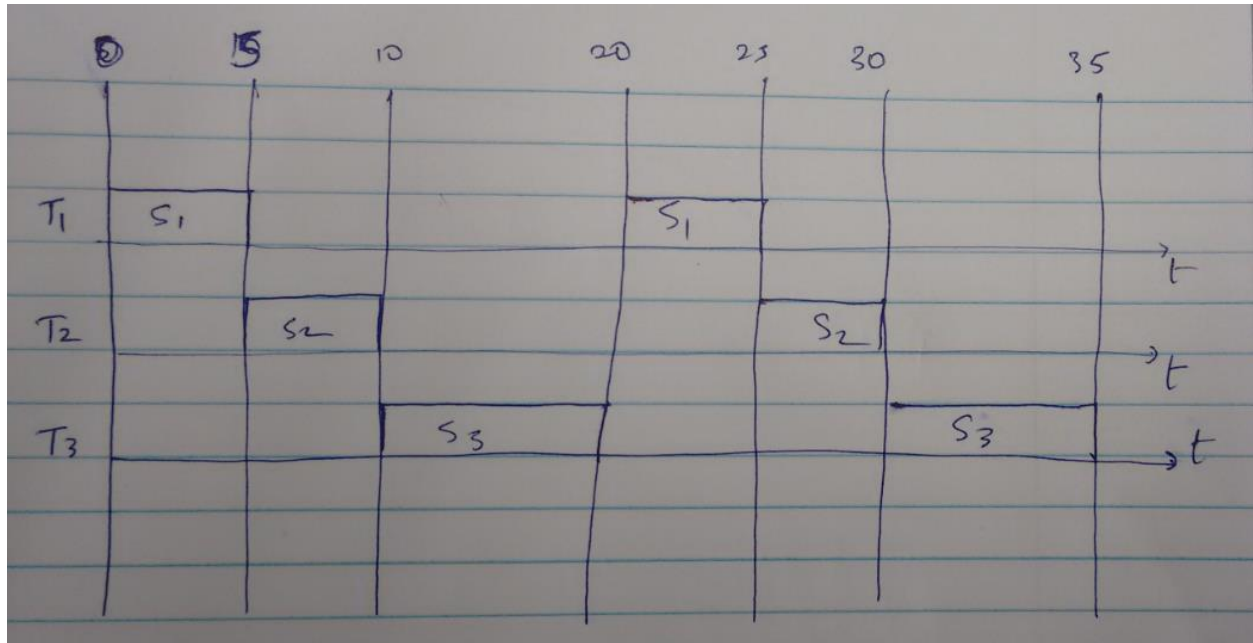
Gyroscope service
Time Elapsed: 0.031708

Motor control service
Time Elapsed: 0.041992

Accelerometer service
Time Elapsed: 0.047241

Gyroscope service
Time Elapsed: 0.052582
```

### TIMING DIAGRAM FOR SCHEDULER:



Each of the tasks waits for the previous task to execute and begins execution as soon as the time allotted to that task is finished. This schedule is reproduced indefinitely till the program is running.

### DEADLINES:

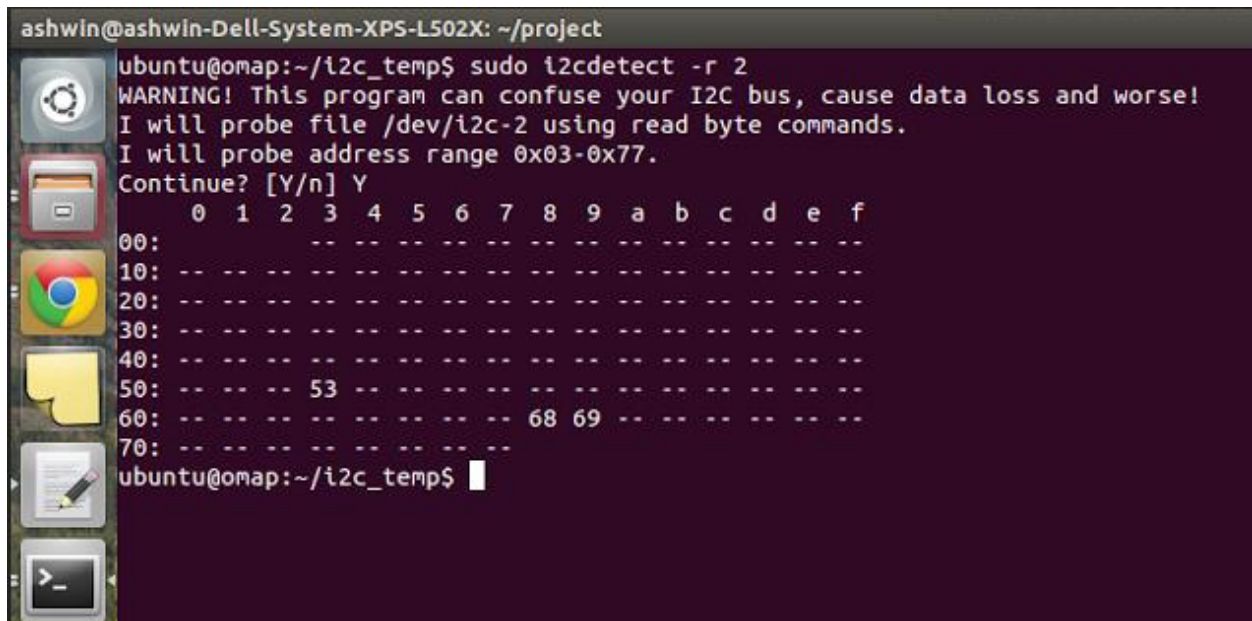
Since we are following the rate monotonic policy, the deadline here equals the period of one particular task. We have ensured that we provide sufficient time for the accelerometer and the gyroscope services to acquire data and for the motor to take corrective actions. However, in case the accelerometer or the gyroscope takes more time to execute than the allotted time frame, the lower priority services will suffer. For example, if accelerometer or the gyroscope services miss their deadlines, this would severely impact on the motor control service. It will not have sufficient time to take corrective action to balance the robot and it may even topple over. Since we have three services only in our system, we have been able to provide sufficient time to the accelerometer and gyroscope services without compromising on the motor service time.

## 4. TESTING PLANS AND RESULTS

In this section, we present our testing methods for the self-balancing robot and the results obtained.

### Detecting I2C devices:

We tested the detection of I2C devices using the i2c-tools utility installed on the Beagle xM. It provides us with commands like i2cdetect which help us detect the peripherals connected to the board. Below is a screenshot of the i2cdetect command. The addresses 0x53, 0x68 and 0x69 are the device addresses of accelerometer, dc motor and gyroscope respectively.

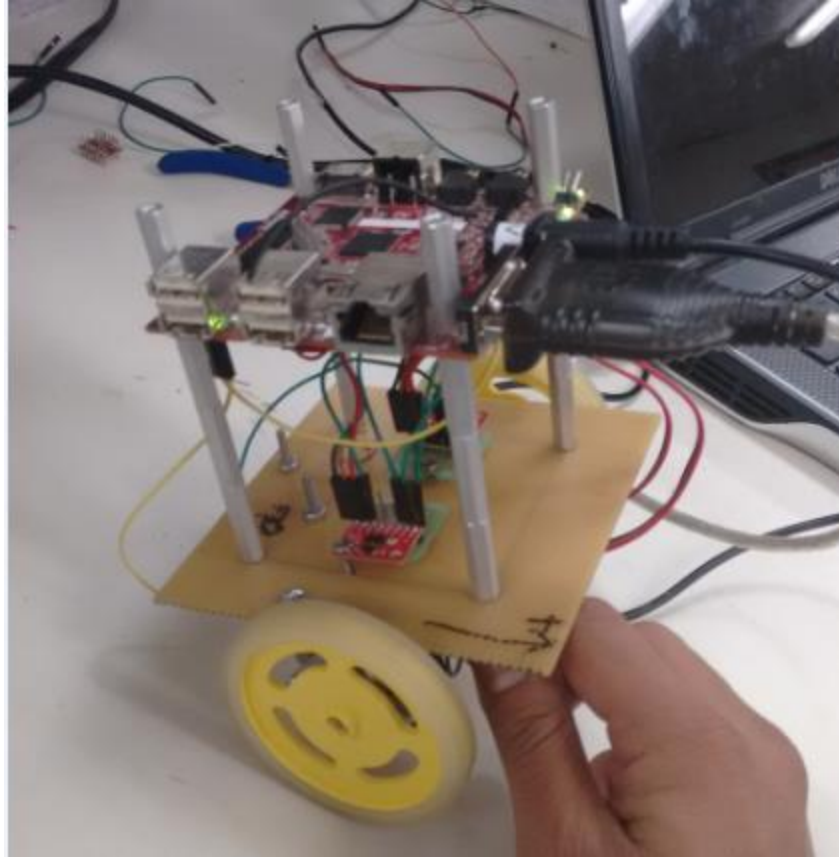


```
ashwin@ashwin-Dell-System-XPS-L502X: ~/project
ubuntu@omap:~/i2c_temp$ sudo i2cdetect -r 2
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-2 using read byte commands.
I will probe address range 0x03-0x77.
Continue? [Y/n] Y
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  -- -- -- -- -- -- -- -- -- -- -- -- -- --
10:  -- -- -- -- -- -- -- -- -- -- -- -- -- --
20:  -- -- -- -- -- -- -- -- -- -- -- -- -- --
30:  -- -- -- -- -- -- -- -- -- -- -- -- -- --
40:  -- -- -- -- -- -- -- -- -- -- -- -- -- --
50:  -- -- -- 53 -- -- -- -- -- -- -- -- -- --
60:  -- -- -- -- -- -- -- 68 69 -- -- -- -- --
70:  -- -- -- -- -- -- -- -- -- -- -- -- -- --
ubuntu@omap:~/i2c_temp$
```

### 1. Forward tilt test and results:

The self-balancing robot has been designed to balance in case of a forward or backward tilt. The readings from accelerometer and gyroscope were observed for a steady state condition. Later on, a forward tilt condition was generated and the readings for this condition were noted. A combination of the accelerometer and the gyroscope readings were applied to a low pass filter to generate angular readings. The robot was calibrated using these readings. The direction of rotation of wheels was decided with respect to the angle of tilt.





**Forward tilt**

ashwin@ashwin-Dell-System-XPS-L502X: ~/project				
ACCELEROMETER : X co-ordinates : 0.230469	Y co-ordinates : 0.042969	Z coordinates : 0.839844	ANGLE : 13.331429	
ACCELEROMETER : X co-ordinates : 0.230469	Y co-ordinates : 0.042969	Z coordinates : 0.839844	ANGLE : 13.331429	
ACCELEROMETER : X co-ordinates : 0.242188	Y co-ordinates : 255.992188	Z coordinates : 0.800781	ANGLE : 14.022793	
ACCELEROMETER : X co-ordinates : 0.242188	Y co-ordinates : 255.992188	Z coordinates : 0.800781	ANGLE : 14.022793	
ACCELEROMETER : X co-ordinates : 0.242188	Y co-ordinates : 255.992188	Z coordinates : 0.800781	ANGLE : 14.022793	
ACCELEROMETER : X co-ordinates : 0.242188	Y co-ordinates : 255.000000	Z coordinates : 0.843750	ANGLE : 14.022793	
ACCELEROMETER : X co-ordinates : 0.250000	Y co-ordinates : 0.000000	Z coordinates : 0.843750	ANGLE : 14.484855	
ACCELEROMETER : X co-ordinates : 0.250000	Y co-ordinates : 0.000000	Z coordinates : 0.843750	ANGLE : 14.484855	
ACCELEROMETER : X co-ordinates : 0.250000	Y co-ordinates : 0.000000	Z coordinates : 0.843750	ANGLE : 14.484855	
ACCELEROMETER : X co-ordinates : 0.218750	Y co-ordinates : 0.011719	Z coordinates : 0.816406	ANGLE : 12.642034	
ACCELEROMETER : X co-ordinates : 0.218750	Y co-ordinates : 0.011719	Z coordinates : 0.816406	ANGLE : 12.642034	
ACCELEROMETER : X co-ordinates : 0.218750	Y co-ordinates : 0.011719	Z coordinates : 0.816406	ANGLE : 12.642034	
ACCELEROMETER : X co-ordinates : 0.218750	Y co-ordinates : 0.011719	Z coordinates : 0.816406	ANGLE : 12.642034	
ACCELEROMETER : X co-ordinates : 0.207031	Y co-ordinates : 0.000000	Z coordinates : 0.804688	ANGLE : 11.954493	
ACCELEROMETER : X co-ordinates : 0.207031	Y co-ordinates : 0.000000	Z coordinates : 0.804688	ANGLE : 11.954493	
ACCELEROMETER : X co-ordinates : 0.207031	Y co-ordinates : 0.000000	Z coordinates : 0.804688	ANGLE : 11.954493	
ACCELEROMETER : X co-ordinates : 0.207031	Y co-ordinates : 0.000000	Z coordinates : 0.785156	ANGLE : 11.954493	
ACCELEROMETER : X co-ordinates : 0.203125	Y co-ordinates : 0.027344	Z coordinates : 0.785156	ANGLE : 11.725705	
ACCELEROMETER : X co-ordinates : 0.203125	Y co-ordinates : 0.027344	Z coordinates : 0.785156	ANGLE : 11.725705	

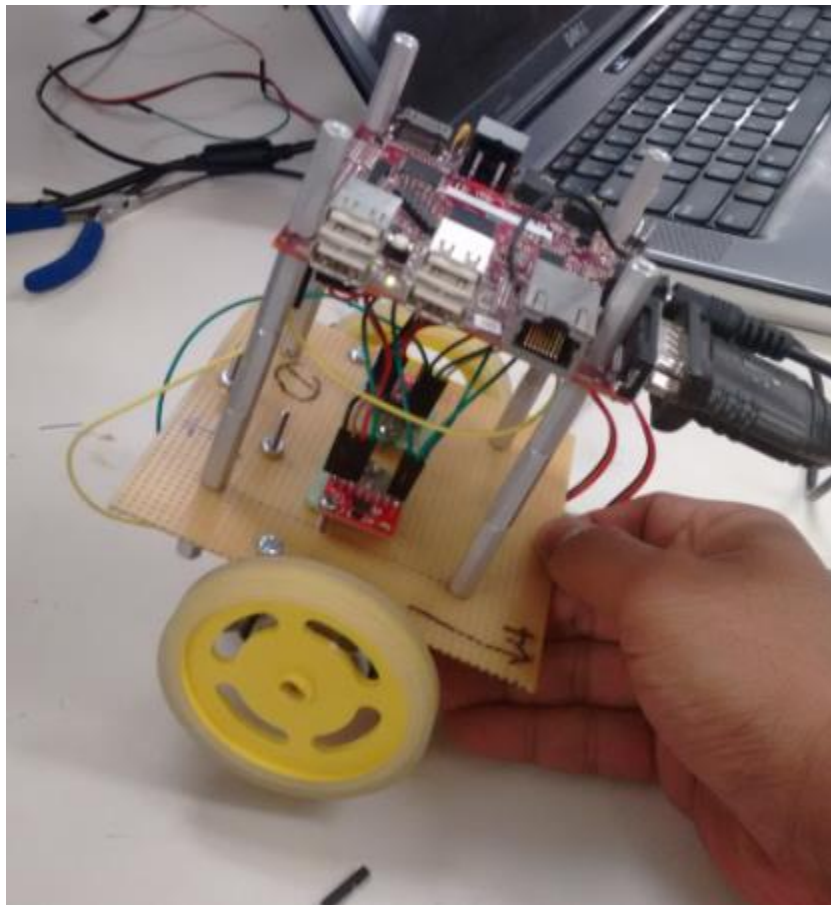
**Forward tilt angular readings**



As we can see in the results, the angle varies in the positive range. The robot has been programmed to accelerate in the opposite direction to that of the tilt.

## **2. Backward tilt test and results:**

The backward tilt condition was generated and tested. The readings from accelerometer and gyroscope were translated to angular readings. The backward tilt requires the motion of wheels to be in the direction opposite of the tilt. This condition was calibrated and tested programmatically.



**Backward tilt**

```
ashwin@ashwin-Dell-System-XPS-L502X: ~/project
ACCELEROMETER : X co-ordinates : 255.824219 Y co-ordinates : 255.937500 Z coordinates : 0.867188 ANGLE : -10.129260
ACCELEROMETER : X co-ordinates : 255.824219 Y co-ordinates : 255.964844 Z coordinates : 0.859375 ANGLE : -10.129260
ACCELEROMETER : X co-ordinates : 255.777344 Y co-ordinates : 255.964844 Z coordinates : 0.859375 ANGLE : -12.871621
ACCELEROMETER : X co-ordinates : 255.777344 Y co-ordinates : 255.964844 Z coordinates : 0.859375 ANGLE : -12.871621
ACCELEROMETER : X co-ordinates : 255.777344 Y co-ordinates : 255.964844 Z coordinates : 0.859375 ANGLE : -12.871621
ACCELEROMETER : X co-ordinates : 255.812500 Y co-ordinates : 255.964844 Z coordinates : 0.855469 ANGLE : -10.812404
ACCELEROMETER : X co-ordinates : 255.812500 Y co-ordinates : 255.964844 Z coordinates : 0.855469 ANGLE : -10.812404
ACCELEROMETER : X co-ordinates : 255.812500 Y co-ordinates : 255.964844 Z coordinates : 0.855469 ANGLE : -10.812404
ACCELEROMETER : X co-ordinates : 255.812500 Y co-ordinates : 255.964844 Z coordinates : 0.855469 ANGLE : -10.812404
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.976562 Z coordinates : 0.800781 ANGLE : -11.725705
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.976562 Z coordinates : 0.800781 ANGLE : -11.725705
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.976562 Z coordinates : 0.800781 ANGLE : -11.725705
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.976562 Z coordinates : 0.800781 ANGLE : -11.725705
ACCELEROMETER : X co-ordinates : 255.808594 Y co-ordinates : 255.949219 Z coordinates : 0.835938 ANGLE : -11.040460
ACCELEROMETER : X co-ordinates : 255.808594 Y co-ordinates : 255.949219 Z coordinates : 0.835938 ANGLE : -11.040460
ACCELEROMETER : X co-ordinates : 255.808594 Y co-ordinates : 255.949219 Z coordinates : 0.835938 ANGLE : -11.040460
ACCELEROMETER : X co-ordinates : 255.808594 Y co-ordinates : 255.960938 Z coordinates : 0.839844 ANGLE : -11.040460
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.960938 Z coordinates : 0.839844 ANGLE : -11.725705
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.960938 Z coordinates : 0.839844 ANGLE : -11.725705
ACCELEROMETER : X co-ordinates : 255.796875 Y co-ordinates : 255.960938 Z coordinates : 0.839844 ANGLE : -11.725705
```

### Backward tilt angular readings

As we can observe, the backward tilt provides a negative angle. This can be used to determine the direction of the tilt and accelerate in the opposite direction.

#### 4.1.1 FINAL RESULTS

We have been able to successfully balance a two wheel robot within a  $\pm 15$ -20 degrees range of angles. The robot is able to balance itself in the absence of external forces however fails to maintain its balance when an external force is applied. This can be attributed to two causes

- The motor used was not able to generate enough torque to apply corrective acceleration for angular range above 20 degrees of tilt. For this reason, we had to use some support at the bottom to curb the fall and the acceleration of the wheel took care to get the robot back to its steady state position.
- As explained earlier, the algorithm is a simpler version of the complement filter algorithm. We are still working on making the algorithm more robust and are trying to implement the Kalman filter for a PID control algorithm.

## 5. CONCLUSIONS

As stated earlier, our objective has been to balance the two wheel robot with the use of sensors and dc motors. We aimed to achieve this task using a real-time scheduling theory. From the tests conducted and the results obtained, we have been able to achieve our main objective of balancing the robot on two wheels using real time services. The tilt of the robot has been accurately calculated using sensors. The output data from sensors has been successfully analyzed to get an accurate angle of tilt. The robot balances up to a tilt angle of about  $\pm 20$  degrees. For tilt angles greater than these, the torque generated by the motor is not enough to accelerate the robot in the opposite direction and balance it.

We have been able to gain good insight in writing Linux device drivers and application of the real-time theory to develop a scheduler. It was a learning experience where we had to apply our physics concepts and design mechanical modules to function efficiently with the electrical inputs given.

This project can be expanded by adding a USB camera to the Beagle xM which can further help in balancing the robot and it be used as a tool to set the robot in motion by recognizing signs and symbols.

## 6. REFERENCES

1. <http://publications.lib.chalmers.se/records/fulltext/163397.pdf>
2. <http://reibot.org/2012/01/04/self-balancing-robot/>
3. <http://www.instructables.com/id/Angle-measurement-using-gyro-accelerometer-and-Ar/step3/Some-extra-information-before-we-proceed/>
4. <http://ozzmaker.com/2013/04/18/success-with-a-balancing-robot-using-a-raspberry-pi/>
5. **GPIO Pin MUX :** <http://technologyrealm.blogspot.com/2014/02/beagleboard-xm-pin-muxing-and-gpio.html>

### ACCELEROMETER

1. <http://thecontinuum.com/tag/adxl345/>
2. <http://morf.lv/modules.php?name=tutorials&lasit=31>
3. <http://www.instructables.com/id/Angle-measurement-using-gyro-accelerometer-and-Ar/step3/Some-extra-information-before-we-proceed/>
4. **Datasheet:** <http://www.analog.com/en/products/mems/mems-accelerometers/adxl345.html>

### GYROSCOPE

1. [https://learn.sparkfun.com/tutorials/itg-3200-hookup-guide?\\_ga=1.35046369.1346084763.1438947795](https://learn.sparkfun.com/tutorials/itg-3200-hookup-guide?_ga=1.35046369.1346084763.1438947795)
2. **Datasheet :** <https://www.sparkfun.com/datasheets/Sensors/Gyro/PS-ITG-3200-00-01.4.pdf>
3. <http://thecontinuum.com/tag/adxl345/>

### MOTOR CONTROLLER

1. **Datasheet:**<http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=drv8830&fileType=pdf>
2. [https://learn.sparkfun.com/tutorials/minimoto-drv8830-hookup-guide?\\_ga=1.101164609.1346084763.1438947795](https://learn.sparkfun.com/tutorials/minimoto-drv8830-hookup-guide?_ga=1.101164609.1346084763.1438947795)
3. [https://en.wikipedia.org/wiki/DC\\_motor](https://en.wikipedia.org/wiki/DC_motor)
4. <http://www.electro-tech-online.com/threads/connecting-two-dc-motor-in-series-or-in-parallel.107153/>

### SCHEDULER

1. Nisheeth Bhat's Independent study

# APPENDIX

## C CODE IMPLEMENTING SELF BALANCING ROBOT

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <sched.h>
#include <unistd.h>
#include <time.h>
#include <syslog.h>
#include <math.h>
#include <sys/param.h>
#include "i2c-dev.h"
#include <fcntl.h>
#include <sys/stat.h>

/* ADXL345 Registers */
#define DATA_FORMAT_REG 0x31
#define POWER_CTRL_REG 0x2D
#define FIFO_CTRL_REG 0x38
#define PI 3.14

#define X_CO_L 0x32
#define X_CO_H 0x33

#define Y_CO_L 0x34
#define Y_CO_H 0x35

#define Z_CO_L 0x36
#define Z_CO_H 0x37

/* ITG3200 Registers */
#define DLPF_FS 0x16
#define SMPLRT_DIV 0x15

#define ITG_X_CO_H 0x1D
#define ITG_X_CO_L 0x1E

#define ITG_Y_CO_H 0x1F
#define ITG_Y_CO_L 0x20

#define ITG_Z_CO_H 0x21
#define ITG_Z_CO_L 0x22

int x_a_val, y_a_val, z_a_val;
double x_acc, y_acc, z_acc;

int x_gyro, y_gyro, z_gyro;

pthread_t task1, task2, task3;
pthread_attr_t task1_sched_attr;
pthread_attr_t task2_sched_attr;
pthread_attr_t task3_sched_attr;
pthread_attr_t main_sched_attr;
int rt_max_prio, rt_min_prio;
struct sched_param task1_param;
struct sched_param task2_param;
struct sched_param task3_param;
struct sched_param main_param;

double start = 0;
sem_t sem_t1, sem_t2, sem_t3;
int flag = 1;
int abort_task1, abort_task2, abort_task3;
double pitch;
```

```

double readTOD(void)
{
    struct timeval tv;
    double ft=0.0;
    if( gettimeofday (& tv, NULL) != 0)
    {
        perror("readTOD");
        return 0.0;
    }
    else
    {
        ft = ((double)(((double)tv.tv_sec) + (((double)tv.tv_usec) /
1000000.0))));
    }
    return ft;
}

/* ADXL345 Accelerometer functions */
void ADXL345_setup(int file)
{
    i2c_smbus_write_byte_data(file, DATA_FORMAT_REG, 0x0B);

    i2c_smbus_write_byte_data(file, POWER_CTRL_REG, 0x08);
}

void* ADXL345_i2c(void *threadid) /**/
{
    int file,i;
    int adapter_no = 2;
    char filename[20];
    double stop = 0.0;

    int addr = 0x53; /* The I2C address */

    char reg; /*= 0x10; Device register to access */
    char res;
    char buf[10];

    int alpha = 0.5;

    snprintf(filename, 19, "/dev/i2c-%d", adapter_no);
    file = open(filename , O_RDWR);

    if (file < 0)
    {
        perror("NO FILE DESCRIPTOR \n");
        exit(1);
    }

    if (ioctl(file, I2C_SLAVE, addr) < 0)
    {
        perror("I2C COMMUNICATION ESTABLISHMENT FAILED");
        exit(1);
    }

    ADXL345_setup(file);

    /* WRITE TO FIFO_CTL REGISTER */
    i2c_smbus_write_word_data(file, FIFO_CTRL_REG, 0x83);

    while(1)
    //for(i=0;i<10;i++)
    {
        /* reading accelerometer data */
        x_a_val = i2c_smbus_read_byte_data(file, X_CO_H)<<8;
        x_a_val |= i2c_smbus_read_byte_data(file, X_CO_L);

        y_a_val = i2c_smbus_read_byte_data(file, Y_CO_H)<<8;
        y_a_val |= i2c_smbus_read_byte_data(file, Y_CO_L);

        z_a_val = i2c_smbus_read_byte_data(file, Z_CO_H)<<8;

```

```

        z_a_val |= i2c_smbus_read_byte_data(file, Z_CO_L);
        /*-----*/

        if ((x_a_val < 0) || (y_a_val < 0) || (z_a_val < 0)) /* ERROR
HANDLING: i2c transaction failed */
            printf("\n I2C TRANSACTION FAILED ");
        else /* res contains the read word */
        {
            x_acc = x_a_val * 0.0039;
            y_acc = y_a_val * 0.0039;
            z_acc = z_a_val * 0.0039;

            x_acc = x_a_val * alpha + (x_acc * (1.0 - alpha));
            y_acc = y_a_val * alpha + (y_acc * (1.0 - alpha));
            z_acc = z_a_val * alpha + (z_acc * (1.0 - alpha));

            if(x_acc >= 0 && x_acc <= 1)
                pitch = (asin(x_acc)*180.0)/PI;
            else
                pitch = -1*(asin(256.0-x_acc)*180.0)/PI;

            //printf("ACCELEROMETER : X co-ordinates : %lf\t Y co-ordinates : %lf\t Z
coordinates : %lf\t ANGLE : %lf\n\n",x_acc,y_acc,z_acc,pitch);

            if(abort_task1 == 1)
            {
                printf("ACCELEROMETER : X co-ordinates : %lf\t Y co-ordinates : %lf\t Z
coordinates : %lf\t ANGLE : %lf\n\n",x_acc,y_acc,z_acc,pitch);
                //stop = readTOD();
                //printf("Time elapsed = %lf\n", (double)(stop-start));
                sem_wait(&sem_t1);
            }
        }
    }
    close(file);
}

/* ITG-3200 Gyroscope functions */
void ITG3200_setup(int file)
{
    char id;
    id = i2c_smbus_read_byte_data(file, 0x00); //read from WHO_AM_I Register;

    printf("WHO AM I : %d\n\n",id);

    //Set the gyroscope scale for the outputs to +/-2000 degrees per second
    i2c_smbus_write_byte_data(file, DLPF_FS, 0x18);

    //Set the sample rate to 100 hz
    i2c_smbus_write_byte_data(file, SMPLRT_DIV, 0x09);
}

void* ITG3200_i2c(void *threadid) //GYROSCOPE
{
    int file,i;
    int adapter_no = 2;
    char filename[20];
    double stop = 0;

    int addr = 0x69; /* The I2C address */

    snprintf(filename, 19, "/dev/i2c-%d", adapter_no);
    file = open(filename , O_RDWR);

    if (file < 0)
    {
        perror("NO FILE DESCRIPTOR \n");
        exit(1);
    }
}

```

```

if (ioctl(file, I2C_SLAVE, addr) < 0)
{
    perror("I2C COMMUNICATION ESTABLISHMENT FAILED");
    exit(1);
}

ITG3200_setup(file);
while(1)
//for(i=0;i<10;i++)
{
    x_gyro = i2c_smbus_read_byte_data(file, ITG_X_CO_H)<<8;
    x_gyro |= i2c_smbus_read_byte_data(file, ITG_X_CO_L);

    y_gyro = i2c_smbus_read_byte_data(file, ITG_Y_CO_H)<<8;
    y_gyro |= i2c_smbus_read_byte_data(file, ITG_Y_CO_L);

    z_gyro = i2c_smbus_read_byte_data(file, ITG_Z_CO_H)<<8;
    z_gyro |= i2c_smbus_read_byte_data(file, ITG_Z_CO_L);

    //if ((x_gyro < 0) || (y_gyro<0) || (z_gyro<0))          /* ERROR HANDLING: i2c
transaction failed */
    //    printf("\n I2C TRANSACTION FAILED ");
    //else          /* res contains the read word */
    //    printf("GYROSCOPE : X co-ordinates : %d\t Y co-ordinates : %d\t Z
coordinates : %d\n",x_gyro,y_gyro,z_gyro);

    if(abort_task2 == 1)
    {
        printf("GYROSCOPE : X co-ordinates : %d\t Y co-ordinates : %d\t Z coordinates
: %d\n",x_gyro,y_gyro,z_gyro);
        //stop = readTOD();
        //printf("Time elapsed = %lf\n", (double)(stop-start));
        sem_wait(&sem_t2);
    }
}
close(file);
}

/* DC MOTOR FUNTIONS */
void stop(int file)
{
    int regValue = 0;
    i2c_smbus_write_byte_data(file, 0x00, regValue);
}

void drive(int file)
{
    int speed;
    char regValue;
    speed = 63;

    i2c_smbus_write_byte_data(file, 0x01, 0x80); // clear the fault status

    regValue = (char)abs(speed);
    regValue = regValue<<2;

    if(pitch >= 3.0)
        regValue |= 0x02;
    else if(pitch < -9.0)
        regValue |= 0x01;
    else if(pitch > -9.0 && pitch < 3.0)
        regValue = 0x00;

    i2c_smbus_write_byte_data(file, 0x00, regValue);
}

```



```

void* DC_motor(void *threadid)
{
    int file,i;
    int adapter_no = 2;
    char filename[20];
    double stop = 0;

    int addr = 0x68; /* The I2C address */

    snprintf(filename, 19, "/dev/i2c-%d", adapter_no);
    file = open(filename , O_RDWR);

    if (file < 0)
    {
        perror("NO FILE DESCRIPTOR \n");
        exit(1);
    }

    if (ioctl(file, I2C_SLAVE, addr) < 0)
    {
        perror("I2C COMMUNICATION ESTABLISHMENT FAILED");
        exit(1);
    }

    while(1)
    {
        drive(file);

        stop = readTOD();
        //printf("Time Elapsed: %lf\n\n", (double) (stop-start));

        if(abort_task3 == 1)
        {
            printf("DC Motor control\n");
            //stop = readTOD();
            //printf("Time elapsed = %lf\n", (double) (stop-start));
            sem_wait(&sem_t3);
        }
    }
}

int main(int argc, char *argv[])
{
    int rc;
    useconds_t t_50,t_500;
    abort_task1 = 1;
    abort_task2 = 1;
    abort_task3 = 1;
    sem_init (&sem_t1, 0, 0);
    sem_init (&sem_t2, 0, 0);
    sem_init (&sem_t3, 0, 0);

    t_50 = 50000;
    t_500 = 500000;

    pthread_attr_init(&task1_sched_attr);
    pthread_attr_init(&task2_sched_attr);
    pthread_attr_init(&task3_sched_attr);
    pthread_attr_init(&main_sched_attr);
    pthread_attr_setinheritsched(&task1_sched_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&task1_sched_attr, SCHED_FIFO);
    pthread_attr_setinheritsched(&task2_sched_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&task2_sched_attr, SCHED_FIFO);
    pthread_attr_setinheritsched(&task3_sched_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&task3_sched_attr, SCHED_FIFO);
    pthread_attr_setinheritsched(&main_sched_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&main_sched_attr, SCHED_FIFO);
    rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    rt_min_prio = sched_get_priority_min(SCHED_FIFO);

```

```

main_param.sched_priority = rt_max_prio;
task1_param.sched_priority = rt_max_prio-1;
task2_param.sched_priority = rt_max_prio-2;
task3_param.sched_priority = rt_max_prio-3;

rc=sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
if (rc)
{
printf("ERROR; sched_setscheduler rc is %d\n", rc); perror(NULL); exit(-1);
}

pthread_attr_setschedparam(&task1_sched_attr, &task1_param);
pthread_attr_setschedparam(&task2_sched_attr, &task2_param);
pthread_attr_setschedparam(&task3_sched_attr, &task3_param);
pthread_attr_setschedparam(&main_sched_attr, &main_param);

//start = readTOD();
rc= pthread_create (&task1, &task1_sched_attr, ADXL345_i2c, ( void *)0 );
if (rc)
{
printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);
}
rc= pthread_create (&task2, &task2_sched_attr, ITG3200_i2c, ( void *)0 );
if (rc)
{
printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);
}
rc= pthread_create (&task3, &task3_sched_attr, DC_motor, ( void *)0 );
if (rc)
{
printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);
}

/* Basic sequence of releases after CI */

start = readTOD();
while(1)
{
abort_task1 = 0;
sem_post(&sem_t1);
usleep(5000);
abort_task1 = 1;
stop = readTOD();
printf("Time elapsed = %lf\n", (double)(stop-start));

abort_task2 = 0;
sem_post (&sem_t2);
usleep(5000);
abort_task2 = 1;
stop = readTOD();
printf("Time elapsed = %lf\n", (double)(stop-start));

abort_task3 = 0;
sem_post(&sem_t3);
usleep(10000);
abort_task3 = 1;
stop = readTOD();
printf("Time elapsed = %lf\n", (double)(stop-start));
}

pthread_join(task1, NULL);
pthread_join(task2, NULL);
pthread_join(task3, NULL);
if(pthread_attr_destroy(&task1_sched_attr) != 0)
perror("attr destroy");
if(pthread_attr_destroy(&task2_sched_attr) != 0)
perror("attr destroy");
if(pthread_attr_destroy(&task3_sched_attr) != 0)
perror("attr destroy");
}

```