Universidad Rey Juan Carlos

"CÉFIRO"

DRONE CUADRICÓPTERO BASADO EN ARDUINO

CURSO 2019-2020

Drone de fotografía basado en Arduino

Drone cuadricóptero ensamblado sobre la plataforma Ardupilot, y equipado con un gimbal de 2 ejes capaz de hacer fotografías y videos aéreos.

Diseño de Sistemas Empotrados – Grupo 9

Adrián González Díaz-Tendero

UNIVERSIDAD REY JUAN CARLOS

Universidad
Rey Juan Carlos

UNIVERSIDAD REY JUAN CARLOS

Tabla de contenido

Tabla de Ilustraciones

RESUMEN

El principal objetivo de este proyecto es el desarrollo de un dron cuadricóptero, basado en Arduino con la posibilidad de hacer fotografía aérea, y que permita colocar diferentes tamaños de cargas, hasta un máximo de medio kilo. Además, se ha de poder utilizar en un gran rango de tareas.

El software utilizado es de tipo abierto para tener flexibilidad y costes bajos. Al igual que el hadware.

En este trabajo se ha estudiado: el nivel de la técnica de los UAV´s (drones) y sus clasificaciones; el diferente hardware libre que existe actualmente, y que se podría utilizar para volar de manera efectiva y segura, así como los distintos softwares de tipo abierto y las funciones que realizan. Basándose en este estudio y en los objetivos, se ha realizado el dimensionamiento general del dron. Justificando debidamente la selección de los componentes elegidos.

Se ha utilizado el programa eCalc para realizar los cálculos necesarios para el dimensionamiento de la propulsión del motor, la batería y las hélices.

La estructura mecánica es de tipo H, y en un principio se iba a desarrollar 3D con el programa SolidWorks. Pero finalmente por reducir costes y asegurar una rigidez y resistencia altas, se ha optado por recurrir al mercado asiático y adquirir un frame (cuerpo del dron) ya impreso.

Finalmente, se ha realizado un test de los componentes que asegurará la estabilidad del equipo. Y también, se han realizado pruebas de los controladores de Ardupilot ajustando los valores de control con un procedimiento empírico hasta conseguir el funcionamiento deseado.

A lo largo de esta memoria se recogerá todo el proceso, las dificultades e imprevistos que han ido surgiendo, soluciones implementadas y el resultado.

1. INTRODUCCIÓN

Los vehículos aéreos no tripulados (VANT) o más coloquiales drones, han ido ganando importancia en los últimos años. Sin embargo, este sistema tiene más antigüedad de lo que se cree en realidad. Los drones tienen sus orígenes en el sector militar; pero al igual que otras tecnologías, en los últimos años, han ganado cada vez más importancia en el sector comercial y en el mercado civil. En esta sección, se define y se explica brevemente la historia y procedencia de los vehículos aéreos no tripulados. Además, se enumeran los posibles campos de aplicación y el modo de proceder para este trabajo.

1.1. Definición

Los vehículos aéreos no tripulados, conocidos en inglés como Unmanned Aerial Vehicles (UAV) y comúnmente conocidos como drones, reciben su nombre del inglés (drone)/ del alemán (Drohne) y hace referencia a la abeja macho o zángano.

En sus inicios, el sector militar desarrolló un vehículo no armado cuyo único objetivo era para realizar ejercicios de tiros. Así mismo, el zángano tiene también un solo trabajo, aparearse con la abeja madre y morirse después. Así, se puede entender la conexión de los nombres.

Aunque UAV es la palabra más oficial y dron se puede considerar como una palabra más coloquial. Como su nombre indica, vehículo aéreo no tripulado, significa que no hay piloto en el vehículo. Además, para que se pueda decir que es un UAV requiere la cualidad de reutilización, que significa que puede volver a la tierra; por eso, por ejemplo, un misil no es un UAV.

La conducción puede tener diferentes grados de autonomía, la más simple sería con un piloto que está en el suelo y mantiene un control vía telecontrol, hasta un comportamiento del UAV completamente autónomo.

1.2. Historia

Uno de los primeros conocimientos de la existencia de UAV y su uso datan del año 1849. En ese tiempo, Austria bombardeaba Venecia con globos sin tripulación. El desarrollo de los UAV no hubiera sido posible sin grandes avances tecnológicos como la radio. El telégrafo fue utilizado por primera vez en 1858, para conectar Europa y los Estados Unidos; poco después, en 1898 Nikola Tesla mostró en Nueva York un sistema de control para un barco a través de señales de radio. En 1931 el “Royal Air Force” utilizó aviones dirigidos por radio para hacer ejercicios de tiro en los entrenamientos de los pilotos de combate. En los años 60, los Estados Unidos emplearon los UAV sobre todo para efectos informativos en Cuba, Vietnam, China y Corea del Norte. Entre los años 1955 y 1975 en la guerra de Vietnam el Air Force utilizó los UAV para localizar misiles tierra-aire. Más tarde, en Europa en los años 1998 y 1999 durante la guerra del Kosovo, los UAV fueron utilizados para poner a disposición marcas de guiados de láser para aviones de combate F16. En el año 1995, los Estados Unidos introdujeron el UAV “Predator” que tenía además de la capacidad de la supervisión y permitía utilizar misiles.

En el año 1998, tuvo lugar el primer vuelo del UAV "Global Hawk", el cual es hasta el momento, el UAV más grande fabricado en serie.

El sector militar es el mayor consumo en el mercado de los UAV; aunque el sector privado está creciendo cada vez más. Así por ejemplo en 2014, DHL empezó a hacer pruebas de envío de paquetes con un UAV entre la ciudad de Norden y la isla Juist en Alemania. Dos años después, Amazon realizó la primera entrega de un paquete con un UAV en Inglaterra, Figura 1.

En 2016, la empresa Ehang presentó el primer "UAV" para transportar a un pasajero en el "Consumer Electronics Show" en Las Vegas. El modelo de UAV empleado fue un octocóptero de tipo coaxial, que puede transportar un peso de hasta 100 kg.

No tiene sentido, hablar de un UAV que puede transportar una persona, ya que se definen como vehículo no tripulado; pero debe mencionarse porque al final, el funcionamiento es semejante al de un cuadricóptero. El funcionamiento es casi autónomo y, según la empresa, puede volar cualquier persona. El pasajero solo tiene que poner el destino y el vehículo se encargará del resto.

Figura 2 – Prototipo de Amazon

Figura 1 - Ehang

1.3. Campos de aplicaciones

Como ya se ha mencionado en el apartado anterior, los drones han sido empleados en gran medida en el campo militar. Por otro lado, también tienen gran aplicación en el sector civil y comercial.

En la siguiente sección están clasificadas las posibles aplicaciones en siete áreas.

1. **Área de vigilancia policial**. Por ejemplo, un UAV podría sustituir un helicóptero en muchas situaciones. Un drone se puede emplear para vigilar fronteras y costas o en general también para la persecución. Adicionalmente, existe la posibilidad de emplear el UAV para monitorear el tráfico y las aglomeraciones.

2. **Área de apoyo logístico a los bomberos.** El empleo de UAV aquí puede hacer el trabajo de los bomberos más seguro y fácil y permitir además poder conseguir mejores resultados. Por ejemplo, se pueden emplear para la vigilancia o extinción de incendios. En catástrofes naturales, se podrían emplear para hacer una peritación de daños o directiva en general. Además, se podrían emplear para buscar y rescatar personas desaparecidas en la montaña o en lugares con difícil acceso.

3. Otro grupo es el **sector de la energía, la industria de aceite y gas y también de la red eléctrica**. Los UAV pueden ayudar en este grupo a mejorar la vigilancia y la seguridad y así asegurar el buen funcionamiento.

4. Otro campo es **la agricultura, la silvicultura y el sector pesquero**. El uso de UAV puede ser por ejemplo el control de plagas, la optimización de los recursos, el control medioambiental o para la siembra de plantas de cultivo.

5. Área de **observaciones terrestres y teledetección**. Este campo se centra en la vigilancia climática, la fotografía aérea, la cartografía y la vigilancia de eventos sísmicos, de incidentes graves y de contaminación.

6. **Sector de la comunicación y radio fusión**. En este caso, se puede utilizar los UAV para proporcionar una buena cobertura de comunicación.

7. **Área de transportes y de logística**, tal como se ha mencionado en el apartado de historia. Además, se pueden utilizar para transportar medicamentos en regiones de difícil acceso. Permitiendo así, ayudar a la población.

8. Una aplicación curiosa que debe ser mencionado aquí es, un cuadricoptero que se ha desarrollado en el Georgia Institue of Technolgy en Atlanta para tener condiciones de cero g. Como es difícil y muy caro hacer experimentos en condiciones con cero g se han hecho un cuadricóptero con hélices especiales para producir para 5 segundos un ambiente de cero g.

1.4. Objetivos

El objetivo de este trabajo es desarrollar un cuadricóptero para poder realizar fotografía aérea, y transportar cargas, con sensores adicionales y tener además una flexibilidad para montar dichas cargas.

La carga máxima son 0,5 kg y debe ser posible montar la carga en el centro, debajo del UAV, con el fin de no desequilibrar el drone en vuelo.

El objetivo de tiempo de volar son 4-5 min. Este tiempo podrá variar si se monta o desmonta el Gimbal de fotografía, o si el dron va cargado o no.

El UAV debe tener las capacidades de control remoto, vuelo automático, aterrizaje automático y la posibilidad de vuelo en primera persona (FPV).

La placa utilizada debería ser suficientemente potente para gestionar todos estos tipos de vuelos fluidamente.

El software tiene que ser de tipo abierto para tener la posibilidad de hacer cambios, ya que se pretende que sea un proyecto escalable. También debería ser posible conectar el dron con una estación de tierra vía radio o wifi.

Para reducir costes y tiempo de desarrollo se han de reutilizar el mayor número posible de componentes.

Se han de realizar las pruebas que permitan demostrar la idoneidad fundamental del sistema desarrollado. El proceso a seguir para alcanzar el objetivo consta de las siguientes fases:

- Búsqueda bibliográfica e investigación del nivel de la técnica
- Selección del soporte físico y soporte lógico
- Desarrollo del concepto
- Construcción e interpretación de los componentes
- Fabricación y montaje de los componentes
- Integración del soporte lógico
- Pruebas

1.5. Procedimiento

El concepto analítico de este trabajo está dividido en cuatro partes, estas son: la preparación, el desarrollo, las pruebas y la documentación.

- La preparación se divide en la familiarización con el tema, la investigación del nivel de la técnica y la selección del soporte físico, así como la selección del software de tipo abierto.

- El desarrollo está dividido en tres otras subcategorías, la electrónica, la mecánica y el software.

    - Para empezar, es necesario hacer un dimensionamiento completo del UAV, determinando así el tamaño, peso, número de motores etc. La parte electrónica incluye además la selección de los sistemas electrónicos.

    - La parte mecánica incluye el diseño del soporte físico, los cálculos e interpretaciones y finalmente la fabricación y el montaje del prototipo.

- La última parte del desarrollo se ciñe a la parte informática. En ella se ha de implementar el software en el microprocesador elegido.

- Una vez construido el prototipo se podrán realizar las pruebas de vuelo y las configuraciones de los parámetros de control.

- La documentación es un proceso continuo que empieza desde el primer día hasta el último día.

2. CAPTACIÓN DE INFORMACIÓN

En este capítulo se estudia el hardware que existe actualmente en el mercado, y muestra las placas más conocidas de los UAV y finalmente, los softwares de tipo abierto más potentes que existen.

2.1. Hardware /placa.

- **Arduino Nano o Mini**. Son varias las desventajas que se encuentran en el montaje de un UAV con Arduino, entre las que se pueden destacar la gestión de interrupciones.

  Para explicar las interrupciones de Arduino se propone un ejemplo muy simplificado: Diseñamos un sistema que tiene que atender una serie de actuadores (por ejemplo, un botón), tendremos que andar sondeándolos periódicamente para ver si se pulsó el botón. ¿Pero qué ocurre si se pulsa un botón mientras el programa anda ocupado en otra tarea y justo en ese momento no se está sondeando la entrada del actuador? En este caso el sistema no será capaz de detectar la pulsación. La solución en este caso es el uso de interrupciones. Cuando se dispara una interrupción, el procesador detiene la ejecución del programa que está corriendo, almacena el estado de los registros (incluido el contador de programa), y ejecuta un subprograma, llamado rutina de servicio de interrupción, que atiende la interrupción (en este caso la gestión de la pulsación del botón). Una vez finalizada la rutina, el microprocesador restaura los registros y continúa la ejecución del programa principal por el mismo lugar en el que fue interrumpido.

  Esta gestión de interrupciones haría inmanejable un equipo caro, que se encuentra levitando a varios metros del suelo, lo que supondría ante cualquier imprevisto un accidente fatal.

  Otra de las desventajas de Arduino, se pone de manifiesto cuando se tiene que ampliar la versión estándar de la placa del microcontrolador por medio de interfaces adicionales y de funciones de entrada y salida. Si bien el hardware estandarizado permite ampliarse por medio de lo que se conoce como shields, la adquisición de estos módulos adicionales, incrementa rápidamente los costes del proyecto, así como el

peso del UAV, lo que tampoco interesa para un equipo que hay que levantar del suelo con cuatro rotores.

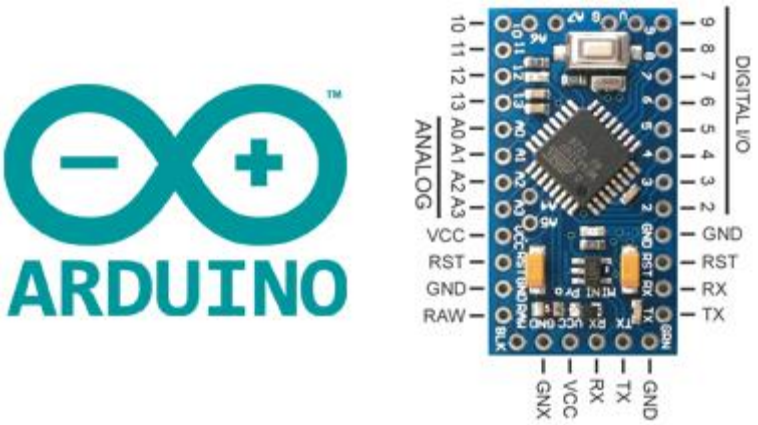


Figura 3 - Arduino Nano

- **Placas de vuelo de drones de carreras.** La principal característica de este tipo de FC es el tipo de chip que contienen. Este chip se caracteriza por la velocidad y capacidad para procesar gran cantidad de datos. El que sea más rápida en procesar datos, significa que los firmwares (programas de la placa de vuelo) podrán estar más optimizados para conseguir un vuelo más estable y preciso.

  Por el contrario, este tipo de controladoras no se adaptan a este proyecto, ya que dan poco juego a la hora de la programación, y no admiten ser ampliables más allá de un vuelo rápido y preciso, por lo que la opción de incorporar un gimbal para fotografía aérea o admitir cargas pesadas sería incompatible.

- **Controladoras de vuelo de drones de fotografía.** En este apartado podemos diferenciar varias alternativas en función del objetivo del UAV y del presupuesto:

  - **Naza** es la solución hardware que ofrece la empresa DJI, referente a día de hoy en UAVs comerciales, con su gama Phantom e Inspire. Esta plataforma ofrece calidad y muy buenas prestaciones RTF, es decir, viene lista para ser volada. Es la opción preferida de aquellos que no quieren ponerse a investigar el funcionamiento de la aeronave y que simplemente quieren salir y volar.

    Entre sus defectos destacan el alto precio en todas las versiones disponibles y las pocas posibilidades para desarrolladores. Entre sus ventajas, la posibilidad de empezar a volar tras sacarla de la caja y la facilidad de uso. Dispone de una IMU completa formada por giroscopio, acelerómetro, barómetro y magnetómetro.

Figura 4 - Controladora de Vuelo Naza de DJI

- **Ardupilot Mega (APM)** es una placa controladora de vuelo con una IMU de calidad basada en la muy conocida plataforma Arduino Mega.

  Este autopiloto puede controlar alas fijas, helicópteros multirotor así como helicópteros tradicionales. Soporta 8 canales de radiocontrol con 4 puertos series.

  ArduPilot Mega consta de una placa principal con el procesador Atmel ATmega2560 (8- bit) y la placa con la IMU que se coloca encima de esta.

  La controladora ArduPilot Mega fue diseñada en 2010 por Jordi Muñoz, cofundador de 3DRobotics. Esta placa ha sido de las más utilizadas durante años para la navegación autónoma de todo tipo de vehículos tanto aéreos como terrestres. Con los años ha sido superada por otras plataformas, hasta llegar a dejar de recibir soporte y actualizaciones. Esta placa fue la primera de muchas que se diseñaron para alojar el software ArduPilot

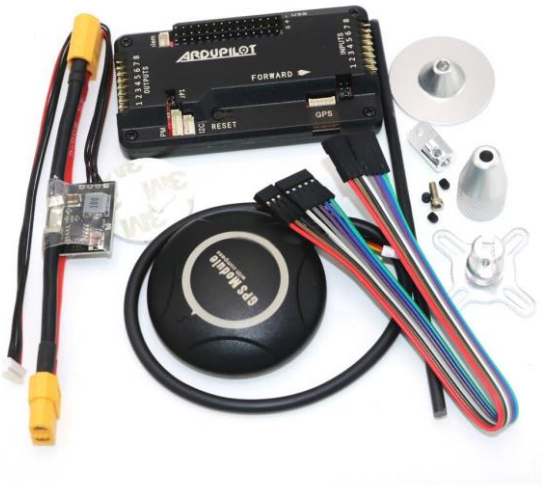


Figura 5 - Ardupilot Mega (APM)

- **PixHawk** es un proyecto independiente y de hardware abierto que intenta dar soluciones hardware para autopiloto de vehículos de alta calidad a las comunidades académicas, de hobby e industriales a un bajo costo y una alta disponibilidad, según se dice en su página oficial. Pixhawk se ha convertido en los últimos tiempos en la plataforma preferida para desarrolladores, por las grandes posibilidades que ofrece. Ha superado a su antecesora, APM. En su interior lleva también el software de Ardupilot.

  Cuenta con un procesador de 32 bits STM32F427 Cortex M4, y otro coprocesador de 32 bit STM32F103 con 256 Kb de RAM y 2 Mb Flash. Dispone de una IMU completa formada por giroscopio, acelerómetro, barómetro y magnetómetro, algunos de ellos por duplicado. Incorpora también 5 puertos UART, 2 puertos CAN, 1 puerto SPI y un puerto I2C. Además, dispone de numerosas entradas para los distintos tipos de receptores de radio, entre otros.

Figura 6 – Ardupilot versión Pixhawk

- **Otros.** Durante el proceso de investigación para el desarrollo de esta práctica se ha podido observar algunos artículos donde se plantea la posibilidad de utilizar placas de desarrollo como Raspberry Pi 3, que podrían ser utilizadas como controladoras de vuelo sin necesidad de un ordenador extra por poseer bluetooth o wifi incorporados y procesadores más potentes. Sin embargo, se han descartado, ya que se intenta utilizar hardware ya testeado para su uso en UAVs, darle una salida comercial y lo ya comentado respecto a peso, y posibilidades de ampliación del proyecto.

2.1.1. Plataforma escogida

Tras analizar las opciones que nos ofrece cada plataforma se ha optado por utilizar para este proyecto la placa PixHawk de Ardupilot, por su versatilidad a la hora de ser programada y configurada para diferentes tipos de proyectos con UAVs. Dado que se quiere personalizar la aeronave con una configuración específica, esta placa nos ofrece una amplia gama de formas de comunicación y de control.

Se ha descartado la opción de APM ya que Pixhawk es su "sucesora". También se han descartado las controladoras NAZA por su alto precio y por su imposibilidad de ser programada.

Al tratarse de un placa descendiente de Arduino, tan solo que con mejores características hardware, posee muchos puertos y protocolos en común entre los que podemos destacar:

- **IMU**

  Es una unidad de medición inercial o IMU (del inglés inertial measurement unit). Es un dispositivo electrónico que mide e informa acerca de la velocidad, orientación y fuerzas gravitacionales de un aparato, usando una combinación de acelerómetros y giróscopos.

- **UART**

  UART, son las siglas en inglés de Universal Asynchronous Receiver-Transmitter, en español: Transmisor-Receptor Asíncrono Universal, es el dispositivo que controla los puertos y dispositivos serie. Se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

- **I2C**

  Circuito inter-integrado (I^2C, del inglés Inter-Integrated Circuit) es un bus serie de datos con un protocolo síncrono. I2C usa solo 2 cables, uno para el reloj (SCL) y otro para el dato (SDA). Esto significa que el maestro y el esclavo envían datos por el mismo cable, el cual es controlado por el maestro, que crea la señal de reloj. I2C no utiliza selección de esclavo, sino direccionamiento.

  Las líneas se llaman:

  - SDA: datos
  - SCL: reloj
  - GND: tierra

- **SPI**

  El Bus SPI (del inglés Serial Peripheral Interface) es un estándar de comunicaciones, usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos. Es otro protocolo serie muy simple. Un maestro envía la señal de reloj, y tras cada pulso de reloj envía un bit al esclavo y recibe un bit de éste. Los nombres de las señales son por tanto SCK para el reloj, MOSI para el Maestro Out Esclavo In, y MISO para Maestro In Esclavo Out. Para controlar más de un esclavo es preciso utilizar SS (selección de esclavo).

2.2.Software

El software empleado hace referencia a los UAV del sector de investigación, de consumo y del sector de economía, pero no del sector militar. El mercado del software es enorme y en general se puede diferenciar entre software comercial y software abierto. El sector de software abierto tiene la posibilidad de añadir y cambiar el código; lo cual permite amoldarlo a las necesidades de cada uno. Además, el hecho de ser gratuito es también algo importante. A continuación, se explica uno de los proyectos más importantes de tipo software abierto que hay actualmente, las placas que lo soportan, y por qué lo hemos elegido.

Ardupilot es uno de los softwares más potentes que hay en la actualidad. Con Ardupilot es posible activar desde un UAV de carrera hasta un UAV de carga. El software aporta distintos tipos de vehículos. Hasta ahora están incluidos los cópteros con una hélice, tres, cuatro, seis y ocho. Además, hay vehículos con alas y combinaciones de cópteros con alas, VTOL. El software de Ardupilot dispone además de funciones tales como modo de seguir, posibilidad de vuelta a casa, despegue y aterrizaje automático. Hay tres posibilidades de controlar el Ardupilot. La más básica sería mediante control remoto. Otra posibilidad es vía “QGroundControl”, se trata de un programa que puede estar en un portátil o un móvil. La última solución consiste en una conexión de datos de Ardupilot con un microcomputador externo. El software Ardupilot trabaja en general con dos placas, Pixhawk y Snapdragon Flight. Se puede confirmar el éxito de este software ya que Intel con su placa Aero y Qualcomm con su placa Snapdragon Flight están utilizando ya el software Ardupilot.

2.3. Otros componentes

- Frame F450

Figura 7 – Frame F450

- Switch armado

Figura 8 – Switch armado

- Gimbal

Figura 9 – Gimbal con dos servos

- Receptor de 6 canales a 2.4Ghz FS-i6AB

Figura 10 – Receptor de 6 canales

- Emisora Eachine FlySky i6

Figura 11 – Emisora FlySky i6

- Cámara de acción tipo Go Pro

Figura 12 – Camara de acción tipo Go Pro

- Hélices de dos palas 10x45

Figura 13 – Hélices de 2 palas 10x45

- Antena GPS

Figura 14 – Módulo de brújula y GPS

- Batería tipo lipo 3s 4200 mah 11,1 V

Figura 15 – Bateria lipo de 3 celdas

- Motores

Figura 16 – 4 motores Ready To Sky

- Antenas de Telemetría a 433Mhz

Figura 17 – Antenas de telemetría

- Power module

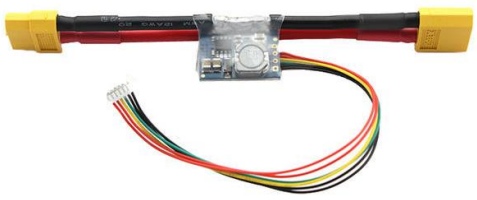


Figura 18 – Power Module

- 4 x Variadores (ESC) 30A para lipos de 2 a 4S

Figura 19 - Variadores

¿Qué es un ESC?

Estas siglas, provienen del Inglés, Electronic Speed controller, también llamados comúnmente en español variadores, son unos componentes electrónicos utilizados para regular la velocidad de giro de motores eléctricos sin escobillas (motores brushless). Existen diferentes protocolos de comunicación entre los variadores, motores brushless y controladoras de vuelo.

Para saber cuáles son los ESC que debemos elegir para los motores brushless hay que tener en cuenta 2 factores:

1. Amperaje o intensidad eléctrica soportada los variadores.
2. Consumo del motor o motores en su máxima potencia.

Los variadores traen siempre una indicación de la cantidad de amperios que soportan, esta cantidad puede ir desde un bajo amperaje tal como 6 amperios hasta cifras mucho más elevadas. Dicho amperaje determina la intensidad de corriente eléctrica que soportan dichos componentes electrónicos, es importante que los ESC que se utilicen admitan más amperaje del que los motores puedan llegar a utilizar en su máxima potencia. Básicamente se trata de no crear cuellos de botella en cuanto al paso de la electricidad se refiere.

Por ello se han escogido 4 variadores de 30A, acordes a la lipo seleccionada, de 2 a 4S (2 a 4 celdas).

- Sonar

- Cargador de Lipo

2.4. Herramientas

- Soldador de estaño
- Llaves
- Bridas
- Termo retráctiles
- Multímetro
- Pinzas
- Alicates
- velcro
- Tijeras pelacables

Figura 20 – Material utilizado

3. DESARROLLO / PUESTA EN MARCHA

3.1. Construcción del frame

En esta fase se procede al ensamblado del drone.

En primer lugar, se presentan los 4 ESCs con sus respectivos cableados, que son, un cable rojo (positivo), un cable negro (negativo) y un tercer cable que irá conectado a la controladora de vuelo, en una posición concreta, Más adelante se explicará en detalle.

Una vez presentados los 4 variadores, se procede a pre estañar la placa, en los pads que ya vienen preparados para ello. Se pre estañea también los cables de los ESCs, y posteriormente se ensamblan dándoles unos puntos de soldadura.

Para acabar este proceso, se añaden dos cables a los terminales + y – de la placa con un punto de soldadura. En el otro extremo del cable, se debería añadir un conecto xt60 macho, para conectarlo al Power Module, pero el distribuidor no lo incluyó.

Como alternativa se decidió doblar sobre si mismas las puntas, para engrosar su perímetro, y posteriormente pre estaneañarlas e introducir las puntas en el conector hembra del power module.

Más tarde, se presentaron el resto de los componentes, para establecer las ubicaciones finales de los mismos, y se fue procediendo a fijarlos con tiras de velcro, y bridas.

El siguiente paso fue atornillar los brazos al cuerpo del drone, y fijar los motores según su posición correspondiente.

Y finalmente se ensambló la antena GPS, a la parte superior del drone, y el gimbal en la base del frame.

3.2. Conexiones del circuito

Una vez colocados todos los componentes en su posición final, procedemos a su conexionado.

3.2.1. Alimentación de la Pixhawk

La Pixhawk generalmente se alimenta a través de su puerto de "potencia", como se muestra en la imagen a continuación. El puerto de alimentación alimenta simultáneamente a la Pixhawk y lee las mediciones analógicas de voltaje y corriente producidas por un módulo de alimentación opcional, que hemos instalado y del que posteriormente se explicará en detalle su funcionamiento.

Figura 21 - Alimentación de la Pixhawk

3.2.2. Conectar entradas de control remoto

Pixhawk es compatible con:

- Receptores de control remoto PPM (R / C)
- Receptores Futaba S.Bus
- Spektrum DSM y receptores DSM2
- Receptores de satélite Spektrum DSM-X
- Receptores IBUS
- Receptores MULTIPLEX SRXL versión 1 y versión 2.

Para los receptores tradicionales de un solo cable por canal (PWM), se puede usar un codificador PPM para convertir las salidas del receptor a PPM-SUM, que es el método que se ha utilizado en este proyecto.

3.2.3. Buzzer y Switch

El zumbador y el botón del interruptor de seguridad son obligatorios para Pixhawk. Hay que conectar los a los puertos BUZZER y SWITCH los conectores de los mismos, como se muestra en la figura siguiente.

Como buena práctica, se recomienda en las guías de Ardupilot, montar el zumbador al menos a 5 cm del controlador de vuelo, o el ruido puede alterar los acelerómetros.

Figura 22 – Conexiones del Buzzer y Switch de armado

3.2.4. GPS + Brújula

Los puertos GPS están conectados con el cable DF13 de seis posiciones, y el puerto MAG está conectado al puerto I2C con el cable DF13 de cuatro posiciones.

Más adelante se muestra información adicional de configuración y montaje.

3.2.5. Conexión de motores

En este punto se explica cómo conectar los ESC, motores y hélices a la controladora de vuelo Pixhawk.

Hay que conectar los cables de alimentación (+), tierra (-) y señal (es) para cada ESC a los pines de salida principales de la controladora de vuelo por número de motor.

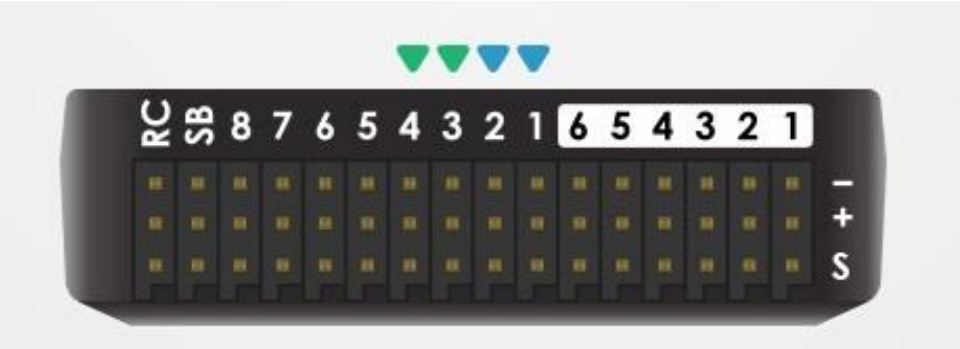
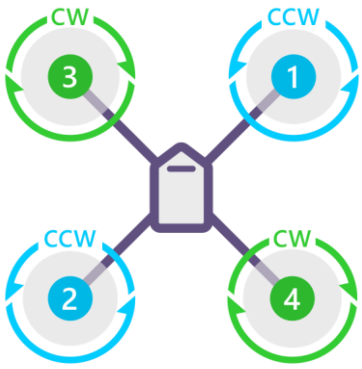


Figura 23 – Conexiones de motores a la controlado

3.2.5.1. Diagramas de orden del motor

Los siguientes diagramas muestran el orden del motor para el tipo de de frame escogido, en este caso “quad X”. Los números indican qué pin de salida del piloto automático debe conectarse a cada motor / hélice. La dirección de la hélice se muestra en verde (en sentido horario, CW) o azul (en sentido antihorario, CCW)

Figura 24 – Sentido de giro de las hélices



QUAD X

Figura 25 – Tipo de Frame escogido y sentido de giro de las hélices

3.2.5.2. Reconocimiento de hélices

Los diagramas anteriores muestran dos tipos de hélices: en sentido horario (llamadas empujadores) y en sentido antihorario (llamadas extractores). La forma más sencilla para reconocer el tipo de hélice correcto, es por su forma como se muestra a continuación. El borde más grueso es el borde de ataque que se mueve en la dirección de rotación. El borde de salida es más festoneado y generalmente más delgado.

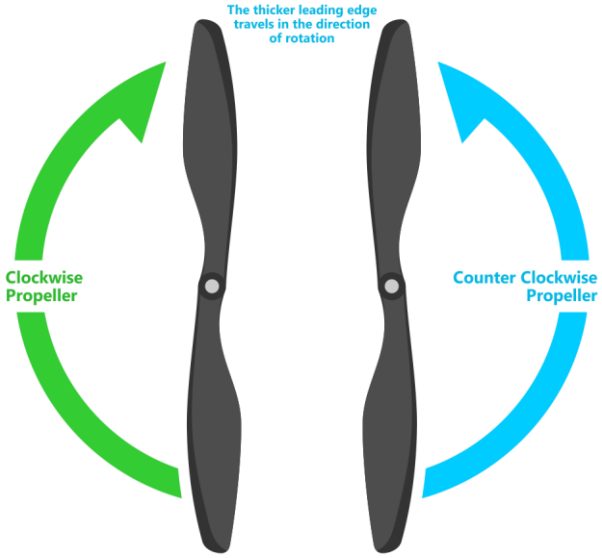


Figura 26 – Reconocimiento de las hélices

3.2.6. Conexión del Gimbal

El gimbal escogido, es el Tarot T-2D que es popular estabilizador de camaras de acción, sin escobillas, de 2 ejes de bajo costo.

Si bien este hardware real se retiró y se suspendió el soporte del fabricante, aún existen vendedores que suministran clónicos de este gimbal.

Figura 27 – Gimbal de 2 ejes escogido

- Los cables de alimentación rojo y negro de la controladora del gimbal, deben conectarse directamente a una batería 2S o 3S.
- El pin "T" debe estar conectado al pin de señal AUX1 de Pixhawk
- Uno de los dos pines "-" debe estar conectado al pin de tierra AUX1 de Pixhawk, como se muestra en la siguiente figura.

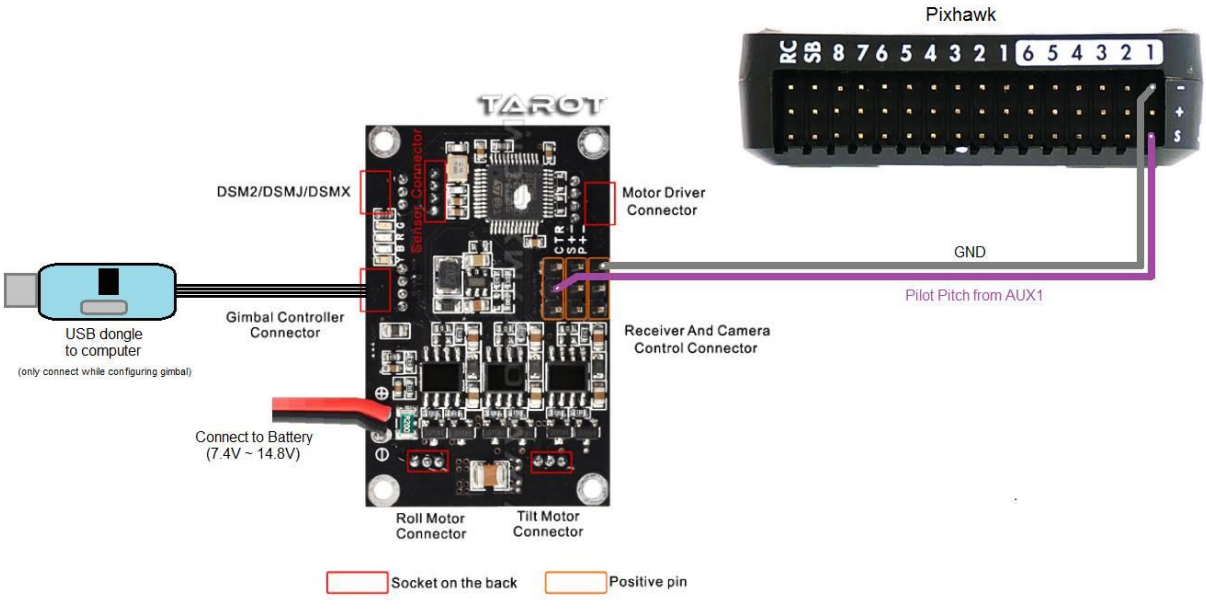


Figura 28 - Conexiones Gimbal a Pixhawk

3.2.7. Esquema de conexiones genérico

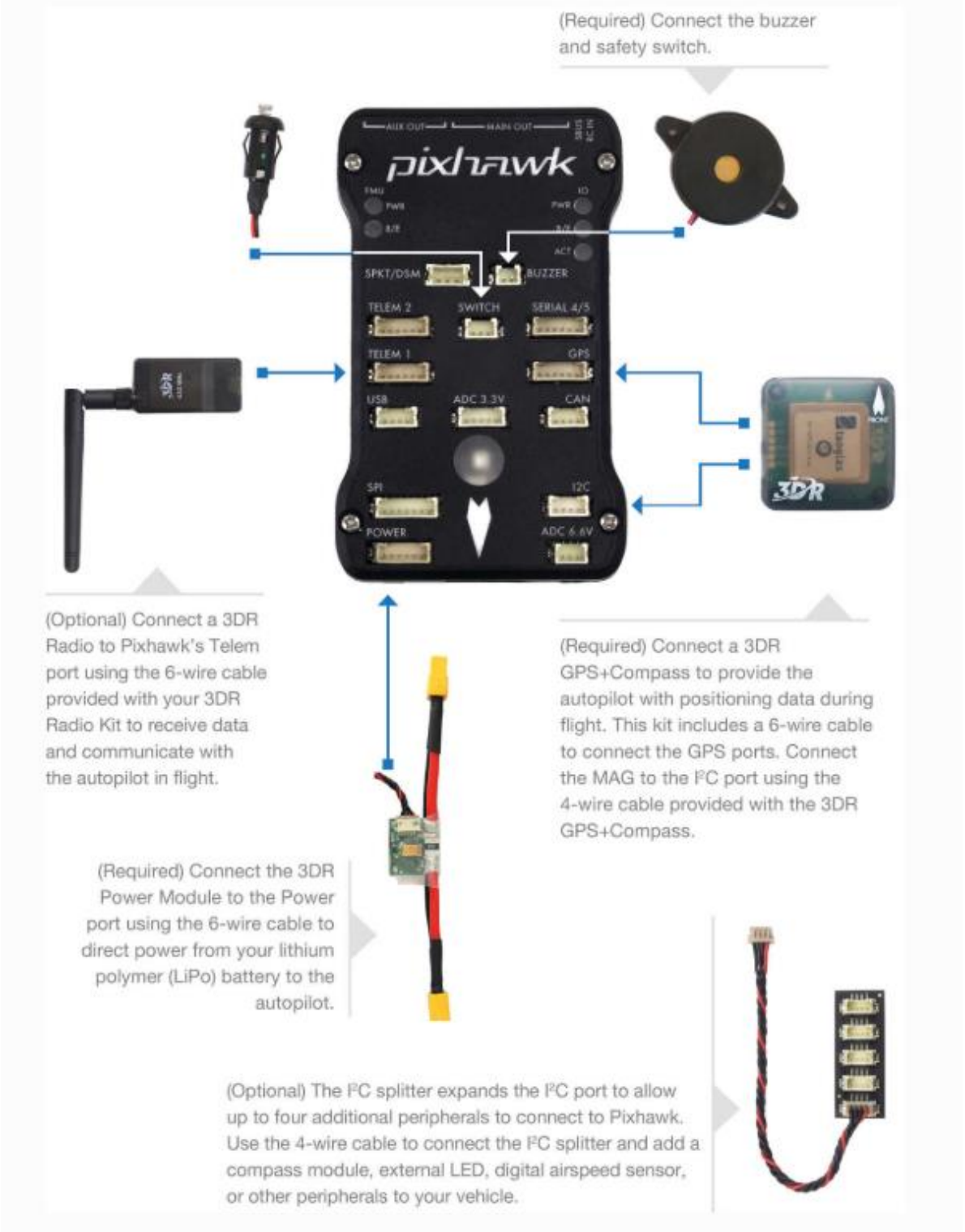


Figura 29 – Esquema de conexiones genérico

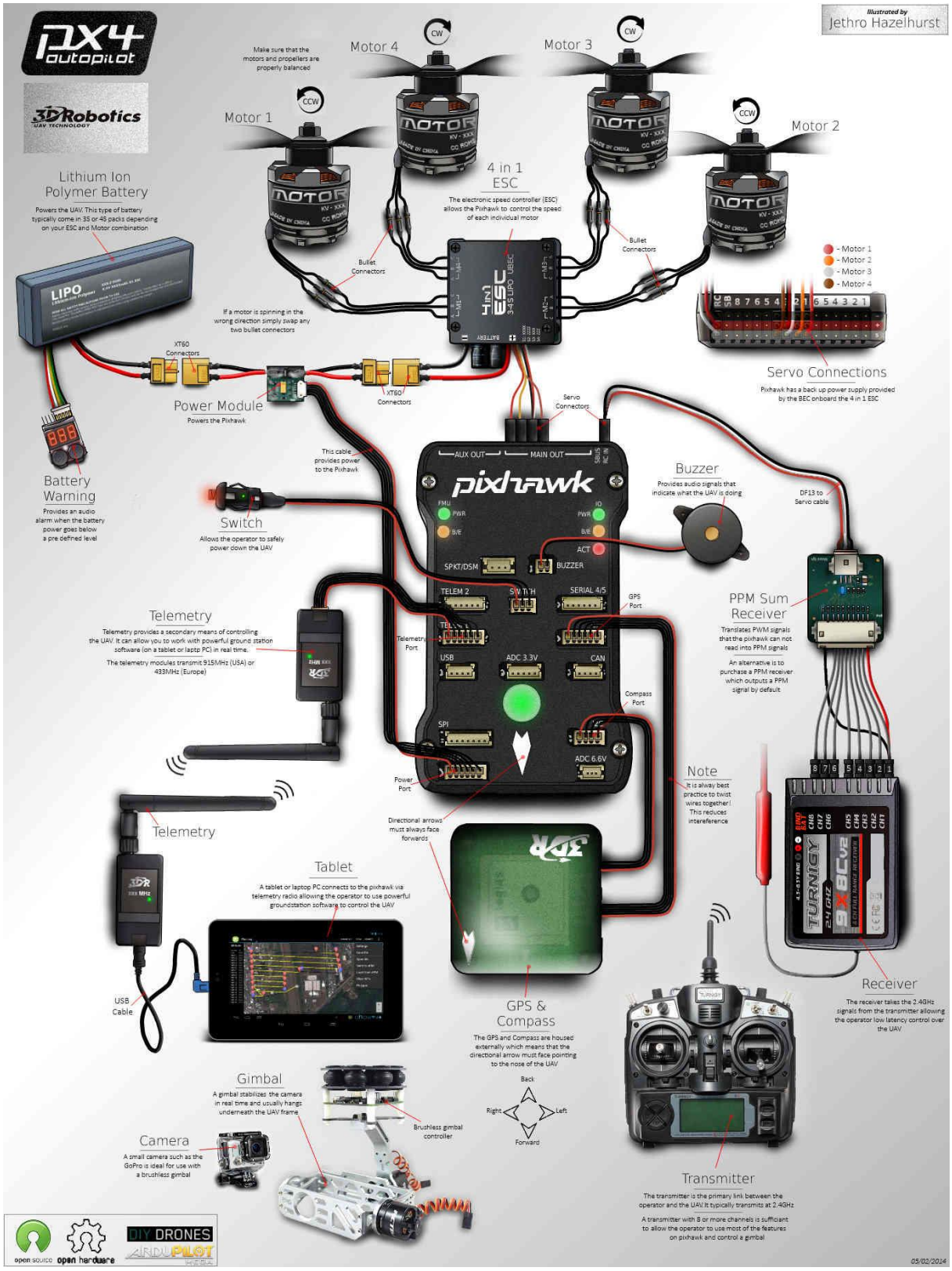
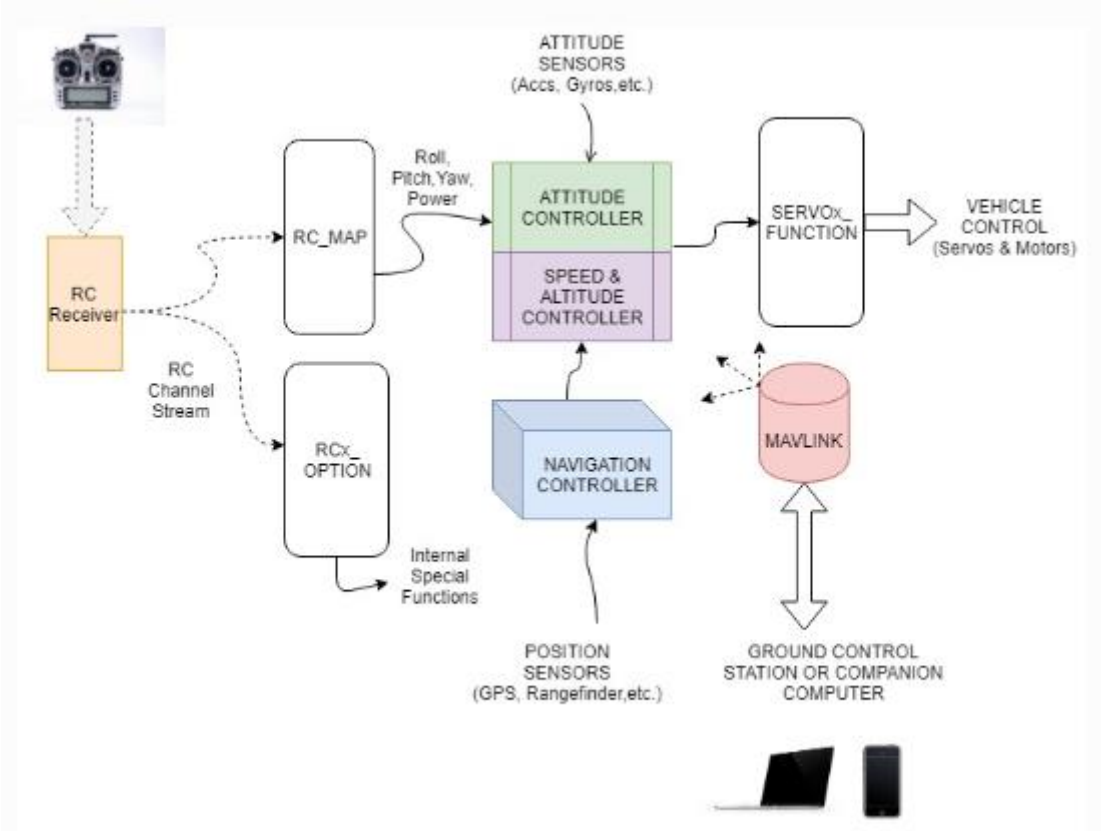


Figura 30 – Esquema de conexiones en detalle

4. CARGA DE CÓDIGO Y CONFIGURACIONES

Esta es una descripción muy básica de las funciones del firmware ArduPilot que se ejecuta en una controladora de vuelo Pixhawk. A continuación, se muestra un diagrama de bloques simple de la operación funcional básica.



El objetivo básico del software es proporcionar el control del vehículo, ya sea de forma autónoma, o mediante la entrada del piloto a través del transmisor de control de radio o la estación de control de tierra, o mediante una computadora complementaria.

4.1. Código

Cómo si de una placa Arduino se tratara, y como el propio software hace referencia, Ardupilot (firmware) que se ejecuta en nuestra placa de vuelo. Es el encargado de gestionar toda la información que recibe del exterior (sensores físicos y radio) y actuar en consecuencia sobre los actuadores de la aeronave.

Actualmente se puede descargar desde su repositorio de GitHub desde el cuál podemos descargar su código e incluso contribuir a su mejora.

La idea inicial era aportar código original a este proyecto, haciendo utilidad de un sonar para detectar el cambio de alturas que puedan propiciar montículos, u obstáculos en el aterrizaje, proporcionado en la asignatura de Diseño de Sistemas Empotrados. Pero por incompatibilidad de dicho sonar, y por el retraso en la llegada de los componentes, no se pudo implementar.

Pero este proyecto no muere aquí, se realizará un pedido de sonares analógicos, y no solo se instalarán en la zona inferior del dron, para facilitar su aterrizaje, e irlo acomodando al gusto del piloto a través del software, sino que también se implementarán sonares analógicos para la detección de objetos, tanto frontal, como lateral, como posterior, con el objetivo de crear un UAV que sea complicado chocarle.

Dejando a un lado las ideas originales frustradas, y ante la falta de tiempo para la entrega del proyecto, se optó por coger el código del repositorio de GitHub, revisarlo, y posteriormente testearlo en nuestra aeronave por si fuesen necesarios ajustes en el código para optimizar el vuelo.

Para cargar el código en nuestra placa, en lugar de abrir la aplicación de Arduino, teníamos que abrir la aplicación de Mission planner, conectarnos a nuestra placa por el puerto serie correspondiente, y posteriormente cargar el código obtenido de GitHub, la librería que se ha decidido cargar es Copter.h y un conjunto de modos de vuelos ya predefinidos. Todo este código se adjunta en el Anexo.

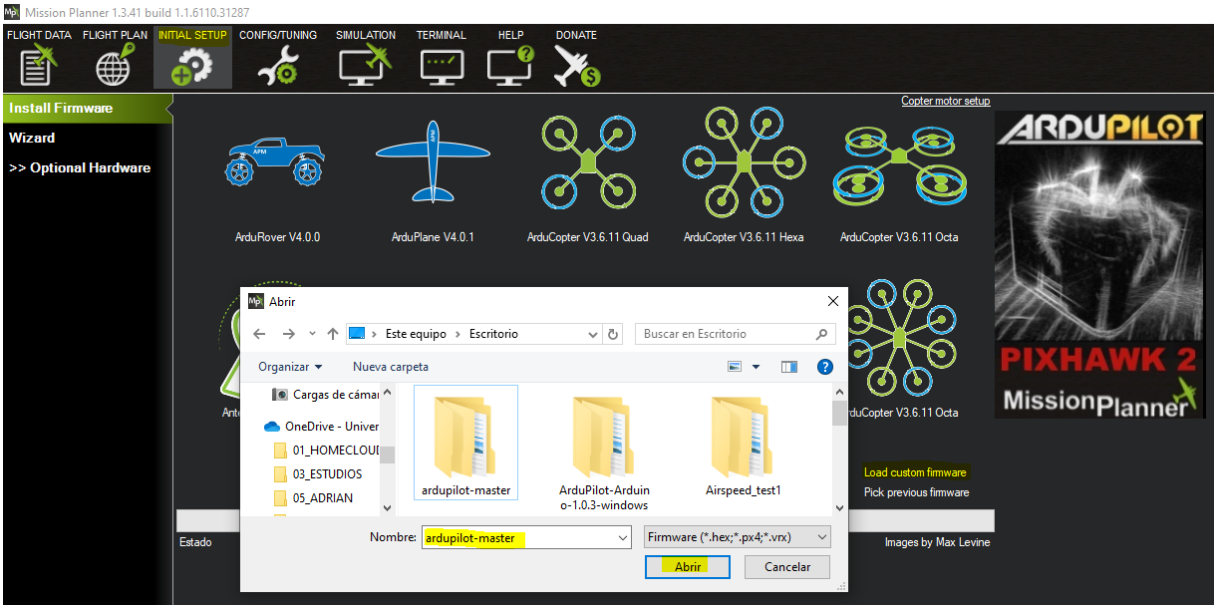


Figura 31 – Carga de Código al Arducopter a través del puerto serie usando MissionPlanner

4.2. Primera configuración

Para realizar la primera configuración en la controladora de vuelo, hay que conectarse a ella, y se puede realizar de dos formas. O conectando el cable usb y seleccionando CONNECT en el botón superior derecho de Mission Planner, que nos conectaríamos por el puerto serie, o por telemetría.

La primera opción para configurar es señalarle al software, el tipo de frame dónde se ha instalado la placa de vuelo.

- Se selecciona QUAD X

Los pasos a seguir serían los siguientes:

1. Abrir Mission Planner,
2. Seleccionar Configuración inicial,
3. Hardware obligatorio
4. Tipo de marco

Este parámetro en el código del Ardupilot corresponde con el parámetro FRAME_CLASS y FRAME_TYPE .

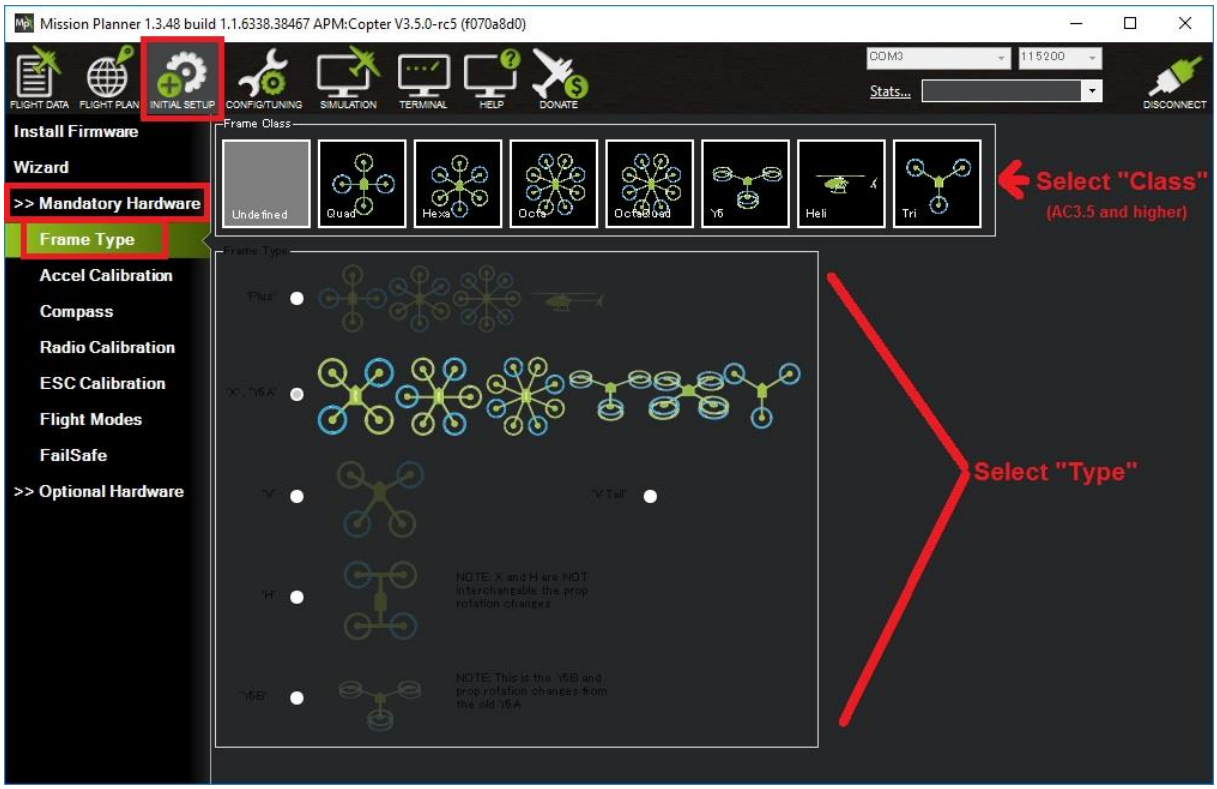


Figura 32 – Selección del tipo de frame

4.3. Calibración del acelerómetro

Para la calibración del acelerómetro hay que asegurarse que la flecha de la controladora de vuelo, ha quedado mirando hacia adelante, en caso contrario habría que modificar el parámetro AHRS_ORIENTATION en Ardupilot.

1. En la Configuración inicial del Hardware obligatorio, se selecciona Calibración de acelerómetro en el menú de lateral izquierdo.

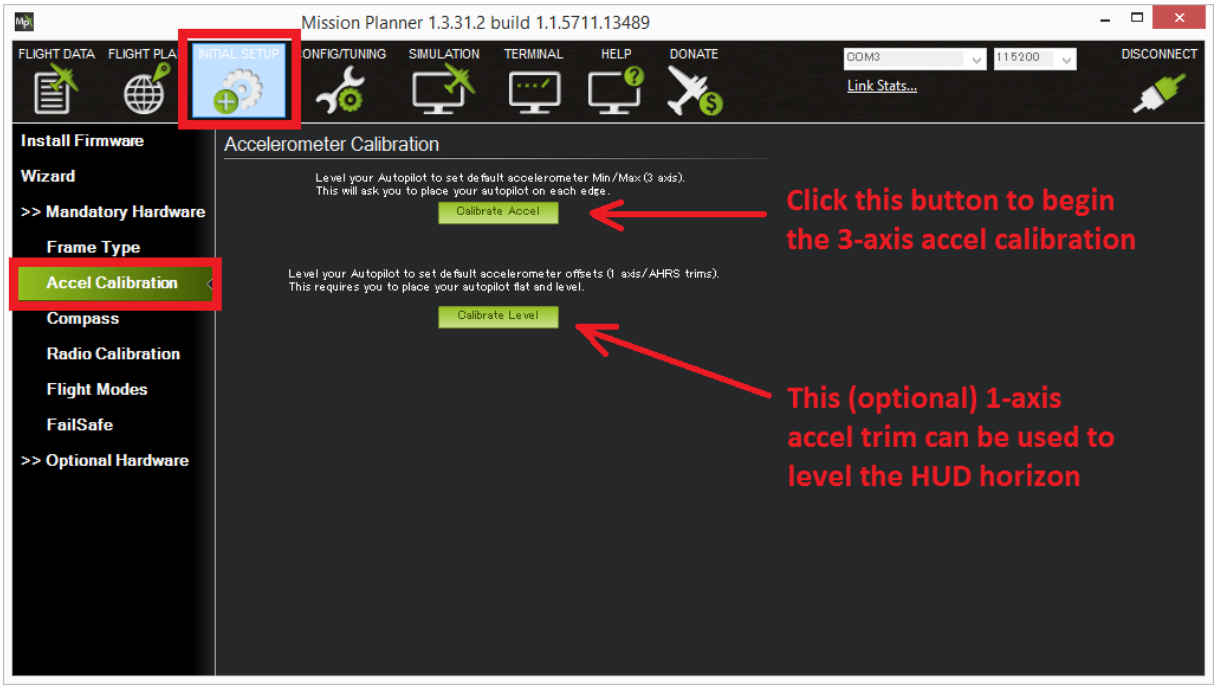


Figura 33 – Calibración del acelerómetro

2. Se hace clic en Calibrate Accel para comenzar la calibración.

   Mission Planner solicitará que se coloque el vehículo en cada posición de calibración. Se irá presionando cualquier tecla para indicar que el UAV está en posición y luego continuar con la siguiente orientación.

   Las posiciones de calibración son: nivelado, lado derecho, lado izquierdo, nariz hacia abajo, nariz hacia arriba y hacia atrás.

Figura 34 – Posiciones calibración acelerómetro

- Es importante que el vehículo se mantenga quieto inmediatamente después de presionar la tecla en cada paso.
- En ocasiones, puede ser recomendable calibrar la placa controladora antes de su montaje, si el tamaño del UAV dificulta su calibración. En el caso de este drone, no ha sido necesario.
- La posición nivelada es la más importante para una buena configuración, ya que esta será la posición en la cual la controladora se considera nivelada mientras vuela.
- Cuando se haya completado el proceso de calibración, Mission Planner mostrará "Calibración exitosa", como se muestra a continuación.

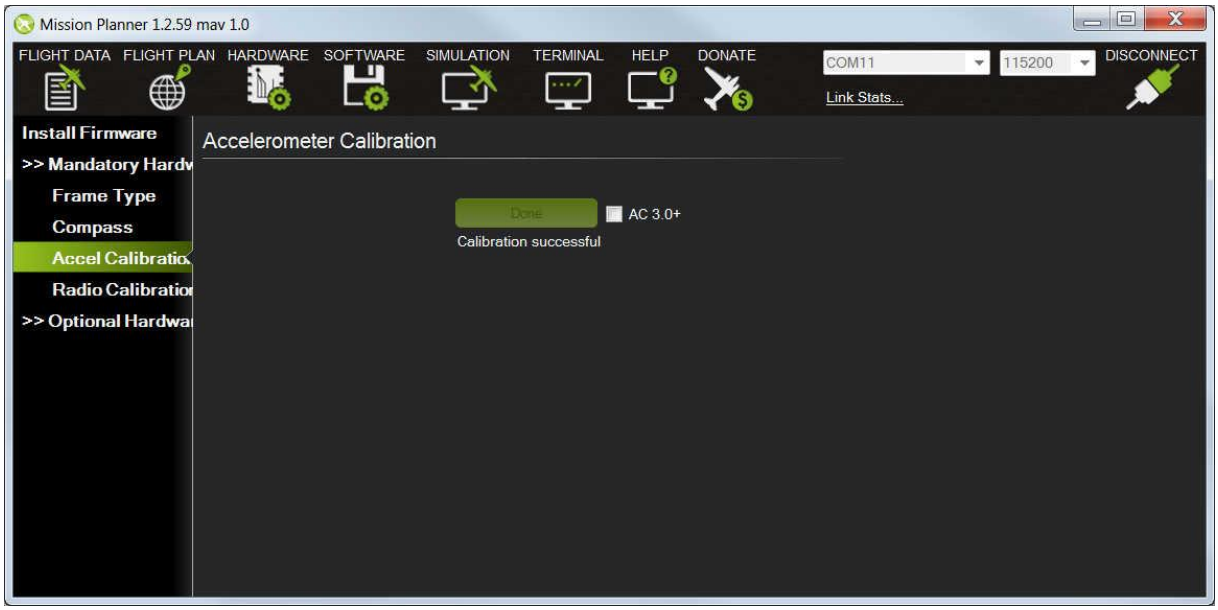


Figura 35 – Calibración exitosa

4.4. Calibración de la brújula

Como buenas prácticas, el manual de Ardupilot recomienda no calibrar las brújulas cerca de ningún objeto que produzca campos magnéticos, como pueden ser computadoras, teléfonos, escritorios metálicos, fuentes de alimentación, etc, o se producirá una calibración incorrecta.

- En Configuración inicial, Hardware obligatorio, se selecciona Brújula.

- Se selecciona la configuración del UAV, y automáticamente Mission Planner introduce la información de configuración más adecuada para la controladora

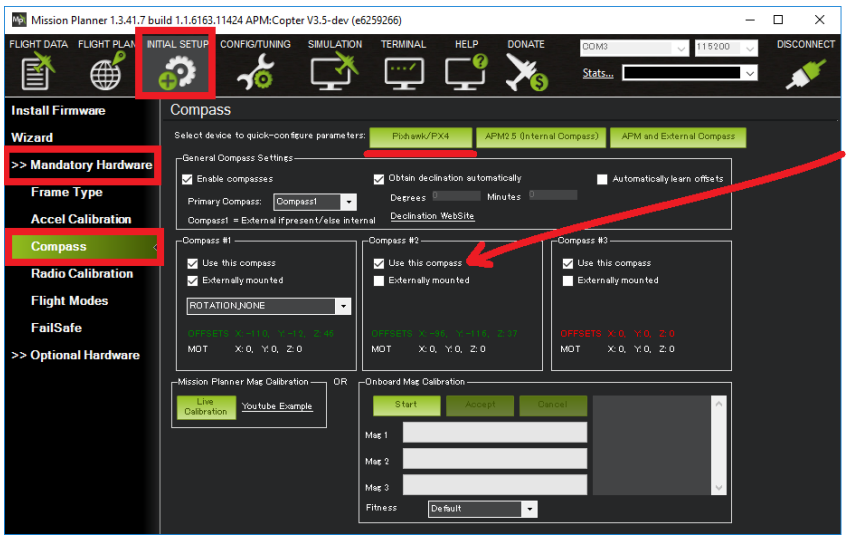


Figura 36 – Calibración de las brújulas

Como anteriormente se ha explicado, la controladora de vuelo Pixhawk, poseé una brújula interna, pero por las vibraciones a las que está expuesta la Pixhawk, se suele instalar un módulo GPS y Brújula externo.

Por lo tanto, en este apartado de configuración, aparecerán dos brújulas / dispositivos GPS, Compass 1 y Compass 2. Conviene seleccionar como Brújula principal el dispositivo externo.

Después de realizar este ajuste, se selecciona calibración en vivo, y debería aparecer una ventana emergente que muestra el estado de la calibración en vivo.

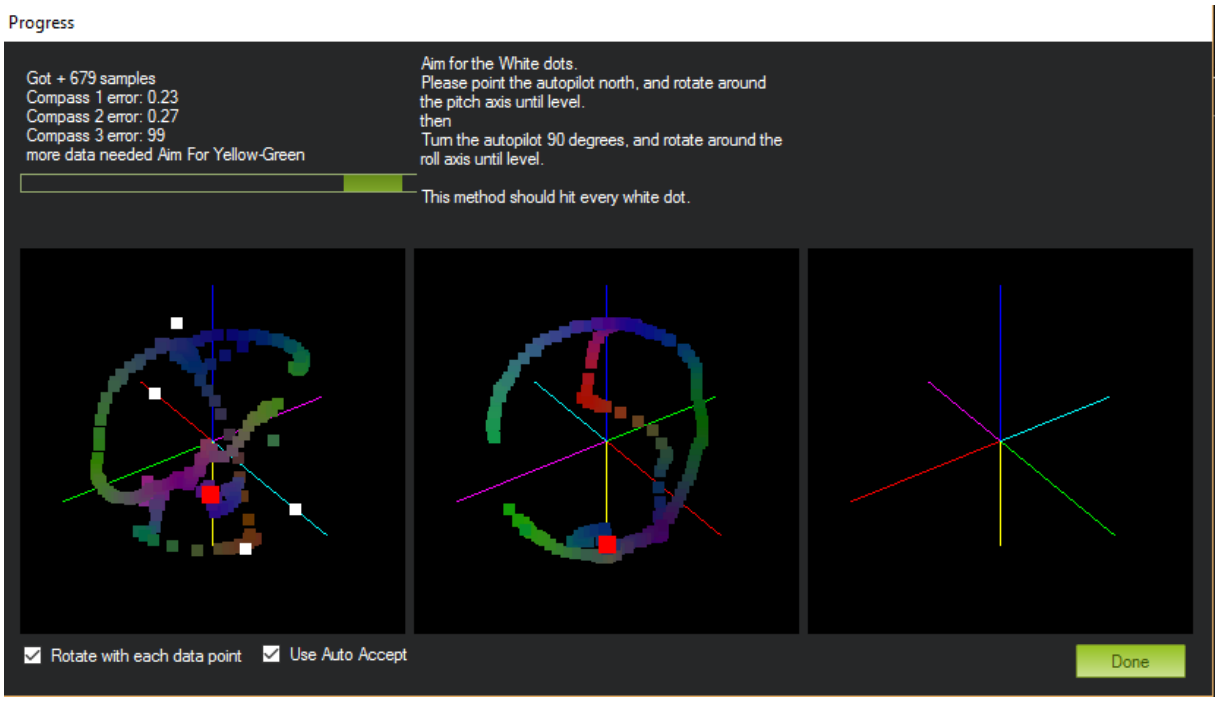


Figura 37 – Calibración de la brújula y del GPS

- El objetivo es rotar el UAV para que el rastro de color golpee cada uno de los puntos blancos. Una forma de hacerlo es mantener el vehículo en el aire y rotarlo lentamente para que cada lado (delantero, trasero, izquierdo, derecho, superior e inferior) apunte hacia la tierra durante unos segundos.

- La calibración se completará automáticamente cuando tenga datos para todas las posiciones. En este punto, aparecerá otra ventana que indicará que está guardando las posiciones recién calculadas. Estos se muestran en la pantalla principal debajo de cada brújula asociada.

4.5. Calibración radio control

Los transmisores RC permiten al piloto establecer el modo de vuelo, controlar el movimiento y la orientación del vehículo y también activar / desactivar las funciones auxiliares, es decir, subir y bajar un tren de aterrizaje, rotar la posición de la cámara de un gimbal etc.

La calibración RC implica capturar los valores mínimos, máximos y de "ajuste" de cada canal de entrada RC para que ArduPilot pueda interpretar correctamente la entrada.

Los pasos a seguir serían los siguientes:

- Asegurarse de que la batería esté desconectada, esto es importante porque es posible armar accidentalmente el vehículo durante el proceso de calibración.
- Asegurarse de que el receptor RC esta vinculado al receptor instalado en la controladora de vuelo.
- Encender el transmisor RC y si tiene "pestañas de ajuste" asegurarse de que estén en el medio.
- Conectar la controladora de vuelo al PC con un cable USB.
- En Mission Planner, presione el botón "Conectar" y abrir la configuración inicial de Mission Planner, Hardware obligatorio, Pantalla de calibración de radio.
- Deben aparecer algunas barras verdes que muestran que el ArduPilot está recibiendo información del transmisor / receptor. Si no aparecen barras, hay que verificar el LED del receptor.

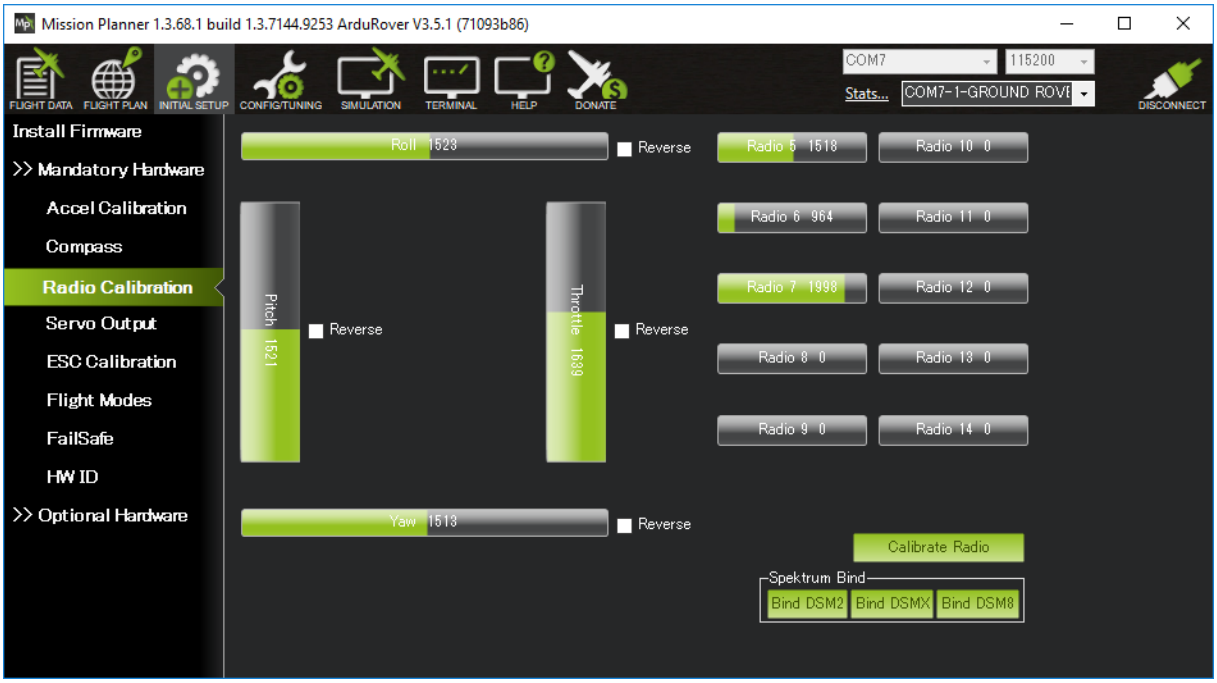


Figura 38 – Calibración Radio Control

- Hay que verificar la asignación de canales en el transmisor, es decir, verificar qué canales de entrada están controlados por las palancas, interruptores y perillas del transmisor. Mover las palancas, perillas e interruptores y observar qué barras verdes se mueven.

  La primera vez que se usa el transmisor con ArduPilot, es probable que sea necesario cambiar la asignación de canales del transmisor. Esto se hace en el transmisor mismo utilizando su menú de configuración incorporado.

  - Consultar si el transmisor es Mode1 o Mode2
  - Roll stick debe controlar el canal 1
  - El Pitch Stick debe controlar el canal 2
  - La palanca del acelerador debe controlar el canal 3
  - El palo de guiñada debe controlar el canal 4
  - Debe configurarse un interruptor de 3 posiciones, para controlar el modo de vuelo, fijándolo en el Canal 5.

- Se mueven los rodillos de balanceo, inclinación, aceleración y guiñada del transmisor para asegurar que las barras verdes se muevan en la dirección correcta:

  - Para los canales de balanceo, aceleración y guiñada, las barras verdes deben moverse en la misma dirección que los palos físicos del transmisor.
  - Para la inclinación, la barra verde debe moverse en la dirección opuesta a la palanca física del transmisor.
  - Si una de las barras verdes se mueve en la dirección incorrecta, hay que invertir el canal en el transmisor.

Paso a paso en misión Planner sería así:

- Mission Planner, Hardware obligatorio, Pantalla de calibración de radio.
- Hacer clic en el botón verde "Calibrar radio" en la parte inferior derecha.
- Presionar "OK" cuando se le solicite que verifique que el equipo de control de radio esté encendido, la batería no esté conectada y las hélices no estén conectadas.

Figura 39 – Calibración Radio

- Mover las palancas de control del transmisor, las perillas y los interruptores a sus límites. Aparecerán líneas rojas en las barras de calibración para mostrar los valores mínimos y máximos vistos hasta ahora.

Figura 40 – Calibración Radio

- Seleccionar “Hacer clic” cuando haya terminado.
- Aparecerá una ventana con el mensaje "Asegúrese de que todos sus palos estén centrados y que el acelerador esté abajo y haga clic en Aceptar para continuar". Muever el acelerador a cero y presionar "OK".
- Mission Planner mostrará un resumen de los datos de calibración. Los valores normales son alrededor de 1100 para mínimos y 1900 para máximos.

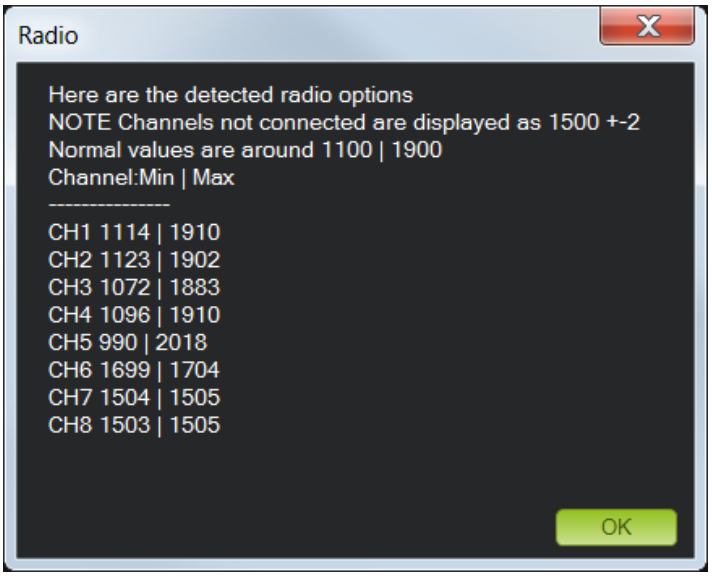


Figura 41 – Ventana de confirmación de calibración RC

Transmisores Modo 1 y Modo 2

Hay dos configuraciones principales de transmisor:

- Modo 1: la palanca izquierda controla el cabeceo y la guiñada, la palanca derecha controlará el acelerador y el balanceo.

- Modo 2: el joystick izquierdo controla el acelerador y la guiñada; la palanca derecha controlará el cabeceo y el balanceo.

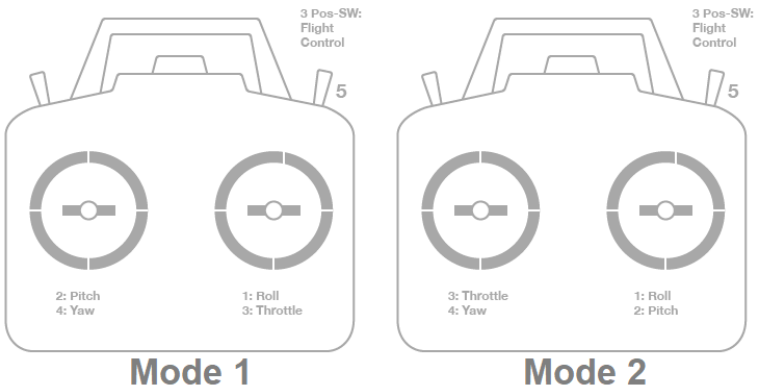


Figura 42 – Modo 1 y Modo 2 RC

4.5.1. Asignaciones de canales

Las asignaciones de canales predeterminadas de un Copter son:

- Canal 1: Roll
- Canal 2: Pitch
- Canal 3: acelerador
- Canal 4: Yaw
- Canal 5: modos de vuelo
- Canal 6: (Opcional) Sintonización a bordo o montaje de cámara (mapeado a la perilla de sintonización del transmisor)
- Canal 7 a 12 : (Opcional) Interruptores de funciones auxiliares

En el caso del UAV de este proyecto, tenemos presente la limitación de 6 canales, por lo tanto se ha seguido la asignación de canales habitual, asignando el canal 6 para el guiñado del Gimbal.

4.6. Configuración de modos de vuelo

El mapeo entre la posición del interruptor y el modo de vuelo se establece en la pantalla del modo de vuelo de Mission Planner.

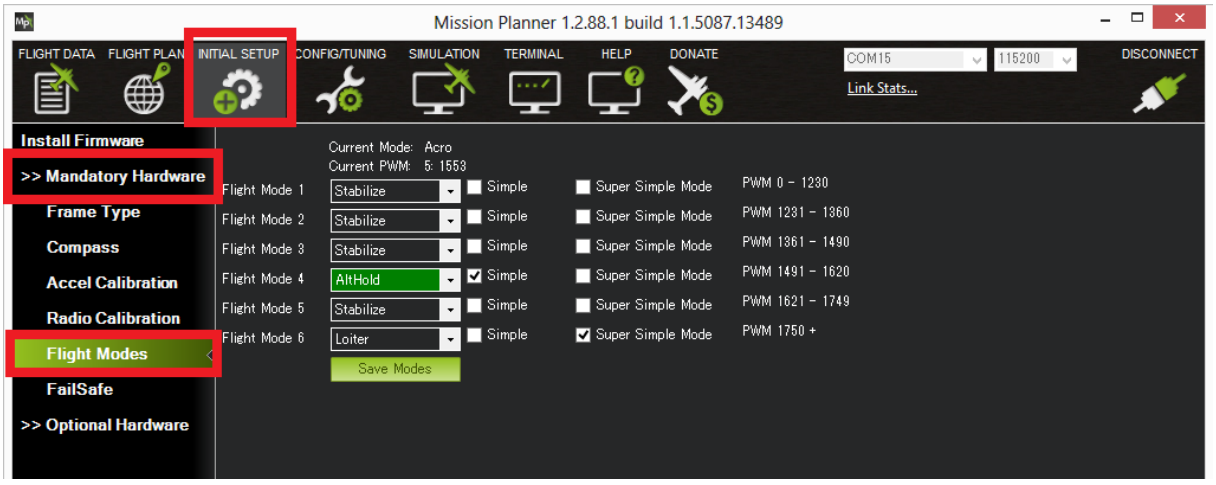


Figura 43 – Configuración de los modos de vuelo

Se pueden configurar los modos de vuelo disponibles en el transmisor haciendo lo siguiente:

- Encender el transmisor RC.
- Conectar el Pixhawk Mission Planner.
- Ir a la configuración inicial, Hardware obligatorio, Pantalla de modos de vuelo.

A medida que mueve el interruptor de modo de vuelo de su transmisor, la barra de estado verde se moverá a una posición diferente.

- Usando el menú desplegable en cada línea, seleccionar el modo de vuelo para esa posición del interruptor.
- Es recomendable reservar al menos una posición del interruptor para el modo de vuelo Stabilize. (Estabilizado)
- Al terminar, presionar el botón Guardar modos.

En esta configuración se han asignado, Stabilize para la posición 1 del stick del canal 5, en la posición 2 del Stick, Possion Hol, y en la tercera posición del stick, RTL, Return To Home, o lo que es lo mismo, Vuelta a Casa, o punto de partida.

4.6.1. Configurar el canal del modo de vuelo

El canal de modo de vuelo es el canal de radio de entrada que ArduPilot monitorea en busca de cambios de modo.

En Copter, este siempre es el canal 5.

4.7. Calibración del controlador electrónico de velocidad (ESC)

Los controladores electrónicos de velocidad son responsables de hacer girar los motores a la velocidad solicitada por la controladora de vuelo. La mayoría de los ESC deben calibrarse para que conozcan los valores mínimos y máximos de pwm que enviará el controlador de vuelo.

Antes de comenzar con la calibración de los ESC, hay que tener una serie de precauciones:

¡Verificación de seguridad!

Asegurarse de que el UAV, no tiene las hélices puestas y que la controladora de vuelo no está conectada a la computadora a través de USB, además de comprobar que la batería Lipo esté desconectada.

Figura 44 – Precauciones en la calibración de los ESC

Pasos a seguir:

1. Encender el transmisor y poner el acelerador al máximo.

Figura 45 – Calibración de los ESC

2. Conectar la batería de Lipo. Los LED rojo, azul y amarillo de la controladora de vuelo se iluminarán en un patrón cíclico. Esto significa que está listo para pasar al modo de calibración ESC la próxima vez que se conecte.

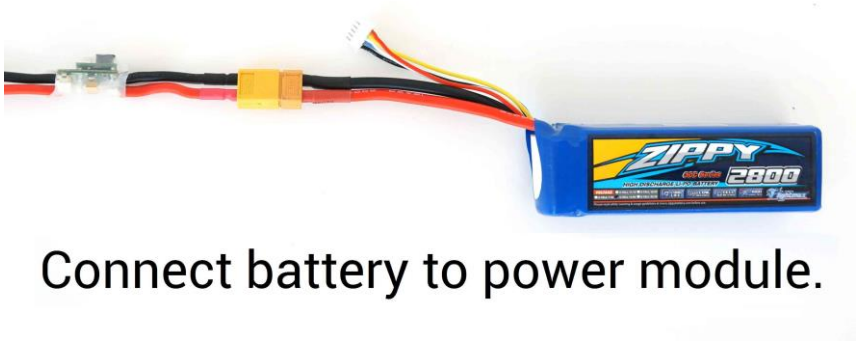


Figura 46 – Paso 1 calibración ESC

3. Con la palanca del acelerador del transmisor todavía alta, desconectar y vuelver a conectar la batería.

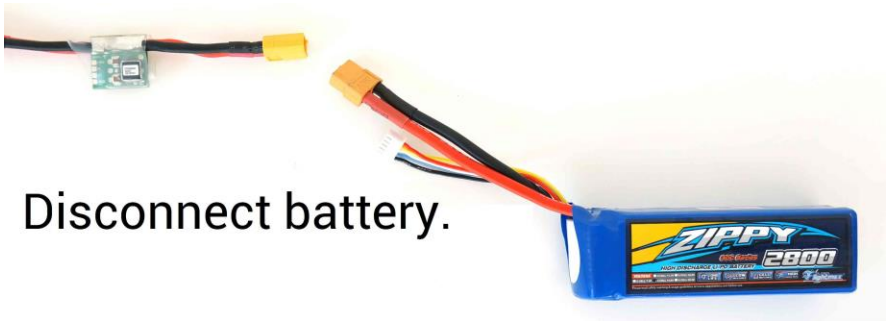


Figura 47 – Paso 2 calibración ESC

4. En el caso de la controladora Pixhawk que se ha instalado en el UAV, en primer lugar hay que mantener presionado el botón de seguridad hasta que se muestre en rojo.
5. La controladora de vuelo ahora está en modo de calibración ESC. Y se podrá apreciar que los LED rojos y azules parpadean alternativamente.
6. Esperar a que los ESC emitan el tono musical, el número regular de pitidos que indica el recuento de celdas de la batería (es decir, 3 para 3S, 4 para 4S) y luego dos pitidos adicionales para indicar que se ha capturado el acelerador máximo.
7. Tirar de la palanca del acelerador del transmisor hacia abajo a su posición mínima.

Figura 48 - Paso 3 Calibración ESC

8. Los ESC deben emitir un tono largo que indique que se ha capturado el acelerador mínimo y que se ha completado la calibración.
9. Si se escucha el tono largo que indica una calibración exitosa, los ESC están "en vivo" y si se sube un poco el acelerador, deberían girar.
10. Ajustar el acelerador al mínimo y desconectar la batería para salir del modo de calibración ESC.

5. FUNCIONAMIENTO Y MANEJO

5.1. Modos de vuelo

Ardupilot tiene 22 modos de vuelo integrados de vuelo, 10 de los cuales se usan regularmente. Hay modos para soportar diferentes niveles / tipos de estabilización de vuelo, un piloto automático sofisticado, un sistema de seguimiento, etc.

Los modos de vuelo se controlan a través de la radio (a través de un interruptor transmisor), a través de comandos de misión o mediante comandos de una estación terrestre (GCS) o una computadora complementaria.

Modo	Alt Ctrl	Pos Ctrl	GPS	Resumen
Acro	-	-		Mantiene actitud, no tiene nivel propio
Alt Hold	s	+		Mantiene la altitud y autonivela el roll & pitch
Auto	UNA	UNA	Y	Ejecuta misión predefinida
AutoTune	s	UNA	Y	Procedimiento automatizado de cabeceo y banco para mejorar los bucles de control
Freno	s	UNA	Y	Trae el helicóptero a una parada inmediata
Circulo	s	UNA	Y	Rodea automáticamente un punto en frente del vehículo
Deriva	-	+	Y	Como estabilizar, pero coordina el guiñada con un giro como un avión
Dar la vuelta	UNA	UNA		Se levanta y completa una vuelta automatizada
FlowHold	s	UNA		Control de posición usando flujo óptico
Seguir	s	UNA	Y	Sigue a otro vehículo
Guiado	UNA	UNA	Y	Navega a puntos individuales comandados por GCS

Figura 49 – Modos de Vuelo 1

Tierra	UNA	s	(Y)	Reduce la altitud hasta el nivel del suelo, intenta ir directamente hacia abajo
Holgazanear	s	s	Y	Mantiene altitud y posición, usa GPS para movimientos
PosHold	s	+	Y	Como merodeador, pero balanceo manual y cabeceo cuando los palos no están centrados
RTL	UNA	UNA	Y	Se vuelven a colocar encima del lugar de despegue, también pueden incluir aterrizaje
Simple / Super simple			Y	Un complemento a los modos de vuelo para usar la vista del piloto en lugar de la orientación de guiñada
SmartRTL	UNA	UNA	Y	RTL, pero traza el camino para llegar a casa
Deporte	s	s		Alt-hold, pero mantiene pitch & roll cuando los palos están centrados
Estabilizar	-	+		Autonivela el eje de balanceo y cabeceo
SysID	-	+		Modo especial de diagnóstico / modelado
Lanzar	UNA	UNA	Y	Mantiene la posición después de un despegue de lanzamiento
Zigzag	UNA	UNA	Y	Útil para la fumigación de cultivos.

Figura 50 – Modos de vuelo 2

Símbolo	Definición
-	Control manual
+	Control manual con límites y autonivelación.
s	El piloto controla la velocidad de ascenso
UNA	Control automático

Figura 51 – Leyendas modos de vuelo

5.1.1. Dependencia de GPS

Los modos de vuelo que usan datos de posicionamiento GPS, requieren un bloqueo GPS activo antes del despegue. Para comprobar si la controladora de vuelo ha adquirido el bloqueo GPS se debe prestar atención a la indicación LED.

A continuación, se muestra un resumen de la dependencia del GPS para los modos de vuelo de Copter.

Requiere bloqueo de GPS antes del despegue:

- Auto
- Circulo
- Deriva
- Seguir
- Sígueme
- Guiado
- Holgazanear
- PosHold
- RTL (Volver al lanzamiento)
- RTL inteligente (retorno al lanzamiento)
- Lanzar
- Zigzag

No requiere bloqueo de GPS:

- Acro
- Alt Hold
- Estabilizar
- Deporte
- SysID
- Tierra

5.2. Verificaciones de seguridad previas al vuelo

ArduPilot incluye un conjunto de comprobaciones de seguridad previas al armado, que evitarán que el vehículo arme su sistema de propulsión. Estas comprobaciones ayudan a evitar accidentes o choques, pero también pueden desactivarse si es necesario.

5.2.1. Armando de los motores

Armar el UAV permite que los motores comiencen a girar. Antes de armar, hay que asegúrarse de que todas las personas, objetos y cualquier parte del cuerpo (por ejemplo, las manos) están fuera del alacance de las hélices. Luego se procederá de la siguiente forma:

1. Se enciende el transmisor.
2. Se enchufa la batería de LiPo. Las luces rojas y azules deben parpadear durante unos segundos mientras se calibran los giroscopios.
3. Las comprobaciones previas al armado se ejecutarán automáticamente y si se encuentra algún problema, el LED RGB parpadeará en amarillo y la falla se mostrará en la estación terrestre o en el Mission Planner si estamos conectados por telemetría.
4. Se comprueba que el interruptor de modo de vuelo esté configurado en Estabilizado, AltHold, Loiter o PosHold.
5. En el caso de la pixhawk al usar un interruptor de seguridad, hay que presionarlo hasta que la luz se encienda.
6. Para armar los motores, hay que mantener presionado el acelerador y el timón hacia la derecha durante 5 segundos.
7. Una vez armados, los LED se encenderán y las hélices comenzarán a girar.
8. Posteriormente mantenemos el acelerador suavemente para despegar.

5.2.2. Desarmando los motores

Desarmar los motores hará que los motores dejen de girar. Para desarmar los motores, se hará lo siguiente:

1. Compruebe que su interruptor de modo de vuelo esté configurado en Estabilizado, AltHold, Loiter o PosHold.
2. Mantener el acelerador al mínimo y el timón hacia la izquierda durante 2 segundos.

3. El LED comenzará a parpadear indicando que el vehículo está desarmado
4. Usando la Pixhawk con un interruptor de seguridad, se debe presionar hasta que el LED comience a parpadear.
5. Se desconecta la batería Lipo.
6. Y se apaga el transmisor.

5.3. Primer vuelo

Se coloca el helicóptero en un terreno nivelado y se conecte la batería, en este momento no se puede mover el cuadricóptero hasta que se complete la calibración del giroscopio (los LED parpadean en rojo y azul). Después hay que asegurarse de que el interruptor de modo RC esté en modo Estabilizado. Se arman motores como se ha explicado anteriormente y se levanta lentamente el acelerador hasta que el UAV se levante del suelo.

6. DIFICULTADES Y SOLUCIONES ADOPTADAS

A lo largo de este proyecto se han ido sucediendo distintas dificultades e imprevistos, tanto a nivel hardware como a nivel software. En algunas ocasionas, estos imprevistos han sido resueltos, pero en otras ha sido muy complicado cumplir los plazos junto con el éxito del proyecto de manera conjunta. Estos son los principales problemas que han ido surgiendo en su transcurso:

- Elección de componentes. A la hora de elegir el proyecto, se partía con la certeza de que no iba a ser un proyecto barato, debido a la envergadura y complejidad de este. Se intentaron reducir los costes del mismo, en todo momento, buscando ofertas de segunda mano, y de grandes mayoristas.

  Estas ofertas no se sucedían, y el tiempo se iba sucediendo. La solución final, fue rebajar el coste, reduciendo la calidad o la categoría de algunos componentes, y realizar la compra.

- Tiempos de entrega. Pese a realizar la compra de componentes con un margen considerable de tiempo, el material provenía de China, y ningún componente cumplió con los plazos de entrega prometidos. Lo que nos retrasó el ensamble del UAV 2 semanas. Los vendedores se excusaron con que: el 11 del 11, y fechas próximas al Blackfriday, suponen una carga muy elevada de trabajo para las compañías de transporte, y se retrasan los envíos.

- Componentes erróneos. Una vez habían llegado todos los componentes, se procedió al ensamblado del drone, y empezaron a sucederse más problemas. El receptor solicitado al vendedor, no había sido enviado, y su lugar, se había enviado otro tipo de receptor que era inviable instalar en nuestra configuración, debido al protocolo de comunicación que habíamos establecido en nuestra configuración.

  La solución adoptada fue recurrir al mercado de segunda mano, y adquirir uno. El cuál también sufrió unos días de retraso en el envío.

- Otros problemas menores en este caso, porque tuvieron una solución más rápida, fue la escasez, o nulidad de conectores que enviaron los proveedores. Concretamente a la hora de alimentar el power module con los cables salientes de la PDB, es necesario un conector xt60, que el proveedor no incluyó. Como se comentaba anteriormente, se optó por doblar el cable sobre si mismo para aumentar su perímetro, pre estañearlo, e introducirlo en el conector xt60 hembra del power module.

  Lo mismo sucedió con la alimentación de la placa controladora del Gimbal, se daba por hecho que se incluiría un conector para alimentarla, pero no. Se optó por solucionarlo con cables de un pack de Arduino, con terminales en formato pin macho y pin hembra, se llevó el pin positivo de alimentación del gimbal, al pin positivo de la lipo, e igualmente con el negativo.

- Otro problema grave que ya se ha mencionado con anterioridad en esta memoria, fue la incompatibilidad del sonar de Arduino que incluía el pack que nos prestó el profesor de la asignatura de Diseño de Sistemas Empotrados.

  En un principio se contaba con él como componente estrella para dificultar la realización de este proyecto, pero hasta que no recibimos todos los componentes, y se realizó el ensamblado de los mismos, no se pudo verificar su incompatibilidad.

  Como se comentaba con anterioridad, la idea inicial era aportar código original a este proyecto, haciendo utilidad de un sonar para detectar el cambio de alturas que puedan propiciar montículos, u obstáculos en el aterrizaje, proporcionado en la asignatura de Diseño de Sistemas Empotrados.

  Pero este proyecto no muere aquí, se realizará un pedido de sonares analógicos, y no solo se instalarán en la zona inferior del dron, para facilitar su aterrizaje, e irlo acomodando al gusto del piloto a través del software, sino que también se implementarán sonares analógicos para la detección de objetos, tanto frontal, como lateral, como posterior, con el objetivo de crear un UAV que sea complicado chocarle.

- El problema determinante para que el UAV no acabara volando fue la lipo. Se desconoce el momento preciso en el cuál la batería llegó al fin de su vida. Hay varias hipótesis:

1. Que llegara ya estropeada desde China.

2. Que durante la configuración del UAV esta bajara su voltaje por debajo del límite y alguna de las celdas sufriera un punto de no retorno.

3. Una de las hipótesis más factibles, es que debido a la compra de un cargador de lipos, muy económico, este dañara la batería al no hacer una carga debidamente balanceada, como necesitan este tipo de lipos.

7. COSTES

En el capítulo de presupuesto están listados todos los costes del proyecto. Los costes se dividen entre costes de mano de obra, y costes hardware. Los costes hardware exceden el presupuesto inicialmente acordado de 50€ por miembro del grupo, lo hacía un total de 200€ estimados.

Costes de mano de Obr

N días * **h** día = **t** total = 70 * 3 ≈ 200 h/día

Horas/día * Coste hora = 200 * 18 ≈ 3.600€

Costes Hardware

Tabla de costes materiales			
COMPONENTES	**Cantidad**	**Coste Unidad**	**Coste Completo**
Frame F450	1	23,56 €	23,56 €
Motores	4	9,03 €	36,12 €
ESC (Variadores)	4	11,38 €	45,52 €
FC Pixhawk (Controladora de vuelo)	1	53,63 €	53,63 €
Gps con módulo de brújula	1	26,91 €	26,91 €
Bateria Lipo 4200 Mah	1	23,48 €	23,48 €
Pack hélices	2	3,57 €	7,14 €
Emisora FlySky FS-i6	1	36,13 €	36,13 €
Receptor FS-iA6	1	10,06 €	10,06 €
Cargador de Lipo	1	2,80 €	2,80 €
TOTAL	17	**265,35 €**	

Figura 52 – Tabla de costes hardware

8. BIBLIOGRAFÍA

- Proyectos en Arduino. Recuperado el 9 de Octubre de 2019.

  https://forum.arduino.cc/

- Definición UAV. Recuperado el 9 de Octubre de 2019.

  https://es.wikipedia.org/wiki/Veh%C3%ADculo_a%C3%A9reo_no_tripulado

- Documentación Ardupilot. Recuperado el 7 de Diciembre de 2019.

  https://ardupilot.org/ardupilot/

- Repositorio de código Libre Ardupilot. Recuperado el 7 de Diciembre de 2019.

  https://github.com/ArduPilot/ardupilot

- Drones DIY. Recuperado el 7 de Diciembre de 2019. Recuperado el 7 de Diciembre de 2019.

  https://diydrones.com/

- Repositorios de código abierto Drone Code. Recuperado el 7 de Diciembre de 2019.

  https://www.dronecode.org/

- Repositorio de código abierto PX4. Recuperado el 7 de Diciembre de 2019.

  https://github.com/PX4/Firmware

9. ANEXO I

9.1. Características técnicas y función de los componentes electrónicos

9.1.1. Controladora de vuelo Pixhawk

9.1.1.1. Especificaciones

- Procesador
  - Núcleo ARM Cortex M4 de 32 bits con FPU
  - 168 Mhz / 256 KB RAM / 2 MB Flash
  - Coprocesador a prueba de fallos de 32 bits

- Sensores
  - MPU6000 como acelerador principal y giroscopio
  - ST Micro giroscopio de 16 bits
  - ST Micro acelerómetro / brújula de 14 bits (magnetómetro)
  - MEAS barómetro

- Alimentación
  - Controlador de diodo ideal con conmutación por error automática
  - Servo rail de alta potencia (7 V) y listo para alta corriente
  - Todas las salidas periféricas protegidas contra sobre corriente, todas las entradas protegidas contra ESD

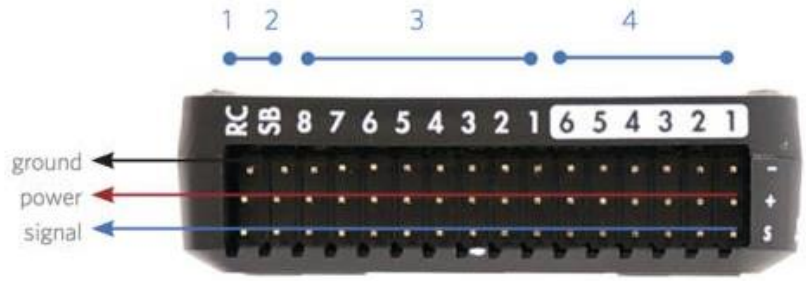
- Interfaces
  - 5 puertos serie UART, 1 de alta potencia, 2 con control de flujo HW
  - Entrada satelital Spektrum DSM / DSM2 / DSM-X
  - Entrada Futaba S.BUS (salida aún no implementada)
  - Señal de suma PPM
  - Entrada RSSI (PWM o voltaje)
  - I2C, SPI, 2x CAN, USB
  - Entradas de 3.3V y 6.6V ADC

- Dimensiones
  - Peso 38 g (1.3 oz)
  - Ancho 50 mm (2.0 ”)
  - Altura 15.5 mm (.6 ")
  - Longitud 81.5 mm (3.2 ")

9.1.1.2. Asignación de pines y conectores

1 Spektrum DSM receiver
2 Telemetry (on-screen display)
3 Telemetry (radio telemetry)
4 USB
5 SPI (serial peripheral interface) bus
6 Power module
7 Safety switch button
8 Buzzer
9 Serial
10 GPS module
11 CAN (controller area network) bus
12 I²C splitter or compass module
13 Analog to digital converter 6.6 V
14 Analog to digital converter 3.3 V
15 LED indicator

1 Input/output reset button
2 SD card
3 Flight management reset button
4 Micro-USB port



1 Radio control receiver input
2 S.Bus output
3 Main outputs
4 Auxiliary outputs

Figura 53 – Asignación de pines y conectores

1 Flight management unit (FMU) power
2 FMU bootloader mode (flashing) or error (solid)
3 Input/output unit power
4 I/O bootloader mode (flashing) or error (solid)
5 Activity (flashing indicates all units responsive)

Figura 54 – Indicadores LEDs de la controladora de vuelo

9.1.1.3. Conectores superiores

Figura 55 - Conectores supueriores de la Pixhawk

9.1.1.4. Otros Conectores

Conectores Pixhawk PWM para servos y ESC y entrada PPM-SUM y salida SBUS

Figura 56 - Conectores Pixhawk PWM para servos y ESC y entrada PPM-SUM y salida SBUS

9.1.1.5. Diagrama genérico de conectores Pixhawk

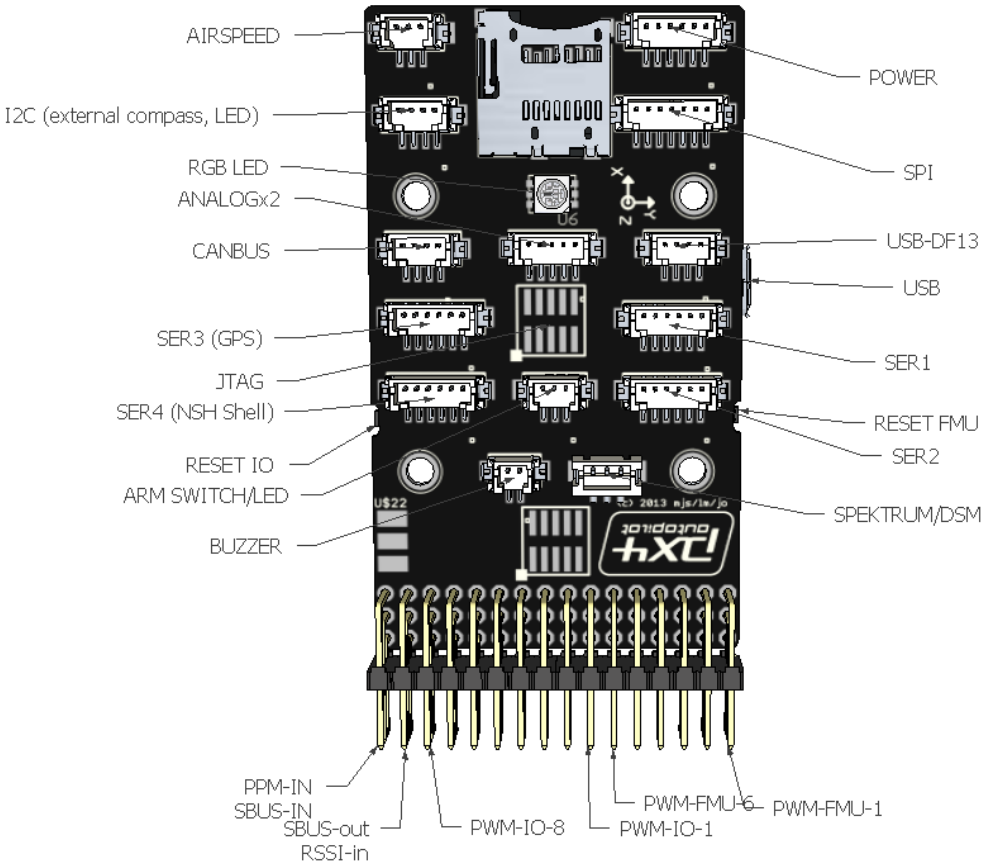


Figura 57 – Diagrama genérico pines pixhawk

9.1.1.6. Puertos TELEM1, TELEM

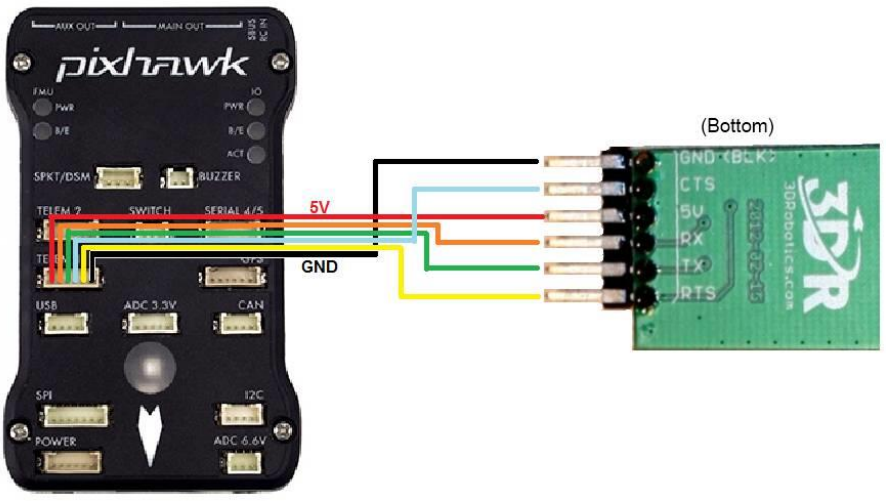


Figura 58 – Pines puerto de Telemetría

Alfiler	Señal	Voltio
1 (rojo)	VCC	+ 5V
2 (negro)	TX (FUERA)	+ 3.3V
3 (negro)	RX (ENTRADA)	+ 3.3V
4 (negro)	CTS	+ 3.3V
5 (negro)	RTS	+ 3.3V
6 (negro)	GND	GND

Figura 59 – Asignación de pines de Telemetría

9.1.1.7. Puerto GPS

Alfiler	Señal	Voltio
1 (rojo)	VCC	+ 5V
2 (negro)	TX (FUERA)	+ 3.3V
3 (negro)	RX (ENTRADA)	+ 3.3V
4 (negro)	CAN2 TX	+ 3.3V
5 (negro)	CAN2 RX	+ 3.3V
6 (negro)	GND	GND

Figura 60 – Asignación de pines del puerto GPS

9.1.1.8. Puerto SERIAL 4/5

Debido a limitaciones de espacio, hay dos puertos en un conector.

Alfiler	Señal	Voltio
1 (rojo)	VCC	+ 5V
2 (negro)	TX (# 4)	+ 3.3V
3 (negro)	RX (# 4)	+ 3.3V
4 (negro)	TX (# 5)	+ 3.3V
5 (negro)	RX (# 5)	+ 3.3V
6 (negro)	GND	GND

Figura 61 – Asignación de pines puerto Serial

9.1.1.9. ADC 6.6V

Alfiler	Señal	Voltio
1 (rojo)	VCC	+ 5V
2 (negro)	ADC IN	hasta + 6.6V
3 (negro)	GND	GND

Figura 62 – Asignación de pines ADC 6.6V

9.1.1.10. ADC 3.3V

Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	ADC IN	up to +3.3V
3 (blk)	GND	GND
4 (blk)	ADC IN	up to +3.3V
5 (blk)	GND	GND

Figura 63 - Asignación de pines ADC 3.3V

9.1.1.11. I2C

Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	SCL	+3.3 (pullups)
3 (blk)	SDA	+3.3 (pullups)
4 (blk)	GND	GND

Figura 64 - Asignación de pines puerto I2C

9.1.1.12. POWER

Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	VCC	+5V
3 (blk)	CURRENT	up to +3.3V
4 (blk)	VOLTAGE	up to +3.3V
5 (blk)	GND	GND
6 (blk)	GND	GND

Figura 65 - Asignación de pines al puerto Power

9.1.1.13. SWITCH

Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	VCC	+5V
3 (blk)	CURRENT	up to +3.3V
4 (blk)	VOLTAGE	up to +3.3V
5 (blk)	GND	GND
6 (blk)	GND	GND

Figura 66 - Asignación de pines al puerto Power

9.1.1.14. Pines de entrada analógica

Esta sección enumera los pines analógicos disponibles en Pixhawk. Estos son pines virtuales, definidos en el firmware.

- **Pin virtual 2 y conector de alimentación Pin 4:** pin de voltaje del conector de administración de energía, acepta hasta 3.3V, generalmente conectado a un módulo de alimentación con escala 10.1: 1

- **Pin virtual 3 y conector de alimentación Pin 3:** pin del conector de administración de energía, acepta hasta 3.3V, generalmente conectado a un módulo de alimentación con escala 17: 1

- **Pin virtual 4 y (sin pin conector):** Este pin virtual lee el voltaje en el conector de suministro de 5V. Se utiliza para proporcionar la lectura HWSTATUS.Vcc que las estaciones terrestres usan para mostrar el estado de 5V.

- **Pin virtual 13 y conector ADC 3.3V Pin 4:** esta toma un máximo de 3.3V. Puede usarse para sonar u otros sensores analógicos.

- **Pin virtual 14 y conector ADC 3.3V Pin 2:** esta toma un máximo de 3.3V. Puede usarse para un segundo sonar u otro sensor analógico.

- **Pin virtual 15 y conector ADC 6.6V Pin 2:** puerto del sensor de velocidad del aire analógico. Esto tiene una escala de 2: 1 incorporada, por lo que puede tomar entradas analógicas de hasta 6.6v. Usualmente se usa para velocidad aérea analógica, pero puede usarse para sonar analógico u otros sensores analógicos.

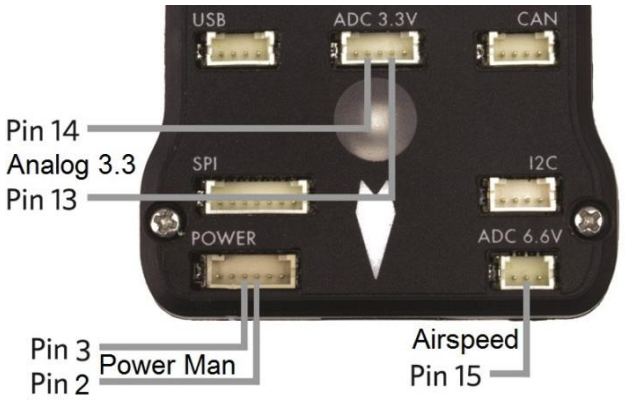


Figura 67 – Pines de entrada analógica

- **Pin virtual 102 :** voltaje de la línea de pines de potencia de los servo. Tiene una escala 3: 1, lo que le permite medir hasta 9.9V.

- **Pin virtual 103 :** voltaje del pin de entrada RSSI (entrada de intensidad de señal recibida) (pin de salida del conector SBus). Este es el voltaje medido por el pin de entrada RSSI en el conector de salida SBUS (el pin inferior del segundo último servo conector en la línea de pines de servo de 14 conectores).

  Alternativamente, esto puede servir como salida SBus configurando el BRD_SBUS_OUTparámetro (Copter , Plane , Rover).

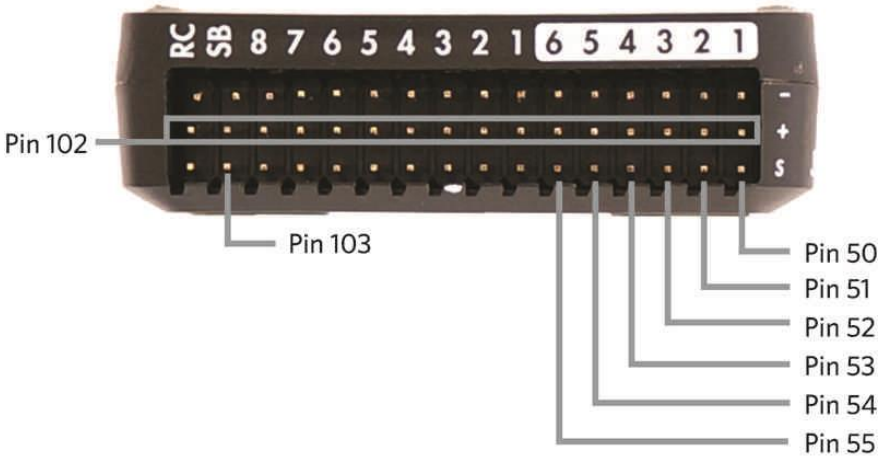


Figura 68 – Pines de entrada analógica de la Pixhawk

9.1.1.15. Salidas y entradas digitales Pixhawk

El Pixhawk no tiene pines de entrada o salida digital dedicados en sus conectores DF13, pero puede asignar hasta 6 de los conectores "AUX SERVO" para que sean salidas / entradas digitales. Estos son los primeros 6 de los 14 servo conectores de tres pines en el extremo de la placa. Están marcados como servo pin AUX 1 - 6 en la serigrafía como se ve arriba.

Para establecer el número de estos pines que están disponibles como entradas / salidas digitales, hay que configurar el parámetro BRD_PWM_COUNT. En Pixhawk, el valor predeterminado es 4, lo que significa que los primeros 4 conectores AUX son para servos (PWM) y los últimos 2 son para entradas / salidas digitales. Si establece BRD_PWM_COUNT en 0, entonces tendría 6 pines digitales virtuales y aún tendría 8 salidas PWM en el resto del conector.

Los 6 pines posibles están disponibles para las variables PIN como números de pin 50 a 55 inclusive. Entonces, si se tiene BRD_PWM_COUNT con el valor predeterminado de 4, los dos pines de salida digital serán los números de pin 54 y 55.

En resumen:

Si BRD_PWM_CNT = 2 entonces

- 50 = RC9
- 51 = RC10
- 52 = Aux 3
- 53 = Aux 4
- 54 = Aux 5
- 55 = Aux 6

Si BRD_PWM_CNT = 4 entonces

- 50 = RC9
- 51 = RC10
- 52 = RC11
- 53 = RC12
- 54 = Aux 5
- 55 = Aux 6

Si BRD_PWM_CNT = 6 entonces

- 50 = RC9
- 51 = RC10
- 52 = RC11
- 53 = RC12
- 54 = RC13
- 55 = RC14

Por defecto, los pines son salidas digitales como se describe anteriormente. En cambio, un pin digital será una entrada digital si se asigna a un parámetro que representa una entrada digital.

Por ejemplo, establecer CAM_FEEDBACK_PIN en 50 hará que el pin 50 sea la entrada digital que recibe una señal de la cámara cuando se toma una fotografía.

9.1.2. Power Module

Muchas controladoras de vuelo se pueden comprar con un módulo de alimentación analógico, que proporciona una fuente de alimentación estable a la controladora de vuelo, y también permite medir el voltaje de la batería y el consumo de corriente.

9.1.2.1. Especificaciones

A continuación, se especifican los límites del power module instalado:

- Voltaje de entrada máximo de 18V (4S lipo)
- Máximo de 90 amperios (pero solo capaz de medir hasta 60 amperios)
- Proporciona una fuente de alimentación de 5.37V y 2.25Amp a la controladora de vuelo.

9.1.2.2. Asignación de pines

El cable de 6 clavijas del módulo de alimentación se conecta al puerto de alimentación de la controladora.

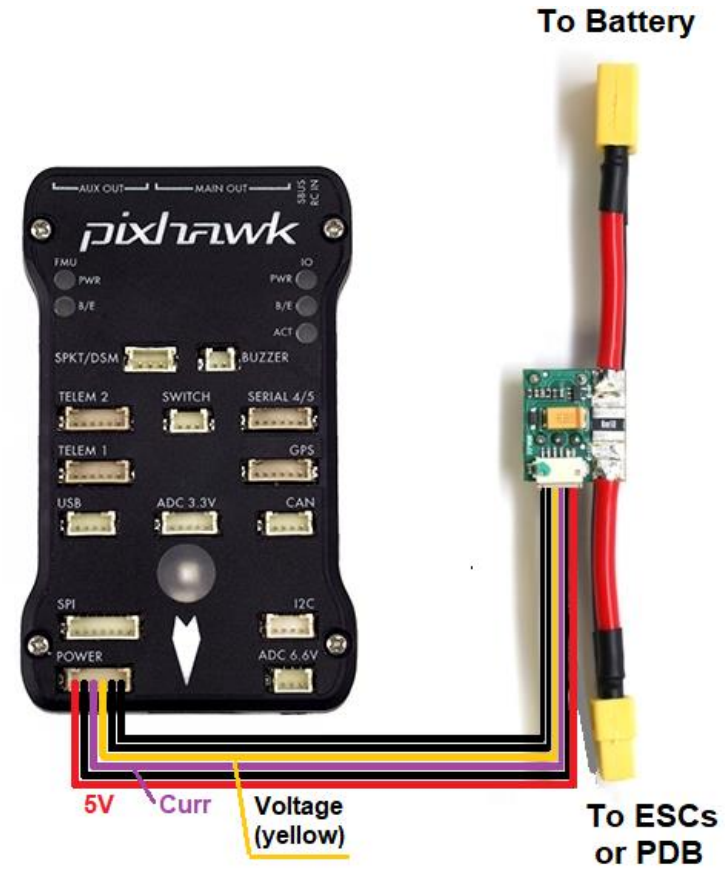


Figura 69 – Asignación de pines del Power Module

La batería está conectada al conector macho del módulo de alimentación. El ESC o el tablero de distribución de energía deben conectarse al conector hembra del módulo de energía.

9.1.2.3. Telemetría

Una *radio de telemetría SiK* es una de las formas más fáciles de configurar una conexión de telemetría entre la controladora de vuelo y una estación terrestre, u ordeandor.

Figura 70 – Conexiones de telemetría

Una radio de telemetría SiK es una plataforma de radio de fuente abierta pequeña, poco pesada y económica, que generalmente permite rangos de más de 300 m, este rango se puede extender a varios kilómetros con el uso de una antenas complementarias. La radio utiliza firmware de código abierto que ha sido especialmente diseñado para funcionar bien con los paquetes MAVLink y para integrarse con Mission Planner, el software que se ha usado en este proyecto para configurar el dron.

Las radios pueden ser 915Mhz o 433Mhz, en este caso se ha usado una de 433Mhz.

9.1.2.4. Características

Las características principales de SiK Radio se enumeran a continuación:

- Tamaño muy pequeño
- Peso ligero (menos de 4 gramos sin antena)
- Disponible en variantes de 900MHz o 433MHz (solo v2)
- Sensibilidad del receptor a -121 dBm
- Potencia de transmisión de hasta 20dBm (100mW)
- Enlace serial transparente

- Velocidades de datos aéreos de hasta 250 kbps
- Enmarcado del protocolo MAVLink e informes de estado
- Espectro extendido de salto de frecuencia (FHSS)
- Multiplexación por división de tiempo adaptativa (TDM)
- Soporte para LBT y AFA
- Ciclo de trabajo configurable
- Código de corrección de errores incorporado (puede corregir hasta 25% de errores de bits de datos)
- Alcance demostrado de varios kilómetros con una pequeña antena omnidireccional
- Se puede usar con un amplificador bidireccional para un alcance aún mayor
- Firmware de código abierto
- Comandos AT para configuración de radio
- Comandos RT para configuración remota de radio
- Control de flujo adaptativo cuando se usa con APM
- Basado en módulos de radio HM-TRP, con microcontrolador Si1000 8051 y módulo de radio Si4432

9.1.2.5. LED de estado

Las radios tienen 2 LED de estado, uno rojo y otro verde. El significado de los diferentes estados LED es:

- LED verde parpadeando - buscando otra radio
- LED verde fijo: el enlace se establece con otra radio
- LED rojo parpadeando - transmitiendo datos
- LED rojo fijo: en modo de actualización de firmware

9.1.2.6. Conexiones

La radio tiene un puerto micro-USB y un puerto DF13 de seis posiciones.

3DR Radio V2
Pin-out Description

Ground	6
RTS* (output)	5
CTS** (input)	4
Autopilot receiver (radio transmitter)	3
Autopilot transmitter (radio receiver)	2
Power (+5 V)	1

*RTS (request to send)
**CTS (clear to send)

Figura 71 – Pines del módulo de telemetría

Se usa el conector DF13 de 6 pines, para conectar la radio al “Telem 1” de su Pixhawk, también se puede usar “Telem 2” o “Serial 4/5” pero se recomienda usar el puerto predeterminado que es “Telem1”.

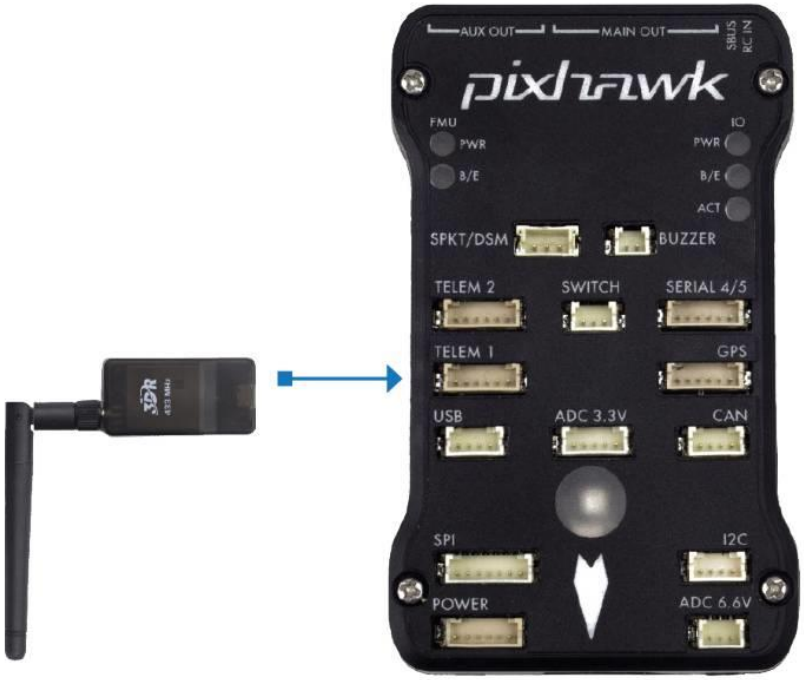


Figura 72 – Conexión a la Pixhawk

9.1.2.7. Conectando a un PC

Para conectarlo a un PC, se debe conectar el cable micro USB al PC. Los controladores necesarios deben instalarse automáticamente y la radio aparecerá como un nuevo "Puerto serie USB" en el Administrador de dispositivos de Windows en Puertos (COM y LPT). El menú desplegable de selección de puertos COM del Mission Planner también debe contener el mismo puerto COM nuevo.

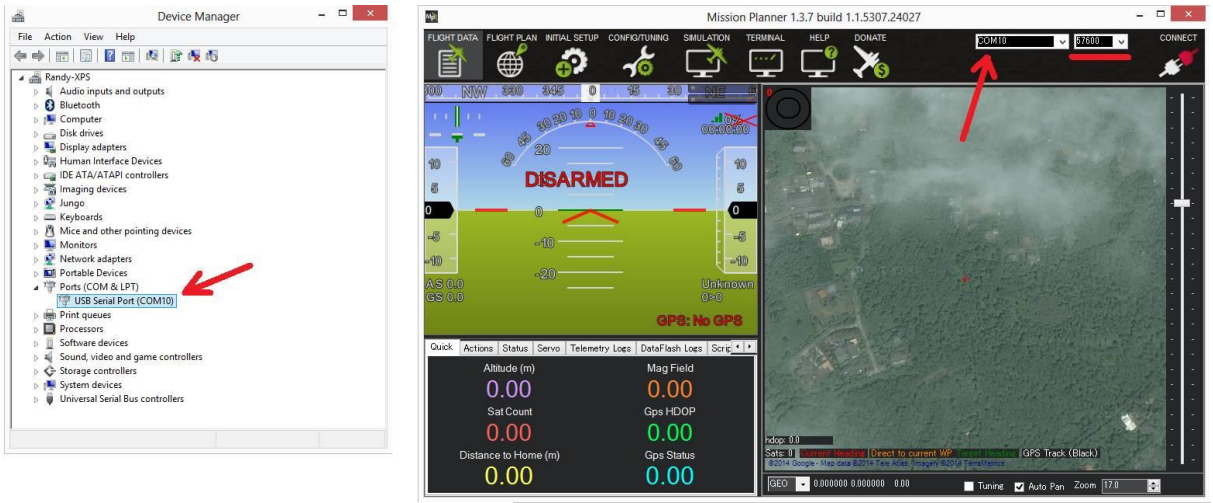


Figura 73- Conexión de la telemetría al PC

Para conectar las radios:

- Se selecciona el nuevo puerto COM, y se configura el menú desplegable de velocidad de transmisión (que aparece entre el puerto COM y los botones de conexión) en 57600.
- Se presiona el botón Conectar y si las dos radios se conectan con éxito, se debería poder inclinar el UAV hacia la izquierda y hacia la derecha y ver su actualización de actitud en el horizonte artificial de la pantalla de datos de vuelo del Mission Planner.

9.1.2.7.1. Conectarse a una tableta Android

La conexión de la radio a una tableta Android, se realiza con un cable micro USB en forma de L, y luego se siguen las instrucciones de la aplicación móvil instalada.

Figura 74 – Conexión del módulo de telemetría a una Tablet Android

10. ANEXO II

10.1. Librería COPTER.H

```
/*
    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.
    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
#pragma once
/*
  This is the main Copter class
 */

////////////////////////////////////////////////////////////////////////////////
// Header includes
////////////////////////////////////////////////////////////////////////////////

#include <cmath>
#include <stdio.h>
#include <stdarg.h>

#include <AP_HAL/AP_HAL.h>

// Common dependencies
#include <AP_Common/AP_Common.h>
#include <AP_Common/Location.h>
#include <AP_Param/AP_Param.h>
#include <StorageManager/StorageManager.h>

// Application dependencies
#include <GCS_MAVLink/GCS.h>
#include <AP_Logger/AP_Logger.h>          // ArduPilot Mega Flash Memory Library
#include <AP_Math/AP_Math.h>            // ArduPilot Mega Vector/Matrix math
Library
```

```
#include <AP_AccelCal/AP_AccelCal.h>                // interface and maths for
accelerometer calibration
#include <AP_InertialSensor/AP_InertialSensor.h>  // ArduPilot Mega Inertial
Sensor (accel & gyro) Library
#include <AP_AHRS/AP_AHRS.h>
#include <AP_NavEKF2/AP_NavEKF2.h>
#include <AP_NavEKF3/AP_NavEKF3.h>
#include <AP_Mission/AP_Mission.h>     // Mission command library
#include <AC_AttitudeControl/AC_AttitudeControl_Multi.h> // Attitude control
library
#include <AC_AttitudeControl/AC_AttitudeControl_Heli.h> // Attitude control
library for traditional helicopter
#include <AC_AttitudeControl/AC_PosControl.h>      // Position control library
#include <AP_Motors/AP_Motors.h>          // AP Motors library
#include <AP_Stats/AP_Stats.h>     // statistics library
#include <Filter/Filter.h>             // Filter library
#include <AP_Airspeed/AP_Airspeed.h>        // needed for AHRS build
#include <AP_Vehicle/AP_Vehicle.h>         // needed for AHRS build
#include <AP_InertialNav/AP_InertialNav.h>     // ArduPilot Mega inertial
navigation library
#include <AC_WPNav/AC_WPNav.h>           // ArduCopter waypoint navigation
library
#include <AC_WPNav/AC_Loiter.h>
#include <AC_WPNav/AC_Circle.h>          // circle navigation library
#include <AP_Declination/AP_Declination.h>     // ArduPilot Mega Declination
Helper Library
#include <AP_Scheduler/AP_Scheduler.h>       // main loop scheduler
#include <AP_RCMapper/AP_RCMapper.h>        // RC input mapping library
#include <AP_BattMonitor/AP_BattMonitor.h>     // Battery monitor library
#include <AP_LandingGear/AP_LandingGear.h>     // Landing Gear library
#include <AC_InputManager/AC_InputManager.h>        // Pilot input handling
library
#include <AC_InputManager/AC_InputManager_Heli.h>   // Heli specific pilot input
handling library
#include <AP_Arming/AP_Arming.h>
#include <AP_SmartRTL/AP_SmartRTL.h>
#include <AP_TempCalibration/AP_TempCalibration.h>
#include <AC_AutoTune/AC_AutoTune.h>
#include <AP_Common/AP_FWVersion.h>

// Configuration
#include "defines.h"
#include "config.h"

#if FRAME_CONFIG == HELI_FRAME
    #define AC_AttitudeControl_t AC_AttitudeControl_Heli
#else
```

```
    #define AC_AttitudeControl_t AC_AttitudeControl_Multi
#endif

#if FRAME_CONFIG == HELI_FRAME
 #define MOTOR_CLASS AP_MotorsHeli
#else
 #define MOTOR_CLASS AP_MotorsMulticopter
#endif

#if MODE_AUTOROTATE_ENABLED == ENABLED
 #include <AC_Autorotation/AC_Autorotation.h> // Autorotation controllers
#endif

#include "RC_Channel.h"         // RC Channel Library

#include "GCS_Mavlink.h"
#include "GCS_Copter.h"
#include "AP_Rally.h"           // Rally point library
#include "AP_Arming.h"

// libraries which are dependent on #defines in defines.h and/or config.h
#if BEACON_ENABLED == ENABLED
 #include <AP_Beacon/AP_Beacon.h>
#endif

#if AC_AVOID_ENABLED == ENABLED
 #include <AC_Avoidance/AC_Avoid.h>
#endif
#if AC_OAPATHPLANNER_ENABLED == ENABLED
 #include <AC_WPNav/AC_WPNav_OA.h>
 #include <AC_Avoidance/AP_OAPathPlanner.h>
#endif
#if SPRAYER_ENABLED == ENABLED
 # include <AC_Sprayer/AC_Sprayer.h>
#endif
#if GRIPPER_ENABLED == ENABLED
 # include <AP_Gripper/AP_Gripper.h>
#endif
#if PARACHUTE == ENABLED
 # include <AP_Parachute/AP_Parachute.h>
#endif
#if PRECISION_LANDING == ENABLED
 # include <AC_PrecLand/AC_PrecLand.h>
 # include <AP_IRLock/AP_IRLock.h>
#endif
#if ADSB_ENABLED == ENABLED
 # include <AP_ADSB/AP_ADSB.h>
```

```
#endif
#if MODE_FOLLOW_ENABLED == ENABLED
 # include <AP_Follow/AP_Follow.h>
#endif
#if AC_FENCE == ENABLED
 # include <AC_Fence/AC_Fence.h>
#endif
#if AC_TERRAIN == ENABLED
 # include <AP_Terrain/AP_Terrain.h>
#endif
#if OPTFLOW == ENABLED
 # include <AP_OpticalFlow/AP_OpticalFlow.h>
#endif
#if VISUAL_ODOMETRY_ENABLED == ENABLED
 # include <AP_VisualOdom/AP_VisualOdom.h>
#endif
#if RANGEFINDER_ENABLED == ENABLED
 # include <AP_RangeFinder/AP_RangeFinder.h>
#endif
#if PROXIMITY_ENABLED == ENABLED
 # include <AP_Proximity/AP_Proximity.h>
#endif
#if MOUNT == ENABLED
 #include <AP_Mount/AP_Mount.h>
#endif
#if CAMERA == ENABLED
 # include <AP_Camera/AP_Camera.h>
#endif
#if BUTTON_ENABLED == ENABLED
 # include <AP_Button/AP_Button.h>
#endif

#if OSD_ENABLED == ENABLED
 #include <AP_OSD/AP_OSD.h>
#endif

#if ADVANCED_FAILSAFE == ENABLED
 # include "afs_copter.h"
#endif
#if TOY_MODE_ENABLED == ENABLED
 # include "toy_mode.h"
#endif
#if WINCH_ENABLED == ENABLED
 # include <AP_WheelEncoder/AP_WheelEncoder.h>
 # include <AP_Winch/AP_Winch.h>
#endif
#if RPM_ENABLED == ENABLED
```

```
 #include <AP_RPM/AP_RPM.h>
#endif

#ifdef ENABLE_SCRIPTING
#include <AP_Scripting/AP_Scripting.h>
#endif

// Local modules
#ifdef USER_PARAMS_ENABLED
#include "UserParameters.h"
#endif
#include "Parameters.h"
#if ADSB_ENABLED == ENABLED
#include "avoidance_adsb.h"
#endif

#if CONFIG_HAL_BOARD == HAL_BOARD_SITL
#include <SITL/SITL.h>
#endif

#include "mode.h"

class Copter : public AP_Vehicle {
public:
    friend class GCS_MAVLINK_Copter;
    friend class GCS_Copter;
    friend class AP_Rally_Copter;
    friend class Parameters;
    friend class ParametersG2;
    friend class AP_Avoidance_Copter;

#if ADVANCED_FAILSAFE == ENABLED
    friend class AP_AdvancedFailsafe_Copter;
#endif
    friend class AP_Arming_Copter;
    friend class ToyMode;
    friend class RC_Channel_Copter;
    friend class RC_Channels_Copter;

    friend class AutoTune;

    friend class Mode;
    friend class ModeAcro;
    friend class ModeAcro_Heli;
    friend class ModeAltHold;
    friend class ModeAuto;
    friend class ModeAutoTune;
```

```
    friend class ModeAvoidADSB;
    friend class ModeBrake;
    friend class ModeCircle;
    friend class ModeDrift;
    friend class ModeFlip;
    friend class ModeFlowHold;
    friend class ModeFollow;
    friend class ModeGuided;
    friend class ModeLand;
    friend class ModeLoiter;
    friend class ModePosHold;
    friend class ModeRTL;
    friend class ModeSmartRTL;
    friend class ModeSport;
    friend class ModeStabilize;
    friend class ModeStabilize_Heli;
    friend class ModeSystemId;
    friend class ModeThrow;
    friend class ModeZigZag;
    friend class ModeAutorotate;

    Copter(void);

    // HAL::Callbacks implementation.
    void setup() override;
    void loop() override;

private:
    static const AP_FWVersion fwver;

    // key aircraft parameters passed to multiple libraries
    AP_Vehicle::MultiCopter aparm;

    // Global parameters are all contained within the 'g' class.
    Parameters g;
    ParametersG2 g2;

    // main loop scheduler
    AP_Scheduler scheduler{FUNCTOR_BIND_MEMBER(&Copter::fast_loop, void)};

    // used to detect MAVLink acks from GCS to stop compassmot
    uint8_t command_ack_counter;

    // primary input control channels
    RC_Channel *channel_roll;
    RC_Channel *channel_pitch;
    RC_Channel *channel_throttle;
```

```
    RC_Channel *channel_yaw;

    AP_Logger logger;

    // flight modes convenience array
    AP_Int8 *flight_modes;
    const uint8_t num_flight_modes = 6;

    struct RangeFinderState {
        bool enabled:1;
        bool alt_healthy:1; // true if we can trust the altitude from the
rangefinder
        int16_t alt_cm;     // tilt compensated altitude (in cm) from rangefinder
        uint32_t last_healthy_ms;
        LowPassFilterFloat alt_cm_filt; // altitude filter
        int16_t alt_cm_glitch_protected;    // last glitch protected altitude
        int8_t glitch_count;    // non-zero number indicates rangefinder is
glitching
        uint32_t glitch_cleared_ms; // system time glitch cleared
    } rangefinder_state, rangefinder_up_state;

    class SurfaceTracking {
    public:
        // get desired climb rate (in cm/s) to achieve surface tracking
        float adjust_climb_rate(float target_rate);

        // get/set target altitude (in cm) above ground
        bool get_target_alt_cm(float &target_alt_cm) const;
        void set_target_alt_cm(float target_alt_cm);

        // get target and actual distances (in m) for logging purposes
        bool get_target_dist_for_logging(float &target_dist) const;
        float get_dist_for_logging() const;
        void invalidate_for_logging() { valid_for_logging = false; }

        // surface tracking surface
        enum class Surface {
            NONE = 0,
            GROUND = 1,
            CEILING = 2
        };
        // set surface to track
        void set_surface(Surface new_surface);

    private:
        Surface surface = Surface::GROUND;
```

```
        float target_dist_cm;       // desired distance in cm from ground or
ceiling
        uint32_t last_update_ms;    // system time of last update to
target_alt_cm
        uint32_t last_glitch_cleared_ms;    // system time of last handle glitch
recovery
        bool valid_for_logging;     // true if target_alt_cm is valid for logging
        bool reset_target;          // true if target should be reset because of
change in tracking_state
    } surface_tracking;

#if RPM_ENABLED == ENABLED
    AP_RPM rpm_sensor;
#endif

    // Inertial Navigation EKF - different viewpoint
    AP_AHRS_View *ahrs_view;

#if CONFIG_HAL_BOARD == HAL_BOARD_SITL
    SITL::SITL sitl;
#endif

    // Arming/Disarming management class
    AP_Arming_Copter arming;

    // Optical flow sensor
#if OPTFLOW == ENABLED
    OpticalFlow optflow;
#endif

    // system time in milliseconds of last recorded yaw reset from ekf
    uint32_t ekfYawReset_ms;
    int8_t ekf_primary_core;

    // vibration check
    struct {
        bool high_vibes;    // true while high vibration are detected
        uint32_t start_ms;  // system time high vibration were last detected
        uint32_t clear_ms;  // system time high vibrations stopped
    } vibration_check;

    // GCS selection
    GCS_Copter _gcs; // avoid using this; use gcs()
    GCS_Copter &gcs() { return _gcs; }

    // User variables
#ifdef USERHOOK_VARIABLES
```

```
# include USERHOOK_VARIABLES
#endif

    // Documentation of GLobals:
    typedef union {
        struct {
            uint8_t unused1                 : 1; // 0
            uint8_t simple_mode             : 2; // 1,2     // This is the state
of simple mode : 0 = disabled ; 1 = SIMPLE ; 2 = SUPERSIMPLE
            uint8_t pre_arm_rc_check        : 1; // 3       // true if rc input
pre-arm checks have been completed successfully
            uint8_t pre_arm_check           : 1; // 4       // true if all pre-
arm checks (rc, accel calibration, gps lock) have been performed
            uint8_t auto_armed              : 1; // 5       // stops auto
missions from beginning until throttle is raised
            uint8_t logging_started         : 1; // 6       // true if logging
has started
            uint8_t land_complete           : 1; // 7       // true if we have
detected a landing
            uint8_t new_radio_frame         : 1; // 8       // Set true if we
have new PWM data to act on from the Radio
            uint8_t usb_connected_unused    : 1; // 9       // UNUSED
            uint8_t rc_receiver_present     : 1; // 10      // true if we have an
rc receiver present (i.e. if we've ever received an update
            uint8_t compass_mot             : 1; // 11      // true if we are
currently performing compassmot calibration
            uint8_t motor_test              : 1; // 12      // true if we are
currently performing the motors test
            uint8_t initialised             : 1; // 13      // true once the
init_ardupilot function has completed.  Extended status to GCS is not sent until
this completes
            uint8_t land_complete_maybe     : 1; // 14      // true if we may
have landed (less strict version of land_complete)
            uint8_t throttle_zero           : 1; // 15      // true if the
throttle stick is at zero, debounced, determines if pilot intends shut-down when
not using motor interlock
            uint8_t system_time_set_unused  : 1; // 16      // true if the system
time has been set from the GPS
            uint8_t gps_glitching           : 1; // 17      // true if GPS
glitching is affecting navigation accuracy
            uint8_t using_interlock         : 1; // 20      // aux switch motor
interlock function is in use
            uint8_t land_repo_active        : 1; // 21      // true if the pilot
is overriding the landing position
            uint8_t motor_interlock_switch  : 1; // 22      // true if pilot is
requesting motor interlock enable
```

```
            uint8_t in_arming_delay         : 1; // 23      // true while we are
armed but waiting to spin motors
            uint8_t initialised_params      : 1; // 24      // true when the all
parameters have been initialised. we cannot send parameters to the GCS until this
is done
            uint8_t compass_init_location   : 1; // 25      // true when the
compass's initial location has been set
            uint8_t unused2                 : 1; // 26      // aux switch
rc_override is allowed
            uint8_t armed_with_switch       : 1; // 27      // we armed using a
arming switch
        };
        uint32_t value;
    } ap_t;

    ap_t ap;

    static_assert(sizeof(uint32_t) == sizeof(ap), "ap_t must be uint32_t");

    // This is the state of the flight control system
    // There are multiple states defined such as STABILIZE, ACRO,
    Mode::Number control_mode;
    ModeReason control_mode_reason = ModeReason::UNKNOWN;

    Mode::Number prev_control_mode;
    ModeReason prev_control_mode_reason = ModeReason::UNKNOWN;

    RCMapper rcmap;

    // intertial nav alt when we armed
    float arming_altitude_m;

    // Failsafe
    struct {
        uint32_t last_heartbeat_ms;      // the time when the last HEARTBEAT
message arrived from a GCS - used for triggering gcs failsafe
        uint32_t terrain_first_failure_ms;  // the first time terrain data access
failed - used to calculate the duration of the failure
        uint32_t terrain_last_failure_ms;   // the most recent time terrain data
access failed

        int8_t radio_counter;            // number of iterations with throttle
below throttle_fs_value

        uint8_t radio               : 1; // A status flag for the radio failsafe
        uint8_t gcs                 : 1; // A status flag for the ground station
failsafe
```

```
        uint8_t ekf                 : 1; // true if ekf failsafe has occurred
        uint8_t terrain             : 1; // true if the missing terrain data
failsafe has occurred
        uint8_t adsb                : 1; // true if an adsb related failsafe has
occurred
    } failsafe;

    bool any_failsafe_triggered() const {
        return failsafe.radio || battery.has_failsafed() || failsafe.gcs ||
failsafe.ekf || failsafe.terrain || failsafe.adsb;
    }

    // sensor health for logging
    struct {
        uint8_t baro        : 1;    // true if baro is healthy
        uint8_t compass     : 1;    // true if compass is healthy
        uint8_t primary_gps : 2;    // primary gps index
    } sensor_health;

    // Motor Output
    MOTOR_CLASS *motors;
    const struct AP_Param::GroupInfo *motors_var_info;

    int32_t _home_bearing;
    uint32_t _home_distance;

    // SIMPLE Mode
    // Used to track the orientation of the vehicle for Simple mode. This value
is reset at each arming
    // or in SuperSimple mode when the vehicle leaves a 20m radius from home.
    float simple_cos_yaw;
    float simple_sin_yaw;
    int32_t super_simple_last_bearing;
    float super_simple_cos_yaw;
    float super_simple_sin_yaw;

    // Stores initial bearing when armed - initial simple bearing is modified in
super simple mode so not suitable
    int32_t initial_armed_bearing;

    // Battery Sensors
    AP_BattMonitor battery{MASK_LOG_CURRENT,
                           FUNCTOR_BIND_MEMBER(&Copter::handle_battery_failsafe,
void, const char*, const int8_t),
                           _failsafe_priorities};

#if OSD_ENABLED == ENABLED
```

```
    AP_OSD osd;
#endif

    // Altitude
    int32_t baro_alt;            // barometer altitude in cm above home
    LowPassFilterVector3f land_accel_ef_filter; // accelerations for land and
crash detector tests

    // filtered pilot's throttle input used to cancel landing if throttle held
high
    LowPassFilterFloat rc_throttle_control_in_filter;

    // 3D Location vectors
    // Current location of the vehicle (altitude is relative to home)
    Location current_loc;

    // IMU variables
    // Integration time (in seconds) for the gyros (DCM algorithm)
    // Updated with the fast loop
    float G_Dt;

    // Inertial Navigation
    AP_InertialNav_NavEKF inertial_nav;

    // Attitude, Position and Waypoint navigation objects
    // To-Do: move inertial nav up or other navigation variables down here
    AC_AttitudeControl_t *attitude_control;
    AC_PosControl *pos_control;
    AC_WPNav *wp_nav;
    AC_Loiter *loiter_nav;

#if MODE_CIRCLE_ENABLED == ENABLED
    AC_Circle *circle_nav;
#endif

    // System Timers
    // --------------
    // arm_time_ms - Records when vehicle was armed. Will be Zero if we are
disarmed.
    uint32_t arm_time_ms;

    // Used to exit the roll and pitch auto trim function
    uint8_t auto_trim_counter;

    // Camera
#if CAMERA == ENABLED
    AP_Camera camera{MASK_LOG_CAMERA, current_loc};
```

```
#endif

    // Camera/Antenna mount tracking and stabilisation stuff
#if MOUNT == ENABLED
    AP_Mount camera_mount;
#endif

    // AC_Fence library to reduce fly-aways
#if AC_FENCE == ENABLED
    AC_Fence fence;
#endif

#if AC_AVOID_ENABLED == ENABLED
    AC_Avoid avoid;
#endif

    // Rally library
#if AC_RALLY == ENABLED
    AP_Rally_Copter rally;
#endif

    // Crop Sprayer
#if SPRAYER_ENABLED == ENABLED
    AC_Sprayer sprayer;
#endif

    // Parachute release
#if PARACHUTE == ENABLED
    AP_Parachute parachute{relay};
#endif

    // Landing Gear Controller
    AP_LandingGear landinggear;

    // terrain handling
#if AP_TERRAIN_AVAILABLE && AC_TERRAIN && MODE_AUTO_ENABLED == ENABLED
    AP_Terrain terrain{mode_auto.mission};
#endif

    // Precision Landing
#if PRECISION_LANDING == ENABLED
    AC_PrecLand precland;
#endif

    // Pilot Input Management Library
    // Only used for Helicopter for now
#if FRAME_CONFIG == HELI_FRAME
```

```
    AC_InputManager_Heli input_manager;
#endif

#if ADSB_ENABLED == ENABLED
    AP_ADSB adsb;

    // avoidance of adsb enabled vehicles (normally manned vehicles)
    AP_Avoidance_Copter avoidance_adsb{adsb};
#endif

    // last valid RC input time
    uint32_t last_radio_update_ms;

    // last esc calibration notification update
    uint32_t esc_calibration_notify_update_ms;

    // Top-level logic
    // setup the var_info table
    AP_Param param_loader;

#if FRAME_CONFIG == HELI_FRAME
    // Mode filter to reject RC Input glitches.  Filter size is 5, and it draws
the 4th element, so it can reject 3 low glitches,
    // and 1 high glitch.  This is because any "off" glitches can be highly
problematic for a helicopter running an ESC
    // governor.  Even a single "off" frame can cause the rotor to slow
dramatically and take a long time to restart.
    ModeFilterInt16_Size5 rotor_speed_deglitch_filter {4};

    // Tradheli flags
    typedef struct {
        uint8_t dynamic_flight          : 1;    // 0   // true if we are moving
at a significant speed (used to turn on/off leaky I terms)
        uint8_t inverted_flight         : 1;    // 1   // true for inverted
flight mode
        uint8_t in_autorotation         : 1;    // 2   // true when heli is in
autorotation
    } heli_flags_t;
    heli_flags_t heli_flags;

    int16_t hover_roll_trim_scalar_slew;
#endif

    // ground effect detector
    struct {
        bool takeoff_expected;
        bool touchdown_expected;
```

```
        uint32_t takeoff_time_ms;
        float takeoff_alt_cm;
    } gndeffect_state;

    bool standby_active;

    // set when we are upgrading parameters from 3.4
    bool upgrading_frame_params;

    static const AP_Scheduler::Task scheduler_tasks[];
    static const AP_Param::Info var_info[];
    static const struct LogStructure log_structure[];

    // enum for ESC CALIBRATION
    enum ESCCalibrationModes : uint8_t {
        ESCCAL_NONE = 0,
        ESCCAL_PASSTHROUGH_IF_THROTTLE_HIGH = 1,
        ESCCAL_PASSTHROUGH_ALWAYS = 2,
        ESCCAL_AUTO = 3,
        ESCCAL_DISABLED = 9,
    };

    enum Failsafe_Action {
        Failsafe_Action_None           = 0,
        Failsafe_Action_Land           = 1,
        Failsafe_Action_RTL            = 2,
        Failsafe_Action_SmartRTL       = 3,
        Failsafe_Action_SmartRTL_Land  = 4,
        Failsafe_Action_Terminate      = 5
    };

    enum class FailsafeOption {
        RC_CONTINUE_IF_AUTO             = (1<<0),   // 1
        GCS_CONTINUE_IF_AUTO            = (1<<1),   // 2
        RC_CONTINUE_IF_GUIDED           = (1<<2),   // 4
        CONTINUE_IF_LANDING             = (1<<3),   // 8
        GCS_CONTINUE_IF_PILOT_CONTROL   = (1<<4),   // 16
    };

    static constexpr int8_t _failsafe_priorities[] = {
                                                        Failsafe_Action_Terminate,
                                                        Failsafe_Action_Land,
                                                        Failsafe_Action_RTL,

Failsafe_Action_SmartRTL_Land,
                                                        Failsafe_Action_SmartRTL,
                                                        Failsafe_Action_None,
```

```
                                                    -1 // the priority list
must end with a sentinel of -1
                                                };

    #define FAILSAFE_LAND_PRIORITY 1
    static_assert(_failsafe_priorities[FAILSAFE_LAND_PRIORITY] ==
Failsafe_Action_Land,
                  "FAILSAFE_LAND_PRIORITY must match the entry in
_failsafe_priorities");
    static_assert(_failsafe_priorities[ARRAY_SIZE(_failsafe_priorities) - 1] == -
1,
                  "_failsafe_priorities is missing the sentinel");



    // AP_State.cpp
    void set_auto_armed(bool b);
    void set_simple_mode(uint8_t b);
    void set_failsafe_radio(bool b);
    void set_failsafe_gcs(bool b);
    void update_using_interlock();

    // ArduCopter.cpp
    void fast_loop();
    void rc_loop();
    void throttle_loop();
    void update_batt_compass(void);
    void fourhundred_hz_logging();
    void ten_hz_logging_loop();
    void twentyfive_hz_logging();
    void three_hz_loop();
    void one_hz_loop();
    void update_GPS(void);
    void init_simple_bearing();
    void update_simple_mode(void);
    void update_super_simple_bearing(bool force_update);
    void read_AHRS(void);
    void update_altitude();

    // Attitude.cpp
    float get_pilot_desired_yaw_rate(int16_t stick_angle);
    void update_throttle_hover();
    void set_throttle_takeoff();
    float get_pilot_desired_climb_rate(float throttle_control);
    float get_non_takeoff_throttle();
    float get_avoidance_adjusted_climbrate(float target_rate);
    void set_accel_throttle_I_from_pilot_throttle();
```

```
    void rotate_body_frame_to_NE(float &x, float &y);
    uint16_t get_pilot_speed_dn();

#if ADSB_ENABLED == ENABLED
    // avoidance_adsb.cpp
    void avoidance_adsb_update(void);
#endif

    // baro_ground_effect.cpp
    void update_ground_effect_detector(void);

    // commands.cpp
    void update_home_from_EKF();
    void set_home_to_current_location_inflight();
    bool set_home_to_current_location(bool lock) WARN_IF_UNUSED;
    bool set_home(const Location& loc, bool lock) WARN_IF_UNUSED;
    bool far_from_EKF_origin(const Location& loc);

    // compassmot.cpp
    MAV_RESULT mavlink_compassmot(const GCS_MAVLINK &gcs_chan);

    // crash_check.cpp
    void crash_check();
    void thrust_loss_check();
    void parachute_check();
    void parachute_release();
    void parachute_manual_release();

    // ekf_check.cpp
    void ekf_check();
    bool ekf_over_threshold();
    void failsafe_ekf_event();
    void failsafe_ekf_off_event(void);
    void check_ekf_reset();
    void check_vibration();

    // esc_calibration.cpp
    void esc_calibration_startup_check();
    void esc_calibration_passthrough();
    void esc_calibration_auto();
    void esc_calibration_notify();
    void esc_calibration_setup();

    // events.cpp
    bool failsafe_option(FailsafeOption opt) const;
    void failsafe_radio_on_event();
    void failsafe_radio_off_event();
```

```
    void handle_battery_failsafe(const char* type_str, const int8_t action);
    void failsafe_gcs_check();
    void failsafe_gcs_on_event(void);
    void failsafe_gcs_off_event(void);
    void failsafe_terrain_check();
    void failsafe_terrain_set_status(bool data_ok);
    void failsafe_terrain_on_event();
    void gpsglitch_check();
    void set_mode_RTL_or_land_with_pause(ModeReason reason);
    void set_mode_SmartRTL_or_RTL(ModeReason reason);
    void set_mode_SmartRTL_or_land_with_pause(ModeReason reason);
    bool should_disarm_on_failsafe();
    void do_failsafe_action(Failsafe_Action action, ModeReason reason);

    // failsafe.cpp
    void failsafe_enable();
    void failsafe_disable();
#if ADVANCED_FAILSAFE == ENABLED
    void afs_fs_check(void);
#endif

    // fence.cpp
    void fence_check();

    // heli.cpp
    void heli_init();
    void check_dynamic_flight(void);
    bool should_use_landing_swash() const;
    void update_heli_control_dynamics(void);
    void heli_update_landing_swash();
    void heli_update_rotor_speed_targets();
    void heli_update_autorotation();
#if MODE_AUTOROTATE_ENABLED == ENABLED
    void heli_set_autorotation(bool autotrotation);
#endif
    // inertia.cpp
    void read_inertia();

    // landing_detector.cpp
    void update_land_and_crash_detectors();
    void update_land_detector();
    void set_land_complete(bool b);
    void set_land_complete_maybe(bool b);
    void update_throttle_thr_mix();

    // landing_gear.cpp
    void landinggear_update();
```

```
    // standby.cpp
    void standby_update();

    // Log.cpp
    void Log_Write_Control_Tuning();
    void Log_Write_Performance();
    void Log_Write_Attitude();
    void Log_Write_EKF_POS();
    void Log_Write_MotBatt();
    void Log_Write_Data(LogDataID id, int32_t value);
    void Log_Write_Data(LogDataID id, uint32_t value);
    void Log_Write_Data(LogDataID id, int16_t value);
    void Log_Write_Data(LogDataID id, uint16_t value);
    void Log_Write_Data(LogDataID id, float value);
    void Log_Write_Parameter_Tuning(uint8_t param, float tuning_val, float
tune_min, float tune_max);
    void Log_Sensor_Health();
#if FRAME_CONFIG == HELI_FRAME
    void Log_Write_Heli(void);
#endif
    void Log_Write_Precland();
    void Log_Write_GuidedTarget(uint8_t target_type, const Vector3f& pos_target,
const Vector3f& vel_target);
    void Log_Write_SysID_Setup(uint8_t systemID_axis, float waveform_magnitude,
float frequency_start, float frequency_stop, float time_fade_in, float
time_const_freq, float time_record, float time_fade_out);
    void Log_Write_SysID_Data(float waveform_time, float waveform_sample, float
waveform_freq, float angle_x, float angle_y, float angle_z, float accel_x, float
accel_y, float accel_z);
    void Log_Write_Vehicle_Startup_Messages();
    void log_init(void);

    // mode.cpp
    bool set_mode(Mode::Number mode, ModeReason reason);
    bool set_mode(const uint8_t new_mode, const ModeReason reason) override;
    void update_flight_mode();
    void notify_flight_mode();

    // mode_land.cpp
    void set_mode_land_with_pause(ModeReason reason);
    bool landing_with_GPS();

    // motor_test.cpp
    void motor_test_output();
    bool mavlink_motor_test_check(const GCS_MAVLINK &gcs_chan, bool check_rc);
```

```
    MAV_RESULT mavlink_motor_test_start(const GCS_MAVLINK &gcs_chan, uint8_t
motor_seq, uint8_t throttle_type, uint16_t throttle_value, float timeout_sec,
uint8_t motor_count);
    void motor_test_stop();

    // motors.cpp
    void arm_motors_check();
    void auto_disarm_check();
    void motors_output();
    void lost_vehicle_check();

    // navigation.cpp
    void run_nav_updates(void);
    int32_t home_bearing();
    uint32_t home_distance();

    // Parameters.cpp
    void load_parameters(void);
    void convert_pid_parameters(void);
    void convert_lgr_parameters(void);
    void convert_tradheli_parameters(void);
    void convert_fs_options_params(void);

    // precision_landing.cpp
    void init_precland();
    void update_precland();

    // radio.cpp
    void default_dead_zones();
    void init_rc_in();
    void init_rc_out();
    void enable_motor_output();
    void read_radio();
    void set_throttle_and_failsafe(uint16_t throttle_pwm);
    void set_throttle_zero_flag(int16_t throttle_control);
    void radio_passthrough_to_motors();
    int16_t get_throttle_mid(void);

    // sensors.cpp
    void read_barometer(void);
    void init_rangefinder(void);
    void read_rangefinder(void);
    bool rangefinder_alt_ok();
    bool rangefinder_up_ok();
    void rpm_update();
    void init_optflow();
    void update_optical_flow(void);
```

```
void compass_cal_update(void);
void accel_cal_update(void);
void init_proximity();
void update_proximity();
void init_visual_odom();
void winch_init();
void winch_update();

// setup.cpp
void report_compass();
void print_blanks(int16_t num);
void print_divider(void);
void print_enabled(bool b);
void report_version();

// switches.cpp
void read_control_switch();
void save_trim();
void auto_trim();

// system.cpp
void init_ardupilot();
void startup_INS_ground();
void update_dynamic_notch();
bool position_ok() const;
bool ekf_position_ok() const;
bool optflow_position_ok() const;
void update_auto_armed();
bool should_log(uint32_t mask);
MAV_TYPE get_frame_mav_type();
const char* get_frame_string();
void allocate_motors(void);
bool is_tradheli() const;

// terrain.cpp
void terrain_update();
void terrain_logging();
bool terrain_use();

// tuning.cpp
void tuning();

// UserCode.cpp
void userhook_init();
void userhook_FastLoop();
void userhook_50Hz();
void userhook_MediumLoop();
```

```
    void userhook_SlowLoop();
    void userhook_SuperSlowLoop();
    void userhook_auxSwitch1(uint8_t ch_flag);
    void userhook_auxSwitch2(uint8_t ch_flag);
    void userhook_auxSwitch3(uint8_t ch_flag);

#if OSD_ENABLED == ENABLED
    void publish_osd_info();
#endif

    Mode *flightmode;
#if MODE_ACRO_ENABLED == ENABLED
#if FRAME_CONFIG == HELI_FRAME
    ModeAcro_Heli mode_acro;
#else
    ModeAcro mode_acro;
#endif
#endif
    ModeAltHold mode_althold;
#if MODE_AUTO_ENABLED == ENABLED
    ModeAuto mode_auto;
#endif
#if AUTOTUNE_ENABLED == ENABLED
    AutoTune autotune;
    ModeAutoTune mode_autotune;
#endif
#if MODE_BRAKE_ENABLED == ENABLED
    ModeBrake mode_brake;
#endif
#if MODE_CIRCLE_ENABLED == ENABLED
    ModeCircle mode_circle;
#endif
#if MODE_DRIFT_ENABLED == ENABLED
    ModeDrift mode_drift;
#endif
#if MODE_FLIP_ENABLED == ENABLED
    ModeFlip mode_flip;
#endif
#if MODE_FOLLOW_ENABLED == ENABLED
    ModeFollow mode_follow;
#endif
#if MODE_GUIDED_ENABLED == ENABLED
    ModeGuided mode_guided;
#endif
    ModeLand mode_land;
#if MODE_LOITER_ENABLED == ENABLED
    ModeLoiter mode_loiter;
```

```
#endif
#if MODE_POSHOLD_ENABLED == ENABLED
    ModePosHold mode_poshold;
#endif
#if MODE_RTL_ENABLED == ENABLED
    ModeRTL mode_rtl;
#endif
#if FRAME_CONFIG == HELI_FRAME
    ModeStabilize_Heli mode_stabilize;
#else
    ModeStabilize mode_stabilize;
#endif
#if MODE_SPORT_ENABLED == ENABLED
    ModeSport mode_sport;
#endif
#if MODE_SYSTEMID_ENABLED == ENABLED
    ModeSystemId mode_systemid;
#endif
#if ADSB_ENABLED == ENABLED
    ModeAvoidADSB mode_avoid_adsb;
#endif
#if MODE_THROW_ENABLED == ENABLED
    ModeThrow mode_throw;
#endif
#if MODE_GUIDED_NOGPS_ENABLED == ENABLED
    ModeGuidedNoGPS mode_guided_nogps;
#endif
#if MODE_SMARTRTL_ENABLED == ENABLED
    ModeSmartRTL mode_smartrtl;
#endif
#if !HAL_MINIMIZE_FEATURES && OPTFLOW == ENABLED
    ModeFlowHold mode_flowhold;
#endif
#if MODE_ZIGZAG_ENABLED == ENABLED
    ModeZigZag mode_zigzag;
#endif
#if MODE_AUTOROTATE_ENABLED == ENABLED
    ModeAutorotate mode_autorotate;
#endif

    // mode.cpp
    Mode *mode_from_mode_num(const Mode::Number mode);
    void exit_mode(Mode *&old_flightmode, Mode *&new_flightmode);

public:
    void mavlink_delay_cb();    // GCS_Mavlink.cpp
    void failsafe_check();      // failsafe.cpp
```

```
};

extern Copter copter;

using AP_HAL::millis;
using AP_HAL::micros;
```

10.2. Modo de vuelo Stabilize

```
#include
"Copter.h"

            /*
             * Init and run calls for stabilize flight mode
             */

            // stabilize_run - runs the main stabilize controller
            // should be called at 100hz or more
            void ModeStabilize::run()
            {
                // apply simple mode transform to pilot inputs
                update_simple_mode();

                // convert pilot input to lean angles
                float target_roll, target_pitch;
                get_pilot_desired_lean_angles(target_roll, target_pitch,
            copter.aparm.angle_max, copter.aparm.angle_max);

                // get pilot's desired yaw rate
                float target_yaw_rate = get_pilot_desired_yaw_rate(channel_yaw-
            >get_control_in());

                if (!motors->armed()) {
                    // Motors should be Stopped
                    motors-
            >set_desired_spool_state(AP_Motors::DesiredSpoolState::SHUT_DOWN);
                } else if (copter.ap.throttle_zero) {
                    // Attempting to Land
                    motors-
            >set_desired_spool_state(AP_Motors::DesiredSpoolState::GROUND_IDLE);
                } else {
                    motors-
            >set_desired_spool_state(AP_Motors::DesiredSpoolState::THROTTLE_UNLIMITED);
                }

                switch (motors->get_spool_state()) {
                case AP_Motors::SpoolState::SHUT_DOWN:
                    // Motors Stopped
```

```
            attitude_control->set_yaw_target_to_current_heading();
            attitude_control->reset_rate_controller_I_terms();
            break;

        case AP_Motors::SpoolState::GROUND_IDLE:
            // Landed
            attitude_control->set_yaw_target_to_current_heading();
            attitude_control->reset_rate_controller_I_terms();
            break;

        case AP_Motors::SpoolState::THROTTLE_UNLIMITED:
            // clear landing flag above zero throttle
            if (!motors->limit.throttle_lower) {
                set_land_complete(false);
            }
            break;

        case AP_Motors::SpoolState::SPOOLING_UP:
        case AP_Motors::SpoolState::SPOOLING_DOWN:
            // do nothing
            break;
        }

        // call attitude controller
        attitude_control-
>input_euler_angle_roll_pitch_euler_rate_yaw(target_roll, target_pitch,
target_yaw_rate);

        // output pilot's throttle
        attitude_control->set_throttle_out(get_pilot_desired_throttle(),
                                           true,
                                           g.throttle_filt);
}
```