Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Electronics Engineering (SENSE)

B. Tech – Electronics & Communications Engineering

BECE403E – EMBEDDED SYSTEM DESIGN

PROJECT REPORT

Submitted By

22BEC1204 –Shwetank Shekhar

22BEC1459 – Subhojeet Sural

22BEC1467 –Arnav Sharma

22BEC1410 – Sneha Sharma

Submitted To

Dr. Sofana Reka S

DATE: 15/04/2025

TABLE OF CONTENTS

INTRODUCTION

➢ OBJECTIVES AND GOALS

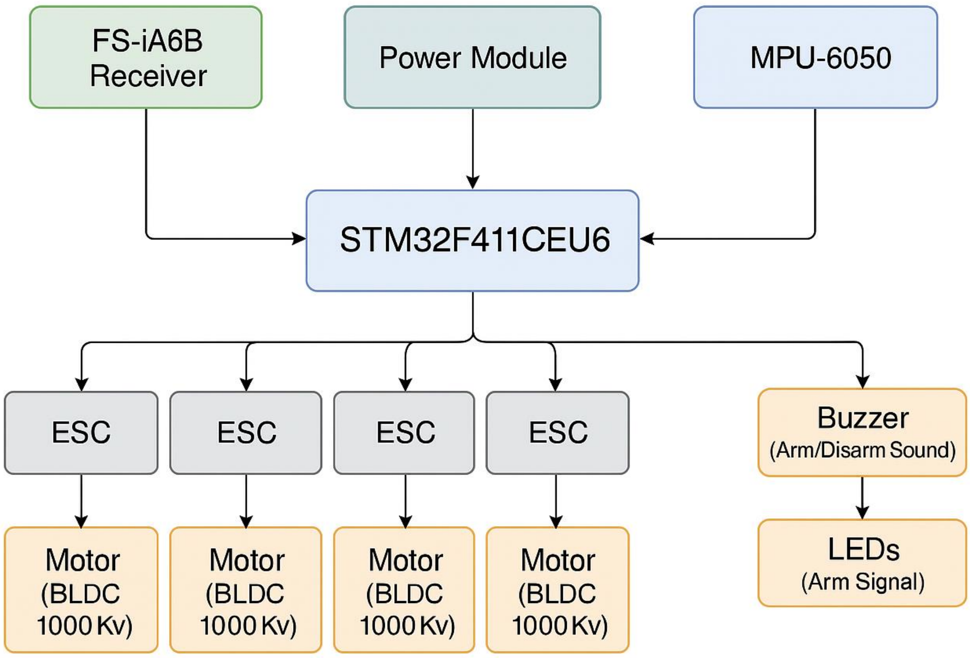
- **Development of functional flight controller on STM32F411CEU6 board:**
  The primary objective of the project is to develop a flight controller which is capable of stabilizing a quadcopter by integration of STM32F411CEU6 board with other sensors and actuators.
- **Exploration of several communication protocols:**
  The microcontroller board has been integrated with several sensors via different communication protocols like I2C and SPI. The setup of receiver and transmitter allows the pilot to send direct commands to the flight controller via radio receiver.
- **To achieve a responsive and stable flight controller:**
  Tuning the PID controllers for optimal performance for the quadcopter which minimize oscillations and achieve predictable responses.

➢ FEATURES:

- **PWM motor Control:**
  Generating precise PWM signals to control the ESCs which send the PWM signals to the motors to have and accurate motor speed control.
- **Telemetry using Arduino Nano:**
  The flight data including the PWM signal values that is sent to the motors and other data like PID values is sent to the ground station and can be seen on the serial terminal through Arduino Nano.
- **Arming and disarming check:**
  Output devices like buzzer and led are integrated with main microcontroller to provide a visual and audio check when the drone gets ready to arm after performing the PID tuning and calibrations.
- **Integration of IMU sensor and filter:**
  IMU sensor provides the readings from accelerometer and gyroscope which are sent to a comparative filter to compare the obtained values with the desired values to provide standard values which can be sent for PID tuning.
- **Modular software design:**
  The code implementation is done using mbed OS 6 along with different reusable functions for each component which makes the design modular for easy debugging.

DESIGN AND IMPLEMENTATION

BLOCK DIAGRAM:



PIN DETAILS:

S.No	Function	Pin	Description
1	USB Serial TX	PA_2	Transmits debug/status messages via UART (to USB)
2	USB Serial RX	PA_3	Receives data if required (used in BufferedSerial)
3	I2C SDA	PB_9	Connected to MPU6050 SDA line for IMU data
4	I2C SCL	PB_8	Connected to MPU6050 SCL line
5	Motor 1 PWM	PB_13	Output PWM signal to ESC 1 (Front Left motor)
6	Motor 2 PWM	PA_10	Output PWM signal to ESC 2 (Front Right motor)
7	Motor 3 PWM	PB_6	Output PWM signal to ESC 3 (Rear Right motor)
8	Motor 4 PWM	PB_5	Output PWM signal to ESC 4 (Rear Left motor)
9	LED Output	PC_13	On-board status LED, used for initialization signal
10	PPM Input	PA_0	Connected to FS-iA6B receiver for reading channels
11	Buzzer Output	PA_15	Drives active buzzer for sound alerts

HARDWARE ANALYSIS

➢ FRAME: F450

F450 is a quadcopter frame with a wheelbase i.e. the distance between opposite motor centres, of approximately around 450mm. It includes an integrated power distribution board to ensure that each motor is provided with same power.

➢ MOTORS: A2212

A2212 motor with 1000kV is used which is a outrunner brushless DC motor which means that is takes 1000 revolutions per minute per volt and is used for standard minimal quadcopters.

➢ ESC: 30A

Electronic speed controller for 30A rating is used for the 1000kV motors which is used to regulate the speed of BLDC motors in quadcopter by varying the PWM signal provided to the motors. It converts the control PWM signal to three phase power required to drive the motor.

➢ BATTERY: 3S LiPo

A 3S battery is sued with overall voltage of 11.1V where 3.7V is per cell voltage with a capacity of 2200mAh which means that 2.2A per hours. It is connected to the power distribution board of the frame and power is equally distributed among the 4 BLDC motors.

➢ RECEIVER and TRANSMITTER:

A 6- channel receiver is used to receive radio signals from a compatible transmitter where the user gives the control signals for the quadcopter. The operating frequency is the standard frequency of 2.4GHz which utilises the technique of automatic frequency hopping.

The transmitter provides PWM outputs which are received by the RX and are further sent to flight controller.

➢ MICRO-CONTROLLER BOARD: STM32F411CEU6

It is an ARM- Cortex M4 32- bit microcontroller which includes 128Kb of SRAM and 512Kb flash memory. It supports communication protocols like SPI, I2C, UART and USB with a C-type interface for both power and communication.

➢ TELEMETRY: ARDUINO NANO

The Arduino Nano is utilised as a telemetry to display the data serially on the ground station which is gathered by the sensors like IMU and the PWM values sent to each motor. It is connected to the STM board via UART protocol.

➢ IMU SENSOR

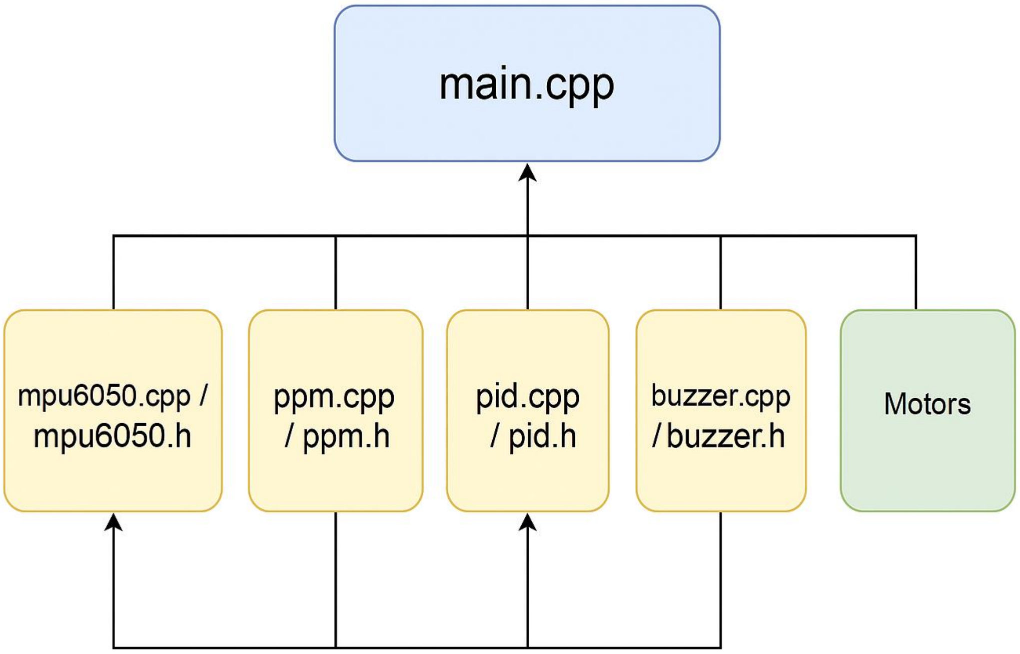
IMU stands for inertial measurement unit which consists of accelerometer and gyroscope which provided the values of motion and angular velocity of the quadcopter in three axes to the STM board which is interfaced via I2C protocol.

SOFTWARE ANALYSIS

The software architecture of the custom flight controller is modular and designed for clarity, safety, and real-time responsiveness. It is built using C++ on the Mbed OS 6 framework, which provides essential abstractions for multitasking and hardware interfaces.

At its core, the system is driven by a main control loop running at a fixed interval, where sensor data is read, user inputs are processed, and motor outputs are updated. The architecture includes dedicated modules for each functional block: a PPM input handler decodes control signals from the FS-i6AB receiver, mapping them to throttle, pitch, roll, yaw, and arm switch channels; the IMU driver communicates with the MPU6050 over I2C, performs initial calibration, and computes orientation using the Mahony filter for accurate real-time attitude estimation.

The PID controller module uses the orientation data to stabilize the drone, with tuneable proportional, integral, and derivative gains for each axis. A motor mixing logic block computes motor outputs based on the combined effect of base throttle and PID corrections. Safety is ensured via an arming mechanism on channel 5, minimum throttle constraints, and override logic when pitch and roll sticks are in the top-right corner. An additional buzzer module provides disarmed feedback through a melody played via a PWM-driven buzzer. All components work together seamlessly in the main loop, offering a responsive, stable, and customizable flight control system.



- **API's USED:**

API	USE CASE	FUNCTIONS
I2C	To integrate MPU6050	I2C:: frequency()
		I2C::write()
		I2C::resd()
DigitalOut	To integrate LEDs	
PwmOut	To send the signal to motors	Period_us()
		Write()
		Period()
		Pulsewidth_us()
BufferedSerial	Serial communication	Write()
Timer	Timing Operations	Start()
		Reset()
		Elapsed_time()
		Sleep_for()
InterruptIn	PPM signal for receiver	Rise()

- **CUSTOM FUNCTIONS:**

FUNCTIONS	USE CASE
MPU6050	IMU Sensor
PID	PID Tuning
Buzzer	Pwm signal to buzzer
PPM Module	Signal received by RX

➢ MAIN PROGRAM:

This is the central file that orchestrates all the modules of the flight controller. It initializes hardware components like the MPU6050 IMU, PPM receiver, PID controllers, and motors. The main loop handles sensor updates, receiver input normalization, PID computation, motor mixing, and safety checks like arming and stick overrides. It also integrates the buzzer trigger when the drone is disarmed. This custom flight controller successfully demonstrates stable flight behavior with responsive PID correction, safe arming logic, and user feedback via a startup buzzer melody. It is designed with modularity, expandability, and safety in mind, making it suitable for further experimentation

```
#include "mbed.h"
#include "mpu6050.h"
#include "pid.h"
#include "ppm.h"
#include "buzzer.h"

using namespace ThisThread;

BufferedSerial usb(PA_2, PA_3, 115200);
I2C i2c(PB_9, PB_8);
MPU6050 mpu(i2c);
DigitalOut Led(PC_13);
char buffer[128];


PwmOut motor1(PB_13); // Front Left
PwmOut motor2(PA_10); // Front Right
PwmOut motor3(PB_6);  // Rear Right
PwmOut motor4(PB_5);  // Rear Left

Buzzer buzzer(PA_15);

float pitch, roll, yaw;
float base_throttle = 0.0f;
float desired_pitch = 0.0f, desired_roll = 0.0f, desired_yaw =
0.0f;

float dt = 0.01f;
PID pid_pitch(0.6f, 0.003f, 0.15f, dt);
PID pid_roll(0.6f, 0.003f, 0.15f, dt);
PID pid_yaw(0.4f, 0.001f, 0.1f, dt);


float apply_deadband(float input, float deadband = 0.025f) {
    return (fabs(input) < deadband) ? 0.0f : input;
}

int main() {
    i2c.frequency(400000);
    usb.write("Initializing MPU6050...\n", 26);
    Led = 1;
```

```
    if (!mpu.initialize()) {
        usb.write("MPU6050 initialization failed!\n", 33);
        Led = 0;
        return 1;
    }

    usb.write("Calibrating sensors...\n", 24);
    mpu.calibrate(1000);
    usb.write("Calibration done.\n", 19);

    ppm_init();

    motor1.period_us(2000);
    motor2.period_us(2000);
    motor3.period_us(2000);
    motor4.period_us(2000);


    motor1.pulsewidth_us(1000);
    motor2.pulsewidth_us(1000);
    motor3.pulsewidth_us(1000);
    motor4.pulsewidth_us(1000);
    sleep_for(2s);

    pid_pitch.reset();
    pid_roll.reset();
    pid_yaw.reset();

    Timer loop_timer;
    loop_timer.start();


    pid_pitch.setTunings(0.33f, 0.003f, 0.06f);
    pid_roll.setTunings(0.31f, 0.003f, 0.07f);
    pid_yaw.setTunings(0.3f, 0.001f, 0.08f);

    pid_pitch.setOutputLimits(-0.10f, 0.10f);
    pid_roll.setOutputLimits(-0.9f, 0.9f);
    pid_yaw.setOutputLimits(-0.04f, 0.06f);
    float pitch_trim = -0.08f;

    while (true) {
        mpu.updateMahony();
        mpu.getOrientation(&pitch, &roll, &yaw);

        int roll_raw     = ppm_read(0); // CH1
        int pitch_raw    = ppm_read(1); // CH2
        int throttle_raw = ppm_read(2); // CH3
        int yaw_raw      = ppm_read(3); // CH4
        int arm_raw      = ppm_read(4); // CH5

        bool armed = arm_raw > 1500;

        if (!armed || throttle_raw < 900 || throttle_raw > 2100)
{
```

```
            motor1.pulsewidth_us(1000);
            motor2.pulsewidth_us(1000);
            motor3.pulsewidth_us(1000);
            motor4.pulsewidth_us(1000);
            buzzer.play_disarm_tune();
            sleep_for(100ms);
            continue;
        }

        base_throttle = (float)(throttle_raw - 1000) / 1000.0f;
        base_throttle = fmaxf(base_throttle, 0.15f);

        desired_roll  = (float)(roll_raw  - 1500) / 500.0f;
        desired_pitch = (float)(pitch_raw - 1500) / 500.0f;
        desired_yaw   = (float)(yaw_raw   - 1500) / 500.0f;

        desired_roll  = fminf(fmaxf(desired_roll, -1.0f), 1.0f);
        desired_pitch = fminf(fmaxf(desired_pitch, -1.0f), 1.0f);
        desired_yaw   = fminf(fmaxf(desired_yaw, -1.0f), 1.0f);

        desired_roll  = apply_deadband(desired_roll);
        desired_pitch = apply_deadband(desired_pitch);
        desired_yaw   = apply_deadband(desired_yaw);

        bool stick_override = (pitch_raw > 1900 && roll_raw >
1900);

        float m1, m2, m3, m4;

        if (stick_override) {
            m1 = m2 = m3 = m4 = 0.25f;
            pid_pitch.reset();
            pid_roll.reset();
            pid_yaw.reset();
        }
        else {
            float pitch_out = pid_pitch.compute(desired_pitch +
pitch_trim, pitch);
            float roll_out  = pid_roll.compute(desired_roll,
roll);
            float yaw_out   = pid_yaw.compute(desired_yaw, yaw);

            m1 = base_throttle + pitch_out - roll_out + yaw_out;
            m2 = base_throttle + pitch_out + roll_out - yaw_out;
            m3 = base_throttle - pitch_out + roll_out + yaw_out;
            m4 = base_throttle - pitch_out - roll_out - yaw_out;

            m1 = fminf(fmaxf(m1, 0.15f), 1.0f);
            m2 = fminf(fmaxf(m2, 0.15f), 1.0f);
            m3 = fminf(fmaxf(m3, 0.15f), 1.0f);
            m4 = fminf(fmaxf(m4, 0.15f), 1.0f);
        }

        motor1.pulsewidth_us((int)(1010 + (m1 - 0.150f) * 1000));
        motor2.pulsewidth_us((int)(1010 + (m2 - 0.150f) * 1000));
        motor3.pulsewidth_us((int)(1010 + (m3 - 0.150f) * 1000));
```

```
        motor4.pulsewidth_us((int)(1010 + (m4 - 0.150f) * 1000));

        int len = snprintf(buffer, sizeof(buffer),"M1: %d, M2: 
%d, M3: %d, M4: %d\n\n",(int)(m1 * 100), (int)(m2 * 100), 
(int)(m3 * 100), (int)(m4 * 100));
        usb.write(buffer, len);

        sleep_for(10ms);
    }
}
```

➢ PROGRAM FOR MPU6050:

This functional block facilitates the integration of MPU6050 with STM32 board. 2 arrays have been initiated to store the values from accelerometer ad gyroscope and then a functional block for Mahony filter has been implemented. The values from the array has been compared to the raw values which are read by the sensor in the Mahony filter.

```
#include "mpu6050.h"
#include <cmath>

#define M_PI 3.14159265358979323846f

MPU6050::MPU6050(I2C &i2c, uint8_t addr) : _i2c(i2c), _addr(addr 
<< 1) {
    _timer.start();
}

bool MPU6050::initialize() {
    writeReg(0x6B, 0x00);
    uint8_t whoami;
    readRegs(0x75, &whoami, 1);
    return whoami == 0x68;
}

void MPU6050::calibrate(int samples) {
    int32_t acc_sum[3] = {0}, gyro_sum[3] = {0};

    for (int i = 0; i < samples; ++i) {
        acc_sum[0] += read16(0x3B);
        acc_sum[1] += read16(0x3D);
        acc_sum[2] += read16(0x3F);
        gyro_sum[0] += read16(0x43);
        gyro_sum[1] += read16(0x45);
        gyro_sum[2] += read16(0x47);
        ThisThread::sleep_for(5ms);
    }

    for (int i = 0; i < 3; ++i) {
        acc_bias[i] = acc_sum[i] / (float)samples;
        gyro_bias[i] = gyro_sum[i] / (float)samples;
    }
```

```
    acc_bias[2] -= 16384.0f;
}

void MPU6050::updateMahony() {
    int16_t raw_acc[3], raw_gyro[3];
    raw_acc[0] = read16(0x3B);
    raw_acc[1] = read16(0x3D);
    raw_acc[2] = read16(0x3F);
    raw_gyro[0] = read16(0x43);
    raw_gyro[1] = read16(0x45);
    raw_gyro[2] = read16(0x47);

    float acc[3], gyro[3];
    for (int i = 0; i < 3; ++i) {
        acc[i] = (raw_acc[i] - acc_bias[i]) / 16384.0f;
        gyro[i] = (raw_gyro[i] - gyro_bias[i]) / 131.0f * M_PI /
180.0f;
    }

    float dt =
std::chrono::duration<float>(_timer.elapsed_time()).count();
    _timer.reset();

    float norm = sqrtf(acc[0]*acc[0] + acc[1]*acc[1] +
acc[2]*acc[2]);
    if (norm == 0.0f) return;
    for (int i = 0; i < 3; ++i) acc[i] /= norm;


    float vx = 2*(q1*q3 - q0*q2);
    float vy = 2*(q0*q1 + q2*q3);
    float vz = q0*q0 - q1*q1 - q2*q2 + q3*q3;

    float ex = (acc[1]*vz - acc[2]*vy);
    float ey = (acc[2]*vx - acc[0]*vz);
    float ez = (acc[0]*vy - acc[1]*vx);

    integralFBx += twoKi * ex * dt;
    integralFBy += twoKi * ey * dt;
    integralFBz += twoKi * ez * dt;

    gyro[0] += twoKp * ex + integralFBx;
    gyro[1] += twoKp * ey + integralFBy;
    gyro[2] += twoKp * ez + integralFBz;


    float dq0 = 0.5f * (-q1*gyro[0] - q2*gyro[1] - q3*gyro[2]);
    float dq1 = 0.5f * ( q0*gyro[0] + q2*gyro[2] - q3*gyro[1]);
    float dq2 = 0.5f * ( q0*gyro[1] - q1*gyro[2] + q3*gyro[0]);
    float dq3 = 0.5f * ( q0*gyro[2] + q1*gyro[1] - q2*gyro[0]);

    q0 += dq0 * dt;
    q1 += dq1 * dt;
    q2 += dq2 * dt;
    q3 += dq3 * dt;
```

```
    norm = sqrtf(q0*q0 + q1*q1 + q2*q2 + q3*q3);
    q0 /= norm; q1 /= norm; q2 /= norm; q3 /= norm;


    pitch = asinf(-2.0f * (q1*q3 - q0*q2)) * 180.0f / M_PI;
    roll  = atan2f(2.0f * (q0*q1 + q2*q3), 1.0f - 2.0f * (q1*q1 +
q2*q2)) * 180.0f / M_PI;
    yaw   = atan2f(2.0f * (q0*q3 + q1*q2), 1.0f - 2.0f * (q2*q2 +
q3*q3)) * 180.0f / M_PI;


    float gyro_z_thresh = 0.02f;
    float acc_z_thresh = 0.05f;

    bool is_stationary = fabsf(gyro[2]) < gyro_z_thresh &&
fabsf(acc[2] - 1.0f) < acc_z_thresh;
    if (is_stationary) {
        yaw *= 0.99985f;
    }
}

void MPU6050::getOrientation(float *pitchOut, float *rollOut,
float *yawOut) {
    *pitchOut = pitch;
    *rollOut = roll;
    *yawOut = yaw;
}

void MPU6050::writeReg(uint8_t reg, uint8_t data) {
    char buf[2] = {static_cast<char>(reg),
static_cast<char>(data)};
    _i2c.write(_addr, buf, 2);
}

void MPU6050::readRegs(uint8_t reg, uint8_t *data, uint8_t
length) {
    char r = reg;
    _i2c.write(_addr, &r, 1, true);
    _i2c.read(_addr, reinterpret_cast<char*>(data), length);
}

int16_t MPU6050::read16(uint8_t reg) {
    uint8_t data[2];
    readRegs(reg, data, 2);
    return (int16_t)((data[0] << 8) | data[1]);
}
```

➢ PROGRAM FOR PID TUNING:

This function consolidates the commands to perform the PID tuning to provide stability to the drone by error detection and error correction mechanism. This function firsts calculate the error and then reset the values to 0 and then change them to new value where error is subtracted.

```
#include "pid.h"

PID::PID(float kp, float ki, float kd, float dt)
    : _kp(kp), _ki(ki), _kd(kd), _dt(dt), _prev_error(0.0f),
_integral(0.0f) {}

float PID::compute(float setpoint, float measured) {
    float error = setpoint - measured;
    _integral += error * _dt;
    float derivative = (error - _prev_error) / _dt;
    _prev_error = error;

    float output = _kp * error + _ki * _integral + _kd *
derivative;

    if (output > _max_output) output = _max_output;
    if (output < _min_output) output = _min_output;

    return output;
}

void PID::reset() {
    _prev_error = 0.0f;
    _integral = 0.0f;
}

void PID::setTunings(float kp, float ki, float kd) {
    _kp = kp;
    _ki = ki;
    _kd = kd;
}

void PID::setOutputLimits(float min, float max) {
    _min_output = min;
    _max_output = max;
}

float PID::getKp() const { return _kp; }
float PID::getKi() const { return _ki; }
float PID::getKd() const { return _kd; }
```

➢ PROGRAM FOR RECEIVER:

This function is utilised to read the values sent by the transmitter under the modulation scheme of pulse position modulation. The maximum number of channel is set as 1500 and signals is received and modulated by receiver until the counter is less than maximum limit.

```
#include "ppm.h"

static InterruptIn ppm_input(PPM_PIN);
static Timer ppm_timer;

static volatile int ppm_channels[MAX_CHANNELS] = {1500};
static volatile uint8_t ppm_channel_index = 0;
static volatile int last_rise_time = 0;

void ppm_isr_rise() {
    int now = ppm_timer.elapsed_time().count();
    int duration = now - last_rise_time;
    last_rise_time = now;

    if (duration > 3000) {
        ppm_channel_index = 0;
    } else {
        if (ppm_channel_index < MAX_CHANNELS) {
            ppm_channels[ppm_channel_index] = duration;
            ppm_channel_index++;
        }
    }
}

void ppm_init() {
    ppm_timer.start();
    ppm_input.rise(&ppm_isr_rise);
}

int ppm_read(uint8_t channel) {
    if (channel >= MAX_CHANNELS) return 1500;
    return ppm_channels[channel];
}
```

➢ PROGRAM FOR BUZZER:

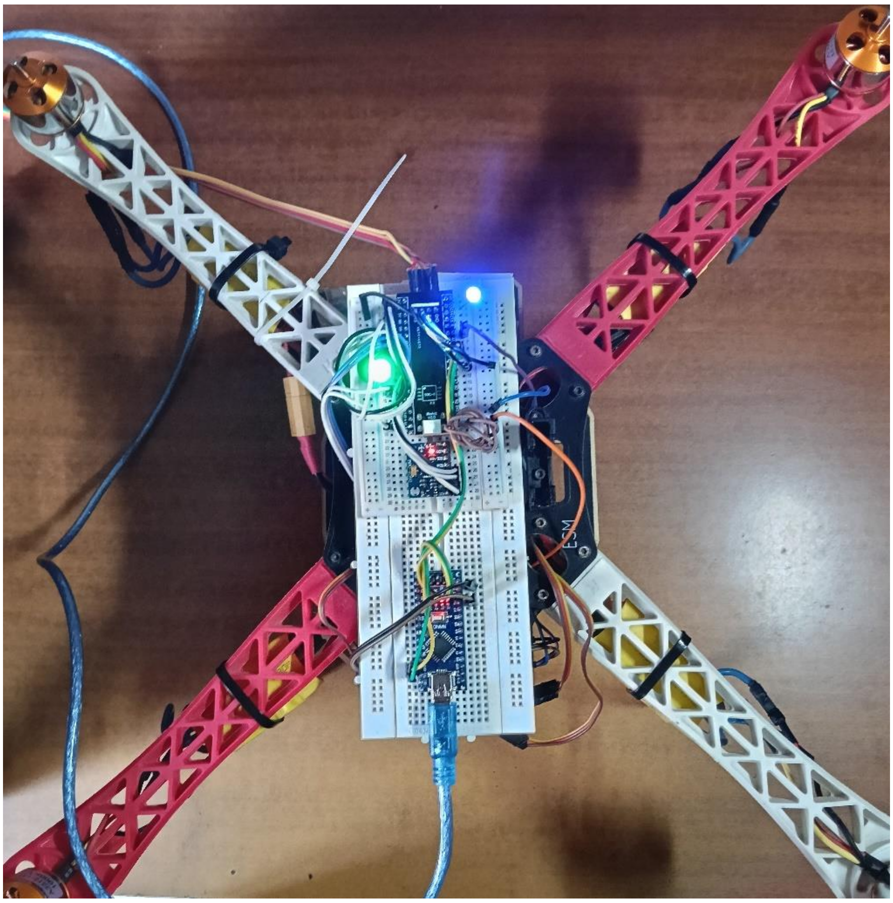
Separate function for buzzer has been compiled to provide a standard melody upon the calibration of motors for the quadcopter. The melody is produced by providing different frequencies of buzzer which set the varying time period for the pwm generation which is given to the buzzer.

```
#include "buzzer.h"
#include "mbed.h"

using namespace std::chrono;

Buzzer::Buzzer(PinName pin) : buzzer(pin) {
    buzzer.write(0.0f);
}

void Buzzer::play_note(float frequency, int duration_ms) {
    if (frequency > 0.0f) {
        buzzer.period(1.0f / frequency);
        buzzer.write(0.5f);
    } else {
        buzzer.write(0.0f);
    }
    ThisThread::sleep_for(milliseconds(duration_ms));
    buzzer.write(0.0f);
}

void Buzzer::play_disarm_tune() {
    float melody[] = { 392, 440, 392, 349, 330 };
    int durations[] = { 200, 200, 200, 200, 400 };

    for (int i = 0; i < 5; i++) {
        play_note(melody[i], durations[i]);
        ThisThread::sleep_for(50ms);
    }
}
```

RESULTS and IMPLEMENTATION

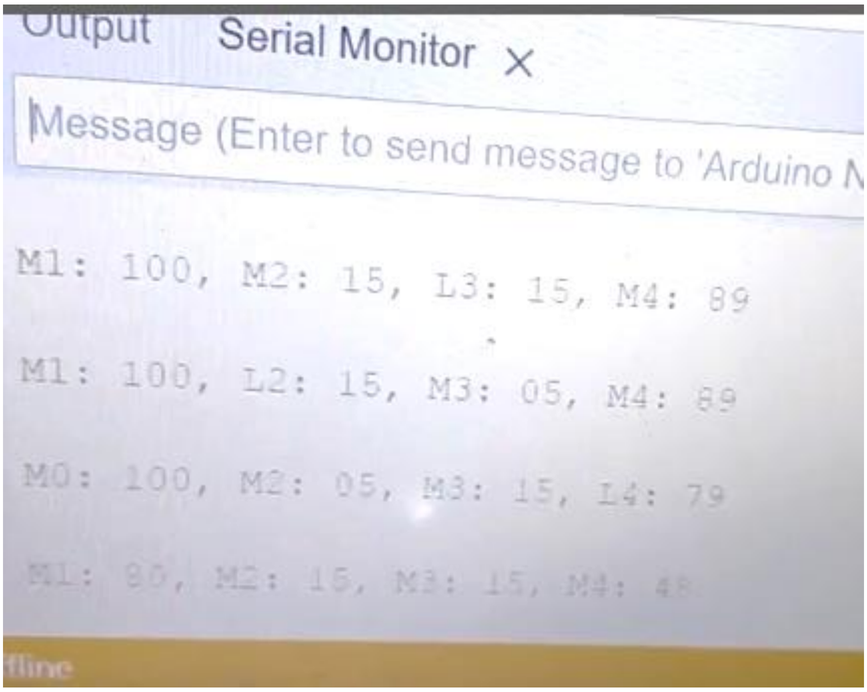
- Initially all the 4 motors beep in random orders which depicts that calibration has not been done. It can also be seen on the serial monitor which displays a message of Initialisation.

- When the flight controller is connected to power supply the booting sequence starts and all the motors get calibrated and produce a standard beep. The calibration done message can be seen on the serial monitor and the output devices like led and buzzer are used to display that the drone is ready to fly.

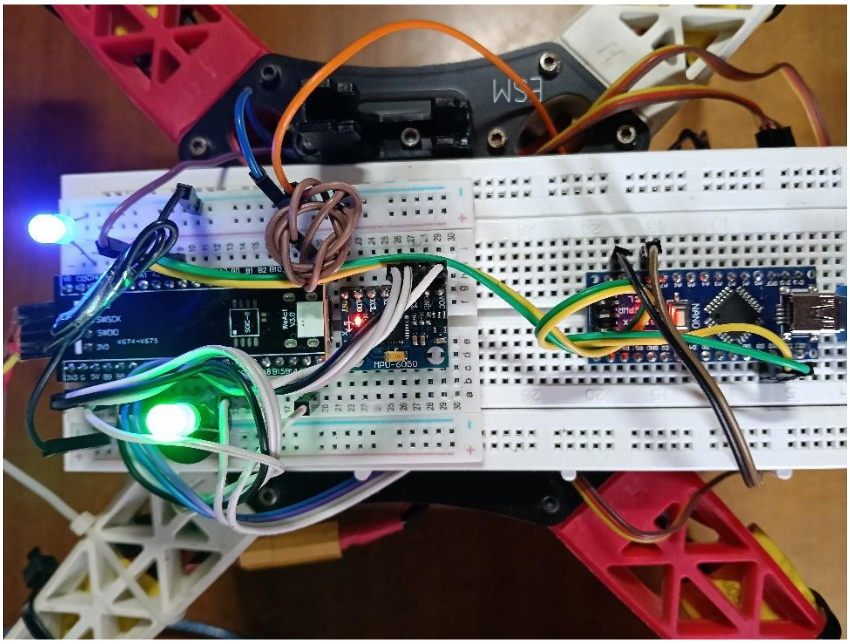


Ready to fly condition

- The drone can be armed b y giving the PPM signal from transmitter which makes the motors rotate as they receive PWM signal. Whenever any control signal is changed from the transmitter, the speed of motor is changed which represents that the PID tuning has been successful.



PWM values given to motors



Hardware Assembly

REFERENCES

- https://ieeexplore.ieee.org/document/10454993
- https://ijirt.org/publishedpaper/IJIRT167412_PAPER.pdf
- https://github.com/saahinduran/STM32-Based-Quadcopter-Flight-Controller-Software
- https://github.com/F33RNI/Liberty-X
- https://www.researchgate.net/publication/319230010_Design_of_Angle_Detection_System_Based_on_MPU6050
- https://ahrs.readthedocs.io/en/latest/filters/mahony.html
- https://www.ijert.org/research/design-of-control-system-for-quadcopter-using-complementary-filter-and-pid-controller-IJERTV3IS041539.pdf
- https://iopscience.iop.org/article/10.1088/1742-6596/1237/3/032080/pdf
- https://www.spiedigitallibrary.org/conference-proceedings-of-spie/13220/132201N/Design-and-implementation-of-UAV-obstacle-avoidance-based-on-STM32/10.1117/12.3037673.short
- https://forum.arduino.cc/t/stm32-based-drone/1192742
- https://github.com/BenGrumm/STM-Quadcopter