



UNIVERSIDAD DE MÁLAGA



Grado en Ingeniería Informática.
Mención: Tecnologías de la Información

Dron automatizado con Arduino
Automated drone with Arduino

Realizado por

Juan de Dios Estrella Cortecero

Tutorizado por

Luis Manuel Llopis Torres

Departamento

Departamento Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, (mes y año)
Septiembre del 2020

Resumen

Este proyecto trata el diseño y construcción de un Dron tipo Quadcopter usando el microcontrolador Arduino, es decir, un Dron de 4 hélices. Esta memoria está dividida en tres grandes bloques:

- El primer bloque se centra en los conceptos generales que se necesitan saber, el diseño estructural y el diagrama eléctrico.
- El segundo bloque se centra en la parte electrónica, donde se detalla todos los componentes que se han usado y la forma de comprobar su funcionamiento. Con esto nos referimos a la batería, ESC, Arduino, motores, sensores, cámara y el Bluetooth.
- El tercer bloque se detalla todo el software realizado. Es decir, todos los algoritmos implementados y los diferentes modos de vuelo.

Abstract

This project involves the design and construction of a Quadcopter-type drone using the Arduino microcontroller, i.e. a 4-propeller drone. This memory is divided into three large blocks:

- The first block focuses on the general concepts that need to be known, the structural design and the electrical diagram.
- The second block focuses on the electronic part, which details all the components that have been used and how to check how they work. By this we mean batteries, ESC, Arduino, motors, sensors, camera and Bluetooth.
- The third block details all the software performed. That is, all the algorithms implemented and the different flight modes.

Palabras Claves:

- Dron Arduino
- Dron Bluetooth
- Quadcopter
- Pixy2
- ESC

Key Word:

- Arduino Dron
- Bluetooth Dron
- Quadcopter
- Pixy2
- ESC

Indices

Resumen	1
Abstract.....	1
Palabras Claves.....	2
Key Word	2
I. Introducción.....	5
I.I. Motivación.....	5
I.II. Objetivos	5
I.III. Contexto tecnológico.....	5
II. Cuerpo del Trabajo	6
II.I. Concepto General y Diseño.....	6
Estructura	7
Diagrama eléctrico.....	9
II.II. Parte Hardware.....	11
Frame 450	11
Tira Led RGBW	11
Bluetooth HC-06	11
Resistencia 270kΩ y 180KΩ	12
Batería Lipo	12
Motor Brushless	14
ESC	14
Hélices	16
Placa Distribución de Potencia.....	18
Arduino Uno.....	19
MPU6050.....	19
Mosfet IRFZ44N	25
Pixy2	25

Imax B6.....	27
II.III. Parte Software	30
Conceptos Generales	30
Modo Acrobático.....	58
Modo Estable	62
Modo Despegue	64
Modo Aterrizaje	65
Modo Automático	65
II.IV. Relación de coste.....	66
II.V. Metodología de Trabajo.....	67
II.VI. Conclusión	68
II.VII. Líneas Futuras	68
III. Bibliografía	69
IV. Manual Usuario.....	70

I. Introducción

I.I. Motivación

La razón de elaboración del proyecto proviene del interés por la electrónica y los elementos interactivos cuyos resultados son muy visuales y entretenidos. Además supone un reto personal el demostrar ser capaz de implementar un dispositivo montado desde cero capaz de realizar vuelos manejados desde un dispositivo Bluetooth y realizar un modo de seguimiento automático de un objeto o un circuito.

I.II. Objetivos

El objetivo del presente proyecto consiste en la realización de un Dron desde cero y dotarlo de automatización.

Realizando un estudio completo desde el diseño hardware y software, contemplando los diferentes contratiempos de los valores reales y ficticios, como el estudio del equilibrio del mismo de forma automática frente a perturbaciones del viento. En este proyecto se realiza tanto trabajos a nivel de Hardware como del software de control, desde la obtención de parámetros mediante sensores hasta la modificación del comportamiento de los actuadores en función de ellos.

En el desarrollo del Hardware se incluye todos los elementos electrónicos, sus esquemas de montajes y consejos a la hora de montarlos.

I.III. Contexto tecnológico

Las aplicaciones civiles a las que son destinados los drones van aumentando con rapidez en los últimos años al permitir labores que por riesgo o dificultad de accesibilidad no son rentables de realizar de otra manera, las principales son: grabación de videos y fotografías, prevención de incendios forestales, labores de vigilancia, etc.

La principal ventaja que suponen los Drones reside en la inexistencia de personas a bordo lo que elimina el riesgo de muerte de los pilotos y permite realizar

funciones que no serían posibles con aeronaves tripuladas como investigación en zonas de alta toxicidad química y radiológica.

Por otro lado están otros aspectos como la posibilidad de automatización del aparato, siendo capaz de actuar por sí mismo.

A pesar de las grandes posibilidades que nos brindan los Drones, presentan también una serie de desventajas técnicas, económicas y éticas.

Las desventajas técnicas tienen en cuenta que este elemento no deja de ser una máquina, controlada a distancia a través de comunicación remota y que requiere una fuente de energía que se va consumiendo, pueden producirse ciertos fallos de comunicación, quedarse sin batería, reacciones lentas a las indicaciones enviadas.

Las desventajas éticas son posiblemente las más importantes relacionadas con estos aparatos, como suele ocurrir al aparecer nuevas tecnologías siempre surgen muchas preguntas sobre su dirección de desarrollo.

II. Cuerpo del Trabajo

II.I. Concepto General y Diseño

Para definir la posición del Dron se utiliza los 3 ángulos de navegación (ilustración 1, ilustración 2) definidos como Pitch, Roll y Yaw.

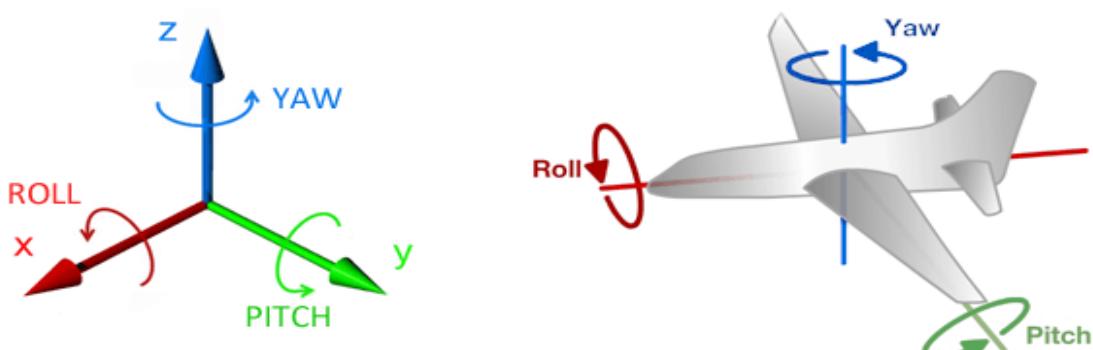


Ilustración 1: Ejes de navegación

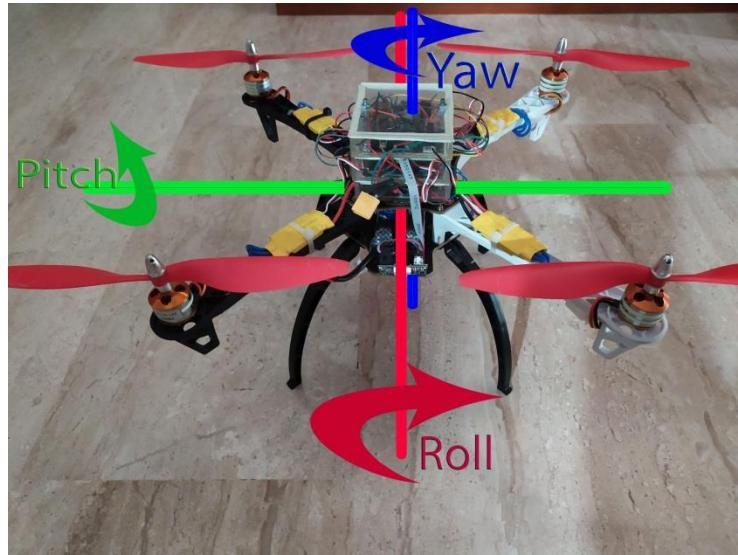


Ilustración 2: ejes del Dron

Inclinando el *Dron* sobre estos ejes haremos que se desplace en la dirección de rotación.

Rotar en el eje Pitch hará a avanzar o retroceder, rotar en el eje *Roll* hará que se incline a la izquierda o derecha, y rotar sobre *Yaw* hará que gire sobre su propio eje vertical. Además de estos tres movimientos, el *Dron* también será capaz de subir o bajar, ganando o perdiendo altura debido al Throttle (velocidad de los motores).

- **Estructura**

Existen dos tipos de configuración que se puede utilizar con un mismo *Frame* (estructura) de *Quadcopter*, la configuración ‘x’ y la configuración ‘+'. Las implicaciones de utilizar una u otra configuración son bastante importantes, ya que esta decisión definirá la orientación del sensor *MPU6050* en el *Frame*.

El *Dron* que se va a construir es de tipo ‘x’. Como vemos en la siguiente (ilustración 3) la configuración que seleccionemos implica que para realizar un mismo movimiento los motores implicados tiene un sentido de giro diferente.

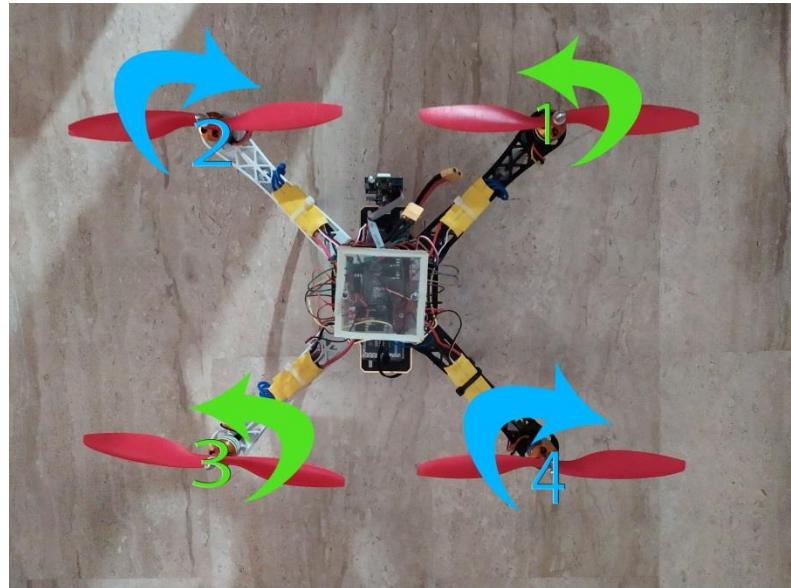


Ilustración 3: Sentido de los motores Dron tipo X

En la configuración tipo ‘x’, para poder avanzar es necesario acelerar los dos motores traseros (3,4) y desacelerar los dos delanteros (1,2), es decir, rotar en el eje *Pitch*

Para inclinar hacia la derecha se acelera los motores de la izquierda (2,3) y reduce los motores de la derecha (1,4), es decir, rotar en el eje *Roll*

Para girar el Dron sobre su eje vertical hacia la derecha, se acelera los motores que giran en sentido horario (2,4) y reduce los motores que giran en sentido antihorario (1,3).

Para ascender el Dron, se acelera todos los motores (1,2,3,4), es decir, aumentamos el Throttle

Para el montaje del Dron, se ha realizado un maquetado con AutoCAD de la caja superior realizada con metacrilato, que contiene toda la circuitería, dando una protección para las vibraciones, golpes y posibles cortocircuitos debido al agua.

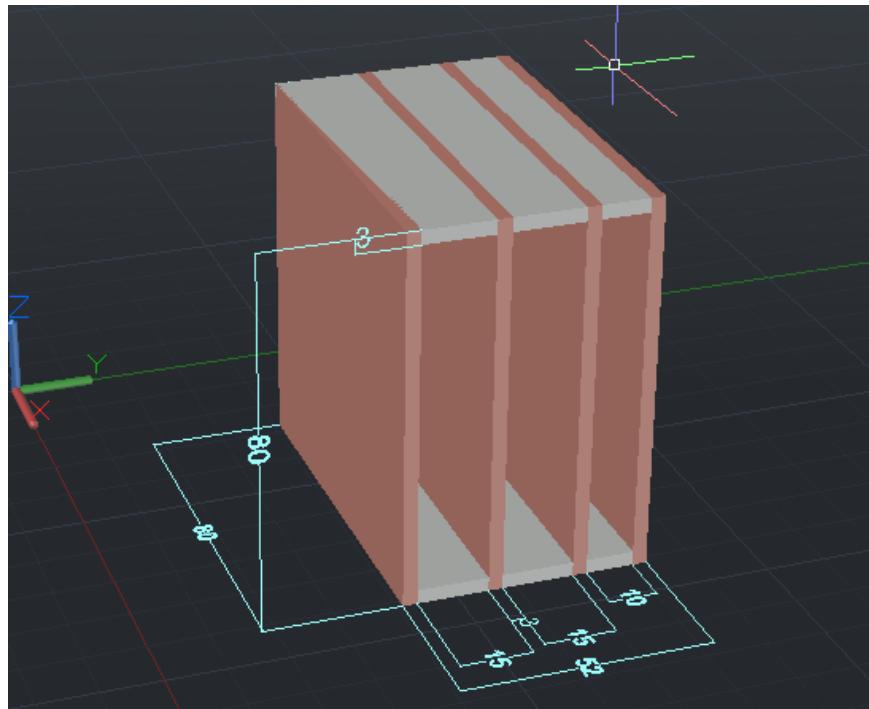


Ilustración 4: Diseño caja metacrilato

• Diagrama Eléctrico

El esquema eléctrico se ha diseñado con Proteus, en el diagrama no se incorpora la cámara pixy2, ya que esta va conectada únicamente por el puerto ISCP.

En el diagrama eléctrico (ilustración 5) se observa que hay conexiones indicada con etiquetas:

Ground Arduino, pertenece al puerto GND de la placa Arduino Uno.

5V Arduino, pertenece al puerto de salida de +5v de la placa Arduino

12v Batería, es una conexión a la salida positiva de la batería LIPO

Ground Batería, es una conexión al negativo de la batería LIPO.

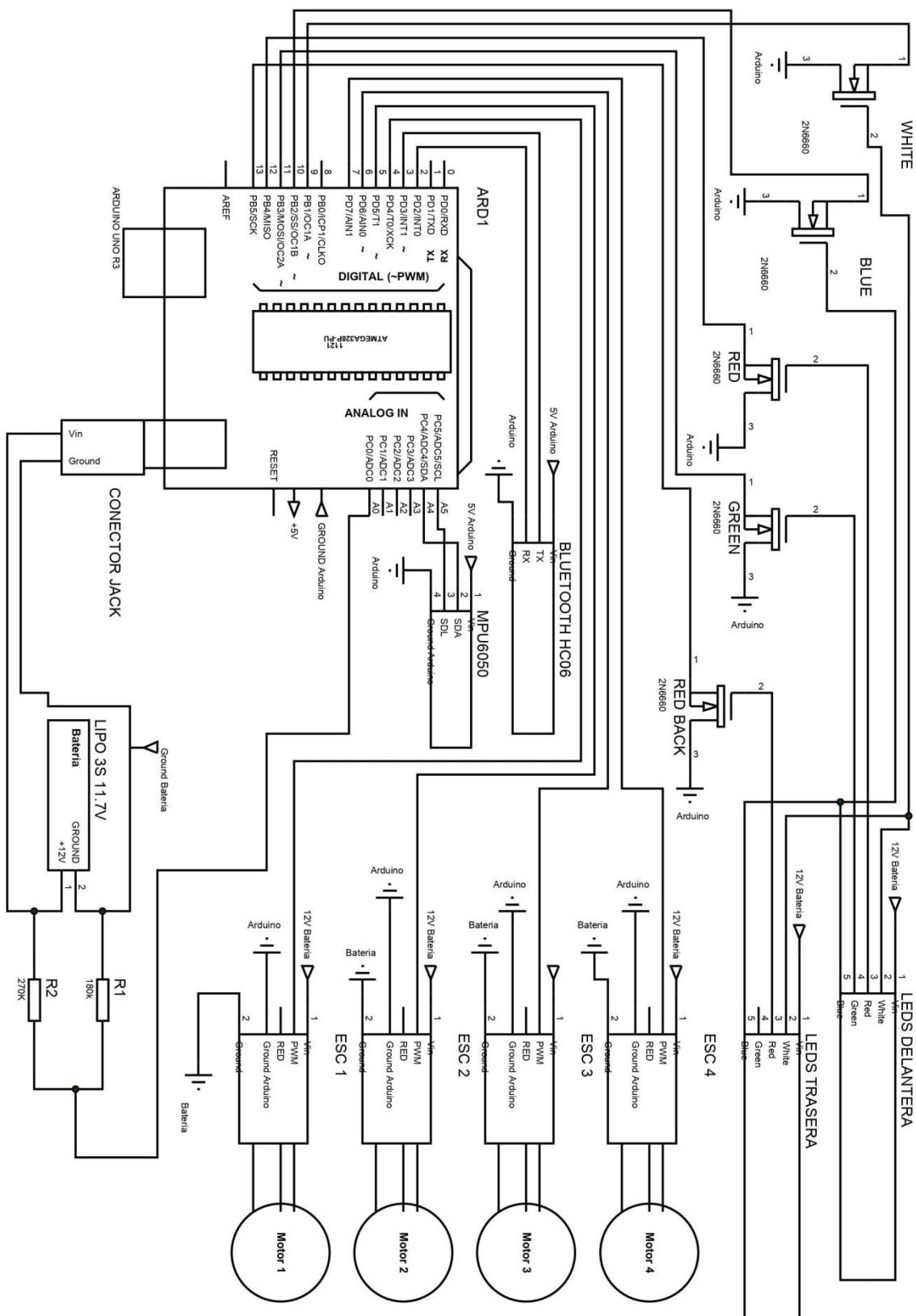


Ilustración 5: Diagrama del circuito Eléctrico

II.II. Parte Hardware

- **Frame 450:** Es la estructura donde vamos a montar todos los componentes hardware (ilustración 6).



Ilustración 6: Frame 450

A partir de este Frame montamos la primera capa de la caja de metacrilato diseñada anteriormente.

- **Tira Led RGBW:** La función RGBW es lo mismo que la función RGB luz multicolor pero además se añade un diodo de luz blanca. Esta tira led tiene ánodo común (los 4 colores comparten el positivo), no es posible regular su intensidad con voltajes, ya que los pines correspondientes a cada color, necesitan conectarse a tierra (0 volts) para prender. Pero si utilizamos un transistor como puerta lógica entre el pin negativo del LED y la tierra, lo que podemos hacer es administrar diferentes intensidades de voltaje a la base del transistor (con PWM) y de esta manera regulamos la cantidad de voltaje que dejamos pasar del pin negativo del LED a tierra. Para comprobar su funcionamiento, se verá más adelante cuando se especifique el transistor Mosfet utilizado. Servirá para indicar los diferentes modos de vuelos, y en caso de algún fallo del Dron.

- **Bluetooth HC-06:** El módulo Bluetooth HC-06 nos permite conectar el proyecto con Arduino a un Smartphone, celular o PC de forma

inalámbrica (Bluetooth), con la facilidad de operación de un puerto serial. La transmisión se realiza totalmente en forma transparente al programador. Todos los parámetros del módulo se pueden configurar mediante comandos AT. El módulo Bluetooth HC-06 viene configurado de fábrica para trabajar como esclavo, es decir, preparado para escuchar peticiones de conexión.

Para comprobar el funcionamiento del dispositivo, ejecuta el código situado en Test/TestBluetooth.cpp, en el caso de ser la primera vez utilizamos el método bluetooth.configure, comentado en el código ya que, este método configura el HC-06 asignando la velocidad, nombre y pin utilizando los comando AT correspondientes del dispositivo. Una vez ejecutado el código TestBluetooth, nos mostrara unas instrucciones a través del Serial Plotter para conectarnos al Bluetooth, ya conectado al Bluetooth nos mostrara por el Serial los datos enviados a la placa Arduino (Ilustración 7).

```
Si no hemos configurado el Bluetooth  
descomentar la siguiente linea  
y cambiar en GlobalVariable.h -> baudrates a 9600  
Ya que por defecto la velocidad HC-06 es 9600  
Nos conectamos al Bluetooth  
Valor leido: A  
Valor leido: B
```

Ilustración 7: Serial Plotter del TestBluetooth.cpp

- **Resistencia 270KOhm y 180KOhm:** Es un componente que dificulta el avance de la corriente eléctrica. Se usa para monitorizar la batería, en el siguiente punto se especifica.
- **Batería Lipo 3s 4200mah 30C:** Batería compuesta de 3 celdas en serie. La tensión de cada celda puede variar en función de la carga de la que disponga (SOC), pero rondará los 3.7V. La batería de 4200mAh (4.2Ah), significa que es capaz de entregar 4.2Amp ininterrumpidamente durante 1h. A esto se le llama descargar la batería a 1C o a corriente nominal.

La corriente máxima de descarga es 30C, lo que equivale a $30 \times 4.2A = 126A$. Entonces la duración de la batería será:

$$Tiempo \text{ (min)} = \frac{\text{Capacidad Batería(Amp * min)}}{\text{Velocidad de Descarga (Amp)}}$$

Con lo cual la duración será aproximadamente de media hora de vuelo.

La tensión de la batería al mínimo será de 10.5V. Es por ello que para monitorizar la batería se incorpora un divisor resistivo (ilustración 8).



Ilustración 8: Divisor resistivo para monitorizar la Batería

Para calibrar correctamente la batería, se mide con un polímetro la entrada +5V respecto de GND y anotar el valor de alimentación real que llega a nuestra placa. Para corregir este offset y no alterar la medida de la tensión de batería.

Para comprobar el funcionamiento del divisor, se ejecuta el código situado en Test/TestBattery.cpp, y se comprueba si el valor que nos muestra por el Serial Plotter es igual al valor real medido con el polímetro (Ilustración 9).

```
Medimos con el polímetro la bateria, y comprobamos si es el mismo voltaje
Voltaje -> 12.50
```



Ilustración 9: Test batería y comprobación con polímetro

⚠ Advertencia: Cuidado a la hora de montar y cablear la batería y el divisor resistivo. Hay que asegurar que el polo positivo y el negativo nunca van a tocarse entre sí.

Cuidado con dejar la batería conectada el consumo de corriente del divisor resistivo es ínfimo, pero puede acabar por descargar por completo la batería. Asegurar que la Batería este bien sujetada y centrada, ya que las baterías Lipo es sensible a la hora de los impactos, produciendo inclusión explosiones.

- **Motor Brushless 1000Kv:** Motores trifásicos sin escobillas, escaso o nulo mantenimiento. Basados en los fundamentos de los de corriente alterna donde un sólo campo magnético provoca el giro. Eso sí, las baterías tienen que ser capaces de dar mucha intensidad pues el consumo puede ir desde los 3-4 amperios hasta 90 o más dando una potencia entre 40 y 1500 vatios o más. Se tiene alimentar las tres fases con tensión alterna de 50Hz, por eso tenemos tres cables de colores para alimentar los motores.

Idealmente, la tensión debería ser perfectamente sinusoidal, pero en la práctica es imposible generar 50Hz ‘limpios’ y sin ruido (armónicos). Una frecuencia de 50Hz equivale a un periodo de 20ms.

El motor girará a una velocidad proporcional a la tensión aplicada. Con lo cual estos giraran a 1000rpm (revoluciones por minuto) por cada voltio de alimentación aplicado.

⚡ **Advertencia:** Asegurar que los motores están bien sujetos al Frame, para evitar accidentes y evitar vibraciones indeseadas.

- **ESC 30Amp:** El controlador de velocidad electrónico ESC 30A, es capaz de definir la velocidad de giro de un motor Brushless mediante la generación de pulsos compatibles con este tipo de motores. Es un circuito electrónico que convierte la tensión DC de entrada (batería) en tensión AC regulable en función de la señal PWM de control que apliquemos.

La señal PWM para gobernar los motores deberá ser de frecuencia constante y de ancho de pulso variable en función de la velocidad de giro que se quiera obtener. Aplicando un pulso de 1ms el motor permanece

parado, con un pulso de 1.5ms el motor girará a mitad de velocidad y ante un pulso de 2ms girará a máxima velocidad (ilustración 10).

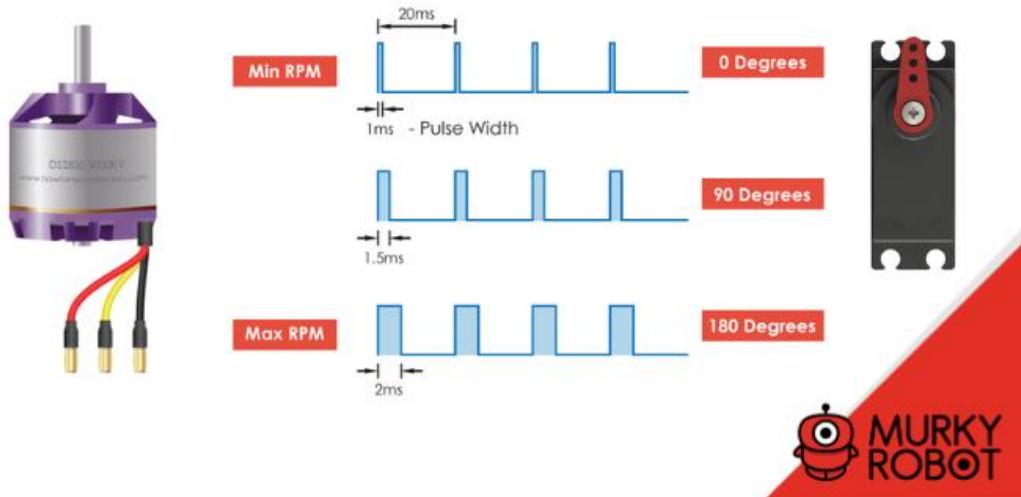


Ilustración 10: Pulso motor Brushless o Servo Motores

La forma de conectar el ESC y el Motor, es aconsejable utilizar conectores o fichas, para una conexión segura y evitar vibraciones (ya que esto puede dañar los componentes) (ilustración 11). Un aspecto importante es marcar los motores con su orden y la dirección de giro (ilustración 12) ya que los motores enfrentados giren en el mismo sentido (ilustración 3). Para variar el sentido de giro basta con intercambiar dos de los cables de alimentación del motor (ilustración 11).



Ilustración 11: Tipo de Conexión ESC con el Motor

Ilustración 12: motores

Para generar las salidas PWM de control se utiliza la librería Servo.h incluida en Arduino ya que como muestra la ilustración 10, el motor se puede manejar como un servo motor entre 0 y 180 grados.

Para comprobar y calibrar los motores, se ejecuta el código situado en Test/TestCalibrarMotores.cpp, y se sigue los siguientes pasos:

*Sin conectar la batería, se conecta la placa Arduino al ordenador y cargamos el programa.

*Seguimos las instrucciones que nos muestra el Serial Plotter (ilustración 13)

```
Conectarse al dron y subir throttle al maximo180      0      0      0
Conectamos la Bateria
El motor emitira unos pitidos, Bajamos el Throttle al minimo
Cuando escuchemos varios pitidos, los motores estaran configurados
Ya podremos manejar los motores
0      0      0      0
5      0      0      0
10     0      0      0
15     0      0      0
20     0      0      0
25     0      0      0
30     0      0      0
35     0      0      0
```

Ilustración 13: Salida Test Calibración ESC y motores.

✗ **Advertencia:** Durante las calibraciones y pruebas de software realizarlas sin hélices. No poner las hélices hasta no tener todo bien comprobado.

- **Hélices 1045:** Hélices de material ABS más fibra de carbono con cojines propulsor de 2mm/3mm/4mm/5mm/6mm/7.8mm, con un diámetro de 25,4 cm para motores de 800Kv hasta 1100Kv.

Si las hélices no están bien calibradas, las vibraciones producidas por el movimiento irregular de estas pueden convertirse en nuestro peor enemigo a la hora de hacer volar el Dron.

Son elementos que giran a gran velocidad y que sustentan todo el peso del Dron, por lo que pequeñas irregularidades en el movimiento de los

mismos pueden acarrear grandes vibraciones que pueden afectar a la estimación de inclinación y hacer que el Dron nunca llegue a ser estable.

Para comprobar y calibrar las Hélices, se ejecuta el código situado en Test/TestCalibrarHelices.cpp, y se sigue los pasos indicado por el Serial Plotter (ilustración 14)

```
Conectarse al dron y bajar throttle al minimo
Ya puede iniciar el archivo de matlab, para mostrar las graficas
-0.00  -0.01  9.82
-0.01  -0.01  9.82
-0.01  -0.01  9.82
-0.01  -0.01  9.83
-0.01  -0.01  9.83
-0.01  -0.02  9.83
-0.00  -0.02  9.82
```

Ilustración 14: Salida Test de calibración de hélices

Para una correcta calibración, las grafica que muestre en Matlab, deben de oscilar entre [-3,3] en los ejes X e Y (ilustración 15) y en el eje Z entre [-15,15] (ilustración 16)

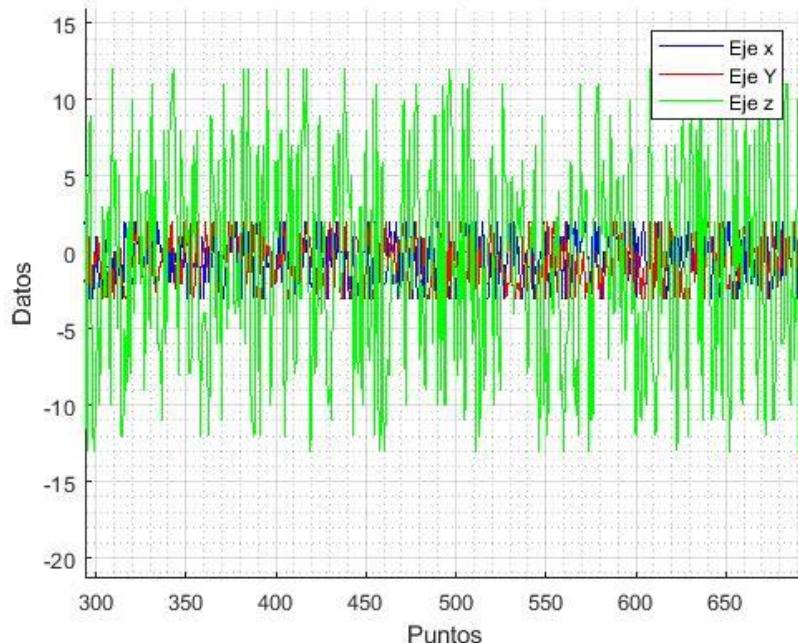


Ilustración 16: Grafica donde se representa la vibración de los ejes X,Y,Z , los valores del eje vertical (Datos) se representa (m/s^2) y el eje horizontal es el número de datos

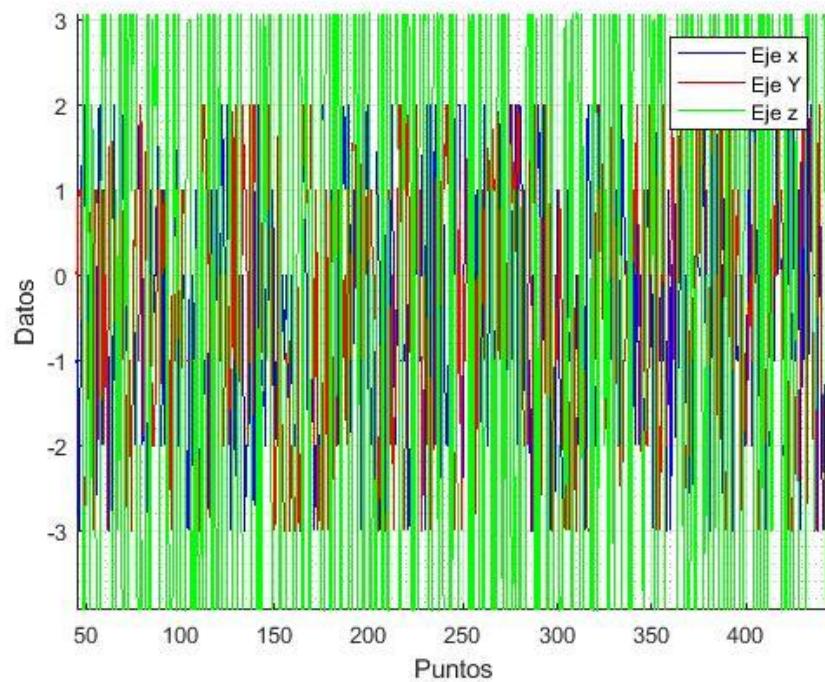


Ilustración 15: Grafica anterior con zoom en los datos Eje X,Y

- **Placa de Distribución de potencia:** Utilizamos esta placa para soldar el divisor resistivo (ilustración 8), conectar el positivo de 12v a las tiras de Leds, el conector Jack para alimentar la placa de Arduino Uno y la conexión de los cuatro ESC de forma segura (ilustración 17) a la batería.

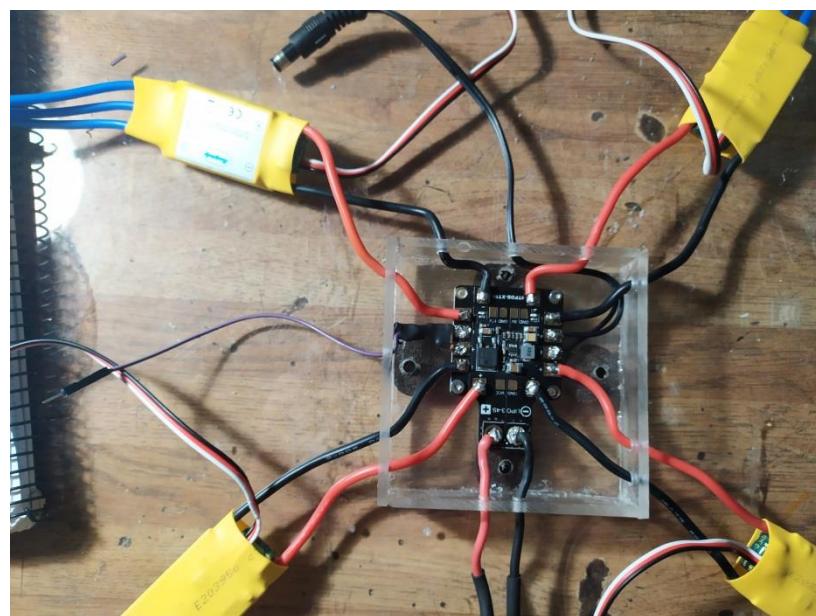


Ilustración 17: Conexión distribuidor de potencia

- **Arduino Uno:** Se trata de una plataforma de código abierto basado en C y Processing de software gratuito. Incluye un procesador ATmega328P en el que se carga la programación deseada. Para el montaje sujetamos la placa a la segunda capa metacrilato (Ilustración 18)

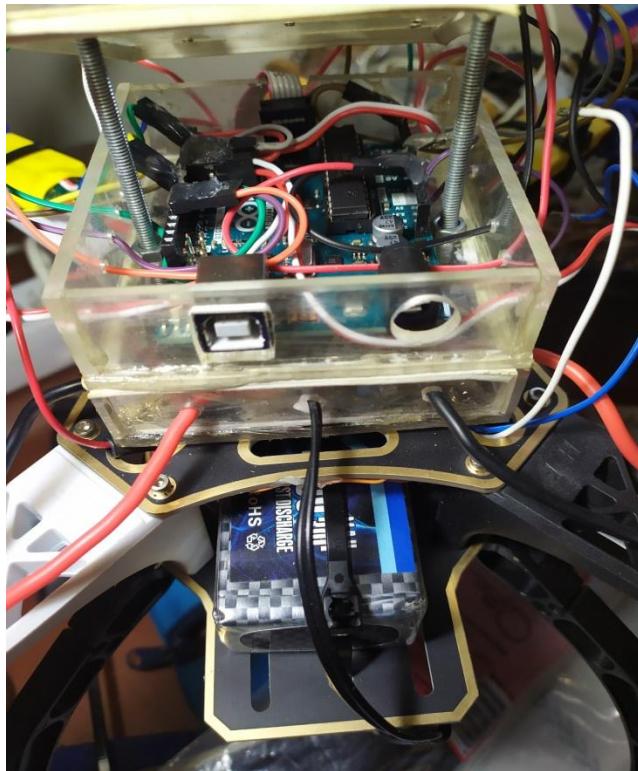


Ilustración 18: Instalación Arduino

- **MPU6050:** IMU o Inertial Measurement Module es el sensor más importante del Dron y sin el cual es imposible mantener el Dron en el aire de forma autónoma. Este sensor proporciona lecturas de velocidad de rotación (giroscopio) y aceleración en los tres ejes de movimiento (acelerómetro). Procesando adecuadamente estas señales podremos obtener las dos variables imprescindibles para volar el Dron: la **velocidad de rotación** de los tres ejes se mide en grados por segundo ($^{\circ}/s$) y la **inclinación del Dron** ($^{\circ}$). La comunicación del módulo es por I2C, esto le permite trabajar con la mayoría de microcontroladores. Los pines SCL y SDA tienen una resistencia pull-up en placa para una conexión directa al Arduino.

Los acelerómetros miden el efecto de la fuerza de gravedad sobre ellos mismos, al disponerse 3 acelerómetros formando 3 ejes ortogonales (ilustración 19), el aumento de percepción de gravedad en uno de los ejes indica que este se está poniendo en vertical, incidiendo toda la fuerza de gravedad sobre él.

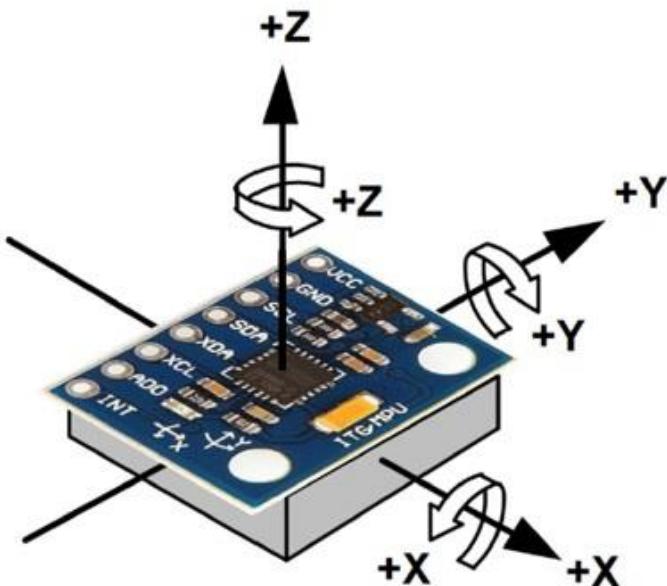


Ilustración 19: Ejes MPU6050

Esta presión produce una alteración en la alineación de los electrones y protones en el material lo que provoca una acumulación de cargas opuestas en ambas superficies del material percibiéndose así una variación en la carga eléctrica que resulta proporcional al esfuerzo experimentado por el material, el cual es proporcional a la aceleración de la gravedad por la segunda ley de Newton $F = m * a$. Esta variación correctamente tratada ayuda a tomar la medida de la cantidad de inclinación que tiene un eje, con una serie de sencillos cálculos trigonométricos se determina el giro del eje.

A continuación se va a explicar el procedimiento de cálculo para el acelerómetro. **La aceleración** medida por el sensor dependerá de la sensibilidad escogida, en este caso de $\pm 8g$. Según la tabla del datasheet del sensor, cada 4096LSB que leamos en raw (en las variables ax , ay , az),

equivale a 1g de aceleración (ilustración 20), o lo que es lo mismo, a 9.8m/s^2 . Por ello, dividiendo los valores raw de aceleración entre 4096 obtendremos la aceleración.

AFS_SEL	Full Scale Range	LSB Sensitivity
0	$\pm 2g$	16384 LSB/g
1	$\pm 4g$	8192 LSB/g
2	$\pm 8g$	4096 LSB/g
3	$\pm 16g$	2048 LSB/g

Ilustración 20: Datasheet Sensibilidad

Por lo tanto, y con esta configuración, se debe leer valores cercanos a 0 en los ejes ‘ax’ y ‘ay’ y valores cercanos 4096 para el eje ‘az’. Aunque todavía se tendrá que calibrar para obtener los valores del offset.

El giroscopio consiste en un aparato que mide la orientación de un objeto en el espacio, estos se fundamentan en el efecto coriolis para tomar sus medidas.

El efecto coriolis está presente en todos los cuerpos de rotación, incluida la propia tierra. Este efecto consiste en la distinta percepción del movimiento al observar desde un sistema de referencia rotatorio (no inercial) a un objeto que se encuentra fuera de este sistema de referencia.

La velocidad de rotación dependerá de la sensibilidad escogida, en este caso una sensibilidad de $\pm 500\text{dps}$, tiene un valor de 65.5LSB de lectura raw (en las variables gx, gy, gz) equivalen a $1^\circ/\text{s}$ de velocidad de rotación (ilustración 21)

6.1 Gyroscope Specifications
 VDD = 2.375V-3.46V, VLOGIC (MPU-6050 only) = 1.8V±5% or VDD, T_A = 25°C

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0 FS_SEL=1 FS_SEL=2 FS_SEL=3		±250 ±500 ±1000 ±2000		°/s °/s °/s °/s	
Gyroscope ADC Word Length			16		bits	
Sensitivity Scale Factor	FS_SEL=0 FS_SEL=1		131 65.5		LSB/(°/s) LSB/(°/s)	
Sensitivity Scale Factor Tolerance			32.8		LSB/(%)	
Sensitivity Scale Factor Variation Over Temperature	FS_SEL=2 FS_SEL=3		16.4		LSB/(%)	
Nonlinearity	25°C	-3	±2	+3	%	
Cross-Axis Sensitivity	Best fit straight line; 25°C		0.2 ±2		%	

Ilustración 21: Datasheet Giroscopio

Por tanto, la velocidad de rotación de cada eje (gx, gy, gz) se obtiene.

$$\text{velocidad rotación } (\text{°}/\text{s}) = \frac{\text{Lectura valor del eje}}{\text{LSB}}$$

Para calcular el **ángulo de inclinación (°)**, debemos saber la velocidad de rotación y el tiempo (segundos) que ha estado rotando a esta velocidad, podemos fácilmente calcular los grados que ha rotado en un determinado eje. Por ejemplo, si el Dron gira a una velocidad de 10°/s durante 1 segundo, el Dron se habrá inclinado 10°.

$$\text{Angulo inclinación} = \text{velocidad rotación} * \text{tiempo ejecución (s)}$$

Esta medida no es exacta incluso cuando no se mueve, el ángulo varía, esto se debe a un error producto de la mala medición del tiempo o del ruido en la lectura del MPU, el error por más pequeño que sea, se va acumulando en cada iteración y creciendo, este error es conocido como **Drift**. Para disminuir el Drift existen varios métodos, la mayoría aplica filtros para eliminar el ruido de las lecturas del sensor. También se pueden usar otros sensores como magnetómetros o acelerómetros y con los ángulos calculados con estos mejorar el cálculo del giroscopio.

Uno de los mejores filtros para eliminar el Drift es el filtro Kalman, pero se necesita una buena capacidad de procesamiento computacional, haciéndolo difícil implementar en Arduino.

Con lo cual se utiliza el filtro de complementario es en realidad una unión de dos filtros diferentes: un High-pass Filter para el giroscopio y un Low-pass Filter para el Acelerómetro. El primero deja pasar únicamente los valores por encima de un cierto límite, al contrario que el Low-pass filter, que sólo permite a los que están por debajo.

La fórmula resultante de combinar los dos filtros es:

$$\text{Angulo} = 0.98 * (\text{AnguloGiroscopio}) + 0.02 * \text{AnguloAcelerometro}$$

Esta fórmula es la misma para el eje X, Y. Si se desea, se puede cambiar los valores de 0.98 y 0.02 de los filtros, pero ambos tienen que sumar 1, ejemplo 0.95 y 0.05.

Por ultimo para **calibrar MPU6050** y obtener **valor offset**, ya que el sensor MPU6050 probablemente no se encuentre 100% en una posición horizontal, ya que puede estar desnivelado agregando un error en cada componente.

Para solucionar este problema, se realiza una estimación del offset cogiendo 3000 muestras tanto del giroscopio como del acelerómetro y se calcula el valor medio, obteniendo el offset del sensor.

A la hora del montaje del sensor se tendrá que centrar lo máximo posible en la estructura Frame, y orientar el eje y en la dirección que se considere recta al eje y (ilustración 22).

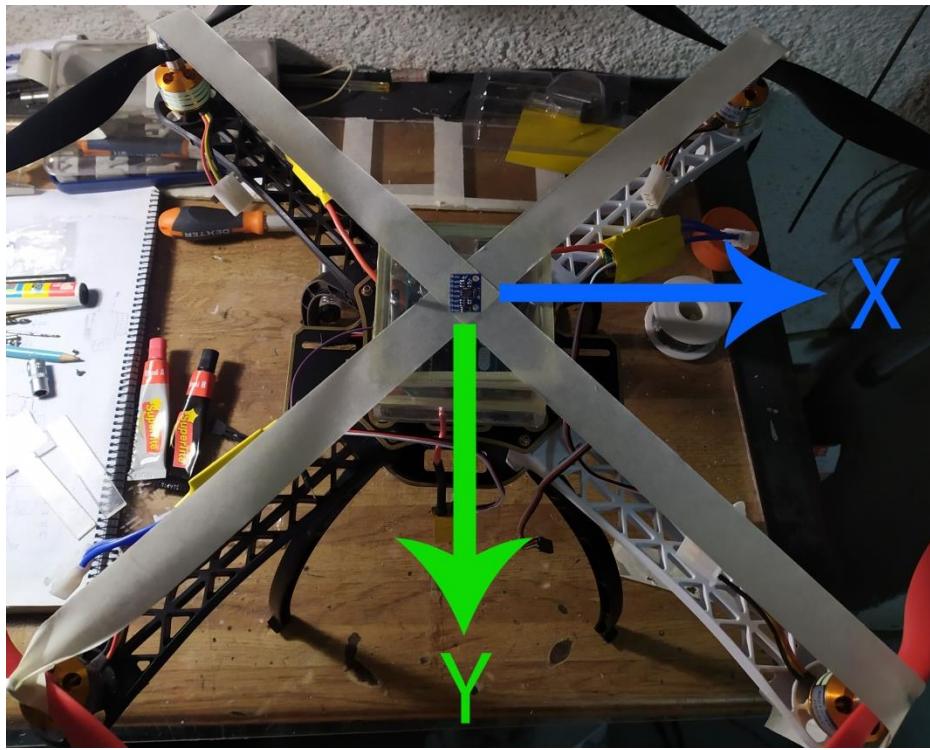


Ilustración 22: Orientación e instalación MPU6050

Para comprobar el funcionamiento y la calibración del MPU6050, se ejecuta el código situado en Test/TestMPU6050.cpp, donde se mostrara por el Serial Plotter el ángulo con el filtro complementario aplicado.

Mover el Dron para confirmar que los datos varían y son datos coherentes (ilustración 23).

```
Importante, no mover calculando el offset del giroscopio y acelerometro
Ya se puede mover el dron
0.08  0.04
0.08  0.04
-0.97 -3.84
-0.89 -4.13
-0.82 -4.43
-0.74 -4.73
-0.68 -5.04
-0.62 -5.35
-0.58 -5.67
-0.54 -6.00
```

Ilustración 23; Test MPU6050, valores de los ángulos

⚠ Advertencia: Asegurar que el sensor están bien sujetos, lo más centrado posible y en una superficie con las menos vibraciones posibles.

- **Mosfet IRFZ44N:** Es un transistor de tecnología MOSFET (Metal–Oxide–Semiconductor – Field Effect Transistor) y de alta potencia que posee destacadas características que lo hacen ideal para aplicaciones de conmutación y en la modulación por ancho de pulso (PWM). Para el montaje se utiliza el hueco disponible en la penúltima caja de metacrilato, donde se ha instalado el MPU6050 (ilustración 24).

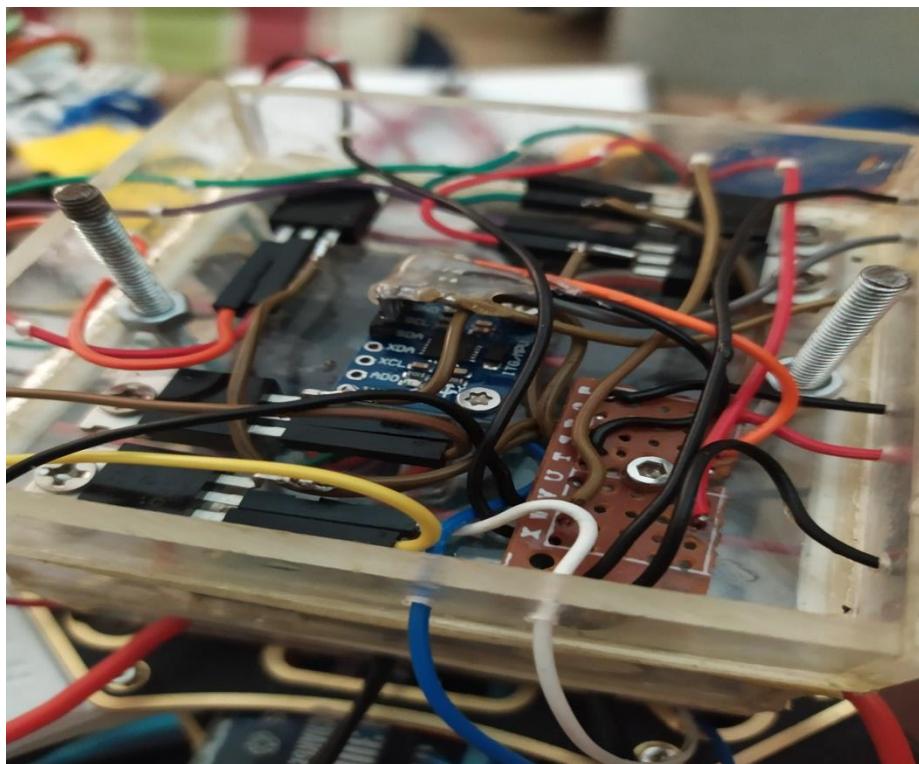


Ilustración 24: Instalación MPU6050, Mosfet y pcb con gnd y 5v de los pines de Arduino

- **Pixy2:** Es un Sensor de Imagen CMUcam5 de Charmed Labs. Pixy2 puede aprender a detectar objetos que usted le enseñe, simplemente presionando un botón. Además, Pixy2 tiene nuevos algoritmos que detectan y rastrean líneas para que sigan líneas. Los nuevos algoritmos también pueden detectar intersecciones y "señales de tráfico".

Para comprobar el funcionamiento se realiza los siguientes pasos:

En primer lugar conectamos la cámara Pixy con su cable micro Usb, y a continuación abrimos el programa PixyMon (ilustración 25)

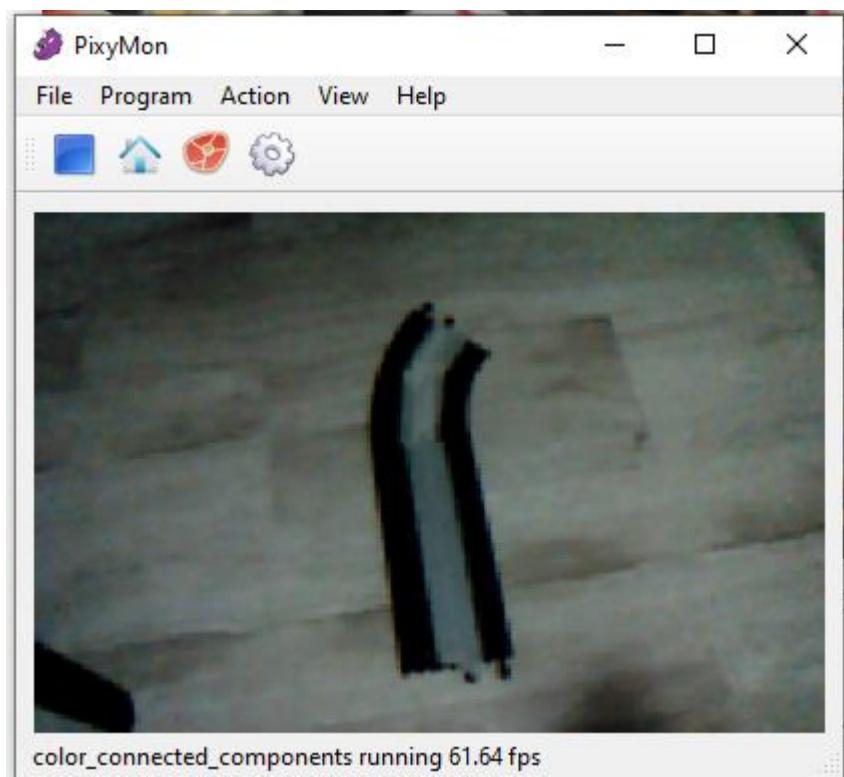


Ilustración 25: Configuración para detectar colores de la cámara Pixy

Como se necesita comprobar que la cámara detecta línea, para realizar el seguimiento, marcamos la opción line_tracking (ilustración 26).

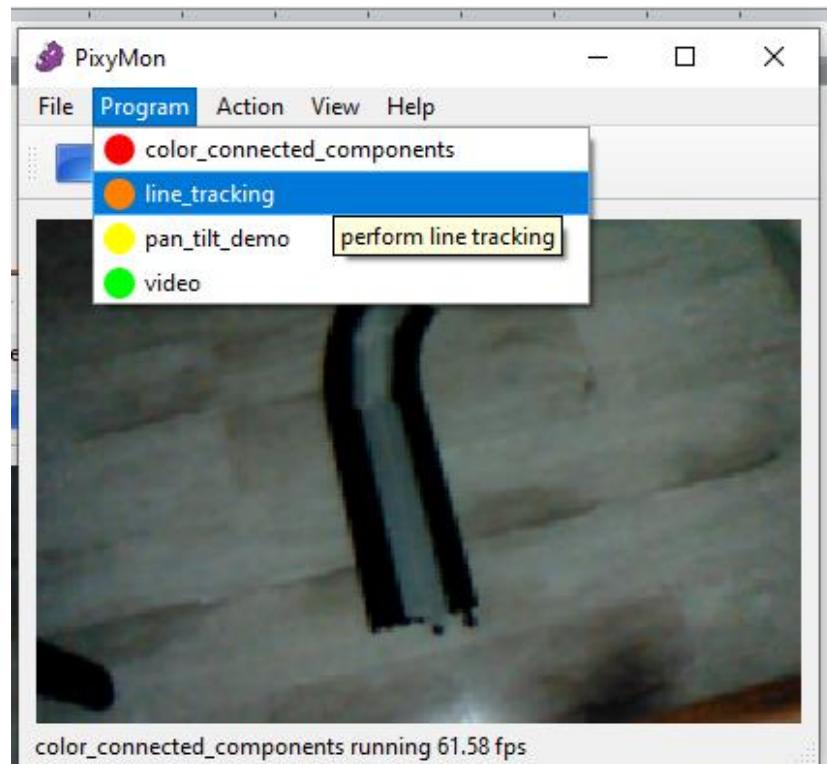


Ilustración 26: Selección del tipo de programa para Pixy

Al cambiar el modo a line_tracking, nos debe identificar la línea y nos marcará la ruta que seguiría la cámara (ilustración 27).

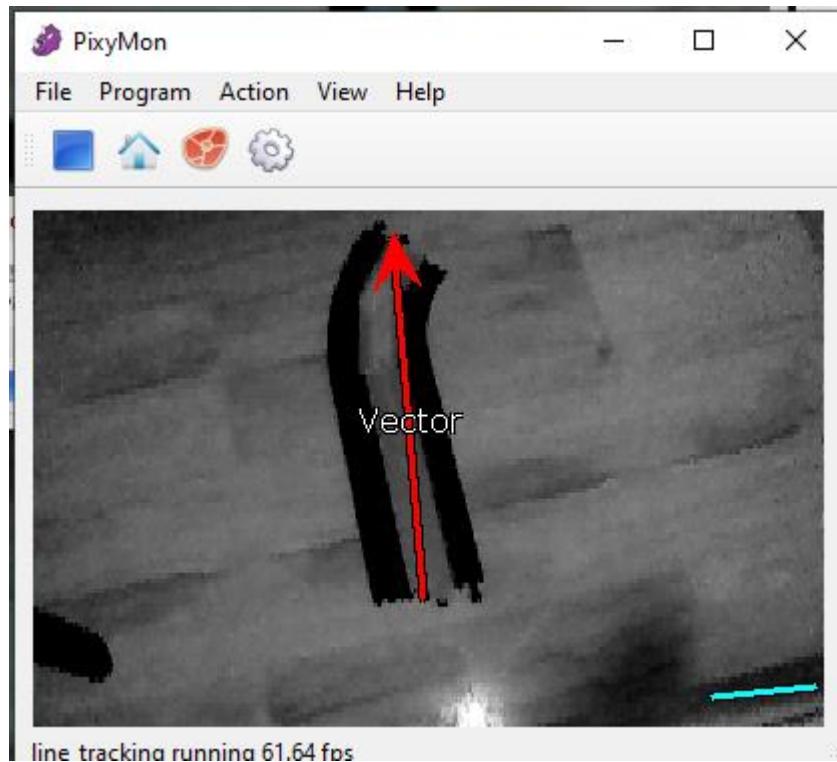


Ilustración 27: Comprobación del seguimiento de líneas y curvas

- **Cargador Batería Imax B6:** Es un cargador universal que es capaz de trabajar con múltiples salidas de alimentación, y cargar todo tipo de baterías, como las de plomo o Pb, las de Ni-Cd (de hasta 15 celdas, las de Ni-MH de hasta 15 celdas, las de Li-Po de hasta 6 celdas, las de Li-Fe de hasta 6 celdas, etc. Cuenta con una potencia máxima de 80w, más que suficiente para alimentar uno o varios proyectos a la vez. Permite también carga rápida, gracias a un microporcesador de alto rendimiento y un software integrado que se adapta de forma específica a cada operación. También dispone de una pantalla LCD, con 2 líneas y 16 caracteres para poder elegir el tipo de batería, programas de protección, tiempo de carga, modo rápido, y todas las opciones de configuración con sus botones. También podrás cargar y descargar baterías de Li-Po de forma cíclica, cada

celda de forma individual, lo que permite revivir celdas y activar el 100% de la batería.

Modo de Uso: Para cargar la batería Lipo empleada en el proyecto se siguen los siguientes pasos.

*En primer lugar vamos a los ajustes (ilustración 28), pulsamos Start y buscamos la opción (ilustración 29), pulsamos start y configuramos la capacidad de corte, con la capacidad de la batería (esto es una forma segura para no dañar la batería)



Ilustración 28: Configuración cargador

Ilustración 29: capacidad de corte de carga

*Luego se conecta la batería como muestra la (ilustración 30), y buscamos la opción indicada en la (ilustración 30).



Ilustración 30: Conexión y programa para cargar la batería

*Pulsamos Start, y dentro de este menú se puede realizar una carga completa (ilustración 31), o una carga segura para guardar la batería durante meses (ilustración 32), o una carga rápida (ilustración 33).



Ilustración 31: Programa de carga

Ilustración 33: Programa de carga rápida

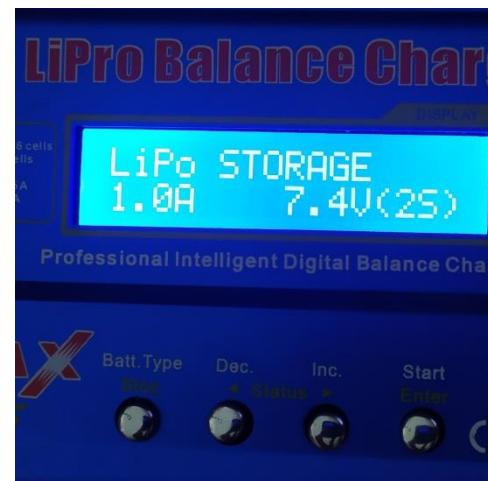


Ilustración 32: Programa de almacenaje

*Una vez seleccionado el programa, se pulsa Start, y se configura el voltaje correspondiente de la batería a cargar y se deja pulsado el botón Start, para iniciar el programa (Ilustración 31).

*Sonara unos pitidos y comenzara a cargar (ilustración 34), si pulsamos Status >, nos indicara el voltaje de cada celda (ilustración 35)



Ilustración 34: Programa de carga iniciado

Ilustración 35: Voltaje de cada celda

II.III. Parte Software

- **Conceptos generales:** Vamos a ir explicando cada elemento de la Estructura Software (ilustración 36)

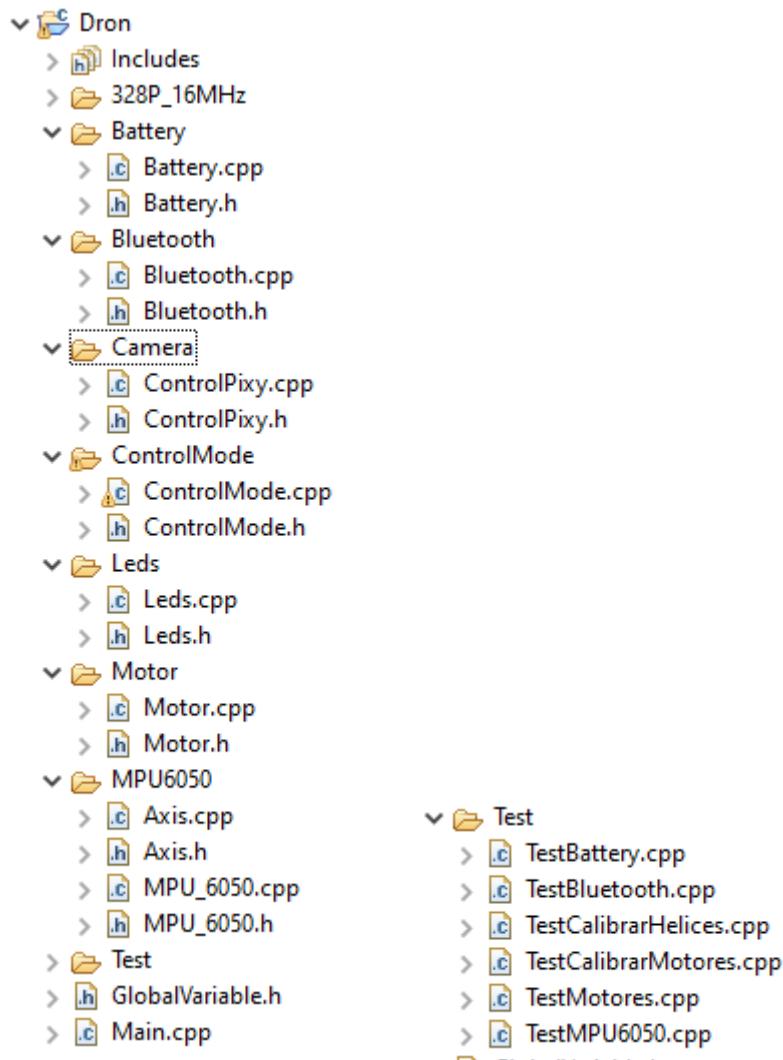


Ilustración 36: Estructura Software

Para comprender el código, vamos a comenzar con las variables globales, que tenemos definida en GlobalVariable.h (ilustración 37,38,39 y 40)

```
//=====================================================================Global Variables=====
#define usCiclo 6000 //ciclo del programa
#define baudrates 115200 //frecuencia del programa
#define pi 3.1416

//=====================================================================Leds=====
#define ledWhite 9
#define ledBlue 10
#define ledRedFront 11
#define ledGreenFront 12
#define ledRedBack 13
```

Ilustración 37: Primera parte del archivo GlobalVariable.h

usCiclo, se refiere a la cantidad en milisegundos que dura el ciclo del programa principal

baudrates, se refiere a la velocidad en la que se ejecuta los elementos, por ejemplo el Serial Plotter o el Bluetooth

Las variables referentes a los Leds, son los pines digitales asignados en Arduino, por ejemplo el led blanco está conectado al pin 9 digital de Arduino

```
//=====Battery=====
#define pinBattery A0
#define offsetVoltage 5.00
#define voltageMinBattery 10.5

//=====MPU6050=====
//-----PIN-----
#define sdaMPU A4
#define sclMPU A5
//-----

#define sensibilityAcc 0x10
#define LSBAccelerometer 4096
#define sensibilityGyr 0x08
#define LSBGyroscope 65.5
#define numberSampleMPU 3000
```

Ilustración 38: Segunda parte del archivo GlobalVariable.h

Continuamos con las variables referentes a la Batería.

pinBattery se refiere al pin analógico donde leeremos el valor de la batería.

offsetVoltage es la variable de conversión para calcular el voltaje real de la batería.

voltageMinBattery es el límite que le imponemos, ya que en caso de que sea menor o igual a este voltaje, esto nos indicaría que la batería está casi descargada.

A continuación vamos a explicar las variables referentes al MPU6050

sdaMPU y **sclMPU** son los pines analógicos de donde leeremos el valor del giroscopio y acelerómetro.

sensibilityAcc y **LSBAccelerometre** es la sensibilidad y el valor LSB que se ha definido para el acelerómetro, este valor lo podemos ver en la tabla (ilustración 20)

sensibilityGyr y LSBGyroscope es la sensibilidad y el valor LSB que se ha definido para el giroscopio, este valor lo podemos ver en la tabla (ilustración 21)

numberSampleMPU se refiere a la cantidad de muestra que vamos a utilizar para poder calcular el offset correspondiente al MPU.

```
//=====Bluetooth=====
//----- PIN -----
#define rxBluetooth 3
#define txBluetooth 2

//-----Control-----
#define degreeMinThrottle 0
#define degreeMaxThrottle 180
#define degreeMin -30
#define degreeMax 30
#define incrementThrottle 5
#define incrementDegree 1

//-----Value Accion-----
#define throttleUp 'A'
#define throttleDown 'B'
#define pitchUp 'P'
#define pitchDown 'H'
#define rollUp 'R'
#define rollDown 'L'
#define yawUp 'W'
#define yawDown 'Y'

#define changeMaxThrottle 'q'
#define changeMinThrottle 'a'
#define valueModeUp 'z'
#define valueModeDown 'x'
#define valueModeAcrobatic 'c'
#define valueModeStable 'v'
#define valueModeAutomatic 'p'
```

Ilustración 39: Tercera parte del archivo GlobalVariable.h

Continuamos el archivo GlobalVariable, con los valores definidos para el Bluetooth.

rxBluetooth y **txBluetooth** se refiere a los pines digitales para la comunicación de Arduino y el Bluetooth.

degreeMinThrottle y **degreeMaxThrottle** es el intervalo que hemos definido como mínimo y máximo para el valor del Throttle.

degreeMin y **degreeMax** es el intervalo que hemos definido para los ejes Pitch, Roll y Yaw.

incrementThrottle es el incremento o decremento que aplicamos al Throttle cada vez que recibimos una orden, de aumentar o reducir el Throttle desde el Bluetooth.

incrementDegree es el incremento o decremento que aplicamos al Pitch, Roll o Yaw cada vez que recibimos una orden, de aumentar o reducir alguno de estos ejes desde el Bluetooth.

El apartado siguiente son todos los elementos contemplados que podemos recibir a través del Bluetooth.

ThrottleUp y **ThrottleDown** es el valor que recibimos a través del Bluetooth para subir o bajar el valor del Throttle.

PitchUp y **PitchDown** es el valor que recibimos a través del Bluetooth para subir o bajar el valor del Pitch.

RollUp y **RollDown** es el valor que recibimos a través del Bluetooth para subir o bajar el valor del Roll.

YawUp y **YawDown** es el valor que recibimos a través del Bluetooth para subir o bajar el valor del Yaw.

changeMaxThrottle se refiere al valor que recibimos a través del Bluetooth para modificar el Throttle y subirlo a su valor más alto.

changeMinThrottle se refiere al valor que recibimos a través del Bluetooth para modificar el Throttle y bajarlo a su valor más bajo.

valueModeDown se refiere al valor que recibimos a través del Bluetooth para cambiar al modo Aterrizaje.

valueModeUp se refiere al valor que recibimos a través del Bluetooth para cambiar al modo Despegue.

valueModeAcrobatic se refiere al valor que recibimos a través del Bluetooth para cambiar al modo Acrobático.

valueModeStable se refiere al valor que recibimos a través del Bluetooth para cambiar al modo Estable.

valueModeAutomatic se refiere al valor que recibimos a través del Bluetooth para cambiar al modo Automático.

```
//=====Motores=====
#define pinM1 4
#define pinM2 5
#define pinM3 6
#define pinM4 7
#define pulseMinMotor 1000      // pulsos de 1000us a 2000 us  980
#define pulseMaxMotor 2100

#define degreeMaxUp 50

//=====PID=====
#define limitVelocity 380
#define limitAngle 130

#define KpPitchAngle 2.2
#define KiPitchAngle 0.06
#define KdPitchAngle 15

#define KpPitchW 1.9
#define KiPitchW 0.07
#define KdPitchW 12

#define KpRollAngle 2.2
#define KiRollAngle 0.06
#define KdRollAngle 15

#define KpRollW 1.9
#define KiRollW 0.07
#define KdRollW 12

#define KpYawW 1.5
#define KiYawW 0.05
#define KdYawW 0

#endif /* GLOBALVARIABLE_H_ */
```

Ilustración 40: Última parte del archivo GlobalVariable.h

Continuamos con las variables referentes a los **motores**.

pinM1, pinM2, pinM3, pinM4 se refiere a los pines de salida digital referentes a Arduino, para los motores 1, 2, 3 y 4.

pulseMinMotor y **pulseMaxMotor** es el valor que definimos como amplitud en milisegundo para los motores, a la hora de su funcionamiento.

degreeMaxUp se refiere al ángulo máximo funcionamiento del Throttle en el modo Despegue.

Ahora continuamos con los valores definidos para el controlador PID
limitVelocity es la amplitud que puede generar el controlador PID referente a la velocidad

limitAngle es la amplitud que puede generar el controlador PID referente al ángulo.

KpPitchAngle, **KiPitchAngle** y **KdPitchAngle** es el valor de las constantes proporcionales, integrales y derivativas, definidas para el PID según el ángulo para el eje Pitch.

KpPitchW, **KiPitchW** y **KdPitchW** es el valor de las constantes proporcionales, integrales y derivativas, definidas para el PID según la velocidad para el eje Pitch.

KpRollAngle, **KiRollAngle** y **KdRollAngle** es el valor de las constantes proporcionales, integrales y derivativas, definidas para el PID según el ángulo para el eje Roll.

KpRollW, **KiRollW** y **KdRollW** es el valor de las constantes proporcionales, integrales y derivativas, definidas para el PID según la velocidad para el eje Roll.

KpYawW, **KiYawW** y **KdYawW** es el valor de las constantes proporcionales, integrales y derivativas, definidas para el PID según la velocidad para el eje Yaw.

Una vez aclarada las variables globales, vamos a comenzar con la clase Main.cpp, ya que es el programa principal del programa.

En primer lugar se define todos elementos que compone nuestro Dron (ilustración 41).

```

9  /*
10 */
11
12 #include "Motor/Motor.h"
13 #include "Battery/Battery.h"
14 #include "ControlMode/ControlMode.h"
15 #include "Camera/ControlPixy.h"
16
17 Battery bateria(pinBattery);
18
19 Bluetooth bluetooth(rxBluetooth,txBluetooth);
20
21 Motor motor1(pinM1);
22 Motor motor2(pinM2);
23 Motor motor3(pinM3);
24 Motor motor4(pinM4);
25
26 Leds leds(ledWhite, ledBlue, ledRedFront, ledGreenFront, ledRedBack);
27
28 MPU6050 mpu6050(0x68, sdaMPU, sclMPU, numberSampleMPU);
29
30 ControlPixy pixy;
31
32 ControlMode controlMode;
33

```

Ilustración 41: Declaración de objetos dentro de Main.cpp

Vamos a ir analizando cada elemento declarado en la (ilustración 41), empezamos con la estructura referente a la batería, situado en Battery/Battery.h (ilustración 42) y Battery/Battery.cpp (ilustración 43)

```

#include "../GlobalVariable.h"

struct Battery{
    private:
        uint8_t pin;
        float voltage;
    public:
        Battery(uint8_t pin);
        int readSensor();
        void updateVoltage(int voltage);
        float getVoltage();
        bool isLowBattery();
};

```

Ilustración 42: Bateria.h, estructura del componente Batería

Esta clase contiene dos atributos, el pin referente a Arduino y el voltaje actual de la batería.

El constructor para el componente de la Batería inicializa el pin que recibimos por argumento, e inicializa el valor del voltaje.

El método readSensor() leemos el pin analógico de Arduino referente al nivel de la batería.

updateVoltage(int voltage) calculamos el valor real de la batería y modificamos el valor del voltaje.

isLowBattery() nos devuelve un booleano indicando si el nivel de la batería es bajo o no. Para ello leemos el valor del pin analógico y comprobamos si el valor es mínimo o no.

```
#include "Battery.h"

Battery::Battery(uint8_t pin){
    this->pin = pin;
    this->voltage = 0.0;
}

int Battery::readSensor(){
    return analogRead(pin);
}

void Battery::updateVoltage(int voltage){
    this->voltage = 2.5 * (voltage * offsetVoltage /1023);
}

float Battery::getVoltage(){
    return voltage;
}

bool Battery::isLowBattery(){
    updateVoltage(readSensor());
    return voltage < voltageMinBattery;
}
```

Ilustración 43: contenido del archivo Battery.cpp

Una vez definido el componente Batería nos vamos a la siguiente declaración de Main.cpp (ilustración 41), referente al componente Bluetooth. Este componente se define en los archivos Bluetooth.h (ilustración 44) y Bluetooth.cpp (ilustración 45,46,47 y 48)

```

struct Bluetooth{
    private:
        SoftwareSerial* BT;
        int degreeThrottle,degreePitch,degreeRoll,degreeYaw;

    public:
        Bluetooth(uint8_t rx , uint8_t tx);
        void initialize();
        void configure(String name,String pin);
        void updatePulse();
        String waitForResponse();

        void changeThrottle(String value);
        void changeAngle(float angle,char valueUp,char valueDown,String value);
        void connectionMotor(String value);

        void modifyThrottle(int value);
        void modifyPitch(int value);
        void modifyRoll(int value);
        void modifyYaw(int value);

        int getThrottle();
        int getPitch();
        int getRoll();
        int getYaw();

        void showData();
};


```

Ilustración 44: Contenido Battery.h

En la estructura Bluetooth observamos que tiene 5 atributos, BT referente al SoftwareSerial que servirá para configurar y comunicarnos con Arduino a través de RX y TX

Y los demás son los valores en ángulo de cada eje que controlamos con el Bluetooth.

El constructor del Bluetooth, crea la comunicación asignando los pines rx y tx, y asigna a cada valor de su eje, el valor intermedio para cada uno de ello.

Por ejemplo para el Throttle como el mínimo es 0 y el máximo 180, lo iniciaremos con un valor de 60, se inicializa con este valor ya que si no nos hemos conectado con el Smartphone al Arduino no puede avanzar el programa Main, ya que tiene que esperar a que nos conectemos.

Para los demás ejes el valor inicial será 0°

El método **initialize()**, inicializa la comunicación con una velocidad predefinida con GlobalVariable.h

```
#include "Bluetooth.h"

> Bluetooth::Bluetooth(uint8_t rx, uint8_t tx)
{
    this->BT = new SoftwareSerial(rx,tx);
    this->degreeThrottle = (degreeMinThrottle + degreeMaxThrottle)/2;
    this->degreePitch = (degreeMax + degreeMin)/2;
    this->degreeRoll = (degreeMax + degreeMin)/2;
    this->degreeYaw = (degreeMax + degreeMin)/2;
}

> void Bluetooth::initialize(){
    this->BT->begin(baudrates);
}
A
```

Ilustración 45: Primera parte Baterry.cpp

El método **configure()**, solo se utiliza la primera vez, si no tenemos configurado, ya que este realiza la configuración del dispositivo HC-06 con sus comando especiales AT.

El método **updatePulse()** esperamos a recibir algún valor desde el Bluetooth y según el valor realizamos la acción correspondiente, si el valor que hemos recibido no es nulo.

```
void Bluetooth::configure(String name, String pin){
    // Should respond with OK
    this->BT->print("AT");
    waitForResponse();
    // Should respond with its version
    this->BT->print("AT+VERSION");
    waitForResponse();
    // Set the name to PRACT_NAME
    this->BT->print("AT+NAME"+name);
    waitForResponse();
    // Set response to ok pin
    this->BT->print("AT+PIN"+pin);
    waitForResponse();
    // Set respond with okbaudrate
    this->BT->print("AT+BAUD8");
    waitForResponse();
}

void Bluetooth::updatePulse(){
    String value = waitForResponse();
    if(!value.equals("")){
        changeThrottle(value);
        changeAngle(degreePitch, pitchUp, pitchDown, value);
        changeAngle(degreeRoll, rollUp, rollDown, value);
        changeAngle(degreeYaw, yawUp, yawDown, value);
        connectionMotor(value);

        //showData();
    }
}
```

Ilustración 46: Segunda parte Bluetooth.cpp

El método **changeThrottle()** leemos el valor que hemos recibido por el Bluetooth y si contiene alguna opción de subida o bajada de Throttle, se realiza la acción, siempre y cuando no sea superior al máximo ángulo del Throttle (si es para subida) o viceversa si es de bajada.

El método **changeAngle()** es igual que el método anterior, solo que en este caso el incremento y decremento de los ejes Pitch, Roll o Yaw varían en su constante de incremento y su amplitud de ángulo.

```
void Bluetooth::changeThrottle(String value){  
    if(value.indexOf(throttleUp) != -1){  
        if(degreeThrottle < degreeMaxThrottle){  
            degreeThrottle = degreeThrottle + incrementThrottle;  
        }  
    }else if(value.indexOf(throttleDown)!=-1){  
        if(degreeThrottle > degreeMinThrottle){  
            degreeThrottle = degreeThrottle - incrementThrottle;  
        }  
    }  
}  
  
void Bluetooth::changeAngle(float angle,char valueUp, char valueDown,String value){  
    if(value.indexOf(valueUp) != -1){  
        if(angle < degreeMax){  
            angle = angle + incrementDegree;  
        }  
    }else if(value.indexOf(valueDown) != -1){  
        if(angle > degreeMin){  
            angle = angle - incrementDegree;  
        }  
    }  
}
```

Ilustración 47: Tercera parte Bluetooth.cpp

El metodo **connectionMotor()** aumenta o disminuye el Throttle al maximo o al minimo según el valor que recibimos por el Bluetooth.

El metodo **waitForResponse()** establecemos una espera hasta recibir algún parametro a traves del Bluetooth

El metodo **modifyThrottle()**, y los demás metodo de modify definidos en Bluetooth.h realiza la misma funcionalidad, modifica el valor de los ejes correspondiente al objeto Bluetooth con el valor que recibimos por el argumento.

```

void Bluetooth::connectionMotor(String value){
    if(value.indexOf(changeMinThrottle) != -1){
        degreeThrottle = degreeMinThrottle;
    }else if(value.indexOf(changeMaxThrottle) != -1){
        degreeThrottle = degreeMaxThrottle;
    }
}

String Bluetooth::waitForResponse(){
    String value = "";
    while (this->BT->available()) {
        value = this->BT->readString();
    }
    return value;
}

void Bluetooth::modifyThrottle(int value){
    this->degreeThrottle = value;
}

```

Ilustración 48: Última parte Bluetooth.cpp

Continuamos la declaración de los componentes del Main.cpp (ilustración 41), el siguiente objeto son los motores definidos en Motor/Motor.h(ilustración 49) y Motor/Motor.cpp(ilustración 50)

```

#ifndef MOTOR_H_
#define MOTOR_H_

#include "../GlobalVariable.h"
#include "Servo.h"

struct Motor{
    private:
        uint8_t pin;
        Servo servo;

    public:
        Motor(uint8_t pin);
        void initialize();
        void updateValue(int value);
        void controller(int throttle,float pidPitch,float pidRoll,float pidYaw);
};

#endif /* MOTOR_H_ */

```

Ilustración 49: Contenido archivo Motor.h

En la estructura del motor tenemos declarados dos atributos, el pin digital referente al motor, y el atributo servo, que es un objeto importado de la librería de Arduino Servo.h

El constructor Motor() inicializa el pin con el pin recibido por el argumento.

El método **initialize()** inicializa el valor del motor, con una amplitud de pulso declarado en GlobalVariable.h

El método **updateValue()** modifica el valor del motor y lo envía al pin de salida utilizando el método write de la clase Servo de Arduino

El método **controller()** recibe un nivel de Throttle y si el nivel de Throttle es bajo, ignoramos los valores recibidos por el controlador PID, en otro caso activamos el controlador PID sumando todos los pulso recibidos y nos aseguramos que en pleno vuelo no se detenga en seco los motores y que no enviamos un pulso superior a la amplitud establecida para los motores.

```
#include "Motor.h"

Motor::Motor(uint8_t pin){
    this->pin = pin;
}

void Motor::initialize(){
    servo.attach(pin, pulseMinMotor, pulseMaxMotor);
}

void Motor::updateValue(int value){
    servo.write(value);
}

void Motor::controller(int throttle, float pidPitch, float pidRoll, float pidYaw){
    if(throttle < 10){
        updateValue(throttle);
    }else{
        float pulso = throttle + pidPitch + pidRoll + pidYaw;
        if(pulso < 5){ //Evitamos que en pleno vuelo se pare por completo el motor
            pulso = 5;
        }
        if(pulso > degreeMaxThrottle){ //Evitamos meter valores superiores al motor
            pulso = degreeMaxThrottle;
        }
        updateValue(pulso);
    }
}
```

Ilustración 50: Contenido archivo Motor.cpp

Continuamos con el siguiente componentes que nos aparece el código Main.cpp (ilustración 41), el siguiente componente son los leds donde su estructura e implementación se encuentra en Leds/Leds.h (ilustración 51) y Leds/Leds.cpp (ilustración 52)

```

#ifndef LEDS_H_
#define LEDS_H_

#include "../GlobalVariable.h"

struct Leds{
    private:
        int length;
        uint8_t leds[5];

    public:
        Leds(uint8_t pinWhite,uint8_t pinBlue,uint8_t pinRedFront,uint8_t pinGreenFront,uint8_t pinRedBack);

        int size();
        void initialize();
        uint8_t getLed(int i);

        void onLedWhite();
        void onLedBlue();
        void onLedFly();
        void onLedWarning();
        void offAllLeds();
};

#endif

```

Ilustración 51: Contenido Leds.h

La estructura Leds contiene dos atributos, length que se refiere al tamaño del array leds, y leds el array que contiene el pin de cada led.

El constructor Leds() inicializada cada valor del array con el pin correspondiente para cada led.

El método **initialize()** inicializa todos los pines digitales de los led de forma output utilizando pinMode incorporado en la librería de Arduino.

El método **onLedBlue()** enciende las tiras leds correspondiente para el blanco.

El método **onLedFly()** enciende las tiras leds correspondiente para el verde en la parte delantera y el rojo para la parte trasera.

El método **onLedWarning()** enciende las tiras leds correspondiente para el rojo adelante y atrás.

El método **offAllLeds()** apagada todos los leds utilizando digitalWrite, incorporado en la librería de Arduino, con un valor de LOW.

```

Leds::Leds(uint8_t pinWhite,uint8_t pinBlue,uint8_t pinRedFront,uint8_t pinGreenFront,uint8_t pinRedBack)
{
    length = 5;
    leds[0] = pinWhite;
    leds[1] = pinBlue;
    leds[2] = pinRedFront;
    leds[3] = pinGreenFront;
    leds[4] = pinRedBack;
}

void Leds::initialize(){
    for(int i = 0 ;i<length; i++){
        pinMode(leds[i], OUTPUT);
    }
}

void Leds::onLedWhite(){
    digitalWrite(leds[0], HIGH);
}

void Leds::onLedBlue(){
    digitalWrite(leds[1], HIGH);
}

void Leds::onLedFly(){
    digitalWrite(leds[3], HIGH);
    digitalWrite(leds[4], HIGH);
}

void Leds::onLedWarning(){
    digitalWrite(leds[2], HIGH);
    digitalWrite(leds[4], HIGH);
}

void Leds::offAllLeds(){
    for(int i = 0 ; i < length; i++){
        digitalWrite(leds[i], LOW);
    }
}

```

Ilustración 52: Contenido Leds.cpp

Continuamos con las declaración de las clases indicada en el archivo Main.cpp (ilustración 41), llegamos a una de las líneas más importante y es el objeto MPU6050, su estructura e implementación se encuentra en MPU6050/MPU6050.h (ilustración 53) y MPU6050/MPU6050.cpp (ilustración 54,55,56,57 y 58)

```

#include "../GlobalVariable.h"
#include "Axis.h"
#include <Wire.h>

struct MPU6050 {
    private:
        int sampleNumber;
        float pitchAngle,rollAngle,temperature;
        long loopTimer,executionTime;
        bool init;
        uint8_t address,sda,scl;
        Axis gyroscope;
        Axis accelerometer;
        Axis offsetGyroscope;
        Axis offsetAccelerometer;

    public:
        MPU6050(uint8_t address,uint8_t sda,uint8_t scl, int sampleNumber);
        bool initialize();

        void calculateOffset(bool calculateGyroscope,bool calculateAccelerometer);

        Axis recalibrateGyroscope();

        void readSensors();
        void processAccelerometer();
        void calculateAngle();

        void transmitData(uint8_t reg,uint8_t data);
        void transmitAndWaitResponse(uint8_t reg,int numberByte);

        void updateLoopTimer();
        void updateExecutionTimer(long loopTimer);
}

```

Ilustración 53: Contenido de MPU6050.h

Esta estructura tiene diferentes atributos que se van a ir explicando.

sampleNumber es el número de muestra que hemos definido para calcular el offset.

pitchAngle, rollAngle es el ángulo que calculamos en el método calculateAngle() donde se le aplica el filtro complementario.

loopTimer y **excutionTime** es el tiempo de ejecución del bucle loop y **executionTime** es el tiempo de cada ciclo en segundos.

Init es un valor booleano que nos sirve para aplicar el filtro complementario solo una vez

Address es la dirección del IC que lo habremos obtenido anteriormente en TestMPU6050, que nos indica la dirección del sensor.

Sda y **Sc** son los pines analógicos asignado en Arduino para poder leer los valores del acelerómetro y del giroscopio, entre otros.

Gyroscope, **accelerometer** son los valores que hemos obtenido al leer los pines del sensor, estos atributos son de tipo Axis situada en MPU6050/Axis.h (ilustración 59)

```
#ifndef MPU6050_AXIS_H_
#define MPU6050_AXIS_H_

struct Axis {
public:
    float x;
    float y;
    float z;

    Axis();
};


```

Ilustración 59: Estructura Axis.h

Básicamente es una estructura donde se almacena los valores de los 3 ejes.

offsetGyroscope y **offsetAccelerometer** son los valores de los offset calculados para el giroscopio y el acelerómetro.

En el constructor de MPU6050 inicializamos todo las variables con valores neutros.

El método **initialize()** nos devolverá un booleano indicando si se ha inicializado correctamente o no el MPU6050, para continuar o no la ejecución del código. En este método se inicializa todo los valores con las sensibilidades elegidas como indica la tabla (ilustración 20 y 21)

```

MPU6050::MPU6050(uint8_t address,uint8_t sda,uint8_t scl, int sampleNumber){
    this->address = address;
    this->sda = sda;
    this->scl = scl;
    this->sampleNumber = sampleNumber;
    this->temperature = 0.0;
    this->pitchAngle = 0.0;
    this->rollAngle = 0.0;
    this->loopTimer = 0;
    this->executionTime = 0;
    this->init = false;
}

bool MPU6050::initialize(){
    bool start = true;
    Wire.begin();
    this->transmitData(0x6B,0x00);
    // PWR_MGMT_1 registro 6B hex - 00000000 para activar
    this->transmitData(0x1B,sensibilityGyr);
    // GYRO_CONFIG registro 1B hex - 00001000: 500dps sensibilidad giroscopio
    this->transmitData(0x1C,sensibilityAcc);
    // ACCEL_CONFIG registro 1C hex - 00010000: +/- 8g sensibilidad acelerometro

    //Test de prueba
    transmitAndWaitResponse(0x1B, 1);
    if (Wire.read() != 0x08) {
        start = false;
        Serial.println("MPU ERROR");
    }

    this->transmitData(0x1A, 0x05);
    // LPF (filtro paso bajos) registro 1A hex - 256Hz(0ms):0x00 - 188Hz(2ms):0x01 - 98Hz
    return start;
}

```

Ilustración 54: Primera Parte contenido MPU6050.cpp

El método calculateOffset() genera un bucle de tamaño según la muestra que hallamos declarado en el GlobalVariable.h, y calcula el valor medio del acelerómetro y giroscopio, si los modos que tenemos activos necesitan el cálculo de ellos, obteniendo así un offset muy precioso.

```

void MPU6050::calculateOffset(bool calculateGyroscope,bool calculateAccelerometer){
    Axis sumAcc;
    Axis sumGy;
    for (int i = 0; i < sampleNumber ; i++) {
        this->readSensors();           // Leemos los datos del MPU6050 3000 veces y :
        if(calculateAccelerometer){
            sumAcc.x += this->accelerometer.x;
            sumAcc.y+= this->accelerometer.y;
            sumAcc.z += this->accelerometer.z;
        }
        if(calculateGyroscope){
            sumGy.x += this->gyroscope.x;
            sumGy.y += this->gyroscope.y;
            sumGy.z += this->gyroscope.z;
        }
        delayMicroseconds(1000);
    }
    this->offsetAccelerometer.x = sumAcc.x / sampleNumber;
    this->offsetAccelerometer.y = sumAcc.y / sampleNumber;
    this->offsetAccelerometer.z = sumAcc.z / sampleNumber;
    this->offsetGyroscope.x = sumGy.x / sampleNumber;
    this->offsetGyroscope.y = sumGy.y / sampleNumber;
    this->offsetGyroscope.z = sumGy.z / sampleNumber;
}

```

Ilustración 55: Segunda Parte contenido MPU6050.cpp

El método **readSensors()** envía una petición de lectura de 14 bytes y esperamos a recibirlos, una vez recibido modificamos los valores del acelerómetro, giroscopio y temperatura que hemos recibido del sensor MPU6050

El método **processAccelerometer()** procesa los valores del acelerómetro para corregir el valor de offset respecto al valor leído del acelerómetro.

```
void MPU6050::readSensors() {
    transmitAndWaitResponse(0x3B, 14); //Pedimos 14 bytes

    this->accelerometer.x = Wire.read() << 8 | Wire.read();
    this->accelerometer.y = Wire.read() << 8 | Wire.read();
    this->accelerometer.z = Wire.read() << 8 | Wire.read();
    this->temperature = Wire.read() << 8 | Wire.read();
    this->gyroscope.x = Wire.read() << 8 | Wire.read();
    this->gyroscope.y = Wire.read() << 8 | Wire.read();
    this->gyroscope.z = Wire.read() << 8 | Wire.read();
}

void MPU6050::processAccelerometer(){
    //-----Corrección valores - offset
    accelerometer.x -= offsetAccelerometer.x;
    accelerometer.y -= offsetAccelerometer.y;
    accelerometer.z -= offsetAccelerometer.z;
    accelerometer.z = accelerometer.z + LSBAccelerometer;
}
```

Ilustración 56: Tercera parte MPU6050.cpp

El método **calculateAngle()** este método calcula el ángulo del eje Pitch y Roll, aplicando el correspondiente filtro Complementario explicado en el apartado (Parte Hardware) MPU6050. En primer lugar, se calibra el giroscopio eliminando el offset del valor real del giroscopio, esto se realiza en **recalibrateGyroscope()** (ilustración 58).

A continuación se calcula el ángulo según los valores leído del giroscopio esto valores se obtiene en ($^{\circ}$), una vez obtenido estos ángulos. Se calcula el ángulo según los valores leído del acelerómetro, como el acelerómetro nos da valores de velocidad angular, tenemos que convertirlo en ángulo, una vez obtenidos. Se aplica el filtro complementario para altas en giroscopio y para bajas en el

acelerómetro, el filtro complementario solo se aplica la primera vez, en los siguientes ciclos se retroalimenta con el valor obtenido en el ciclo anterior.

```
void MPU6050::calculateAngle() {
    Axis recalibrateG = recalibrateGyroscope();
    //----- Calculo del Angulo

    pitchAngle += recalibrateG.x * executionTime / 1000 ;
    rollAngle += recalibrateG.y * executionTime / 1000 ;

    pitchAngle += rollAngle * sin(recalibrateG.z * executionTime * (pi/(180*LSBGyroscope*1000)));      // tiempo
    rollAngle -= pitchAngle * sin(recalibrateG.z * executionTime * (pi/(180*LSBGyroscope*1000))); // (pi/(180*LSBGyroscope*1000))

    float totalVector = sqrt(pow(accelerometer.x, 2) + pow(accelerometer.y, 2) + pow(accelerometer.z, 2));
    float pitchAngleAcc = asin((float)accelerometer.y / totalVector) * 57.2958;      // 57.2958 = Conversion de radianes a grados
    float rollAngleAcc = asin((float)accelerometer.x / totalVector) * -57.2958;

    if(init){
        pitchAngle = pitchAngle * 0.99 + pitchAngleAcc * 0.01;      // Corregimos en drift con filtro complementario
        rollAngle = rollAngle * 0.99 + rollAngleAcc * 0.01;
    }else{
        pitchAngle = pitchAngleAcc;
        rollAngle = rollAngleAcc;
        init = true;
    }
}
```

Ilustración 57: Cuarta parte MPU6050.cpp

El método **updateLoopTimer()** actualiza el tiempo de lazo Loop, para calcular los ciclos del programa.

El método **updateExecutionTimer()** convierte el tiempo de lazo menos el tiempo actual del programa en segundos.

El método **transmitData()** se comunica con el sensor y modifica el registro que pase por argumento con el dato enviado por argumento.

El método **transmitAndWaitResponse()** se comunica con el sensor y pide al sensor una cantidad de byte y esperamos a recibir la cantidad de bytes que hemos solicitado.

```

Axis MPU6050::recalibrateGyroscope(){
    Axis recalibrate;
    recalibrate.x = (gyroscope.x - offsetGyroscope.x) / LSBGyroscope;
    recalibrate.y = (gyroscope.y - offsetGyroscope.y) / LSBGyroscope;
    recalibrate.z = (gyroscope.z - offsetGyroscope.z);
    return recalibrate;
}

void MPU6050::updateLoopTimer(){
    this->loopTimer = micros();
}

void MPU6050::updateExecutionTimer(long loopTimer){
    executionTime = (micros() - loopTimer) / 1000;
}

void MPU6050::transmitData(uint8_t reg,uint8_t data){
    Wire.beginTransmission(address);
    Wire.write(reg);
    Wire.write(data);
    Wire.endTransmission();
}

void MPU6050::transmitAndWaitResponse(uint8_t data, int numberByte){
    Wire.beginTransmission(address);
    Wire.write(data);
    Wire.endTransmission();
    Wire.requestFrom(address, numberByte); // Pedimos n numero de bytes //
    while (Wire.available() < numberByte);
}

```

Ilustración 58: Última parte MPU6050.cpp

Continuamos con las últimas declaraciones de las clases indicada en el archivo Main.cpp (ilustración 41), a continuación se declara el controlador de la cámara, esta estructura e implementación se incorpora en Camera/ControlPixy.h (ilustración 60) Camera/ControlPixy.cpp(ilustración 61)

```

#ifndef CAMERA_PIXY_C_
#define CAMERA_PIXY_C_

#include "../GlobalVariable.h"
#include "Pixy2.h"
#include "../Bluetooth/Bluetooth.h"

struct ControlPixy{
    private:
        Pixy2 pixy;
    public:
        void initialize();
        void updateBlocks();
        void updatePulse(Bluetooth bt);
};

#endif /* CAMERA_PIXY_C_ */

```

Ilustración 60: Contenido ControlPixy.h

La estructura del ControlPixy contiene el atributo pixy, que es una clase que importamos de la librería de Pixy2.

El método **initialize()**, inicializa la clase Pixy2 que viene incorporada en la librería de Pixy2

El método **updateBlock()**, refresca la cámara y escribe los vectores que detecta la cámara para poder analizar, el movimiento correspondiente en el método **updatePulse()**.

Este método **updatePulse()** analiza la dirección del vector del objeto línea que detecta la cámara, y nos indica si está centrado en el eje x entre [120,190] debemos seguir recto, para ello modificamos el valor del Eje Pitch, siempre y cuando el valor sea válido.

Si nos indica que el eje x esta entre [10,120] debemos girar hacia la derecha, para ello modificamos el Valor del Eje Yaw para producir una rotación, siempre y cuando la rotación hacia la derecha sea posible.

Si nos indica que el eje x esta entre [190] debemos girar hacia la izquierda, para ello modificamos el Valor del Eje Yaw para producir una rotación, siempre que la rotación hacia la izquierda sea posible.

```
#include "ControlPixy.h"

void ControlPixy::initialize(){
    pixy.init();
}

void ControlPixy::updateBlocks(){
    for (int i=0; i<pixy.line.numVectors; i++){
        pixy.line.vectors[i].print();
    }
}

void ControlPixy::updatePulse(Bluetooth bt){
    //avanzar
    if((pixy.line.vectors[0].m_x0>120)&&(pixy.line.vectors[0].m_x0<190)){
        if(bt.getPitch() > degreeMin){
            bt.modifyPitch(bt.getPitch() - incrementDegree);
        }
    }

    //girar a la derecha
    if((pixy.line.vectors[0].m_x0 < 120) && (pixy.line.vectors[0].m_x0 > 10)){
        if(bt.getYaw() > degreeMin){
            bt.modifyYaw(bt.getYaw() - incrementDegree);
        }
    }

    //girar a la izquierda
    if((pixy.line.vectors[0].m_x0 > 190)){
        if(bt.getYaw() < degreeMax){
            bt.modifyYaw(bt.getYaw() + incrementDegree);
        }
    }
}
```

Ilustración 61: Contenido del archivo ControlPixy.cpp

Continuamos con las últimas declaraciones de las clases indicada en el archivo Main.cpp (ilustración 41), a continuación se declara el controlador más

importante para controlar el Dron el controlador de modo, este se ubica en ControlMode/ControlMode.h (ilustración 62) y ControlMode/ControlMode.cpp (ilustración 63)

```
struct ControlMode {  
    private:  
        float pidAnglePitch,pidAngleRoll;  
        float actionIntegralPitchW,actionIntegralRollW,actionIntegralYawW;  
        float actionDerivativePitchW,actionDerivativeRollW,actionDerivativeYawW;  
        float actionIntegralPitchAng,actionIntegralRollAng;  
        float actionDerivativePitchAng,actionDerivativeRollAng;  
        float outPidPitch,outPidRoll,outPidYaw;  
        bool modeAcrobatic,modeUp,modeDown,modeAutomatic;  
        bool calculateAccelerometer;  
        bool calculateGyroscope;  
    public:  
        ControlMode();  
        void PIDSpeed(Bluetooth bt,MPU6050 mpu);  
        void PIDAnglee(Bluetooth bt,MPU6050 mpu);  
        void activateModeAcrobatic();  
        void activateModeStable();  
        void activateModeDown(Bluetooth bt);  
        void activateModeUp(Bluetooth bt);  
        void activateModeAutomatic(Bluetooth bt);  
        void onLedAccordingMode(Leds leds);  
  
        float calculateOutPID(float pidError,float valueGiroscope,float kp,float act  
        bool isModeAcrobatic();  
        bool isModeDown();  
        bool isModeUp();  
        bool isModeAutomatic();  
  
        bool isCalculateAcelerometer();  
        bool isCalculateGyroscope();
```

Ilustración 62: Contenido del archivo ControlMode.h

La estructura de ControlMode contiene varios atributos, y vamos a proceder a definirlos.

pidAnglePitch, pidAngleRoll estos valores representan el ángulo de los dos ejes después de haber sido procesado por el PID según el ángulo.

actionIntegralPitchW,actionIntegralRollW y **actionIntegralYawW** estos son los valores de la acción Integral que se va retroalimentando cada vez que ejecutamos el controlador PID según la velocidad de los ángulos.

actionDerivativePitchW,actionDerivativeRollW y **actionDerivativeYawW**

estos son los valores de la acción derivativa que se va retroalimentando cada vez que ejecutamos el controlador PID según la velocidad de los ángulos.

actionIntegralPitchAng,actionIntegralRollAng estos son los valores de la acción Integral que se va retroalimentando cada vez que ejecutamos el controlador PID según el ángulo.

actionDerivativePitchAng,actionDerivativeRollAng estos son los valores de la acción derivative que se va retroalimentando cada vez que ejecutamos el controlador PID según el ángulo.

outPidPitch, outPidRoll, outPidYaw estos son los valores de salida de cada eje de PID del controlador de PID de velocidad.

modeAcrobatic, modeUp, modeDown, modeAutomatic estas variables booleanas indica si está activo el modo o no.

calculateAccelerometer y **calculateGyroscope** estas variables booleanas indican si tenemos que calcular el offset para el acelerómetro y si tenemos que calcular el offset para el Giroscopio.

El constructor de ControlMode inicializada todas las variables con elementos neutros, y por defecto dejamos activo el modo estable.

```
#include "ControlMode.h"

ControlMode::ControlMode(){
    this->calculateAccelerometer = true;
    this->calculateGyroscope = true;
    this->modeAcrobatic = false;
    this->modeAutomatic = false;
    this->modeDown = false;
    this->modeUp = false;
    this->pidAnglePitch = 0.0;
    this->pidAngleRoll = 0.0;
    this->actionIntegralPitchW = 0.0;
    this->actionIntegralRollW = 0.0;
    this->actionIntegralYawW = 0.0;
    this->actionDerivativePitchW = 0.0;
    this->actionDerivativeRollW = 0.0;
    this->actionDerivativeYawW = 0.0;
    this->actionIntegralPitchAng = 0.0;
    this->actionIntegralRollAng = 0.0;
    this->actionDerivativePitchAng = 0.0;
    this->actionDerivativeRollAng = 0.0;
    this->outPidPitch = 0.0;
    this->outPidRoll = 0.0;
    this->outPidYaw = 0.0;
}
```

Ilustración 63: Primera parte ControlMode.cpp

El método **PIDAngle()** se refiere al controlador PID para el ángulo, este método se explicara mejor en el modo Estable.

```
void ControlMode::PIDAngle(Bluetooth bt, MPU6050 mpu){
    //===== PITCH GYRO ang =====
    float pidErrorPitch = bt.getPitch() - mpu.getPitchAngle(); // error entre le
    actionIntegralPitchAng += (KiPitchAngle * pidErrorPitch); // Parte integral (sum
    actionIntegralPitchAng = constrain(actionIntegralPitchAng, -limitAngle, limitAngle); // Limitar parte integral

    outPidPitch = calculateOutPID(pidErrorPitch, mpu.getPitchAngle(), KpPitchAngle, actionIntegralPitchAng, KdPitchAngle,
    actionDerivativePitchAng = mpu.getPitchAngle()); // Error anterior
    //===== ROLL GYRO ang =====
    float pidErrorRoll = bt.getRoll() - mpu.getRollAngle();
    actionIntegralRollAng += (KirollAngle * pidErrorRoll);
    actionIntegralRollAng = constrain(actionIntegralRollAng, -limitAngle, limitAngle);

    outPidRollAngle = calculateOutPID(pidErrorRoll, mpu.getRollAngle(), KpRollAngle, actionIntegralRollAng, KdRollAngle, a
    actionDerivativeRollAng = mpu.getRollAngle();
}
```

Ilustración 64: Controlador PID para el Angulo ControlMode.cpp

El método **PIDSspeed()** se refiere al controlador PID según la velocidad, este método se explicara mejor en el modo Estable.

```
void ControlMode::PIDSspeed(Bluetooth bt, MPU6050 mpu){
    float inPitch,inRoll;
    float pidErrorPitch,pidErrorRoll,pidErrorYaw;
    if (modeAcrobatic) {
        inPitch = bt.getPitch();
        inRoll = bt.getRoll();
    }else {
        inPitch = pidAnglePitch;
        inRoll = pidAngleRoll;
    }

    Axis recalibrateGyro = mpu.recalibrateGyroscope();
    recalibrateGyro.z = recalibrateGyro.z / LSBgyroscope;

    //===== PITCH GYRO w =====
    pidErrorPitch = inPitch - recalibrateGyro.x;
    actionIntegralPitchW += (KiPitchW * pidErrorPitch);
    actionIntegralPitchW = constrain(actionIntegralPitchW, -limitVelocity, limitVelocity);

    outPidPitch = calculateOutPID(pidErrorPitch, recalibrateGyro.x,KpPitchW,actionIntegralPitchW,KdPitchW,actionDerivativePitchW,limitVelocity);
    actionDerivativePitchW = recalibrateGyro.x;
    //===== ROLL GYRO w =====
    pidErrorRoll = inRoll - recalibrateGyro.y;
    actionIntegralRollW += (KirollW * pidErrorRoll);
    actionIntegralRollW = constrain(actionIntegralRollW, -limitVelocity, limitVelocity);

    outPidRoll = calculateOutPID(pidErrorRoll, recalibrateGyro.y,KpRollW,actionIntegralRollW,KdRollW,actionDerivativeRollW,limitVelocity);
    actionDerivativeRollW = recalibrateGyro.y;
    //===== YAW GYRO w =====
    pidErrorYaw = bt.getYaw() - recalibrateGyro.z;
    actionIntegralYawW += (KiyawW * pidErrorYaw);
    actionIntegralYawW = constrain(actionIntegralYawW, -limitVelocity, limitVelocity);

    outPidYaw = calculateOutPID(pidErrorYaw, recalibrateGyro.z,KpYawW,actionIntegralYawW,KdYawW,actionDerivativeYawW,limitVelocity);
    actionDerivativeYawW = recalibrateGyro.z;
}
```

Ilustración 65 : implementación PID según la velocidad

El método **activateModeAcrobatic()** y **activateModeStable()** activa y desactiva los valores correspondiente para este modo.

```

void ControlMode::activateModeAcrobatic(){
    this->modeAcrobatic = false;
    this->calculateAccelerometer = false;
    this->calculateGyroscope = true;
    this->modeAutomatic = false;
    this->modeDown = false;
    this->modeUp = false;
}

void ControlMode::activateModeStable(){
    this->modeAcrobatic = true;
    this->calculateAccelerometer =true;
    this->calculateGyroscope = true;
    this->modeAutomatic = false;
    this->modeDown = false;
    this->modeUp = false;
}

```

Ilustración 66: activación modo acrobático y modo estable

El método `activateModeUp()`,`activateModeDown()` y `activateModeAutomatic()` activa y desactiva los valores correspondiente para este modo y modifica los valores de los Ejes del controlador Bluetooth.

```

void ControlMode::activateModeUp(Bluetooth bt){
    bt.modifyPitch(0);
    bt.modifyRoll(0);
    bt.modifyYaw(0);
    this->modeUp = true;
    this->calculateAccelerometer = true;
    this->calculateGyroscope = true;
    this->modeAutomatic = false;
    this->modeDown = false;
    this->modeAcrobatic = false;
}

void ControlMode::activateModeDown(Bluetooth bt){
    bt.modifyPitch(0);
    bt.modifyRoll(0);
    bt.modifyYaw(0);
    this->modeDown = true;
    this->calculateAccelerometer =true;
    this->calculateGyroscope = true;
    this->modeAutomatic = false;
    this->modeUp = false;
    this->modeAcrobatic = false;
}

```

Ilustración 67: activación modo despegue y modo aterrizaje

El método `onLedAccordinMode()` activara los leds correspondiente segúin el modo de vuelo.

```

    void ControlMode::activateModeAutomatic(Bluetooth bt){
        bt.modifyPitch(0);
        bt.modifyRoll(0);
        bt.modifyYaw(0);
        this->modeAutomatic = true;
        this->calculateAccelerometer =true;
        this->calculateGyroscope = true;
        this->modeDown = false;
        this->modeUp = false;
        this->modeAcrobatic = false;
    }

    void ControlMode::onLedAccordingMode(Leds leds){

        if(modeAcrobatic){
            leds.onLedBlue();
            leds.onLedFly();
        }else if(modeAutomatic){
            leds.onLedWhite();
        }else if(modeUp){
            leds.onLedBlue();
        }else if(modeDown){
            leds.onLedWarning();
        }else{
            leds.onLedFly();
        }
    }
}

```

Ilustración 68: activación modo automático y acción de los leds según el modo

Una vez declarado todo los componentes, y explicado todas sus métodos, procedemos a analizar el `setup()` del programa principal Main.cpp

(ilustración 69)

```

void setup() {
    bluetooth.initialize();

    pixy.initialize();

    leds.initialize();
    leds.offAllLeds();
    leds.onLedBlue();

    if(!mpu6050.initialize() || bateria.isLowBattery()){
        leds.offAllLeds();
        leds.onLedWarning();
        Serial.println("ERROR no se inicia MPU o la Batería esta baja");
        while (true)delay(10);
    }

    mpu6050.calculateOffset(controlMode.isCalculateGyroscope(),controlMode.isCalculateAcelerometer());

    Serial.begin(baudrates);

    motor1.initialize();
    motor2.initialize();
    motor3.initialize();
    motor4.initialize();

    Serial.print("Conectarse al dron y bajar throttle al minimo");
    while (bluetooth.getThrottle() > degreeMinThrottle){
        bluetooth.updatePulse();
    }

    leds.offAllLeds();
    mpu6050.updateLoopTimer();
}

```

Ilustración 69: setup del programa principal Main.cpp

En primer lugar inicializamos el Bluetooth, la cámara y los leds.

Y encendemos los leds azules hasta que acabe la configuración.

Si por alguna razón el MPU no se inicializa correctamente o la Batería esta descargada, encenderemos las luces rojas y dejaremos el programa bloqueado, para que no pueda continuar. En otro caso calculamos el offset del MPU6050, inicializamos el Serial y los motores.

Y a continuación esperamos a que nos conectemos con el bluetooth y bajemos el Throttle al minimo, para activar los motores y por precaución.

Finalmente apagamos todos los leds y refrescamos el valor del tiempo del loop.

Finalmente analizamos el loop del programa principal (ilustración 70)

```
void loop() {  
    controlMode.onLedAccordingMode(leds);  
    while (micros() - mpu6050.getLoopTimer() < usCiclo);  
    mpu6050.updateLoopTimer();  
    leds.offAllLeds();  
  
    if(controlMode.isCalculateGyroscope()) {  
        mpu6050.calculateAngle();  
    }  
  
    if(!controlMode.isModeAcrobatic()){  
        controlMode.PIDAngle(blueooth, mpu6050);  
    }  
    controlMode.PIDSpeed(blueooth, mpu6050);  
  
    motor1.controller(blueooth.getThrottle(), controlMode.getPidPitch(),-controlMode.getPidRoll(), -controlMode.getPidYaw());  
    motor2.controller(blueooth.getThrottle(), controlMode.getPidPitch(), controlMode.getPidRoll(), controlMode.getPidYaw());  
    motor3.controller(blueooth.getThrottle(), -controlMode.getPidPitch(), controlMode.getPidRoll(), -controlMode.getPidYaw());  
    motor4.controller(blueooth.getThrottle(), -controlMode.getPidPitch(),-controlMode.getPidRoll(), controlMode.getPidYaw());  
  
    if(!controlMode.isModeUp() && !controlMode.isModeDown() && !controlMode.isModeAutomatic()){  
        blueooth.updatePulse();  
    }  
  
    if(controlMode.isModeUp()){  
        int throttle = blueooth.getThrottle() + incrementThrottle;  
        if(throttle <= degreeMaxUp){  
            blueooth.modifyThrottle(throttle);  
        }  
    }  
  
    if(controlMode.isModeDown() || bateria.isLowBattery()){  
        int throttle = blueooth.getThrottle() - incrementThrottle;  
        if(throttle >= degreeMinThrottle){  
            blueooth.modifyThrottle(throttle);  
        }  
    }  
  
    if(controlMode.isModeAutomatic()){  
        pixy.updateBlocks();  
        pixy.updatePulse(blueooth);  
    }  
  
    mpu6050.readSensors();  
    if(controlMode.isCalculateAccelerometer()){  
        mpu6050.processAccelerometer();  
    }  
}
```

Ilustración 70: implementación loop contenido en Main.cpp

En primer lugar activamos los leds según el modo que tenemos seleccionado, después realizamos el nuevo ciclo del programa y refrescamos el valor de tiempo del loop.

Luego apagamos todos los leds, según si tenemos activado el cálculo de giroscopio realizamos el cálculo del ángulo.

A continuación si no tenemos activo el modo acrobático, es decir, si es cualquier otro modo menos el acrobático, calculamos el controlador del PID para el ángulo en los ejes Pitch y Roll.

Luego realizamos el cálculo del PID según la velocidad, una vez calculado los valores de PID activamos los motores.

Para activar los motores tenemos que saber cuál es cada motor (ilustración 3) y saber que por ejemplo si giramos de forma positiva en el eje Pitch tenemos que acelerar el motor (1, 2) y reducir (3, 4). Esto se especificó en el apartado Conceptos Generales dentro del Cuerpo del Trabajo.

Una vez actualizado los motores, actualizamos los valores de los ejes si no es ninguno de los modos automáticos.

Si el modo activo es el modo despegue vamos aumentando en cada ciclo si no hemos llegado a la altura indicada.

Si el modo activo es el modo aterrizaje vamos reduciendo el Throttle, hasta llegar al mínimo del motor.

Si el modo activo es el modo automático vamos actualizando las lecturas de la cámara y modificamos los ejes según la dirección a seguir.

Finalmente leemos los valores del MPU, si está activo el cálculo del acelerómetro recalibraremos los datos del acelerómetro.

- **Modo Acrobático:** En este modo de control solo se utiliza las lecturas de velocidad angular calculadas a partir de los datos obtenidos del

sensor MPU6050. Si por ejemplo uno de los ejes da una vuelta completa en un segundo, la velocidad será de 360º/s.

Si el eje Pitch del Dron rotara por cualquier razón, porque uno de los motores tiene más potencia, porque hay viento... el control tendrá que contrarrestar esta desviación actuando sobre los motores correspondientes. En este caso, el acelerando o reduciendo el motor (2, 4) y (1, 3)(ilustración 3). Pero el Dron no volverá a su posición inicial 0, simplemente compensara la rotación hasta detener el Dron. Esto es debido a que únicamente se utiliza como referencia la velocidad de rotación de los ejes.

Esto se consigue con el PID (controlador proporcional, integral y derivativo) es un mecanismo de control simultáneo por realimentación. Este calcula la desviación o error entre un valor medido y un valor deseado.

El algoritmo del control PID consta de tres parámetros distintos: el **proporcional**, el **integral**, y el **derivativo**. El valor proporcional depende del error actual, el integral depende de los errores pasados y el derivativo es una predicción de los errores futuros. La suma de estas tres acciones es usada para ajustar el proceso por medio de un elemento de control.

Ajustando estas tres variables en el algoritmo de control del PID, el controlador puede proveer una acción de control adaptada a los requerimientos del proceso en específico (ilustración 71).

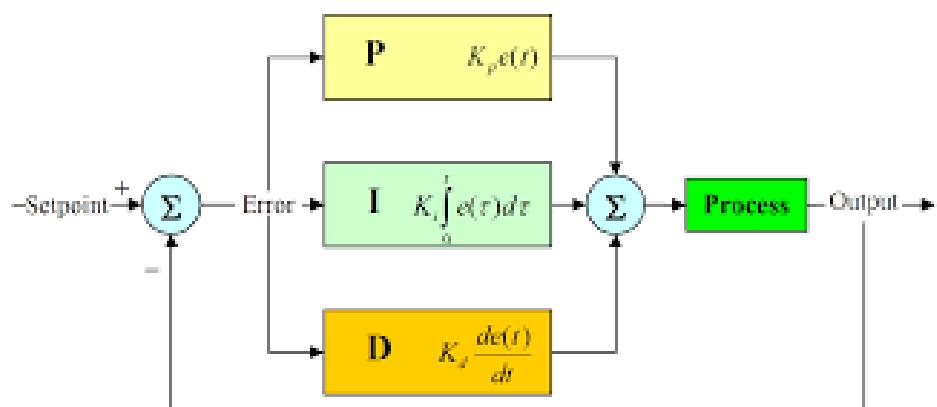


Ilustración 71: Diagrama controlador PID

Como se ve, los encargados de calcular el error de cada ángulo y actuar en consecuencia son los PID.

El objetivo de un PID es conseguir un error entre la consigna de velocidad y la velocidad real de 0º/s (metros, grados... según la aplicación), es decir, que la velocidad de rotación real sea igual a la consigna que llega desde el Bluetooth.

Lo primero que hace esta estructura de control es comparar la referencia de **velocidad angular** que nos llega desde el **Bluetooth**, con la lectura que recibimos del sensor **MPU6050**. Haciendo la resta de estas dos señales se consigue el valor de desviación o error del eje.

Tras acelerar los motores, el Dron empezará a rotar en el sentido que hayamos indicado hasta alcanzar la velocidad deseada, momento en el que el error bajará a 0º/s y habrá finalizado la operación.

Este error lo recibe el PID y genera una salida en función de los parámetros **K_p** (**proporcional**), **K_i** (**integral**) y **K_d** (**derivativo**) que hayamos establecido.

Para establecer los valores se debe saber a qué se refiere cada uno:

La **acción proporcional** $P(t)$, amplia o reduce el error en función de una ganancia proporcional.

$$P(t) = K_p * \text{error}(t)$$

K_p es la ganancia proporcional que establecemos.

La **acción integral** $I(t)$, acumula el error a lo largo del tiempo. Se coge el error actual y lo se multiplica por el término **K_i**, pero en cada nuevo ciclo de control sumamos el valor obtenido en el ciclo anterior. De esta forma conseguimos que el error en régimen permanente sea de 0.

$$I(t) = K_i \int_0^t \text{error}(t)dt$$

La acción derivativa $D(t)$, evalúa el incremento del error en función del tiempo, ponderado mediante la ganancia derivativa **K_d**. Sirve para suavizar la respuesta del control.

$$D(t) = K_d * \frac{d}{dt} \text{error}(t)$$

La expresión final de la acción de control es la suma de cada componente.

$$U(t) = C + P(t) + I(t) + D(t)$$

Con lo cual la estrategia para el PID en el modo acrobático quedaría representada en la (ilustración 72)

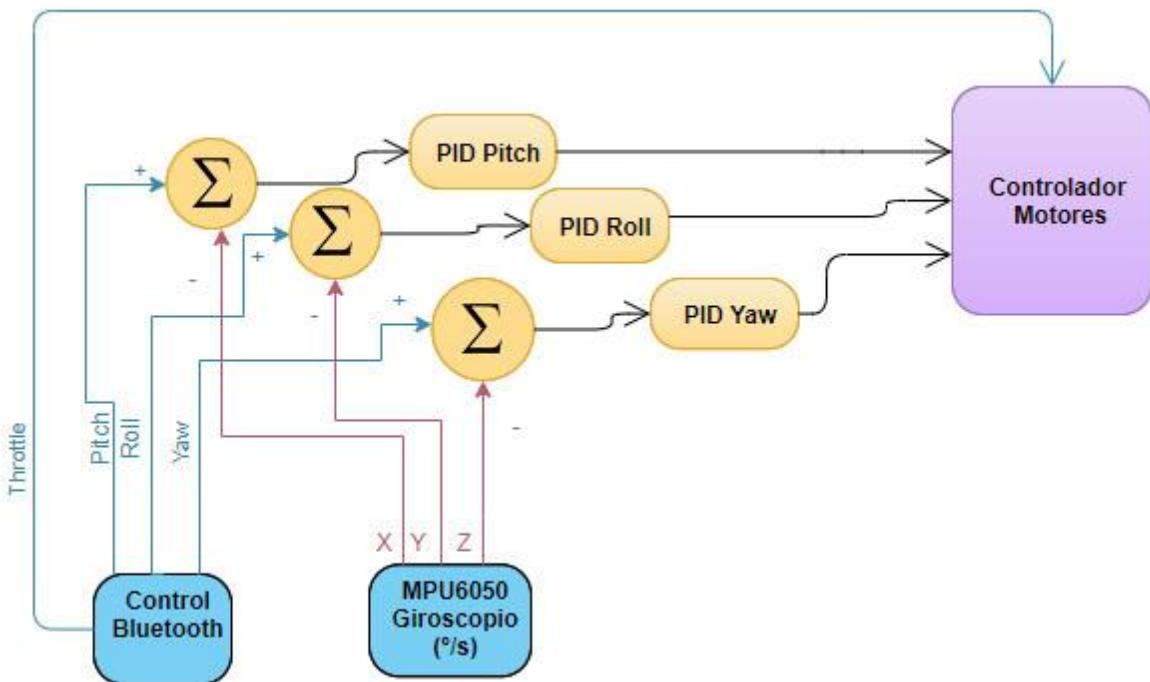


Ilustración 72: Diagrama PID modo acrobático

A la hora de implementar el controlador PID en el modo acrobático, la única variación, es como se observa en la (ilustración 65), la entrada del Eje Pitch y el Eje Roll, corresponde al valor actual de nuestros controles manuales.

En primer lugar recalibraremos el valor del giroscopio, a continuación calculamos el error comparando la entrada del eje Pitch menos el valor recibido por el MPU6050.

Una vez tenemos el Error del Pitch calculamos la acción integral multiplicando la constante integral por el error del Pitch y limitamos la salida de la acción integral a la amplitud de velocidad establecida en GlobalVariable.h

Una vez tenemos el valor de la acción integral con su retroalimentación y limitado, calculamos el PID de salida para el Pitch calculando la acción derivativa y agrupando todas las acciones, obteniendo así la salida del PID para el Pitch, por ultimo almacenamos en la acción derivativa el valor del error del ciclo actual para realizar así la retroalimentación en el próximo ciclo.

Finalmente volvemos a proceder de la misma forma con los otro PID de los ejes Roll y Yaw.

- **Modo Estable:** Este modo necesita de dos PID en cascada por cada eje a controlar, además de lecturas de velocidad de rotación y aceleración del sensor MPU6050 a partir de los cuales calcular el ángulo de inclinación de cada eje en grados ($^{\circ}$). La ventaja de este modo de vuelo es que el Dron es completamente estable. Al contrario que en el modo acrobático, cuando los valores de Pitch, Roll o Yaw vuelve a su posición de 0° de inclinación, el Dron volverá automáticamente a su posición de 0° . La siguiente figura muestra la estrategia de control (ilustración 73). Es parecida a la figura mostrada para el modo acrobático, solo que utilizando un PID más en los ejes pitch y roll. El primer PID toma la lectura de inclinación ($^{\circ}$) calculada a partir de las lecturas del sensor MPU6050 y la compara con la que se manda por él Bluetooth. Si hay una desviación de inclinación, este primer PID genera una referencia de velocidad para el siguiente lazo, acelerando los correspondientes motores y contrarrestando la inclinación. El segundo PID controla la velocidad a la que rota el Dron mientras contrarresta la inclinación.

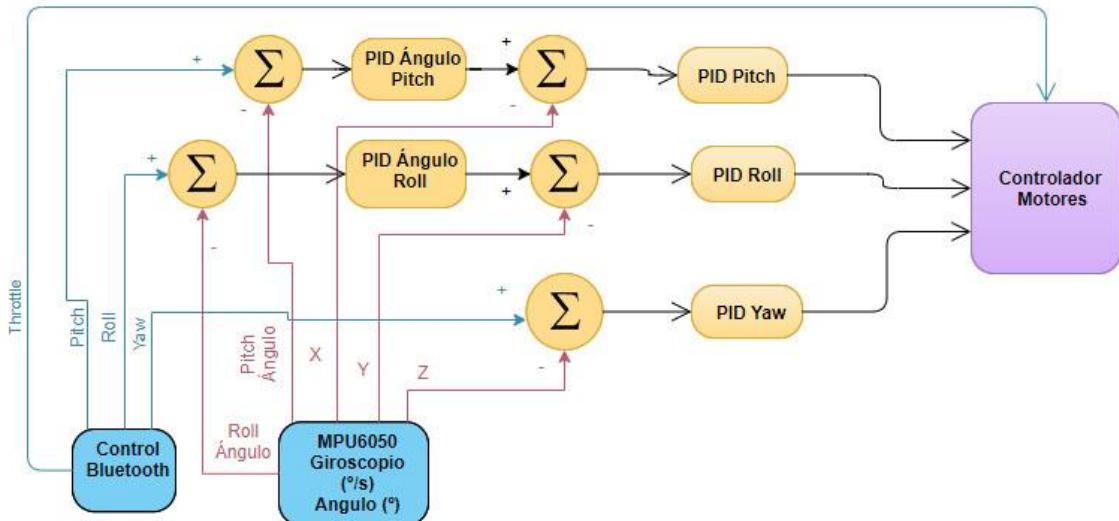


Ilustración 73: Diagrama PID modo estable

A la hora de implementar el controlador PID en el modo estable, se tiene que implementar el cálculo generar los ángulos Pitch y Roll, aplicando el filtro complementario en el MPU6050 (ilustración 57 y 58)

Una vez que tenemos el valor de los ángulos Pitch y Roll, pasamos al cálculo del controlador PID para el ángulo (ilustración 64).

En primer lugar obtenemos el error del Pitch, comparando el valor leído por el Bluetooth y el ángulo que recibimos por el MPU6050

En segundo lugar calculamos la acción integral del PID y los limitamos para que no de variables indeseadas.

A continuación calculamos la salida del PID pitch para el ángulo, calculando la acción derivativa y se limita el resultado para obtener variables dentro del rango. Por ultimo retroalimentamos la acción derivativa con el valor del ángulo que hemos usado en este ciclo para el cálculo del PID.

Y repetimos la misma operación realizada en el eje Pitch, para el eje Roll.

Finalmente realizamos la misma operación que para el modo acrobático, solo que ahora en el PID según la velocidad, el valor de entrada para PID es el ángulo obtenido en el PID anterior.

- **Modo Despegue:** Este modo utiliza el mismo controlador PID que el modo estable, la única variación es la entrada del PID(ilustración 74).

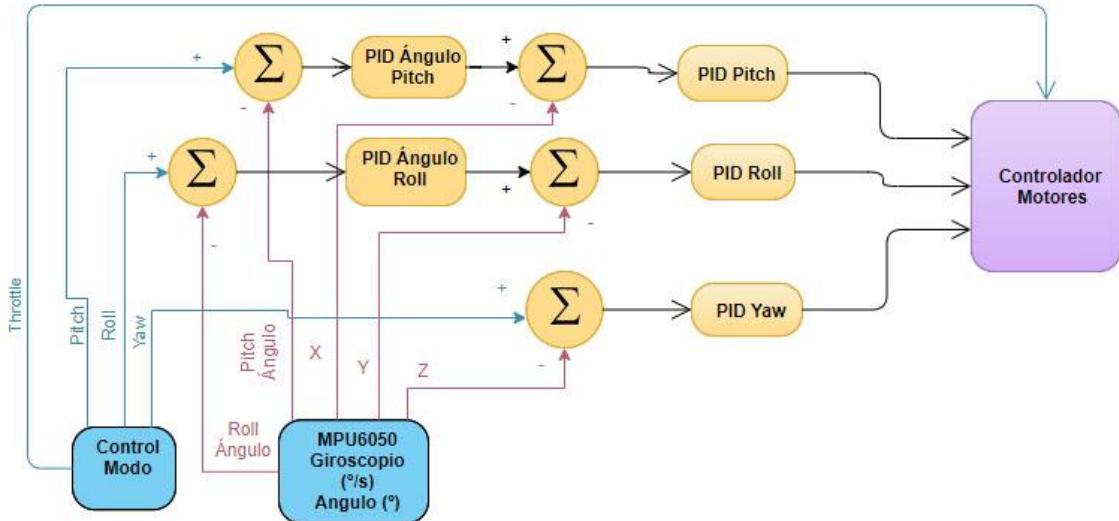


Ilustración 74: Diagrama PID para el modo Despegue y Aterrizaje

La entrada ahora será para los ejes Pitch, Roll y Yaw de 0° y para el valor Throttle irá aumentando en cada ciclo hasta llegar a valor definido como la altura máxima para este modo (ilustración 66).

Condicionamos, para desactivar la modificación de pulso del Bluetooth, ya que los tres modos que se indica, en la condición, se maneja de forma automática.

```
if(!controlMode.isModeUp() && !controlMode.isModeDown() && !controlMode.isModeAutomatic()){
    bluetooth.updatePulse();
}
```

☞ **Advertencia:** El Bluetooth no se bloquea solo la actualización de los pulsos de control, ya que si recibimos algún código de error o cambiar de modo a través del Bluetooth, lo tiene que permitir.

El valor del Throttle va a ir aumentando hasta llegar a la altura definida como máxima, como se puede ver en la ilustración.

```

if(controlMode.isModeUp()){
    int throttle = bluetooth.getThrottle() + incrementThrottle;
    if(throttle <= degreeMaxUp){
        bluetooth.modifyThrottle(throttle);
    }
}

```

- **Modo Aterrizaje:** Este modo utiliza el mismo controlador PID que el modo estable, la única variación es la entrada del PID(ilustración 74).

La entrada ahora será para los ejes Pitch, Roll y Yaw de 0° y para el valor del Throttle comenzara desde el último valor actual e ira decreciendo en cada ciclo hasta llegar a valor definido como mínimo (ilustración 66).

- **Modo Automático:** Este modo utiliza el mismo controlador PID que el modo estable, la única variación es la entrada del PID ahora es con el controlador de la cámara Pixy, en vez del Bluetooth(ilustración 75).

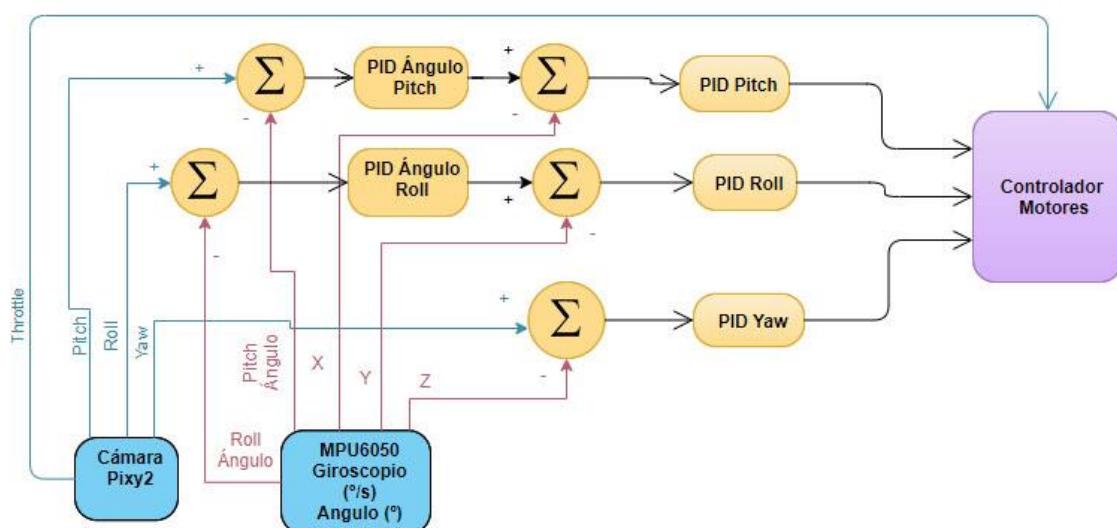


Ilustración 75: Diagrama PID para el modo Automático

Pasemos a su implementación, para el control del PID a través de la cámara bloqueamos la modificación de pulso a través del Bluetooth
Seguidamente vamos actualizando la lectura y según las lecturas modificamos los pulsos que llegaran al PID

```

if(controlMode.isModeAutomatic()){
    pixy.updateBlocks();
    pixy.updatePulse(blueooth);
}

```

Para controlar la cámara Pixy se ha implementado una clase especificada ubicada en Camera/ControlPixy donde podemos encontrar su estructura en el archivo ControlPixy.h (ilustración 60) y ControlPixy.cpp (ilustración 61), esta clase se ha especificado en el apartado Parte Software, Concepto General

II.IV. Relación de Coste

En este apartado se realiza una enumeración de los elementos utilizados y su correspondiente precio unitario.

En primer lugar se evalúan los costes del material (Tabla 1) de un solo ejemplar.

Material	Unidades	Precios Unidad	Precio Total
Frame 450	1	15	15
Arduino Uno	1	22	22
Bluetooth HC-06	1	2	2
Placa distribución de Potencia	1	3	3
Tira led RGBW	1	2	2
ESC, Motor Brushless	4	5,4	21,6
Hélices 1045	10	0,5	5
Resistencia 270KOhm	10	0,02	0,2
Resistencia 180KOhm	10	0,02	0,2
Mosfet IRFZ44N	10	0,1	1
Pack Ficha terminales	1	9,5	9,5
Bridas	50	0,06	3
Pixy2	1	66	66
MPU6050	1	0,65	0,65
Cargador Batería Imax B6	1	23	23
Precio Total			174,15€

Tabla 1: Costes del Material

En segundo lugar, realizamos el cálculo de los costes de Recursos Humanos (Tabla 2)

Roles	Precio/Hora	Horas Estimadas	Precio Total
Diseño para el Montaje	20	30	600
Construcción	10	80	800
Implementación del Software	30	200	6000
310			7400€

Tabla 2: Costes del Recursos Humanos

Juntando ambos costes, obtenemos el coste económico que representa el proyecto (Tabla 3).

Conceptos	Coste Estimado
Costes de Recursos Humanos	174,15
Coste del Material	7400
	7574,15€

Tabla 3: Coste Estimado

Realizando un análisis superficial de los precios actuales del mercado, se ofertan drones de distintas envergaduras y potencias, con una base de 400€ a nada que se busque un poco de fiabilidad.

Hay otros ejemplares con mayor calidad de materiales y precisión de control que alcanzan cifras de 10.000 o 15.000€.

Al estar realizado con un código abierto como Arduino, la venta de este producto podría proporcionar al cliente la oportunidad de controlar ciertos parámetros y realizar añadidos lo cual vuelve al producto un elemento muy versátil.

Para estipular una estimación del precio de venta al público para el Dron sería

Concepto	Unidades	Coste Estimado	Precio Total
Coste del Material	100	174,15	17415
Coste de Diseño	1	600	600
Coste Construcción	100	800	80000
Coste Software	1	6000	6000
			104015€

Nos daría un total de 104015€ con lo cual la unidad saldría a 1040,15€ por unidad con lo cual para obtener beneficios la venta al público del Dron sería 1150€ lo cual daría un beneficio de 10985€, al vender las 100 unidades.

II.V. Metodología de Trabajo

La duración del proyecto es aproximadamente 3 meses. La fecha de inicio del proyecto es el 20 de junio del 2020 y la fecha de finalización es el 20 de septiembre de 2020.

Es importante tener en cuenta que las fechas calculadas en la planificación inicial pueden ser modificadas y actualizadas durante la realización del proyecto, debido a que pueden surgir complicaciones inesperadas.

La metodología usada es de desarrollo ágil. Consecuentemente si las etapas mencionadas en el punto anterior tienen una duración diferente de lo esperado, se modificarán consecuentemente. Por ejemplo, si la etapa tiene una duración inferior de lo esperado, se iniciará de inmediato la siguiente. Aunque, si la tarea dura más de lo esperado, que retrasará las siguientes tareas.

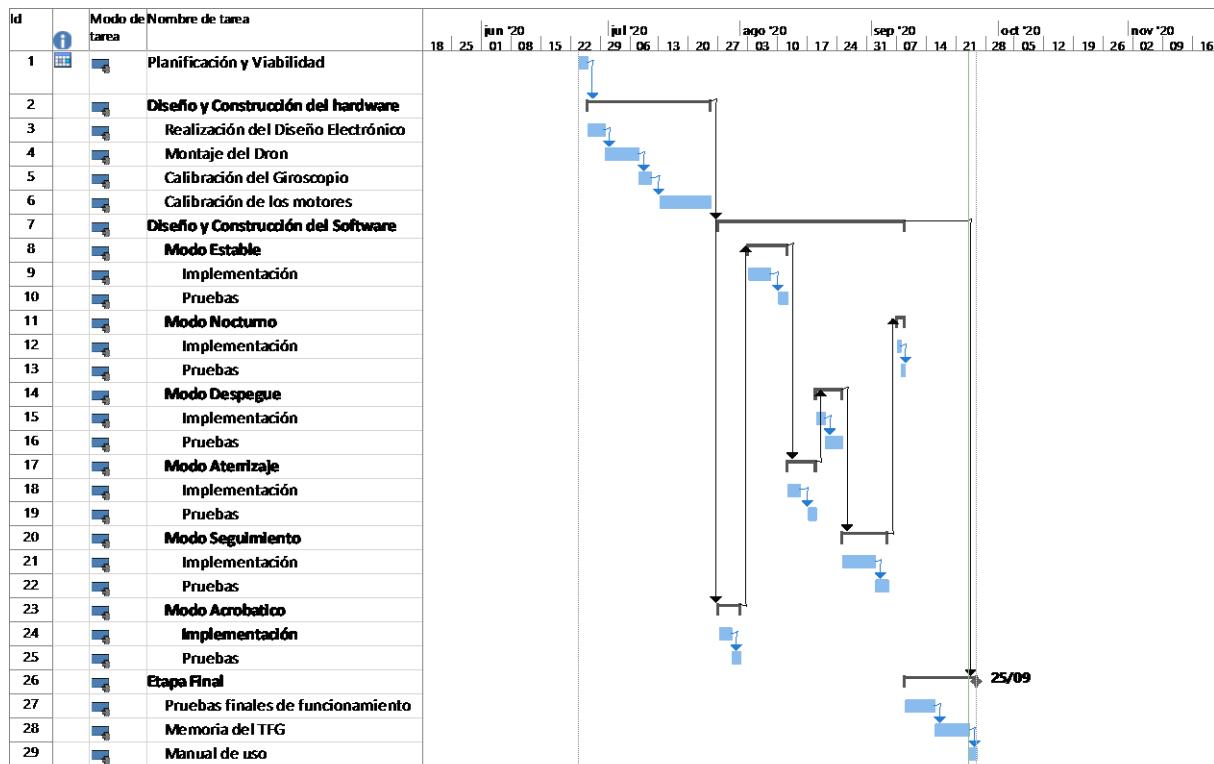


Tabla 4: Diagrama de Gantt

II.VI. Conclusión

- Se ha diseñado el sistema completo de un Dron de tipo Quadcopter con el análisis de sus componentes y funciones y la cohesión entre ellos.
- El esquema eléctrico de conexiones del sistema ha sido diseñado y desarrollado mediante el software Proteus.
- Se ha implementado el análisis de imágenes con la cámara Pixy y sus librerías, para darle una automatización al Dron.
- La programación del código en Arduino, permite la actuación de los distintos componentes y funciones de forma coordinada. Entre los objetivos alcanzados se encuentran la obtención del ángulo de los sensores, su conversión a un valor estable mediante el filtrado, la asignación de la acción de control de los actuadores, la medida de carga de la batería y la preparación del control PID para su aplicación. La configuración de Pixy ha sido implementada con éxito permitiendo la comunicación con el controlador de motores. A su vez en Arduino se ha conseguido realizar un código que permite el control de los parámetros del Dron, según la dirección del circuito, que capte la cámara.

II.VII. Líneas Futuras

- Construcción mediante impresión 3D de una estructura para recubrir el Dron y que disponga de apoyos.
- Construcción mediante impresión 3D de unas protecciones para las hélices

- Ampliar funcionalidad, con una incorporación de cámara GoPro, para un uso más comercial.
- Inclusión del compás HMC6352 y el sensor de presión BMP 180.
- Programación del filtro Kalman.

III. Bibliografía

- https://naylampmechatronics.com/blog/15_Configuraci%C3%B3n--del-m%C3%B3dulo-bluetooth-HC-06-usa.html
- <https://arduproject.es/conceptos-generales-sobre-drones/>
- <https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:start>
- <https://www.luisllamas.es/medir-la-inclinacion-imu-arduino-filtro-complementario/>
- <https://www.luisllamas.es/programar-arduino-con-eclipse/>
- <https://robologs.net/2016/02/01/programacion-de-un-esc-con-arduino/>
- <https://www.murkyrobot.com/guias/actuadores/motores-brushless>
- <https://cdn.sparkfun.com/datasheets/Sensors/Accelerometers/RM-MPU-6000A.pdf>
- https://naylampmechatronics.com/blog/45_Tutorial-MPU6050-Aceler%C3%B3metro-y-Giroscopio.html
- <https://robologs.net/2014/10/15/tutorial-de-arduino-y-mpu-6050/>

- <https://chespix.wordpress.com/2013/09/13/tiras-led-rgb-anodo-comun-y-arduino/>
- <https://www.hwlible.com/imax-b6/#:~:text=bater%C3%ADas%20y%20acumuladores,,%C2%BFQu%C3%A9%20es%20IMAX%20B6%3F,de%20hasta%206%20celulas%2C%20etc.>
- Building a Quadcopter with Arduino (Vasilis Tzivaras)
ISBN 978-1-78528-184-6
- https://es.wikipedia.org/wiki/Controlador_PID

IV. Apéndice Manual Usuario

Para comenzar a usar el Dron comenzamos conectando la conexión de la batería (ilustración 76)

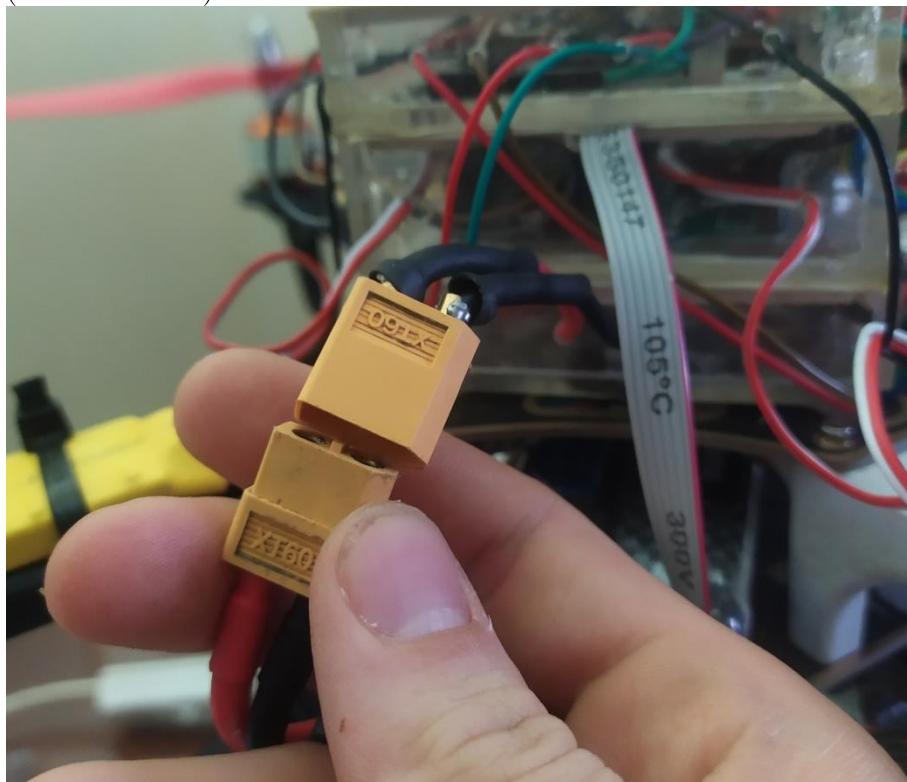


Ilustración 76: Conexión XT60 de la batería Lipo

Seguidamente conectamos la conexión Jack con la placa de Arduino (ilustración 77)

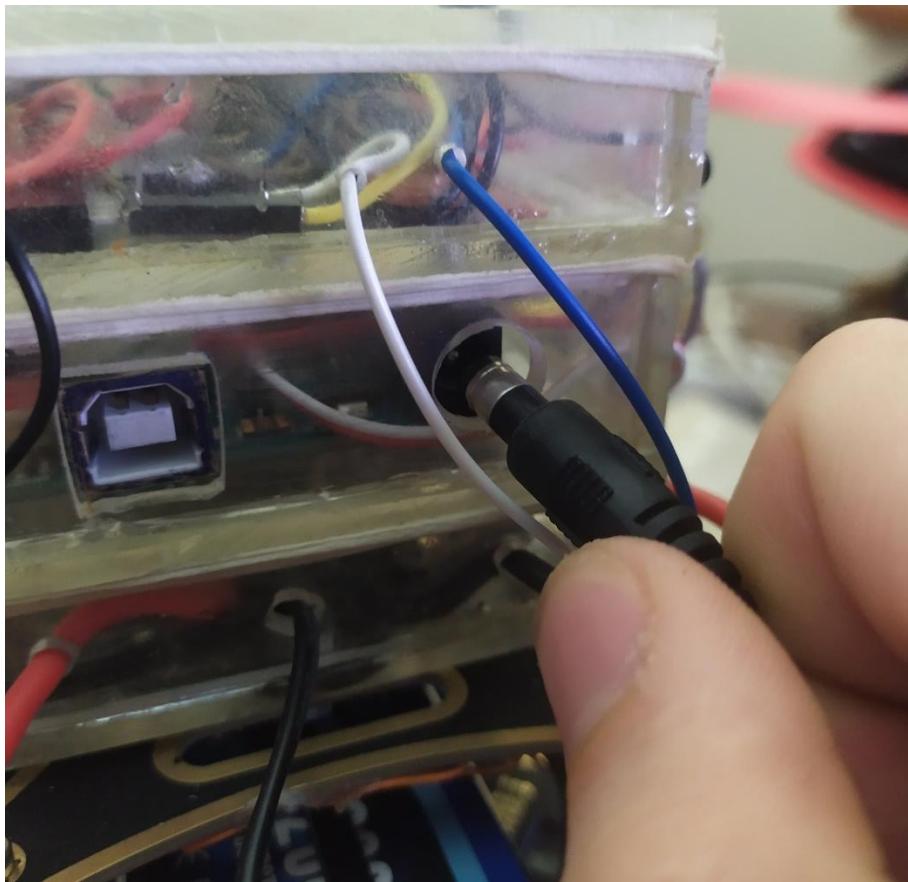


Ilustración 77: Conexión Jack placa Arduino

Cuando conectemos el Jack se encenderá la luz azul, que nos indica que el Dron está listo para conectarse (ilustración 78) en caso de que haya un error al iniciar el MPU o tenga poca batería las luces cambiaran al color rojo (ilustración 79), con lo cual debemos desconectar la batería y volver a conectar para realizar un reinicio completo del sistema.



Ilustración 79: Error al iniciar, debido a batería baja o error en el MPU o modo Aterrizaje



Ilustración 78: Esperando conexión del Bluetooth o modo Despegue

Ahora nos abrimos la aplicación ArduinoRC y nos mostrara la siguiente pantalla (ilustración 80).

The image consists of two side-by-side screenshots of the ArduinoRC mobile application. The left screenshot shows the main screen with a teal header bar containing the app logo and a 'Read, write, listen - experience the whole package of language learning with Busuu Premium' advertisement. Below the header is a 'Current UUID' field showing the value '00001101-0000-1000-8000-00805F9B34FB'. At the bottom are two buttons: 'Change UUID' and 'Proceed'. The right screenshot shows a modal dialog titled 'select a device to connect'. This dialog lists several paired devices with their MAC addresses: BS-41 (BF:E8:BC:CE:C9:CC), Energy Tower8 (30:23:1C:46:7C:62), OBDII (00:1D:A5:68:98:8B), Wireless Controller (30:0E:D5:8A:A7:D7), I9-TWS (41:42:00:12:11:10), DronJD (00:15:83:35:59:A5), Parrot MKi9000 v2.20 (00:26:7E:2B:61:ED), and BT-163 (79:23:65:67:54:69). At the bottom of the modal are 'Scan for devices', 'Change UUID', and 'Proceed' buttons.

Ilustración 80: Inicio app ArduinoRC

Pulsamos en Proceed para poder conectarnos al Dron, nos aparecerá la siguiente pantalla (ilustración 81)

Nos conectamos al DronJD y nos aparecerá la siguiente pantalla (ilustración 82)

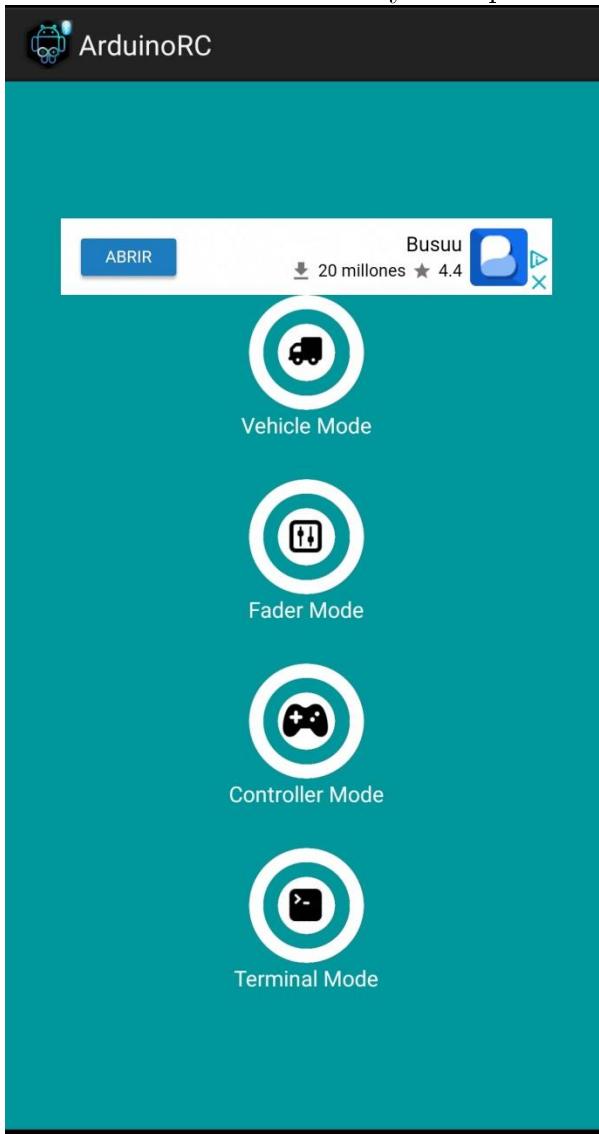


Ilustración 82: Menú principal

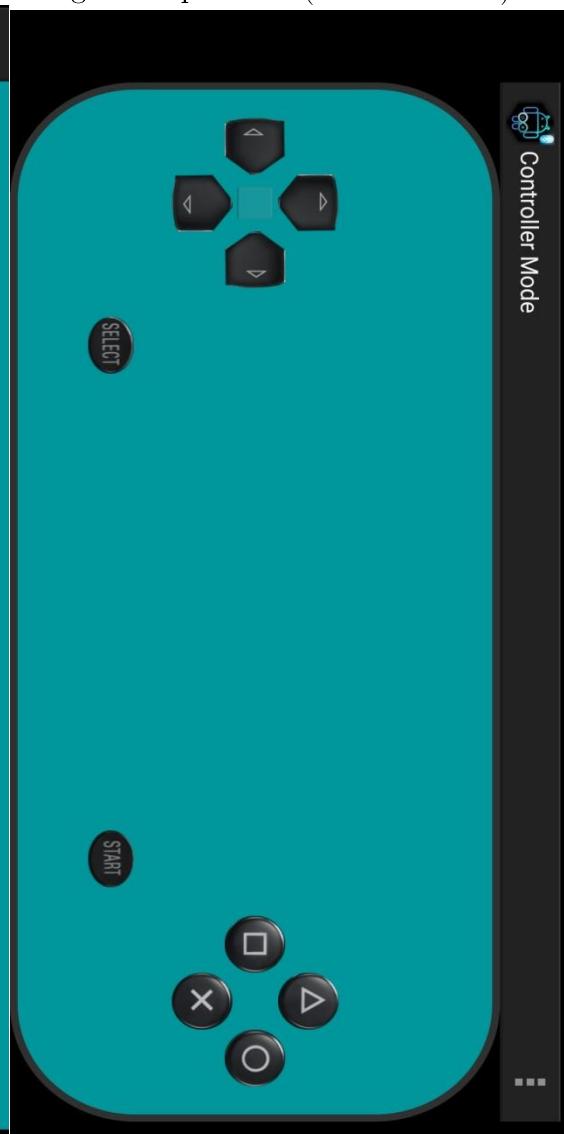


Ilustración 83: Control principal

Para el manejo del Dron pulsamos en Controller Mode, y nos aparecerán los mandos (ilustración 83)

Pulsamos en ajuste y configuramos los botones para el manejo del Dron, estás son las opciones disponibles:

- A esta opción ira incrementa gradualmente el Throttle
- B esta opción ira baja gradualmente el Throttle
- P esta opción avanza el Dron hacia delante gradualmente
- H esta opción retrocede el Dron gradualmente
- R esta opción inclina el Dron hacia la derecha
- L esta opción inclina el Dron hacia la izquierda
- W esta opción gira el Dron hacia la derecha sin inclinarlo
- Y esta opción gira el Dron hacia la izquierda sin inclinarlo
- q esta opción aumenta el Throttle al máximo
- a esta opción reduce el Throttle al mínimo

- z esta opción activa el modo Despegue
- x esta opción activa el modo Aterrizaje
- c esta opción activa el modo Acrobático
- v esta opción activa el modo Estable
- p esta opción activa el modo Seguimiento
-

Los diferentes de Modos de vuelos son:

- **Modo Estable:** Por defecto si no elegimos ninguna opción después de conectar el Dron y bajar al mínimo el Throttle entrara en este modo, se apagara las luces y cambiara al de la (ilustración 84).



Ilustración 84: Modo Estable

En este modo el Dron se mantiene de forma estable aunque las condiciones atmosféricas sean adversas.

- **Modo Despegue:** El Dron cambiara la luz al de la figura (ilustración 78)
Este modo es automático realiza un despegue hasta una altura de un metro y cuando llega a esa altura cambia al modo Estable (ilustración 84), donde podremos controlar el Dron.
- **Modo Aterrizaje:** El Dron cambiara la luz al de la figura (ilustración 79)
Este modo es automático realiza un aterrizaje recto hasta el suelo, de forma progresiva cuando llega al suelo, el Dron cambiara al modo Estable (ilustración 84), donde podremos controlar el Dron.
- **Modo Acrobático:** El Dron cambiara la luz al de la figura (ilustración 85), este modo es manual y la diferencia al modo Estable, es que si por alguna razón se desvía el Dron no estabilizara, pero si contrarrestara con lo cual debemos intentar estabilizarlo nosotros con el mando.



Ilustración 85: Modo Acrobático

- **Modo Seguimiento:** El modo seguimiento o automático el Dron encenderá las luces blanca (ilustración 86)



Ilustración 86: Modo Automático

Si el Dron inicia desde el suelo primero sube hasta una altura de 1m como con el modo Despegue seguidamente la cámara interpretara la línea o circuito (ilustración 27) que se encuentre debajo y seguirá el hasta el final de la línea, cuando llegue al final de la línea, el Dron aterriza y cambiara al modo Estático.



UNIVERSIDAD
DE MÁLAGA | **uma.es**

E.T.S. DE INGENIERÍA
INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Pasteur, 35
Campus de Teatinos
29071 Málaga