HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

SELF BALANCING TWO WHEEL ROBOT

Group Members	Matriculation #
Chandrahas Kasoju	35070
Jagruth Medavarapu	35111
Praneeth Avula	35139
Osama Bin Zafar	35052

A scientific project submitted in partial fulfillment of the requirements for the degree of Master of Science in Mechatronics (MM)

Supervisor : Prof. Dr-Ing. Raphael Ruf

Date: 27th September 2022

Table of Contents

List of Figures

List of Tables

Abstract

The main objective of this project is to design and construct a two wheel robot which will balance and prevent itself from falling hence called “Self-balancing two wheel robot”. The main components are STM32F446RE microcontroller, MPU6050 inertial sensor, two stepper motors, motor driver, CNC shield, and power supply. The design of robot is based on big wheels with vertical platforms in which hardware is mounted. The designed of the robot was done in solid works and then manufactured in house. Firstly, with the help of MPU6050 we find out the angular rotation values of robot so that if it’s falling forward it will send signals to stepper motors through microcontroller STM32F446 to move in forward direction and vice versa. A PID controller algorithm is used to minimize the error value between the current and targeted value. STM32 Cube IDE is used to program the robot.

1. Introduction

These days, robots are integral part of human life. Complex and hard tasks can easily be performed by robots with high level of precision. The field of advanced robotics has great impact on the individual intellect worldwide. It was the desire and ambition of humans to create such a machine that recreates and replicates them in every aspect of day-to-day life. They could use such a machine that can reflect a person's thoughts, gestures, postures, and carry out everyday tasks. As this field has developed over the last few decades, dreams have become a reality. We have achieved this success so much thanks to the utilization of microcontrollers and sensitive sensors. At present, we encounter robots on a daily basis. There are many types of robots in use today, whether they are used in factories on production lines, developed as smart machines to perform certain tasks, or used as business tools. In comparison with humanoid types of robots, wheeled portable robots hold a huge advantage over humanoid robots in that they are lightning fast and are able to change direction even more effectively even while moving, making them extremely valuable for a few applications. Among the wheeled robots, two self-balancing ones, which are commonly known as Segway and Ninebot, have gained popularity and are used as transportation materials as well as vehicles for driving. In addition to that, self-balancing wheeled robots are also currently being used as platforms for service robots. It is well known that the typical self-balancing robot is not steady, and it will move around the rotation axis of the wheels if it is not controlled externally, and sooner or later it will fall over. If the robot's motor drives in the right direction, the robot will return to the right position it was in before. Although, the robot has many advantages over the statically stable multi-wheeled robots, it is typically unsteady. The robot must be supported by a distinctive electromechanical mechanism that allows it to balance on two wheels and stand upright. The robot is prone to falling off the vertical axis if the base on which it is supported is unstable if the platform is not balanced. This time, a gyro chip is anticipated to provide the PID controller with information on the angular position of one self-balancing robot's base. The robot must operate on any surface and is powered by two motors, one for each wheel that were constructed.

1.1. Design of Robot

Design of Robot is done in solidworks as shown in Figure 1. The parts which we have used in our project are as follows :

1. Two wooden base plates
2. Two 3D printed motor mounts.
3. Two wheels (120mm Φ)
4. Four M5 bolts and nuts
5. STM32F446 microcontroller
6. MPU6050 sensor
7. Jumper wires
8. Two stepper motors.

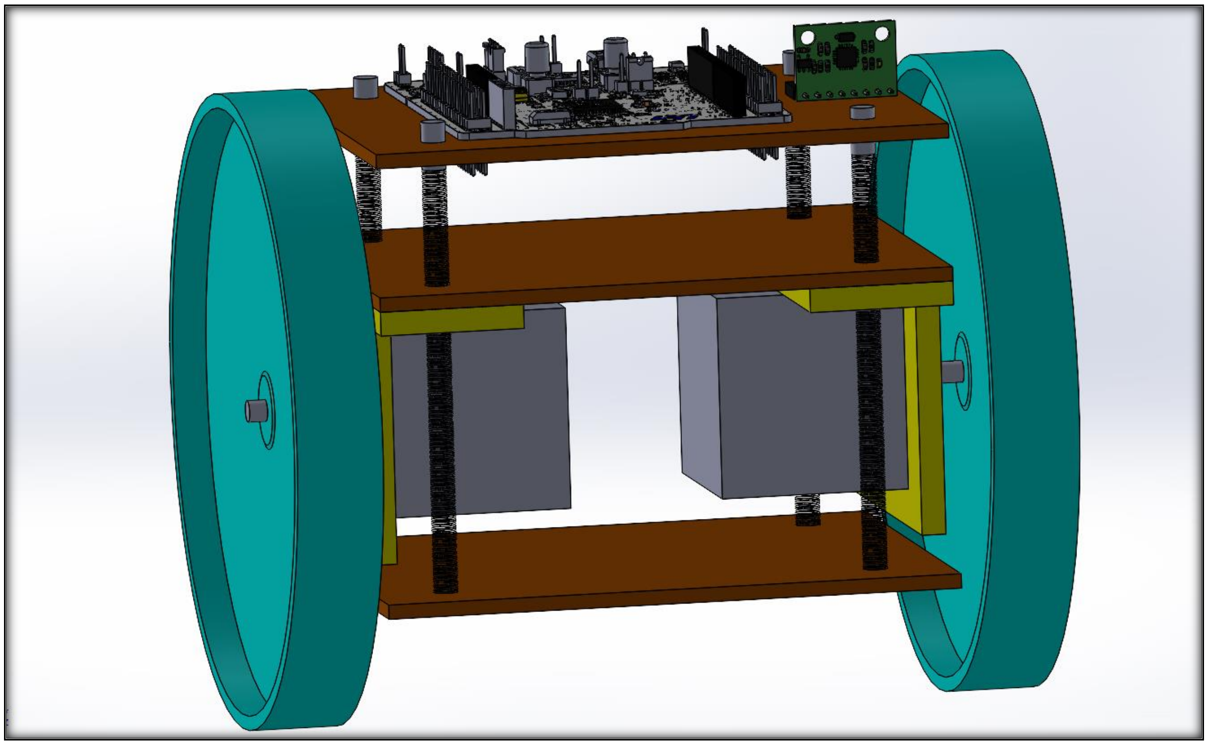


Figure 1: Design of the robot in solidworks

After designing and constructing the parts in 3D printer, the final look of robot after mounting the hardware on it can be shown in Figure 2

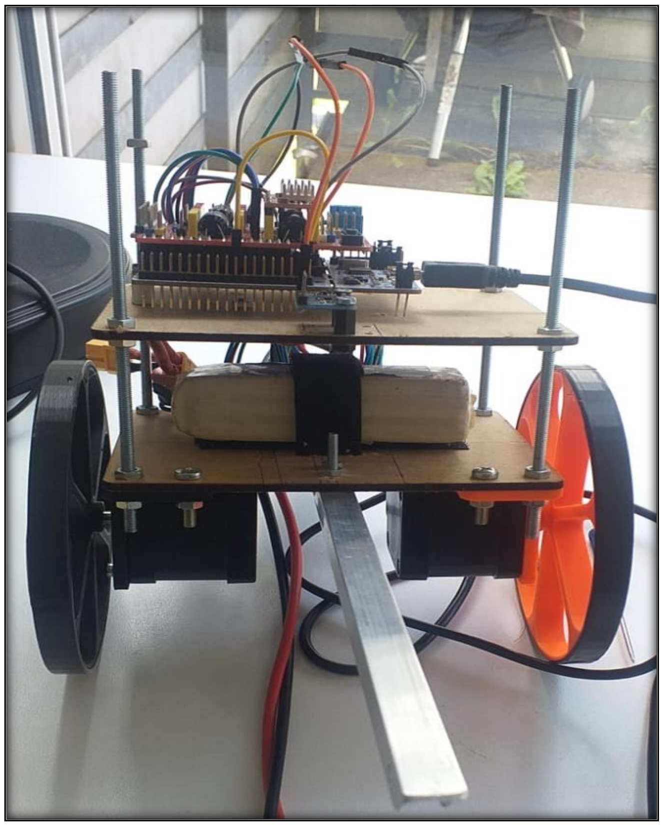


Figure 2: Final look of robot with hardware mounting on it

1.2. Mounting the robot with the Hardware:

After the design phase and the frame assembly, the microcontroller setup was mounted. The STM32F446RE provides 64 GPIO pins including Arduino pins. These Arduino pins are of special importance in this project because a middleware known as a CNC shield is used. The main purpose of using a CNC shield is to make the connection between the microcontroller and the motor drivers compact and easy.

The microcontroller along with the CNC shield is bolted to the top shelf of the robot body. A power distribution PCB is bolted to the same plate on the other side. The battery is mounted on the bottom plate.

The crucial part is to mount MPU6050. This sensor has to be placed carefully on the top shelf.

MPU6050 can be placed in different orientations and we have chosen the x-axis of the sensor to be parallel to the wheel axis of the robot. The power supply from the battery is connected to the power management chip and from there a voltage of 12V is given to the CNC shield and 5V is given to the microcontroller.

From the CNC shield, the motors are powered and controlled.

2. Components:

2.1. STM Micro-controller:

STM32 Nucleo-64 board with STM32F446RE MCU is used for the Project. Arduino UNO V3 expansion is provided inbuilt.

Table 1: Specification of STM32F446RE microcontroller

Memories	512K bytes Flash Memory, 128 Kbytes SRAM
Timers	2x watchdogs, 1x Sys Tic timer, twelve 16 bit timers and Two 32 bit timers.
Communication Interfaces	4 I2C buses, 4 SPI, 2 UARTS
Clock, Reset and supply management	1.7 to 3.6 V application supply and I/0s, 4 to 26 MHZ Crystal Oscillator.
Debug mode	SWD and JTAG interfaces

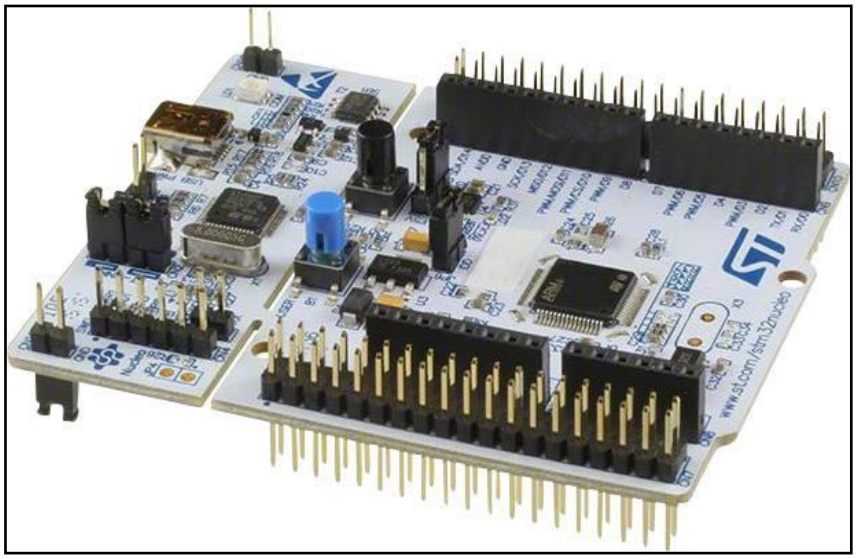


Figure 3: STM32F446RE microcontroller used in our project

2.2. Stepper Motors:

NEMA 17 high torque stepper motors are used. It provides 1.8 degree step angle (2000 steps/ revolution).

Table 2: Specification of Stepper motor Nema 17

Rated Voltage	12 V
Current	1.2 A at 4 V
No. of phases	3
Holding torque	3.2 kg-cm

Figure 4: Stepper motor NEMA 17

2.3. The CNC shield:

The CNC shield can accommodate a maximum of 4 motor drivers on it at the same time. It needs to be provided with an input voltage between 12V and 36V. It also provides pins for changing the micro stepping of the motor just by connecting a few jumpers. The CNC shield can directly sit on the Arduino pins provided in STM32F446RE. From the CNC shield, four wires from every motor driver connect to the motor. Two wires for voltage and ground and the other two for step and direction,

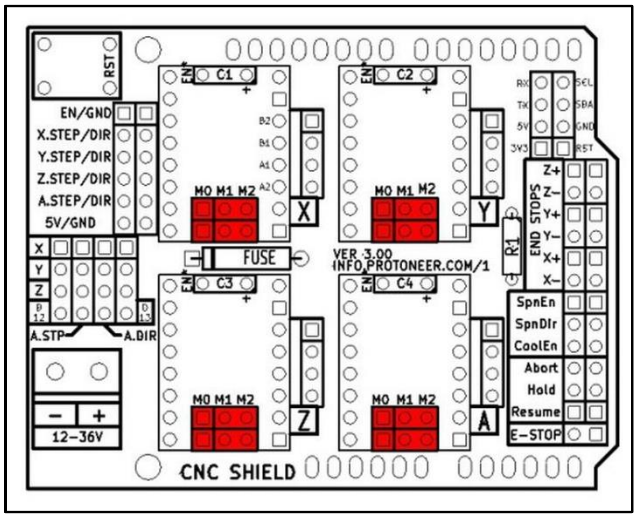
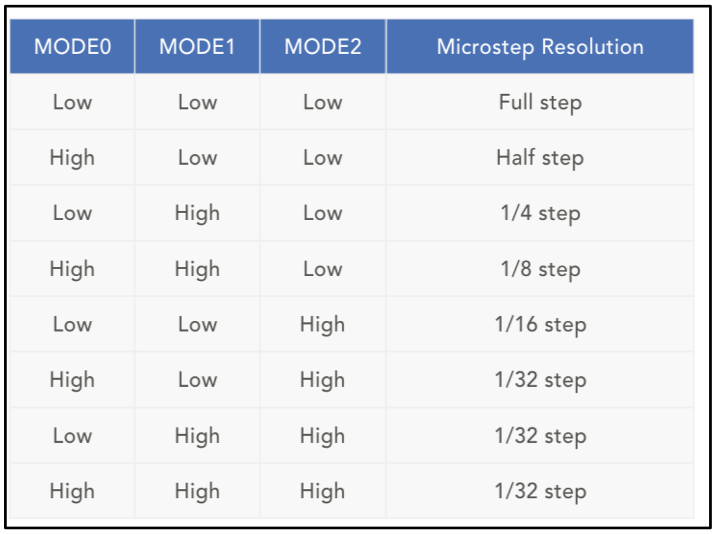


Figure 5: Schematic of CNC shield

The Figure 3 shows the schematic of the CNC shield and the pins in red color are responsible for micro-stepping. The Figure 4 shows the table of different micro-stepping options possible.



MODE0	MODE1	MODE2	Microstep Resolution
Low	Low	Low	Full step
High	Low	Low	Half step
Low	High	Low	1/4 step
High	High	Low	1/8 step
Low	Low	High	1/16 step
High	Low	High	1/32 step
Low	High	High	1/32 step
High	High	High	1/32 step

Figure 6: Micro-stepping possibilities

2.4. Motor Drivers

A4988 motor driver is used. A4988 is a complete microstepping motor with built-in translator for easy operation. It has output driver capacity of up to 35V and – 2 A to + 2 A.
Simply inputting one pulse on step input drives the motor one micro step. There are 5 selected step modes available, which are full, ½, ¼, 1/8, 1/16.

Figure 7: Motor Driver A4988

2.5. MPU6050:

MPU6050 is an integrated 6-axis motion tracking device. This is designed for the low power, low cost, and high-performance requirements of smartphones, tablets and wearable sensors. It combines a 3-axis gyroscope, a 3-axis accelerometer, and a Digital Motion Processor(DMP). It can process complex 9-axis sensor fusion algorithms using the field-proven and proprietary MotionFusion engine.

2.5.1. Properties of MPU 6050:

- It has a Tri-Axis angular rate sensor (gyro) with a sensitivity of up to 131 LSBs/dps(degree per second) and a full-scale range of ±250, ±500, ±1000, and ±2000dps
- It has a Tri-Axis accelerometer with a programmable full-scale range of ±2g, ±4g, ±8g and ±16g

- No user intervention is required for embedded algorithms for run-time bias and compass calibration in the library.
- There are additional features including an embedded digital output temperature sensor and an on-chip oscillator with ±1% variation over the operating temperature range.

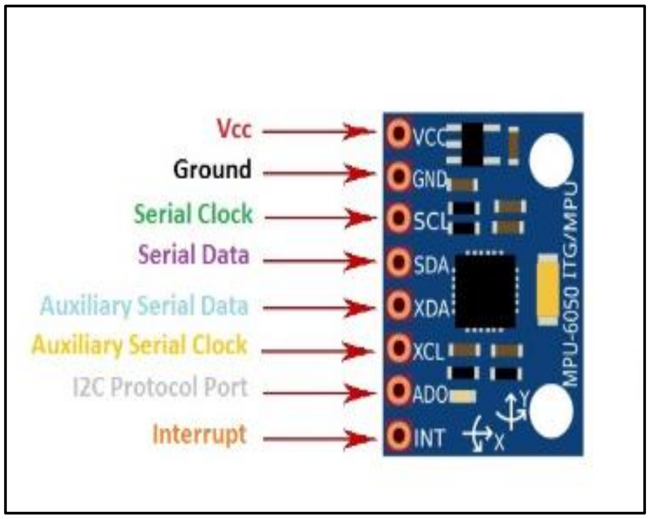


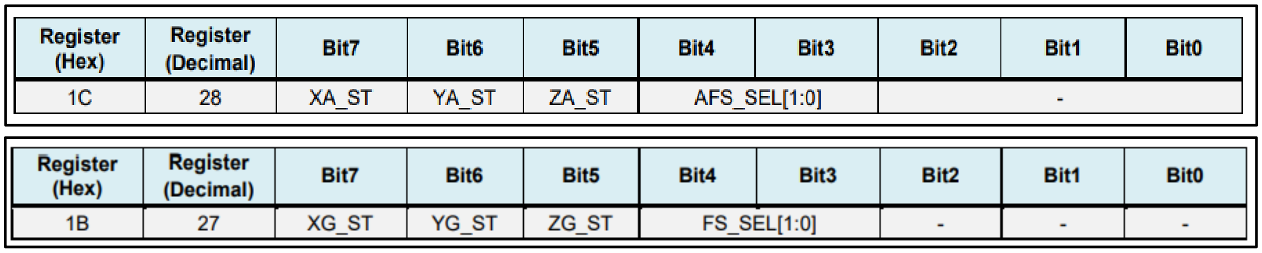
Figure 8: MPU 6050 sensor

MPU 6050 can be accessed through the I2C communication protocol. It has 8 pins as shown in the Figure 8. Vcc and Ground are used to power up the sensor and initiate all inner peripherals. SCL(Serial Clock) and SDL(Serial Data) lines are used in the aspect of I2C communication protocol. XDA (Auxiliary Serial Data) and XCL (Auxiliary Serial Clock) are used when there are other auxiliary equipment or sensor connected to MPU6050. As we are not using any other equipment, We are adhering to this description for the first 4 pins only.

Because of the digital motion processing, which relieves the microcontroller by directly doing the sensor fusion and calculating the position angles, a sensor of the MPU family was chosen. In the final prototype, out of MPU9250 and MPU6050, the latter is used because there is no need for a magnetometer for the project. You can also use MPU9250, which is associated with an in-built magnetometer and for that, you just have to change the corresponding entry in config.h. To use the digital motion processing, the accelerometer must be set to 2g and the gyroscope must be set to 250°/s.

2.5.2. Configuring MPU6050:

After connecting 4 pins as described above, we have to check if the sensor is ready. If the register WHO_AM_I (0x75) responds with 0x68 (in hexadecimal) or 104 (in decimal) value, this means it is ready and the sensor responded as desired. We used HAL_I2C_Mem_Read & HAL_I2C_Mem_Write functions to directly read from and write onto the given memory register respectively. The next step was to set the clock up to 8 MHz and wake up the sensor. This can be done by writing (0x00) to the PWR_MGMT_1(0x6B) Register. After Initializing the sensor then we proceeded to configure Acceleration and Gyroscope registers by setting the suitable values. There are two registers in MPU6050 named ACCEL_CONFIG(0x1C) and GYRO_CONFIG(0x1B) registers.

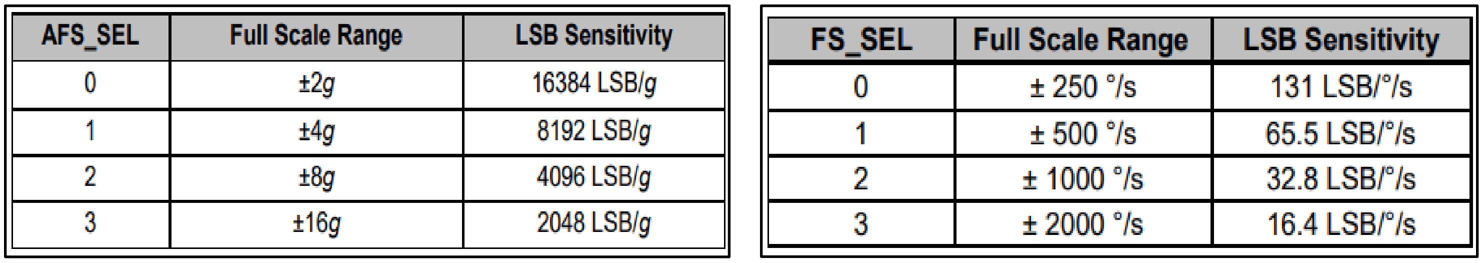


Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1C	28	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]		-		

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	-

Figure 9: MPU 6050 Configuration

FS_SEL 2-bit unsigned value. Selects the full-scale range of gyroscopes. According to MPU Datasheet, AFS_SEL selects the full-scale range of the acceleration and gyro outputs according to the following tables.



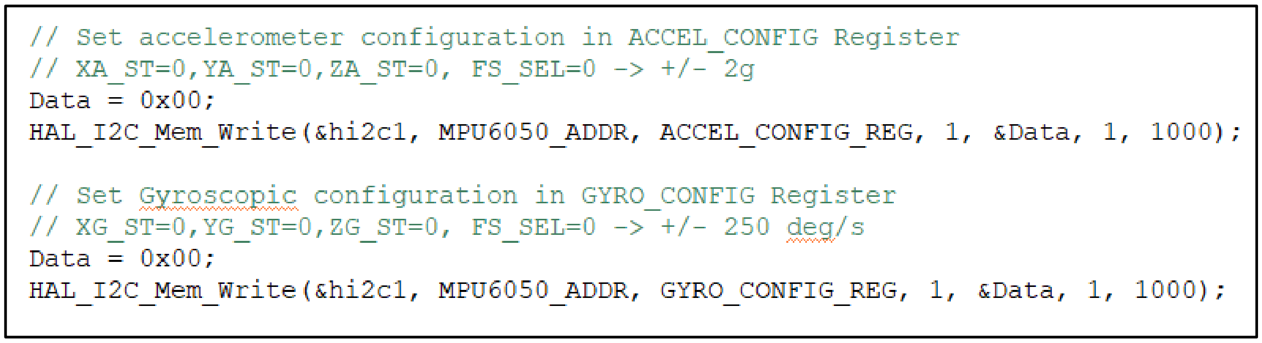
AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2*g*	16384 LSB/*g*
1	±4*g*	8192 LSB/*g*
2	±8*g*	4096 LSB/*g*
3	±16*g*	2048 LSB/*g*

FS_SEL	Full Scale Range	LSB Sensitivity
0	± 250 °/s	131 LSB/°/s
1	± 500 °/s	65.5 LSB/°/s
2	± 1000 °/s	32.8 LSB/°/s
3	± 2000 °/s	16.4 LSB/°/s

Figure 10: MPU 6050 Datasheet

± 2g and ± 250 °/s were taken as Full-Scale Range for our application in this project. So, by writing 0x00 in both ACCEL_CONFIG(0x1C) and GYRO_CONFIG(0x1B) registers, we were able to configure the sensor to the desired Full-Scale Range.

For this, we wrote a small code using the HAL_I2C_Mem_Write function.

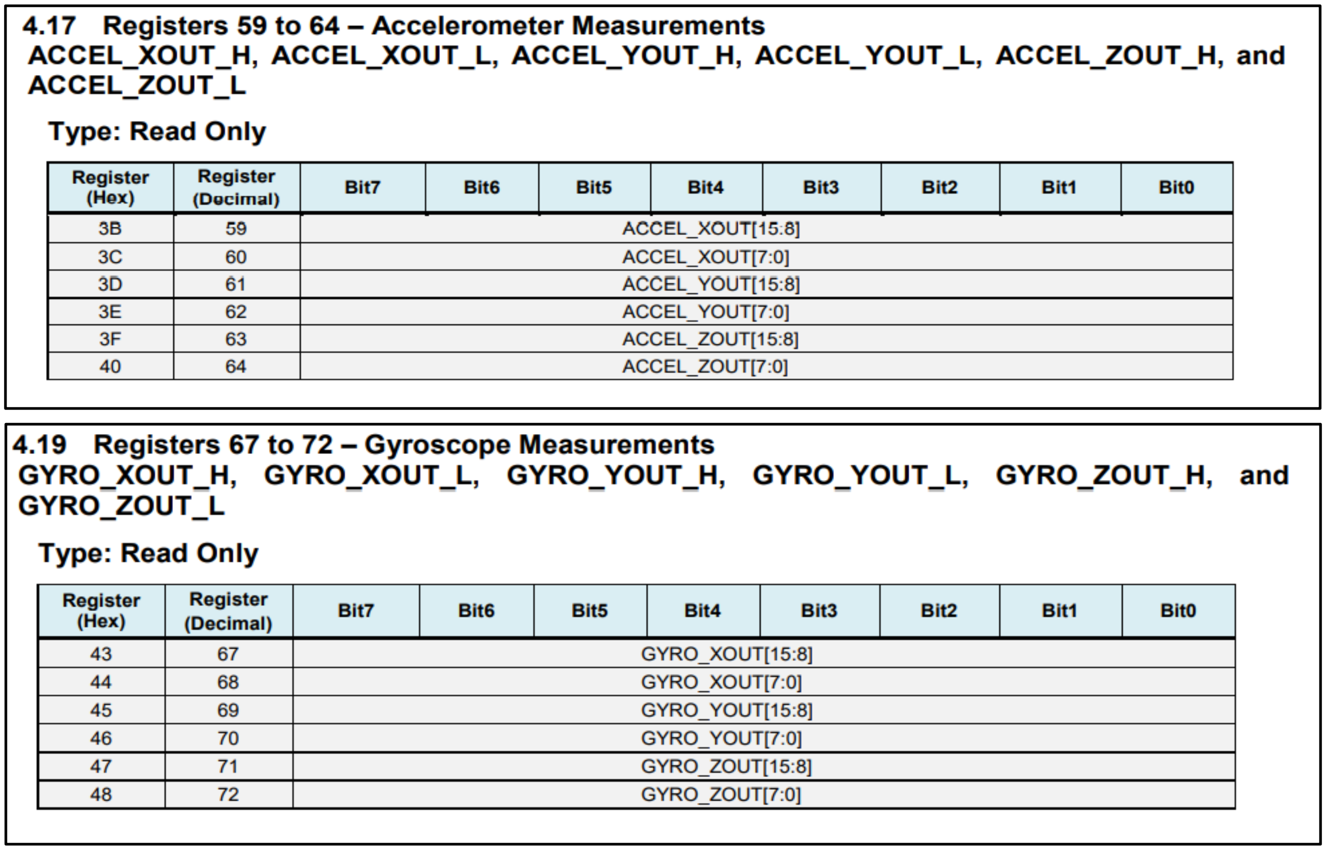


```
// Set accelerometer configuration in ACCEL_CONFIG Register
// XA_ST=0,YA_ST=0,ZA_ST=0, FS_SEL=0 -> +/- 2g
Data = 0x00;
HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, ACCEL_CONFIG_REG, 1, &Data, 1, 1000);

// Set Gyroscopic configuration in GYRO_CONFIG Register
// XG_ST=0,YG_ST=0,ZG_ST=0, FS_SEL=0 -> +/- 250 deg/s
Data = 0x00;
HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, GYRO_CONFIG_REG, 1, &Data, 1, 1000);
```

Figure 11: Coding of MPU 6050 using HAL function

After configuring the Accelerometer and Gyro registers, we collected the data using the stored values in the respective registers. Registers 59 to 64(in decimal) store the most recent accelerometer measurements. Registers 43 to 48(in decimal) store the most recent



4.17 Registers 59 to 64 – Accelerometer Measurements
ACCEL_XOUT_H, ACCEL_XOUT_L, ACCEL_YOUT_H, ACCEL_YOUT_L, ACCEL_ZOUT_H, and ACCEL_ZOUT_L

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT[7:0]							

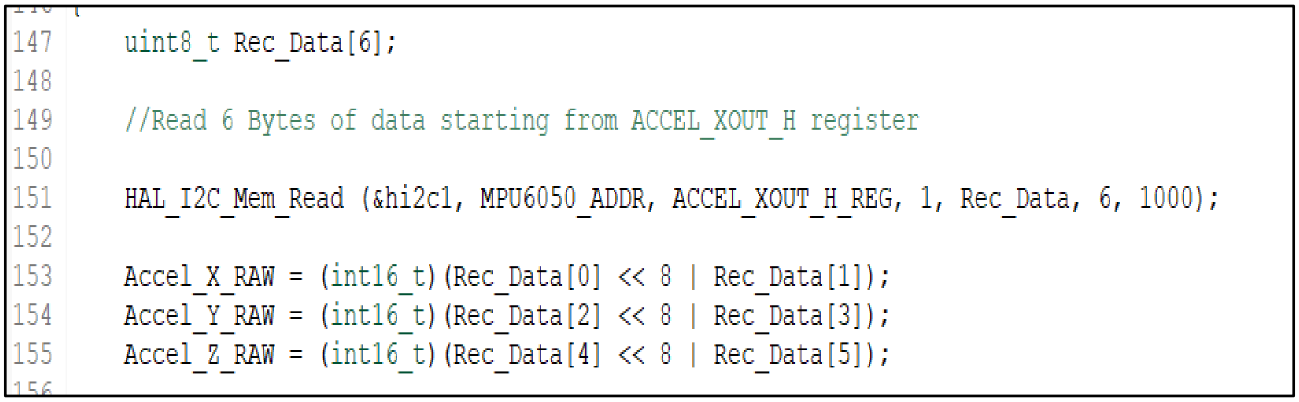
4.19 Registers 67 to 72 – Gyroscope Measurements
GYRO_XOUT_H, GYRO_XOUT_L, GYRO_YOUT_H, GYRO_YOUT_L, GYRO_ZOUT_H, and GYRO_ZOUT_L

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
43	67	GYRO_XOUT[15:8]							
44	68	GYRO_XOUT[7:0]							
45	69	GYRO_YOUT[15:8]							
46	70	GYRO_YOUT[7:0]							
47	71	GYRO_ZOUT[15:8]							
48	72	GYRO_ZOUT[7:0]							

Figure 12: MPU 6050 Accelerometer and Gyroscope data values stored in respective registers

ACCEL_XOUT is 16-bit 2's complement value and it stores the most recent X axis accelerometer measurement. Similarly ACCEL_YOUT and ACCEL_ZOUT. GYRO_XOUT is 16-bit 2's complement value and it stores the most recent X axis accelerometer measurement. Similarly GYRO_YOUT and GYRO_ZOUT.



```
    uint8_t Rec_Data[6];

    //Read 6 Bytes of data starting from ACCEL_XOUT_H register

    HAL_I2C_Mem_Read (&hi2c1, MPU6050_ADDR, ACCEL_XOUT_H_REG, 1, Rec_Data, 6, 1000);

    Accel_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data[1]);
    Accel_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data[3]);
    Accel_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data[5]);
```

Figure 13: Coding of MPU 6050 to store Accelerometer and Gyroscope data values

We read the Raw acceleration values by creating the Array Rec_Data[6], one for each register. Thus, at first, storing the value from register 59 and shifting them by 8 bits and adding the value from register 60 at the end gives the total raw acceleration value in X direction. This was followed similarly in Y and Z direction. The same logic was used for obtaining the raw gyro values in three directions out of 6 registers. Later, these raw values are scaled down to real values by dividing acceleration values with Sensitivity values. The LSB sensitivity for ± 2g is 16384 LSB/g and for ± 250 °/s is 131 LSB/°/s. Then, we subtracted the offset values from all the six values. These offset values are calculated by keeping the sensors in rest position and in upright position.

2.5.3. Measuring Angle of Inclination Using Accelerometer:

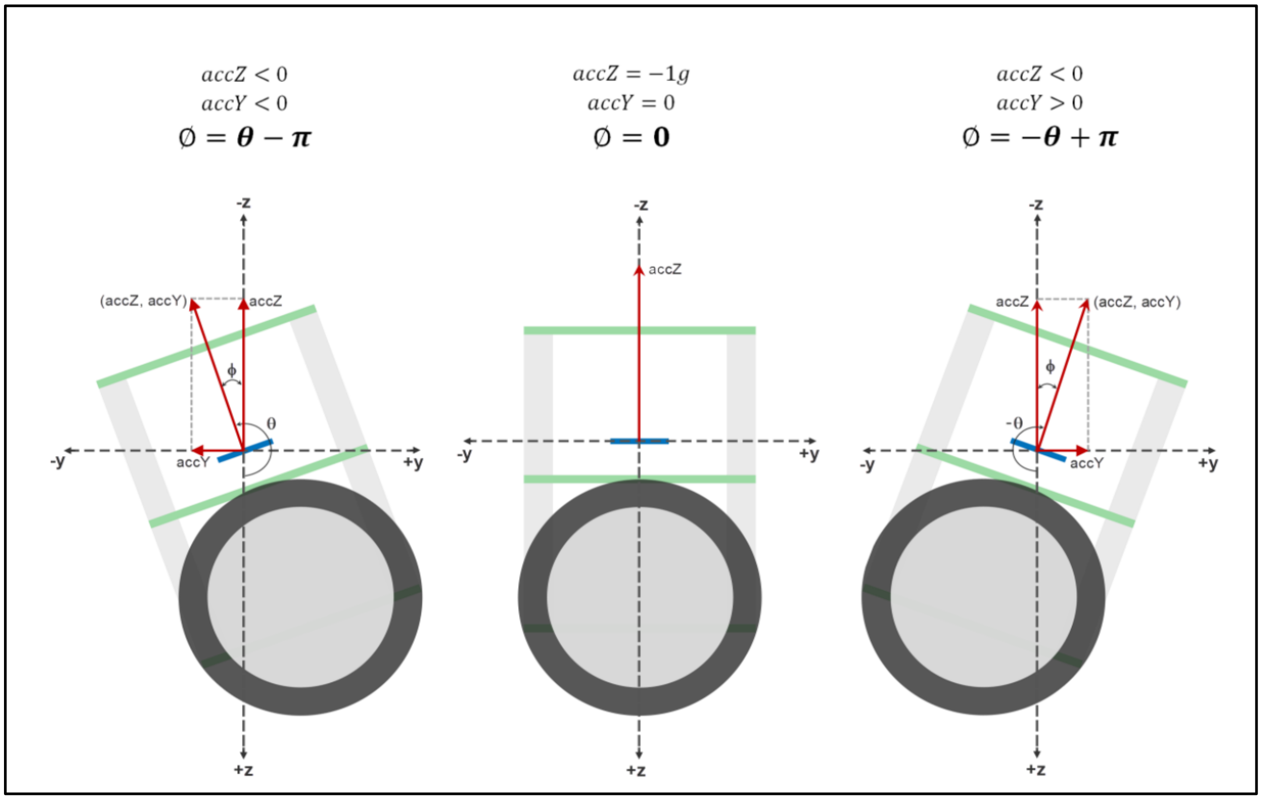


Figure 14: Illustration of angle of inclination measurement

The MPU6050 has a 3-axis accelerometer and a 3-axis gyroscope. The accelerometer measures acceleration along the three axes and the gyroscope measures angular rate about the three axes. To measure the angle of inclination of the robot, we needed acceleration values along y and z-axes. The atan2(y,z) function gives the angle in radians between the positive z-axis of a plane and the point given by the coordinates (z,y) on that plane, with positive sign for counter-clockwise angles (right half-plane, y > 0), and negative sign for clockwise angles (left half-plane, y < 0). However, when we tried moving the robot forward and backward while keeping it tilted at some fixed angle. We observed that the angle shown suddenly changes. This is due to the horizontal component of acceleration interfering with the acceleration values of y and z-axes.

2.5.4. Measuring Angle of Inclination Using Gyroscope:

The 3-axis gyroscope of MPU6050 measures angular rate (rotational velocity) along the three axes. For our self-balancing robot, the angular velocity along the x-axis alone is sufficient to measure the rate of fall of the robot.

In the coding part, we read the gyro value about the x-axis, convert it to degrees per second and then multiply it with the loop time to obtain the change in angle. The position of the MPU6050 when the program starts running was the zero inclination point. The angle of inclination was measured with respect to this point.

We observed that the angle was gradually increasing or decreasing when the robot was kept steadily at a fixed angle. It was not staying steady. This was due to the drift which is inherent to the gyroscope.

In the coding part, loop time is calculated using the HAL_GetTick() function which is built into the STM32CubeIDE. In later steps, we will be using timer interrupts to create precise sampling intervals. This sampling period will also be used in generating the output using a PID controller.

2.5.5. Combining the Results with a Complementary Filter:

We have two measurements of the angle from two different sources. The measurement from accelerometer gets affected by sudden horizontal movements and the measurement from gyroscope gradually drifts away from actual value. In other words, the accelerometer reading gets affected by short duration signals and the gyroscope reading by long duration signals. These readings are, in a way, complementary to each other. Combine them both using a Complementary Filter and we get a stable, accurate measurement of the angle. The complementary filter is essentially a high pass filter acting on the gyroscope and a low pass filter acting on the accelerometer to filter out the drift and noise from the measurement.

```
currentAngle = 0.95*(currentAngle + gyroAngle) + 0.05*(accAngle);
```

The low pass filter allows any signal longer than this duration to pass through it and the high pass filter allows any signal shorter than this duration to pass through. The response of the filter can be tweaked by picking the correct time constant. Lowering the time constant will allow more horizontal acceleration to pass through.

3.Methodology:

The motors were tested individually to check the working behavior. The stepper motors are connected to the motor drivers through the CNC shield. The bipolar stepper motor works when the poles are shifted from one step to another with some time delay.The speed of the motor is inversely proportional to the time delay between the steps.

NEMA 17 takes 200 steps to complete one revolution. The rpm of the motor under a safety factor can go to 600. In order to control a motor, two pins are needed. One pin is called a step pin and another pin is called a direction pin. By setting the direction pin to high or low, the motor rotates clockwise or anti-clockwise.

As the CNC shield is involved, there are fixed pins for steps and directions that should be initialized. For this, a pinout configuration of the CNC shield on Arduino Uno is compared to Arduino pins on STM32F446RE. The pins that correspond to the X and Z configuration on the shield are:

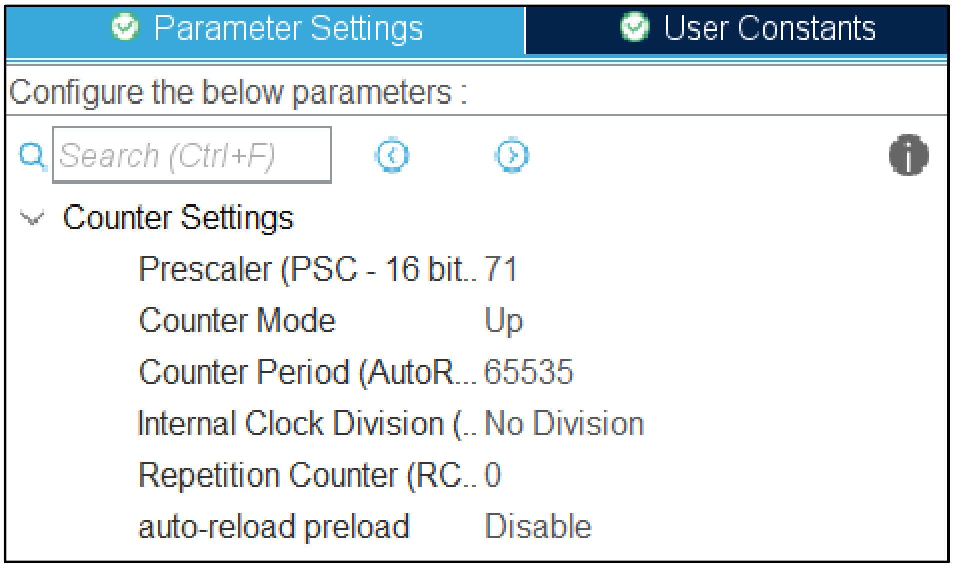
STEP-X - PA10

STEP-Z - PB5

DIRECTION-X - PB4

DIRECTION-Z - PA8

Now in the STM32CubeIDE in the IOC tab, initialize all the above-mentioned pins as GPIO output. Then in the clock configuration select HSE and set the HCLK frequency to 72MHz. Then initialize TIM1 and set the clock source to the internal clock and set the following parameters as in the Figure 15.



Configure the below parameters :

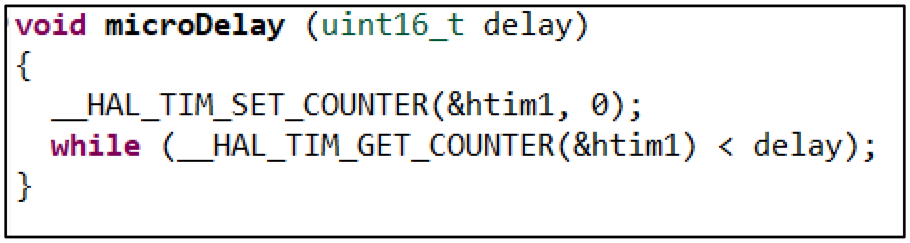
Counter Settings

Prescaler (PSC - 16 bit..	71
Counter Mode	Up
Counter Period (AutoR...	65535
Internal Clock Division (..	No Division
Repetition Counter (RC..	0
auto-reload preload	Disable

Figure 15: Parameter settings in CubeIDE

Here the Prescaler is set to 71 and the output frequency is 1MHz and the Counter period is set to the maximum value of 16-Bit. Timers play an important role in generating delays that are required to run the motor.

In the main loop, the clock has to be started by using HAL_TIM_Base_Start(&htim1). Then using HAL_GPIO_WritePin(DIR_PORT_O, DIR_PIN_O, GPIO_PIN_SET) set or reset the direction pin in the required sense of rotation. Then toggle the step pin on and off with a delay of a few microseconds using a Delay function as shown in the Figure 16.



```
void microDelay (uint16_t delay)
{
  __HAL_TIM_SET_COUNTER(&htim1, 0);
  while (__HAL_TIM_GET_COUNTER(&htim1) < delay);
}
```

Figure 16: Main loop by using HAL function

The step delay for the same velocity changes for different micro-stepping. In this project, 1/16 micro-stepping is used due to which it takes 3200 steps to complete one rotation.

3.1. Self-balancing Methodology:

All the codes from the inertial sensor, the motors, and PID are assembled together in the main.c file and called in the main function. In the while loop the following processes take place:

1. Reading the Acceleration and Gyro values from MPU registers.
2. Calculation of the current angle in deg using complementary function.
3. PID calculation.
4. Running the motors.

After debugging this code into the microcontroller, the performance of the motor was not as expected.

3.2. MPU 6050 Methodology:

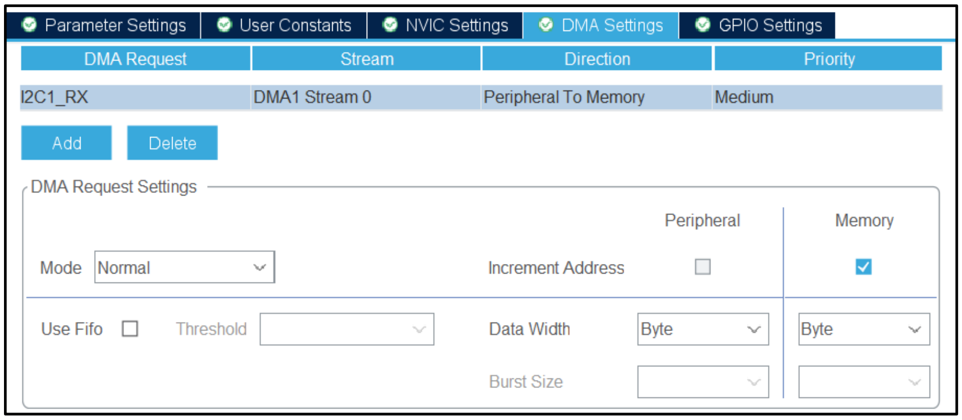
MPU6050 is used for obtaining gyro values and acceleration values which are again converted to some special orientation value. To obtain these values, we need to configure the sensor first. We use STM32CubeIDE for configuring the sensor. As we already mentioned, we are going to work on 4 pins on the sensor, Vcc, Ground, SCL and SDA pins. Vcc has to be connected to 3.3V and the Ground is connected to the Ground of STM32, then SCL(Serial Clock Pin) and SDA(Serial Data Pin) are connected to respective SCL and SDA pins in MCU. In STM32F446, PB6 pin and PB7 pin are configured as SCL and SDA respectively by default.

3.3. Code without PID:

To debug the problem, we tried working with the simpler code first. The PID code was removed and the in the while loop we only used MPU code and motor code. Still, the motor did not work as we expected. The motor was revolving slower than expected. It was found out that the load on the microcontroller was more due to calling the MPU register and some floating point calculations, that are involved in the while loop. To tackle this we have implemented Direct Acess Memory(DMA) in the code.

3.4. Direct Access Memory (DMA):

DMA allows the device to transfer the data directly to/from memory without any interference from the CPU. Using a DMA controller, the device requests the CPU to hold its data, address, and control bus, so the device is free to transfer data directly to/from memory. Here, the DMA can be used to transfer the data from the I2C peripheral to memory without the interference of the CPU. Here, under the I2C1 tab add a DMA request as shown in the Figure 17.



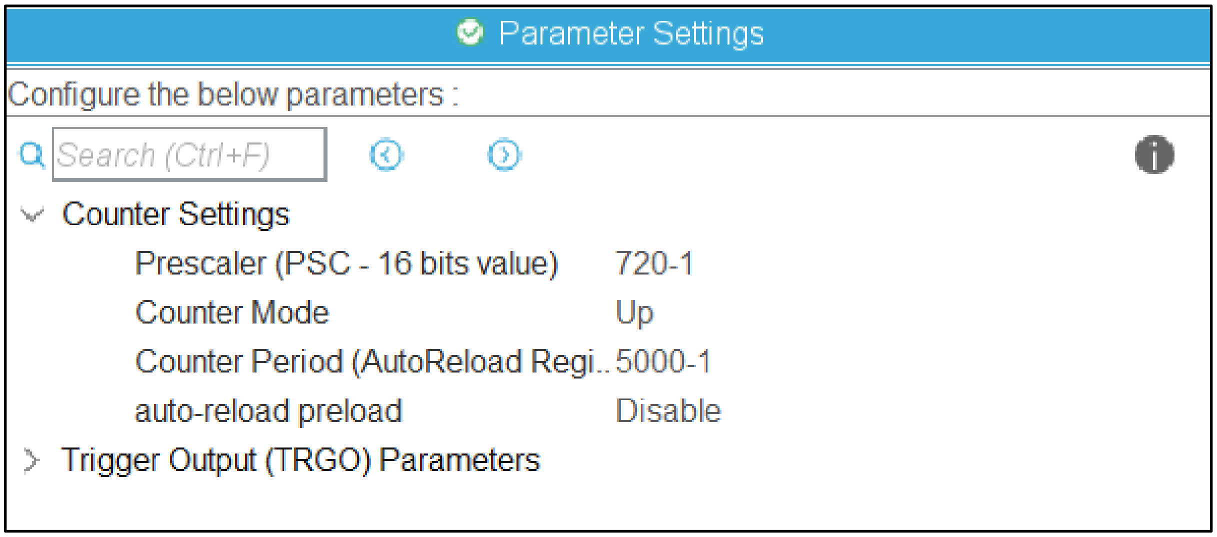
DMA Request	Stream	Direction	Priority
I2C1_RX	DMA1 Stream 0	Peripheral To Memory	Medium

		Peripheral	Memory
Mode: Normal	Increment Address	☐	☑
Use Fifo ☐ Threshold	Data Width	Byte	Byte
	Burst Size		

Figure 17: DMA settings in CubeIDE

An Interrupt Service routine (ISR) should be implemented with a certain period in order to read those values periodically.

For this, a separate timer is enabled in the .ioc file. Using the proper Prescaler and counter period, we have set the interrupt timer to 5 ms. Therefore every 5ms the ISR is called and the values are updated. The settings are done as shown in the Figure 18.

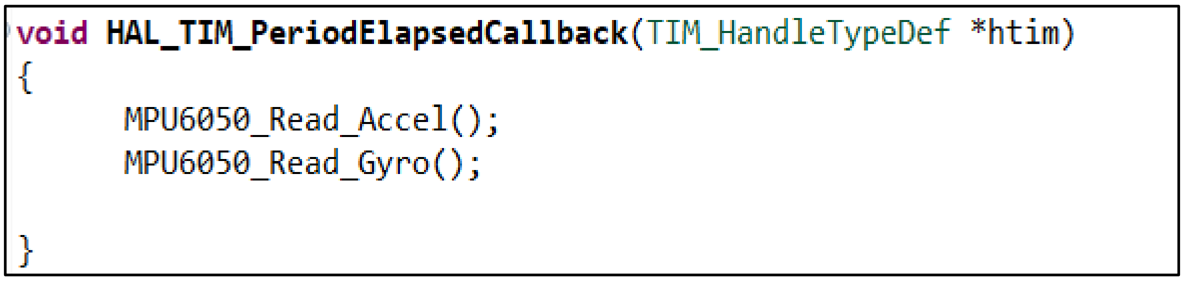


```
Counter Settings
    Prescaler (PSC - 16 bits value)       720-1
    Counter Mode                          Up
    Counter Period (AutoReload Regi..     5000-1
    auto-reload preload                   Disable
Trigger Output (TRGO) Parameters
```

Figure 18: Prescaler and counter period settings for desire Interrupt timer in CubeIDE

The NVIC should also be enabled in order to make the ISR function.

To start an interrupt timer use HAL_TIM_Base_Start_IT(&htim7) and a callback function has to be written outside the main function as shown in the Figure 19.



```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
	MPU6050_Read_Accel();
	MPU6050_Read_Gyro();

}
```

Figure 19: Callback function outside the main function

Inside the callback function, two functions have been used that call the Acceleration and Gyro values from I2C peripherals.

The rest of the code is put in the while loop. After debugging the code into the microcontroller the code worked as expected. It was initially giving the current angle value perfectly, but when the power to the motors was given, the motor rotates for a while and suddenly the program used got stuck. The behavior was random.

3.5. PID Control:

The forward and backward movement of the robot is given with respect to the tilt of robot. If the robot tilts forward, The angle from IMU will be a +ve value and the velocity is given in forward direction and Vice versa.

The velocity of robot is controlled using PID controller. In this method, The speed is fed in to the motors by considering 3 factors, which are:

- Error between measured angle and set angle.
- The accumulation of error over time.
- The rate of change of that error.

The PID algorithm is made using these 3 variables.

$$Error\_P \;=\; setpoint - Pitch$$

Here, Setpoint = 0 and Pitch is our Current angle

$$Error\_I \mathrel{+}= Error\_P$$

Here Error_P is iterated to give Error_P.

$$\text{Error\_D} \;=\; \text{Error\_p} - \text{Prev Error\_P} \quad \text{(Rate of change of Error\_P)}$$

PID algorithm, It provides output signal as angular velocity.

$$PID = Kp * Error_{\_P} + Ki * Error\_I + Kd * Error\_D$$

Every Robot has its Unique Kp, Ki, Kd values. The values need to be tuned while Balancing the Robot. The value Obtained from PID will be in Radians/sec. The signal to the stepper motor will be in the form of Step delay. So, the rad/sec is converted to Steps/sec.

$$step\ delay = \left(\frac{Velocity}{2*pi}\right) * Steps\_per\_rev$$

Rad/sec is converted to rev/sec by dividing it with 360. The stepper motor used provides 2000 steps/rev. the rev/sec is multiplied by 2000 for the required steps/sec.
The step delay is feeded in to the stepper motor using Microdelay function

3.6. DWT Cycle Counter:

DWT is an inbuilt cycle counter which counts at the clock frequency of the micro-controller. When enabled, this counter counts the number of core cycles, except when the core is halted.

Applications and debuggers can use the counter to measure elapsed execution time. By subtracting a start and an end time, an application can measure the time between in-core clocks (other than when Halted in debugging).
To access the DWT cycle counter the pointer should be assigned with the address of the register. The address of the register is 0xE0001004. Then another register with address 0XE0001000 should be assigned with the value equal to 1. This register enables the counter. The counter register should be set to 0. To check the execution cycles for the program, we need the pointer value after that specific line of code that we want to check.

In the case of the robot program, the execution time for both angle calculations and MPU calling functions is taking around 30,930 cycles. The execution time for the calculation is around 3,728 cycles. So, it is inferred that the MPU calling functions are taking an ample amount of time to get executed. Because of this huge number of cycles, there is a delay which is the reason for not getting the proper signal in an appropriate manner which results in a decrease in the velocity of the robot.

To tackle this issue we have implemented DMA and the result was a little bit better, however, the step delay signal to the motors is still not what we expected

The calculation below shows the time taken for each part of the code in the main loop:

Number of cycles before application of DMA = 30500 cycles.

Number of cycles after application of DMA = 3400 cycles (without motor code)

CPU clock frequency = 72MHz.

Time taken for 3400 cycles = $\frac{3400}{72*10^6}$ = 48* $10^{-6}s$ = 48 microseconds.

Delay for motor= 50 microseconds.

4. Details of Hardware Implementation:

Figure 20 shows the overall strategy in the form of flowchart to easily understand the execution of project.

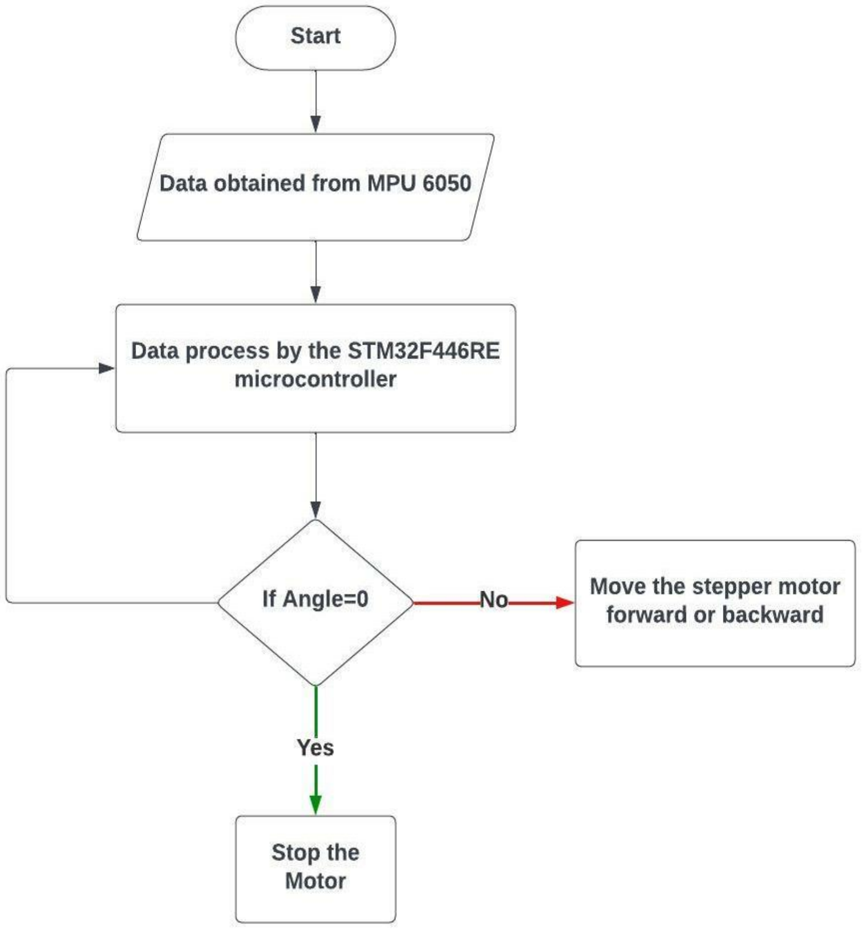

Start
Data obtained from MPU 6050
Data process by the STM32F446RE microcontroller
If Angle=0
No
Move the stepper motor forward or backward
Yes
Stop the Motor

Figure 20: Flowchart of the project

5. Result:

The logic was implemented to balance the robot. The MPU6050 and the motor codes were properly implemented to run. However the following issues mentioned below occurred which hindered the process of self-balancing:

1. Hardware issues with the MPU connectivity and sensitivity.
2. A very low step-delay(in micro seconds) to the motor made the issue of main loop time consumption very sensitive.

After the DWT calculations, we have found that number of cycles that is taken for the angle calculations is around 50 micro-seconds and which was not allowed. This delay caused an unsmooth rotation of the motor. This jerking behavior impacted the I2C values received from the MPU 6050 producing incorrect values.

We have tried decreasing the micro stepping and increasing the delay value . We thought a huge delay will help in decreasing the impact of the time taken for calculations, but that did not help. We have set the microcontroller to its maximum working clock frequency, still the output was not smooth.

It seemed that the only way to mitigate this effect would be removing the calculation part from the while loop.

REFERENCES

1. https://controllerstech.com/interface-stepper-motor-with-stm32/ -reference for stepper motor working
2. https://www.st.com/content/ccc/resource/training/technical/product_training/group0/10/5f/2c/5e/70/3e/49/8c/STM32G0-System-Direct-memory-access-controller-DMA/files/STM32G0-System-Direct-memory-access-controller-DMA.pdf/jcr:content/translations/en.STM32G0-System-Direct-memory-access-controller-DMA.pdf -DMA reference
3. https://controllerstech.com/stm32-i2c-configuration-using-registers/ - Reference for I2C communication protocol
4. https://www.instructables.com/Arduino-Self-Balancing-Robot-1/ - Refence for Complimentary function