Self-Balancing Robot

Internees

Muhammad Zeeshan Waleed Umer Muhammad Ali
BS Electronics Engineering (6th Semester)
PIEAS

Supervisor

Muhammad Rizwan Chughtai
Research Officer

NATIONAL INSTITUTE OF ELECTRONICS
MINISTRY OF SCIENCE AND TECHNOLOGY
ISLAMABAD
September, 2021

Self-Balancing Robot

Internees

Muhammad Zeeshan
Waleed Umer
Muhammad Ali

BS Electronics Engineering (6th Semester)
PIEAS

A project repot is submitted for fulfillment of internship

Supervisor:

Muhammad Rizwan Chughtai

Research Officer

Lab Head Signature:_____

Supervisor Signature: _____

NATIONAL INSTITUTE OF ELECTRONICS
MINISTRY OF SCIENCE AND TECHNOLOGY
ISLAMABAD
September 2021

ACKNOWLEDGEMENTS

We are extremely grateful to NIE for providing us this opportunity of internship at their office. And we are also thankful to Mr. Muhammad Rizwan Chughtai for guiding us throughout this six week internship. His technical insight helped us explore sensors in a novel way. We are also thankful to Mr. Tanveer Abbas, internship coordinator at PIEAS, for choosing us for this internship.

ABSTRACT

This report provides a detailed description of our project on Self Balancing Robot done during internship at NIE. We built an upright, self-balancing, two-wheeled robot utilizing an IMU and a PID feedback control loop to maintain stability. This report contains a thorough discussion of our project, including details about the mechanics, electronics, software, and everything else that went into designing, building, and testing our self-balancing robot.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I: INTRODUCTION

1.1 Statement of the Problem

We are required to develop a self-balance robot data without the aid of library available for the sensors concerned in the project.

1.2 Objectives:

- Interfacing motor with Arduino through motor driver L298
- Interfacing MPU 6050 with sensor through basic equations in the datasheet
- Employing PID control for the self-balancing action and tuning the values for K_i ,K_d ,K_p .

1.3 Organization of the thesis:

- Introduction
- Components
- Software Architecture
- Conclusion

CHAPTER II: COMPONENTS

The following electronic components are used in our project:

- Arduino Uno
- MPU 6050
- Geared DC Motor
- Motor Driver (L298)
- Battery

The functionality and use of each of these components is described in detail below:

2.1. MPU 6050

MPU6050 is a Micro Electro-mechanical system (MEMS), it consists of three-axis accelerometer and three-axis gyroscope. It helps us to measure velocity, orientation, acceleration, displacement and other motion like features.

MPU6050 consists of Digital Motion Processor (DMP), which has property to solve complex calculations. MPU 6050 consists of a 16-bit analog to digital converter hardware. Due to this feature, it captures three-dimension motion at the same time.

This module has some famous features which are easily accessible, due to its easy availability it can be used with a famous microcontroller like Arduino. If you are looking for a sensor to control a motion of your Drone, Self-Balancing Robot, RC Cars and something like this, then MPU6050 will be a good choice for you. This module uses the I2C module for interfacing with Arduino.

MPU6050 is less expensive, its main feature is that it can easily combine with accelerometer and gyro

.

Figure 1: MPU 6050 Sensor

2.1.1. Specifications

MPU6050 module is composed of the following blocks and functions.

- A 3-axis MEMS rate gyroscope sensor with three 16-bit ADC's and signal conditioning.
- A 3-axis MEMS accelerometer sensor with three 16-bit ADC's and signal conditioning.
- An on-chip Digital motion Processor engine.
- Primary 12C digital communication interfaces.
- Auxiliary I2C interfaces for communication with external sensors such as Magnetometer.
- Internal Clocking.
- Data registers for storing sensor data.
- FIFO memory which helps in reducing power consumption.
- User-programmable interrupts.
- A digital output temperature sensor.
- Self-test for gyroscope and accelerometer.

- LDO and Bias.
- Charge Pump.
- Status registers.

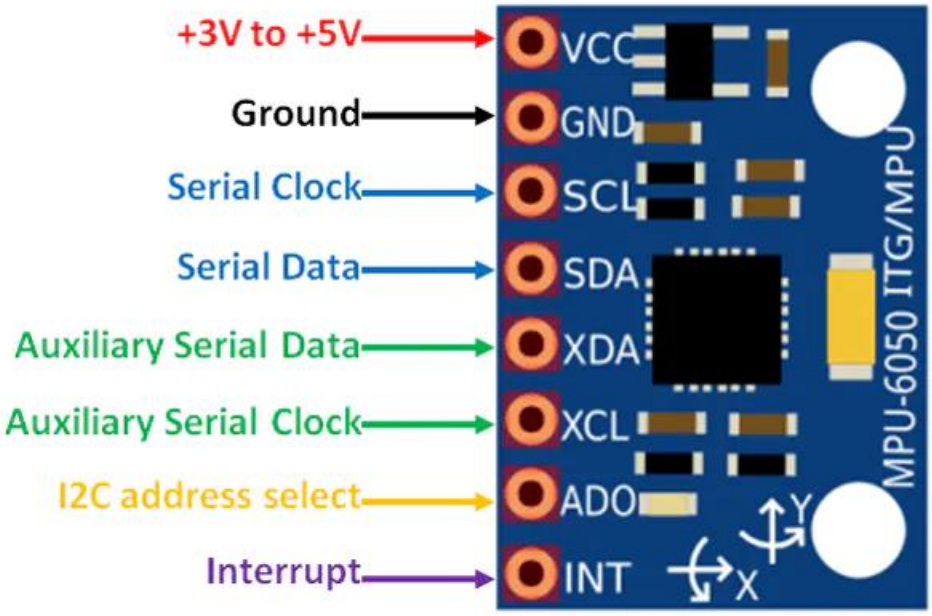

Figure 2.2 MPU 6050 Pin Diagram

2.2. Arduino Uno

We are using Arduino Uno microcontroller as our processing unit. It is based on the ATmega328P. Its primary function is to acquire data from MPU 6050, process it to get orientation of robot and then set the angular velocities of DC motors through L298 drivers to balance the robot. It executes PID controller to set motor velocities. It is powered by +5V output from L298 motor driver.

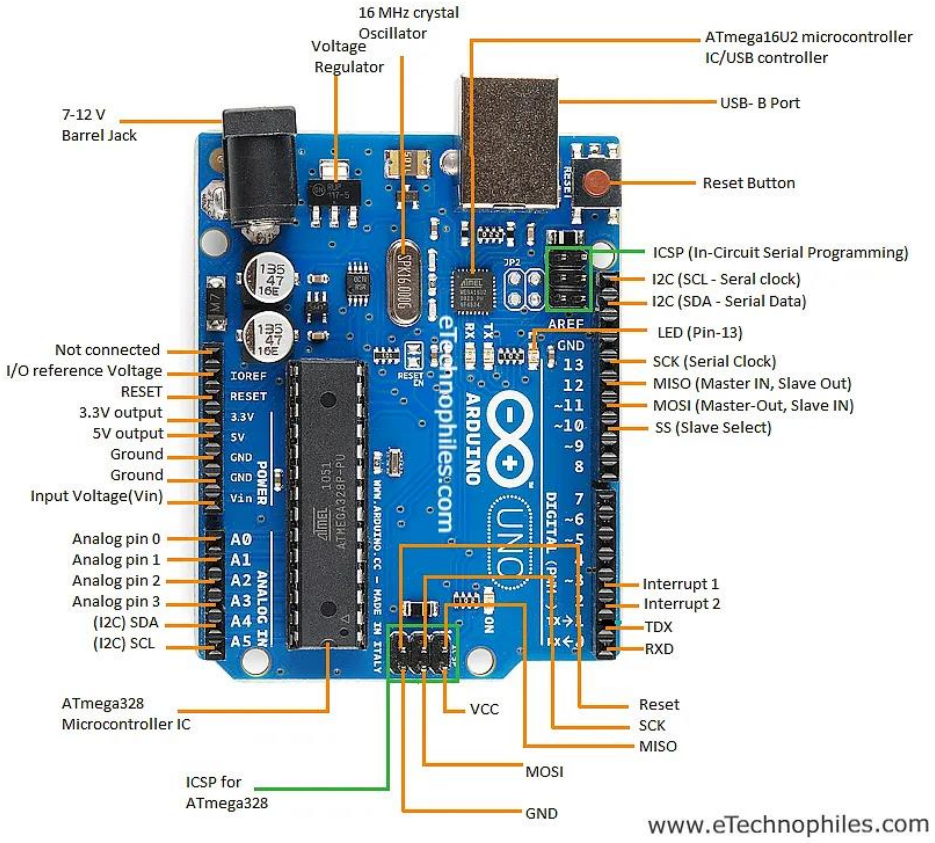

Figure 2.3 Arduino Uno

2.2.1. I2C-Communication protocol:

I2C is the two-wire serial communication protocol. It stands for Inter-Integrated Circuits. The I2C uses two lines to send and receive data: a serial clock pin uses (SCL) and a serial data (SDA) (SDA) pin.

SCL: It stands for **Serial Clock**. It is the pin or line that transfers the clock data. It is used to synchronize the shift of data between the two devices (master and slave). The Serial Clock is generated by the master device.

SDA:It stands for **Serial Data**. It is defined as the line used by the slave and master to send and receive the data. That's why it is called the **data line,** while SCL is called a clock line.

Figure 2.4 I2C Pins

2.3. Geared DC Motor

A geared DC Motor has a gear assembly attached to the motor. The speed of motor is counted in terms of rotations of the shaft per minute and is termed as RPM. The gear assembly helps in increasing the torque and reducing the speed. Using the correct combination of gears in a gear motor, its speed can be reduced to any desirable figure.

We interfaced the geared dc motor with our arduino uno board.

Figure 2.5 Geared DC motor

2.4. Motor Driver (L298)

L298N module is a high voltage, high current dual full-bridge motor driver module for controlling DC motor and stepper motor. It can control both the speed and rotation direction

of two DC motors. This module consists of an L298 dual-channel H-Bridge motor driver IC. This module uses two techniques for the control speed and rotation direction of the DC motors. These are PWM – For controlling the speed and H-Bridge – For controlling rotation direction. These modules can control two DC motor or one stepper motor at the same time.This motor driver module consists of two main key components, these are L298 motor driver IC and a 78M05 5V regulator.

According to the L298 datasheet, its operating voltage is +5 to +46V, and the maximum current allowed to draw through each output 3A. This IC has two enable inputs, these are provided to enable or disable the device independently of the input signals.

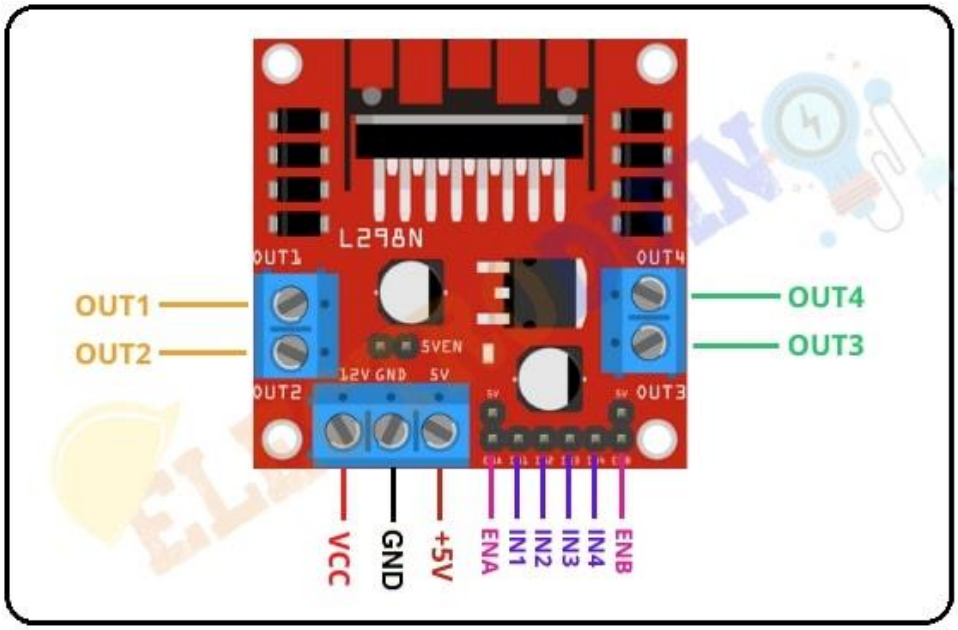

Figure 2.6 Motor Driver L298

2.4.1. PWM Techniques:

L298n motor driver module uses the PWM technique to control the speed of rotation of a DC motor. In this technique, the speed of a DC motor can be controlled by changing its input voltage.

Pulse Width Modulation is a technique where the average value of the input voltage is adjusted by sending a series of ON-OFF pulses. The average voltage is proportional to the width of the pulses, these pulses known as Duty Cycle.

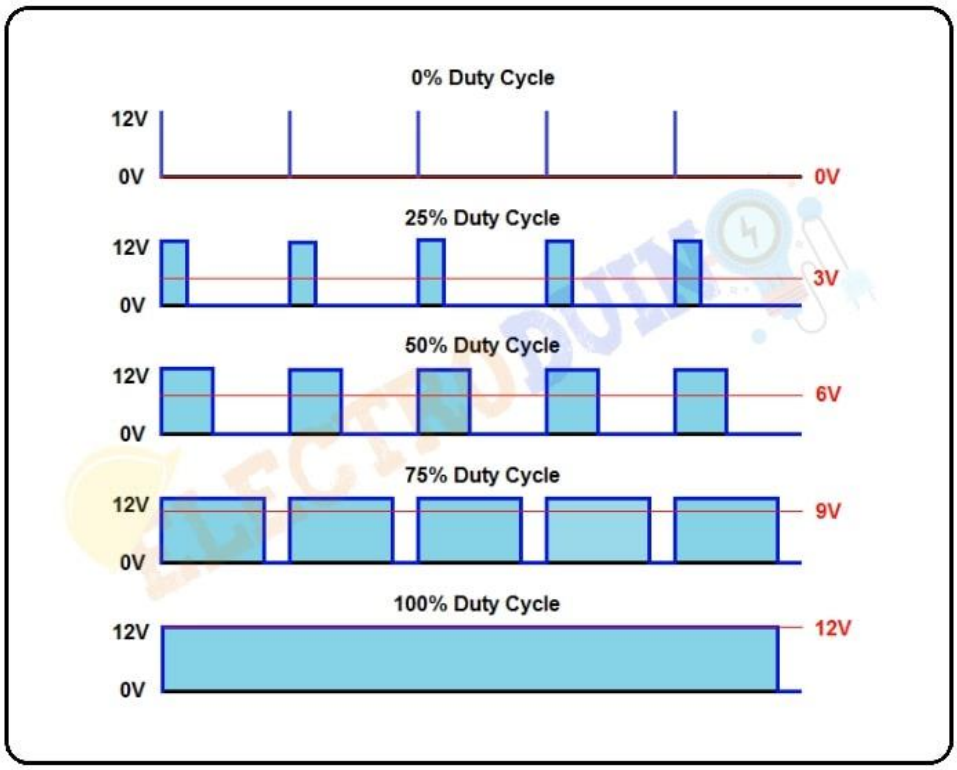

Figure 2.7PWM duty cycle profile

2.5. Battery

Lipo battery voltage is specified by the number of cells in series. Each cell has a nominal voltage 3.7 V. Different battery companies mark their lipos in different ways but most people tend to refer to their batteries as 1S, 2S, 3S, etc.

As we used battery with two cells so 2S .So it's voltage would be as follows:

1S = 1 cell in series x 3.7 V = 3.7 V

2S = 2 cells in series x 3.7 V = 7.4 V

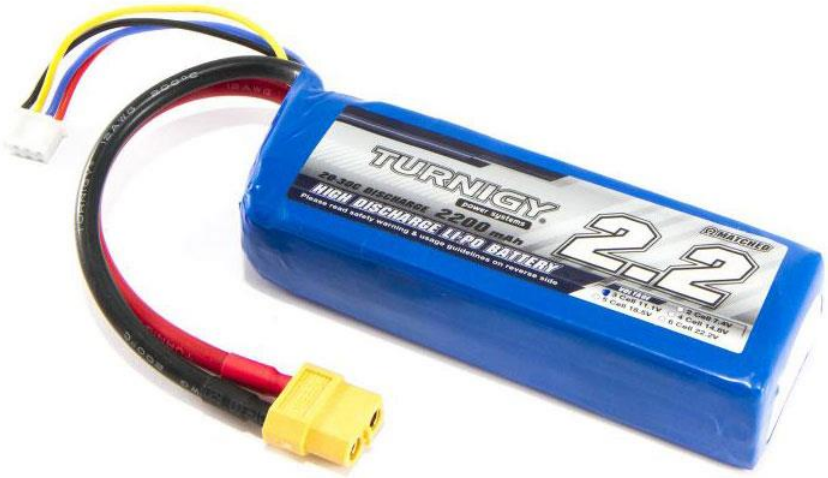

Figure 2.8 Lipo dual cell battery

We will be connecting our battery with arduino as given below:

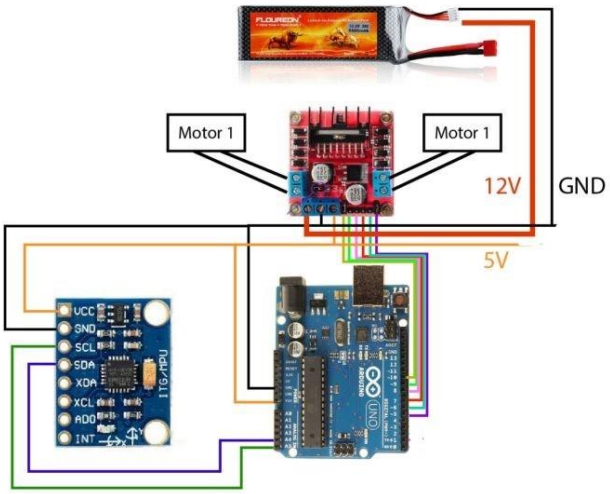

Figure 2.9 Complete Configuration of Circuit

CHAPTER III: PHYSICAL LAYOUT

We used the following components to make the self-balancing robot:

- Tires
- Plastic Sheet
- Screws
- Nuts
- Electronic components:
    - Arduino Uno
    - Lipo battery
    - Motor Driver
    - MPU6050 IMU
    - Geared Motor
- Plastic strips
- Hot air gun
- Metal saw
- Glue gun

We made the self-balancing robot from Plastic sheet, geared motor, tires, and other electronic components.

First we marked the plastic sheet according to correct dimensions, then we cut the sheet with the help of metal saw. Then we bend the sheet with the help of hot air gun to give it the correct shape.

After that we made holes in plastic sheet for all the components with the help of drill. We first mounted geared motors on the plastic structure with the help of plastic strips but they weren’t stable, so we use glue gun to make them stable.

Then we mounted all the electronic components on the sheet with the help of screws and nuts.

CHAPTER IV: SOFTWARE ARCHITECTURE

4.1. Motor Control

The self-balancing robot will have two motors: one on right and other on the left. A DC motor has two wires coming out of it. The direction and speed of the motor can be varied by changing the polarity and magnitude of applied DC voltage. This is done using L298 motor driver. Two PWM pins on Arduino uno are used to control the speed of a single motor. The right motor is connected to pins 7 and 8 while the left motor is connected to pins 5 and 6. All these are digital pins and are set to *OUTPUT*. There are also enable pins: one for the right motor (connected to pin 9 on Arduino) and one for the left motor (connected to pin 10 on Arduino). Both these are PWM pins. The speed of the motor is controlled by changing the duty cycle of PWM output from these pins.

The Arduino function used to generate a PWM signal is *analogWrite()*. By default, it has a resolution of 8 bits i.e., it can take values from 0 to 255. However, for better speed control, the resolution can be increased to 16 bits by using 16-bit timer (Timer1). The range of values, then become 0 to 65535.

The code of controlling the speed of motors is given in appendix A.1.

4.2. Gyroscope Data Acquisition

The MPU 6050 contains 3 axis gyroscope that can provide angular rate of change about the X, Y and Z axis. The sensor communicates with Arduino using I2C communication protocol. According to the sensor data sheet, to access MPU 6050 data, we use the address *0x68*. The precision of gyroscope data is 16 bits whereas the sensor has 8 bits wide registers. The accelerometer data starts from address 0x43. Therefore, 6 registers need to

be read to get the angular velocity along X, Y and Z axis respectively starting from address 0x43. Note that the lower register contains most significant byte (MSB), therefore it is shifted to the left by 8 bits, ORed with least significant byte (LSB) and stored in a variable holding gyroscope data as shown in figure ():

```
  Wire.beginTransmission(MPU);
  Wire.write(0x43);
  Wire.endTransmission(false); //must have
  Wire.requestFrom(MPU, 6, true);
GyroX = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroY = (Wire.read() << 8 | Wire.read()) / 131.0;
GyroZ = (Wire.read() << 8 | Wire.read()) / 131.0;
```

Figure 4.1 Code snap for gyro

The factor 131.0 comes from the data sheet which is sensitivity scale factor. Additionally, to remover offset error, we take 200 readings at rest, take average, and subtract the average error from each gyroscope axis reading we take. The code is given in appendix A.2.

4.3. Accelerometer Data Acquisition

The MPU 6050 also contains a 3-axis accelerometer that provides linear acceleration along X, Y and Z axis. The precision of accelerometer data is 16 bits whereas the sensor has 8 bits wide registers. The accelerometer data starts from address 0x3B. Therefore, 6 registers need to be read to get the linear acceleration along X, Y and Z axis respectively starting from address 0x3B. Similar to gyroscope data, the lower register contains most significant byte (MSB), therefore it is shifted to the left by 8 bits, ORed with least significant byte (LSB) and stored in a variable holding accelerometer data as shown in figure ():

```
Wire.beginTransmission(MPU);
Wire.write(0x3B);
Wire.endTransmission(false);
Wire.requestFrom(MPU, 6, true);
AccX = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
AccY = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0 ;
```

Figure 4.1 Code snap for accelerometer

The factor 16384.0 comes from the data sheet which is sensitivity scale factor. Additionally, to remover offset error, we take 200 readings at rest, take average, and subtract the average error from each accelerometer axis reading we take. The code is given in appendix A.2.

4.4. Extracting Orientation Information

Orientation of an object in 3D space is described with yaw, pitch and roll which is rotation of the object about cartesian coordinate axes i.e., around X, Y and Z axis. Orientation information can be extracted from gyroscope by integrating it. However, it suffers from drift error because of offset error. Therefore, we balance it by calculating the orientation from accelerometer. We combine the two by using complementary filter as:

$$Orientation = a * (Orientation\ from\ Accelerometer) + (1 - a) * (Orientation\ from\ Gyroscope)$$

Where a < 1.

The code is given in appendix A.2.

4.5. Timer Interrupt

We are using timer interrupt to get data from gyroscope and accelerometer at a frequency of 125Hz i.e., after every 8ms. This time interval also defines how often our main code will

be executed. In other words, the main program (loop function) will run 8000 times in one second. To generate periodic timer interrupt, we are using timer 2 which is an 8-bit counter. The counter is incremented on each tick of the timer's clock. CTC (Clear Timer on Compare Match) mode timer interrupts are triggered when the counter reaches a specified value, stored in the compare match register. Once a timer counter reaches this value, it will clear (reset to zero) on the next tick of the timer's clock and will continue to count up to the compare match value again. By choosing the compare match value and setting the speed at which the timer increments the counter, we can control the frequency of timer interrupts.

The speed at which the timer increments, depends on the clock frequency and a pre-scalar defined by the data sheet as shown in figure ().

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Figure 4.2 Timer Interrupt Table

The interrupt frequency is defined as:

$$Interrupt\ Frequency = \frac{Clock\ Frequency}{Prescalar * (Compare\ Match\ Register + 1)}$$

If the interrupt frequency is known as in our case, we can calculate the compare match register value using the above equation in the form:

$$Compare\ Match\ Register = \frac{Clock\ Frequency}{Prescalar * Desired\ Interrupt\ Frequency} - 1$$

The compare match register value comes out to be 124 for generating 125Hz interrupt. The code for generating interrupt is given in appendix A.3.

4.6. PID Controller

Proportional-Integral-Derivative (PID) is a cornerstone algorithm in control theory. The PID algorithm smoothly and precisely controls a system. A PID controller works by calculating an amount of error based upon the difference between a set (desired) value and a feedback (current) value and provides an adjustment to the output to correct that error. The control and decision of the adjustment is done using mathematical tools such as differentiator, integrator instead of simple logic statements such as if...else statements.

4.6.1. Mathematics Behind PID:

Setting up a PID controller involves solving an algorithm recursively. The output is sum of three parts as indicated by the name: proportional, integral, and derivative. The equation for the PID algorithm attempts to lower difference between a setpoint (the value desired) and a measured value, also known as the feedback. The output is altered so that the setpoint is maintained. PID controllers can easily work with systems that have control over a variable output by providing error signal to actuator that can change the state of our system. The block diagram is shown in figure below:

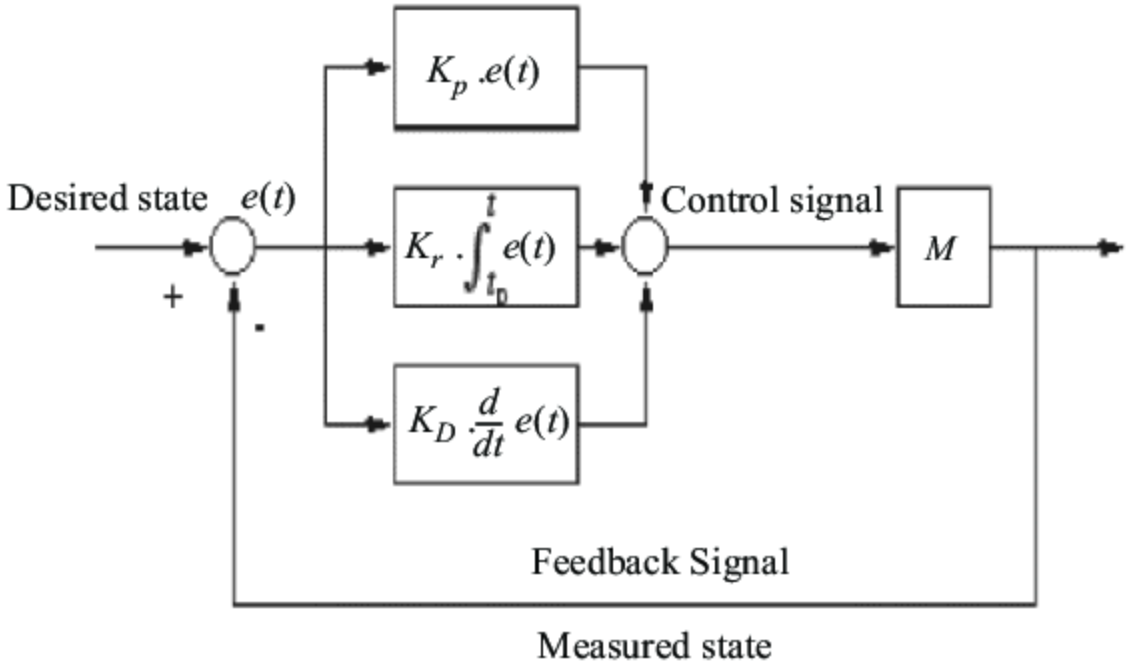

Figure 4.3 PID Control Diagram

The variables of the equation are

- **e(t):** The calculated error determined by subtracting the output (current state) from the setpoint.
- **Δt:** Time difference between two consecutive runs of PID. It also determines how often the state of the system is updated.
- **K_p:** Gain for the proportional component
- **K_i:** Gain for the integral component
- **K_d:** Gain for the derivative component

4.6.2. The Proportional Term:

The P in PID is a proportional term that amplifies the error. Kp is the gain value and determines how the P statement reacts to change in error; the lower the gain, the less the

system reacts to an error. Kp is what tunes the proportional part of the equation. The gain value is set either predefined or dynamically changed via a user input. The proportional statement aids in the steady-state error control by always trying to keep the error minimal. The steady state describes when a system has reached the desired setpoint. The first part of the proportional code will calculate the amount of error and will appear something like this:

$$error = Setpoint - input$$

The second part of the code multiplies the error by the gain variable:

$$Pout = Kp * error$$

The proportional statement attempts to lower the error to zero where $input = Setpoint$. A pure proportional controller, with this equation and code, will not settle at the setpoint, but usually somewhere below the setpoint. It is because the proportional control always tries to reach a value of zero, and the settling is a balance between the input and the feedback. The integral statement is responsible for achieving the desired setpoint.

4.6.3. The Integral Term:

The I in PID is for integration. Put simply, integration is the calculation of the area under a curve. This is accomplished by constantly adding a very small area to an accumulated total. The area is calculated by length × width; to find the area under a curve, the length is determined by the function's value and a small difference that then is added to all other function values. For reference, the integral in this type of setup is similar to a Riemann sum. The PID algorithm does not have a specific function; the length is determined by the

error, and the width of the rectangle is the change in time. The program constantly adds this area up based on the error. The code for the integral is:

$$errorsum = (errorsum\ + currenterror\ ) * \Delta t$$

$$Iout = Ki * \Delta t$$

The integral reacts to the amount of error multiplied by the duration of error. The *errorsum* value increases when the input value is below the *setpoint*, and decreases when the input is above the *setpoint*. The integral will hold at the setpoint when the error becomes zero and there is nothing to subtract or add. When the integral is added to proportional statement, the integral corrects for the offset to the error caused by the proportional statement's settling. The integral will control how fast the algorithm attempts to reach the setpoint: lower gain values approach at a slower rate; higher values approach the setpoint quicker, but have the tendency to overshoot and can cause ringing by constantly overshooting above and below the setpoint and never settling.

4.6.4. The Derivative Statement:

The D in PID is the derivative term, which is just a snapshot of the slope of an equation. The slope is calculated as rise over run—the rise comes from the change in the error, or the current error subtracted from the last error; the run is the change in time. When the rise is divided by the time change, the rate at which the input is changing is known. Code for the derivative component is

$$Derror = \frac{error - lasterror}{timechange}$$

$$Dout = Kd * Derror$$

Or

$$Derror = \frac{input - lastinput}{timechange}$$

$$Dout = Kd * Derror$$

The derivative aids in the control of overshooting and controls the ringing that can occur from the integral. High gain values in the derivative can have a tendency to cause an unstable system that will never reach a stable state. The two versions of code both work, and mostly serve the same function. The code that uses the slope of the input reduces the derivative kick caused when the setpoint is changed; this is good for systems in which the setpoint changes regularly. By using the input instead of the calculated error, we get a better calculation on how the system is changing; the code that is based on the error will have a greater perceived change, and thus a higher slope will be added to the final output of the PID controller.

4.6.5. Adding It All Up:

With the individual parts calculated, the proportion, integral, and the derivative terms have to be added together to achieve a usable output. One line of code is used to produce the output:

$$Output = Pout + Iout + Dout$$

4.6.6. Tuning the PID Constants for self-balancing robot

1. Set Ki and K_d to zero and gradually increase Kp so that the robot starts to oscillate about its stable vertical position.

2. Increase K_i so that the response of the robot is faster when it is out of balance. Ki should be large enough so that the angle of inclination does not increase. The robot should come back to stable position if it is inclined.
3. Increase K_d so as to reduce the oscillations. The overshoots will also be decrease by increasing K_d.
4. Repeat the above steps by fine tuning each parameter to achieve the best result.

CONCLUSION

In this project of self-balancing car ,we interfaced mpu 6050 sensor with arduino through equations rather than the library available for it which made the calibration of our sensor quite difficult to achieve . Moreover , we employed PID control for the self –balancing action . The proportional statement aids in the steady-state error control by always trying to keep the error minimal. The integral will control how fast the algorithm attempts to reach the setpoint: lower gain values approach at a slower rate; higher values approach the setpoint quicker . The derivative aids in the control of overshooting and controls the ringing that can occur from the integral. High gain values in the derivative can have a tendency to cause an unstable system that will never reach a stable state. That's why for K_d value is always kept smaller . In mpu 6050 calibration each axis calibration was done separately and then combined

APPENDIX A

A.1.1. Motor Control

```
//right motor to pins: 7(in3) and 8(in4)
//right motor speed control pin: 9
//left motor to pins: 6(in1) and 5(in2)
//left motor speed control pin: 10

#define in1 6
#define in2 5
#define in3 7
#define in4 8
#define enaR 9
#define enaL 10
void setup() {
 setupPWM16();
 pinMode(in1, OUTPUT);
 pinMode(in2, OUTPUT);
 pinMode(in3, OUTPUT);
 pinMode(in4, OUTPUT);
 pinMode(enaR, OUTPUT);
 pinMode(enaL, OUTPUT);

}
void loop() {
 for (int i=0; i<=65535; i=i+5000) {
 movForw(i);
 delay(1000);
 }
}

void movForw(uint16_t setspeed){
 analogWrite16(enaR, 0);
 analogWrite16(enaL, 0);
 digitalWrite(in1, LOW);
 digitalWrite(in2, HIGH);
 digitalWrite(in3, LOW);
 digitalWrite(in4, HIGH);
 analogWrite16(enaR, setspeed);
 analogWrite16(enaL, setspeed);
}

void movBack(uint16_t setspeed){
 analogWrite16(enaR, 0);
 analogWrite16(enaL, 0);
 digitalWrite(in1, HIGH);
 digitalWrite(in2, LOW);
 digitalWrite(in3, HIGH);
 digitalWrite(in4, LOW);
 analogWrite16(enaR, setspeed);
 analogWrite16(enaL, setspeed);
}

void setupPWM16() {
   DDRB |= _BV(PB1) | _BV(PB2);
/* set pins as outputs */
   TCCR1A = _BV(COM1A1) |
_BV(COM1B1) /* non-inverting PWM
*/
      | _BV(WGM11); /* mode
14: fast PWM, TOP=ICR1 */
   TCCR1B = _BV(WGM13) |
_BV(WGM12)
      | _BV(CS11); /* prescaler:
clock / 8 */
   ICR1 = 0xffff; /* TOP
counter value (freeing OCR1A*/
}

void analogWrite16(uint8_t pin, uint16_t
val)
{
   switch (pin) {
      case 9: OCR1A = val; break;
      case 10: OCR1B = val; break;
```

```
  }
```

A.1.2. Timer Interrupt

```
bool toggle2 = 0;
void setup() {
 // put your setup code here, to run once:
 Serial.begin(115200);
 pinMode(7, OUTPUT);
 initTimer();
}

void loop() {
 // put your main code here, to run repeatedly:
}

ISR(TIMER2_COMPA_vect){
 //timer1 interrupt 100Hz
 if (toggle2){
  digitalWrite(7,HIGH);
  toggle2 = 0;
 }
 else{
  digitalWrite(7,LOW);
  toggle2 = 1;
 }
}

void initTimer(){
 cli();//stop interrupts
 //set timer2 interrupt at 125Hz
 TCCR2A = 0;// set entire TCCR2A register to 0
 TCCR2B = 0;// same for TCCR2B
 TCNT2  = 0;//initialize counter value to 0
 // set compare match register for 100Hz interrupt frequency
 OCR2A = 124;// = ((16*10^6) / (125*1024)) - 1 (must be <256)
 // turn on CTC mode
 TCCR2A |= (1 << WGM21);
 // Set CS21 bit for 1024 prescaler
 TCCR2B |= 0x07;
 // enable timer compare interrupt
 TIMSK2 |= (1 << OCIE2A);
 sei();//allow interrupts
}
```

A.1.3. Acquiring Data From MPU 6050

```
#include <Wire.h>

const int MPU = 0x68; // MPU6050 I2C address
struct vector {
  float Xaxis;
  float Yaxis;
  float Zaxis;
};

vector Gyro, Acc, GyroError, AccError;
uint8_t GyroXHigh, GyroYHigh, GyroZHigh;
uint8_t GyroXLow, GyroYLow, GyroZLow;
uint8_t AccXHigh, AccYHigh, AccZHigh;
uint8_t AccXLow, AccYLow, AccZLow;
float accAngleX, accAngleY, gyroAngleX, gyroAngleY, gyroAngleZ;
float roll, pitch, yaw;

void setup() {
  // initialize MPU6050
  init_MPU6050();
  // calculate Gyro error
  calculate_gyro_error();
  delay(10);
  initTimer(); //should be in the last. When called, timer starts counting.
}

void loop() {
 //PID here
}

ISR(TIMER0_COMPA_vect){
  //timer1 interrupt 125Hz
  // Read gyro values
  readGyro();
  // Calculate Pitch, Roll and Yaw
  yaw += Gyro.Zaxis * 0.01;
  gyroAngleX += Gyro.Xaxis * 0.01;
  gyroAngleY +=  Gyro.Yaxis * 0.01;

  accAngleX += ((atan((Acc.Yaxis) / sqrt(pow((Acc.Xaxis), 2) + pow((Acc.Zaxis), 2))) * 180 / PI));
  accAngleY += ((atan(-1 * (Acc.Xaxis) / sqrt(pow((Acc.Yaxis), 2) + pow((Acc.Zaxis), 2))) * 180 / PI));
  accAngleX -= AccError.Xaxis;
  accAngleY -= AccError.Yaxis;

  //complementary filter
  roll = 0.96 * gyroAngleX + 0.04 * accAngleX;
  pitch = 0.96 * gyroAngleY + 0.04 * accAngleY;
  Serial.print(" Pitch = ");
  Serial.print(pitch);
  Serial.print(" Roll = ");
  Serial.print(roll);
  Serial.print(" Yaw = ");
  Serial.println(yaw);
}

void initTimer(){
  cli();//stop interrupts
  //set timer2 interrupt at 100Hz
  TCCR2A = 0;// set entire TCCR2A register to 0
  TCCR2B = 0;// same for TCCR2B
  TCNT2  = 0;//initialize counter value to 0
  // set compare match register for 100Hz interrupt frequency
  OCR2A = 155;// = ((16*10^6) / (100*1024)) - 1 (must be <256)
  // turn on CTC mode
  TCCR2A |= (1 << WGM21);
  // Set CS21 bit for 1024 prescaler
  TCCR2B |= 0x07;
  // enable timer compare interrupt
  TIMSK2 |= (1 << OCIE2A);
  sei();//allow interrupts
```

```
}

vector readGyro(void){
  Wire.beginTransmission(MPU);
  Wire.write(0x43);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  GyroXHigh = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x44);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  GyroXLow = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x45);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  GyroYHigh = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x46);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  GyroYLow = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x47);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  GyroZHigh = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x48);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  GyroZLow = Wire.read();
  Wire.endTransmission(true);

  Gyro.Xaxis = (GyroXHigh << 8 |
GyroXLow) / 131.0;
  Gyro.Yaxis = (GyroYHigh << 8 |
GyroYLow) / 131.0;
  Gyro.Zaxis = (GyroZHigh << 8 |
GyroZLow) / 131.0;

  Gyro.Xaxis -= GyroError.Xaxis;
  Gyro.Yaxis -= GyroError.Yaxis;
  Gyro.Zaxis -= GyroError.Zaxis;
  return Gyro;
}

vector readAcc(void){
  Wire.beginTransmission(MPU);
  Wire.write(0x3B);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  AccXHigh = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x3C);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  AccXLow = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x3D);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  AccYHigh = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x3E);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  AccYLow = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x3F);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
```

```
  AccZHigh = Wire.read();
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x40);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 1, true);
  AccZLow = Wire.read();
  Wire.endTransmission(true);

  Acc.Xaxis = (AccXHigh << 8 | AccXLow) / 16384.0;
  Acc.Yaxis = (AccYHigh << 8 | AccYLow) / 16384.0;
  Acc.Zaxis = (AccZHigh << 8 | AccZLow) / 16384.0;

  return Acc;
}

void calculate_acc_error() {
  // We can call this funtion in the setup section to calculate the accelero data error.
  // Read accelerometer values 200 times
  uint8_t c = 0;
  float SumX = 0, SumY = 0;
  while (c < 200) {
    readAcc();
    // Sum all readings
     SumX += ((atan((Acc.Yaxis) / sqrt(pow((Acc.Xaxis), 2) + pow((Acc.Zaxis), 2))) * 180 / PI));
     SumY += ((atan(-1 * (Acc.Xaxis) / sqrt(pow((Acc.Yaxis), 2) + pow((Acc.Zaxis), 2))) * 180 / PI));
    c++;
  }
  //Divide the sum by 200 to get the error value
  AccError.Xaxis = SumX / 200;
  AccError.Yaxis = SumY / 200;

  Serial.print("AccErrorX: ");
  Serial.println(AccError.Xaxis);
  Serial.print("AccErrorY: ");
  Serial.println(AccError.Yaxis);
}

void calculate_gyro_error(){
  uint8_t c = 0;
  float SumX = 0;
  float SumY = 0;
  float SumZ = 0;
  // Read gyro values 200 times
  while (c < 200) {
    readGyro();
    // Sum all readings
    SumX += Gyro.Xaxis;
    SumY += Gyro.Yaxis;
    SumZ += Gyro.Zaxis;
    c++;
    delay(10);
  }

  //Divide the sum by 200 to get the error value
  GyroError.Xaxis = SumX / 200;
  GyroError.Yaxis = SumY / 200;
  GyroError.Zaxis = SumZ / 200;

  // Print the error values on the Serial Monitor
  Serial.print("GyroErrorX: ");
  Serial.println(GyroError.Xaxis);
  Serial.print("GyroErrorY: ");
  Serial.println(GyroError.Yaxis);
  Serial.print("GyroErrorZ: ");
  Serial.println(GyroError.Zaxis);
}

void init_MPU6050(void){
  Serial.begin(9600);
  Wire.begin();                 // Initialize comunication
  Wire.beginTransmission(MPU);  // Start communication with MPU6050 // MPU=0x68
  Wire.write(0x6B);             // Talk to the register 6B
  Wire.write(0x00);             // Make reset - place a 0 into the 6B register
```

```
  Wire.endTransmission(true);         //end          }
the transmission
```

ANNEXURE

Annexure (if any) should be placed at the end of thesis report.

Hard Copy Information*:*

Page Size: A4

Page Margins: Top & Bottom = 1” and Left & Right = 1.25”

Page orientation: Portrait

Line Spacing: Double

Page Numbering: Bottom Center

Printing: One Side

Note about Binding**:**

Kindly submit report in spiral binding