Master's Thesis
MsC Degree in Industrial Engineering

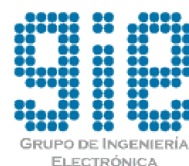
Design and Implementation of a Drone Quadcopter Using Low Cost Microcontrollers

Author: Pablo López Flores

Tutor: Ramón González Carvajal

Dept. Electronic Engineering
Higher Technical School of Engineering
University of Seville

Sevilla, 2020



Proyecto Fin de Máster
Máster Universitario en Ingeniería Industrial

Design and Implementation of a Drone Quadcopter Using Low Cost Microcontrollers

Autor:

Pablo López Flores

Tutor:

Ramón González Carvajal

Catedrático

Dpto. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020

Master’s Thesis: Design and Implementation of a Drone Quadcopter Using Low Cost Microcontrollers

Author: Pablo López Flores

Tutor: Ramón González Carvajal

The tribunal appointed to judge the work indicated above, which is composed of the following prefessors:

President:

Vocals:

Secretary:

Agree to grant him the qualification of:

Sevilla, 2020

The secretary of the tribunal

To my family and friends

Acknowledgements

I would like to express my gratitude to all the people who has supported me during the development of this project. First to my family and friends, who has been always by my side and encouraged me to keep working, even though the appearance of difficulties. Without them, this would not have been possible.

Thanks to Ramón for his support as well, always welcome to my doubts, that has made this project keep going until the end.

A special mention and acknowledgement to the GIE's Hardware Team (Grupo de Ingeniería Electrónica) of the University of Seville for printing the PCB's, especially to David Pablos Márquez.

And to all the others, that without the need, have also helped in any way.

As a final project that this is, I do not want to end my journey in the Engineering School of Seville without showing my sincere appreciation to all of my colleges. Also to the professors and other workers, which make possible for many people to become not only an engineer, but also a bigger person.

To all of them, thank you.

Abstract

The objective of this project was defined as the design and implementation of a drone quadcoper using low cost microcontrollers.

As it is described among the successive sections, a selection of the basic components was first made. The assembly of those served as a platform under which to develop an embedded control system, based on the microcontroller ESP8266. In conjunction to it, a separate remote controller was also designed and built to send the appropriate commands to the drone. In this case, the MSP430 microcontroller was used.

As a complete system to develop, the hardware was conceived first, for both remote and on-board controllers. Starting with the definition of the needed auxiliary components to interact with the microcontrollers, the PCB's were then designed.

After that, the firmware to be executed in the boards could be written, using the IDE's Code Composer Studio for the MSP430 and Arduino for the ESP8266, this last one mounted in the development board ESP-12E.

The main content of this project corresponds to the development of the drone's controller firmware, which consists of several tasks being executed sequentially. The most relevant is the one that calculates the control outputs, needed to maintain the drone under the desired attitude, implementing three Proportional Integral Derivative (PID) structures. To get initial PID parameters, a simple dynamic model of the drone was built with Matlab Simulink.

The complete system was tested statically to debug firmware and hardware issues and to verify its robustness and safety. Then, flight tests were arranged to finish the PID tuning and prove that the drone can operate in a real scenario.

Index

Acknowledgements **9**

Abstract **11**

Index **13**

1 Introduction **15**

2 Component Selection **16**

3 Hardware Development **25**

3.1. Remote Controller Hardware Design *25*

3.2. On-board Hardware Design *29*

4 Control Design and Matlab Simulations **32**

4.1 Control Design – PID *33*

4.2 Control Design – Extra features *36*

4.3 Control Signals *37*

4.4 Matlab Simulink Model *37*

4.4.1 Physical Model 38

4.4.2 Measurements Block 41

4.4.3 PID Controller Model 42

4.5 Simulated PID Tuning *43*

5 Firmware Development **47**

5.1 Remote Controller Firmware *47*

5.1.1 Peripherals Configuration 48

5.1.2 Low Power Mode 52

5.1.3 GPIO Read/write and ADC Measurements 53

5.1.4 Message Construction 54

5.2 Drone Controller Firmware *57*

5.2.1 Start-up State 57

5.2.2 Console State 59

5.2.3 Ready State 61

5.2.4 Running State 62

5.2.5 Finished and Error State 68

5.2.6 Common Tasks 69

6 Testing **75**

6.1 Static Testing *75*

6.2 Flight Testing *82*

7 Conclusions And Future Work **86**

List of Figures **87**

References **89**

Annex A – Matlab Code For Flight Test Analysis **90**

1 INTRODUCTION

The continuous growth of the drone's market, in conjunction to a passion for the embedded system design and for the aerospace field, has definitively served as the motivation to elaborate this project, which has enabled us to understand the principles of work of a drone in depth, and to show the capabilities of low cost microcontrollers. Meeting all the challenges that supposes the design of an on-board embedded control system, it has served as an opportunity to put together the knowledge and skills acquired during the course of the Master's degree.

The purpose is to design and implement a quadcopter using low cost microcontrollers, which was safe and could fly as stable as possible. In addition, the development of a remote controller was required.

As it is described in the successive sections, we first made the selection of the basic components. Then the hardware and firmware was developed. By using simulations of the system's dynamics, a first iteration of the controller was made. Finally, static and dynamic test were arranged in order to debug errors, finish the controller's tuning and validate the results, which were successfully compliant with the proposed objectives.

2 COMPONENT SELECTION

In this section, we are going to show the chosen drone's components. As common criteria, we chose the best suitable parts to build a race-size drone quadcopter, keeping the price as low as possible. The remote controller components are detailed in the section *3.1*.

Before going into the details of the actual devices chosen, we need to understand which elements do we need to build a drone. We can divide the complete system in tree groups: the drone's platform, the flight controller and the remote controller.

1) The drone's platform consists of a frame, where four motors are attached. In order to power them up, we need a battery, which supplies a power distribution board, where all the motors are connected. We also need to have an electronic speed controller (ESC) for each motor.

   Note that the necessity of the ESCs comes from the type of motors that we are using, like most of the drones do, brushless [11]. The reasons why we decided to choose this type of motors are:

   - Lifetime and maintenance, because they have no brushes.
   - Better speed and torque due to the absence of brushes. Brush friction increases with speed.
   - Efficiency. Brushless motors do not have contract resistance losses and the heat.
   - Wider speed range than brushed motors.
   - Better heat dissipation due to the construction compared to brushed motors.

2) The flight controller is in charge of setting the right power for each motor. The principle of work is simple. On the one hand, an Inertial Measurement Unit is used to estimate the orientation of the drone. On the other han, the desired orientation is received via radio. The duty of the flight controller is to calculate the correct power for each motor, so that the flight is stable and the actual orientation meets the desired one.

3) The remote controller acts as the interface between the drone and the user, who can send the desired orientation over time. To do that, we need a radio communication system.

Each needed elements are now described.

- **Flight Controller**

The chosen device to develop the flight controller has been the microcontroller ESP8266, mounted in the ESP-12E development board, produced by *Ai-Thinker*. This is a low cost microcontroller, whose most popular capability is Wi-Fi communications. Although we are not using this feature, given those listed in [1], this "highly integrated and durability MCU" (Microcontroller Unit) was considered suitable for the intended work.

As described in [2] "The ESP8266 is a low-cost Wi-Fi microchip, with a full TCP/IP stack and microcontroller capability, produced by Espressif Systems in Shanghai"

In the next image, we can see the pin out.

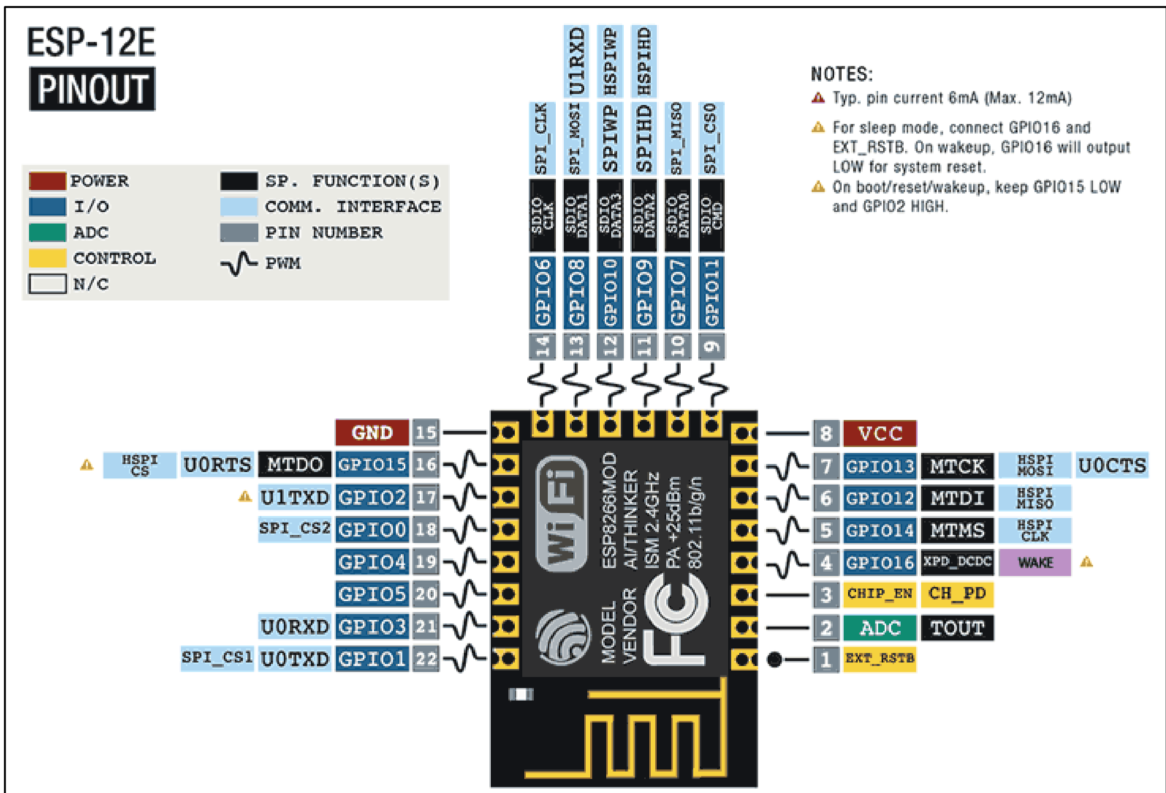


Figure 2.1. ESP8266 Pinout

And the key features in (from [1]):

	ESP-12E
Microcontroller	ESP8266
ESP8266 Processor	L106 32-bit RISC @80MHz
ESP8266 Memory	- 32 KiB instruction RAM - 32 KiB instruction cache RAM - 80 KiB user-data RAM - 16 KiB ETS system-data RAM
ESP8266 Features	- 16 GPIO - IEEE 802.11 b/g/n Wi-Fi - SPI - I²C (software implementation)[6] - I²S interfaces with DMA (sharing pins with GPIO) - UART on dedicated pins, plus a transmit-only UART can be enabled on GPIO2

	- 10-bit ADC (successive approximation ADC)
Number of GPIO available on ESP-12E	10
External QSPI Flash Memory	4 MiB
TTL-to-USB adapter	CH340G
Power Supply	4B2X (Input: 5V / Output: 3.3V)
Weight	5.8 g
Unit Price	2.63 €

The programming software used for this project has been the open source Arduino SDK (Software Development Kit).

- **Frame**

Spidex 220 FPV Drone By Quaternium

Wheelbase	220mm
Weight	95g
Unit Price	3.24€

- **Motors**

Turnigy D2206-2300KV 31g Brushless Motor CCW

Turnigy D2206-2300KV 31g Brushless Motor CW

RPM/V	2300kv
Voltage	2~4s LiPoly (7.4~14.8v)
Output	<310W
Max Current	23A
Suggested prop	5045~6045
Thrust	880g (5045 prop/4s)
Efficiency	<3.3 g/W
ESC	30A
Shaft	M5 CCW
Motor Mount Holes	M3 x 16/19mm
Dimensions	28 x 19.5mm
Weight	31g
Unit Price (x4 used)	7.52 €

- **ESCs (Electronic Speed Controllers)**

Turnigy Multistar BLheli_32 ARM 21A 2g Race Spec ESC 2~4S (OPTO)

Constant Current	21A
Cells	2~4S
Input Voltage	8.4v ~ 16.8v
BEC	None (opto only)
MCU	Arm Cortex-M0
Timing	Auto
Frequency	48MHz
Programmable	Yes
PCB Size	25 x 12 x 5mm
Weight	2g
Unit Price (x4 used)	6.26€

- **Radio Modules**

Ebyte E34-2G4D20D

Frequency	2.4 – 2.518 GHz
Power	10 – 20 dBm
Receiving sensitivity	-102 dBm
Air data rate	Self-adapted baud rate
Baud rate	1200 – 115200
Manufacturer Tested Distance	2 km (In open and clear air, with maximum power, 5dBi antenna gain, height of 2m.)
Interface	UART
Weight	6.3 g
Unit Price (x3 used)	4.31 €

Note that the third radio module was used as a comminicaion sniffer.

- **Antenna**

Ebyte TX2400-JK-11

Frequency	2.4GHz
Interface	SMA-J
Resistance	50Ω
Gain	2.5dB
Material	Rubber

Length	90mm
Weight	8g
Unit Price (x3 used)	4.46 $

- **IMU**

"MPU-9250 is a multi-chip module (MCM) consisting of two dies integrated into a single QFN package. One die houses the 3-Axis gyroscope and the 3-Axis accelerometer. The other die houses the AK8963 3-Axis magnetometer from Asahi Kasei Microdevices Corporation. Hence, the MPU-9250 is a 9-axis MotionTracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor™ (DMP) all in a small 3x3x1mm package available as a pin-compatible upgrade from the MPU-6515".

Source: [4].

	MPU-9250
	General Properties
Interface	I^2C
Consumption	3.5mA operating current when all 9 motion sensing axes and the DMP are enabled
Vdd	2.4 – 3.6V
Further features	- User-programmable digital filters - 512 byte FIFO buffer
	Accelerometer
Programmable range	±2g, ±4g, ±8g, ±16g
ADC resolution	16-bit
Normal operating current	450 µA
	Gyroscope
Programmable range	±250, ±500, ±1,000, ±2,000°/sec
ADC resolution	16-bit
Operating current	3.2 mA
Sleep mode current	8 µA
	Hall Sensor
ADC Resolution	14-bit (0.6 µT/LSB)

Full scale measurement range	±4800 µT
Weight	2.5 g
Unit Price	3.60 €

- **Battery**

Turnigy 1000mAh 3S Lipo 20C

Capacity	1000mAh
Configuration	3S1P / 11.1v / 3CELL
Discharge	20C Constant / 30C Burst
Weight	87g
Dimensions	77 x 33 x 20 mm
Plug	JST-XH
Unit Price	6.32 €

- **Distribution board:**

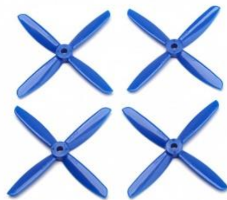
HobbyKing Mini Power Distribution Board

Amps	50 A
Size	36 x 36 x 1.5mm
Pads	2 input through holes, 8postive and 8negitive through hole outputs
Weight	4g
Unit Price	1.09€

- **Propellers**

Dalprops Bull Nose 4045 Propellers CW/CCW Set (2 pairs)

Size	4045 (4.0” x 4.5”)
Hub	5mm
Material	Glass reinforced polycarbonate
Weight	3.1g
2 Pairs Price	0.49 €



Dalprop Q4045 Bull Nose 4 Blade Propellers CW/CCW Set (2 pairs)

Size	4045 (4.0" x 4.5")
Color	Blue
Hub	5mm
Material	High quality flexible composite
Blades	4
2 Pairs Price	1.34 €

- **Propeller Nuts**

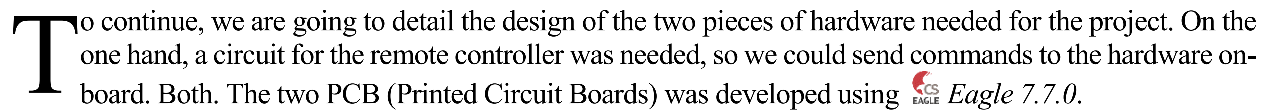
Aluminum Low Profile Nyloc Nut M5 Silver CW and CCW

Hub	5mm
Unit Price (x4 used)	0.275 €

- **Total amount**

Adding up all the the prices, the total is €100.42

3 HARDWARE DEVELOPMENT



To continue, we are going to detail the design of the two pieces of hardware needed for the project. On the one hand, a circuit for the remote controller was needed, so we could send commands to the hardware on-board. Both. The two PCB (Printed Circuit Boards) was developed using *Eagle 7.7.0*.

3.1. Remote Controller Hardware Design

The Remote Controller (RC) hardware has been based on the MS430 microcontroller from Texas Instruments. In particular, we have used the MSP430G2553 (PDIP format).

As described in *Component Selection*, this microcontroller belongs to the MSP430G2x53 series, which are:

"*ultra-low-power mixed signal microcontrollers with built-in 16- bit timers, up to 24 I/O capacitive-touch enabled pins, a versatile analog comparator, and built-in communication capability using the universal serial communication interface. In addition the MSP430G2x53 family members have a 10-bit analog-to-digital (A/D) converter*". [5].

Other common characteristics are: 16MHz MCU, 16KB flash and 512B SRAM.

Continuing now with the description on the PCB components, to provide power to the circuit, we have a 1.5V batteries holder, with four of them (6V). The voltage is reduced to 3.3V using the voltage regulator LM1117.

To generate the user inputs, we have a couple of two-axis gimbals. Each of them consists of two potentiometers, plus a push button. Other two additional buttons have been added. Note that only one of the buttons is used.

Those potentiometers are read using the inputs of the internal 10-bit converter. Prior to the microcontroller's input, we have an adaptation circuit based on a low pass RC filter and a voltage follower circuit, using the Op-Amp LM6144. The buttons, connected to a pull-up resistor from one side and to ground in the other, are directly read using the GPIOs.

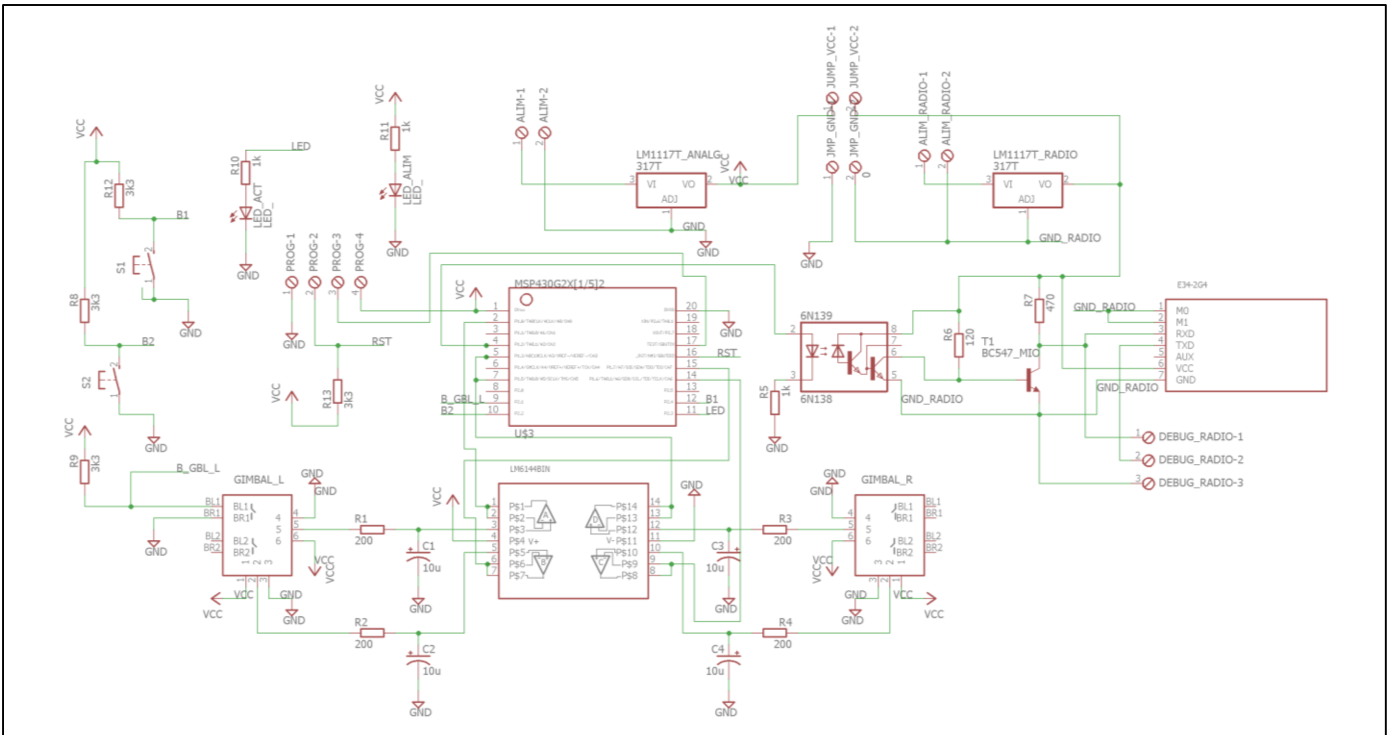


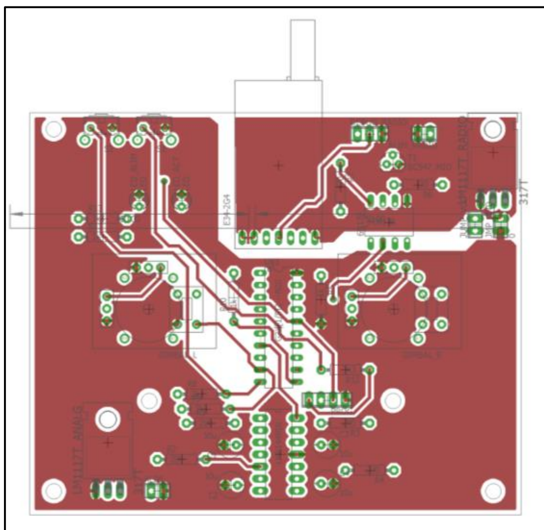
Figure 3-1. Remote Controller Circuit Schematic.

The communication module used is the *Ebyte E34-2G4D20D*, which interfaces via UART. The supply circuit of this module has been separated from the MSP430 one, with the possibility of connecting them via two jumpers. By doing it, we can reduce the interferences between the radio module and the analogic readings.

Hence, the transfer of information between the two sub-circuits is performed with an opto-isolator (6N139).

Two extra separate buttons have been added to provide further control capabilities.

The PCB layout is shown in the following images:



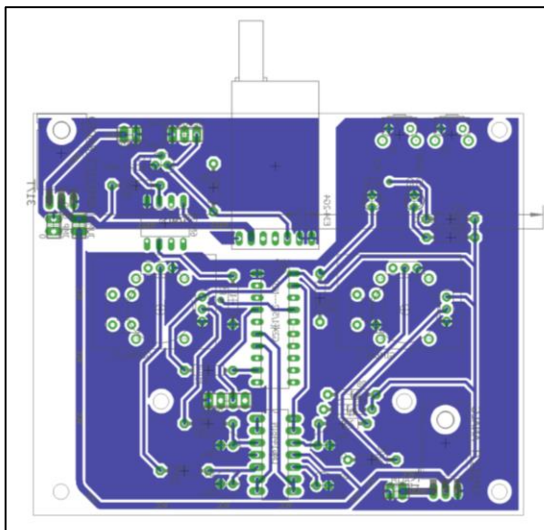


Figure 3-2. Remote Controller Layout. Top view in red and bottom view in blue.

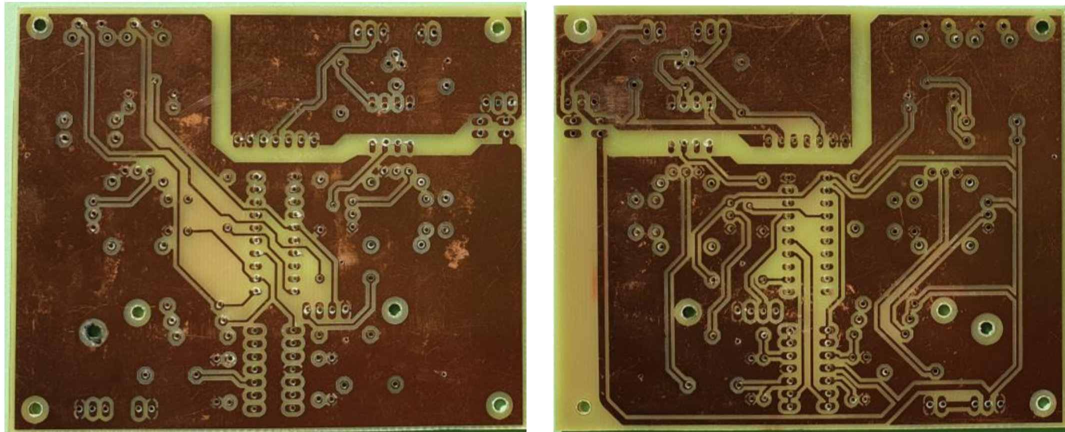


Figure 3-3. Printed Circuit. Top layer on the left. Bottom layer on the right.

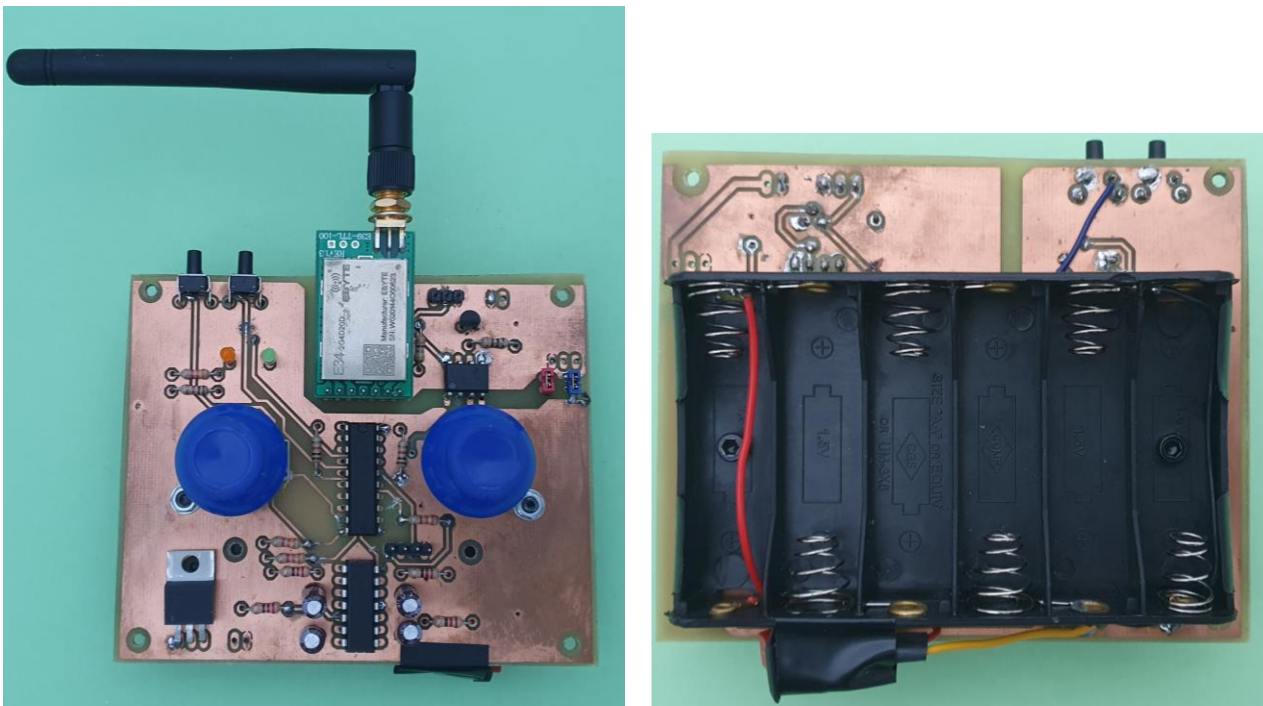


Figure 3-4. Assembled Remote Controller. Top view on the left. Bottom view on the right.

3.2. On-board Hardware Design

The on-board hardware is not that complicated as the previous one, as we just need a circuit that connect the components, without the need of any analogic design

As said in *Component Selection,* the chosen microcontroller has been the ESP8266, mounted on the ESP-12E development board. This communicates to the also described MPU-9250 IMU and Ebyte E34-2G4D20D radio module.

A buzzer has been added to let the user know the status of the drone.

The connections are shown in the figure below.

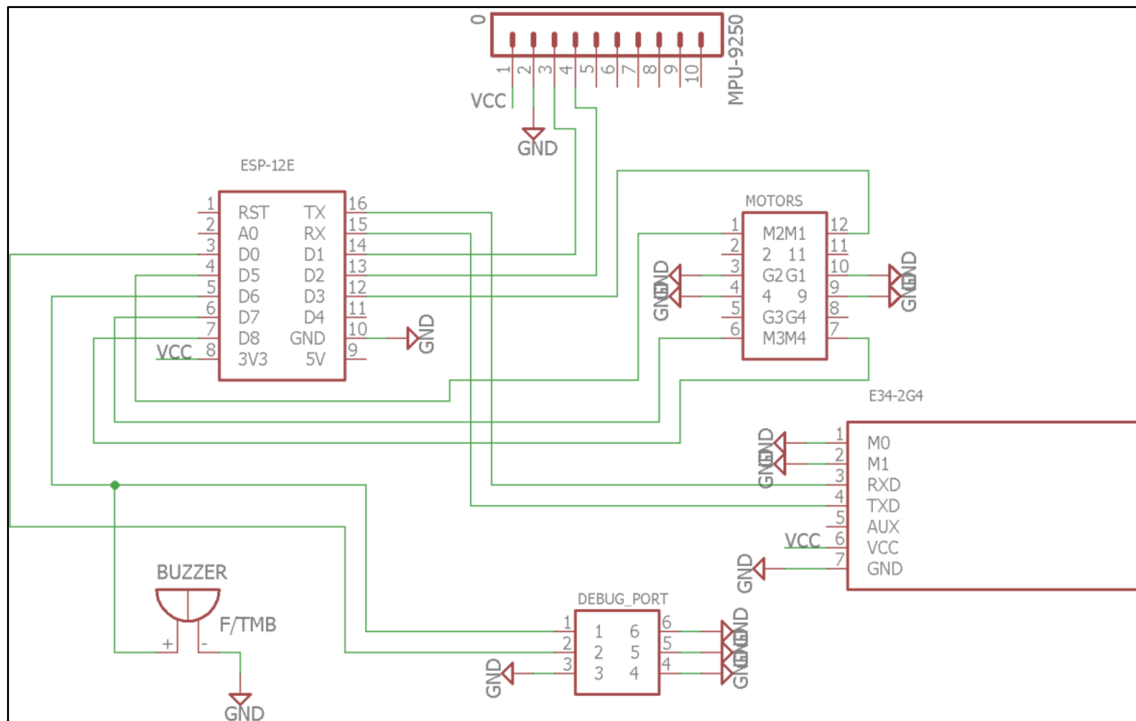


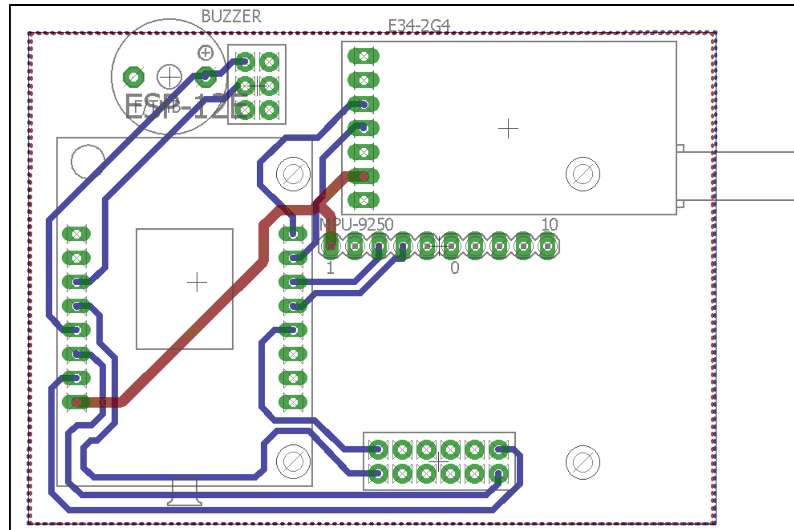
Figure 3-5. On-board Circuit Schematic.

The layouts of the PCB in the following images.

Note the four holes for mounting purposes, which match with those in the centre of the frame.

The IMU is the most noise sensitive part of the circuit, so the MPU-9250 has been placed in the centre of the holes, as far away as possible from the motors and ESC's, which generates a high frequency Electromagnetic Interference (EMI). This is further discussed in the section *Static Testing*.

The other components have been located as near as possible from each other so save space.



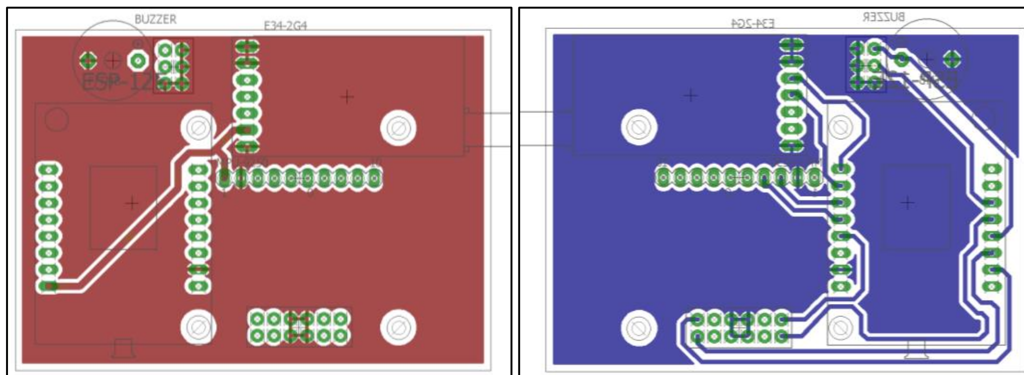


Figure 3-6. On-board hardware Layout. We can see the top view in red and the bottom one in blue.

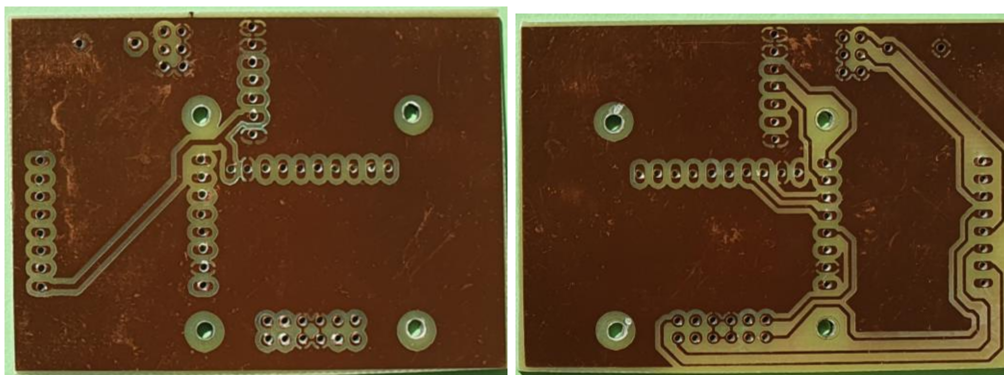


Figure 3-7. On-board Circuit Printed Board. Top layer on the left and bottom layer on the right.

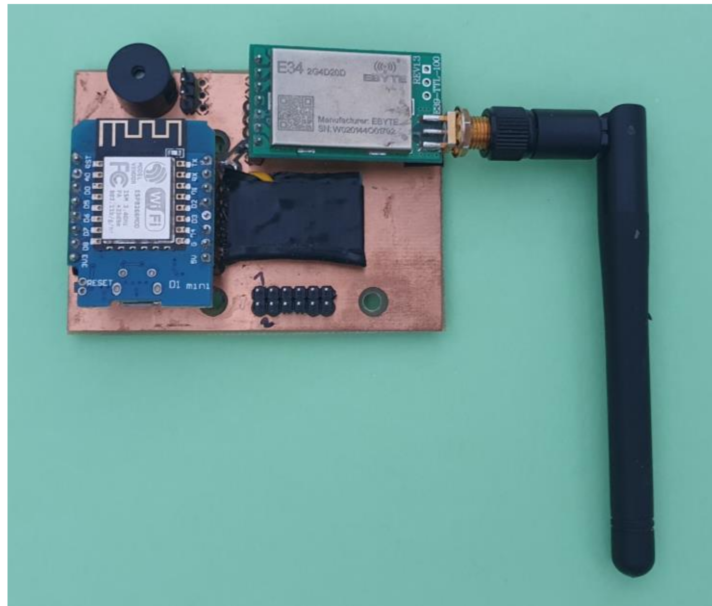


Figure 3-8. On-board Circuit Assembly.

And in the next image the complete drone assembly is shown:

Figure 3-9. Drone Assembly.

4 CONTROL DESIGN AND MATLAB SIMULATIONS

Unlike a plane, the flight stability of a drone quadcoper depends to great extent on the control inputs. On the one hand, thanks to the aerodynamic forces in the traditional planes, those can fly stable without the need of any control input, as long as the air speed is enough. That is not the case of the quadcopters, which only depends on the forces of each motor in order to keep in the air.

Many factors can affect to the stability of the flight, such as the changing wind speed and direction, displacement of the centre of gravity of the drone, or small discrepancies in the speed/trust ratio of each motor and propeller. For this reason, it would be almost impossible to control it only with the inputs on the radio transmitter. The need of an internal flight controller that takes care of all of those source errors is unquestionable.

In the following image, we can see the three angles that we need to deal with: Pitch, Roll and Yaw. The objective of our controller is to maintain the desired attitude of the drone, by keeping each of them under the desire values.

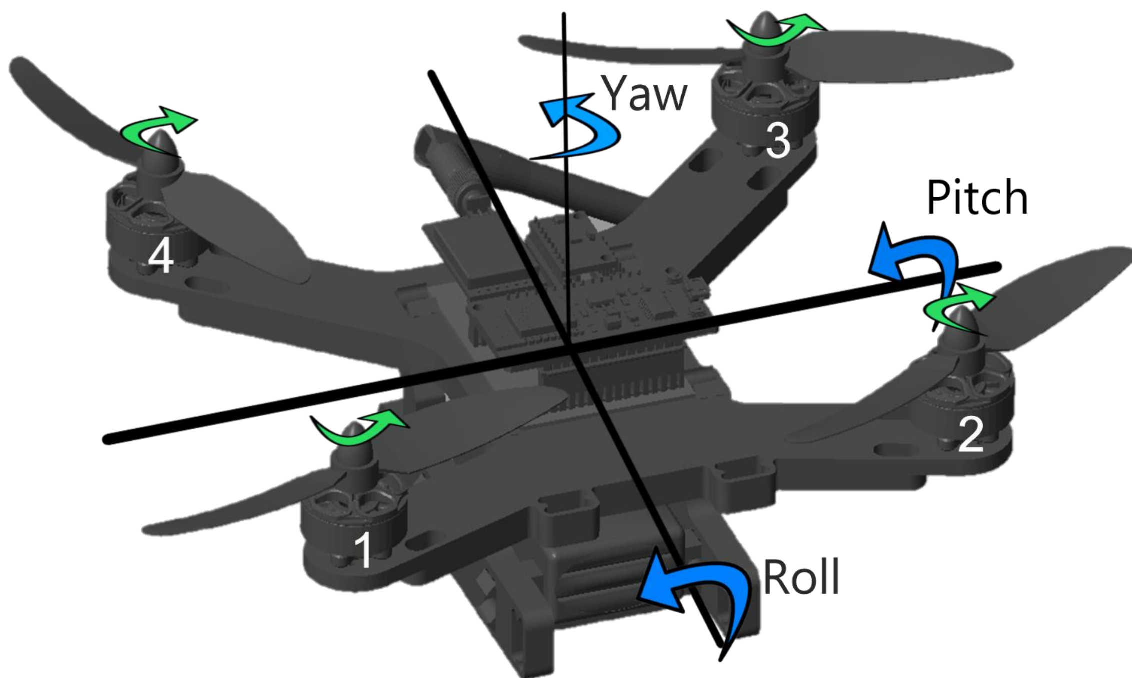


Figure 4-1. In blue: angles to control. In green: direction of rotation of each motor.

To do that, we need to calculate the correct motor's inputs based on the angle readings. As already said, we are using an IMU (Inertial Measurement Unit), which provides us with accelerations and angular velocities. That information is processed to estimate the instantaneous value of each angles and the rate of change.

The attitude estimation has been implemented using an existing open source C++ library, RTIMULib (2015).

Once we know how the drone is moving, we need to act on the motors to correct the deviations in the trajectory. Looking at the previous image, we see the positive direction of the angles. Therefore, we can define the needed actions for the angles to change:

- To get a positive pitch, the power of the motors 1 and 2 need to be higher than 3 and 4.
- To get a positive roll, the power of the motors 2 and 3 need to be higher than 1 and 4.
- To get a positive yaw, the power of the motors 2 and 4 need to be higher than 1 and 3.

If we want the altitude to remain the same when turning the drone in one of the three axes, we need that the overall power remains unchanged. For this reason the correct pitch, roll and yaw movements are achieved by increasing the power in two of the motors and reducing the same amount in the other two, as per the pointed above.

As is logical, to increase the altitude without any change in angle, the power needs to be increased the same amount for all of them.

Regarding the yaw movement, we make use of an angular momentum, produced by the difference of speed of each pair of motors (and propellers). Note that two of the motors rotates clockwise (2 and 4) and two rotates counter-clockwise (1 and 3). Thank to this, the torque forces counteract each other; ideally, if rotating all at the same speed, the quadcopter will not yaw. Furthermore, the position of each of them is key. Only with those defined in the previous image, we can dissociate the yaw movement from the other two: if increasing the speed of the motors 1 and 3 and decreasing 2 and 4 (or vice versa), the drone will not pitch or roll.

4.1 Control Design – PID

The chosen control architecture has been an independent *PID* (Proportional Integral Derivative) controller for each angle. Hence, three PID controllers have been implemented.

As said in [6], "*a proportional–integral–derivative controller (PID controller or three-term controller) is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an error value e(t) as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively), hence the name*".

In our specific case, the setpoints are the three angles (Pitch, Roll, Yaw), and the process variables are the measurements of those angles, collected by the IMU.

Although the next image corresponds to the Matlab simulations, explained in the next section, we can use it visualize the controller architecture. Note how the independent PIDs are combined.

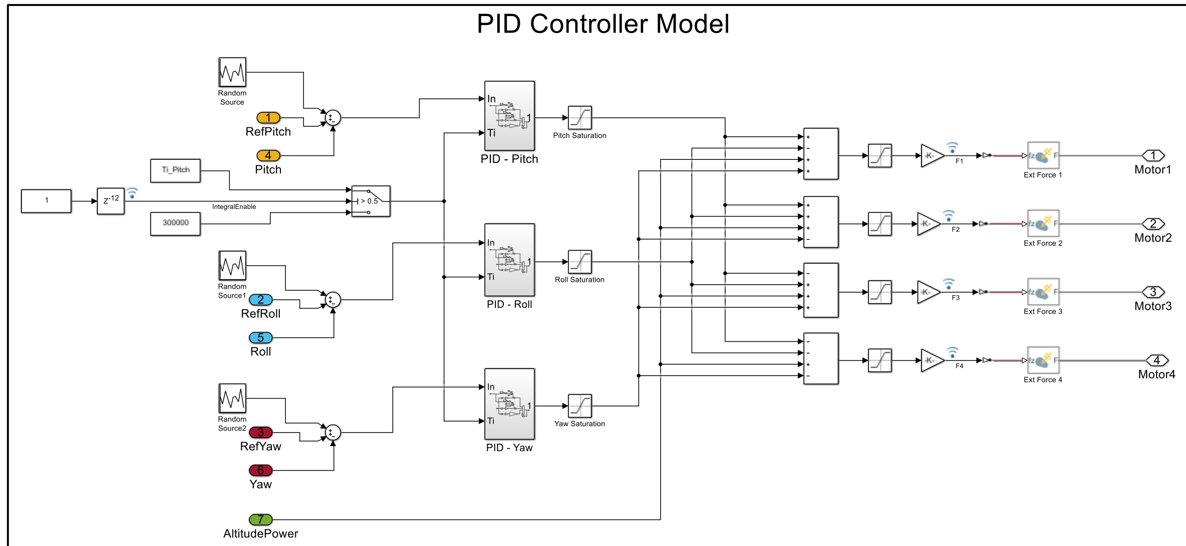


Figure 4-2. PID Controller Model – Combination of the three controllers.

On the left, we see the calculation of the errors. For simulation purposes, a random source was added to the Matlab model to simulate noise in the sensors.

In the centre, we see the PID modules. Note that there are two inputs: the error and an enable signal for the "I" component.

On the right, we see the merger of the PID outputs.

Following the logic of the previous section,

- The M1 power would be: +main power +pitch -roll -yaw corrections.
- The M2 power would be: +main power +pitch +roll +yaw correction.
- The M3 power would be: +main power -pitch +roll -yaw corrections.
- The M4 power would be: +main power -pitch -roll +yaw corrections.

For pitch and roll, the PID inputs are the angles (in degrees), whereas the yaw movement is controlled based on the angular velocity (in degrees per second).

- **PID Controller discretisation**.

The expression of the output of a continuous PID controller $u(t)$ is, from [7]:

$$u(t) = Kp\left(e(t) + \frac{1}{Ti}\int_0^t e(\tau)d\tau + Td\frac{de(t)}{dt}\right) \quad (4\text{–}1)$$

Where Kp is the proportional gain, $e(t)$ is the error, Ti is the integral time constant and Td is the derivative time constant.

As we are dealing with a real system, we need to discretise the controller, by approximating the error curve. The best way to do that is using the trapezoidal method.

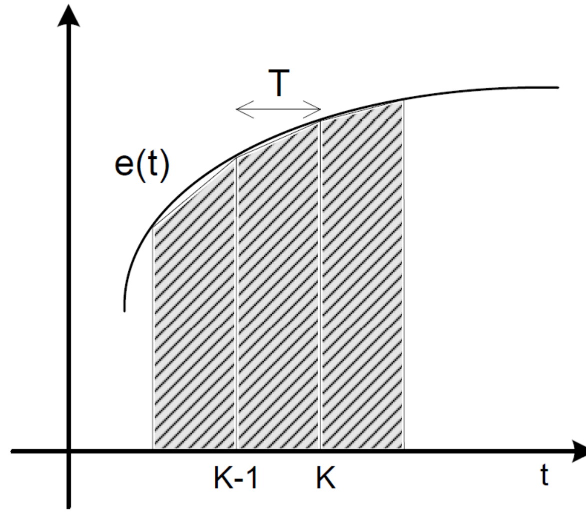


Figure 4-3. Discretisation of the error curve.

The approximation of the integral is:

$$\int_0^t e(\tau)d\tau = \sum_{i=1}^{k}\left(e\left(i-1+\frac{e(i)-e(i-1)}{2}\right)T\right) = \sum_{i=1}^{k} T\frac{e_i+e_{i-1}}{2} \tag{4–2}$$

So that,

$$u_k - u_{k-1} = q_0 e_k + q_1 e_{k-1} + q_2 e_{k-2} \tag{4–3}$$

Where,

$$\begin{gathered} q_0 = Kp\left(1+\frac{T}{2Ti}+\frac{Td}{T}\right) \\ q_1 = Kp\left(\frac{T}{2Ti}-1+\frac{2Td}{T}\right) \\ q_1 = Kp\left(\frac{Td}{T}\right) \end{gathered} \tag{4–4}$$

T is the sample time of the system. Given the processing time restrictions of our controller, the sample time used is 15ms (66.67 Hz).

Now, the objective is to calculate the best suitable PID parameters, so that the controller interacts properly with the system, i.e. stabilising it.

In order to calculate the controller's parameters, first, a Matlab Simulink model of the drone has been constructed to get initial values, based on the theoretical physical behaviour. A set field test PID tuning was later performed to adjust them further, based on the actual performance

4.2 Control Design – Extra features

Apart from the PID implementation, we also needed to add an extra feature to the basic controller.

We have to keep in mind how the integral component of the controller works. Given an error over time, the integral part arises by adding a correction to the current control outputs. This is good to erase any small errors in the desired angles when the drone is flying, as the other components (P or D) would not be very reactive to them.

However, this could cause problems when the error over time does not decrease. As said before, a correction is added when an error over time exists. Therefore, if that error keeps for a long period, it would result in a large integral correction.

This is the case when the drone is on the ground. In this situation, the angle values will keep constant, not necessarily equal to the reference, so the error over time will be very big. If we try to fly the drone with such a large initial correction, the quadcopter would destabilize as soon as the flight starts.

To solve that we have added an extra feature, that allows us to disable the integral part until the errors over time can be reduced, i.e. until the drone is in the air.

In addition, when simulating this feature in Matlab, we have seen that by doing that, the drone response improves, as we observe less overshoot when delaying the time when the integral part enables. In the next image we can see a couple of simulations, where we compare the pitch response when the integral part is delayed (in blue) and not (in red).

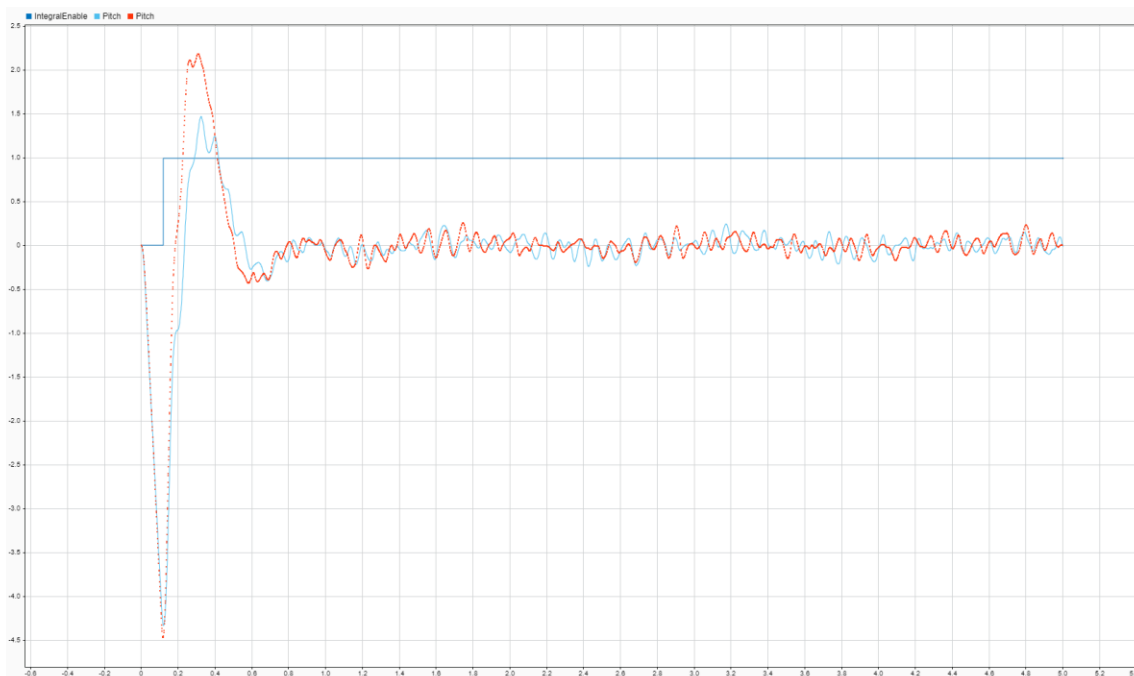


Figure 4-4. Comparison of the pitch response when the integral part is delayed (in blue) and not (in red).

4.3 Control Signals

As said in *Component Selection*, the chosen motors has been the *Turnigy D2206-2300KV 31g Brushless Motors*, plus *Turnigy Multistar BLheli_32 ARM* ESC's. Thanks to the ESC's firmware, we do not need to worry about the actual tree-phase power input signals for the motors, but send the desired speed to the ESC's. This is done using a PWM signal of 150 Hz.

The minimum set point (velocity = zero) is interpreted when the duty cycle is 15% and maximum when 30%.

4.4 Matlab Simulink Model

As said above, a Matlab Simulink model have been designed to get initial parameters of the PID controllers. In this section, we are describing in detail how the model has been done.

The model consists of three main parts, as seen in the next figure:

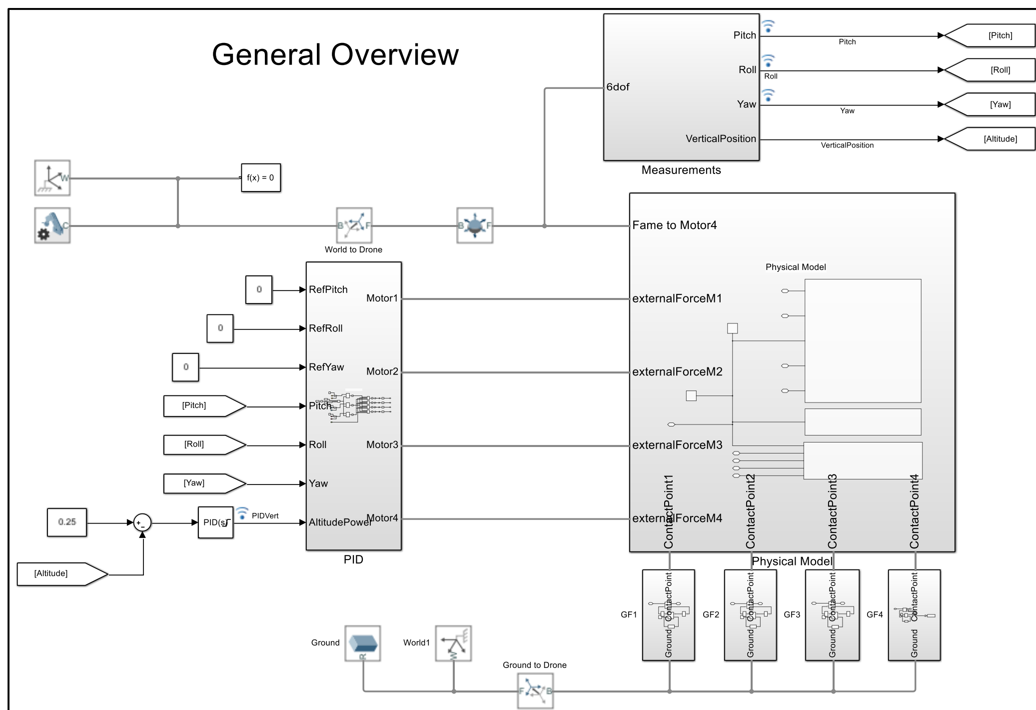


Figure 4-5. General overview of the Matlab Simulink Model

1) The Physical Model, at the right. This module simulates the physics of the drone. Given certain forces applied on each motor over time, the position and orientation in relation to the *World* is calculated. To implement that we have used the *Simscape Multibody* solids, joints and forces. At the bottom, we can see the join between the drone model and the ground.
2) Measurements block, at the top. In this module, we collect the movement measurements, which are the inputs for the PID module.
3) PID Controller Model, at the left. Based on the measurements, this block implement the PID calculations.

4.4.1 Physical Model

Inputs: Forces from the motors and ground. **Output:** Position and speed of the drone.

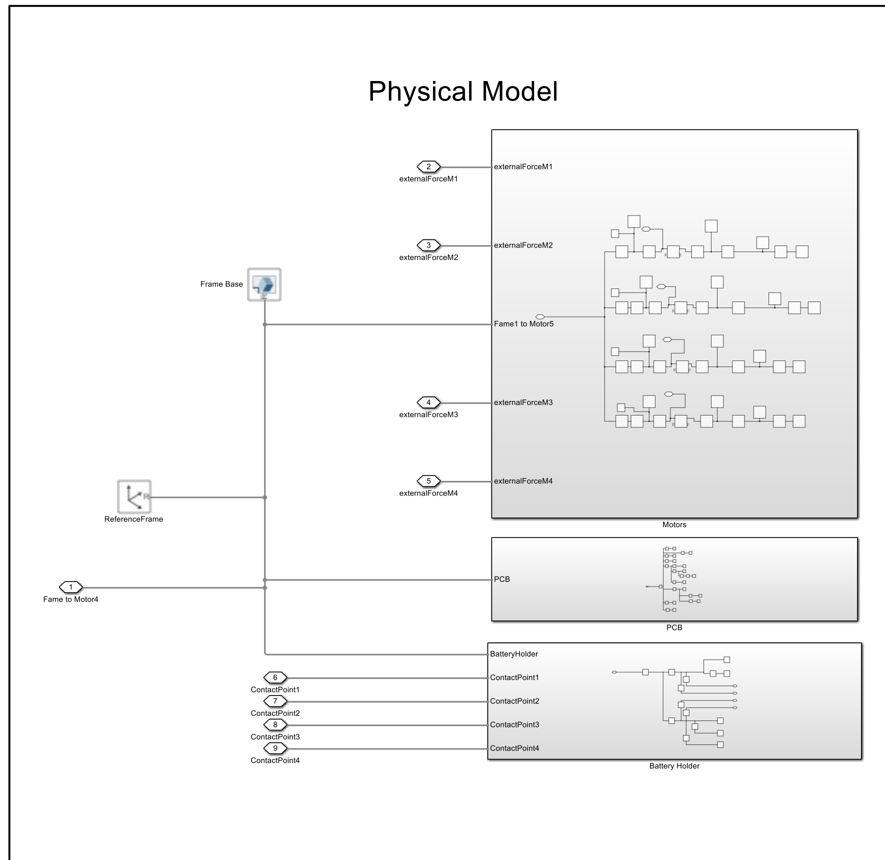


Figure 4-6. Simulink – Physical Model

The physical model has been created using the *Simscape Multibody* toolbox. In relation to a Reference Frame, we have added all the parts that make up the drone: The Base, Motors, Hardware and Battery Holder. The source of each CAD model is https://grabcad.com/.

Then, defining the weight or density information of each solid, the toolbox is able to calculate the centre of gravity and the dynamics of the drone, based on the applied input forces.

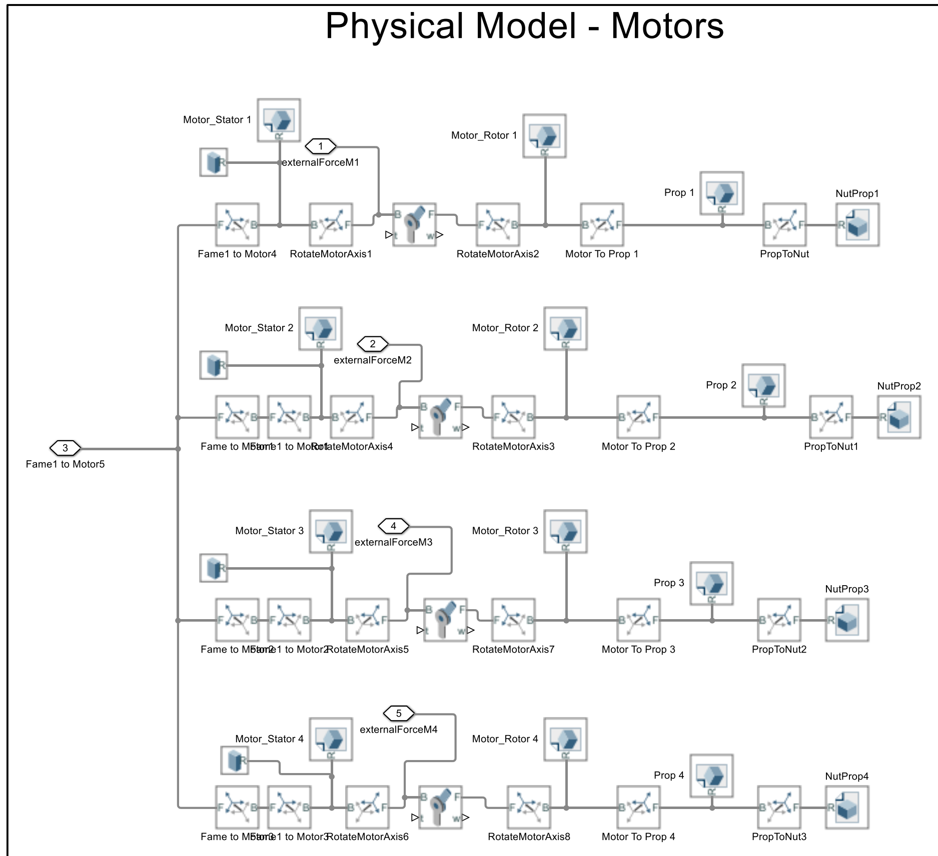


Figure 4-7. Simulink – Physical Model. Motors

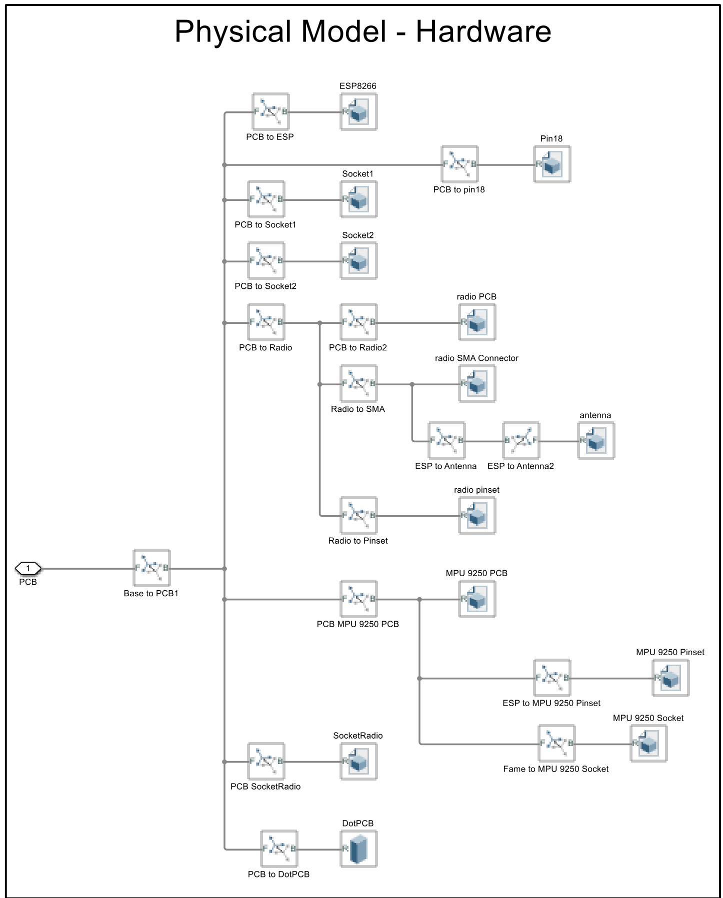


Figure 4-8. Simulink – Physical Model. Hardware

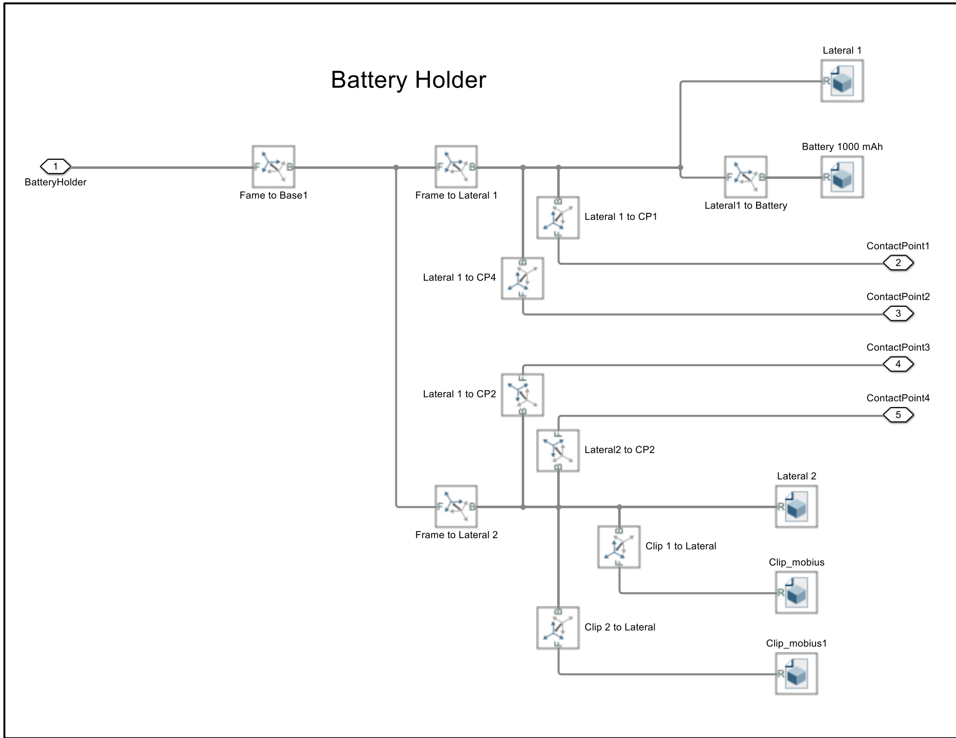


Figure 4-9. Simulink – Physical Model. Battery Holder

The visual result of all of those components can be seen in the next figure.

Figure 4-10. Simulink – Visual output of the physical model

4.4.2 Measurements Block

Input: 6-degree of freedom attribute from the physical model. **Output**: Pitch, Roll, Yaw and Vertical Position. This block is based on [8].

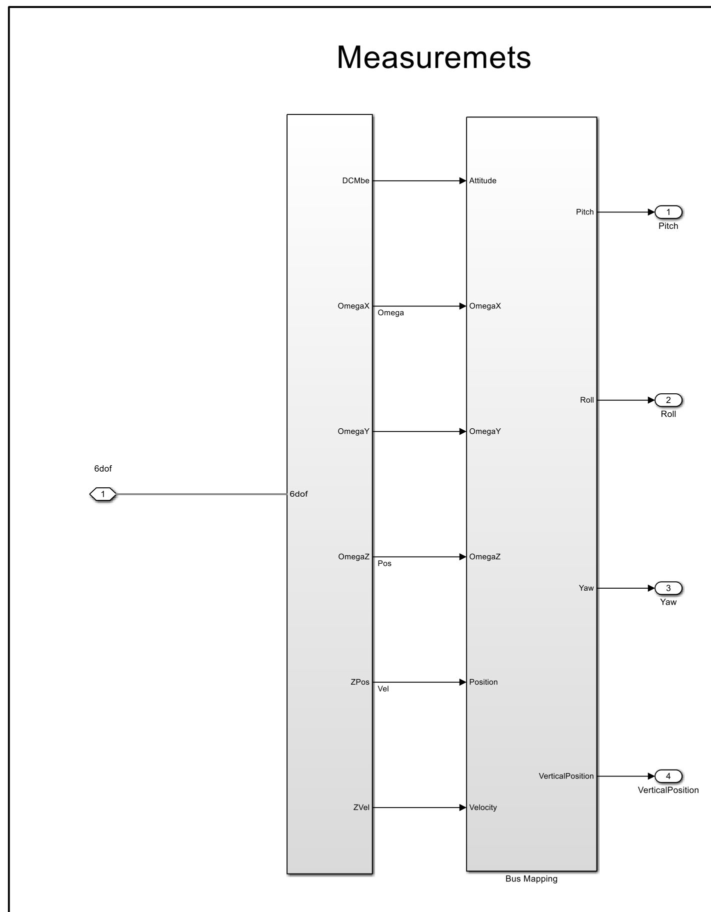


Figure 4-11. Simulink – Measurements Block

4.4.3 PID Controller Model

Input: Angles Measurements. **Output**: forces for each motor.

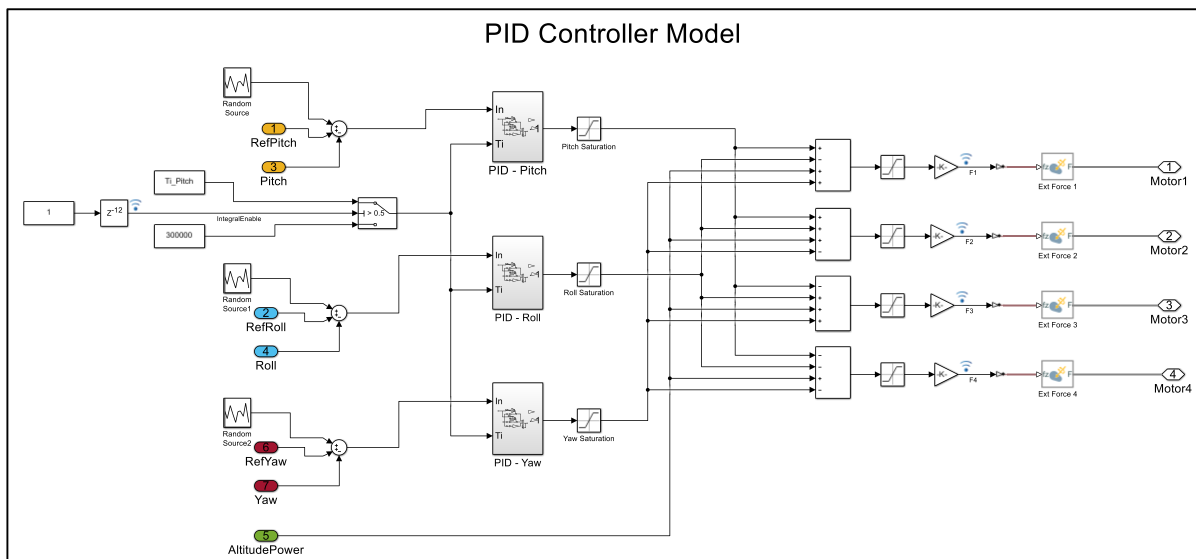


Figure 4-12. Simulink – PID Controller Model

Here we need to clarify that an assumption has been made, as we consider that the thrust force produced by each motor is a linear dependence of the control input, provided by the PID's.

Although the relationship between the motor's control inputs and the overall thrust produced by the attached propellers is not linear, we have consider it to be, so the model can be simplified. As we are using it as a starting point in the design, this assumption can be acceptable.

So as we can see in the following image, the thrust is simulated by using *External Force Simscape* modules. Those produce a signal linearly related to the input [Fmin, Fmax] in Newton, which is the PID outputs [0..1]. The proportionality constants are defined in an initialisation Matlab file, based on the motor's datasheet: we expect a maximum thrust of around 880 grams (5045 prop/4s).

In the following figure, we can see the PID implementation, directly based on the previous formula (4-3).

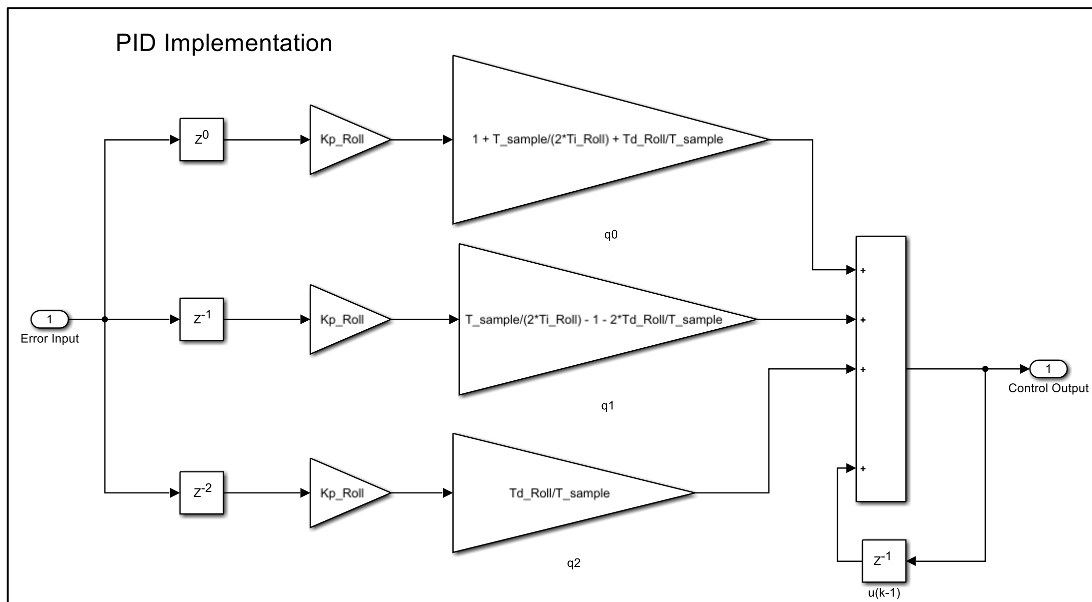


Figure 4-13. Simulink – PID Implementation

4.5 Simulated PID Tuning

In this section, we are going to describe the simulated PID tuning on Simulink.

After making sure that the model behaves correctly when loop is open (the drone is stopped by the ground and lifts when forces applied), we could start the tuning process.

Although a first System identification & Auto-Tuning was executed, the values of the resulted PDI parameter were not able to keep the drone under a controlled flight. For that reason, we decided to perform a manual PID tuning.

In the following image, we can see a comparison of the pitch angle response when using different PID parameters. We have considered that the roll movement is equivalent, so the tuning has been performed using the pitch angle only.

- The red curve is the response of the system when only the P and D components are more than zero. We can see that the permanent regime is not reached.

- The green curve is the response when introducing the integral part. The permanent regime is reached now, but the stabilisation time and overshoot are too big.
- If increasing the proportional gain, the stabilisation time and overshoot reduce (in purple).

In the blue curve, we can see that if increasing now the derivative component, the stabilisation time increases, but the overshoot is reduced.

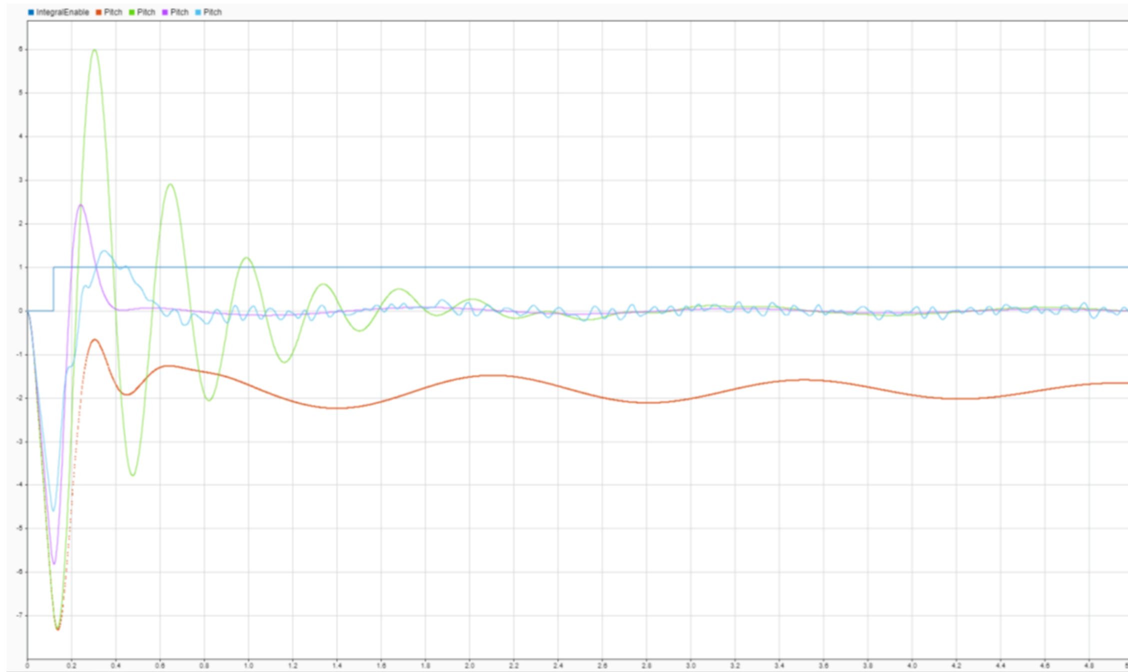


Figure 4-14. Comparison of the pitch angle response when using different PID parameters

We have added a *Random Noise Source*, which allows us to simulate our actual measurements. Those are not perfectly stable, having a variance of ±0.1 degrees. That is the reason why the curve in blue from the previous image is not plane.

Note that the Enable curve shows the moment when the "I" term is applied. This allows us to simulate the real behaviour, where that component is activated once flying.

As previously said in *Control Design – Extra features*, by delaying the activation of the integral component, we observe that the response of the system improves, as the overshoots is reduced.

As we know, the integral factor of the PID is good at reducing the error in the permanent regime, but makes the response to overshoot. Therefore, by disabling it at the beginning, we are able to enhance the transitory regime.

In the figure below, we can see that the error increases by a significant amount at the beginning, because of the little imbalances of the drone's construction. That error is quickly corrected thanks to the contribution of the "P" and "D" components. The "I" one only would integrate that error, so that when it is equal to zero, a further correction is performed by this component, until the overall integral is zero, causing the system to overshoot.

In red we see the response when the "I" component is enabled from the start, whereas in blue we can see the response when the activation is delayed.

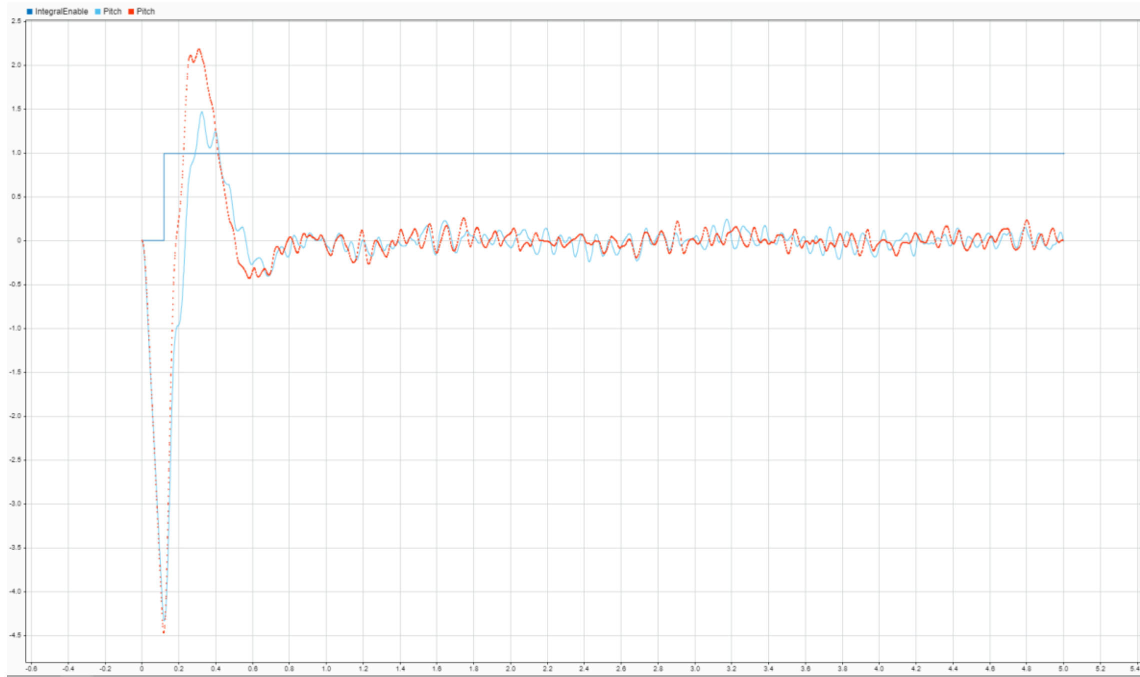


Figure 4-15. Pitch angle response when delaying the integral component of the PID (in blue)

If we further delay the activation, as shown in the next image, we can see that the response can enhances (in red).

Figure 4-16. Pitch angle response when further delaying the integral component of the PID (in blue)

If delaying it too much, the response starts to slow down at the beginning, as shown in the following image (in green).

Figure 4-17. Pitch angle response when delaying too much the integral component of the PID (in blue)

To implement this in the drone, we enables it when any of pitch or roll angles has changed by 5 degrees and the main power applied is 40%.

The final PID parameters obtained were:

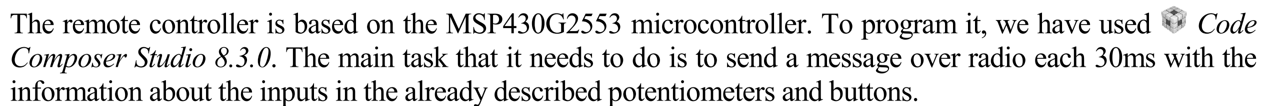
	Kp	**Ti**	**Td**
Pitch	0.01	0.1	0.1
Roll	0.01	0.1	0.1

Regarding the yaw ones, as we did not model the motor's torque, we were not able to simulate the yaw controller. Nevertheless, this was not an issue because once we had a stable implementation for pitch and roll in the flight tests, we could easily find yaw parameters that kept that movement under control.

5 FIRMWARE DEVELOPMENT

In this section, we are going to detail the firmware development for both remote controller and the drone's on-board controller.

5.1 Remote Controller Firmware



The remote controller is based on the MSP430G2553 microcontroller. To program it, we have used *Code Composer Studio 8.3.0*. The main task that it needs to do is to send a message over radio each 30ms with the information about the inputs in the already described potentiometers and buttons.

The overall logic is as follows:

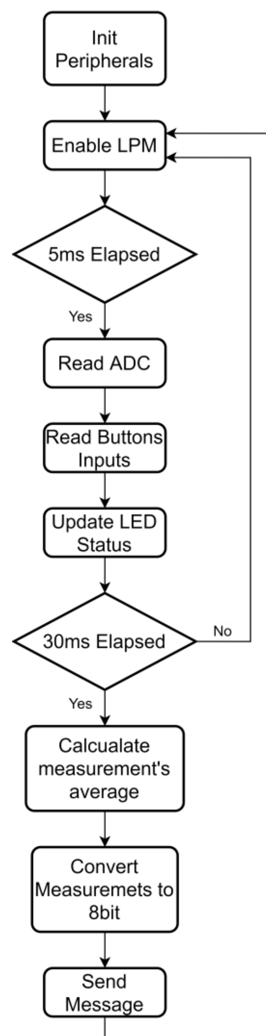


Figure 5-1. Remote Controller Firmware Logic

As we see, the program is executed each 5ms. The Analogic to Digital Converter (ADC) peripheral is first read, followed by the status of the buttons. Then the blinking status LED is updated. Once having done this 6 times (after 30ms), we calculate the average of the ADC readings and construct a message, which is sent over radio using the UART.

5.1.1 Peripherals Configuration

The first action is to configure the peripherals, though. We need to select the right configuration parameters, in accordance to the way we will use them later.

Note that all the figures related to the peripherals can be found in the device datasheet ([9]).

- **Clock Configuration**.

Given that the microcontroller's task does not require much speed (the functional period is 5ms), we can use a relatively slow one, like 8MHz, which allows us to save some battery.

As we can see in the next image, we use the RSEL and DCO bits to select the desired frequency.

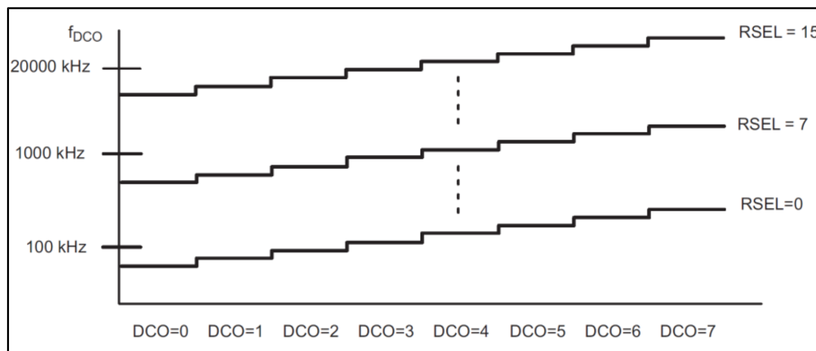


Figure 5-2. MSP430 Clock Frequency configuration parameters

The implemented code can be found at the end of this sub-section.

- **UART Configuration.**

To configure the UART, we first select the control bits of the GPIO Port 1 as seen below.

Table 16. Port P1 (P1.0 to P1.2) Pin Functions

PIN NAME (P1.x)	x	FUNCTION	CONTROL BITS AND SIGNALS (1)				
			P1DIR.x	P1SEL.x	P1SEL2.x	ADC10AE.x INCH.x=1 (2)	CAPD.y
P1.0/	0	P1.x (I/O)	I: 0; O: 1	0	0	0	0
TA0CLK/		TA0.TACLK	0	1	0	0	0
ACLK/		ACLK	1	1	0	0	0
A0 (2)/		A0	X	X	X	1 (y = 0)	0
CA0/		CA0	X	X	X	0	1 (y = 0)
Pin Osc		Capacitive sensing	X	0	1	0	0
P1.1/	1	P1.x (I/O)	I: 0; O: 1	0	0	0	0
TA0.0/		TA0.0	1	1	0	0	0
		TA0.CCI0A	0	1	0	0	0
UCA0RXD/		UCA0RXD	from USCI	1	1	0	0
UCA0SOMI/		UCA0SOMI	from USCI	1	1	0	0
A1 (2)/		A1	X	X	X	1 (y = 1)	0
CA1/		CA1	X	X	X	0	1 (y = 1)
Pin Osc		Capacitive sensing	X	0	1	0	0
P1.2/	2	P1.x (I/O)	I: 0; O: 1	0	0	0	0
TA0.1/		TA0.1	1	1	0	0	0
		TA0.CCI1A	0	1	0	0	0
UCA0TXD/		UCA0TXD	from USCI	1	1	0	0
UCA0SIMO/		UCA0SIMO	from USCI	1	1	0	0
A2 (2)/		A2	X	X	X	1 (y = 2)	0
CA2/		CA2	X	X	X	0	1 (y = 2)
Pin Osc		Capacitive sensing	X	0	1	0	0

(1) X = don't care
(2) MSP430G2x53 devices only

Figure 5-3. MSP430 UART configuration parameters

The chosen clock souce has been SMCLK (8MHz).

Regarding the communication speed, we are using the maximun speed that the radio module can handle (115200 bauds), so that the time needed for each serial transmission is the lowest possible.

To achieve that velocity, we need to configure the Clock prescaler registers (UCBRx). Given that the suorce clock is 8MHz:

$$UCBRx = \frac{8000000}{115200} = 69\ (0x0045)$$

The implemented code can be found at the end of this sub-section.

- **GPIO Configuration.**

As described in the section *Remote Controller Hardware Design*, we are using a number of GPIO to control the LED and the buttons. We can see the mapping in the next table.

HW	**Port**	**Bit**	**Input/output**
LED	2	3	Output
Button 1	2	4	Input
Button 2	2	2	Input
Left Gimbal Button	2	1	Input

The implemented code can be found at the end of this sub-section.

- **ADC Configuration**

To help understand the ADC configuration, we can see an overview of its registers in the following image.

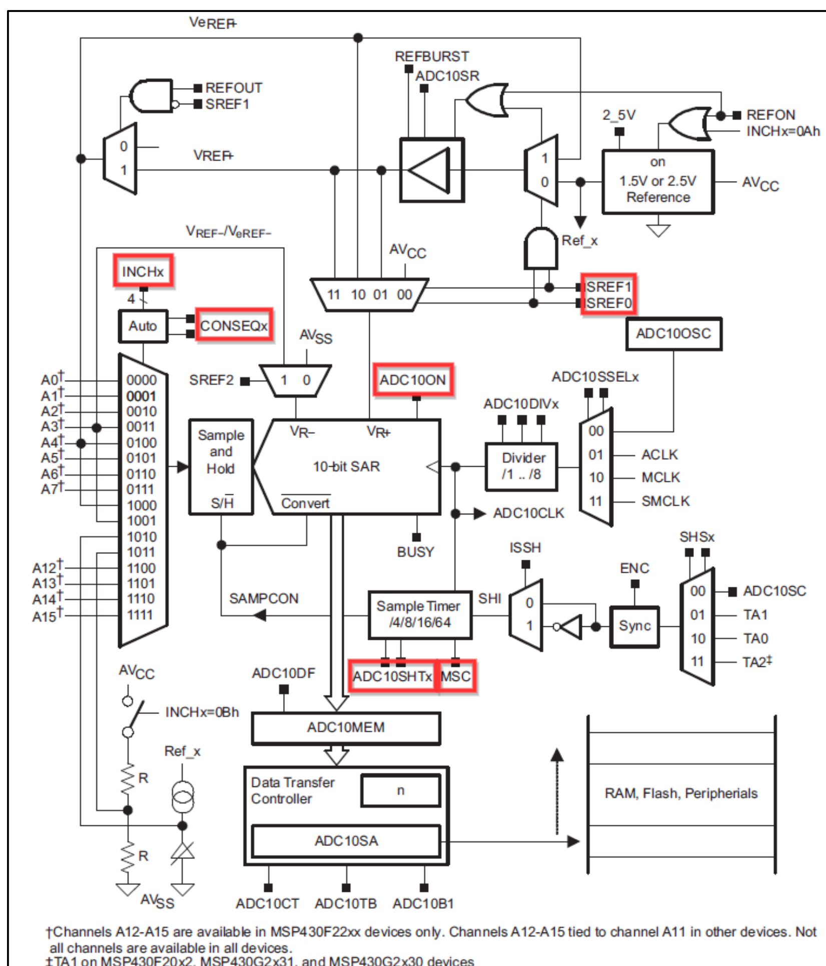


Figure 5-4. Overview of the MSP430 ADC peripheral

The chosen ADC configuration is described as follows (the parameters not mentioned are left as default):

- ADC10CTL0 (ADC10 Control Register 0) Configuration:
  - *Voltage Reference:* VR+ = VCC and VR- = VSS (SREFx = 000)
  - *Sample-and-hold time:* 16 × ADC10CLKs (ADC10SHTx = 10)
  - *Enabled*: Yes (ADC10ON = 1)
  - *Multiple Sample and conversion:* Automatically triggered (MSC = 1)
  - *Interrupt:* Enabled (ADC10IE = 1)

- ADC10CTL1 (ADC10 Control Register 1) Configuration:
  - *Input channel select:* highest channel for a sequence of conversions = A7 (INCHx = 0111)
  - *Conversion sequence mode select*: Sequence-of-channels (CONSEQx = 01)

- ADC10AE0 (Analog (Input) Enable Control Register 0) Configuration:
  - Analog enable channels: all except A2 (ADC10AE0x = 0xFB)

- ADC10DTC1 (Data Transfer Control Register 1) Configuration:
  - *Number of transfers in each block*: 8 (ADC10DTC1 = 8)

In the next table, we can see the analogic inputs mapping.

Measurement	**Analogic Port**
Power (Left Gimbal – Vertical Movement)	A0
Pitch (Right Gimbal – Vertical Movement)	A3
Roll (Right Gimbal – Horizontal Movement)	A6
Yaw (Left Gimbal – Horizontal Movement)	A7

- **Timer Configuration**

In case of the timer peripherals, the overview would be the following:

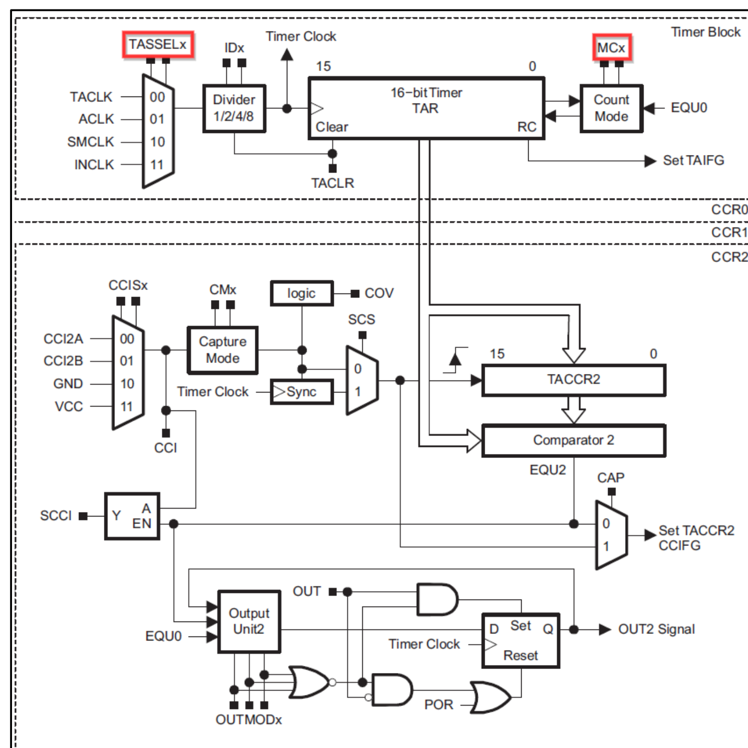


Figure 5-5. Overview of the MSP430 Timer peripheral

As we can see in the firmware flow diagram from the start of this section, we need to wait 5ms to do each action. To do that we use the Timer A, configured to execute its interruption routine with a period of 5ms.

To achieve it, we have configured the peripheral as follows (the parameters not mentioned are left as default):

- TACTL (Timer_A Control Register) Configuration:
  - *Clock Source:* SMCLK, 8MHz (TASSELx = 10)
  - *Mode Control:* Up mode: the timer counts up to TACCR0. (MCx = 01)
- TACCR0 (Timer_A Capture/Compare Register 0) Configuration:
  - *Timer_A capture/compare register:* TACCR0 = 40000

$$TACCR0 = \frac{8000000\ Hz}{1/0.005s} = 40000$$

- TACCTL (Capture/Compare Control Register) Configuration
  - *Capture/compare interrupt enable*: Yes (CCIE = 1). When the count is equal to TACCR0, the interrupt is triggered, as MCx = 01. MCx = 01

```
/*** Clock Configuration. 8MHz ***/
WDTCTL = WDTPW + WDTHOLD;          /* Stop WDT */
if (CALBC1_8MHZ==0xFF)             /* If calibration constant erased */
{
  while(1);                        /* do not load, trap CPU!! */
}
DCOCTL = 0;                        /* Select lowest DCOx and MODx settings */
BCSCTL1 = CALBC1_8MHZ;             /* Set DCO */
DCOCTL = CALDCO_8MHZ;

/*** UART Configuration ***/
P1SEL = BIT1 + BIT2 ;              /* P1.1 = RXD, P1.2=TXD */
P1SEL2 = BIT1 + BIT2 ;             /* P1.1 = RXD, P1.2=TXD */
UCA0CTL1 |= UCSSEL_2;              /* SMCLK */
#ifdef CPU8_UART115200
UCA0BR0 = 0x45;                    /* 8MHz / 115200 = 69 (0x0045) */
UCA0BR1 = 0x00;
#else
UCA0BR0 = 0x41;                    /* 8MHz / 9600 = 833 (0x0341) */
UCA0BR1 = 0x03;
#endif
UCA0MCTL = UCBRS0;                 /* Modulation UCBRSx = 1 */
UCA0CTL1 &= ~UCSWRST;              /* Initialize USCI state machine */
IE2 |= UCA0RXIE;                   /* Enable USCI_A0 RX interrupt */

/*** GPIO Configuration ***/
P2DIR |= BIT3;                     /* Output pin for LED */
P2DIR &= ~BIT4;                    /*Input Mode - B1*/
P2DIR &= ~BIT2;                    /*Input Mode - B2*/
P2DIR &= ~BIT1;                    /*Input Mode - Button Gimbal Left*/

/*** ADC Configuration ***/
ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE + MSC;
/*
* ADC10SHT_2  : S/H time. 16 × ADC10CLKs
* ADC10ON     : ADC10 On/Enable
* ADC10IE     : ADC10 Interrupt Enable
* MSC         : Multiple SampleConversion
* SREF_6      : VR+ = VeREF+ and VR- = VREF-/ VeREF-. Devices with VeREF+/- pins only.
*/
#ifdef EXTERN_REFERENCE_VREF_GROUND
ADC10CTL0 |= SREF_7;
#endif
ADC10CTL1 = CONSEQ_1 + INCH_7;        /* Conversion code singed format, input A1 */
ADC10AE0 = 0xFB;                      /* Analog input's enable */
ADC10DTC0 = 0xE1;
ADC10DTC1 = 8;                        /* Number of coversions on each DTC block (Data Transfer
Controller)*/

/*** Timer A0 Configuration ***/
TA0CTL = TASSEL_2;                 /* SMCLK */
TA0CCR0  = 40000;                  /* 5ms */
TA0CCTL0 = CCIE;                   /* CCR0 interrupt enabled */
```

5.1.2 Low Power Mode

The MSP430 family is well known as its low consumption, thanks to a variety of Low Power Modes. The typical consumption is shown in the next figure.

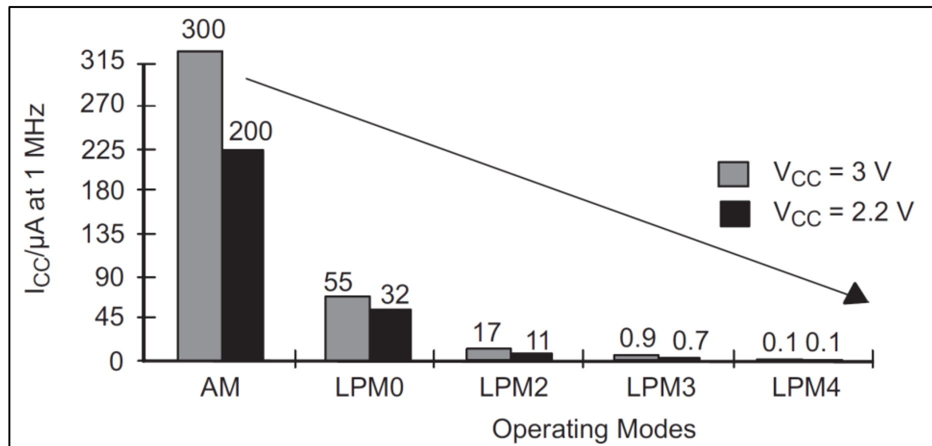


Table 2-2. Operating Modes For Basic Clock System

SCG1	SCG0	OSCOFF	CPUOFF	Mode	CPU and Clocks Status
0	0	0	0	Active	CPU is active, all enabled clocks are active
0	0	0	1	LPM0	CPU, MCLK are disabled, SMCLK, ACLK are active
0	1	0	1	LPM1	CPU, MCLK are disabled. DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active.
1	0	0	1	LPM2	CPU, MCLK, SMCLK, DCO are disabled. DC generator remains enabled. ACLK is active.
1	1	0	1	LPM3	CPU, MCLK, SMCLK, DCO are disabled. DC generator disabled. ACLK is active.
1	1	1	1	LPM4	CPU and all clocks disabled

Figure 5-6. MSP430 Low Power Modes

The chosen Low Power Mode has been *LPM1*, which allows us to save the most energy while keeping the SMCLK is active, so that the timer's interrupt can be triggered to wake the microcontroller up. Note that SMCLK has been the chosen source clock for the 'Timer A', as described before.

5.1.3 GPIO Read/write and ADC Measurements

In order to read the status of the buttons and turn the LED on/off, we access to the correspondent port. For example:

`!(P2IN & BIT4)` to read the status of the BIT4 of Port 2 (our button 1).

`P2OUT |= BIT1` to set an '1' in the BIT1 of the Port 2 (our LED).

Regarding the buttons readings, we have implemented an anti-bouncing 10ms filter, so we get rid of erroneous values when the buttons are being pressed or un-pressed. To do that, we use counters to wait certain time after the fulfilment of the conditions. The code implementation for the Button 1 would be: (note that for the other two buttons the code is equivalent)

```
/* GPIO Status Reading/Write Definitions */
#define B1_PRESSED      !(P2IN & BIT4)

(...)

  /*** Read Buttons ***/
  /* B1 */
  if(B1_PRESSED)
    buttonCounterB1++;
  else
  {
    buttonPressedForSure = 0;
    buttonCounterB1 = 0;
    if(timerAterPressingB1 > 0)
      timerAterPressingB1--;
  }
  if(buttonCounterB1 >= MS_10)
  {
    buttonPressedForSure = 1;
    timerAterPressingB1 = TIMER_AFTER_PRESSING_START_TIME;
  }
```

Regarding the ADC measurements, as per the peripheral construction, each conversion sequence starts from the most significant channel (INCH_x), and ends on A0. If we wanted to convert A0, A3, A6 and A6 in the same sequence, all the channels will need to be converted inevitably.

At the beginning of each conversion sequence, we need to indicate where the memory space is (of 8 unsigned integers). In our case it will be: `ADC10SA = ADCMeasuremets5;`. And then trigger the sequence by `ADC10CTL0 |= ENC + ADC10SC;`.

As shown in the previous flow diagram, a conversion is done each 5ms, and the messages are sent each 30ms. So once the 30ms elapse, we make the average of the last 6 measurements.

Additionally we need to right shift the measurements by 2, so the 10-bit data fits the 8-bit based UAR transmissions:

```
    /* Filter the 5ms-based ADC measurements. 30ms window */
    if(++t <= 6)
    {
      for(i=0;i<8;i++)
        ADCMeasuremets30[i] += ADCMeasuremets5[i];
    }
    if(t >= 6) /* When 30ms elapsed, make the average of the previous readings */
    {
      t=0;
      /* Also right shift the result, so the register is converted 10bit -> 8bit */
      /* Note that the UART communication is 8bit-based */
      for(i=0;i<8;i++)
        ADCMeasuremets30[i] = (ADCMeasuremets30[i] / 6 ) >> 2;


      (...)

     }
```

5.1.4 Message Construction

Before going into detail about the actual radio messages, we are going to describe the user's perspective operation of the remote controller.

As we see in the following image, and in accordance with that described in the hardware development section, the remote controller consist of two potentiometers and three buttons. In the next image we can see the User Interface diagram of the Remote Controller.

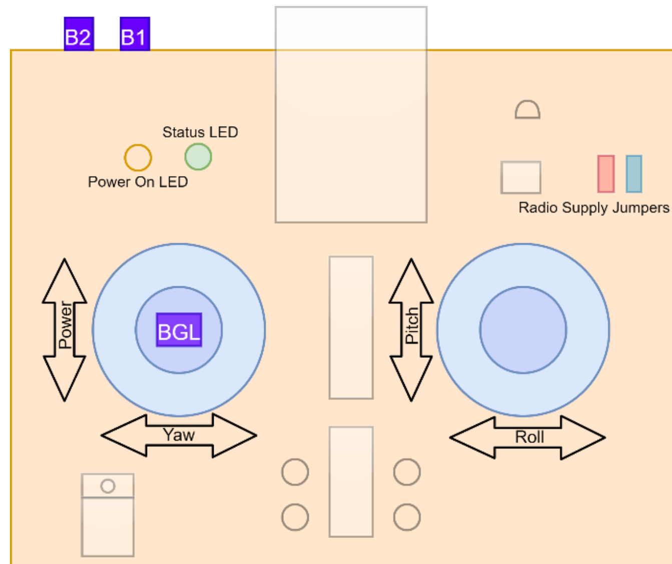


Figure 5-7. User Interface diagram of the Remote Controller

The operation of each of them is:

- Left Gimbal:
    - Vertical movement: Increases/Decreases the main power (increasing upwards). The middle position corresponds to a power of 0% at the beginning. This value is changed when the joystick is moved up/down.
    - Horizontal movement: Yaw (increasing to the left). The middle position corresponds to a set point of 0 degrees per second.
    - GBL (Button of the Gimbal at Left): Cut off the main power to zero.
- Right Gimbal:
    - Vertical movement: Pitch (increasing upwards). The middle position corresponds to a set point of 0 degrees by default (can be changed with B1).
    - Horizontal movement: Roll (increasing to the right). The middle position corresponds to a set point of 0 degrees by default (can be changed with B1).
- Separate buttons:
    - B1 (Button 1). Is used as a trim. When it is pressed and released, the pitch and roll values are frozen until the right gimbal returns to the middle position. These pitch and roll angles (trim values) will be the default set points (instead of zero). Further movements will be added to them.
      It is further detailed in the section *Running State*, as the actual implementation of this functionality has been done in the drone's firmware.
    - B2 (Button 2). It is used to slow down the rate of change of the main power. This is also detailed in the section *Running State,* as the actual implementation of this functionality has been done in the drone's firmware.

To establish a communication between the RC Controller with the drone, we are using a couple of radio modules (E34-2G4D20D). First, we send a message to the transmitter one over UART. The message is then sent through the air, and received by the other in the drone.

Apart from the message that we want to transmit, the radio modules construct a more complicated one, containing the addresses, and other fields that are not under the control of us. What we know so far about the actual communication is [10]:

"*The module has data encryption and compression capabilities. The data transmitted by the module in the air is random, and the data interception loses its meaning through strict encryption and decryption algorithms. The*

data compression function has the probability of reducing the transmission time, reducing the probability of interference, improving reliability and transmission efficiency."

Our messages structure is as follows:

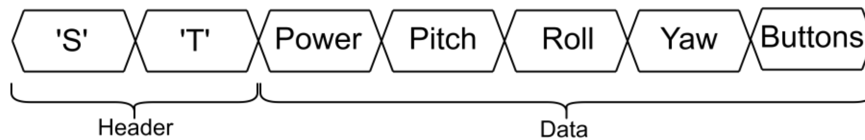


Figure 5-8. Message Structure

It consists of 7 bytes, where the first two are the '*Header*' and rest the '*Data*'. The 'Header' is used to synchronise the reception of the message.

The first four bytes of the 'Data' represents the inputs on the potentiometers of RC controller from 0 to 255. 0 is sent when the potentiometers are in the left or down position, whereas 255 is sent when in the right or up positions.

The last byte shows the status of each button. In this case, only the least three significant bits are used: 0b00000[B2][B1][BGLeft]. Where [B2] is the status of the Button 2, [B1] is the status of the Button 1 and [BGLeft] is the status of the button of the left gimbal. '1' represents pressed and '0' not pressed.

Note that none CRC (Cyclic Redundancy Check) has been implemented, considered being unnecessary, as the radio modules implement reliability functions already. In addition, given that we receive a new message each 30ms, we can afford any punctual error.

The code implementation is shown. Note that we need to copy each byte in `UCA0TXBUF`.

```
	/* Message Header */
	send('S');
	send('T');

	/* Send the message content */
	send(ADCMeasuremets30[7]); /* A0 (P1.0) - Power */
	send(ADCMeasuremets30[4]); /* A3 (P1.3) - Pitch */
	send(ADCMeasuremets30[1]); /* A6 (P1.6) - Roll  */
	send(ADCMeasuremets30[0]); /* A7 (P1.7) - Yaw   */
	buttonsStatus = (buttonB2PressedForSure<<2 | buttonPressedForSure<<1 |
buttonBGLeftPressedForSure);
	send(buttonsStatus);
```

```
void send(char byte)
{
  while (!(IFG2&UCA0TXIFG));                /* USCI_A0 TX buffer ready? */
    UCA0TXBUF = byte;
}
```

5.2 Drone Controller Firmware

In this section, we are going to detail the drone's firmware design, the program that will be running in the on-board ESP8266.

To compile and load the code into the microcontroller we have used *Arduino IDE 1.8.13*.

The main structure of the drone's firmware is shown in the image below. We can see the possible different status where the done can be, the main tasks that are executed on each state, and the common tasks, that provides information and time managing to the first ones.

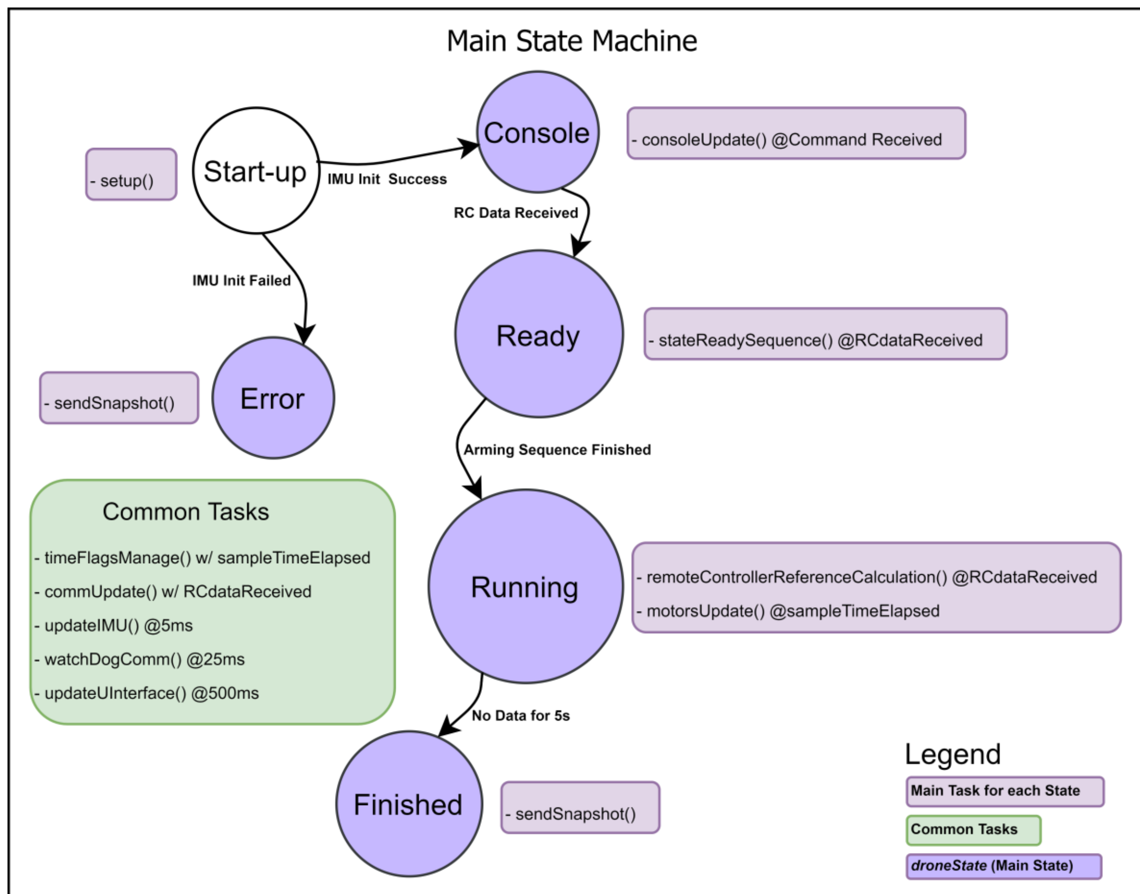


Figure 5-9. Drone's Firmware Structure – Main State Machine

When the device is booted up, we execute some start-up actions to initialise the system, which leads to the *Error* state if unsuccessful, or going forward to the next status. After being in the *Console* and *Ready* status, the drone is able to fly when in the *Running* state. Once the flight finishes, the status changes to *Finished*.

In what follows, we describe each status and common tasks separately.

5.2.1 Start-up State

Functionality implemented in the following function:

- `setup()`

The start-up sequence is executed after the device is powered on, and allows us to initialise the system. In the next image, we can see the actions performed.

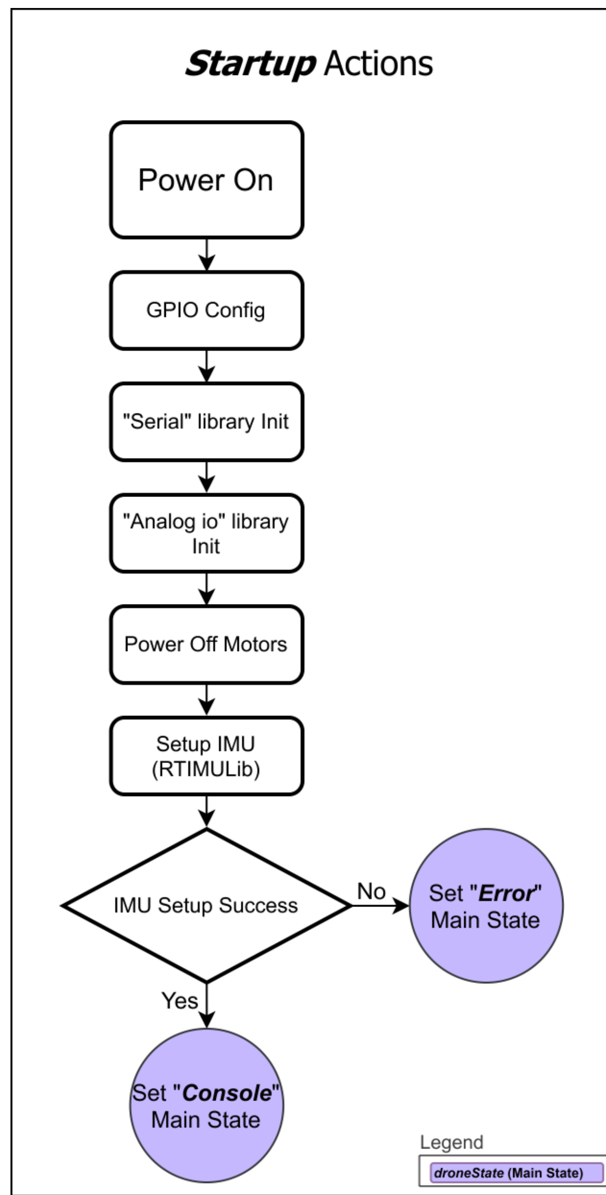


Figure 5-10. Drone's Firmware Structure – Start-up State

First, the GPIO are configured as per the table below:

HW	Port	Input/output
LED	D4	Output
Buzzer	D6	Output
Debug Port	D0	Output
Motor 1	D3	Output (PWM)
Motor 2	D5	Output (PWM)

Motor 3	D7	Output (PWM)
Motor 4	D8	Output (PWM)

In order to communicate with the radio module, we use the UART peripheral as we did in the Remote Controller firmware. In this case, we initialise it just by using the initialisation function of the "Serial" library.

The PWM signals will be handled by the "Analog io" library. To initialise it we need to set the working frequency and the maximum range (1023) for its input. As discussed earlier, we are using four 150Hz signals, where the minimum power is set then the duty cycle is 15%, and maximum when 30%.

As seen in the code implementation, we use the function `analogWrite(Mx, value)` where *Mx* is a motor port, and *value* represents the duty cycle, based on the configured maximum range. Then, to get the minimum power, we would write 15% of 1023 = 153. To get full power we set 30% of 1023 = 306.

The interface between the IMU and our program is done using the existing library RTIMULib (2015), which needs to be initialised as well.

- *RTIMULib-Arduino Copyright Notice and Permission Notice.*

```
This file is part of RTIMULib-Arduino

Copyright (c) 2014-2015, richards-tech

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Figure 5-11. RTIMULib-Arduino Copyright Notice and Permission Notice

To do that, we create an RTIMU object using `RTIMU::createIMU(&settings)`

If the initialisation process is successful, we continue with the program, setting the drone main state to "Console". In other case, we set "Error" and the system would need to be rebooted.

5.2.2 Console State

Functionality implemented in the following function:

- `consoleUpdate()`

After the start up actions, the drone runs a console functionality, where we can use a number of commands to configure certain parameters, as for example the PID constants.

Its flow diagram is the following:

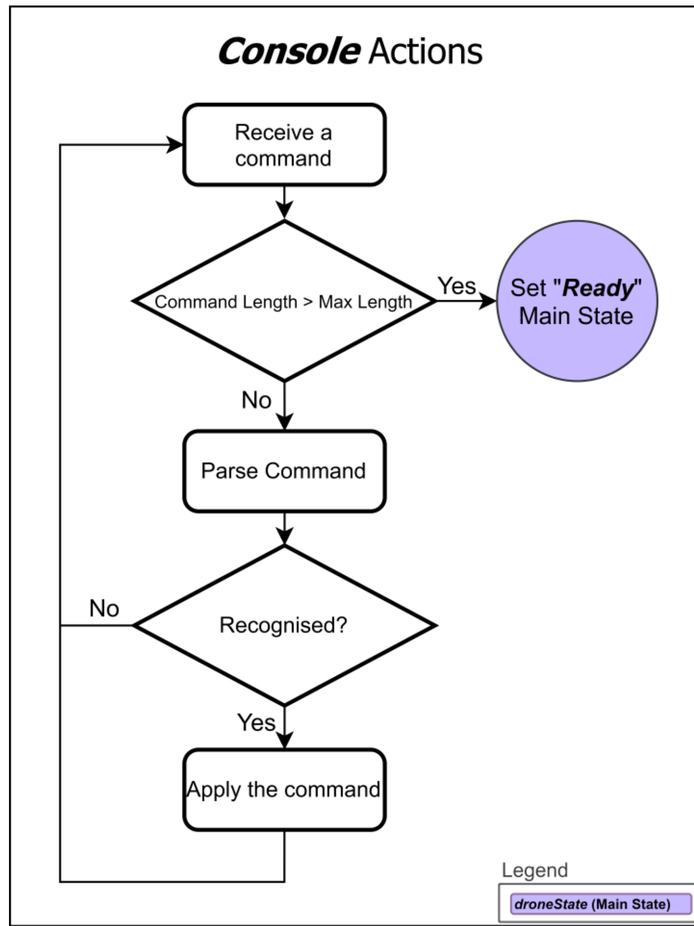


Figure 5-12. Drone's Firmware Structure – Console State

First, we read any incoming serial data and interpret it as a command. The command terminators can be: '\r', '\n' or ';'. Additionally, if the terminator is not found after receiving `MAXBYTESCONSOLE` bytes, the console functionality terminates and the main status is switched to "Ready". This is the case when the RC controller is powered on, so RC data is constantly received.

If the length of the data received is less than the maximum, we parse and apply the command if recognised.

The format needs to be:

[Command] [value];

Where the separation is a single space.

For example, if receiving "KpPitch 10" the proportional constant of the Pitch PID is changed to 10/100 = 0.1, as described below.

Commands list:

- *help*. To check that the console functionality works.
- *filterWindow*. To select the length of the IMU readings filter window. Default: 4.
- *PID*. To show the current PID parameters.
- *Snapshot.* To send the snapshot data.
- *SnapshotW*. Debug command to generate snapshot data.
- *KpPitch*. Parameter setting (x100).
- *TiPitch.* Parameter setting (x100).

- *TdPitch*. Parameter setting (x100).
- *KpRoll*. Parameter setting (x100).
- *TiRoll*. Parameter setting (x100).
- *TdRoll*. Parameter setting (x100).
- *KpYaw*. Parameter setting (x100).
- *TiYaw*. Parameter setting (x100).
- *TdYaw*. Parameter setting (x100).

5.2.3 Ready State

Functionality implemented in the following function:

- `stateReadySequence()`

As said above, the *Console* status is left when receiving certain number of bytes (i.e. the RC controller is powered-up). This leads to the *Ready* state, where the arming sequence is available. This has been implemented as a security procedure, so the motors cannot rotate unless completed.

The arming sequence is detailed in the next image.

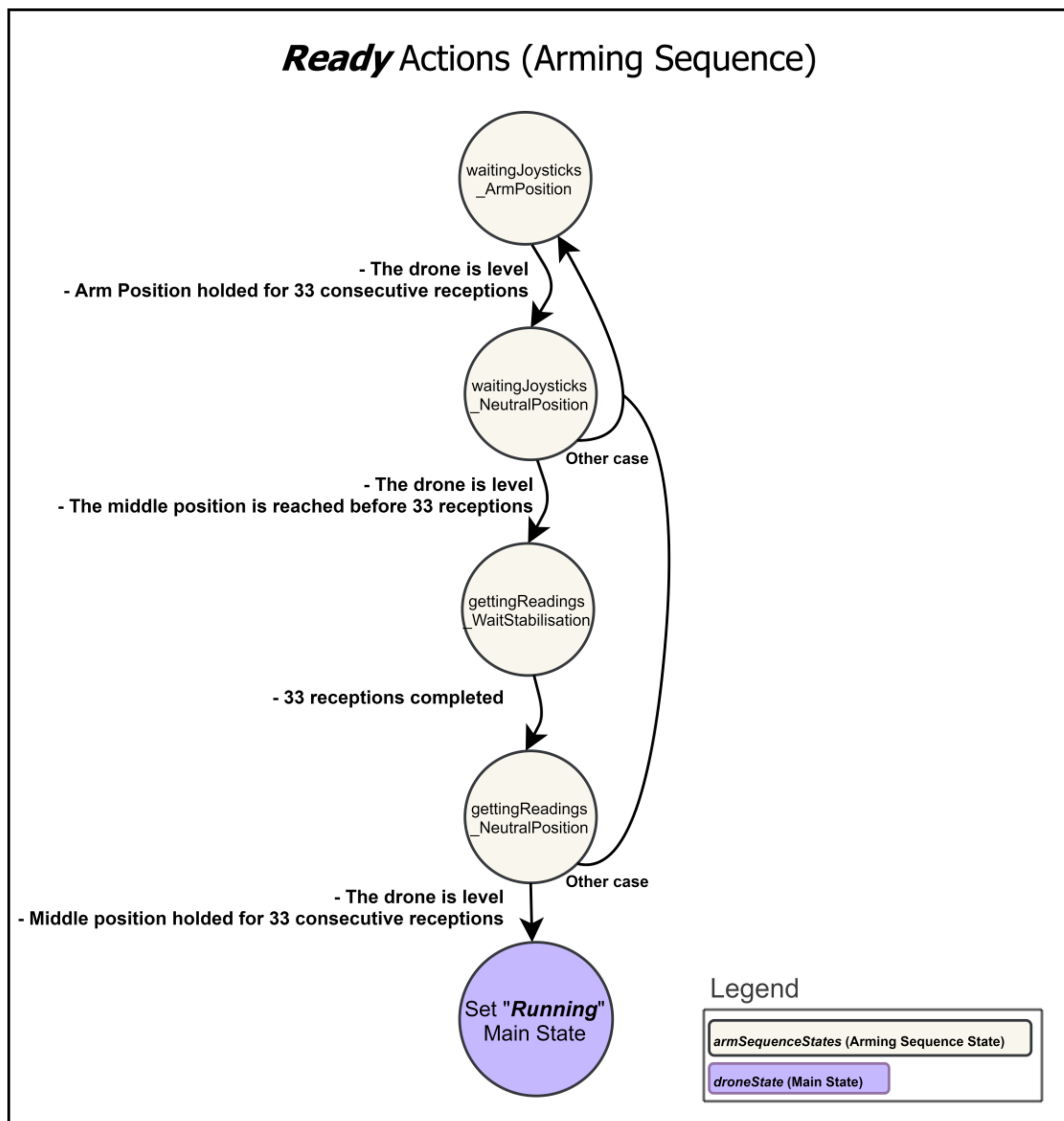


Figure 5-13. Drone's Firmware Structure – Ready State

Where the middle and arming positions are shown below:

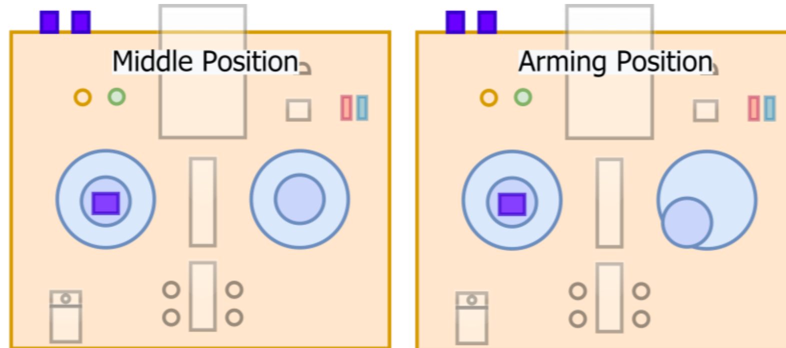


Figure 5-14. Middle and Arming positions in the Remote Controller

As we can see in the flow diagram, the status of the arming procedure is stored in the variable *armSequenceStates*. Once completed, the main status, *droneState*, is changed to *Running*.

In order to advance in the sequence, the conditions described has to be fulfilled. These consist of the drone being levelled (the IMU readings are within the expected limits) and the needed joysticks positions are held for a number of receptions (33).

In normal circumstances, 33 receptions will be received in 33*30ms = 0.99 s. Note that the RC controller sends each message each 30ms.

In addition to the security reasons, the arming procedure is used to get some initial measurements of the IMU and the remote controller when the drone is stopped in the ground, before the flight. Those measurements are:

- The set points received from the RC controller. These are used to calculate the relative movement from the middle position of the joysticks.
- The initial pitch and roll angles. Used in the *fly condition* detection.
- The initial yaw angle. Is used as a reference to know the yaw degrees turned since the start of the flight, instead of the start-up of the system.

5.2.4 Running State

Functionality implemented in the following functions:

- `remoteControllerReferenceCalculation()`
- `motorsUpdate()`

Running is the state where the drone is most of the time. In this state, we control the flight based on the IMU readings and the set points received from the remote controller.

In the next image, we can see a flow diagram of the actions done.

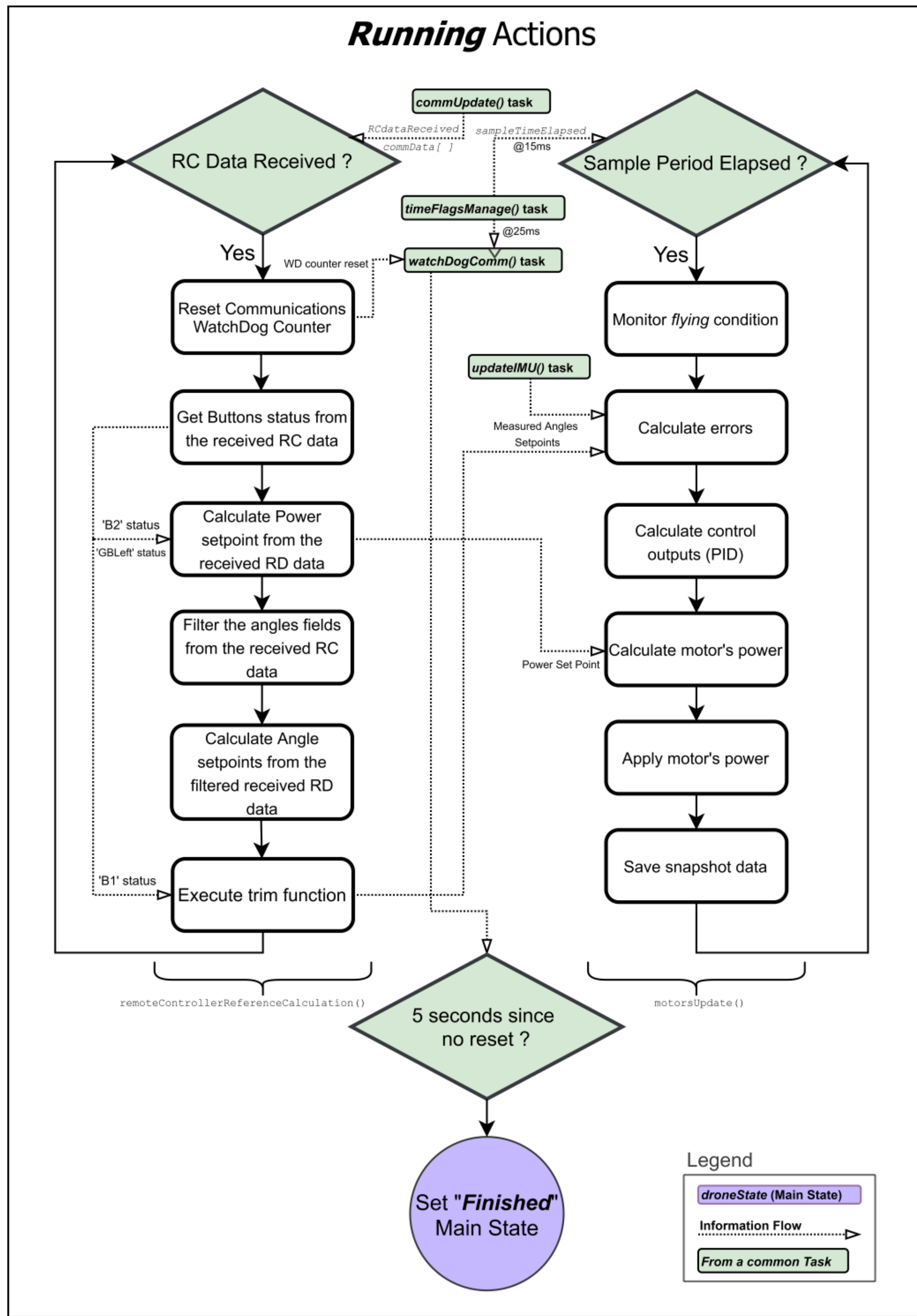


Figure 5-15. Drone's Firmware Structure – Running State

On the left, we see the processing of the data received by radio. This information, together with the IMU measurements, serves as the input for the calculation of the control outputs (on the right).

This status finishes when no data is received for 5 seconds. The task `watchDogComm()` is in charge of that.

1) remoteControllerReferenceCalculation()

As shown in the flow diagram above, the RC data processing is done in the `remoteControllerReferenceCalculation()` function: when the `RCdataReceived` flag is activated, the received data (stored in `commData[ ]`) is interpreted as the set points described in *5.1.4*. The common task `commUpdate()` is the

responsible of managing the flag and the data. Before interpreting the received data, we reset the communications watchdog:

```
void remoteControllerReferenceCalculation(void)
{
  /* Low pass filter auxiliary variables */
  static uint8_t commDataPitchHistory[COMM_DATA_FILTER_LENGTH];
  static uint8_t commDataRollHistory[COMM_DATA_FILTER_LENGTH];
  static uint16_t buttonGBLeftPressedCount;
  /* If data received */
  if(RCdataReceived == true)
  {
    RCdataReceived = false;
    newReference = true;

    /*** Feed the communication watchdogs by settin the count to zero ***/
    watchDogCommCountSTerm = 0;
    watchDogCommCountLTerm = 0;

   (...)
   }
```

As we also discussed in *5.1.4*, the buttons status are encoded in the last byte of the message. Therefore, to decode it, we just need to read each individual bit. Regarding the BGL button bit, we consider that it is actually activated when receiving `BUTTON_GBLEFT_PRESSED_COUNT_UP` consecutive 1's:

```
   /*** Buttons ***/
    buttonB1Pressed = ((commData[COMM_DATA_BUTTONS_ID] & COMM_DATA_B1_MASK) > 0) ? 1:0;
    buttonB2Pressed = ((commData[COMM_DATA_BUTTONS_ID] & COMM_DATA_B2_MASK) > 0) ? 1:0;
    buttonGBLeftPressed = ((commData[COMM_DATA_BUTTONS_ID] & COMM_DATA_GBLeft_MASK) > 0) ? 1:0;


    if(buttonGBLeftPressed == true)
    {
      buttonGBLeftPressedCount++;
    }
    else
    {
      buttonGBLeftPressedCount = 0;
    }
    if(buttonGBLeftPressedCount >= BUTTON_GBLEFT_PRESSED_COUNT_UP)
    {
      buttonGBLeftKeepedPressed = true;
    }
    else
```

After that, the power set point is calculated. As we are using two joysticks whose resting position is in the middle, the main power is selected by the pilot incrementally. To start the flight the left joystick needs to be moved upwards, so the main power is increased. The more vertical movement, the more rate of change of it. Additionally by pressing the button 2 (B2), the maximum rate is reduced from `MAX_REF_POWER_DELTA_PER_SAMPLE` to `MAX_REF_POWER_DELTA_PER_SAMPLE_BUTTON_PRESSED`. These parameters are defined in DroneControl_Defines.h. All the needed calculations are done in the function `calculateSetPntPower()`, resulting in the main power, stored in `setPntPower` (range: 0 - 1).

```
    /*** Calculate the Set Points - 1***/
    /* Main Power */
    calculateSetPntPower();
```

To get the angles set points from the received data, we first filter them, so they do not change too quickly. We

have implemented that to make sure that the PID inputs are smooth enough. Nevertheless, it can be eventually removed, so the drone handling is improved.

Then the set points are scaled from the raw data [0, 255] to [0, MAX_REF_PITCH_DEG], [0, MAX_REF_ROLL_DEG] and [0, MAX_REF_YAW_DDPS]. These parameters are also defined in DroneControl_Defines.h.

After doing that we have the set points stored in setPntPitch, setPntRoll and setPntYawdDPS.

```
    /*** Apply filter to the reference parameters ***/
    commDataPitch = 0;
    commDataRoll  = 0;
    for(uint16_t j=1;j<COMM_DATA_FILTER_LENGTH;j++)
    {
      commDataPitchHistory[j-1] = commDataPitchHistory[j];
      commDataRollHistory[j-1] = commDataRollHistory[j];

      commDataPitch += commDataPitchHistory[j-1];
      commDataRoll  += commDataRollHistory[j-1];
    }
    commDataPitch += commData[COMM_DATA_PITCH_ID];
    commDataPitch = commDataPitch/COMM_DATA_FILTER_LENGTH;
    commDataRoll  += commData[COMM_DATA_ROLL_ID];
    commDataRoll  = commDataRoll/COMM_DATA_FILTER_LENGTH;

    commDataPitchHistory[COMM_DATA_FILTER_LENGTH-1] = commData[COMM_DATA_PITCH_ID];
    commDataRollHistory[COMM_DATA_FILTER_LENGTH-1]  = commData[COMM_DATA_ROLL_ID];

    /*** Calculate the Set Points - 2***/
    /* Pitch */
    setPntPitch = MAX_REF_PITCH_DEG - (MAX_REF_PITCH_DEG / commData_1_Mean) * commDataPitch;
    /* Roll */
    setPntRoll  = - MAX_REF_ROLL_DEG  + (MAX_REF_ROLL_DEG / commData_2_Mean) * commDataRoll;
    /* Yaw (velocity) */
    setPntYawdDPS   =   MAX_REF_YAW_DDPS   -   (MAX_REF_YAW_DDPS   /   commData_3_Mean)   *
commData[COMM_DATA_YAW_ID];
```

The last thing to be done is to apply the trim values. As said in the B2 button definition above, "when the B2 button is pressed and released, the received pitch and roll values are frozen until the right gimbal returns to the middle position. These pitch and roll angles (trim values) will be the default set points (instead of zero). Further movements will be added to them". Note that this functionality has been implemented in the drone itself: when the joystick is in the any position, we will always receive the same correspondent data, regardless of the button pressing.

When we detect a falling edge in B2, the set points are stored in setPntPitchToHold and setPntRollToHold. These values will be forced to be the set points regardless of the position of the right gimbal, until it reach the middle position.

Then to detect the middle position, we wait until the pitch and roll data are within SET_PNT_PITCH_ZERO_LOW_LIMIT, SET_PNT_PITCH_ZERO_HIGH_LIMIT and SET_PNT_ROLL_ZERO_LOW_LIMIT, SET_PNT_ROLL_ZERO_HIGH_LIMIT. These parameters are also defined in the DroneControl.ino file.

From this time, the values received will be added to the trim ones (setPntPitch += trimPitc; setPntRoll += trimRoll;).

```
    if(buttonB1Pressed == 0 && buttonB1PressedPrev == 1)
    {
      buttonB1FallingEdge = true;
    }
    buttonB1PressedPrev = buttonB1Pressed;
    if((setPntPitch < SET_PNT_PITCH_ZERO_HIGH_LIMIT && setPntPitch > SET_PNT_PITCH_ZERO_LOW_LIMIT)
&& (setPntRoll < SET_PNT_ROLL_ZERO_HIGH_LIMIT && setPntRoll > SET_PNT_ROLL_ZERO_LOW_LIMIT)) /* The
right joystick is in the middle position */
    {
      waitUntilSetPntReceivedZero = false;
    }
    setPntPitch += trimPitch; /* Apply trim values */

    setPntRoll  += trimRoll;
    if(buttonB1FallingEdge == true) /* Update trim values if button released */
    {
      trimPitch = setPntPitch;
      trimRoll = setPntRoll;
      setPntPitchToHold = setPntPitch;
      setPntRollToHold = setPntRoll;
      waitUntilSetPntReceivedZero = true;
      buttonB1FallingEdge = false;
    }
    if(waitUntilSetPntReceivedZero) /* While the right joystick has not reached the middle position
*/
    {
      setPntPitch = setPntPitchToHold; /* Overwrite the previously calculated values*/
      setPntRoll = setPntRollToHold;
    }
```

2) remoteControllerReferenceCalculation()

In the right hand of the previous flow diagram we can see the way we update the control outputs for the motors, in `motorsUpdate()`.

Once the sample time elapses (`sampleTimeElapsed` is activated), we perform the following actions.

First we execute the function `flyingConditionMonitor()`, where we judge if the flight has started, based on the change of the angles since the execution of the arming procedure and the received power set point. The actual conditions to determine the *flying* condition are:

- 1- The last `FLY_CONDITION_IMU_FILTER_LENGTH` (3 by default) IMU readings of pitch or roll angles has changed by `FLY_CONDITION_ANGLE_CHANGE` (5 degrees by default).
- 2- The power set point is equal or greater than `FLY_CONDITION_REF_WARNING_POWER` (42% by default).
- 3- `FLY_CONDITION_TIMER_MS` (50 milliseconds by default) have elapsed since the previous two conditions were fulfilled.

After the three conditions are true, we enable the extra feature explained in *Control Design – Extra features,* by activating the integral part of the PID. From this moment, the content of the function `flyingConditionMonitor()` then is no longer executed.

```
    /* Flying condition */
   if(timerTriggered==false && (setPntPower >= FLY_CONDITION_REF_POWER && (pitchChangeSinceStart
> FLY_CONDITION_ANGLE_CHANGE || rollChangeSinceStart > FLY_CONDITION_ANGLE_CHANGE)))
   {
     timeForFlyActivation = currentTime + FLY_CONDITION_TIMER_MS*1000;
     timerTriggered = true;
     beep(SHORT_BEEP);
   }

   if(timerTriggered==true && currentTime >= timeForFlyActivation)
   {
     flying = true;
     droneFlyingTimestamp = micros()/1000;
     pitchAtStartFlyingTime = IMUPitch;
     rollAtStartFlyingTime  = IMURoll;
     /* Enable the integral component of the PID controller */
     ActiveTiPitch = TiPitch;
     ActiveTiRoll = TiRoll;
     ActiveKpPitch = KpPitch;
     ActiveKpRoll  = KpRoll;

     beep(SHORT_BEEP);
   }
```

The errors between the IMU data and the received set points are calculated.

```
   /*** Angle errors calculation  ***/
   errorPitch = setPntPitch - IMUPitch;
   errorRoll  = setPntRoll  - IMURoll;
   errorYaw   = setPntYawdDPS  - IMUYawdDPS;
```

Note that the units of the first two are degrees, whereas the yaw error is measured in deci-degrees per second. The calculation of the IMU values is described in the section *Common Tasks*.

Now, the PID controllers are implemented using the `refCalculation()` function, which implements the PID formulas described in *Control Design – PID*.

```
float refCalculation (float T, float Kp, float Ti, float Td, float uPrev, float error, float
prevError, float prevPrevError)
{
  float u=0, q0, q1, q2;
  if(T!=0)
  {
    q0 = Kp*( 1 + T/(2*Ti) + Td/T );
    q1 = Kp*( T/(2*Ti) - 1 - 2*Td/T );
    q2 = Kp * Td/T;
    u = uPrev + q0*error + q1*prevError + q2*prevPrevError;
  }
  return u;
}


void motorsUpdate(void)
{
(...)

    uPitch = refCalculation(sampleTime, ActiveKpPitch, ActiveTiPitch, TdPitch, uPitch, errorPitch,
prevErrorPitch, prevPrevErrorPitch);

    uRoll   = refCalculation(sampleTime, ActiveKpRoll, ActiveTiRoll, TdRoll, uRoll, errorRoll,
prevErrorRoll, prevPrevErrorRoll);

    uYaw     = refCalculation(sampleTime, KpYaw, TiYaw, TdYaw, uYaw, errorYaw, preverrorYaw,
prevPreverrorYaw);


(...)
}
```

Based on the outputs of each PID we calculate and apply the composed power for each motor, as described in *Control Design – PID*.

```
powerM1 = setPntPower + uPitch - uRoll - uYaw;
powerM2 = setPntPower + uPitch + uRoll + uYaw;
powerM3 = setPntPower - uPitch + uRoll - uYaw;
powerM4 = setPntPower - uPitch - uRoll + uYaw;


pwm_M1 = ( PWMMAXRANGE / (1000.0 / PWM_FREQ) ) * (1 + powerM1);
pwm_M2 = ( PWMMAXRANGE / (1000.0 / PWM_FREQ) ) * (1 + powerM2);
pwm_M3 = ( PWMMAXRANGE / (1000.0 / PWM_FREQ) ) * (1 + powerM3);
pwm_M4 = ( PWMMAXRANGE / (1000.0 / PWM_FREQ) ) * (1 + powerM4);
analogWrite(M1, (int)pwm_M1);
analogWrite(M2, (int)pwm_M2);
analogWrite(M3, (int)pwm_M3);
analogWrite(M4, (int)pwm_M4);
```

Having done that, we save the current data in the snapshot structure, so the data can be retrieved when the flight finishes.

```
snapshot[iSnapshot].timestamp = currentTime/1000;

snapshot[iSnapshot].referencePower = setPntPower*100;

snapshot[iSnapshot].setPntPitch = setPntPitch;
snapshot[iSnapshot].setPntRoll = setPntRoll;
snapshot[iSnapshot].setPntYaw = setPntYawdDPS;

snapshot[iSnapshot].uPitch = uPitch*10000;
snapshot[iSnapshot].uRoll  = uRoll*10000;
snapshot[iSnapshot].uYaw   = uYaw*10000;

snapshot[iSnapshot].anglePitch = IMUPitch*100;
snapshot[iSnapshot].angleRoll  = IMURoll*100;
snapshot[iSnapshot].angleYaw   = IMUYawdDPS*100;
```

5.2.5 Finished and Error State

Functionality implemented in the following function:

- `sendSnapshot ()`

As said before, when no data is received for 5 seconds, the state is changed to *Finished*. In this situation, the motors are shut down and the snapshots (`structInfo snapshot[SNAPSHOT_MAX_NUMBER]`) can be sent over radio. These actions are the same as those done in the Error state.

Due to memory restrictions, the maximum number of snapshots is:

```
#define SNAPSHOT_MAX_NUMBER 1500
```

As the snapshots are recorded each 15ms (samplePeriod), it means that the data is captured for 22.5 seconds. We can decide if we keep the first or the last 22.5 seconds of the flight with the line: `iSnapshot = 0; /* Uncomment to overwrite */`, from the function `motorsUpdate()`.

In order to activate the data sending, a single data message needs to be received. For example "ST00000".

```
#define SNAPSHOT_MAX_NUMBER 1500
extern struct structInfo snapshot[SNAPSHOT_MAX_NUMBER];

void sendSnapshot(void)
{

  /* Shut down motors */
  analogWrite(M1, (int)( PWMMAXRANGE / (1000.0 / PWM_FREQ) ));
  analogWrite(M2, (int)( PWMMAXRANGE / (1000.0 / PWM_FREQ) ));
  analogWrite(M3, (int)( PWMMAXRANGE / (1000.0 / PWM_FREQ) ));
  analogWrite(M4, (int)( PWMMAXRANGE / (1000.0 / PWM_FREQ) ));

  if(RCdataReceived == true) /* Wait until receive some data through the radio module to start
printing */
  {
    RCdataReceived = false;
    delay(100);

    Serial.print("%------------------------------\n\n");
    Serial.print("testName = 'T';\n\n");


    Serial.print("KpPitch = ");
    Serial.print(KpPitch,5);
    Serial.print(";\nTiPitch = ");
    Serial.print(TiPitch,5);

    (...)

    for(uint16_t iSnapshot=0;iSnapshot<SNAPSHOT_MAX_NUMBER;iSnapshot++)
    {
      yield(); /* for builtin ESP8266 watchDog */

      /* End the transmission if found the first empty data */
      if(snapshot[iSnapshot].timestamp == 0)
      {
        iSnapshot = SNAPSHOT_MAX_NUMBER;
        continue;
      }

      Serial.print(snapshot[iSnapshot].timestamp);
      Serial.print(", ");
      Serial.print(snapshot[iSnapshot].referencePower);
      Serial.print(", ");
      Serial.print(snapshot[iSnapshot].setPntPitch);
      Serial.print(", ");
      Serial.print(snapshot[iSnapshot].setPntRoll );

      (...)
    }
  }
}
```

5.2.6 Common Tasks

In this section, the functionality of the common tasks is detailed. Those serve as a source of information and timing to the main actions described in the previous sections.

- **timeFlagsManage() task**

The `timeFlagsManage()` task serves timing services to the other. In the next image, we can see the outputs of it.

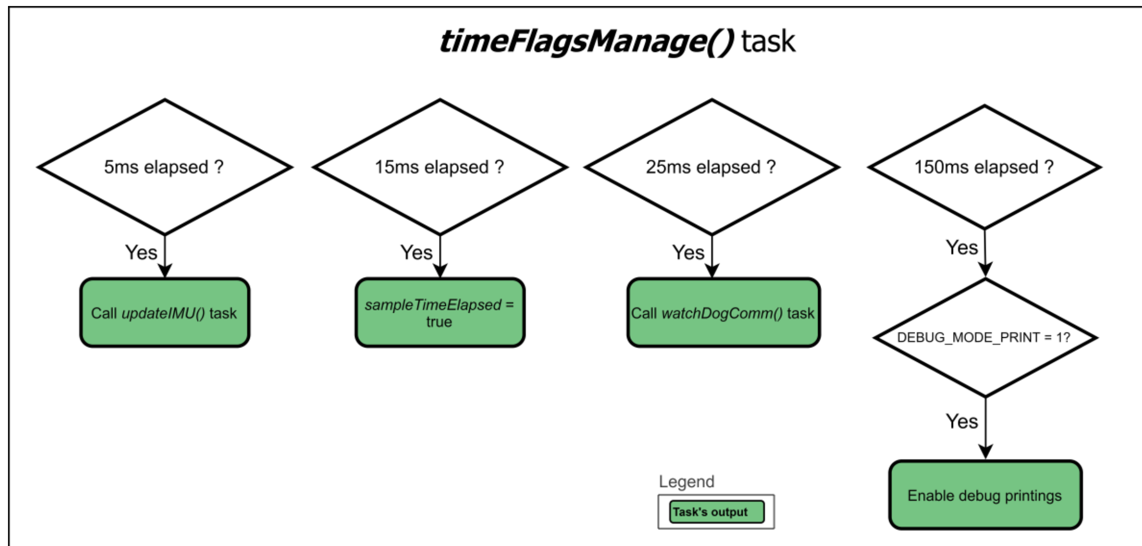


Figure 5-16. Drone's Firmware Structure – `timeFlagsManage()` task

- **commUpdate() task.**

The `commUpdate()` task serves as an interface between the radio module and the firmware. It is structured as a state machine with three possible status, stored in `commState`. The initial one is the status 'S', where we wait until receiving the first byte of the messages header. Once received the status is updated to 'T', where we expect to get the second one. If anything else received, we return to 'S'. Alternatively, if the full header is correctly received, the status is changed to 'Data', where we read the subsequent 5 bytes and store them in `commData[ ]`.The flag `RCdataReceived` is then set to true, so the other tasks can know that new data is available.

In the next image, we can see a flow diagram of this task.

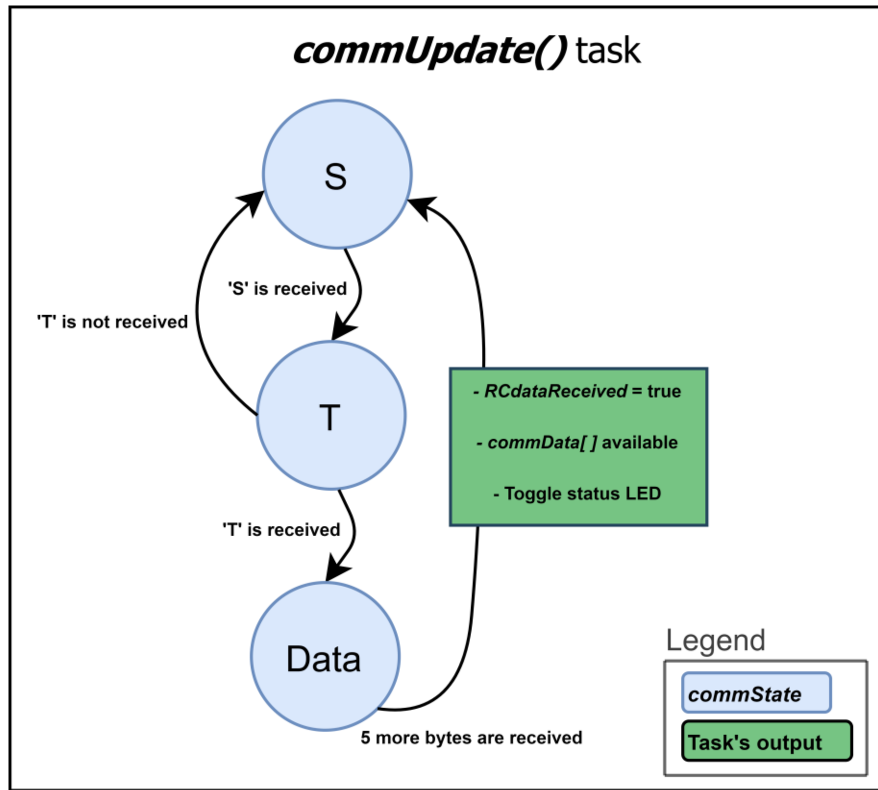


Figure 5-17. Drone's Firmware Structure – `commUpdate()` task

- **updateIMU() task.**

The `updateIMU()` task makes use of the RTIMULib library to estimate the pitch, roll and yaw angles from the three accelerations and angular velocities, provided by the accelerometer and the gyroscope of the IMU (MPU-9250), respectively.

The function is called each 5ms from `timeFlagsManage()`. First, we execute the `imu->IMURead()` RTIMULib function until we get a new set of inertial data.

Then we call `fusion.newIMUData()` to estimate the angles from the previous data.

Although the IMU provides filtered measurements already, in the static tests, we found that this filtering is enough only when the motors are powered-off. When increasing the power, we start to get some noise in the measurements. As described in the mentioned section, we could reduce it by using isolation ground planes, but we still needed to apply further filtering to them.

In the next image, we can see its structure.

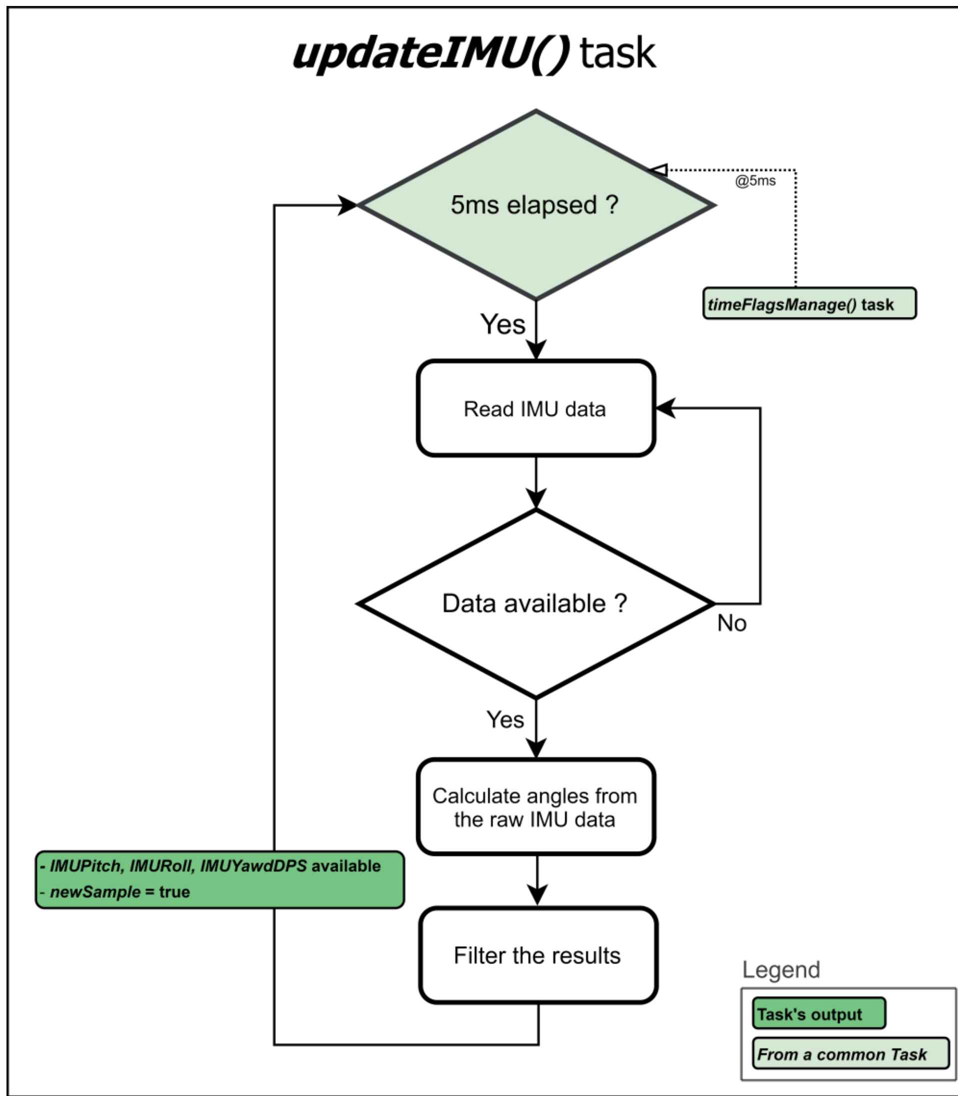


Figure 5-18. Drone's Firmware Structure – `updateIMU()` task

- **watchDogComm() task**

The `watchDogComm()` task supervises the status of the communications, so that if we do not receive data for a certain period of time, a couple of actions are performed:

- If not receiving data for 0.6 seconds (short term communication loss), the motors are powered off.
- If not receiving data for 5 seconds (long term communication loss), the main status of the drone is changed to *Finished*.

The next image shows its structure.

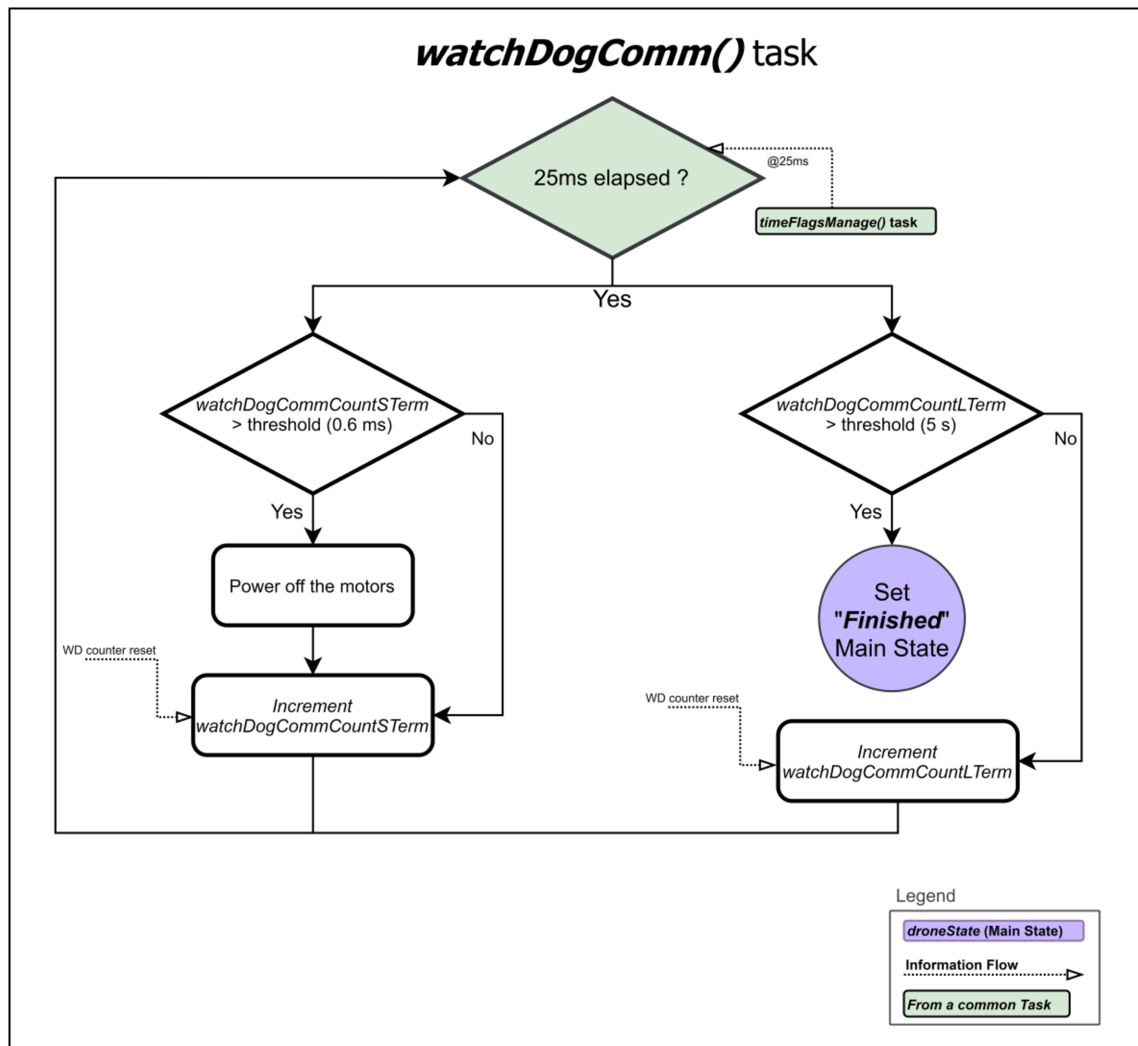


Figure 5-19. Drone's Firmware Structure – `watchDogComm()` task

- **updateUInterface() task**

The `updateUInterface()` task controls the status LED and the buzzer.

Regarding the status LED, it is toggle each 500ms. Note that in the commUpdate() task the LED is also toggled when a new set of data is received, so we will only see the 500ms toggling when no receiving data.

The buzzer is activated depending on the execution of the `beep()` function. When this function is executed from any other task, as new entry is added to `BeepSchedule`. As we see in the example of the figure below, it consists of pairs of an action and the time when to execute it. The structure of the task is also shown in the same figure.

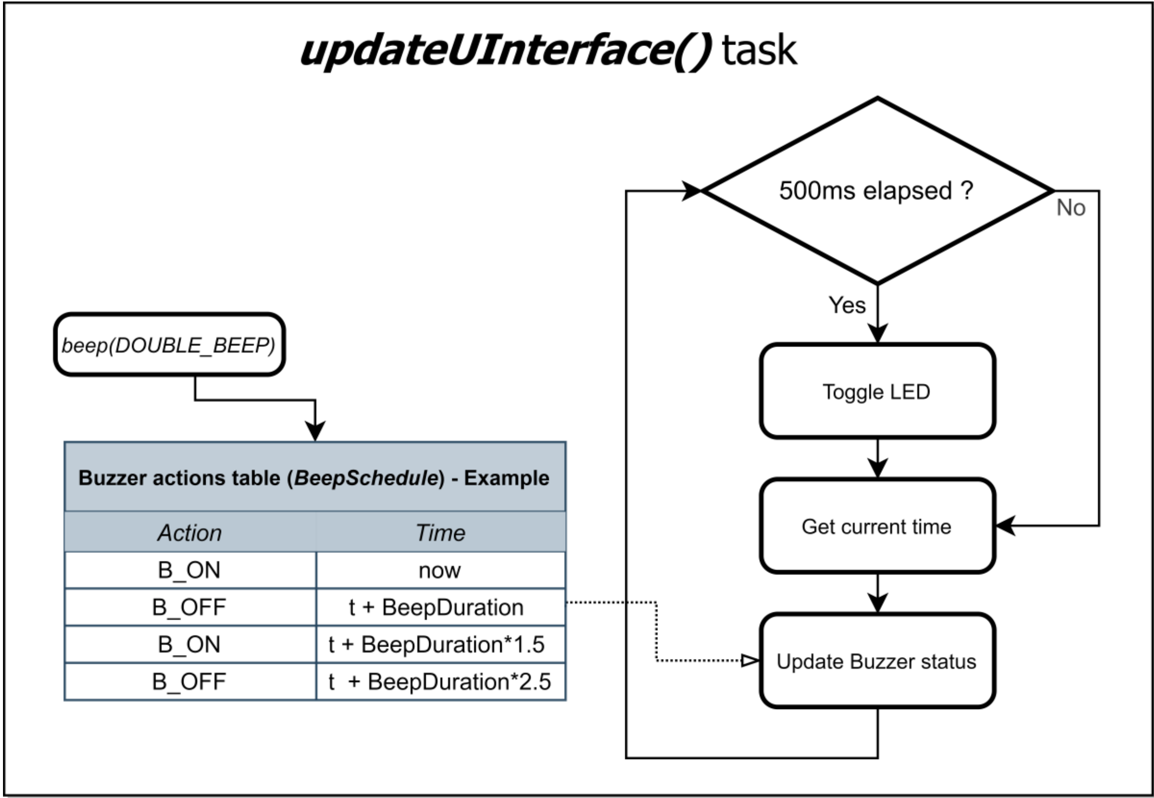


Figure 5-20. Drone's Firmware Structure – `updateUInterface()` task

6 TESTING

In this section, we are going to collect some of the test performed during the firmware development. First, static testing was made, so we could ensure that the behaviour of the firmware was the expected, prior to the flight tests where the PID controllers tuning was finished.

6.1 Static Testing

Regarding the static testing, two types of test were made:

1) **Timing Tests**

The timing tests allows us to determine whether the time requirements are fulfilled. Those are done with drone fully assembled except the motor's connections. The PCB is powered up by the drone's battery as it is during the flights. The tools used are an 8-channel 24MHz Logic Analyser, plus the software *Saleae Logic 1.2.18*.

In the "*commUpdate_timing*" test evidence below, we observe that the communications are completed within the expected time.

"commUpdate_timing" Test

Action	Condition
Debug Pin high (Ch. 0)	First header byte received
Debug Pin low (Ch. 0)	Last message byte received
Measured Timing	
Typical Time between receptions	30.8ms ± 2ms
Typical Duration of each reception processing	1400 µs ± 100 µs

The readings from the logic analyser are shown in the image below.

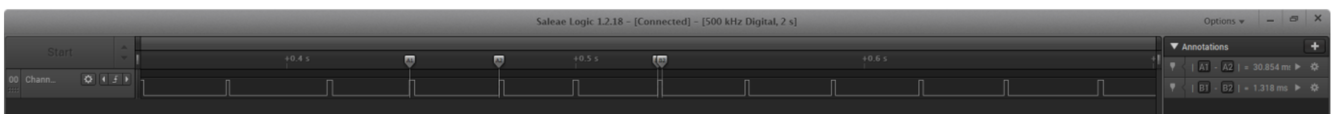


Figure 6-1. *commUpdate_timing* test results – Logic Analyser output.

In the "*motorsUpdate_and_updateIMU_timing*" test, we evaluate the ability to get new measurements from the IMU and apply the calculated control outputs to the motors. As we can see in the table and image below, the sample time is the expected and new measurements are updated at the correct rate.

"motorsUpdate_and_updateIMU_timing" Test

Action	Condition
Debug Pin high (Ch. 2)	`motorsUpdate()` started executing
Debug Pin low (Ch. 2)	`motorsUpdate()` finished executing
Debug Pin high (Ch. 1)	`updateIMU ()` started executing
Debug Pin low (Ch. 1)	`updateIMU ()` finished executing
Debug Pin high (Ch. 0)	`updateIMU ()` started processing a new measurement

Debug Pin high (Ch. 0)	`updateIMU ()` finished processing a new measurement
Measured Timing	
Typical sample time	15.50 ± 0.5ms
Typical period of `updateIMU()` calls	5.20 ± 0.5ms
Typical `updateIMU()` execution time	5.1ms ± 0.05 ms if IMU ready 0.56 ms ± 0ms if IMU not ready
Typical time between new IMU processed measurements	12.00 ± 3ms

The readings from the logic analyser are shown in the image below.

Figure 6-2. *motorsUpdate_and_updateIMU_timing* test results – Logic Analyser output.

2) Bench Tests

These are static tests where we involve the complete system, including the motors. While the drone is secured on the ground, we power it up and send RC data the same way we would do in a real flight. Then we analyse the recorded snapshot data, once the bench test finishes.

The data analysis has been done using the software Matlab R2020a – academic use.

The firmware mode needed is: "`DISABLE_PID = 1`". This way, the input for all the motors will be the same, depending only on the "power" received from the remote controller. The pitch, roll and yaw errors are omitted. Note that by the nature of these tests, we are not able to verify the calculations of the control outputs, as the drone is always in the same position, regardless of the difference in power on each motor.

The steps done to do these test are:

- Power up the drone and the RC controller.
- Perform the arming procedure as it is described in the section *Ready State*.
- Increase the power and maintain the maximum for 2 to 10 seconds.
- Decrease the power to zero.
- Turn off the RC controller and wait 5 seconds, so the "Finished" status is active.
- Send "ST00000" over radio to start receiving the snapshot recorded data.
- Keep a communications sniffer reading the data sent by the drone.
- Analyse the data. Note that the PID parameters are not relevant in this static test.

The typical received data is shown below:

```
%------------------------------

testName = 'T';

KpPitch = 0.00180;
TiPitch = 0.70000;
TdPitch = 0.10000;
KpRoll = 0.00180;
TiRoll = 0.35000;
TdRoll = 0.10000;
KpYaw = 0.00500;
TiYaw = 0.20000;
TdYaw = 0.00000;
FilterWindow = 4;

droneFlyingTimestamp = 17418;
pitchAtStartFlyingTime = -4.90;
rollAtStartFlyingTime = -0.41;

flightData = [
14594, 0, -1.95, 0.23, 0.00, -2.63, 5.00, -3.20, -1.76, -2.70, 0.00
14609, 0, -1.95, 0.23, 0.00, -2.63, 5.00, -3.20, -1.76, -2.69, 0.00
14624, 0, -1.95, 0.23, 0.00, -2.63, 5.00, -3.20, -1.76, -2.69, 0.00

(...)

38308, 0, -1.95, 0.23, 0.00, -2.65, 5.00, -3.20, -1.77, -2.69, 0.00
38323, 0, -1.95, 0.23, 0.00, -2.65, 5.00, -3.20, -1.77, -2.69, 0.00
38340, 0, -1.95, 0.23, 0.00, -2.65, 5.00, -3.20, -1.77, -2.69, 0.00
]';
```

Where we can find the flight data. From the left to the right column, the `flightData` consists of: timestamp (milliseconds), setPntPower (%), setPntPitch (deg), setPntRoll (deg), setPntYawdDPS (deci-deg/s), uPitch (%), uRoll (%), uYaw (%), IMUPitch (deg), IMURoll (deg), IMUYawdDPS (deci-deg/s).

Note that in order to automate as much as possible these tests, we implemented the set of Matlab scripts, found in *Annex A – Matlab Code For Flight Test Analysis*. With these, we can easily plot the data to be analysed.

As we are going to detail below, by doing the bench tests we were able to find one of the greatest issues encountered during the development of this project. We refer to the electromagnetic noise that the ESC's and motors generates, which can greatly affect the measurements of the IMU.

Before going deeply into that problem, it is worth noting that the static tests revealed some other firmware issues, fixed before the flight tests. Namely, for example, the incorrect implementation of the PID structures due to using variable sample times, or the incorrect implementation of the trim functionality that could lead to 'infinite' angle set points sometimes.

Now going into the main issue, in the next image we can see the angles measured by the IMU during a bench test, in function of the power of the motors, in blue. We observe that when the power reaches 15%, the angle measurements start to display strange behaviour. Even though the drone was not pitching, rolling or yawing, when the power is 43%, the IMU reads -40 pitch degrees, 10 roll degrees and a rising yaw movement.

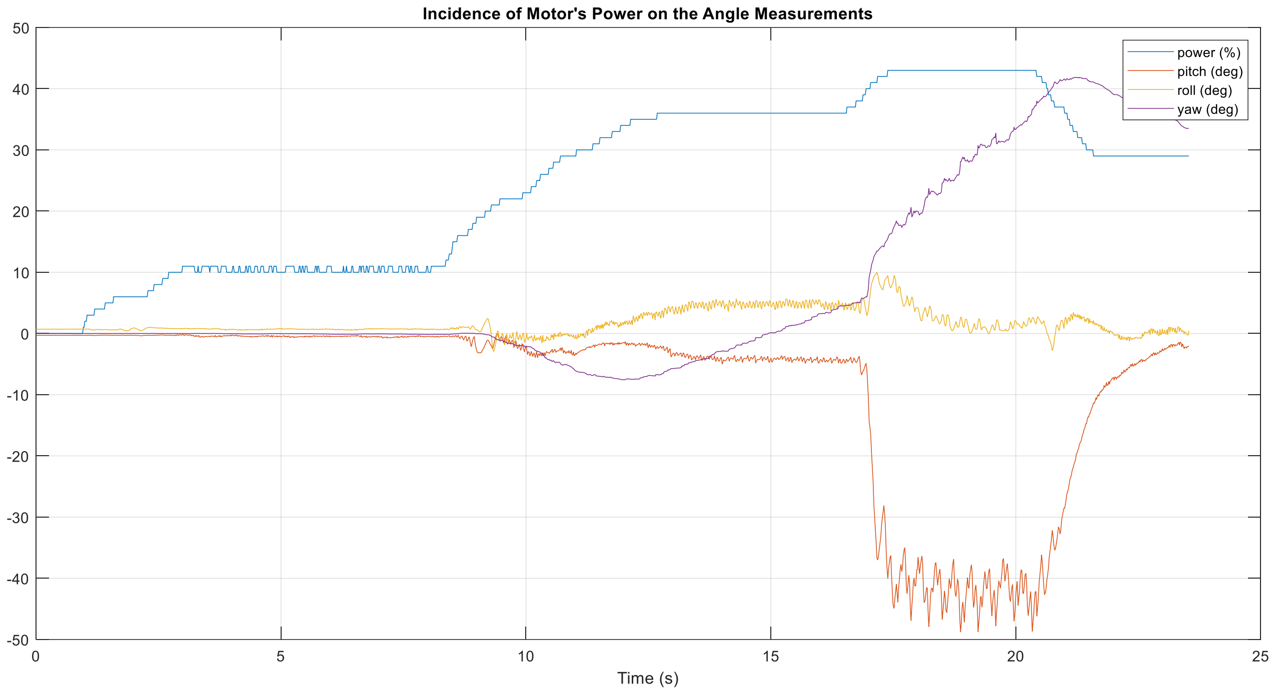


Figure 6-3. Incidence of the Motor's power on the angle measurements, before doing any action to reduce it.

After this test, we repeated it, but now separating the IMU MPU-9250 15cm from is normal position. As we see in the next image, the IMU measurements are now not affected by the motor's power at all.

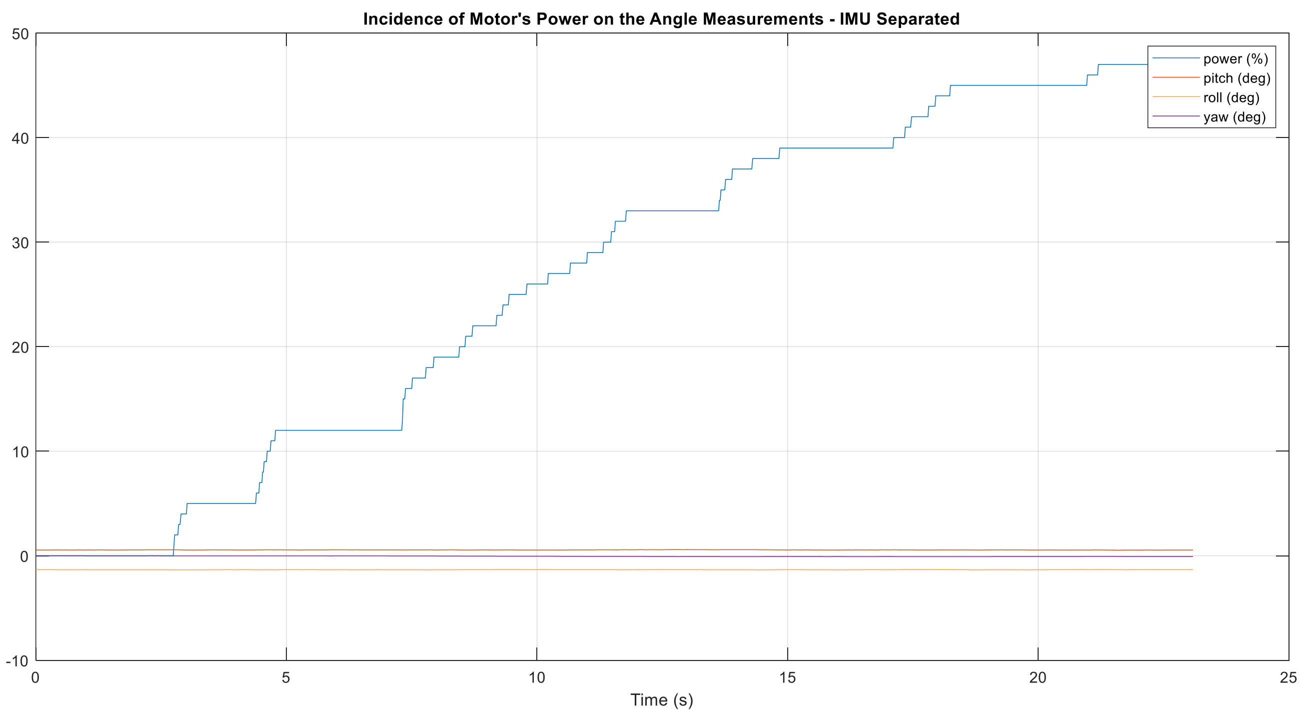


Figure 6-4. Incidence of the Motor's power on the IMU angle measurements when the IMU is separated 15cm.

Therefore, we concluded that the source of error was the electromagnetic noise produced by the power circuit of the motors. The power cables are shown in the next photo. Note that this time we were using a test PCB, prior to assembling the newest hardware. Under the drone, we can see the platform where the drone was secured. This platform was also pressed against the floor using weights during the bench tests.

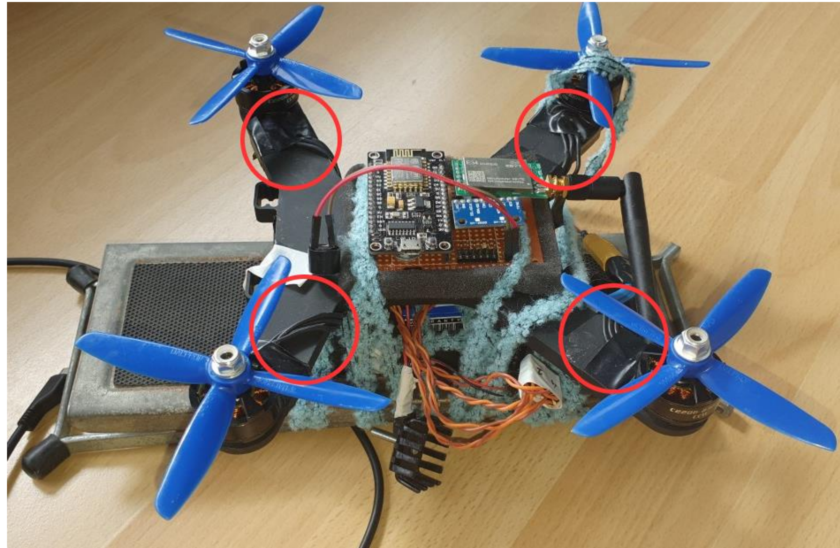


Figure 6-5. Drone configuration during the first bench tests. Note that the motor's power cables are long.

The thin brown cables makes the connection between the PCB and the individual ESC's. These cables must not be a source of any noise, as they are always carrying a PWM signal, that varies only slightly (duty cycle 15-30%) when increasing the power set points. If they were a source of noise, we would see that the IMU measurements are also affected when the power set point is low. This is consistent to the fact these signals are control ones, that does not demand any special amount of current from the microcontroller.

The first action to try to reduce the noise was trimming the power cables as much as possible. The result of that is shown in the next figure.

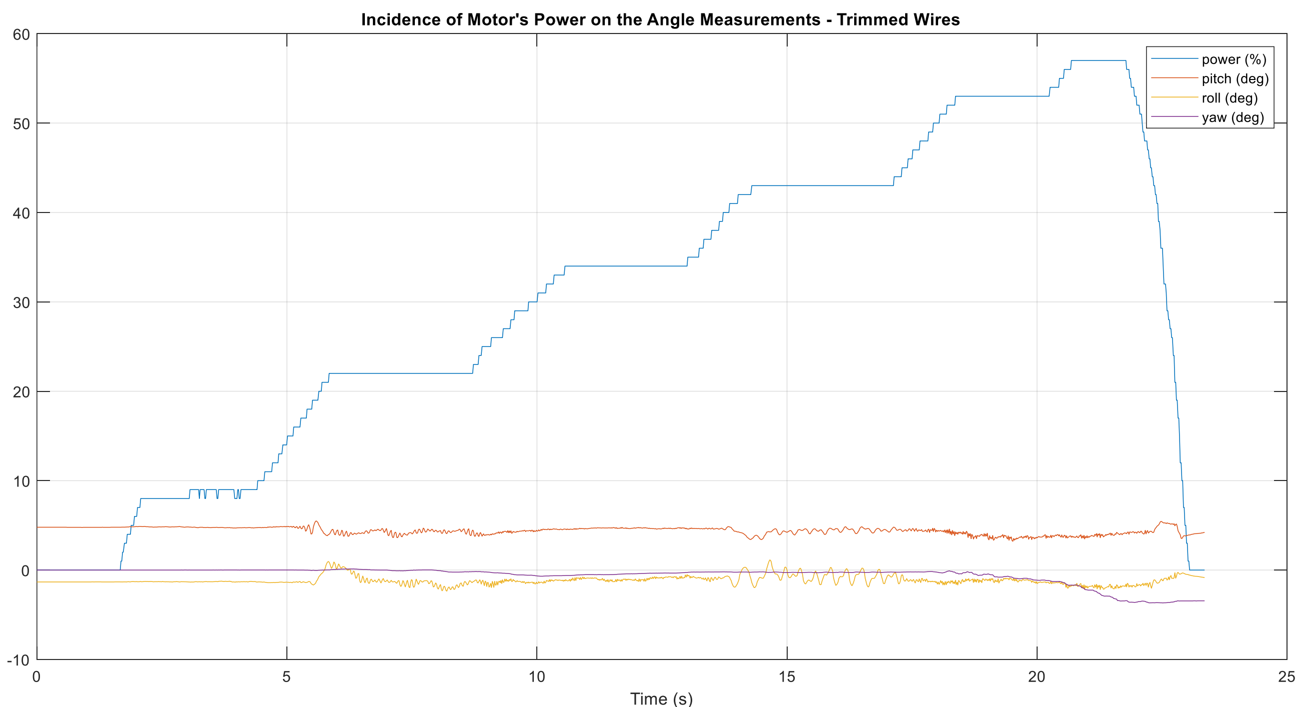


Figure 6-6. Incidence of the Motor's power on the IMU angle measurements. Power wires trimmed.

We can see that the noise was considerably reduced.

If shielding the ESC's, we got some further enhancement, as shown below.

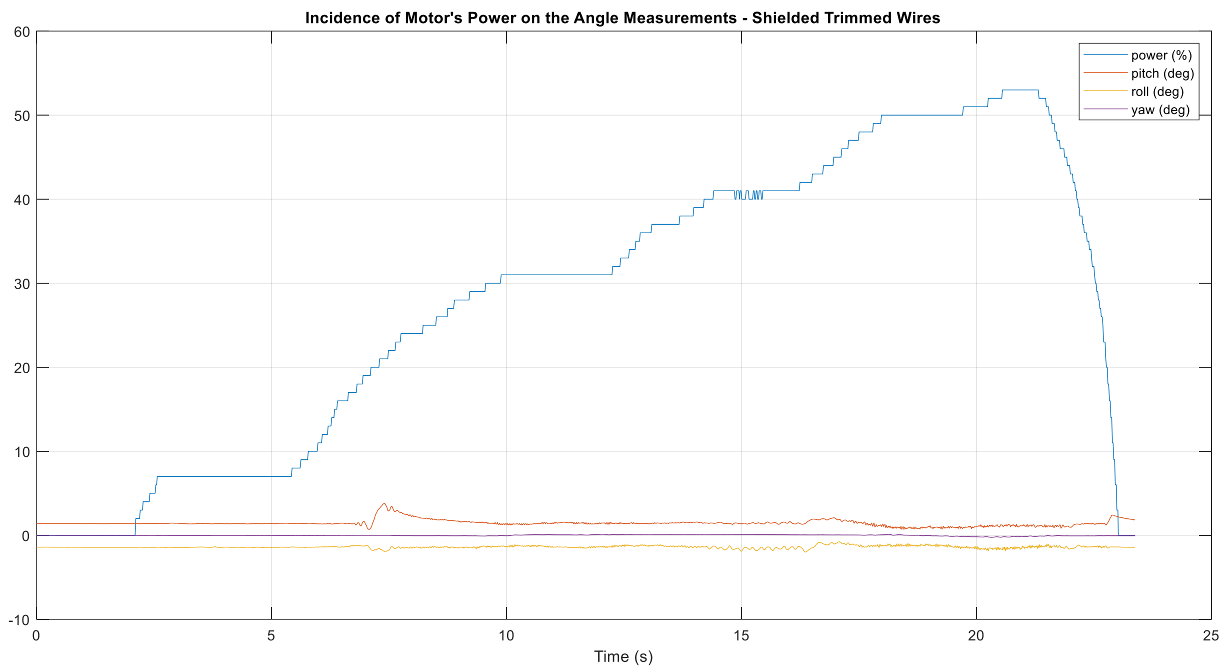


Figure 6-7. Incidence of the Motor's power on the IMU angle measurements. Power wires trimmed and shielded.

Now, increasing the maximum tested power, we can see that the noise increases when the power reaches 65%, as shown in the next image:

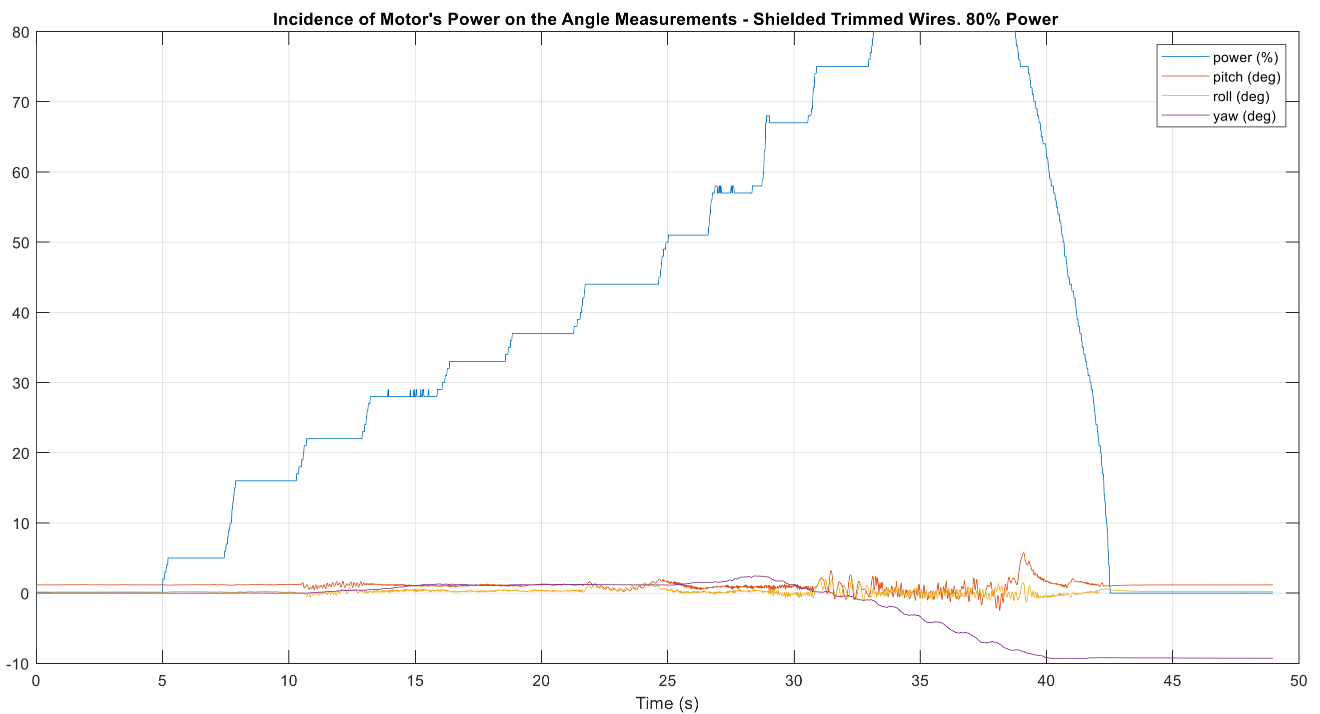


Figure 6-8. Incidence of the Motor's power (up to 80%) on the IMU angle measurements. Power wires trimmed and shielded.

In the next test evidence, we see an improvement after doing these two actions:

- Add an extra ground plane under the PCB, which covers an area bigger than the PCB
- Modify the RTIMULib configuration to recalibrate the IMU, so that the range of the accelerations were changed from ±8g to ±16g and the angular velocities range from 1000 to 2000 degrees per seconds. That way, the noise is divided by two.

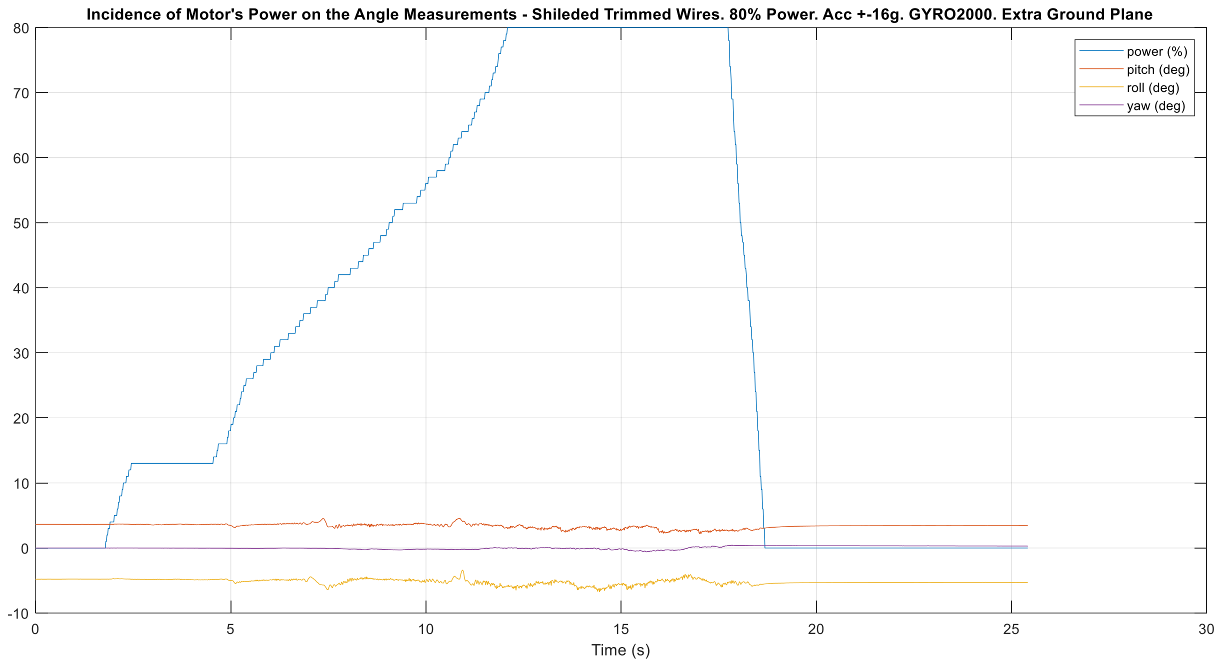


Figure 6-9. Incidence of the Motor's power (up to 80%) on the IMU angle measurements. Power wires trimmed and shielded. RTIMULib reconfigured. Extra ground plane.

The next images show the implemented shields and the extra ground plane.

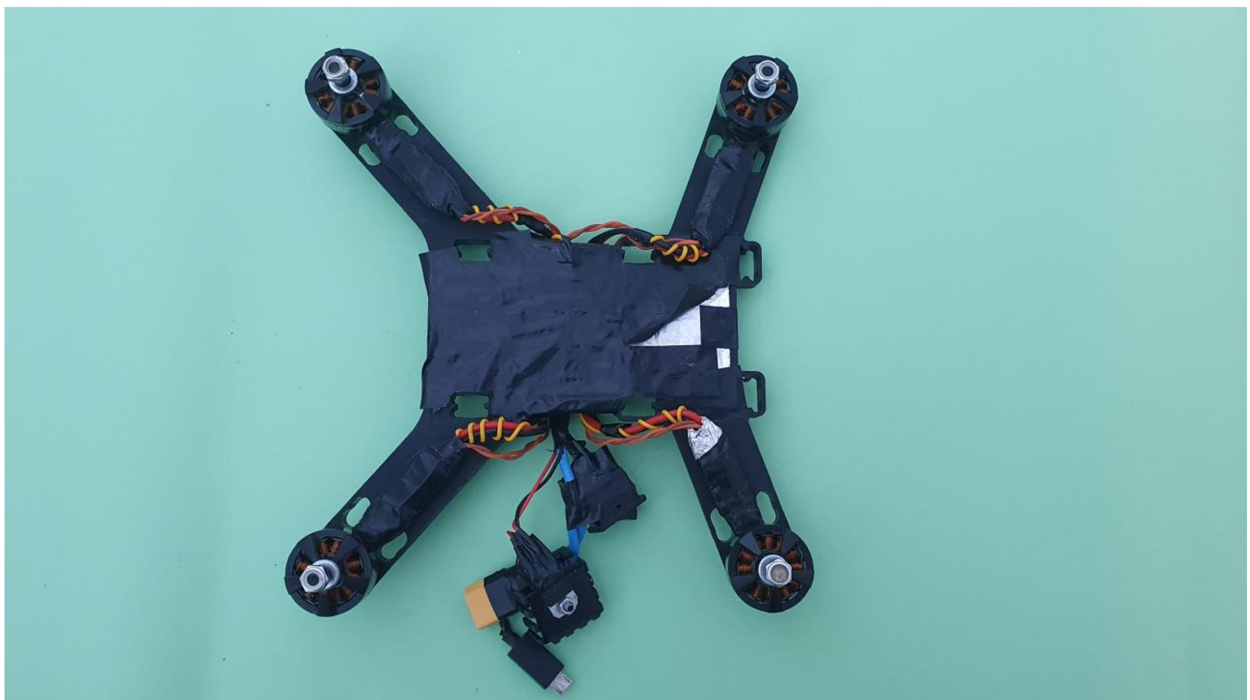


Figure 6-10. Shielded motors and extra ground plane for EMI reduction. Top View

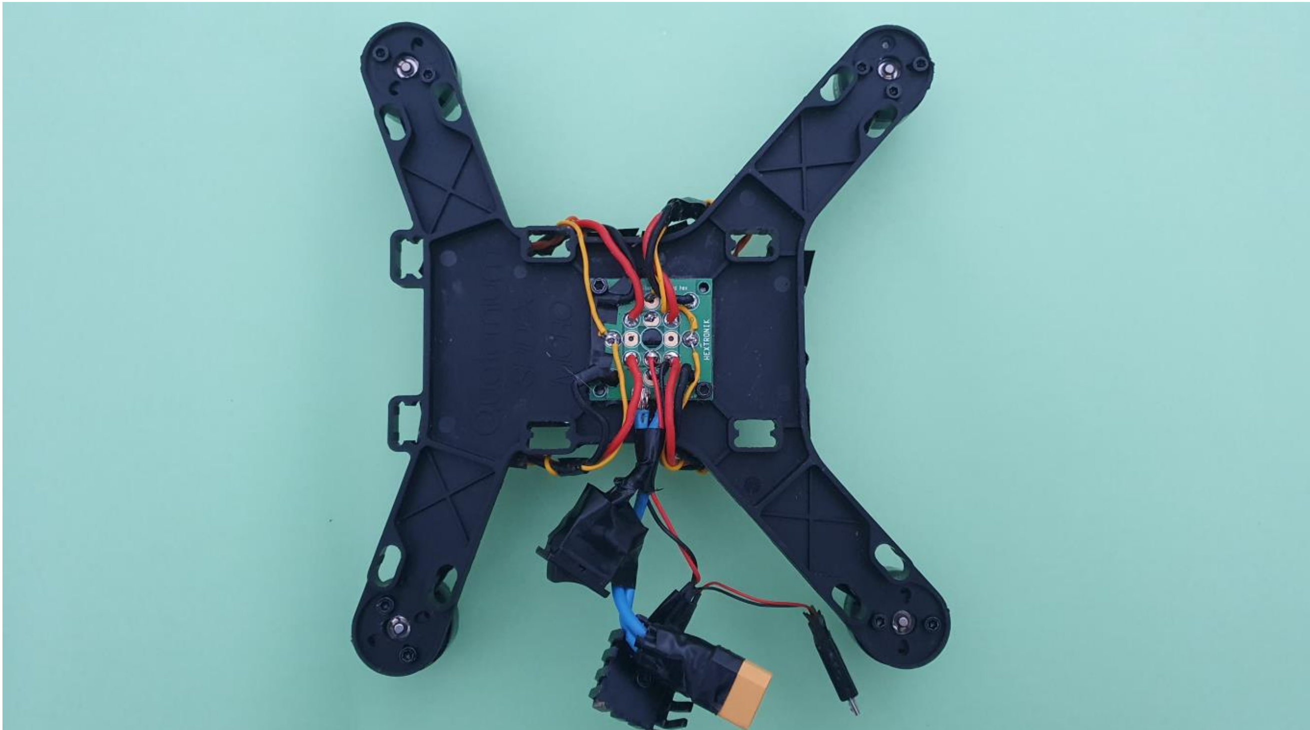


Figure 6-11. Shileded motors and extra ground plane for EMI reduction. Bottom View

6.2 Flight Testing

With the flight tests, we could verify that the complete system works in a real scenario, and adjust the controller's PID parameters following an iterative process.

In the next three images, we can see the data of the first 22.5 seconds of a flight test. The blue plot is the main power received from the remote controller. The brown dashed line is the set point for each angle. The real angles are shown in yellow and the control outputs in purple.

As observed, the angles keep stable around the set point, which was always zero in this case. Note that the yaw graph shows the actual angle, respect to the initial one, not the angular velocity.

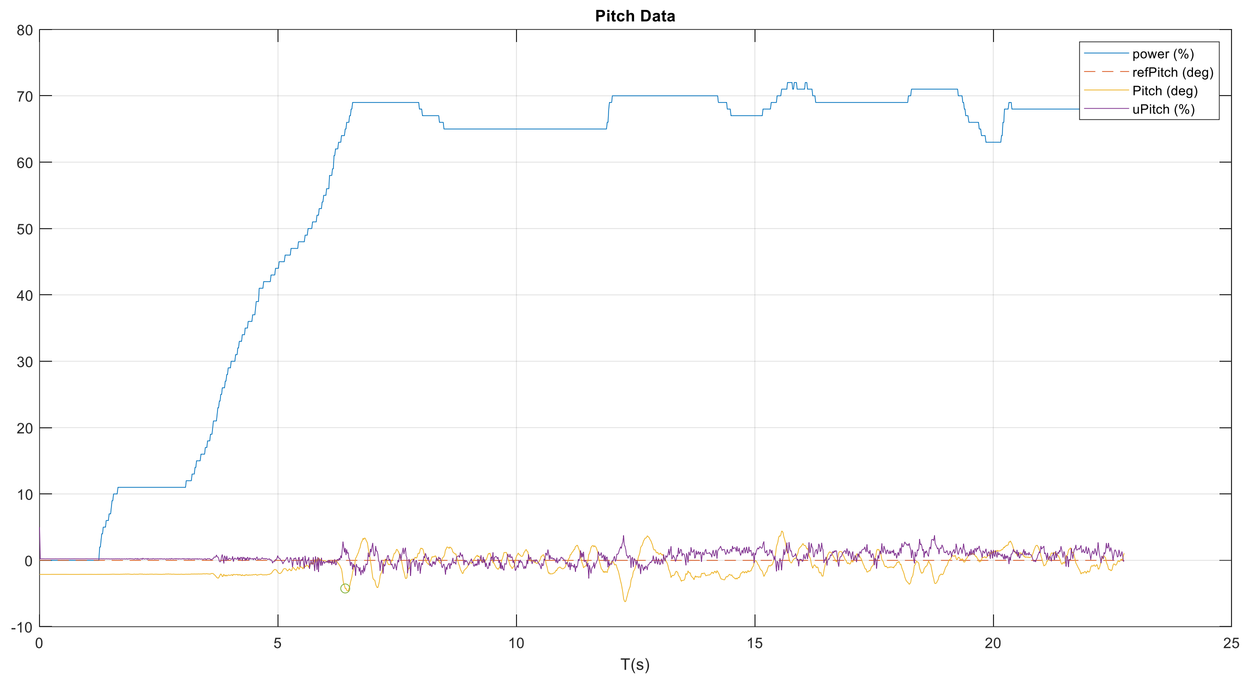


Figure 6-12. Captured data during a flight test - Pitch

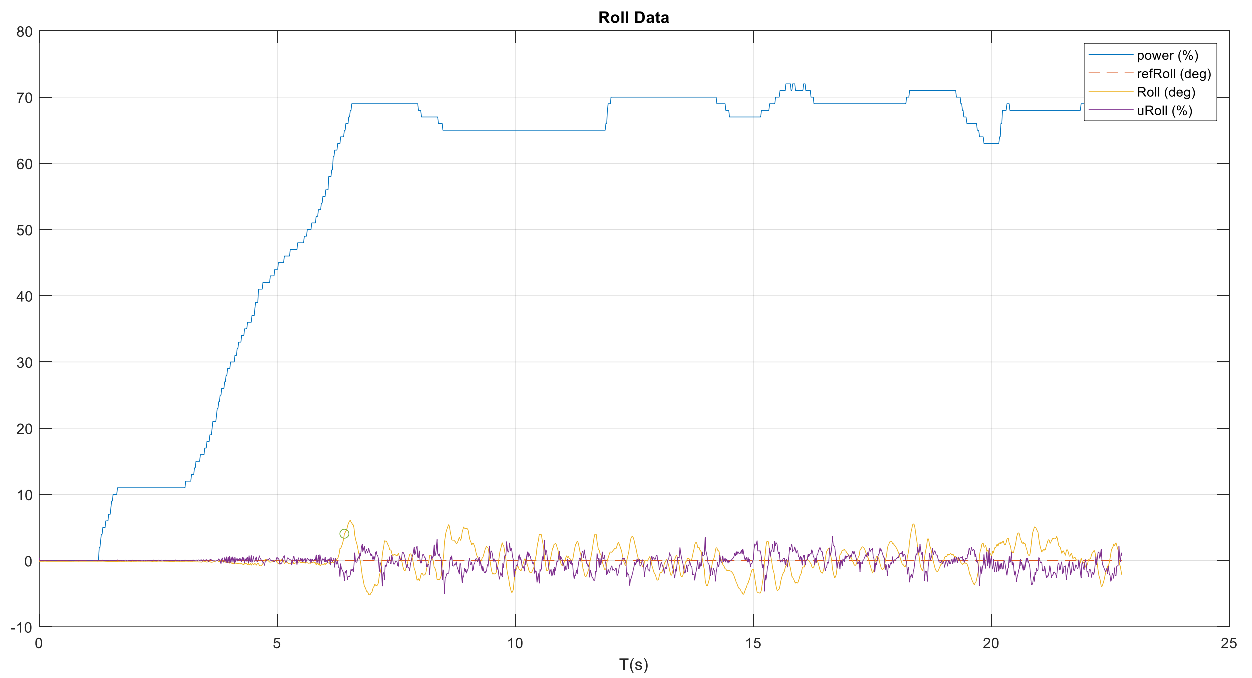


Figure 6-13. Captured data during a flight test - Roll

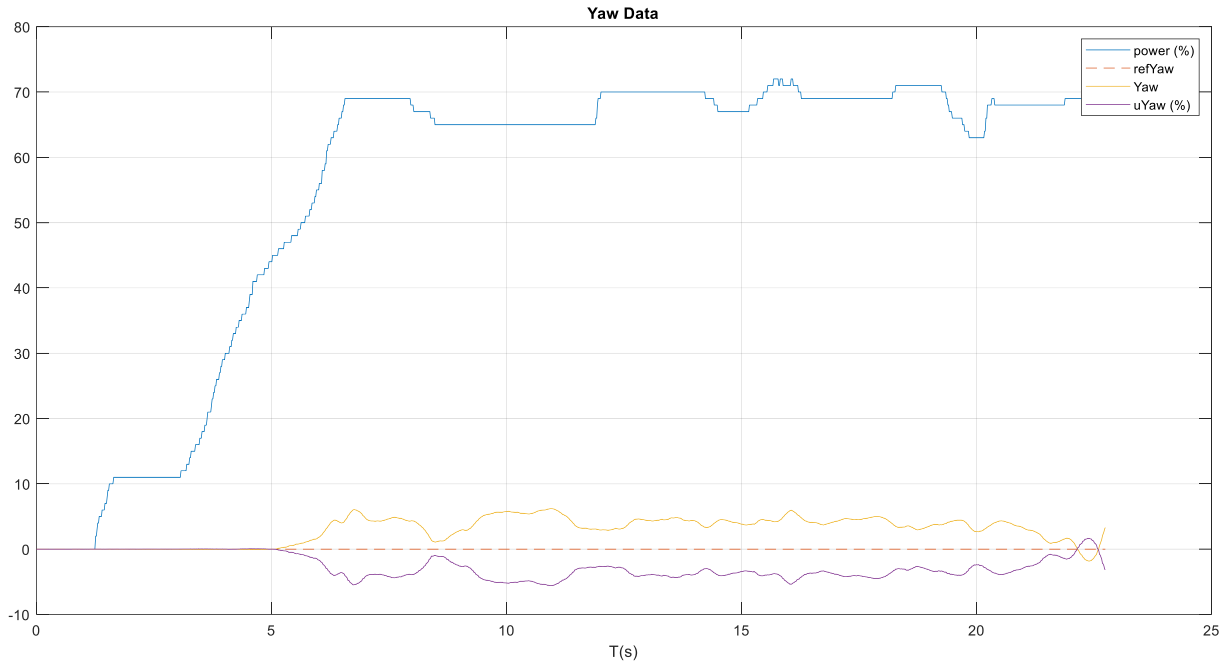


Figure 6-14. Captured data during a flight test - Yaw

Comparing the simulated PID's parameters values, with the final ones after the flight tests:

	Initial values from simulation			**Final values from the flight tests**		
	Kp	Ti	Td	Kp	Ti	Td
Pitch	0.01	0.1	0.1	0.0018	0.7	0.1
Roll	0.01	0.1	0.1	0.0018	0.35	0.1
Yaw	-	-	-	0.005	0.2	0

We can see that the proportional gain needed to be reduced, and the integral part incremented (i.e. to be less active, note that the integral part is present in the denominator of the PID expressions (4-4) from *Control Design – PID*) The derivative values were maintained.

To conclude this section, part of one of the successfully stabilised flight tests is shown in the four frames below.

Figure 6-15. Video frames during a flight test

7 CONCLUSIONS AND FUTURE WORK

As per what we have descripted in all the previous sections, the main conclusion of this project is that it is possible to implement a functional drone quadcopter using low cost microcontrollers, like the ESP8266 and MSP430.

Although it was not intended at the beginning, we can also conclude that during a drone design, the awareness of the existence of electromagnetic noise sources is key.

Collected in this memory, this project could serve as an example of a design and the implementation of a reliable embedded system, endorsed by static and flight tests. In addition, the traditional methodology of control systems was applied; starting from simulations and following with real world iterations, we achieved a controllable dynamic system.

As a future work, the handling of the drone could keep being enhanced to get more responsive and smoother movements. To achieve that, more flight tests and extra features would need to be considered, like dynamic change of PID parameters, or the study of other control approaches.

The implementation of a Kalman Filter for the attitude estimations could also be a future work, so that we do not need to use RTIMULib.

Also, GPS and artificial vision for autopilot functionalities could be added, giving the possibility to recover from a possible loss of communications, or perform other high level tasks demanded by the market within the drone's growing industry, like surveillance, footage recording or industrial plants maintenance actions.

LIST OF FIGURES

Figure 2.1. ESP8266 Pinout 17
Figure 3-1. Remote Controller Circuit Schematic. 26
Figure 3-2. Remote Controller Layout. Top view in red and bottom view in blue. 27
Figure 3-3. Printed Circuit. Top layer on the left. Bottom layer on the right. 28
Figure 3-4. Assembled Remote Controller. Top view on the left. Bottom view on the right. 28
Figure 3-5. On-board Circuit Schematic. 29
Figure 3-6. On-board hardware Layout. We can see the top view in red and the bottom one in blue. 30
Figure 3-7. On-board Circuit Printed Board. Top layer on the left and bottom layer on the right. 30
Figure 3-8. On-board Circuit Assembly. 31
Figure 3-9. Drone Assembly. 31
Figure 4-1. In blue: angles to control. In green: direction of rotation of each motor. 32
Figure 4-2. PID Controller Model – Combination of the three controllers. 34
Figure 4-3. Discretisation of the error curve. 35
Figure 4-4. Comparison of the pitch response when the integral part is delayed (in blue) and not (in red). 36
Figure 4-5. General overview of the Matlab Simulink Model 37
Figure 4-6. Simulink – Physical Model 38
Figure 4-7. Simulink – Physical Model. Motors 39
Figure 4-8. Simulink – Physical Model. Hardware 40
Figure 4-9. Simulink – Physical Model. Battery Holder 41
Figure 4-10. Simulink – Visual output of the physical model 41
Figure 4-11. Simulink – Measurements Block 42
Figure 4-12. Simulink – PID Controller Model 43
Figure 4-13. Simulink – PID Implementation 43
Figure 4-14. Comparison of the pitch angle response when using different PID parameters 44
Figure 4-15. Pitch angle response when delaying the integral component of the PID (in blue) 45
Figure 4-16. Pitch angle response when further delaying the integral component of the PID (in blue) 45
Figure 4-17. Pitch angle response when delaying too much the integral component of the PID (in blue) 46
Figure 5-1. Remote Controller Firmware Logic 47
Figure 5-2. MSP430 Clock Frequency configuration parameters 48
Figure 5-3. MSP430 UART configuration parameters 49
Figure 5-4. Overview of the MSP430 ADC peripheral 50
Figure 5-5. Overview of the MSP430 Timer peripheral 51
Figure 5-6. MSP430 Low Power Modes 53
Figure 5-7. User Interface diagram of the Remote Controller 55

Figure 5-8. Message Structure 56
Figure 5-9. Drone's Firmware Structure – Main State Machine 57
Figure 5-10. Drone's Firmware Structure – Start-up State 58
Figure 5-11. RTIMULib-Arduino Copyright Notice and Permission Notice 59
Figure 5-12. Drone's Firmware Structure – Console State 60
Figure 5-13. Drone's Firmware Structure – Ready State 62
Figure 5-14. Middle and Arming positions in the Remote Controller 62
Figure 5-15. Drone's Firmware Structure – Running State 63
Figure 5-16. Drone's Firmware Structure – `timeFlagsManage()` task 70
Figure 5-17. Drone's Firmware Structure – `commUpdate()` task 71
Figure 5-18. Drone's Firmware Structure – `updateIMU()` task 72
Figure 5-19. Drone's Firmware Structure – `watchDogComm()` task 73
Figure 5-20. Drone's Firmware Structure – `updateUInterface()` task 74
Figure 6-1. *commUpdate_timing* test results – Logic Analyser output. 75
Figure 6-2. *motorsUpdate_and_updateIMU_timing* test results – Logic Analyser output. 76
Figure 6-3. Incidence of the Motor's power on the angle measurements, before doing any action to reduce it. 78
Figure 6-4. Incidence of the Motor's power on the IMU angle measurements when the IMU is separated 15cm. 78
Figure 6-5. Drone configuration during the first bench tests. Note that the motor's power cables are long. 79
Figure 6-6. Incidence of the Motor's power on the IMU angle measurements. Power wires trimmed. 79
Figure 6-7. Incidence of the Motor's power on the IMU angle measurements. Power wires trimmed and shielded. 80
Figure 6-8. Incidence of the Motor's power (up to 80%) on the IMU angle measurements. Power wires trimmed and shielded. 80
Figure 6-9. Incidence of the Motor's power (up to 80%) on the IMU angle measurements. Power wires trimmed and shielded. RTIMULib reconfigured. Extra ground plane. 81
Figure 6-10. Shielded motors and extra ground plane for EMI reduction. Top View 81
Figure 6-11. Shileded motors and extra ground plane for EMI reduction. Bottom View 82
Figure 6-12. Captured data during a flight test - Pitch 83
Figure 6-13. Captured data during a flight test - Roll 83
Figure 6-14. Captured data during a flight test - Yaw 84
Figure 6-15. Video frames during a flight test 85

REFERENCES

[1] https://www.espressif.com/en/products/socs/esp8266, ESPRESSIF, ESP8266 Overview

[2] https://en.wikipedia.org/wiki/ESP8266, Features, SDKs, Ai-Thinker modules.

[3] http://www.ebyte.com/en/product-view-news.aspx?id=153

[4] MPU-9250 Product Specification / PS-MPU-9250A-01, Revision: 1.1

[5] MSP430G2x53, MSP430G2x13 Mixed Signal Microcontroller datasheet (Rev. J)

[6] https://en.wikipedia.org/wiki/PID_controller. PID controller.

[7] Controllers Design. Control Engineering – Chapter 6. Teodoro Alamo Cantarero.

[8] https://www.mathworks.com/help/aeroblks/quadcopter-project.html

[9] MSP430x2xx Family Users Guide (Rev J).

[10] http://www.ebyte.com/en/product-view-news.aspx?id=153, E34-2G4D20D User Manual

[11] https://oscarliang.com/brushed-vs-brushless-motor/

ANNEX A – MATLAB CODE FOR FLIGHT TEST ANALYSIS

```
%% Get data from the snapshots received
T = flightdata(1,:);
droneFlyingTimestamp = (droneFlyingTimestamp-T(1))/1000;
T=T-T(1);
T=T/1000;
power=flightdata(2,:);
refPitch=flightdata(3,:);
refRoll=flightdata(4,:);
refYaw=flightdata(5,:);
uPitch=flightdata(6,:);
uRoll=flightdata(7,:);
uYaw=flightdata(8,:);
pitch=flightdata(9,:);
roll=flightdata(10,:);
yaw=flightdata(11,:);

%% Calculate the error
errorPitch = -(refPitch-pitch);
errorRoll = -(refRoll-roll);
errorYaw = -(refYaw-yaw);

%% Plots Compare - Pre
previousTestName = '';
prevTData = load (previousTestName);
startIndexCurrentData = find(T==droneFlyingTimestamp);
startIndexPreviousData = find(prevTData.T==prevTData.droneFlyingTimestamp);
timeDiff = droneFlyingTimestamp - prevTData.droneFlyingTimestamp;


%% Plots
%%% Combined Angles %%%
figure
plot(T,power), grid on
hold on,
plot(T,pitch);
plot(T,roll);
plot(T,yaw);
legend('potencia (%)','pitch (deg)','roll (deg)','yaw (deciDeg/s)')
title(['Angulos ' testName])

%%% Pitch %%%
figure,
plot(T,power),hold on,
plot(T,refPitch,'--')
plot(T,pitch)
plot(T,uPitch)
plot(droneFlyingTimestamp, pitchAtStartFlyingTime, 'o')
grid on,
legend('power (%)','refPitch (deg)','Pitch (deg)','uPitch (%)')
title([testName 'Pitch. Kp = ' num2str(KpPitch) ' Td = ' num2str(TdPitch) ' Ti = '
num2str(TiPitch)])
xlabel('T(s)')

%%% Roll %%%
figure,
plot(T,power),hold on,
plot(T,refRoll, '--')
plot(T,roll)
plot(T,uRoll)
plot(droneFlyingTimestamp, rollAtStartFlyingTime, 'o')
grid on,
```

```
legend('power (%)','refRoll (deg)','Roll (deg)','uRoll (%)')
title([testName 'Roll. Kp = ' num2str(KpRoll) ' Td = ' num2str(TdRoll) ' Ti = '
num2str(TiRoll)])
xlabel('T(s)')

%%% Yaw %%%
figure,
plot(T,power),hold on,
plot(T,refYaw,'--')
plot(T,yaw)
plot(T,uYaw)
grid on,
legend('power (%)','refYaw','Yaw','uYaw (%)')
title([testName 'Yaw. Kp = ' num2str(KpYaw) ' Td = ' num2str(TdYaw) ' Ti = '
num2str(TiYaw)])
xlabel('T(s)')


%% Plots - Compare flight tests
figure, % Pitch
plot(T((find(T==droneFlyingTimestamp)-100):end),pitch((find(T==droneFlyingTimestamp)-
100):end)), hold on,
plot(prevTData.T((find(prevTData.T==prevTData.droneFlyingTimestamp)-
100):end)+timeDiff,prevTData.pitch((find(prevTData.T==prevTData.droneFlyingTimestamp)-
100):end),'--')
plot(droneFlyingTimestamp, pitchAtStartFlyingTime, 'o')
plot(prevTData.droneFlyingTimestamp+timeDiff, prevTData.pitchAtStartFlyingTime, 'o')
grid on,
legend(['pitch (curr) -> (Kp = ' num2str(KpPitch) ' Td = ' num2str(TdPitch) ' Ti = '
num2str(TiPitch) ')'] , ...
       ['pitch (prev) -> (Kp = ' num2str(prevTData.KpPitch) ' Td = '
num2str(prevTData.TdPitch) ' Ti = ' num2str(prevTData.TiPitch) ')'])
xlabel('T(s)')
ylabel('deg')
title(['Pitch. ' testName ' vs ' prevTData.testName])


figure, % roll
plot(T((find(T==droneFlyingTimestamp)-100):end),roll((find(T==droneFlyingTimestamp)-
100):end)), hold on,
plot(T((find(T==droneFlyingTimestamp)-
100):end),refRoll((find(T==droneFlyingTimestamp)-100):end))
plot(prevTData.T((find(prevTData.T==prevTData.droneFlyingTimestamp)-
100):end)+timeDiff,prevTData.roll((find(prevTData.T==prevTData.droneFlyingTimestamp)-
100):end),'--')
plot(prevTData.T((find(prevTData.T==prevTData.droneFlyingTimestamp)-
100):end)+timeDiff,prevTData.refRoll((find(prevTData.T==prevTData.droneFlyingTimestamp
)-100):end),'--')
plot(droneFlyingTimestamp, rollAtStartFlyingTime, 'o')
plot(prevTData.droneFlyingTimestamp+timeDiff, prevTData.rollAtStartFlyingTime, 'o')
grid on,
legend(['roll (curr) -> (Kp = ' num2str(KpRoll) ' Td = ' num2str(TdRoll) ' Ti = '
num2str(TiRoll) ')'] , ...
        'refRroll (curr)',...
       ['roll (prev) -> (Kp = ' num2str(prevTData.KpRoll) ' Td = '
num2str(prevTData.TdRoll) ' Ti = ' num2str(prevTData.TiRoll) ')'],...
        'refRroll (prev)')
xlabel('T(s)')
ylabel('deg')

title(['Roll. ' testName ' vs ' prevTData.testName])
```