**AUTONOMOUS NAVIGATING QUADCOPTER**

internally funded PROJECT REPORT

Submitted by

**Sreenidhi Ganapathi Raman - 312215105108**

**Krishna Kumar R M – 31221510105062**

**T R Madhavan - 312215105**

**Vikram AS**

**Supervisor**

**Dr. M. SENTHIL KUMARAN**

**Asso. Professor, EEE Department**

DEPARTMENT OF electrical and ELECTRONICS ENGINEERING

SSN COLLEGE OF ENGINEERING

KALAVAKKAM-603110

table of contents

1. INTRODUCTION 3
2. QUADCOPTER KINEMATICS 6
3. CONSTRUCTION AND COMPNENTS 7
4. FLIGHT CONTROLLER 17
5. SETTING UP FOR THE 1st FLIGHT 21
6. ANNEXURE 28

**CHAPTER 1**

**INTRODUCTION**

**UNMANNED AERIAL VEHICLES**

An unmanned aerial vehicle (UAV) (or un-crewed aerial vehicle, commonly known as a drone) is an [aircraft](https://en.wikipedia.org/wiki/Aircraft) without a human [pilot](https://en.wikipedia.org/wiki/Aircraft_pilot) on board. UAVs are a component of an [unmanned aircraft system (UAS)](https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle#Terminology); which include a UAV, a ground-based controller, and a system of communications between the two. The flight of UAVs may operate with various degrees of [autonomy](https://en.wikipedia.org/wiki/Vehicular_automation): either under remote control by a human operator or autonomously by onboard computers.

**PROBLEM STATEMENT**

The problem involves an unmanned Aerial vehicle which is remote controlled. A flight controller will be designed which will contain a gyroscope, accelerometer and a barometer. The vehicle will have the ability to be manoeuvred in a controlled environment.

**OBJECTIVES**

**Overall objectives**

To design a quadcopter than can be manoeuvred in a controlled environment.

**Specific objectives**

The following are the specific objectives in order to achieve the overall objective

1. Literature survey.

2. Building the Quadcopter.

3. Designing and Implementing the flight controller.

4. Tuning the system.

5. Communication between the remote and the quadcopter.

6. Testing the Final Design.

**PROJECT SCOPE**

The unmanned Aerial vehicles find its application in

* Defence: Smaller and portable drones are being used by ground forces on a regular basis. Drones in the military are used mostly for surveillance and offensive operations.
* Emergency Response and Disaster Management: As cameras can be mounted on drones, they have now become lifesavers as well. Drones can be very useful during times of natural disaster for the surveillance of disaster-affected areas. After consequences of natural disaster, UAVs can be used to assess damage, locate victims, and deliver aid. In certain circumstances, they are being used to prevent disasters altogether.
* Urban Planning: As urbanization is growing up with the fast pace and with the emergence of smart cities concept all around the world, it is very important to map and survey the land which has to be developed. In this, drones have a great role to play. They can provide instant mapping and ready to use data for the purpose. Drones are also light on the budget side which can help city planners to decide which areas may benefit most from green space, without causing further congestion.
* Health Care: With increasing number of life-threatening diseases, modern medicine is the need of the time to increase the life expectancy and save lives.  However, the availability is easy in urban areas, the reach of the modern medical facility is still time taking in rural areas. Here drones can play an important role by delivering quick access to drugs, blood, and medical technology in remote areas.
* Agriculture:  Drones in agriculture can help farmers to gather data and automate redundant processes to maximize efficiency. They can also be used to spray medicines to kill insects impacting their health. In the process of planting crops which is tedious and equally energy-intensive, drones can ease the work by distributing seed on the land.

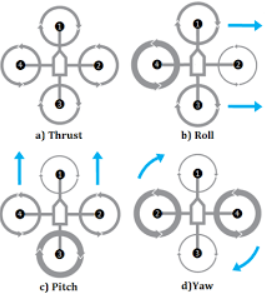
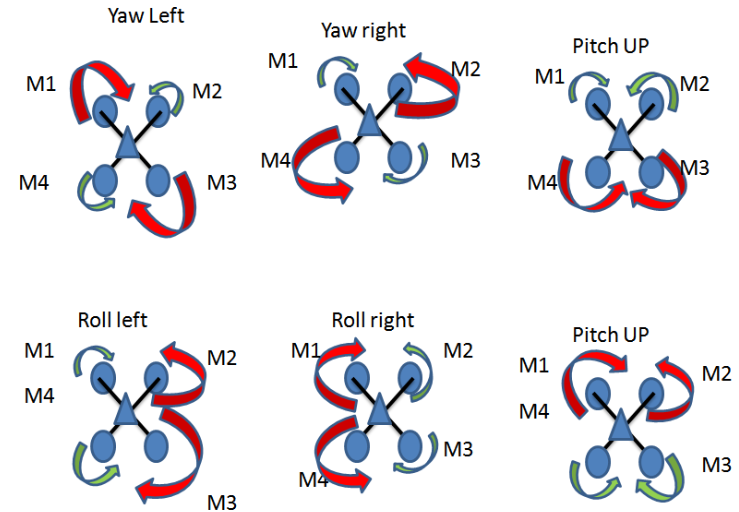
  

**CHAPTER 2**

**QUADCOPTER KINEMATICS**

A quadcopter,  also called a  quadrotor helicopter or quadrotor, is a [multi-rotor](https://en.wikipedia.org/wiki/Multirotor) [helicopter](https://en.wikipedia.org/wiki/Helicopter) that is lifted and propelled by four [rotors](https://en.wikipedia.org/wiki/Helicopter_rotor). Quadcopters are classified as [rotorcraft](https://en.wikipedia.org/wiki/Rotorcraft), as opposed to [fixed-wing aircraft](https://en.wikipedia.org/wiki/Fixed-wing_aircraft), because their [lift](https://en.wikipedia.org/wiki/Lift_(force)) is generated by a set of [rotors](https://en.wikipedia.org/wiki/Helicopter_rotor) (vertically oriented [propellers](https://en.wikipedia.org/wiki/Propeller)). Quadcopters differ from conventional helicopters, which use rotors that are able to vary the pitch of their blades dynamically as they move around the rotor hub.

Quadcopters generally use two pairs of identical fixed pitched propellers; two [clockwise](https://en.wikipedia.org/wiki/Clockwise) (CW) and two [counter clockwise](https://en.wikipedia.org/wiki/Counterclockwise) (CCW). These use independent variation of the speed of each rotor to achieve control. By changing the speed of each rotor it is possible to specifically generate a desired total [thrust](https://en.wikipedia.org/wiki/Thrust); to locate for the [centre of thrust](https://en.wikipedia.org/wiki/Centre_of_thrust) both laterally and longitudinally; and to create a desired total [torque](https://en.wikipedia.org/wiki/Torque), or turning force.

**CHAPTER 3**

**CONSTRUCTION AND COMPONENTS**

**PROPELLERS:**

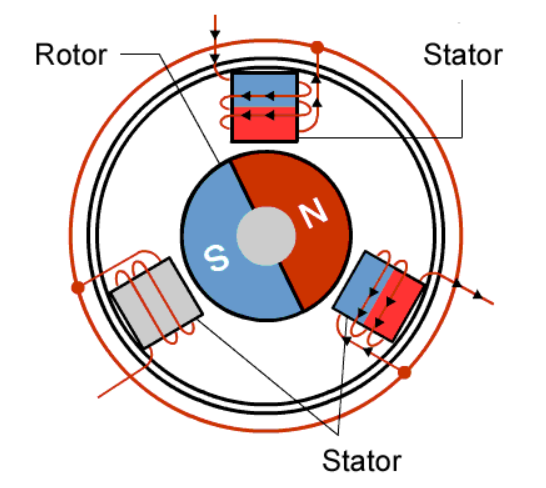
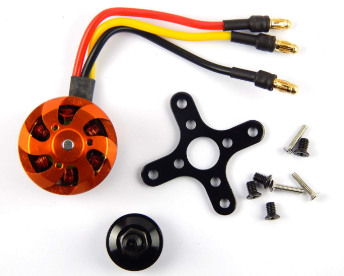
The propellers are available in two configurations. One pushes down air when rotated in clockwise direction and the other pushes down air when rotated in counter clockwise direction.

****

**BLDC MOTORS**

Brushless DC motors do not use brushes. With brushed motors, the brushes deliver current through the commutator into the coils on the rotor. In a BLDC motor the coils are not located on the stator. The rotor is a permanent magnet. Because the coils do not move, there is no need for brushes and a commutator.

With the brushed motor, rotation is achieved by controlling the magnetic fields generated by the coils on the rotor, while the magnetic field generated by the stationary magnets remains fixed. To change the rotation speed, the voltage for the coils is changed. With a BLDC motor, it is the permanent magnet that rotates; rotation is achieved by changing the direction of the magnetic fields generated by the surrounding stationary coils. To control the rotation, the magnitude and direction of the current into these coils is adjusted.

**ELECTRONIC SPEED CONTROLLER**

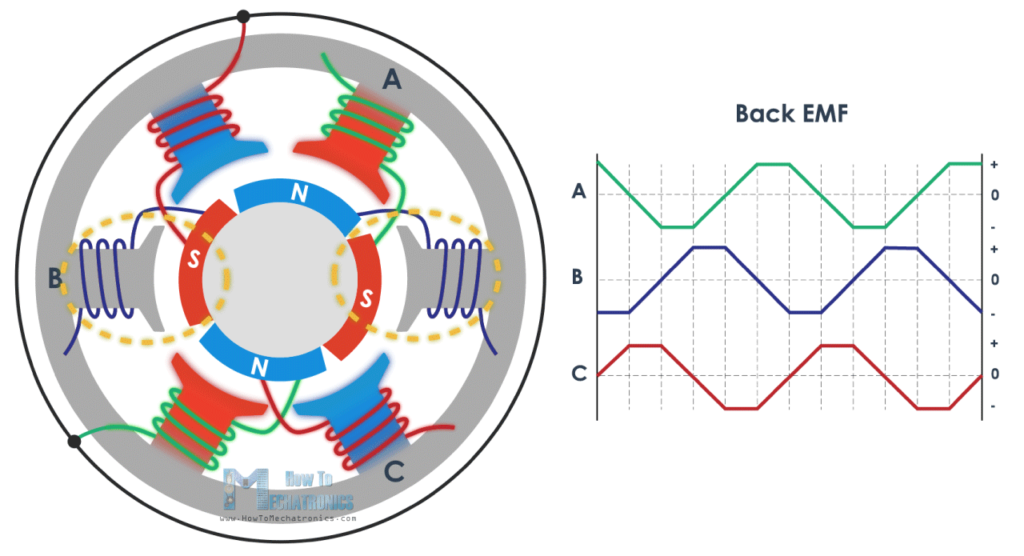
An ESC or an Electronic Speed Controller controls the brushless motor movement or speed by activating the appropriate MOSFETs to create the rotating magnetic field so that the motor rotates. The higher the frequency or the quicker the ESC goes through the 6 intervals, the higher the speed of the motor will be. To activate the appropriate phase, the position of the rotor is determined by two common methods.

The first common method is by using [Hall-effect](https://howtomechatronics.com/how-it-works/electrical-engineering/hall-effect-hall-effect-sensors-work/) sensors embedded in the stator, arranged equally 120 or 60 degrees from each other.



As the rotors permanent magnets rotate the Hall-effect sensors sense the magnetic field and generate a logic “high” for one magnetic pole or logic “low” for the opposite pole. According to this information the ESC knows when to activate the next commutation sequence or interval.

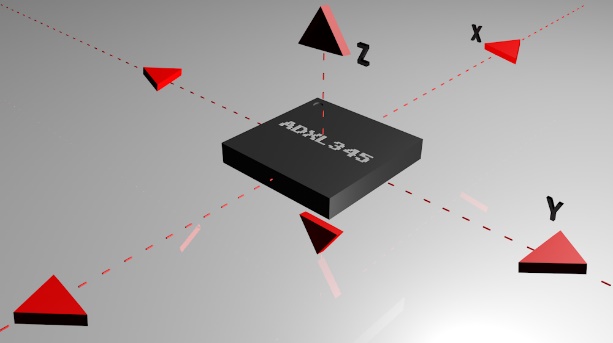
The second common method used for determining the rotor position is through sensing the back electromotive force or back EMF. The back EMF occurs as a result of the exact opposite process of generating a magnetic field or when a moving or changing magnetic field pass through a coil it induces a current in the coil.



So, when the moving magnetic field of the rotor pass through the free coil, or the one that’s not active, it will induce a current flow in coil and as result a voltage drop will occur in that coil. The ESC captures these voltage drops as they occur and based on them it predicts or calculates when the next interval should take place.

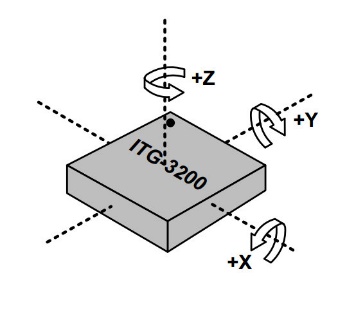
**ACCELEROMETER**

Accelerometers are devices that measure [acceleration](http://en.wikipedia.org/wiki/Acceleration), which is the rate of change of the [velocity](http://en.wikipedia.org/wiki/Velocity) of an object. They measure in meters per second squared (m/s2) or in G-forces (g). A single G-force for us here on planet Earth is equivalent to 9.8 m/s2, but this does vary slightly with elevation (and will be a different value on different planets due to variations in gravitational pull). Accelerometers are useful for sensing vibrations in systems or for orientation applications.



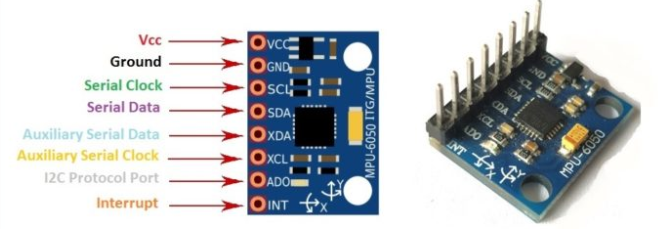
**GYROSCOPE**

Gyroscopes, or gyros, are devices that measure or maintain rotational motion. [MEMS](http://en.wikipedia.org/wiki/Microelectromechanical_systems) (microelectromechanical system) gyros are small, inexpensive sensors that measure angular velocity. The units of angular velocity are measured in degrees per second (°/s) or revolutions per second (RPS). Angular velocity is simply a measurement of speed of rotation.



**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.
* MPU6050 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 uses the I2C module for interfacing with the STM32.



**BAROMETER**

A **barometer** is a scientific instrument that is used to measure air pressure in a certain environment. Barometers and [pressure altimeters](https://en.wikipedia.org/wiki/Pressure_altimeter) (the most basic and common type of altimeter) are essentially the same instrument, but used for different purposes. An altimeter is intended to be used at different levels matching the corresponding atmospheric pressure to the [altitude](https://en.wikipedia.org/wiki/Altitude), while a barometer is kept at the same level and measures subtle pressure changes.

**MS 6511**

The MS6511 is a high precision pressure sensor module and includes a linear

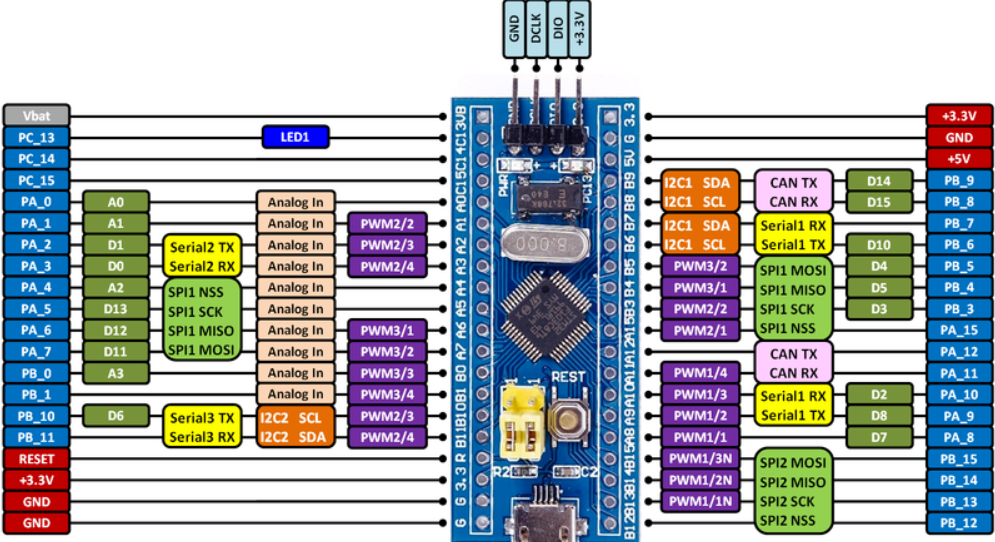
pressure measurement element (pressure die) and an ultra low power 24 bit

A/D- converter with internal factory calibrated coefficients. The main function is to convert the uncompensated analogue voltage from pressure die to a 24 bit digital value, as well as providing a 24 bit value for temperature. These values can be calculated with the aid of the internal coefficients in any external processor to the real calibrated values.



**STM 32**

**STM32** is a family of 32-bit [microcontroller](https://en.wikipedia.org/wiki/Microcontroller) [integrated circuits](https://en.wikipedia.org/wiki/Integrated_circuit) by [STMicroelectronics](https://en.wikipedia.org/wiki/STMicroelectronics).



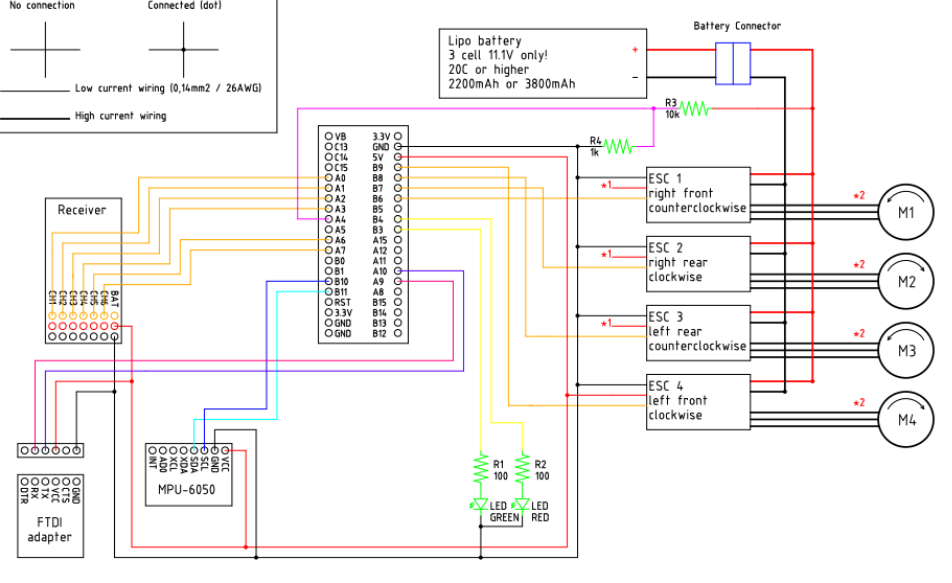
**LIPO BATTERY**

A **lithium polymer battery**, or more correctly **lithium-ion polymer** is a [rechargeable battery](https://en.wikipedia.org/wiki/Rechargeable_battery) of [lithium-ion](https://en.wikipedia.org/wiki/Lithium-ion_battery) technology using a polymer electrolyte instead of a liquid electrolyte.

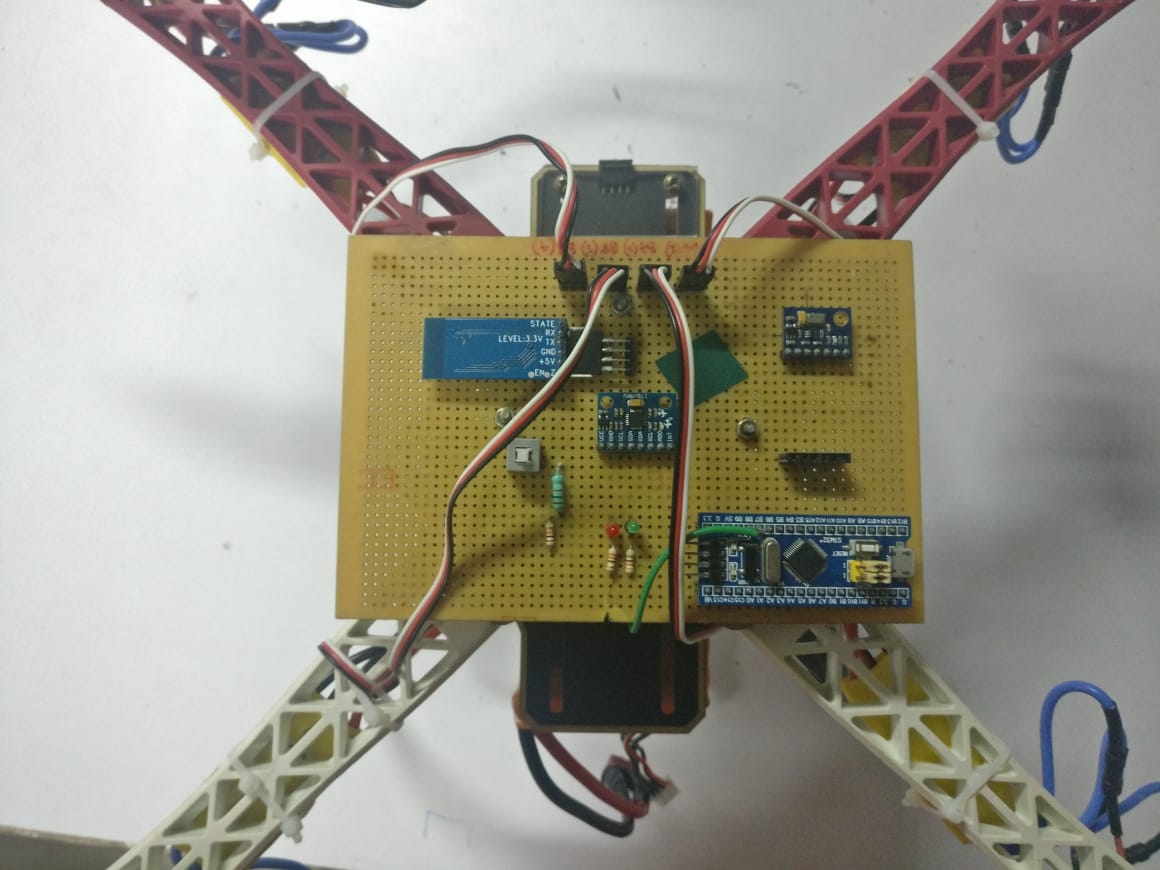


**CONSTRUCTION**

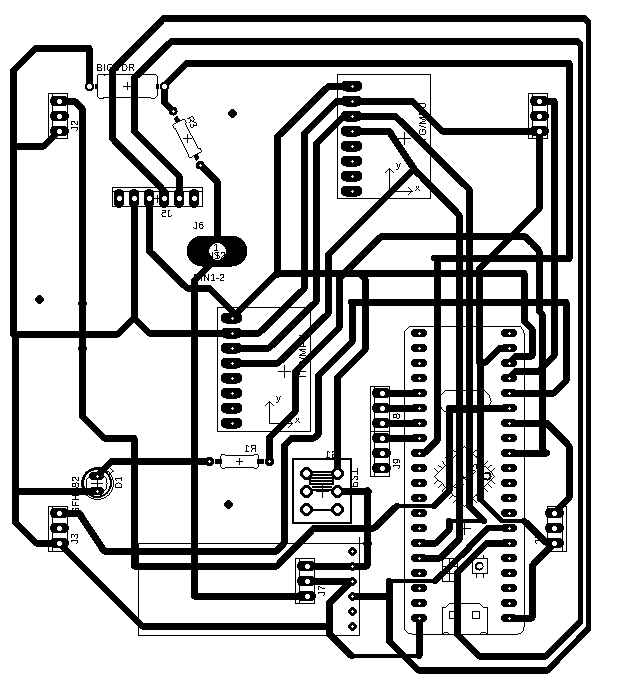
**CIRCUIT DIAGRAM**



**CIRCUIT IMPLEMENTATION**



**PCB LAYOUT FOR FLIGHT CONTROLLER**



**COMPLETED SETUP**



**CHAPTER 4**

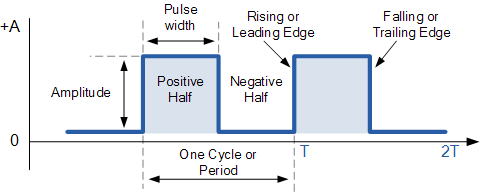
**FLIGHT CONTROLLER**

The flight controller is the main part of this project as it controls the movement of the quadcopter based on the input from the RC receiver. The STM32 microcontroller acts as the flight controller that is aided by the Inertial measurement unit (IMU) and consists of MPU6050 sensor which in turn consists the gyroscope and accelerometer.

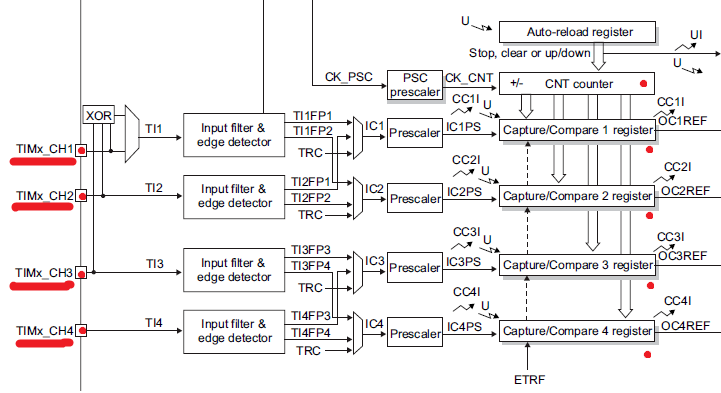
The pulses from the RC receiver is obtained using input capture mode and the output to the electronic speed controller (ESC) is based on the PID output.

**Input Capture Mode**

It is one of the functions of the Timer which identifies the width of input signals by using Input capture. It will record a timestamp in memory when an input signal is received. It will also set a flag indicating that an input has been captured so that you can read out the capture value easily through interrupt or event polling. Input capture will help determine the cycle period or the pulse width of the input pulse.

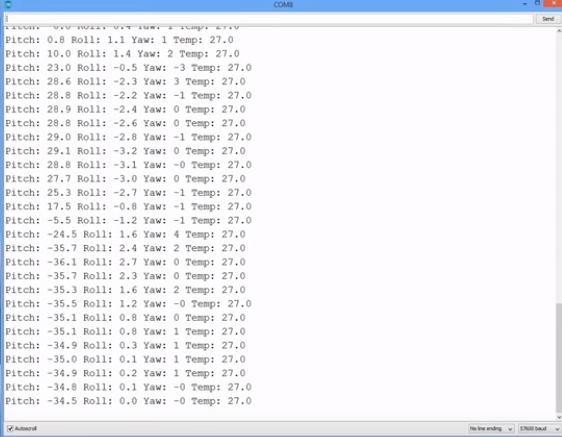


Similar to external input counter function, the input capture function also uses these pins with name “TIMx\_CH1″,…,”TIMx\_CH4”. Each time the input capture is triggered, it will latch the counter value into the Capture/Compare register of the respective channel.

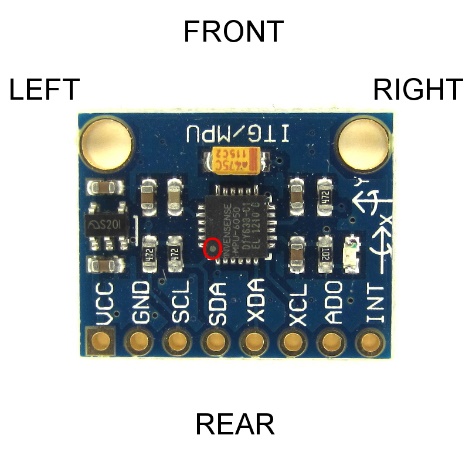


The flight controller has two main functions. The first one is to balance the quadcopter making minute corrections to roll and pitch angle. The second function is to manoeuvre the quadcopter based on the input to the remote controller.

The IMU consisting of gyroscope and accelerometer help in finding the orientation of the quadcopter i.e. to find the roll and pitch angle. The gyroscope gives the angular velocity. By integrating the angular velocity corresponding angular displacement i.e. the roll and pitch angles are calculated. Any deviation in angle values is corrected using the accelerometer readings.

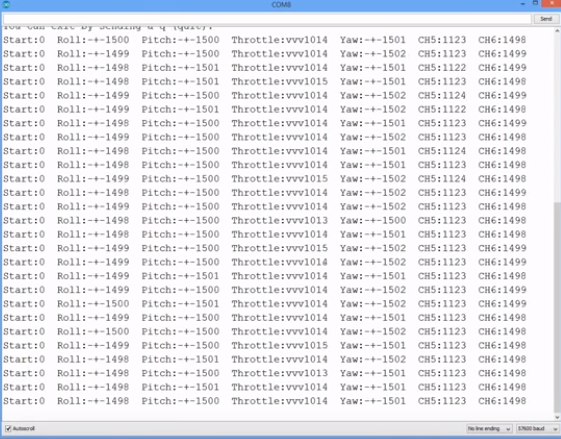


The MPU6050 should be installed in particular orientation with respect to the quadcopter as shown.



The PID controller makes sure the quadcopters orientation is such that the roll and pitch angles are within the given tolerance values.

The values from the transmitter varies between 1000us to 2000us. When the esc is fed with 1000us, the motor attached to it is at rest and when it is 2000us, it rotates at the maximum speed.



**CHAPTER 5**

**SETTING UP FOR THE 1st FLIGHT**

**THE TRANSMITTER AND RECEIVER**

Connect the roll (aileron), pitch (elevator), yaw (rudder) and throttle output of the receiver to the STM32 Ports. The order is not important as the setup software will recognize each separate channel. The receiver is powered by the +5V output of the STM32.

**RECEIVER AND GYRO CHECK**

To make sure that everything is working correct it's necessary to run some basic checks. The propellers should be removed, and the flight battery should be disconnected and the ESC calibration program is uploaded to the STM32. The serial monitor is opened at 57600baud.

**Receiver input check**

The receiver monitor can be started by sending the letter ‘r’. The sticks are moved and checked if the values on the screen correspond with the movements of the sticks. All the channels should read 1000us till 2000us with a centre position of 1500 (+/-8).

**Gyro / accelerometer angle check**

After the receiver check is completed the angle check can be started by sending the letter ‘a’. The quadcopter should not be moved because the gyro needs to calibrate itself. After the calibration the roll and pitch angles are shown. The yaw value is the output of the gyro and will go back to zero if the yaw rotation stops.

It is checked if the angles correspond with the movement of the quadcopter

* Nose up is positive pitch and nose down is negative pitch.
* Left wing up is positive roll and left wing down is negative roll.
* Nose right is positive yaw and nose left is negative yaw.

**ELECTRONIC SPEED CONTROLLER**

Only the ground and the signal wires of the ESC's are connected. The +5V from the ESC is not connected because the STM32 gets its power directly from the flight battery via the diode D1.

In some cases, the ground of the ESC doesn't have to be connected. It can be checked with a multi-meter if the ground of the battery connection is connected to the ground / - of the esc connection wire. If these are connected the ground of the ESC does not need to be connected to the Arduino because they share the same battery ground. Also, the direction of rotation has to be checked.

Electronic speed controllers or ESC's for short are controlled with a 1000us till 2000us pulse. 1000us means off and 2000us means full throttle. To make sure that all the ESC's react the same way it's important to calibrate the 1000us and 2000us point. Without calibration the motors will perform different and the quadcopter doesn’t fly well or might even crash.

All the propellers are removed and the ESC calibration program is uploaded to the STM32. The USB cable is disconnected and following the instructions are implemented in the manual to calibrate the ESC's.

In most cases this is done with the following steps:

1. Place the throttle stick in the upper position (full throttle)
2. Connect the flight battery
3. After some beeps place the throttle stick in the lowest position
4. Disconnect the flight battery

BALANCE THE MOTOR AND PROPELLERS

Balancing the propellers is incredibly important! Without well balanced props and motors the gyro and accelerometer will produce noise that makes the motors react jerky. There is minimal stability and the quadcopter can't level itself.

To get the best performance the propellers and motors need to be balanced perfectly. Putting the gyro / accelerometer on vibration dampeners does not help and can only make things worse.

The propellers are mounted on the motors and checked if the counter clock wise and clock wise props are in the right position. The ESC calibration program is uploaded and Serial monitor is opened at 57600baud. # '1' is sent via the serial monitor and wait for the response "Test motor 1 (right front CCW.)".

The numbers that are printed on the screen represents the amount of vibration measured by the accelerometer. This is not a standardized value and should only be used to minimize the amount of vibration of your YMFC-AL quadcopter.

The quadcopter is held firmly down. The throttle is placed in the lowest position and connected the flight battery. The throttle is slowly increased until motor 1 starts to spin. It must be checked if the direction of rotation and that the prop produces upward thrust. If the motor rotates in the wrong direction, two of the three motor wires have to be switched. The throttle must be put in the lowest position to stop the motor.

The motor frame is held firmly and the throttle is increased to half throttle. The numbers on the screen are checked.



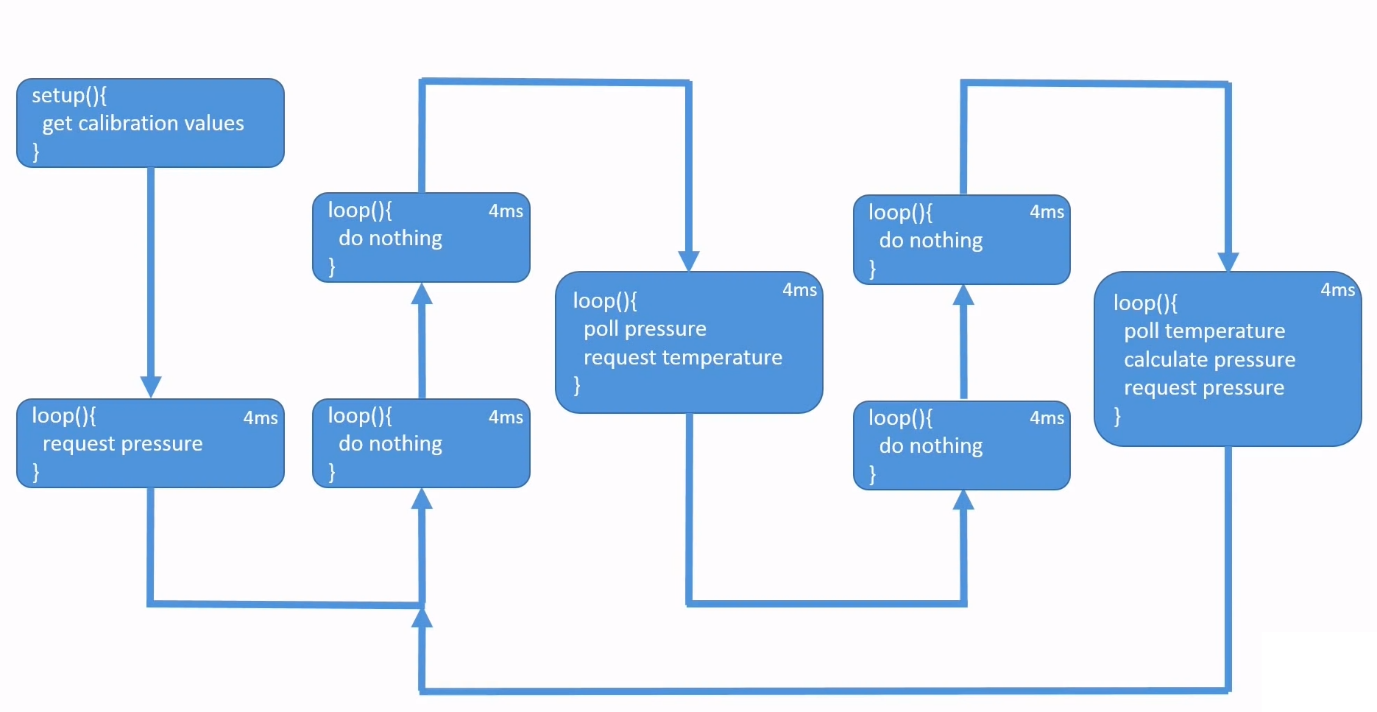
The motor is stopped a small piece of tape is put on one of the blades and the test is repeated. If the vibrations do not reduce, a piece of tape is put on the other blade. This is done until the motors and propeller run as smoothly as possible. This process is repeated for all the motors.

**HEIGHT STABLISATION:**

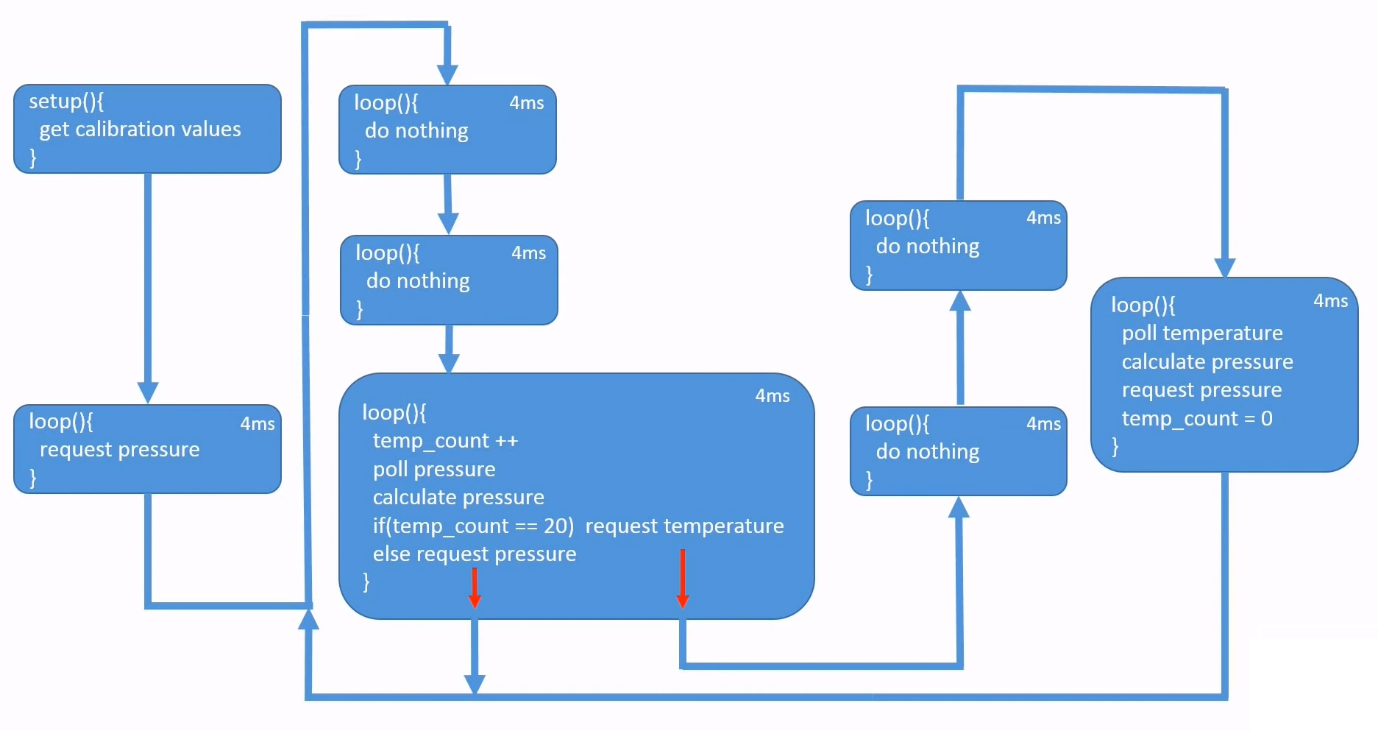
During the initial calibration process, after the calibration of the IMU, the barometer is also calibrated. This helps set the ground level of the quadcopter. Before the flight, a desired set position is specified in the program. During the flight, when Altitude Hold mode is activated, the quadcopter tries to reach the desired position and tries to maintain the state. The PID tuning was done using trial and error method.

The MS6511 barometer is a very sensitive yet accurate barometer. The IC is sensitive to even small change in temperatures. In order to produce accurate readings, a 3D casing was printed and painted in black to enclose the MS6511.

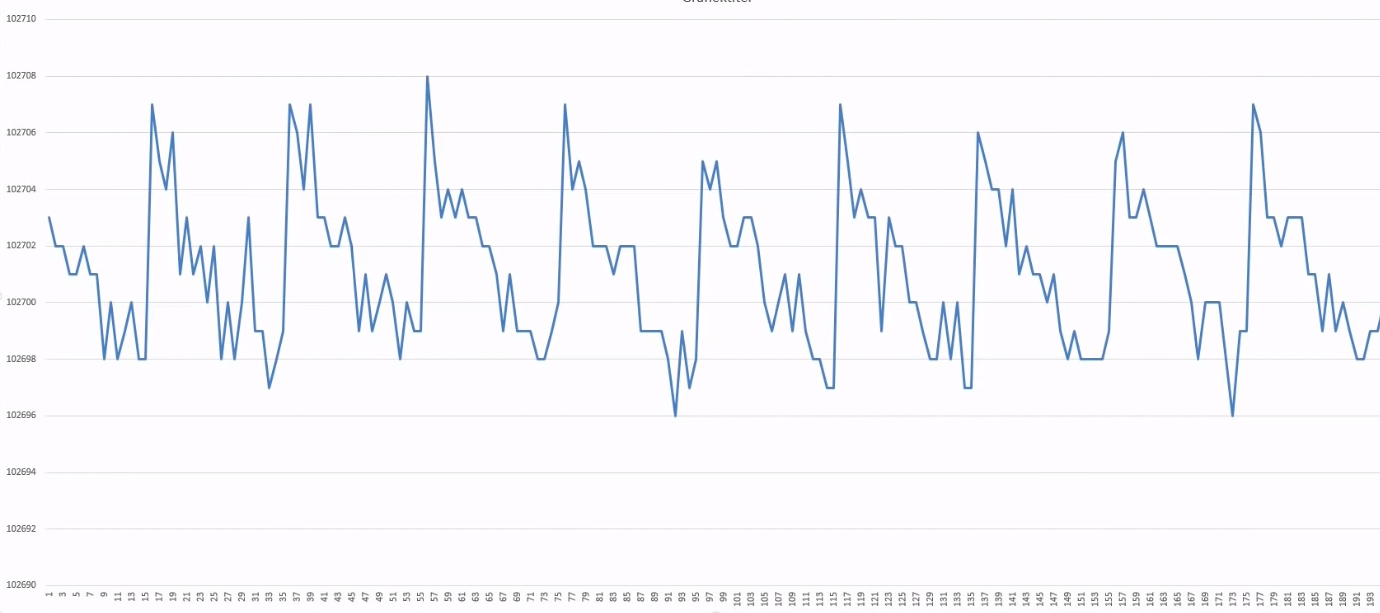
Flow chart for obtaining pressure reading

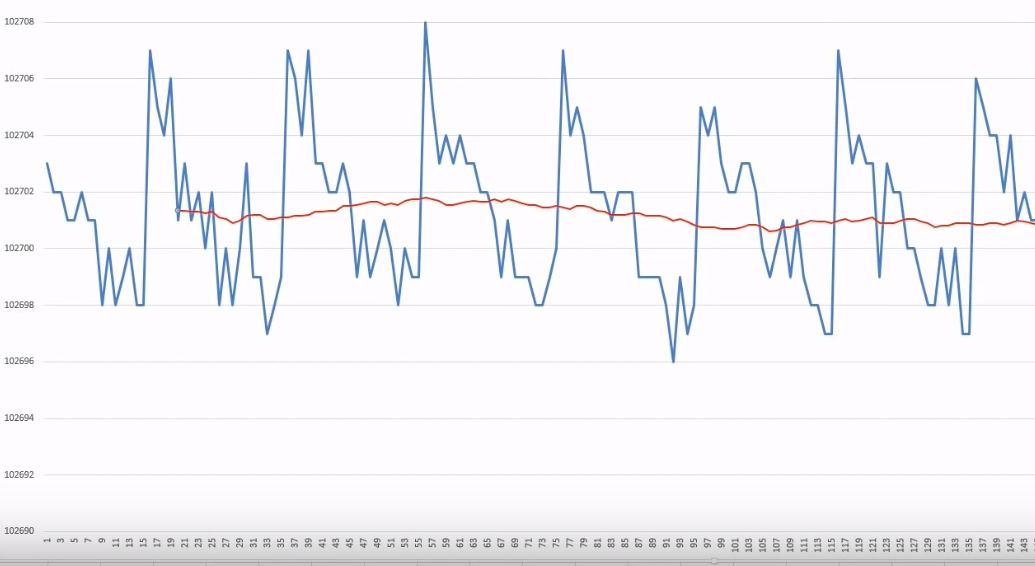


This method can obtain 41.7 readings per second. But in order to increase the refresh rate, instead of reading the temperature in every cycle, it can be read once in every twenty cycles.



On observation of the readings it can be determined that, in all the cycles in which the temperature reading is obtained, a spike occurs in the pressure reading.

In order to overcome this problem, the pressure reading is averaged over twenty cycles.



The averaged pressure reading is indicated in red colour.

**OVERALL WORKING OF THE PROGRAM**

When the battery is connected to the circuit, the stm32 sends a 1000us pulse and then a 2000us pulse to all ESCs to calibrate the motors. Then the values of the gyroscope are read to find the offset and corresponding correction is made during the main program. Each axis of the gyroscope gives the value of angular velocity as a 16bit number. This is then converted to decimal value and fed to the PID controller. Also, the values obtained via input capture from the RC receiver channels is fed to the PID controller. The output of the PID controller varies between 1000-2000us which is fed to the ESCs which in turn run the 4 motors.

**REFERENCES**

[1] YouTube Channel – Joop Brokking - <https://www.youtube.com/watch?v=2BLb6qUKikI>

**ANNEXURE**

>>> **MAIN PROGRAM**

#include <Wire.h> //Include the Wire.h library so we can communicate with the gyro.

TwoWire HWire(2, I2C\_FAST\_MODE); //Initiate I2C port 2 at 400kHz.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//PID gain and limit settings

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

float pid\_p\_gain\_roll = 0.45; //Gain setting for the pitch and roll P-controller (default = 1.3).

float pid\_i\_gain\_roll = 0.001; //Gain setting for the pitch and roll I-controller (default = 0.04).

float pid\_d\_gain\_roll = 3.15; //Gain setting for the pitch and roll D-controller (default = 18.0).

int pid\_max\_roll = 400; //Maximum output of the PID-controller (+/-).

float pid\_p\_gain\_pitch = pid\_p\_gain\_roll; //Gain setting for the pitch P-controller.

float pid\_i\_gain\_pitch = pid\_i\_gain\_roll; //Gain setting for the pitch I-controller.

float pid\_d\_gain\_pitch = pid\_d\_gain\_roll; //Gain setting for the pitch D-controller.

int pid\_max\_pitch = pid\_max\_roll; //Maximum output of the PID-controller (+/-).

float pid\_p\_gain\_yaw = 4.0; //Gain setting for the pitch P-controller (default = 4.0).

float pid\_i\_gain\_yaw = 0.02; //Gain setting for the pitch I-controller (default = 0.02).

float pid\_d\_gain\_yaw = 0.0; //Gain setting for the pitch D-controller (default = 0.0).

int pid\_max\_yaw = 400; //Maximum output of the PID-controller (+/-).

boolean auto\_level = true; //Auto level on (true) or off (false).

//Manual accelerometer calibration values for IMU angles:

int16\_t manual\_acc\_pitch\_cal\_value = 38;

int16\_t manual\_acc\_roll\_cal\_value = -33;

//Manual gyro calibration values.

//Set the use\_manual\_calibration variable to true to use the manual calibration variables.

uint8\_t use\_manual\_calibration = false; // Set to false or true;

int16\_t manual\_gyro\_pitch\_cal\_value = 0;

int16\_t manual\_gyro\_roll\_cal\_value = 0;

int16\_t manual\_gyro\_yaw\_cal\_value = 0;

uint8\_t gyro\_address = 0x68; //The I2C address of the MPU-6050 is 0x68 in hexadecimal form.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Declaring global variables

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//int16\_t = signed 16 bit integer

//uint16\_t = unsigned 16 bit integer

uint8\_t last\_channel\_1, last\_channel\_2, last\_channel\_3, last\_channel\_4;

uint8\_t highByte, lowByte, flip32, start;

uint8\_t error, error\_counter, error\_led;

int16\_t esc\_1, esc\_2, esc\_3, esc\_4;

int16\_t throttle, cal\_int;

int16\_t temperature, count\_var;

int16\_t acc\_x, acc\_y, acc\_z;

int16\_t gyro\_pitch, gyro\_roll, gyro\_yaw;

int32\_t channel\_1\_start, channel\_1;

int32\_t channel\_2\_start, channel\_2;

int32\_t channel\_3\_start, channel\_3,k;

int32\_t channel\_4\_start, channel\_4;

int32\_t channel\_5\_start, channel\_5;

int32\_t channel\_6\_start, channel\_6;

int32\_t acc\_total\_vector;

int32\_t gyro\_roll\_cal, gyro\_pitch\_cal, gyro\_yaw\_cal;

uint32\_t loop\_timer, error\_timer;

float roll\_level\_adjust, pitch\_level\_adjust;

float pid\_error\_temp;

float pid\_i\_mem\_roll, pid\_roll\_setpoint, gyro\_roll\_input, pid\_output\_roll, pid\_last\_roll\_d\_error;

float pid\_i\_mem\_pitch, pid\_pitch\_setpoint, gyro\_pitch\_input, pid\_output\_pitch, pid\_last\_pitch\_d\_error;

float pid\_i\_mem\_yaw, pid\_yaw\_setpoint, gyro\_yaw\_input, pid\_output\_yaw, pid\_last\_yaw\_d\_error;

float angle\_roll\_acc, angle\_pitch\_acc, angle\_pitch, angle\_roll;

float battery\_voltage;

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Setup routine

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void setup() {

pinMode(4, INPUT\_ANALOG); //This is needed for reading the analog value of port A4.

//Port PB3 and PB4 are used as JTDO and JNTRST by default.

//The following function connects PB3 and PB4 to the

//alternate output function.

afio\_cfg\_debug\_ports(AFIO\_DEBUG\_SW\_ONLY); //Connects PB3 and PB4 to output function.

//On the Flip32 the LEDs are connected differently. A check is needed for controlling the LEDs.

pinMode(PB3, INPUT); //Set PB3 as input.

pinMode(PB4, INPUT); //Set PB4 as input.

if (digitalRead(PB3) || digitalRead(PB3))flip32 = 1; //Input PB3 and PB4 are high on the Flip32

else flip32 = 0;

pinMode(PB3, OUTPUT); //Set PB3 as output.

pinMode(PB4, OUTPUT); //Set PB4 as output.

green\_led(LOW); //Set output PB3 low.

red\_led(HIGH); //Set output PB4 high.

//Serial.begin(57600); //Set the serial output to 57600 kbps. (for debugging only)

//delay(250); //Give the serial port some time to start to prevent data loss.

timer\_setup(); //Setup the timers for the receiver inputs and ESC's output.

delay(50); //Give the timers some time to start.

HWire.begin(); //Start the I2C as master

HWire.beginTransmission(gyro\_address); //Start communication with the MPU-6050.

error = HWire.endTransmission(); //End the transmission and register the exit status.

while (error != 0) { //Stay in this loop because the MPU-6050 did not responde.

error = 2; //Set the error status to 2.

error\_signal(); //Show the error via the red LED.

delay(4);

}

gyro\_setup(); //Initiallize the gyro and set the correct registers.

if (!use\_manual\_calibration) {

//Create a 5 second delay before calibration.

for (count\_var = 0; count\_var < 1250; count\_var++) { //1250 loops of 4 microseconds = 5 seconds

if (count\_var % 125 == 0) { //Every 125 loops (500ms).

digitalWrite(PB4, !digitalRead(PB4)); //Change the led status.

}

delay(4); //Delay 4 microseconds

}

count\_var = 0; //Set start back to 0.

}

calibrate\_gyro(); //Calibrate the gyro offset.

//Wait until the receiver is active.

while (channel\_1 < 990 || channel\_2 < 990 || channel\_3 < 990 || channel\_4 < 990) {

error = 3; //Set the error status to 3.

error\_signal(); //Show the error via the red LED.

delay(4);

}

error = 0; //Reset the error status to 0.

//Wait until the throtle is set to the lower position.

while (channel\_3 < 990 || channel\_3 > 1050) {

error = 4; //Set the error status to 4.

error\_signal(); //Show the error via the red LED.

delay(4);

}

error = 0; //Reset the error status to 0.

//When everything is done, turn off the led.

red\_led(LOW); //Set output PB4 low.

//Load the battery voltage to the battery\_voltage variable.

//The STM32 uses a 12 bit analog to digital converter.

//analogRead => 0 = 0V ..... 4095 = 3.3V

//The voltage divider (1k & 10k) is 1:11.

//analogRead => 0 = 0V ..... 4095 = 36.3V

//36.3 / 4095 = 112.81.

battery\_voltage = (float)analogRead(4) / 112.81;

loop\_timer = micros(); //Set the timer for the first loop.

green\_led(HIGH); //Turn on the green led.

}

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Main program loop

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void loop() {

error\_signal(); //Show the errors via the red LED.

gyro\_signalen(); //Read the gyro and accelerometer data.

//65.5 = 1 deg/sec (check the datasheet of the MPU-6050 for more information).

gyro\_roll\_input = (gyro\_roll\_input \* 0.7) + (((float)gyro\_roll / 65.5) \* 0.3); //Gyro pid input is deg/sec.

gyro\_pitch\_input = (gyro\_pitch\_input \* 0.7) + (((float)gyro\_pitch / 65.5) \* 0.3);//Gyro pid input is deg/sec.

gyro\_yaw\_input = (gyro\_yaw\_input \* 0.7) + (((float)gyro\_yaw / 65.5) \* 0.3); //Gyro pid input is deg/sec.

//Gyro angle calculations

//0.0000611 = 1 / (250Hz / 65.5)

angle\_pitch += (float)gyro\_pitch \* 0.0000611; //Calculate the traveled pitch angle and add this to the angle\_pitch variable.

angle\_roll += (float)gyro\_roll \* 0.0000611; //Calculate the traveled roll angle and add this to the angle\_roll variable.

//0.000001066 = 0.0000611 \* (3.142(PI) / 180degr) The Arduino sin function is in radians and not degrees.

angle\_pitch -= angle\_roll \* sin((float)gyro\_yaw \* 0.000001066); //If the IMU has yawed transfer the roll angle to the pitch angel.

angle\_roll += angle\_pitch \* sin((float)gyro\_yaw \* 0.000001066); //If the IMU has yawed transfer the pitch angle to the roll angel.

//Accelerometer angle calculations

acc\_total\_vector = sqrt((acc\_x \* acc\_x) + (acc\_y \* acc\_y) + (acc\_z \* acc\_z)); //Calculate the total accelerometer vector.

if (abs(acc\_y) < acc\_total\_vector) { //Prevent the asin function to produce a NaN.

angle\_pitch\_acc = asin((float)acc\_y / acc\_total\_vector) \* 57.296; //Calculate the pitch angle.

}

if (abs(acc\_x) < acc\_total\_vector) { //Prevent the asin function to produce a NaN.

angle\_roll\_acc = asin((float)acc\_x / acc\_total\_vector) \* 57.296; //Calculate the roll angle.

}

angle\_pitch = angle\_pitch \* 0.9996 + angle\_pitch\_acc \* 0.0004; //Correct the drift of the gyro pitch angle with the accelerometer pitch angle.

angle\_roll = angle\_roll \* 0.9996 + angle\_roll\_acc \* 0.0004; //Correct the drift of the gyro roll angle with the accelerometer roll angle.

pitch\_level\_adjust = (angle\_pitch - 0.1)\* 15; //Calculate the pitch angle correction.

roll\_level\_adjust = (angle\_roll - 0.2) \* 15; //Calculate the roll angle correction.

if (!auto\_level) { //If the quadcopter is not in auto-level mode

pitch\_level\_adjust = 0; //Set the pitch angle correction to zero.

roll\_level\_adjust = 0; //Set the roll angle correcion to zero.

}

//For starting the motors: throttle low and yaw left (step 1).

if (channel\_3 < 1050 && channel\_4 < 1070)start = 1;

//When yaw stick is back in the center position start the motors (step 2).

if (start == 1 && channel\_3 < 1050 && channel\_4 > 1425) {

start = 2;

green\_led(LOW); //Turn off the green led.

angle\_pitch = angle\_pitch\_acc; //Set the gyro pitch angle equal to the accelerometer pitch angle when the quadcopter is started.

angle\_roll = angle\_roll\_acc; //Set the gyro roll angle equal to the accelerometer roll angle when the quadcopter is started.

//Reset the PID controllers for a bumpless start.

pid\_i\_mem\_roll = 0;

pid\_last\_roll\_d\_error = 0;

pid\_i\_mem\_pitch = 0;

pid\_last\_pitch\_d\_error = 0;

pid\_i\_mem\_yaw = 0;

pid\_last\_yaw\_d\_error = 0;

}

//Stopping the motors: throttle low and yaw right.

if (start == 2 && channel\_3 < 1050 && channel\_4 > 1850) {

start = 0;

green\_led(HIGH); //Turn on the green led.

}

//The PID set point in degrees per second is determined by the roll receiver input.

//In the case of deviding by 3 the max roll rate is aprox 164 degrees per second ( (500-8)/3 = 164d/s ).

pid\_roll\_setpoint = 0;

//We need a little dead band of 16us for better results.

if (channel\_1 > 1508)pid\_roll\_setpoint = channel\_1 - 1508;

else if (channel\_1 < 1492)pid\_roll\_setpoint = channel\_1 - 1492;

pid\_roll\_setpoint -= roll\_level\_adjust; //Subtract the angle correction from the standardized receiver roll input value.

pid\_roll\_setpoint /= 3.0; //Divide the setpoint for the PID roll controller by 3 to get angles in degrees.

//The PID set point in degrees per second is determined by the pitch receiver input.

//In the case of deviding by 3 the max pitch rate is aprox 164 degrees per second ( (500-8)/3 = 164d/s ).

pid\_pitch\_setpoint = 0;

//We need a little dead band of 16us for better results.

if (channel\_2 > 1508)pid\_pitch\_setpoint = channel\_2 - 1508;

else if (channel\_2 < 1492)pid\_pitch\_setpoint = channel\_2 - 1492;

pid\_pitch\_setpoint -= pitch\_level\_adjust; //Subtract the angle correction from the standardized receiver pitch input value.

pid\_pitch\_setpoint /= 3.0; //Divide the setpoint for the PID pitch controller by 3 to get angles in degrees.

//The PID set point in degrees per second is determined by the yaw receiver input.

//In the case of deviding by 3 the max yaw rate is aprox 164 degrees per second ( (500-8)/3 = 164d/s ).

pid\_yaw\_setpoint = 0;

//We need a little dead band of 16us for better results.

if (channel\_3 > 1050) { //Do not yaw when turning off the motors.

if (channel\_4 > 1508)pid\_yaw\_setpoint = (channel\_4 - 1508) / 3.0;

else if (channel\_4 < 1492)pid\_yaw\_setpoint = (channel\_4 - 1492) / 3.0;

}

calculate\_pid(); //PID inputs are known. So we can calculate the pid output.

//The battery voltage is needed for compensation.

//A complementary filter is used to reduce noise.

//1410.1 = 112.81 / 0.08.

battery\_voltage = battery\_voltage \* 0.92 + ((float)analogRead(4) / 1410.1);

//Turn on the led if battery voltage is to low. In this case under 10.0V

if (battery\_voltage < 10.0 && error == 0)error = 1;

throttle = map(channel\_3,1000, 1950,1000,1650); //We need the throttle signal as a base signal.

if (start == 2) { //The motors are started.

if (throttle > 1800) throttle = 1800; //We need some room to keep full control at full throttle.

esc\_1 = throttle - pid\_output\_pitch + pid\_output\_roll - pid\_output\_yaw; //Calculate the pulse for esc 1 (front-right - CCW).

esc\_2 = throttle + pid\_output\_pitch + pid\_output\_roll + pid\_output\_yaw; //Calculate the pulse for esc 2 (rear-right - CW).

esc\_3 = throttle + pid\_output\_pitch - pid\_output\_roll - pid\_output\_yaw; //Calculate the pulse for esc 3 (rear-left - CCW).

esc\_4 = throttle - pid\_output\_pitch - pid\_output\_roll + pid\_output\_yaw; //Calculate the pulse for esc 4 (front-left - CW).

if (esc\_1 < 1100) esc\_1 = 1100; //Keep the motors running.

if (esc\_2 < 1100) esc\_2 = 1100; //Keep the motors running.

if (esc\_3 < 1100) esc\_3 = 1100; //Keep the motors running.

if (esc\_4 < 1100) esc\_4 = 1100; //Keep the motors running.

if (esc\_1 > 2000)esc\_1 = 2000; //Limit the esc-1 pulse to 2000us.

if (esc\_2 > 2000)esc\_2 = 2000; //Limit the esc-2 pulse to 2000us.

if (esc\_3 > 2000)esc\_3 = 2000; //Limit the esc-3 pulse to 2000us.

if (esc\_4 > 2000)esc\_4 = 2000; //Limit the esc-4 pulse to 2000us.

}

else {

esc\_1 = 1000; //If start is not 2 keep a 1000us pulse for ess-1.

esc\_2 = 1000; //If start is not 2 keep a 1000us pulse for ess-2.

esc\_3 = 1000; //If start is not 2 keep a 1000us pulse for ess-3.

esc\_4 = 1000; //If start is not 2 keep a 1000us pulse for ess-4.

}

TIMER4\_BASE->CCR1 = esc\_1; //Set the throttle receiver input pulse to the ESC 1 output pulse.

TIMER4\_BASE->CCR2 = esc\_2; //Set the throttle receiver input pulse to the ESC 2 output pulse.

TIMER4\_BASE->CCR3 = esc\_3; //Set the throttle receiver input pulse to the ESC 3 output pulse.

TIMER4\_BASE->CCR4 = esc\_4; //Set the throttle receiver input pulse to the ESC 4 output pulse.

TIMER4\_BASE->CNT = 5000; //This will reset timer 4 and the ESC pulses are directly created.

>>> **LED\_CONTROL**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//These functions handle the red and green LEDs. The LEDs on the flip 32 are inverted. That is why a Flip32 test is needed.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void red\_led(int8\_t level) {

if (flip32)digitalWrite(PB4, !level); //If a Flip32 is detected invert the output.

else digitalWrite(PB4, level); //When using the BluePill the output should not be inverted.

}

void green\_led(int8\_t level) {

if (flip32)digitalWrite(PB3, !level); //If a Flip32 is detected invert the output.

else digitalWrite(PB3, level); //When using the BluePill the output should not be inverted.

}

>>> **CALCULATE\_PID**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Subroutine for calculating pid outputs

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//The PID controllers are explained in part 5 of the YMFC-3D video session:

//https://youtu.be/JBvnB0279-Q

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void calculate\_pid(void){

//Roll calculations

pid\_error\_temp = gyro\_roll\_input - pid\_roll\_setpoint;

pid\_i\_mem\_roll += pid\_i\_gain\_roll \* pid\_error\_temp;

if(pid\_i\_mem\_roll > pid\_max\_roll)pid\_i\_mem\_roll = pid\_max\_roll;

else if(pid\_i\_mem\_roll < pid\_max\_roll \* -1)pid\_i\_mem\_roll = pid\_max\_roll \* -1;

pid\_output\_roll = pid\_p\_gain\_roll \* pid\_error\_temp + pid\_i\_mem\_roll + pid\_d\_gain\_roll \* (pid\_error\_temp - pid\_last\_roll\_d\_error);

if(pid\_output\_roll > pid\_max\_roll)pid\_output\_roll = pid\_max\_roll;

else if(pid\_output\_roll < pid\_max\_roll \* -1)pid\_output\_roll = pid\_max\_roll \* -1;

pid\_last\_roll\_d\_error = pid\_error\_temp;

//Pitch calculations

pid\_error\_temp = gyro\_pitch\_input - pid\_pitch\_setpoint;

pid\_i\_mem\_pitch += pid\_i\_gain\_pitch \* pid\_error\_temp;

if(pid\_i\_mem\_pitch > pid\_max\_pitch)pid\_i\_mem\_pitch = pid\_max\_pitch;

else if(pid\_i\_mem\_pitch < pid\_max\_pitch \* -1)pid\_i\_mem\_pitch = pid\_max\_pitch \* -1;

pid\_output\_pitch = pid\_p\_gain\_pitch \* pid\_error\_temp + pid\_i\_mem\_pitch + pid\_d\_gain\_pitch \* (pid\_error\_temp - pid\_last\_pitch\_d\_error);

if(pid\_output\_pitch > pid\_max\_pitch)pid\_output\_pitch = pid\_max\_pitch;

else if(pid\_output\_pitch < pid\_max\_pitch \* -1)pid\_output\_pitch = pid\_max\_pitch \* -1;

pid\_last\_pitch\_d\_error = pid\_error\_temp;

//Yaw calculations

pid\_error\_temp = gyro\_yaw\_input - pid\_yaw\_setpoint;

pid\_i\_mem\_yaw += pid\_i\_gain\_yaw \* pid\_error\_temp;

if(pid\_i\_mem\_yaw > pid\_max\_yaw)pid\_i\_mem\_yaw = pid\_max\_yaw;

else if(pid\_i\_mem\_yaw < pid\_max\_yaw \* -1)pid\_i\_mem\_yaw = pid\_max\_yaw \* -1;

pid\_output\_yaw = pid\_p\_gain\_yaw \* pid\_error\_temp + pid\_i\_mem\_yaw + pid\_d\_gain\_yaw \* (pid\_error\_temp - pid\_last\_yaw\_d\_error);

if(pid\_output\_yaw > pid\_max\_yaw)pid\_output\_yaw = pid\_max\_yaw;

else if(pid\_output\_yaw < pid\_max\_yaw \* -1)pid\_output\_yaw = pid\_max\_yaw \* -1;

pid\_last\_yaw\_d\_error = pid\_error\_temp;

}

>>> **CALIBRATE\_GYR0**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//This subroutine handles the calibration of the gyro. It stores the avarage gyro offset of 2000 readings.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void calibrate\_gyro(void) {

if (use\_manual\_calibration)cal\_int = 2000; //If manual calibration is used set cal\_int to 2000 to skip the calibration.

else {

cal\_int = 0; //If manual calibration is not used.

manual\_gyro\_pitch\_cal\_value = 0; //Set the manual pitch calibration variable to 0.

manual\_gyro\_roll\_cal\_value = 0; //Set the manual roll calibration variable to 0.

manual\_gyro\_yaw\_cal\_value = 0; //Set the manual yaw calibration variable to 0.

}

if (cal\_int != 2000) {

//Let's take multiple gyro data samples so we can determine the average gyro offset (calibration).

for (cal\_int = 0; cal\_int < 2000 ; cal\_int ++) { //Take 2000 readings for calibration.

if (cal\_int % 25 == 0) digitalWrite(PB4, !digitalRead(PB4)); //Change the led status every 125 readings to indicate calibration.

gyro\_signalen(); //Read the gyro output.

gyro\_roll\_cal += gyro\_roll; //Ad roll value to gyro\_roll\_cal.

gyro\_pitch\_cal += gyro\_pitch; //Ad pitch value to gyro\_pitch\_cal.

gyro\_yaw\_cal += gyro\_yaw; //Ad yaw value to gyro\_yaw\_cal.

delay(4); //Small delay to simulate a 250Hz loop during calibration.

}

red\_led(HIGH); //Set output PB3 low.

//Now that we have 2000 measures, we need to devide by 2000 to get the average gyro offset.

gyro\_roll\_cal /= 2000; //Divide the roll total by 2000.

gyro\_pitch\_cal /= 2000; //Divide the pitch total by 2000.

gyro\_yaw\_cal /= 2000; //Divide the yaw total by 2000.

manual\_gyro\_pitch\_cal\_value = gyro\_pitch\_cal; //Set the manual pitch calibration variable to the detected value.

manual\_gyro\_roll\_cal\_value = gyro\_roll\_cal; //Set the manual roll calibration variable to the detected value.

manual\_gyro\_yaw\_cal\_value = gyro\_yaw\_cal; //Set the manual yaw calibration variable to the detected value.

}

}

>>> **GYRO\_SETUP**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//In this part the various registers of the MPU-6050 are set.

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void gyro\_setup(void){

HWire.beginTransmission(gyro\_address); //Start communication with the MPU-6050.

HWire.write(0x6B); //We want to write to the PWR\_MGMT\_1 register (6B hex).

HWire.write(0x00); //Set the register bits as 00000000 to activate the gyro.

HWire.endTransmission(); //End the transmission with the gyro.

HWire.beginTransmission(gyro\_address); //Start communication with the MPU-6050.

HWire.write(0x1B); //We want to write to the GYRO\_CONFIG register (1B hex).

HWire.write(0x08); //Set the register bits as 00001000 (500dps full scale).

HWire.endTransmission(); //End the transmission with the gyro.

HWire.beginTransmission(gyro\_address); //Start communication with the MPU-6050.

HWire.write(0x1C); //We want to write to the ACCEL\_CONFIG register (1A hex).

HWire.write(0x10); //Set the register bits as 00010000 (+/- 8g full scale range).

HWire.endTransmission(); //End the transmission with the gyro.

HWire.beginTransmission(gyro\_address); //Start communication with the MPU-6050.

HWire.write(0x1A); //We want to write to the CONFIG register (1A hex).

HWire.write(0x03); //Set the register bits as 00000011 (Set Digital Low Pass Filter to ~43Hz).

HWire.endTransmission(); //End the transmission with the gyro.

}

>>> **INPUT\_CAPTURE\_MODE\_HANDLERS**

void handler\_channel\_1(void) { //This function is called when channel 1 is captured.

if (0b1 & GPIOA\_BASE->IDR >> 0) { //If the receiver channel 1 input pulse on A0 is high.

channel\_1\_start = TIMER2\_BASE->CCR1; //Record the start time of the pulse.

TIMER2\_BASE->CCER |= TIMER\_CCER\_CC1P; //Change the input capture mode to the falling edge of the pulse.

}

else { //If the receiver channel 1 input pulse on A0 is low.

channel\_1 = TIMER2\_BASE->CCR1 - channel\_1\_start; //Calculate the total pulse time.

if (channel\_1 < 0)channel\_1 += 0xFFFF; //If the timer has rolled over a correction is needed.

TIMER2\_BASE->CCER &= ~TIMER\_CCER\_CC1P; //Change the input capture mode to the rising edge of the pulse.

}

}

void handler\_channel\_2(void) { //This function is called when channel 2 is captured.

if (0b1 & GPIOA\_BASE->IDR >> 1) { //If the receiver channel 2 input pulse on A1 is high.

channel\_2\_start = TIMER2\_BASE->CCR2; //Record the start time of the pulse.

TIMER2\_BASE->CCER |= TIMER\_CCER\_CC2P; //Change the input capture mode to the falling edge of the pulse.

}

else { //If the receiver channel 2 input pulse on A1 is low.

channel\_2 = TIMER2\_BASE->CCR2 - channel\_2\_start; //Calculate the total pulse time.

if (channel\_2 < 0)channel\_2 += 0xFFFF; //If the timer has rolled over a correction is needed.

TIMER2\_BASE->CCER &= ~TIMER\_CCER\_CC2P; //Change the input capture mode to the rising edge of the pulse.

}

}

void handler\_channel\_3(void) { //This function is called when channel 3 is captured.

if (0b1 & GPIOA\_BASE->IDR >> 2) { //If the receiver channel 3 input pulse on A2 is high.

channel\_3\_start = TIMER2\_BASE->CCR3; //Record the start time of the pulse.

TIMER2\_BASE->CCER |= TIMER\_CCER\_CC3P; //Change the input capture mode to the falling edge of the pulse.

}

else { //If the receiver channel 3 input pulse on A2 is low.

channel\_3 = TIMER2\_BASE->CCR3 - channel\_3\_start; //Calculate the total pulse time.

if (channel\_3 < 0)channel\_3 += 0xFFFF; //If the timer has rolled over a correction is needed.

TIMER2\_BASE->CCER &= ~TIMER\_CCER\_CC3P; //Change the input capture mode to the rising edge of the pulse.

}

}

void handler\_channel\_4(void) { //This function is called when channel 4 is captured.

if (0b1 & GPIOA\_BASE->IDR >> 3) { //If the receiver channel 4 input pulse on A3 is high.

channel\_4\_start = TIMER2\_BASE->CCR4; //Record the start time of the pulse.

TIMER2\_BASE->CCER |= TIMER\_CCER\_CC4P; //Change the input capture mode to the falling edge of the pulse.

}

else { //If the receiver channel 4 input pulse on A3 is low.

channel\_4 = TIMER2\_BASE->CCR4 - channel\_4\_start; //Calculate the total pulse time.

if (channel\_4 < 0)channel\_4 += 0xFFFF; //If the timer has rolled over a correction is needed.

TIMER2\_BASE->CCER &= ~TIMER\_CCER\_CC4P; //Change the input capture mode to the rising edge of the pulse.

}

}

void handler\_channel\_5(void) { //This function is called when channel 5 is captured.

if (0b1 & GPIOA\_BASE->IDR >> 6) { //If the receiver channel 5 input pulse on A6 is high.

channel\_5\_start = TIMER3\_BASE->CCR1; //Record the start time of the pulse.

TIMER3\_BASE->CCER |= TIMER\_CCER\_CC1P; //Change the input capture mode to the falling edge of the pulse.

}

else { //If the receiver channel 5 input pulse on A6 is low.

channel\_5 = TIMER3\_BASE->CCR1 - channel\_5\_start; //Calculate the total pulse time.

if (channel\_5 < 0)channel\_5 += 0xFFFF; //If the timer has rolled over a correction is needed.

TIMER3\_BASE->CCER &= ~TIMER\_CCER\_CC1P; //Change the input capture mode to the rising edge of the pulse.

}

}

void handler\_channel\_6(void) { //This function is called when channel 6 is captured.

if (0b1 & GPIOA\_BASE->IDR >> 7) { //If the receiver channel 6 input pulse on A7 is high.

channel\_6\_start = TIMER3\_BASE->CCR2; //Record the start time of the pulse.

TIMER3\_BASE->CCER |= TIMER\_CCER\_CC2P; //Change the input capture mode to the falling edge of the pulse.

}

else { //If the receiver channel 6 input pulse on A7 is low.

channel\_6 = TIMER3\_BASE->CCR2 - channel\_6\_start; //Calculate the total pulse time.

if (channel\_6 < 0)channel\_6 += 0xFFFF; //If the timer has rolled over a correction is needed.

TIMER3\_BASE->CCER &= ~TIMER\_CCER\_CC2P; //Change the input capture mode to the rising edge of the pulse.

}

}

>>> **READ\_GYRO\_DATA**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//This part reads the raw gyro and accelerometer data from the MPU-6050

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void gyro\_signalen(void) {

HWire.beginTransmission(gyro\_address); //Start communication with the gyro.

HWire.write(0x3B); //Start reading @ register 43h and auto increment with every read.

HWire.endTransmission(); //End the transmission.

HWire.requestFrom(gyro\_address, 14); //Request 14 bytes from the MPU 6050.

acc\_y = HWire.read() << 8 | HWire.read(); //Add the low and high byte to the acc\_x variable.

acc\_x = HWire.read() << 8 | HWire.read(); //Add the low and high byte to the acc\_y variable.

acc\_z = HWire.read() << 8 | HWire.read(); //Add the low and high byte to the acc\_z variable.

temperature = HWire.read() << 8 | HWire.read(); //Add the low and high byte to the temperature variable.

gyro\_roll = HWire.read() << 8 | HWire.read(); //Read high and low part of the angular data.

gyro\_pitch = HWire.read() << 8 | HWire.read(); //Read high and low part of the angular data.

gyro\_yaw = HWire.read() << 8 | HWire.read(); //Read high and low part of the angular data.

gyro\_pitch \*= -1; //Invert the direction of the axis.

gyro\_yaw \*= -1; //Invert the direction of the axis.

acc\_y -= manual\_acc\_pitch\_cal\_value; //Subtact the manual accelerometer pitch calibration value.

acc\_x -= manual\_acc\_roll\_cal\_value; //Subtact the manual accelerometer roll calibration value.

gyro\_roll -= manual\_gyro\_roll\_cal\_value - 1 ; //Subtact the manual gyro roll calibration value.

gyro\_pitch -= manual\_gyro\_pitch\_cal\_value; //Subtact the manual gyro pitch calibration value.

gyro\_yaw -= manual\_gyro\_yaw\_cal\_value; //Subtact the manual gyro yaw calibration value.

}

>>> **TIMER\_SETUP**

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//In this file the timers for reading the receiver pulses and for creating the output ESC pulses are set.

void timer\_setup(void) {

Timer2.attachCompare1Interrupt(handler\_channel\_1);

Timer2.attachCompare2Interrupt(handler\_channel\_2);

Timer2.attachCompare3Interrupt(handler\_channel\_3);

Timer2.attachCompare4Interrupt(handler\_channel\_4);

TIMER2\_BASE->CR1 = TIMER\_CR1\_CEN;

TIMER2\_BASE->CR2 = 0;

TIMER2\_BASE->SMCR = 0;

TIMER2\_BASE->DIER = TIMER\_DIER\_CC1IE | TIMER\_DIER\_CC2IE | TIMER\_DIER\_CC3IE | TIMER\_DIER\_CC4IE;

TIMER2\_BASE->EGR = 0;

TIMER2\_BASE->CCMR1 = 0b100000001; //Register is set like this due to a bug in the define table (30-09-2017)

TIMER2\_BASE->CCMR2 = 0b100000001; //Register is set like this due to a bug in the define table (30-09-2017)

TIMER2\_BASE->CCER = TIMER\_CCER\_CC1E | TIMER\_CCER\_CC2E | TIMER\_CCER\_CC3E | TIMER\_CCER\_CC4E;

TIMER2\_BASE->PSC = 71;

TIMER2\_BASE->ARR = 0xFFFF;

TIMER2\_BASE->DCR = 0;

Timer3.attachCompare1Interrupt(handler\_channel\_5);

Timer3.attachCompare2Interrupt(handler\_channel\_6);

TIMER3\_BASE->CR1 = TIMER\_CR1\_CEN;

TIMER3\_BASE->CR2 = 0;

TIMER3\_BASE->SMCR = 0;

TIMER3\_BASE->DIER = TIMER\_DIER\_CC1IE | TIMER\_DIER\_CC2IE;

TIMER3\_BASE->EGR = 0;

TIMER3\_BASE->CCMR1 = 0b100000001; //Register is set like this due to a bug in the define table (30-09-2017)

TIMER3\_BASE->CCMR2 = 0;

TIMER3\_BASE->CCER = TIMER\_CCER\_CC1E | TIMER\_CCER\_CC2E;

TIMER3\_BASE->PSC = 71;

TIMER3\_BASE->ARR = 0xFFFF;

TIMER3\_BASE->DCR = 0;

TIMER4\_BASE->CR1 = TIMER\_CR1\_CEN | TIMER\_CR1\_ARPE;

TIMER4\_BASE->CR2 = 0;

TIMER4\_BASE->SMCR = 0;

TIMER4\_BASE->DIER = 0;

TIMER4\_BASE->EGR = 0;

TIMER4\_BASE->CCMR1 = (0b110 << 4) | TIMER\_CCMR1\_OC1PE |(0b110 << 12) | TIMER\_CCMR1\_OC2PE;

TIMER4\_BASE->CCMR2 = (0b110 << 4) | TIMER\_CCMR2\_OC3PE |(0b110 << 12) | TIMER\_CCMR2\_OC4PE;

TIMER4\_BASE->CCER = TIMER\_CCER\_CC1E | TIMER\_CCER\_CC2E | TIMER\_CCER\_CC3E | TIMER\_CCER\_CC4E;

TIMER4\_BASE->PSC = 71;

TIMER4\_BASE->ARR = 5000;

TIMER4\_BASE->DCR = 0;

TIMER4\_BASE->CCR1 = 1000;

TIMER4\_BASE->CCR1 = 1000;

TIMER4\_BASE->CCR2 = 1000;

TIMER4\_BASE->CCR3 = 1000;

TIMER4\_BASE->CCR4 = 1000;

pinMode(PB6, PWM);

pinMode(PB7, PWM);

pinMode(PB8, PWM);

pinMode(PB9, PWM);

}