Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Diseño y construcción de un dron para el seguimiento de un objeto de forma autónoma mediante visión artificial.

Autor: Francisco Ciudad Fernández

Tutor: D. Pedro Alfonso Revenga de Toro

2017

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Diseño y construcción de un dron para el seguimiento de un objeto de forma autónoma mediante visión artificial.

Autor: Francisco Ciudad Fernández

Director: D. Pedro Alfonso Revenga de Toro

Tribunal:

Presidente: D. Luis Miguel Bergasa Pascual

Vocal 1º: D. Alfredo Gardel Vicente

Vocal 2º: D. Pedro Alfonso Revenga de Toro

Calificación: .....

Fecha: .....

Resumen

En el presente documento se describe el proceso llevado a cabo para diseñar y construir un Dron cuatrirrotor ligero robotizado y la electrónica asociada para que siga objetos mediante visión artificial. Para ello, se han utilizado dos sistemas interconectados, el sistema de control de vuelo del don, con el software *LibrePilot* y una placa procesadora de imagen que suministra los comandos de alto nivel para el seguimiento de los objetos, con *GNU/Linux* y *OpenCV*. Se ha modificado el sistema *LibrePilot* para interconectar este sistema de control del dron con el sistema de procesamiento de imagen. Se han desarrollado también, los programas de procesamiento de imágenes y de seguimiento de objetos en tiempo real en la placa procesadora de imagen, *NanoPi Neo Air*, con *GNU/Linux* y *OpenCv*.

Palabras clave: Dron, UAV, Librepilot, GNU/Linux, OpenCv.

Abstract

This paper describes the process carried out to design and build a light four-rotor robotic drone and its associate electronics to follow objects through artificial vision. For this to happen, two interconnected systems have been used, the drone flight control system which consists of the *LibrePilot* software, and an image processing board to provide the high level commands for tracking of objects with the use of *GNU / Linux and OpenCv*. The *LibrePilot* system has been modified to interconnect this drone control system with the image processing system. Image processing and object tracking programs have also been developed in real time on the image processing board, *NanoPi Neo Air*, with GNU / Linux and OpenCv.

Keywords: Dron, UAV, Librepilot, GNU/Linux, OpenCv.

Resumen extendido

Desde hace unos años, y propiciados por el continuo desarrollo de la electrónica, se han puesto de moda los ***V**ehículos **A**éreos no **T**ripulados* (VANT). También se les denomina UAV por sus siglas en inglés ***U**nmanned **A**erial **V**ehicle* o Drones.

Todos estos nombres definen un mismo concepto: Un vehículo aéreo que no tiene piloto en su interior. Las habilidades y sentidos que usan los pilotos son sustituidos por sensores electrónicos como los acelerómetros y giróscopos, de los que hablaremos más adelante. La principal ventaja que suponen los drones es la inexistencia de pilotos en su interior lo que reduce el riesgo de muerte de dichas personas y permite así, realizar funciones que no serían posibles con aeronaves tripuladas como por ejemplo la investigación en zonas de alta toxicidad química y radiológica.

En un principio, estos drones estaban pilotados por una persona desde un lugar remoto, teniendo acceso a la mayoría de los datos de aviación que tendría si estuviese físicamente en éste. Actualmente este sistema de control sigue estando presente aunque va proliferando cada vez más el control autónomo de dicho dispositivo. Para este último objetivo se utilizan diferentes sensores tales como el GPS, sensores de proximidad, barómetros y cámaras entre otros. En nuestro caso, usaremos los últimos tres sensores citados anteriormente y dejaremos de lado al GPS debido a que nuestra aplicación se debe poder utilizar tanto en espacios abiertos como cerrados.

En este proyecto lo que se va a realizar es el diseño y la construcción de un dron con capacidad para soportar una cámara con la que se va a llevar a cabo el procesamiento de las imágenes que ésta capte. Esta cámara no tiene por qué ser de gran resolución ya que la usaremos para la detección de un objeto y no para grabar vídeos en alta resolución. Además de la cámara, necesitamos otros componentes tales como: el chasis, los motores, los reguladores de velocidad o ESCs, la controladora de vuelo, las hélices, unas baterías LiPo (*Polímeros de Litio*) y algunos sensores. Todos estos componentes los describiremos más adelante en este documento.

El control básico del dron lo realizaremos mediante la controladora de vuelo *Mini CC3D Revolution* que ha sido elegida por proporcionar vuelos muy suaves y estables gracias a su controlador STM de 32 bits. Para programar esta placa elegimos el proyecto *Librepilot* debido a que es un proyecto de código abierto y por lo cuál, podemos modificarlo para que cumpla con nuestras especificaciones.

Para el procesamiento de la imagen elegimos el *NanoPi Neo Air*, otro micro-controlador que es capaz de comunicarse con la controladora de vuelo y que realiza el procesamiento de las imágenes necesario para nuestra aplicación.

El dron tendrá un modo de funcionamiento manual, a partir del cual, cualquier persona podrá pilotarlo mediante un sistema de radio-control. En este modo de funcionamiento el usuario deberá dirigir el dron en todo momento ya que éste no dispondrá de autonomía de control.

Por otro lado, tendremos otro modo de funcionamiento autónomo en el que el propio dron seguirá a una persona o a un objeto que hayamos seleccionado anteriormente. En este caso no será necesario

ningún sistema de radio-control, sino que lo podremos manejar simplemente con una pequeña pelota que queramos que siga o cualquier otro objeto, siempre y cuando, lo hayamos configurado para tal fin.

Para conseguir el objetivo anterior debemos hacer que, en primer lugar, la aeronave se estabilice y se mantenga a una cierta altura, y en segundo lugar, conseguir que el dron siga al objeto deseado. La altura se puede adecuar a las necesidades que nosotros queramos, y por lo tanto, la podremos ajustar en cualquier momento del vuelo, aunque normalmente, se adecuará a la altura que tenga el objeto al que deba seguir. El dron conseguirá mantener la altura gracias a la utilización conjunta de dos sensores: un sonar y un barómetro. El primero de ellos medirá la distancia existente entre el dispositivo y el suelo, obteniendo resultados bastante acertados dentro de un rango (0-4 metros). El segundo mide la presión atmosférica existente en un instante y mediante una transformación, obtenemos la altura sobre el nivel del mar en la que nos encontramos. Para poder seguir al objeto deseado usaremos una cámara con la que captaremos las imágenes y las procesaremos con un micro-controlador En nuestro caso, segmentaremos el color rojo de las imágenes obtenidas y obtendremos así las coordenadas X, Y y el radio de la pelota de color rojo que estemos usando. Estos datos los pasaremos a la controladora de vuelo mediante una comunicación *UART* y haremos que las variables que controlan el movimiento del dron se modifiquen de manera proporcional a lo que se mueve la pelota ante el dispositivo.

Por último, comentar que este proyecto se va a realizar dejando abierta la integración de distintas funcionalidades en un futuro, por lo que el desarrollo no se tiene porque encasillar en una aplicación concreta sino que en adelante se continuará avanzando y por ello debemos dar la oportunidad a ese avance.

Lo anteriormente descrito se ha estructurado en diferentes capítulos en los que se detallan los pasos que se han llevado a cabo para la creación de este proyecto.

Índice general

Índice de figuras

Índice de tablas

Memoria

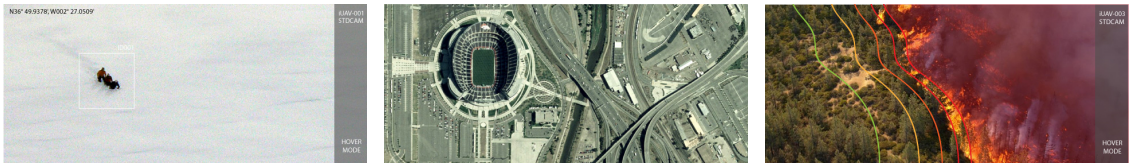
Capítulo 1

Introducción

En estos últimos años, el desarrollo de vehículos aéreos no tripulados (del inglés UAV - ***U**nmanned **A**erial **V**ehicle*) o comúnmente conocidos como Drones, ha tenido un avance muy significativo, sobre todo para el uso en aplicaciones civiles. Las aplicaciones usuales de este tipo de vehículo se centran en tareas donde tienen entornos de difícil acceso o que representa algún peligro, como, por ejemplo: tareas de vigilancia y seguimiento, reconocimiento geográfico, etc.

El desarrollo de estos sistemas ha sido posible gracias al desarrollo de micro-controladores, que ofrecen la posibilidad de hacer cálculos muy complejos en poco espacio de tiempo, a la miniaturización de los sensores y a las mejoras en los sistemas de almacenamiento de energía. Si a estos avances se les añaden las diversas aplicaciones que tienen estas plataformas robóticas, se abre un gran abanico de diversos proyectos con estos dispositivos. En la figura 1.1 se pueden observar unos ejemplos de aplicaciones con drones hoy en día [1]. Una de las más importantes es la de la búsqueda de personas desaparecidas en lugares abiertos o de difícil acceso como zonas montañosas. El reducido tamaño de los UAVs permite tenerlos siempre disponibles en estaciones de montaña, reduciendo considerablemente el tiempo de búsqueda. El bajo coste de estos aparatos comparados con el coste de un helicóptero tradicional los hacen idóneos para esta tarea.

Gracias a los sensores inerciales y a la posibilidad de introducir dispositivo GPS, estos UAVs son capaces de realizar vuelos exteriores autónomos, es decir, sin la intervención humana. Desafortunadamente, los entornos internos aún no tienen acceso a dichos sistemas de localización externos. Esto se ha podido ir solucionando con la inclusión de diversos sensores que deben ser cuidadosamente elegidos debido a los rangos de medida de éstos y de las limitaciones de carga útil del dron. En muchas tareas de navegación es muy importante que el sistema conozca su localización y consiga un control adecuado, por ello, la elección de los tipos de sensores a utilizar es de gran importancia.



(a) Búsqueda de personas desaparecidas.

(b) Fotografía, Vídeo y Cartografía Aérea.

(c) Prevención y control de incendios.

Figura 1.1: Aplicaciones con drones.

1.1 Objetivos del proyecto

El objetivo general de este proyecto es diseñar y construir un cuatrirotor ligero robotizado y la electrónica asociada para que siga objetos mediante visión artificial. El sistema constará de cuatro rotores que estarán controlados desde un micro-controlador el cuál a su vez recibirá información de los sensores necesarios para una correcta estabilización de vuelo del cuadricóptero.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- Estudio de las características básicas de los drones y de su funcionamiento.
- Elección de los materiales necesarios.
- Diseño y construcción de la aeronave.
- Estudio del proyecto *LibrePilot* para posteriormente realizar las oportunas modificaciones.
- Diseño de los algoritmos de control.
- Configuración de la controladora de vuelo.
- Integración y configuración del sensor de ultrasonidos en la controladora.
- Estudio de la librería *OpenCV* para realizar el procesamiento de las imágenes que se capten con la cámara.
- Integración de la librería *OpenCV* en la controladora *NanoPi Neo Air*.
- Integración y configuración de la cámara en la controladora anteriormente citada.
- Procesamiento de las imágenes captadas y obtención de los parámetros que definen la posición del objeto a seguir en cada instante.
- Habilitación de una comunicación para transmitir datos de un microprocesador a otro.
- **Pruebas finales**: Una vez que estén todos los dispositivos funcionando, se llevará a cabo diferentes pruebas con las que se probarán si todos los dispositivos funcionan correctamente de forma conjunta.

Por otro lado, se plantean otros objetivos personales conseguidos con este proyecto:

- Aprendizaje sobre vehículos RC (*control remoto*).
- Afrontar un problema que combine hardware + software.
- Profundizar los conocimientos de electrónica de los que dispongo.
- Aprendizaje de las librerías *OpenCv* y del proyecto *LibrePilot*.

1.2 Estructura del documento

Este documento está dividido en varias secciones, de las cuales la presente introducción es la primera. En ésta se desarrolla el contexto en el que nos encontramos y los objetivos de este proyecto.

En el capítulo 2 se hablará de los drones más conocidos en la actualidad y de sus características más importantes. Además, se describirá la dinámica de los multirrotores y se hará una descripción detallada

del proyecto *LibrePilot* y de la librería *OpenCV* que utilizaremos para realizar el procesamiento de las imágenes.

En el capítulo 3 se describirán los micro-controladores utilizados, sus características y los diferentes sensores que se han acoplado a ellos para que el conjunto funcione correctamente. Además, se explicará el montaje que ha sido necesario, los componentes que se han utilizado y la implementación del algoritmo necesario a nuestro dispositivo para obtener los resultados deseados.

Las conclusiones del proyecto las recogeremos en el capítulo 4. En los siguientes capítulos incluiremos un manual de usuario en el que se explicarán los pasos necesarios para que se pueda reproducir el proyecto por cualquier lector, planos y diagramas, un pliego de condiciones en el que se explicarán las condiciones de materiales, y por último, el presupuesto.

Pata finalizar incluiremos una lista de la bibliografía utilizada.

Capítulo 2

Estudio teórico

2.1 Introducción

En este capítulo se hablará de algunos de los drones profesionales más utilizados en la actualidad y de las características por las que llaman la atención. En otro apartado se describirá la dinámica de los multirrotores y con ello, dar a entender un poco mejor como estas aeronaves mantienen vuelos estables y consiguen realizar los movimientos que nosotros deseemos. Además, se explicará el proyecto *LibrePilot* y la librería *OpenCv* con lo que se conseguirá la finalidad de nuestra aplicación.

2.2 Estado del Arte

En la actualidad, el uso de vehículos no tripulados de radio-control está creciendo cada vez más tanto para actividades de ocio como para trabajos de investigación que necesiten mayor conocimiento científico, introduciendo mejoras en hardware y en software.

Existen una gran variedad de proyectos que afrontan el reto de construir un cuatrirrotor. Dependiendo del enfoque desde donde los miremos, se pueden dividir en proyectos de código abierto y código propietario. Existen varias comunidades que desarrollan software de código abierto orientadas a determinadas plataformas. Otras en cambio, no publican su código ni en muchas ocasiones, los esquemas de los componentes hardware que utilizan.

A continuación se describirán las características que presentan algunos de los vehículos aéreos que se utilizan en sectores profesionales como la industria, la seguridad, la agricultura o el mundo audiovisual [2].

2.2.1 Yuneec Typhoon H

Se trata de un hexacóptero (figura 2.1) dotado de un sistema de detección de proximidad ultrasónico para evitar obstáculos y una potente cámara que graba vídeos en 4K y realiza instantáneas en 12 megapíxeles.

El *Typhoon H* resulta de la evolución lógica de los *Typhoon Q500*. Es un multirrotor de seis motores que brinda prestaciones profesionales a un precio muy competitivo. Otra de las características que no se puede olvidar es la facilidad de transporte del *Typhoon H*. Tanto los brazos como el tren de aterrizaje son completamente plegables y hacen posible que el *Typhoon H* quepa en una mochila de reducido tamaño.

Figura 2.1: Yuneec Typhoon H.

2.2.2 DJI Phantom 4

Este dron, que se muestra en la figura 2.2, puede ejercer como dispositivo de uso doméstico pero sus características lo convierten en un dron capacitado para numerosas actividades profesionales, además de la posibilidad de grabar en 360 grados y con resolución 4K. Su diseño pulido, su autonomía de 28 minutos y una velocidad punta de 72 kilómetros por hora, son su carta de presentación, que se ha visto potenciada por su alianza comercial con *Apple*.

DJI[1] ha incorporado nueva tecnología a su flamante dron, que ahora cuenta con un sistema de detección de obstáculos. Dos cámaras frontales captan el entorno, que se analiza con visión artificial mediante un software llamado *OBstacle Sensing System*. Éste permite que los nuevos drones rodeen obstáculos cuando lo necesiten, sin desviarse nunca de su trayectoria previamente definida.

El dron también cuenta con un modo de vuelo autónomo, algo que se pedía a la compañía desde los círculos de pilotos y aficionados. El usuario puede seleccionar desde la aplicación de *DJI* un objeto y el *Phantom 4* lo seguirá, manteniéndolo en su punto de mira. Así, el dispositivo adquiere la capacidad de perseguir objetos móviles, aunque también se puede utilizar para programar vuelos hacia lugares alejados.

Figura 2.2: Phantom 4 dji.

[1] https://www.dji.com/esç

2.2.3 Parrot Bebop 2

El nuevo *Bebop 2* (figura 2.3) es un dron que se caracteriza por un tamaño y unas formas más ajustadas que su versión anterior lo que le permite que el peso sea de 499 gramos y el tiempo de vuelo pueda alcanzar los 25 minutos (gracias también a la batería de ion de litio recargable de 2700 mAh). La velocidad máxima (horizontal) que puede llegar a alcanzar es de 60 km/h. Lleva incorporada una cámara frontal con objetivo de ojo de pez (*fisheye len*) de 14 megapíxeles orientable que permite realizar vídeos y fotografías cenitales. Como viene siendo habitual en este tipo de dispositivos las imágenes que captura el dron pueden verse y gobernarse en tiempo real por parte del usuario.

Figura 2.3: Parrot Bebop 2.

En cuanto a dispositivos de control, el *Bebop 2* sigue los pasos de su antecesor y puede ser pilotado desde un *smartphone/tablet* (a través de la aplicación *FreeFlight 3*) o desde el dispositivo de radio control desarrollado por *Parrot*, el *Skycontroller*. Según indica el fabricante, la conexión *Wi-Fi* garantizaría el control hasta una distancia de 300 metros y el dispositivo RC hasta 2 Km.

2.2.4 Hexo+

Hexo+ (figura 2.4) es un hexacóptero que está tratando de competir con el *3DR Solo*, tanto en características como en precio. Entre las características de este modelo destaca la capacidad de volar de forma completamente autónoma, la consecución de una imagen estable en las grabaciones y los propios modos de grabación: *Follow*, vuelo en 360º, entre otros. Además es compatible con las cámaras *GoPro* y puede alcanzar los 72 kilómetros por hora.

Figura 2.4: Hexo+.

2.2.5 Lily Camera

Lily Camera (figura 2.5) es básicamente una cámara, una cámara que vuela.

Aparte de su asombroso y fácil manejo, tanto en vuelo como cuando es transportado, *Lily* está especialmente concebido para seguir y grabar las actividades del usuario, con mínimos controles y sin necesidad de pilotaje. Destaca por tanto, por su autonomía.

La principal característica de *Lily* radica en su funcionalidad *follow-me* a través de un sistema de guiado portátil que puede llevarse en la muñeca y en dónde se especifican los principales parámetros de vuelo (altitud, velocidad y ángulos). Estos parámetros también pueden definirse vía *app* en nuestro dispositivo móvil.

Lily cuenta con una cámara que permite realizar imágenes de 12 MP y video en HD 1080p a 60 fps. Con un peso de 1,3Kg dispone de una batería que proporciona una duración de vuelo de 20 minutos. Además *Lily* es resistente al agua.

Figura 2.5: Lily Camera.

A continuación se describen algunos proyectos de código abierto que podremos utilizar como referencia para el desarrollo de nuestro proyecto.

2.2.6 PCB Quadcopter

Figura 2.6: PCB Quadcopter.

El *PCB Quadcopter* [3] es un proyecto *open-source* tanto software como hardware desarrollado por un par de aficionados a la electrónica. El objetivo de este proyecto es crear un dron en el que la propia estructura sea la placa electrónica.

En cuanto a las características técnicas destacamos que se trata de un dron que mide de motor a motor 16,5 cm, usa motores Brushless con hélices de 4 pulgadas. Utiliza una batería de 11.1 V de 370 mAh, está controlado por un mando radiocontrol, pesa 138 g y puede volar 8 min autónomamente.

2.2.7 DroneCode

Figura 2.7: Logo DroneCode.

Dronecode [4] es un proyecto de desarrollo de vehículos aéreos no tripulados bajo una plataforma de código abierto. Un proyecto de ámbito colaborativo entre compañías especializadas que reúne y centraliza los proyectos bajo una plataforma común con base en el *Open UAV* de *3D Robotics*.

2.2.8 LibrePilot

Figura 2.8: Logo *LibrePilot*.

Este proyecto de código abierto fue fundado en julio de 2015. Se centra en la investigación y desarrollo de software y hardware para ser utilizado en una gran variedad de aplicaciones. Uno de los objetivos del proyecto es proporcionar un ambiente colaborativo en el que todo aquel que lo desee puede realizar su aportación y con ello, convertirlo en el hogar ideal para el desarrollo de ideas innovadoras.

2.3 Dinámica de los multirrotores

En este tipo de aeronaves los principios físicos que explican su sustentación son algo más sencillos que los de aeronaves de ala fija o aviones. Lo principal es crear una fuerza de empuje en dirección contraria a la gravedad, como se observa en la figura 2.9, que se consigue con rotores a los que se les acoplan unas hélices que dependiendo del tamaño y de la velocidad conseguirán ejercer una determinada fuerza. Cuando todos los rotores producen la misma fuerza de sustentación y además esta fuerza se encuentra en equilibrio con el peso, el dron se mantiene en vuelo estacionario. Cuando este equilibrio se altera porque uno de los rotores se encuentra a diferentes revoluciones que los demás , se produce el balanceo de la aeronave.

El principal problema viene cuando se hace girar todas las hélices en la misma dirección y produce que el torque total haga girar la aeronave de forma descontrolada. Para solucionarlo, se instalan hélices en sentidos de rotación opuestos de forma diametral, es decir, se alternan hélices de giro a derecha con hélices

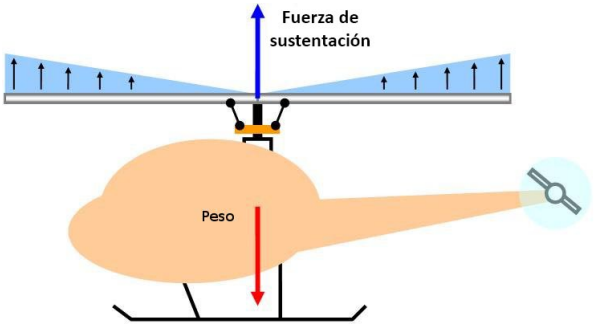


Figura 2.9: Sustentación.

de giro a izquierdas resultando nula la suma de las fuerzas que generan. En la figura 2.10 mostramos los diferentes movimientos que puede hacer un multirotor y la velocidad que debe llevar cada uno de los rotores para conseguir cada uno de esos movimientos [5].

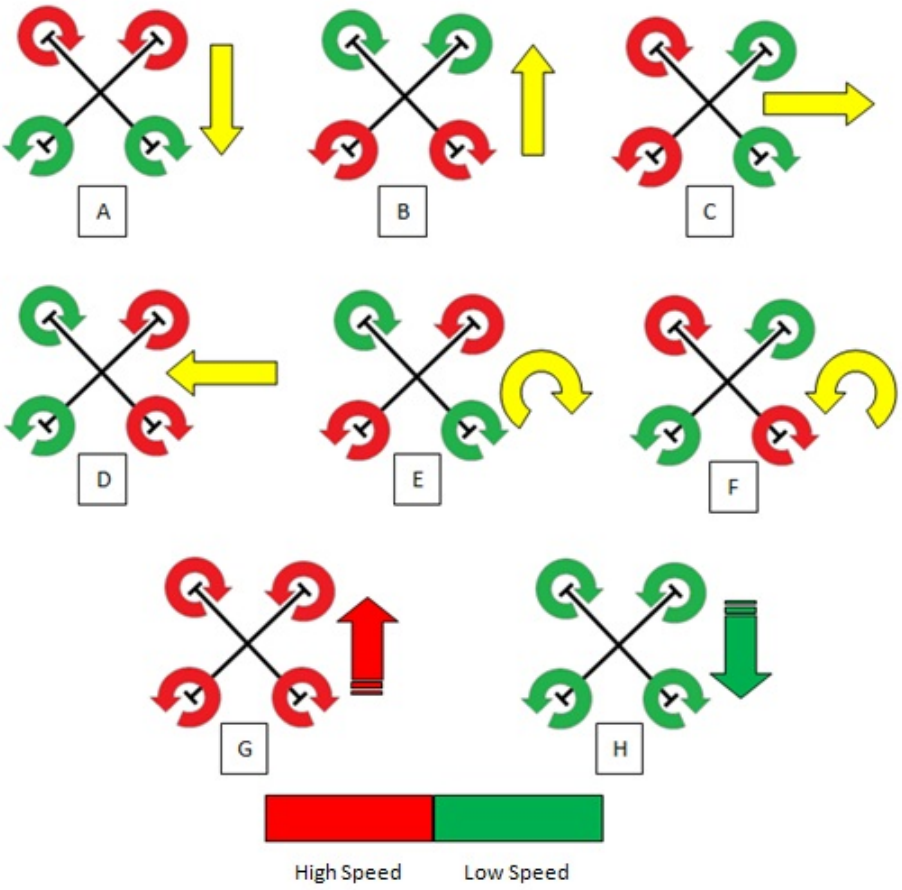


Figura 2.10: Movimientos multirotor.

Como podemos observar en la figura 2.10 existen 3 tipos de giros principales: El alabeo, el cabeceo y la guiñada. Además, todo multirrotor requiere el movimiento de ascenso y descenso de la altura.

- **Altitud**: Es el movimiento más sencillo que puede realizar un cuadricóptero. Ésta se varía aumentando o disminuyendo la velocidad de los 4 rotores por igual. A más velocidad el dron tomará una mayor altitud, y si se disminuye irá hacia abajo.

- **Guiñada**: Es el giro con respecto al eje vertical del aparato. Se consigue por el principio de con-

servación del momento cinético al disponer 2 rotores en cada sentido de giro. Cuando dos rotores que giran en el mismo sentido aceleran, y los otros dos aminoran su velocidad, se aumenta la fuerza de par de los rotores que aceleran y tiene lugar una guiñada hacia el lado contrario al que girar los rotores acelerados. De esta forma, con todos los rotores girando a la misma velocidad, se consigue equilibrar el momento cinético y mantener el aparato recto.

- **Cabeceo y Alabeo**: Para realizar este movimiento se debe hacer balancear el dron en la dirección hacia la que queramos volar. Para llevar a cabo esto, debemos bajar las revoluciones del rotor del lado del aparato que cae y aumentar las revoluciones del rotor del lado que sube. Al girar varios motores con velocidades de giro diferentes se descentra el eje de los mismos con respecto a la posición de equilibrio horizontal produciendo que el empuje tenga una componente en la dirección hacia la que se han desviado.

Como hemos visto en este apartado, es bastante complejo el control de velocidades que se debe tener sobre los 4 motores para conseguir una estabilidad y maniobrabilidad perfecta. La ventaja de controlar todos los movimientos con tan sólo variar 4 valores de tensión tiene la desventaja de tener que ajustar dichos valores con una precisión del orden de milivoltios [6].

2.4 Algoritmo de estabilización. Controlador PID

Los controladores PID son un tipo de estructura de control que calcula la desviación entre un valor medido y un valor al que se quiere llegar, haciendo una corrección. Este tipo de controladores son muy robustos y por eso, son los más utilizados en la mayoría de los procesos industriales en lazo cerrado.

El algoritmo de control calcula tres parámetros diferentes: el proporcional, el integral y el derivativo. El Proporcional, es directamente proporcional al error actual, el Integral hace una corrección del error acumulado en el tiempo (integral del error) y el Derivativo determina la reacción del tiempo en el que el error se produce.

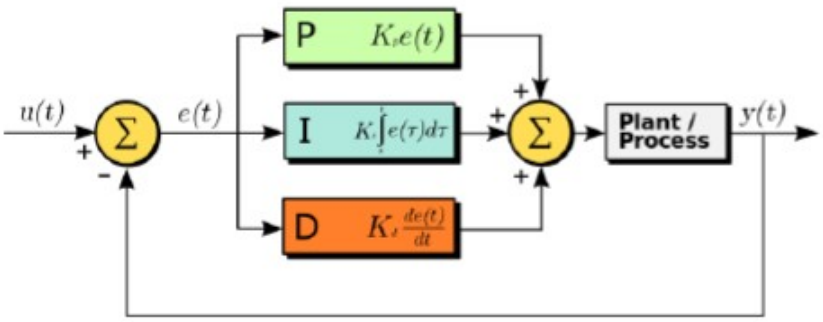


Figura 2.11: Esquema PID.

Las variables más importantes que se manejan en este tipo de control son:

- PV (Process Variable). Valor medido.
- SP (Set Point). Valor deseado.
- MV (Manipulated Variable). Valor de entrada al sistema.
- E (Error). Diferencia entre valor medido(PV) y el deseado(SP).

El valor de entrada al sistema (MV) se calcula por tanto como la suma del valor Proporcional, Integral y Derivado. Es importante decir que Kp Ki y Kd son constantes que debemos ajustar en nuestro sistema.

- El valor **proporcional** es el producto entre la constante proporcional Kp y el error (SP-PV). La parte proporcional no considera el tiempo, por lo tanto, la mejor manera de solucionar el error permanente y hacer que el sistema contenga alguna componente que tenga en cuenta la variación respecto al tiempo, es incluyendo y configurando las acciones integral y derivativa.
- El valor correspondiente al control **Integral**, es proporcional tanto a la magnitud del error, como a la duración del mismo. Es la suma de los errores en el tiempo e indica el cúmulo de errores que tendrían que haberse corregido previamente. Este error acumulado se multiplica por la constante Ki.
- La acción **derivativa** actúa cuando hay un cambio en el valor absoluto del error. La función de la acción derivativa es mantener el error al mínimo corrigiéndolo proporcionalmente con la misma velocidad que se produce, de esta manera evita que el error se incremente

2.5 Entornos de desarrollo para aplicaciones robóticas

Todas las aplicaciones robóticas tienen como objetivo principal el proporcionar una serie de conocimientos o autonomía a un robot. Hoy en día se consiguen soluciones comerciales o de código abierto que permiten realizar esta tarea fácilmente mediante librerías y módulos de trabajo.

La comunidad robótica comenzó a desarrollar plataformas de desarrollo con el objetivo de realizar el trabajo citado anteriormente de una forma más sencilla y con ello, reducir los tiempos de desarrollo y obtener soluciones a diversos problemas que puedan surgir en un futuro.

Algunos de los entornos de desarrollo para aplicaciones robóticas más conocidos son los siguientes:

- ***Player***[2]: Se trata de un framework de programación de código abierto y libre, para el desarrollo de software enfocado a plataformas robóticas. P*layer* es un servidor de red para el control de un robot que proporciona una potente interfaz con una gran variedad sensores y actuadores. Tiene dos simuladores asociados: *Staage* y *Gazebo* que funcionan en espacios de trabajo 2D y 3D respectivamente. *Stage* es un simulador robótico que proporciona un mundo virtual de una población de robots móviles y sensores, incluyendo diversos objetos para la detección y manipulación de los robots. *Gazebo* cuenta con un mundo tridimensional que permite simular modelos robóticos en 3D y diversos tipos de sensores haciendo posible un fácil diseño y uso del robot en algunos de los escenarios que el simulador proporciona en su base de datos.
- ***CARMEN***[3]: se trata de un conjunto de software de código abierto que controla robots móviles basándose en un diseño de software modular que permite realizar diferentes aplicaciones como la navegación, mapeado, localización, etc.

Figura 2.12: Logo Carmen.

[2] http://playerstage.sourceforge.net/
[3] http://carmen.sourceforge.net/

- ***ROS***[4]: Se trata de un framework que proporciona al usuario una serie de herramientas de visualización, librerías, monitorización y análisis, todas ellas en código abierto específicas para desarrollo de software robótico.

Figura 2.13: Logo Ros.

- ***LibrePilot***: Es una plataforma de código abierto que es usada tanto para vehículos aéreos no tripulados como modelos de aeronaves o robots. Implementa la fusión de diversos sensores, el control del dispositivo y la navegación autónoma. Todo ello es totalmente configurable desde la aplicación.

2.5.1 *LibrePilot*

Este proyecto de código abierto fue fundado en julio de 2015. Se centra en la investigación y desarrollo de software y hardware para ser utilizado en una gran variedad de aplicaciones. Uno de los objetivos del proyecto es proporcionar un ambiente colaborativo en el que todo aquel que lo desee puede realizar su aportación y con ello, convertirlo en el hogar ideal para el desarrollo de ideas innovadoras.

Figura 2.14: Logo *LibrePilot*.

LibrePilot[7] fomenta el intercambio y la colaboración con otros proyectos, además permite añadir soporte para hardware o software existente, todo ello bajo el espíritu de ser un proyecto de código abierto.

LibrePilot proviene de un proyecto anterior denominado *OpenPilot* que era muy similar al actual ya que muchos de sus desarrolladores se encuentran ahora trabajando en el nuevo software. *LibrePilot* es gobernado por un consejo de miembros que utilizan métodos de consenso para tomar decisiones importantes y para establecer la dirección general del proyecto.

2.5.1.1 Arquitectura del sistema

LibrePilot es un proyecto de código abierto utilizado en vehículos aéreos no tripulados (UAV), modelos de aeronaves y otros robots. El código se ejecuta en una controladora de vuelo. A través de los datos recogidos por los sensores, la controladora envía órdenes a una serie de actuadores como motores *brushless*, luces de navegación, zumbadores, servos, etc.

2.5.1.2 Arquitectura del Hardware

Un UAV puede ser controlado a partir de transmisores de control remoto cuyas señales serán recibidas por algún tipo de receptor que, en algunos casos, puede controlar directamente a los actuadores. En el caso de los multirrotores o vehículos con control aumentado, un controlador se interpone entre transmisor

[4] http://www.ros.org/

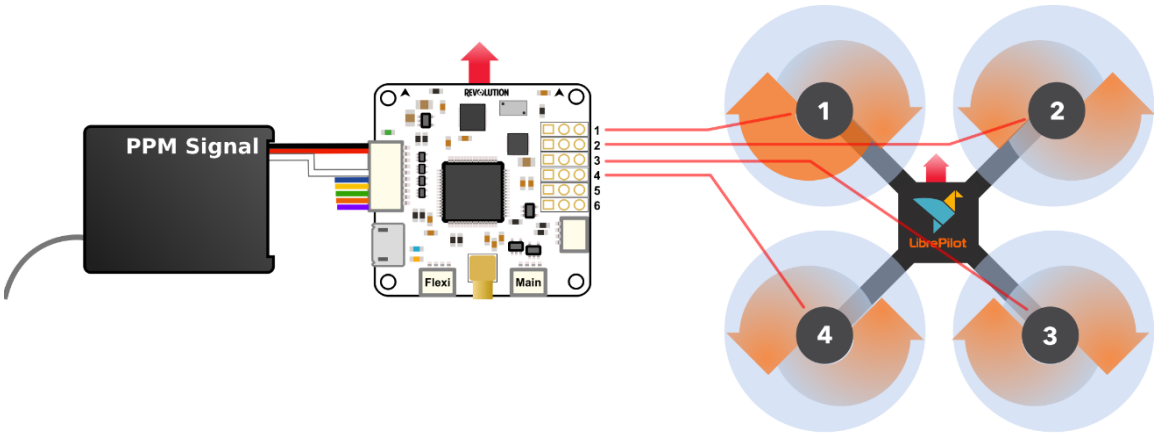


Figura 2.15: Arquitectura del hardware.

y actuadores para interpretar los comandos de control y ejecutar algoritmos para realizar el control del vehículo. Para poder llevar a cabo dicho control es necesaria la información de los sensores integrados en el vehículo, normalmente una Unidad de Medición Inercial (IMU) y de otros sensores auxiliares como un módulo GPS[5] Por último, la controladora recoge toda la información necesaria y controla los actuadores tales como los controladores de velocidad, para controlar los motores, y servos.

2.5.1.3 Arquitectura del Software

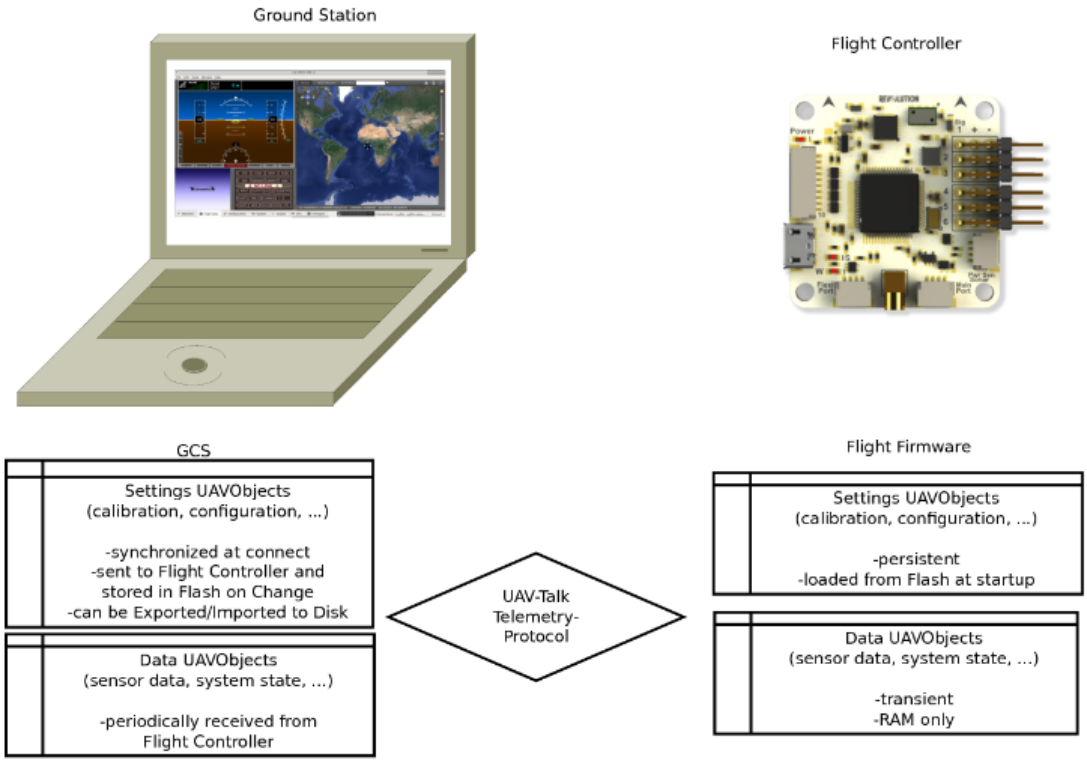


Figura 2.16: Arquitectura del Software.

En la figura 2.16 podemos observar la arquitectura del proyecto *LibrePilot*. A continuación se explicará en que consiste cada uno de los elementos que aparecen en ella.

[5]Sistema de Posicionamiento Global, más conocido por sus siglas en inglés, GPS (siglas de Global Positioning System).

- **GCS**[6]: Es el software con el que se puede configurar la controladora de vuelo y que normalmente se ejecuta en un ordenador. Se conecta con la controladora de vuelo mediante un cable USB, un cable serie o alguna forma de telemetría de radio. Además de la posibilidad de configurar la controladora, también puede registrar los datos de telemetría de vuelo. Se implementa en el lenguaje C++ utilizando el framework Qt5[7].

- ***Flight Firmware***: Se ejecuta en la controladora de vuelo y funciona tanto cuando está conectado al GCS como de forma independiente. Se implementa en Embedded C y C++ usando FreeRTOS, que es un sistema operativo que trabaja en tiempo real.

- ***UAVObjects***: Todos los datos se guardan en contenedores de datos bien definidos llamados *UAVObjects*. Se definen en tiempo de compilación usando la descripción XML[8]. Los ajustes persistentes y los datos de tiempo de ejecución transitorios también se guardan en estos contenedores. Éstos pueden ser intercambiados o incluso modificados tanto por GCS como por *Flight Firmware* haciendo uso del protocolo *UAVTalk*. A continuación se explica brevemente en que consiste el protocolo *UAVTalk*, la configuración de los *UAVObjects* y los datos que almacenan éstos.

  - **Protocolo *UAVTalk***: Este protocolo se utiliza para intercambiar datos guardados en los *UAVObjects* entre componentes que comparten el mismo conjunto de objetos definidos. Típicamente entre *Flight Firmware* y GCS. *UAVTalk* implementa mensajes para:

    1. Enviar un *UAVObject* de un lado al otro (con o sin solicitud de confirmación).
    2. Solicitar al otro lado que envíe un *UAVObject* específico.
    3. Reconocer un objeto definido.

  - **Configuración *UAVObjects***: La configuración de los *UAVObjects* sólo se realiza en el GCS y posteriormente, se envían al *Flight Firmware* cada vez que se modifiquen. Estos datos se cargan en el tiempo de arranque desde la memoria no volátil de la controladora de vuelo. Todos los datos de la configuración se toman de la controladora de vuelo cada vez que se establece la conexión entre ésta y el GCS. Todos los ajustes que se configuren se pueden importar o exportar para poderlos guardar y usarlos posteriormente cuando se desee.

  - **Datos *UAVObjects***: Los *UAVObjects* almacenan datos, comandos de control o información de estado durante el tiempo de ejecución. Por lo general, sólo se envían desde el *Flight Firmware* hasta el GCS en intervalos de tiempo periódicos a excepción de los objetos de control. Éstos son objetos de datos que se utilizan para influir en el comportamiento en tiempo de ejecución.

- **GCS Plugins**: Todos los componentes del software GCS se implementan como *Plugins*[9]. Cada *Plugin* define *widgets*[10] de interfaz de usuario que se pueden mostrar en una tabla dentro del GCS. Para cada *Plugin* se pueden configurar uno o varios *widgets*, cada uno con su propia configuración. Algunos de los *Plugins* más importantes son:

  - *Primary Flight Display*: Nos muestra un horizonte artificial y gran cantidad de datos del vuelo.
  - *OPMap*: Muestra un mapa del mundo y la localización de nuestro dispositivo en él.
  - *System Health*: Muestra una tabla en la que se puede observar el estado de varias alarmas.
  - *Scope*: Éstos muestran gráficamente el contenido de cualquier *UAVObject*. Es útil para mostrar datos de sensores en tiempo real.

[6] Ground Control Software.
[7] http://www.qt.io/
[8] siglas en inglés de eXtensible Markup Language, traducido como 'Lenguaje de Marcado Extensible'.
[9] Aplicación o complemento que se relaciona con otro/a para agregarle una función nueva.
[10] Pequeña aplicación que da fácil acceso a funciones usadas con frecuencia.

  - *UAVObjectBrowser*: Permite mostrar y modificar el contenido de cada *UAVObject*, instruir al *Flight Firmware* para cargar, guardar o eliminar un *UAVObject* desde la memoria *Flash*, y modificar las reglas de transferencia de datos.

  Los *Plugins* operan en uno o más *UAVObjects*. Cada modificación que se haga a cualquier *UAVObject*, por ejemplo cuando se reciben datos del *Flight Firmware* o cuando se modifican por otro plugin, desencadena en un evento al que cualquier *Plugin* puede responder.

- **Módulos del *Flight Firmware***: Los componentes software del *Flight Firmware* se implementan como módulos. Su ejecución puede ser provocada por eventos hardware (disponibilidad de nuevos datos de sensores) o por la actualización de *UAVObjects* específicos. Los módulos se comunican exclusivamente utilizando *UAVObjects*, tanto como si la comunicación es entre ellos o con el GCS. A continuación se da una breve descripción de cada uno de los módulos del *Flight Firmware*.

  - Módulo Sensores: Recibe datos de sensores como giróscopo, acelerómetro y los almacena en sus respectivos *UAVObjects* (*GyroSensor* , *AccelSensor*).
  - Módulo receptor: Recibe datos del receptor conectado, verifica su integridad y seguridad, y almacena el resultado en sus respectivos *UAVObjects* (*ManualControlCommand*).
  - Módulo *ManualControl*: Interpreta los datos del *UAVObject ManualControlCommand* y establece otros *UAVObjects* basados en el modo de vuelo y la entrada de control. Incluye el *UAVObject FlightStatus* en el que se almacena el modo de vuelo actual, el estado de armado del motor, entre otros.
  - Módulo *StateEstimation*: Fusiona los datos de varios sensores usando filtros.
  - Modulo de estabilización: Lee el dato de control para el control de vuelo desde el *UAVObject StabilizationDesired*, y con este dato calcula la salida de control requerida y la almacena en el *UAVObject ActuatorDesired*.
  - Módulo *Actuators*: Lee la salida de control deseada de *ActuatorDesired*, aplica una matriz denominada *Mixer Matrix* y con los datos que se obtienen, controla los motores. La función de esta matriz se explica más adelante.
  - Módulo de Telemetría: Es el responsable de intercambiar y sincronizar los datos de los *UAVObjects* con el GCS utilizando un dispositivo COM.
  - Módulo *PathPlanner*: Implementa la navegación como una máquina de estados. El comportamiento actual del vuelo se escribe en el *UAVObject PathDesired*.
  - Módulo *PathFollower*: Interpreta el comportamiento de *PathDesired* que, dependiendo del modo de vuelo, puede ser configurado por el módulo *ManualControl* o por el módulo *PathPlanner*. Ejecuta bucles de control de vuelo y calcula el punto de consigna para la estabilización de vuelo almacenada en *StabilizationDesired*.

2.5.1.4 Arquitectura de Control

La arquitectura de control depende del modo de vuelo y de los ajuste que se hayan realizado. En cuanto a los canales de control, todos ellos tienen una característica común, y es que pueden asumir valores entre -1 y 1. A continuación describimos cada uno de los canales de control:

- *Roll*: Define el comando de control para la rotación alrededor del eje X. 0 significa neutro, +1 significa máxima rotación hacia la derecha y -1 significa máxima rotación hacia la izquierda.

- *Pitch*: Define el comando de control para la rotación alrededor del eje Y. 0 significa neutral, +1 significa máxima rotación de la cabeza hacia arriba y -1 significa máxima rotación de la cabeza hacia abajo.

- *Yaw*: Define el comando de control para la rotación alrededor del eje Z. 0 significa neutral, +1 significa máxima rotación en el sentido de las agujas del reloj y -1 significa máxima rotación en el sentido contrario a las agujas del reloj.

- *Thrust*: Define el comando de control para el actuador que controla la energía. 0 significa neutro, +1 significa máximo empuje positivo, -1 significa máximo empuje negativo.

Estos canales de control virtual se asignan a actuadores físicos usando una matriz (figura 2.17) que se almacena en el *UAVObject MixerSettings* con información sobre el tipo de canal.

$$M = \begin{Bmatrix} & Roll & Pitch & Yaw & Thrust \\ Channel\,1 & m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ Channel\,2 & m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \\ Channel\,3 & m_{3,1} & m_{3,2} & m_{3,3} & m_{3,4} \\ Channel\,4 & m_{4,1} & m_{4,2} & m_{4,3} & m_{4,4} \\ \dots & \dots & \dots & \dots & \dots \end{Bmatrix}$$

Figura 2.17: Mixer Matrix.

A continuación se describen los diferentes modos de control que se pueden llegar a usar:

- **Control Manual**: En este modo de control, los datos del *UAVObject ManualControlCommand* que provienen del receptor de control alimentan directamente al objeto *ActuatorDesired*. Por lo tanto, los controles de vuelo se dirigen directamente a los actuadores con un control en lazo abierto. El propio piloto es el que cierra el lazo mediante el accionamiento del transmisor de control. En la figura 2.18 se presenta el algoritmo que sigue este modo de control.

$$\left\langle \begin{matrix} Flight \\ Control \\ Receiver \end{matrix} \right\rangle \Rightarrow \begin{matrix} Receiver \\ Module \\ (Output = Input) \end{matrix} \Rightarrow [ManualControlCommand] \Rightarrow$$

$$\Longleftarrow$$

$$\hookrightarrow\Rightarrow \begin{matrix} ManualControl \\ Module \\ (Output = Input) \end{matrix} \Rightarrow [ActuatorDesired] \Rightarrow \begin{matrix} Actuator \\ Module \\ (Output + Input * M) \end{matrix} \Rightarrow \langle Actuators \rangle$$

Figura 2.18: Algoritmo del modo control manual.

- **Control Estabilizado**: En este modo de control, los datos del *UAVObject ManualControlCommand* provenientes del receptor de control alimentan al objeto *StabilizationDesired*. En el modo de vuelo *Rate*, se usa un único bucle de control PID para el control del dispositivo, mientras que en el modo *Attitude*, se utilizan dos bucles PID en cascada donde el bucle exterior controla la velocidad angular basándose en el error de posición, y el bucle interno controla el actuador basado en el error de la velocidad angular. En la figura 2.19 y 2.20 podemos observar un esquema donde se explican los algoritmos que siguen estos dos modos de vuelo que hemos citado anteriormente.

$$\left\{ \begin{array}{l} \left\langle \begin{array}{c} Flight \\ Control \\ Receiver \end{array} \right\rangle \Rightarrow \cdots \Rightarrow \begin{array}{c} ManualControl \\ Module \\ (Output = Input) \end{array} \Rightarrow [StabilizationDesired] \\ \left\langle \begin{array}{c} Gyroscope \\ Sensor \end{array} \right\rangle \Rightarrow \begin{array}{c} Sensors \\ Module \\ (Output = k * Input + b) \end{array} \Rightarrow [GyroSensor] \Rightarrow \begin{array}{c} StateEstimation \\ Module \\ (Output = Input + bias) \end{array} \Rightarrow [GyroState] \end{array} \right\} \Rightarrow$$

$$\Longleftarrow$$

$$\hookrightarrow\Rightarrow \begin{array}{c} Stabilization \\ Module \\ (Output = PID\,(Input, k)) \end{array} \Rightarrow [ActuatorDesired] \Rightarrow \cdots$$

Figura 2.19: Algoritmo del modo de control Rate.

$$\left\{ \begin{array}{l} \left\langle \begin{array}{c} Flight \\ Control \\ Receiver \end{array} \right\rangle \Rightarrow \cdots \Rightarrow \begin{array}{c} ManualControl \\ Module \\ (Output = Input) \end{array} \Rightarrow [StabilizationDesired] \\ \left\{ \left\langle \begin{array}{c} Gyroscope \\ Sensor \\ Accelerometer \\ Sensor \\ \cdots \end{array} \right\rangle \right\} \Rightarrow \cdots \Rightarrow \begin{array}{c} StateEstimation \\ Module \\ (Output = EKF\,(Input)) \end{array} \Rightarrow \left\{ \begin{array}{c} [GyroState] \\ [AttitudeState] \end{array} \right\} \end{array} \right\} \Rightarrow$$

$$\Longleftarrow$$

$$\hookrightarrow\Rightarrow \begin{array}{c} Stabilization \\ Module \\ (Output = PID\,(PID\,(Input, k))) \end{array} \Rightarrow [ActuatorDesired] \Rightarrow \cdots$$

Figura 2.20: Algoritmo del modo de control Attitude.

- **Control Autopilot**: En los modos de vuelo que impliquen utilizar este control, la nave volará de forma autónoma gracias a las instrucciones dadas según la trayectoria a seguir que está definida en el *UAVObject PathDesired*. Se inicializa a la posición actual, establecida dinámicamente en función del *UAVObject ManualControlCommand* a partir de varios modos de vuelo asistido, o incluso puede ser configurado por el equipo de navegación implementado en el módulo *PathPlanner*. En la figura 2.21 se representa el algoritmo que sigue este modo de control.

$$\left\{ \begin{array}{l} \begin{array}{c} ManualControl \mid PathPlanner \\ Module \end{array} \Rightarrow [PathDesired] \\ \left\{ \left\langle \begin{array}{c} GPS \\ Sensor \\ Barometer \\ Sensor \\ \cdots \end{array} \right\rangle \right\} \Rightarrow \cdots \Rightarrow \begin{array}{c} StateEstimation \\ Module \\ (Output = EKF\,(Input)) \end{array} \Rightarrow \left\{ \begin{array}{c} [PositionState] \\ [VelocityState] \\ [AttitudeState] \\ \cdots \end{array} \right\} \end{array} \right\} \Rightarrow$$

$$\Longleftarrow$$

$$\hookrightarrow\Rightarrow \begin{array}{c} PathFollower \\ Module \\ (Output = F\,(Input)) \end{array} \Rightarrow [StabilizationDesired] \Rightarrow \cdots$$

Figura 2.21: Algoritmo del modo de control Autopilot.

El algoritmo de control F varía en función del tipo de vehículo y del *PathMode* que se seleccione en el *UAVObject PathDesired*. Los modos de control soportados hasta ahora son:

- *GoToEndpoint*: *Autopilot* intentará que la nave llegue a la posición indicada en una coordenada final fija. La nave no irá necesariamente en línea recta hacia ese punto final.
- *FollowVector*: *Autopilot* hará que la nave vuele en línea recta y corregirá cualquier desviación lateral del camino deseado hasta llegar al punto final deseado.
- *CircleRight, CircleLeft*: *Autopilot* indicará a la nave que debe circular alrededor de una coordenada final a una distancia fija dada por la coordenada de inicio.
- *SetAccessory*: *Autopilot* establecerá canales de salida auxiliares, por ejemplo, para parpadear luces.
- *DisarmAlarm: Autopilot* iniciará una secuencia de desarme autónoma que cortará la energía a todos los motores. Útil para realizarlo tras un aterrizaje.
- *Ground*: *Autopilot* realizará una secuencia de aterrizaje automatizada.
- *Break*: Se utiliza internamente por los modos de vuelo asistido para reducir la velocidad.
- Velocidad: *Autopilot* intentará mantener una velocidad en el espacio 3D.
- *AutoTakeoff*: Autopilot realizará una secuencia de despegue totalmente automatizada.

La correcta función del *Autopilot* depende de muchos factores, incluyendo:

- Una correcta calibración de los sensores.
- Un ajuste adecuado de los coeficientes PID de estabilización.
- Un ajuste de los parámetros de control propios del modo *Autopilot*.

2.5.1.5 Arquitectura del código de vuelo

En esta sección se pretende esbozar la arquitectura del código de vuelo (el código que se ejecuta en las controladoras de vuelo).

Primero echaremos un vistazo a la estructura de directorios para conocer cómo se distribuye, con la finalidad de saber encontrar lo que se esté buscando. Después, se explicará cómo arranca *LibrePilot* cuando se enciende la controladora de vuelo y qué archivos están involucrados.

El código de vuelo se encuentra en el directorio *flight/* del proyecto. En la figura 2.22 podemos observar la estructura de este directorio:

Directory structure of flight/

```
flight/
├── libraries/
├── make/
├── modules/
├── pios/
├── Project/
├── targets/
├── templates/
├── tests/
├── uavobjects/
└── uavtalk/
```

Figura 2.22: Estructura directorio flight/.

Tabla 2.1: Directorios más importantes de flight/.

Directorio	Contenido
modules/	Contiene módulos que comprenden todas las características del código de vuelo
pios/	Contiene *PiOS*, una capa de abstracción de hardware (HAL)
targets/	Contiene el firmware y el bootloader para las placas compatibles con este software.

Para comprender la estructura básica de este proyecto, en la tabla 2.1 se explican los directorios más importantes.

Muchas veces el mejor punto para comenzar a aprender cómo funciona un programa es el inicio de éste. Por ello, lo primero que se suele hacer cuando analizas código es buscar el fichero *main*. Para el proyecto *LibrePilot*, este fichero se encuentra en el directorio *targets/*. Cada procesador tiene sus propios directorios de firmware y de bootloader, por esta razón, tienen dos funciones *main*, una para el bootloader y otra para el firmware.

Cuando se enciende una controladora de vuelo con *LibrePilot* cargado, el primer código que se ejecutará es el gestor de arranque (bootloader). Ésto se debe a que el vinculador está configurado para colocar el código del gestor de arranque al comienzo del espacio de direcciones del procesador.

Entre las tareas más importantes del gestor de arranque encontramos las siguientes:

1. Comprobar si el usuario solicitó una actualización del firmware a través del cable USB (por ejemplo, con el botón *Rescue* en la tabla *Firmware* dentro del GCS).
2. Descargar la actualización del firmware a través del cable USB (si se solicita).
3. Iniciar el firmware.

Una vez que el gestor de arranque haya finalizado, el procesador salta a la parte de código del firmware. Al comienzo del código de firmware se puede encontrar la función *main*, que se puede encontrar en *flight/targets/<board name>/firmware/<board name>.cpp*.

La función *main* del firmware llama a algunas funciones de inicialización de bajo nivel usando *PiOS* (la capa de abstracción del hardware). A continuación, se iniciará el módulo del sistema, que a su vez iniciará todos los otros módulos.

2.5.1.6 El protocolo UAVTalk

UAVTalk es un protocolo binario eficiente, flexible y totalmente abierto que está diseñado para la comunicación con *UAVs*.

El protocolo *UAVTalk* utiliza el concepto de objetos de telemetría, ésto hace que el protocolo sea extremadamente flexible y escalable. El diseño basado en objetos permite optimizar el protocolo para canales de baja velocidad, además se pueden configurar para tiempos de actualización rápida.

El protocolo es independiente del canal que se utilice, ya que se espera que sea utilizado sobre una variedad de módems y canales inalámbricos. Se espera que todos los canales apoyen la comunicación bidireccional. Muchos de los canales que se utilizarán serán *semidúplex*, aunque también existirán casos en los que se emulará el modo *duplex* completo, como por ejemplo en los módems Xbee[11]. El enlace descendente soportará mensajes de telemetría tales como datos del GPS, del nivelador, de la navegación,

[11] http://www.digi.com/blog/iot/flying-eye-relies-on-digi-xbee-for-drone-connectivity-and-parachute-deploymer

y otra información que sea necesaria para el GCS. El enlace ascendente se utilizará, por ejemplo, para controlar el piloto automático, para la actualización de planes de vuelo, ajustes y plug-ins. Se deben tomar algunas precauciones para evitar la transferencia de archivos grandes y limitar algunos comandos durante el vuelo.

Debido a que este protocolo soporta una gran variedad de objetos de datos, el protocolo de bajo nivel no es consciente de la estructura de datos que se transmiten. El embalaje, desembalaje y la interpretación de datos se realiza en las capas superiores. En la figura 2.23 se muestra la estructura de este protocolo y a continuación se procederá a describirlo con más detalle.

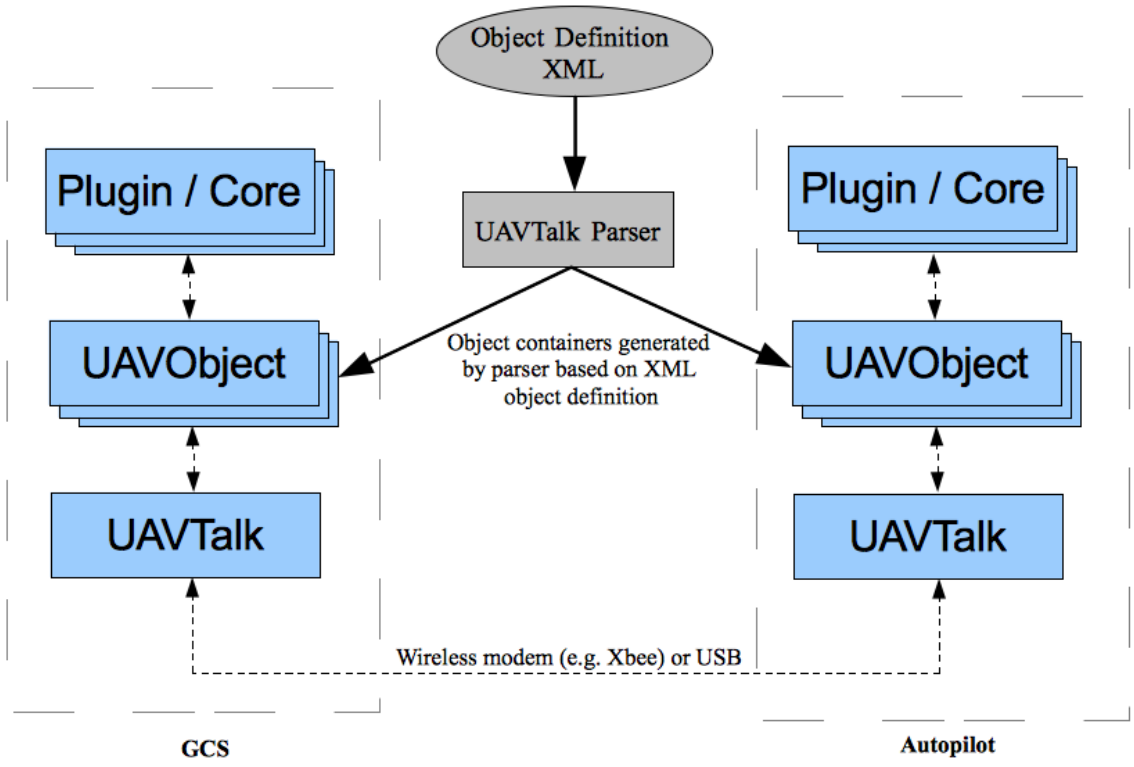


Figura 2.23: Estructura protocolo UAVTalk.

El protocolo *UAVTalk* implementa la comunicación de bajo nivel entre el GCS y *Autopilot*, y para ello proporciona el transporte de las estructuras de datos definidas por los *UAVObjects*. Para hacer más fácil la adicción de *UAVObjects*, el protocolo *UAVTalk* no necesita conocer los detalles de la estructura de datos sino que simplemente envía los arrays de bytes y enruta los recibidos al objeto adecuado.

Tabla 2.2: Formato mensajes UAVTalk.

Parte del mensaje	**Campo**	**Longitud (bytes)**	**Valor**
Encabezamiento	Sync Val	1	0x3C
	Tipo de mensaje	1	• 0x0 Object Data (OBJ) • 0x1 Solicitud de objeto (OBJ-REQ) • 0x2 Objeto con solicitud de acuse de recibo (OBJ-ACK) • 0X3 Reconocimiento (ACK) • 0X4 Reconocimiento negativo (NACK) • Bits 5-7 Versión del protocolo. • 0x8 Indica una marca de tiempo.
	Longitud	2	No es necesario ya que el ID del objeto lo predice. Contiene la longitud del encabezado y datos.
	ID de objeto	4	Todo objeto tiene un único ID (generado por el analizador)
	ID de instancia	2	Todo objeto tiene un único ID de instancia. Siempre 0 para instancia única
	Timestamp	2	Timestamp para objetos timestamped (no manejado por GCS)
Datos	Datos	0-255	Objeto serializado.
Suma de control	Suma de control	1	CRC-8 checksum (Encabezado y Datos)

En este protocolo existen diferentes tipos de transacciones que se describen a continuación.

- **Mensaje de objeto**:

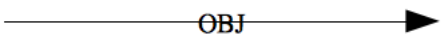


Figura 2.24: Mensaje de objeto.

Se trata de una actualización simple de un objeto. Cuando llega este mensaje, se invoca la función de descompresión del objeto correspondiente y se actualizan los campos de tal objeto. No se manda ninguna respuesta y habitualmente se utiliza en la actualizaciones periódicas.

- **Mensaje de solicitud de objeto**:

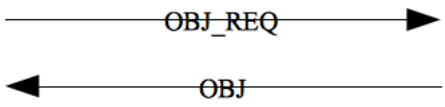


Figura 2.25: Mensaje de solicitud de objeto.

Se envían siempre que el remitente requiera el último valor de un objeto. El mensaje *OBJ-REQ* no tiene carga útil y el destinatario responde inmediatamente rellenando los datos del objeto y enviando un mensaje *OBJ*. En caso de que no se encuentre el *ObjectID* solicitado, el receptor responderá con un mensaje *OBJ-NACK*, en lugar del mensaje *OBJ*.

- **Mensaje de objeto reconocido**:

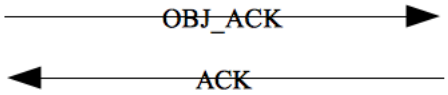


Figura 2.26: Mensaje de objeto reconocido.

Este mensaje devuelve una confirmación al remitente para confirmar que el objeto fue recibido por el extremo remoto. El acuse de recibo está en forma de un mensaje *ACK* sin carga útil.

- **Mensaje de objeto no reconocido**: Es un mensaje sin carga útil. Se envía en respuesta a un mensaje *OBJ-REQ*, siempre que el *ObjectID* solicitado no exista en el lado remoto. El *objectID* se envía de nuevo con el *OBJ-NACK* para que el remitente puede tener en cuenta la ausencia de este *ObjectID* en el lado remoto y actuar en consecuencia.

El ID o identificador del objeto es un entero único de 32 bits que identifica el objeto al que representan los datos del mensaje. El identificador del objeto es generado automáticamente por el analizador aplicando un *hash* en la definición XML del objeto. Esto tiene dos ventajas, primero, el ID es único por lo que existe poca probabilidad de colisión, y segundo, cada vez que se cambia la definición del objeto, el ID también cambiará. Ésta es una forma conveniente de asegurarse de que los *UAVObjects* en el GCS y en el *Autopilot* tengan la misma definición.

La capa *UAVTalk* también es responsable de las actualizaciones periódicas de objetos. Cada objeto registrado tiene un temporizador asociado con él, cuando el temporizador expira un mensaje *OBJ* es enviado. Los valores del temporizador se pueden actualizar dinámicamente.

Los metadatos de objetos pueden ser solicitados y enviados usando *ObjectId* +1. El objeto de metadatos tiene el mismo formato que un objeto normal con instancia 0 y los campos de los metadatos siguen el formato que se puede ver en la tabla 2.3.

Tabla 2.3: Formato metadatos.

Campo	**Longitud del campo (bit)**	**Detalles de la bandera**	**Longitud de la bandera (bit)**	**Valor**
Banderas	16	acceso	1	Define el nivel de acceso para las transacciones locales (*readonly* = 0 y *readwrite* = 1)
		GcsAccess	1	Define el nivel de acceso para las transacciones GCS locales (*readonly* = 0 y *readwrite* = 1), no utilizado en el vuelo
		Telemetría	1	Define si se requiere un *ack* para las transacciones de este objeto (1: *acked*, 0: *not acked*)
		GcsTelemetryAcked	1	Define si se requiere un *ack* para las transacciones de este objeto (1: *acked*, 0: *not acked*)
		TelemetryUpdateMode	2	Modo de actualización utilizado por el módulo de telemetría (*UAVObjUpdateMode*)
		GcsTelemetryUpdateMode	2	Modo de actualización utilizado por el GCS (*UAVObjUpdateMode*)
		LoggingUpdateMode	2	Modo de actualización utilizado por el módulo de registro (*UAVObjUpdateMode*)
VueloTelemetry UpdatePeriod	16			Define indicadores para los modos de actualización y registro y si una actualización debe ser *ACK*
GcsTelemetry UpdatePeriod	16			Periodo de actualización utilizado por el GCS (sólo si el modo de telemetría es *PERIODIC*)
Logging UpdatePeriod	16			Período de actualización utilizado por el módulo de registro (sólo si el modo de registro es *PERIODIC*)

Normalmente, la capa *UAVTalk* sólo se utiliza a través de los *UAVObjects* y no directamente por la aplicación. Los *UAVObjects* mantienen una serie de campos de datos y proporcionan las funciones de embalaje y desempaquetado a la capa *UAVTalk*. El código real para el *UAVObject* es generado por el analizador basado en una definición XML del objeto y sus campos. El analizador generará entonces el código GCS del *UAVObject*. Habrá varios *UAVObjects*, uno para cada objeto definido en el XML. Se espera que cada *Plugin* tenga su propio XML y un conjunto de *UAVObjects*. Además, el analizador generará las funciones necesarias para el acceso genérico a los datos, lo que será utilizado por partes del GCS que no necesita conocer el tipo de objeto real. El analizador también generará los campos necesarios para configurar remotamente el período de las actualizaciones automáticas de los objetos.

2.6 *OpenCV*

Figura 2.27: Logo OpenCV.

2.6.1 ¿Qué es *OpenCV*?

OpenCV[8] es una librería de visión por computador de código abierto que se encuentra disponible en *http://SourceForge.net/projects/opencvlibrary*. Esta librería está escrita en los lenguajes C y C++ y es compatible con *Linux, Windows y Mac OS X*. Cuenta con un desarrollo activo en interfaces para *Python, Ruby, Matlab* y otros lenguajes.

OpenCV ha sido diseñado para ser eficiente en cuanto a gasto de recursos computacionales y enfocado a aplicaciones en tiempo real. Está escrito y optimizado en C y toma las ventajas que nos puedan proporcionar los procesadores de varios núcleos.

Uno de los objetivos de *OpenCV* es proveer una infraestructura de visión por computador fácil de utilizar que ayuda a los programadores a desarrollar aplicaciones sofisticadas de CV (*Computer Vision*) rápidamente. La librería contiene más de 500 funciones que abarcan muchas áreas de CV, entre las que podemos encontrar, inspección en la producción de productos, imágenes médicas, seguridad, calibración de cámaras y aplicaciones robóticas. Además, *OpenCV* proporciona una sublibrería de propósito general dedicada al aprendizaje automático denominada *Machine Learning Library* (MLL). Esta sublibrería está centrada en el reconocimiento estadístico de patrones y *clustering*.

2.6.2 ¿Qué es Visión Artificial?

Visión Artificial es la transformación de datos provenientes de un fotograma o de una vídeo cámara en una decisión o una nueva representación. Todas estas transformaciones se realizan para el logro de algún objetivo en particular. Los datos de entrada pueden incluir alguna información contextual, como que la cámara está montada en un auto o que el telémetro láser indica un objeto a 1 metro de distancia. La decisión podría ser que hay una persona en esta escena o que hay 14 células tumorales en esta diapositiva. La nueva representación se refiere, por ejemplo, a la transformación de la imagen en color a una escala de grises o eliminar el movimiento de la cámara de una secuencia de imágenes.

El cerebro humano divide las señales provenientes del sentido de la vista en muchos canales de visión que hacen fluir diferentes tipos de información en el cerebro. Nuestro cerebro tiene un sistema de atención

que identifica las partes más importantes de una imagen para ser examinadas y suprime aquellas partes que no lo sean, además, los diferentes sensores que tenemos permiten al cerebro ir aprendiendo a identificar las diferentes señales que nos rodean y actuar en consecuencia.

En los sistemas de visión artificial, un equipo recibe un conjunto de números desde una cámara o desde un disco, y eso es todo. No aprende con años de experiencia como nuestro cerebro, ni tiene un control automático de enfoque y apertura como nos ocurre a nosotros. La mayoría de los sistemas de visión son aún bastante ingenuos comparados con el cerebro. A continuación se explica con un ejemplo las dificultades que pueden encontrar los sistemas de visión en nuestro día a día.

En la figura 2.28 se encuentra la imagen de un automóvil. En esta imagen se puede ver que dicho coche tiene un espejo lateral en el lado del conductor y se puede observar que lo que el ordenador puede 'ver' es simplemente un conjunto de números. Algunos de estos números pueden tener alguna componente de ruido y, por lo tanto, nos llega a dar una información no muy exacta. Nuestra tarea consistiría en conseguir, de ese conjunto de números lleno de ruido, en una percepción adecuada de lo que vemos por el espejo.

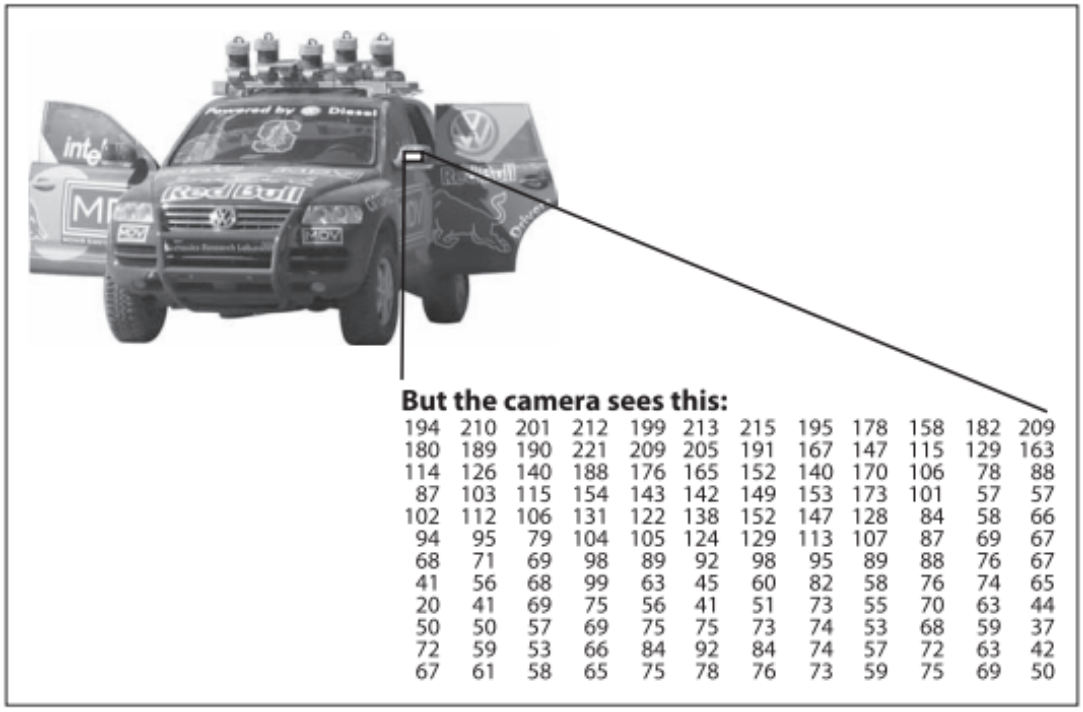


Figura 2.28: Para un ordenador, lo que muestra el espejo del coche es un conjunto de números.

En la figura 2.29 se da otro ejemplo del porque la visión artificial es tan complicada. Existen varias maneras de reconstruir las señales 3D en una visión 2D y es difícil saber cuál es la correcta. La misma imagen 2D puede corresponder a infinitas combinaciones de la escena 3D, incluso cuando los datos estuviesen correctos y sin ruidos. Sin embargo, se sabe que en el mundo existen diversos ruidos que nos hará tener una percepción equivocada de la realidad en los sistemas de visión. Estos ruidos se deben a variaciones en el mundo (clima, iluminación, reflejos, movimientos), las imperfecciones en la lente y la configuración mecánica, tiempo de integración finito en el sensor (desenfoque de movimiento), ruido eléctrico en el sensor u otros aparatos electrónicos y artefactos de compresión después de la captura de la imagen.

En el diseño de un sistema práctico, el conocimiento contextual puede ser de gran ayuda para evitar las limitaciones impuestas sobre nosotros por medio de sensores visuales. Considerando el ejemplo de un

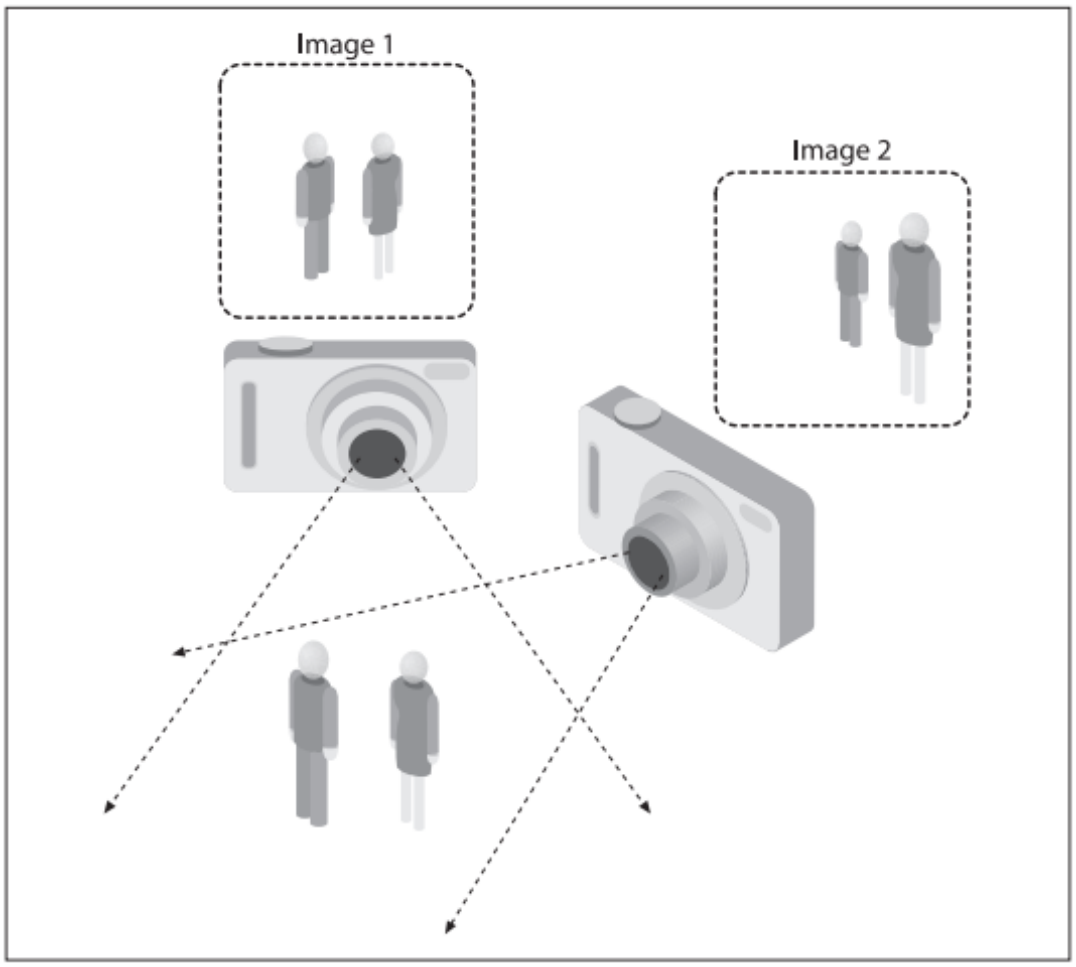


Figura 2.29: En 2D, la apariencia de los objetos puede cambiar radicalmente con el punto de vista.

robot móvil que desea encontrar y coger una grapadora que se encuentra en el interior de un edificio, el robot debe saber que un escritorio es un objeto que se encuentra en el interior de las oficinas y que la grapadora suele encontrarse sobre el escritorio. Ésto le dará una referencia implícita del tamaño que puede tener una grapadora y, al darle la información de que debe de estar sobre un escritorio, elimina lugares donde no puede encontrarse. Para que el robot sepa lo que es una grapadora, necesita tener una base de datos con imágenes de grapadoras reales de diferentes tamaños, colores y posiciones en las que se pueda encontrar. Cuanto más grande sea esa base de datos más posibilidades tendrá el robot de encontrar dicha grapadora.

El siguiente problema que se encuentra nuestro robot es el ruido. Normalmente éste se puede reducir usando métodos estadísticos. Por ejemplo, es imposible detectar un borde en una imagen simplemente comparando un punto con sus vecinos inmediatos. Pero si nos fijamos en estadísticas sobre una región local, la detección del borde será mas sencilla. También es posible compensar el ruido tomando estadísticas en el tiempo. Además, existen otras técnicas para reducir el ruido basadas en construir modelos explícitos aprendidos directamente de los datos disponibles.

Las acciones y decisiones que toma un dispositivo basadas en los datos tomados con una cámara son realizados en el contexto de una tarea específica. Es necesario eliminar todo el ruido que se pueda de las imágenes y con ello, conseguiremos que nuestro sistema de seguridad, por ejemplo, pueda detectar un problema si alguien intenta acercarse o que nuestros sistema de vigilancia pueda contar la gente que cruza un determinado área. El software de visión para robots que deambulan por los edificios de oficinas

empleará estrategias diferentes a las del software de visión para las cámaras de seguridad fijas porque los dos sistemas tienen contextos y objetivos muy diferentes. Como regla general: cuanto más limitado es un contexto de visión por computador, más se puede confiar en esas limitaciones para simplificar el problema y más confiable será nuestra solución final.

OpenCV está dirigido a proveer las herramientas básicas necesarias para resolver problemas de visión artificial. En algunos casos, las funciones de alto nivel que encontramos en la librería son suficientes para resolver problemas complejos de visión artificial. Incluso cuando éste no es el caso, los componentes básicos de la librería son suficientes para permitir la creación de una solución completa a cualquier problema de visión artificial. En caso de que las soluciones anteriores no resuelvan nuestros problemas, existen muchos métodos de prueba y error usando esta librería que nos podrían dar una solución; todos ellos comienzan resolviendo el problema utilizando tantos componentes de la librería como sea posible. Normalmente, tras conseguir tu primera solución compilada y funcionando, podrás ver los fallos de ésta y resolverlos para la próxima compilación. Una vez que tengas tu solución podrás usarla de punto de referencia para tus próximos proyectos.

2.6.3 Estructura y contenido de *OpenCV*

Opencv está estructurado en cinco componentes principales, cuatro de ellos se muestran en la figura 2.30.

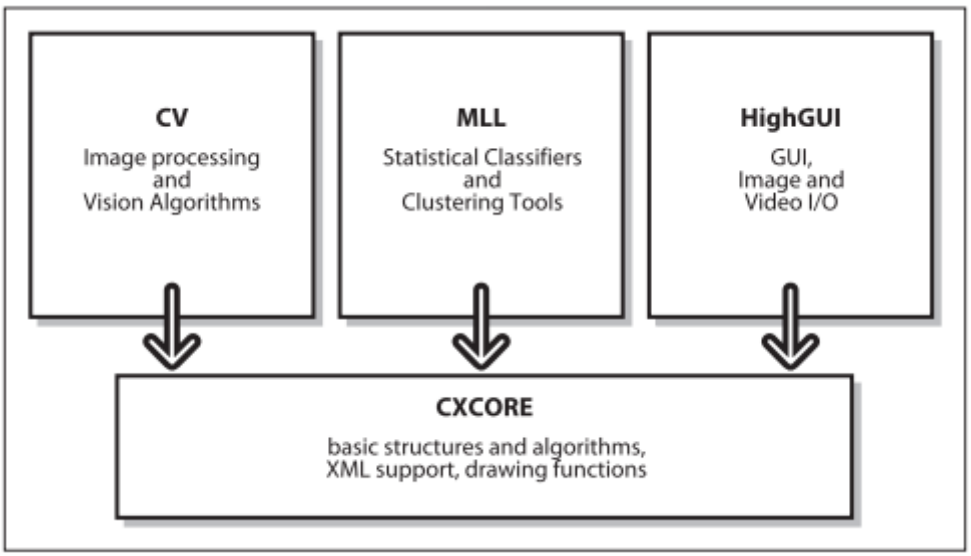


Figura 2.30: Estructura básica de Opencv.

A continuación se describen brevemente cada uno de estos componentes:

- ***CV***: Se encarga del procesamiento de imágenes y de los algoritmos de visión computarizada de alto nivel.
- ***ML***: Es la librería de aprendizaje automático, que incluye herramientas de clasificación estadística y clustering.
- ***HighGUI***: Se encarga de la gestión de entrada y salida y lo referente a la interfaz de usuario. A su vez provee codecs para imagen y vídeo.
- ***CXCore***: Contiene las estructuras básicas de OpenCV como son Mat (estructura para representar matrices) y funciones básicas que utilizan el resto de los componentes.

El otro componente que falta por describir es el *CvAux* que contiene las funciones auxiliares (algoritmos experimentales). Otros componentes que podemos destacar son *imgproc* que se encarga del procesamiento de imágenes mediante algunas de las operaciones más habituales (filtrado, transformación geométrica, conversión de color, entre otras), *calib3d* que posee múltiples algoritmos de visión geométrica, calibración de sonido y cámara, estimación de posición de objetos, reconstrucción 3D, y *features2d* que posee detectores de características sobresalientes, descriptores y comparadores descriptores.

2.6.4 Portabilidad

OpenCV fue diseñado para ser portable. Fue escrito originalmente para compilar en *Borland C++*, MSVC++ y los compiladores de *Intel*. Esto significa que el código en C y C++ tenía que ser medianamente estándar para hacer más fácil el soporte multiplataforma. La figura 2.31 muestra las plataformas sobre las que se sabe que corre *OpenCV*. El soporte para la arquitectura *Intel* de 32 bits (IA32) en *Windows* es el más maduro, seguido por Linux en la misma arquitectura. La portabilidad en *Mac OS X* se convirtió en una prioridad sólo después de que *Apple* comenzase a usar procesadores *Intel*. La portabilidad menos madura es la del hardware de *Sun* y otros sistemas operativos. Si una arquitectura o sistema operativo no aparece en la figura 2.31, esto no significa que no hay portabilidad para *OpenCV*. *OpenCV* ha sido portado a casi todos los sistemas comerciales.

	IA32	EM64T	IA64	Other (PPC, Sparc)
Windows	✓ (w. IPP; MSVC6, .NET2005+OMP, ICC, GCC, BCC)	✓ (w. IPP; MSVC6+PSDK.NET2005+OMP, PSDK)	± (w. IPP; PSDK, some tests fail)	N/A
Linux	✓ (w. IPP; GCC, BCC)	✓ (w. IPP; GCC, BCC)	✓ (GCC, ICC)	✗
MacOSX	✓ (w. IPP, GCC, native APIs)	? (not tested)	N/A	✓ (iMac G5, GCC, native APIs)
Others (BSD, Solaris…)	✗	✗	✗	Reported to build on UltraSparc Solaris

Figura 2.31: Guía de portabilidad OpenCV del release1.0: los sistemas operativos se muestran a la izquierda; los tipos de arquitectura de computadora en la parte superior.

Capítulo 3

Desarrollo

En este capítulo se describirán los micro-controladores utilizados, sus características y los diferentes sensores que se han acoplado a ellos para que el conjunto funcione correctamente. Además, se explicará el montaje que ha sido necesario, los componentes que se han utilizado y la implementación del algoritmo necesario a nuestro dispositivo para obtener los resultados deseados.

3.1 Estructura y componentes

Debido a las imperfecciones, existen una gran cantidad de variables que intervienen en el proceso de control y por ello, se debe diseñar un sistema de control que elabore una respuesta en función de la inclinación que tenga en cada momento el aparato.

En este apartado se describe la estructura funcional de nuestro dispositivo y los componentes físicos necesarios para formarlo [9].

Figura 3.1: Estructura drone.

3.1.1 Estructura funcional

La estructura funcional estará compuesta por los diferentes componentes que en este apartado se explican. Uno de los elementos más importantes son los **motores**, en nuestro caso 4, con velocidades de giro independientes que controlar. De esto último se encarga la **controladora de vuelo** sobre la que programamos el código que convertirá los datos de entrada en valores de tensiones para los motores. En un principio tenemos dos orígenes diferentes de datos de entrada: los sensores dinámicos y el control manual. Los primeros indican a la controladora la orientación espacial de la aeronave, y el segundo será la interfaz con la persona que lo esté pilotando. Esta persona envía datos inalámbricamente mediante un **módulo de comunicación**. Toda la electrónica mencionada anteriormente se alimentará a partir de una **batería**. Por último, todos estos elementos irán sujetos mediante un **chasis o *frame***.

3.1.2 Descripción de los componentes

A continuación se describen cada uno de los componentes que se han elegido para realizar este trabajo.

3.1.2.1 Motores

Para este proyecto se usan motores *brushless* o sin escobilla que son los responsables de mover las hélices. La principal ventaja que éstos tienen frente a los motores con escobillas es que no se produce tanto desgaste en su funcionamiento.

En los motores *brushless* la corriente eléctrica pasa directamente por las bobinas dispuestas en el estátor. Esta corriente genera pequeños campos magnéticos que obligan al rotor a girar. Estos motores van asociados a un variador que lo que hace es enviar la tensión de alimentación a las bobinas de forma secuencial. Con este proceso se consigue que los polos del rotor se vayan moviendo según el campo magnético generado por las bobinas de forma secuencial. La velocidad del rotor y su eje, dependerá de la velocidad de secuenciación del variador. Además, el variador disminuye o aumenta la tensión de alimentación de las bobinas para extraer el máximo rendimiento a los motores.

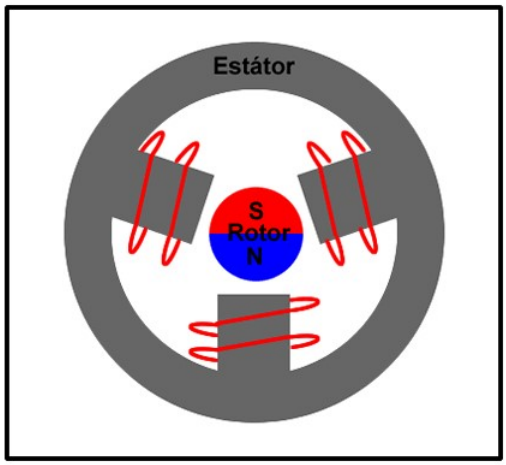


Figura 3.2: Motor *brushless*.

Este tipo de motor tiene una numeración que corresponde a sus características esenciales [10]. Las cuatros primeras cifras que aparecen se leen de dos en dos y se corresponden con la longitud del motor y con su altura. Las siguientes cuatro cifras se corresponden a los KV, cifra que se refiere a la constante de revoluciones de un motor, es decir, el número de revoluciones por minuto (rpm) que será capaz de

ofrecernos cuando se le aplique 1V (voltio) de tensión. Este número depende algunos factores como el número de espiras, el diámetro del hilo de cobre utilizado y el bobinado, la potencia de los imanes y la geometría del motor.

El fabricante calcula el valor anterior usando un banco de pruebas sin ningún tipo de peso en los motores, por ello, este valor no deja de ser teórico y muy distinto al valor que resultará posteriormente con una hélice.

Cada motor suele acompañarse de unas dimensiones de hélices recomendadas a utilizar. El conjunto de hélice, motor nos dará un empuje y éste es el que se debe calcular para conseguir elevar el dron. Si se eleva la tensión, se consigue más potencia y, por lo tanto, un mayor empuje. El inconveniente de todo esto es que aumenta el consumo eléctrico. Una mala relación entre tamaño de la hélice y potencia podría sobre-calentar el motor, por esta razón se debe estudiar la curva de rendimiento del fabricante y tomar la decisión.

En los motores con un KV bajo el total de amperios que circulará por éste es inferior a otros con KV superior. En estos motores el número de espiras es mayor, por tanto, el hilo de cobre es más fino. Está recomendado para drones que necesiten mucho par y poca velocidad. Es ideal para mover hélices de grandes dimensiones.

En los motores con un KV alto el total de amperios que circulará por éste es superior a otros con KV inferior. En estos motores el número de espiras es menor , por tanto, el hilo de cobre es más grueso. Está recomendado para drones que necesiten poco par y mucha velocidad. Ideal para mover hélices de dimensiones pequeñas.

Para nuestro multicóptero hemos escogido el motor denominado *BRUSHLESS MULTISTAR 1704* [11]. Se trata de un motor diseñado especialmente para multirrotores, cuenta con imanes de alta gama, rodamientos de alta calidad y una precisión equilibrada con un funcionamiento extraordinariamente suave.

Figura 3.3: Brushless Multistar 1704.

En la figura 3.4 podemos observar las especificaciones que nos presenta el fabricante de este motor.

Specs:
KV(RPM/V): **2300**
Lipo cells: **2~3s**
Max Watts(W): **49**
Max. current: **4.5A (10 secs)**
No Load Current: **0.5A**
Internal Resistance: **0.405 ohm**
Number of Poles: **12**
Shaft Size: **3 x 2mm**
Dimensions(Dia.xL): **21 x 15 mm**
Bolt Holes Spacing: **14mm**
Bolt thread: **M2**
Connector: **2mm bullet type connector**
Weight: **15.5g (motor only without adapters and screws)**

Power/PWM %	Prop	Voltage (V)	Current (A)	Thrust (gf)	RPM
10	5x4 STD	11.1	0.22	10.16	3027
25	5x4 STD	11.1	1.14	66.94	7233
50	5x4 STD	11.1	3.06	146.22	10491
75	5x4 STD	11.1	6.23	240.3	13244
100	5x4 STD	11.1	7.11	263.28	13675
10	5x4 STD	14.8	0.28	17.09	3906
25	5x4 STD	14.8	1.61	106.8	9049
50	5x4 STD	14.8	4.38	221.71	12861
75	5x4 STD	14.8	9.10	348.39	15729
100	5x4 STD	14.8	9.84	349.904	16094

Figura 3.4: Especificaciones motor multistar 1704.

3.1.2.2 Hélices

El objetivo principal de toda hélice [12] es generar impulso mediante su giro. Cuanto mayor sea el aire que puede llegar a mover, mayor será el empuje que consiga.

En los cuadricópteros podemos encontrar dos sentidos de giro para las hélices. Dos de ellas giran en el sentido de las agujas del reloj y la otras dos en sentido contrario.

El tamaño y grado de inclinación de las hélices viene determinado por dos números. El tamaño de la hélice se mide de punta a punta, también se le denomina como diámetro por la circunferencia que genera al girar. El aumento de una de estas dos características implica un aumento en el empuje que éstas realicen. Las hélices más pequeñas paran y aceleran más rápidamente que las hélices grandes. Por otro lado, las hélices más grandes tardan más en cambiar de revoluciones debido a la inercia.

El material de construcción de las hélices es muy variado. Entre los más comunes se encuentran el plástico, fibra de carbono y madera. Para cada aplicación que se desee se utiliza un material u otro. Las de plástico son muy duraderas y resistentes a los golpes. Las fibra de carbono y las de madera proporcionan más suavidad en el vuelo.

El número de palas que tengan las hélices también es un dato importante. Existen hélices bipala, tripala y cuatripala. En general, a mayor número de palas mayor es la superficie de área y, por lo tanto, mayor empuje. Además, a mayor número de palas mayor será el consumo.

En nuestro caso hemos escogido unas hélices tripala de tamaño 5x3 para obtener un mayor empuje.

Figura 3.5: Helices tripala 5x3.

3.1.2.3 ESC o variador

El ESC (*Electronic Speed Controller*) [13] es un sistema capaz de definir la velocidad de giro de un motor *brushless* mediante la generación de pulsos compatibles con este tipo de motores. Se componen de los siguientes elementos: dos cables para la entrada de alimentación de la batería, tres cables que se conectan al motor para proveerle la alimentación mediante pulsos, un conector con dos cables que van conectados a la controladora de la que recibe los datos para mover el motor, y por último, el ESC que contiene la parte electrónica que hace posible su funcionamiento.

La mayoría de los ESC incorporan un sistema BEC[1], el cual, hace posible regular un voltaje estable para poder hacer funcionar el receptor y los servos. Ésto elimina la necesidad de llevar una batería extra dentro de nuestro dron para alimentar estos componentes.

Los variadores son controladores de modulación por ancho de pulso (PWM) que controlan motores eléctricos. La controladora de vuelo manda una señal PWM al ESC con variaciones entre 1 y 2 milisegundos. En 1 milisegundo el motor estará parado, en 1,5 milisegundos el motor funcionará a la mitad de su potencia, en 2 milisegundos el motor funcionará a su máxima potencia.

Los ESC que se utilizan para motores *brushless* crean una corriente alterna trifásica a partir de corriente continua que proviene de la batería. Por esta razón, los variadores se conectan a los motores con 3 cables. Uno de los polos del motor genera un voltaje que es proporcional a la velocidad a la que esté girando el motor en ese momento. Con este voltaje, el ESC determina cómo de rápido y en qué dirección gira el motor. Gracias a ésto el variador conoce cómo mandar la corriente a los electroimanes del motor para que éste gire.

Los variadores se clasifica en función de la corriente máxima que puede soportar. Lo más recomendable es optar por un ESC que se encuentre por encima de la demanda de corriente del motor.

Para nuestra aplicación hemos escogido el ESC de la figura 3.6 . Este variador permite conducir una corriente máxima de 12 amperios, adecuado para nuestros motores ya que el límite de corriente es superior al consumido por cada motor. Además, incluye un BEC con el que se puede alimentar la controladora de vuelo.

[1] regulador de tensión que puede alimentar a nuestra controladora

Figura 3.6: AFRO ESC 12A con BEC de 5v y 0.5A.

3.1.2.4 Chasis o Frame

El chasis de un dron es el soporte que mantiene unidos todos los componentes y los fija en sus posiciones adecuadas. En nuestro caso necesitaremos que tenga 4 brazos y que sea lo más simétrico posible para evitar que un motor tenga que trabajar más que otros para mantener la estabilidad.

Existen chasis de muy variados materiales desde los hechos en madera de balsa hasta en fibra de carbono ultra resistente, pasando por los hechos en plástico por impresoras 3D.

En nuestro caso utilizaremos el *frame* que aparece en la figura . Se trata de un chasis de 250 mm de diámetro que está construido con materiales plásticos de alta resistencia. Además, incluye trenes de aterrizaje que se pueden instalar sin usar tornillos.

Figura 3.7: Frame de 250 mm con tren de aterrizaje.

3.1.2.5 Batería

Una batería es un dispositivo que almacena carga eléctrica para liberarla posteriormente a medida que se necesite. Existen diferentes tipos de baterías, las más utilizadas en radio-control son las de polímero de litio (LiPo) [14]. Son baterías con una excelente relación entre capacidad, peso, volumen y tensión.

Las baterías LiPo están formadas por elementos o células de 3.7V. Habitualmente, las baterías son de 1, 2 ó 3 elementos con voltajes de 3.7V, 7.4V y 11.1V respectivamente. El voltaje es importante ya que los motores *brushless* dan una cantidad de revoluciones por minuto en función del voltaje de la batería.

El tema de la capacidad y la duración ha sido el gran avance que se ha conseguido con este tipo de baterías. Existen baterías con una gran capacidad, ligeras y con menos volumen que otro tipo de baterías

con la misma capacidad. Además, éstas se pueden conectar unas con otras en paralelo y conseguir así mayor capacidad.

En este tipo de baterías se especifica una referencia de descarga máxima que viene expresada con un número seguido de una 'C'. Para saber qué amperaje descarga la batería se debe multiplicar los miliamperios de ésta por el número que acompaña a la letra C. El descargar la batería con amperajes superiores a los indicados por el fabricante hará que la batería se dañe y sea inestable.

Las baterías LiPo tienen 2 conectores a excepción de las de un solo elemento que tienen 1. Uno de estos conectores se denomina balanceador y se utiliza a la hora de la carga. Se usa para conseguir que todos los elementos se carguen por igual. Si no se utiliza este conector para cargar la LiPo, existen muchas posibilidades de sobrecargar un elemento con respecto a los otros y que a la hora de usar esta batería uno de los elementos se descargue más de la cuenta y se estropee.

La batería que hemos escogido es la que se muestra en la figura 3.8. Se trata de una batería de 11.1V (3S), 2200 ma de capacidad y 25C de descarga. Si multiplicamos la capacidad por la referencia de descarga obtenemos el amperaje que descarga esta batería.

$$I = 2200ma * 25C = 55amperios$$

Este amperaje debe ser suficiente para alimentar los motores y toda la electrónica. Cada uno de los motores tenía un consumo máximo de 4.5 amperios que entre los cuatro suman 18 amperios. Por lo tanto, el amperaje que suministra la batería es más que suficiente para alimentar todo.

Figura 3.8: Batería LiPo 3S 2200 mah.

A continuación se calcula el tiempo de vuelo a máxima potencia que puede estar nuestro dispositivo con la batería que se ha escogido.

$$T.Vuelo = \frac{2{,}2ah}{4{,}5a * 4motores} = 7{,}3minutos$$

Ésta es una de la grandes limitaciones que tienen los multirrotores, el poco tiempo que puede estar volando.

3.1.2.6 Módulo de comunicación

Debido a que nuestro dispositivo va a ser móvil, necesitamos enviarle órdenes a distancia sin ningún tipo de cables. Para conseguirlo se debe implementar una interfaz de control en un dispositivo a distancia con el que se comunicará mediante un sistema inalámbrico.

Para la comunicación con nuestra aeronave hemos escogido el módulo *OPLink mini*. Se trata de un transceptor con un procesador de paquetes digitales ARM32 diseñado específicamente para el proyecto *OpenPilot*. Es un sistema de radio que permite transmitir la información de telemetría en tiempo real hacia la estación de control terrestre, permite configuración inalámbrica e incluso permite controlar nuestra aeronave a partir de un sistema de radio-control. El sistema de radio-control que hemos escogido es *Turnigy 9XR PRO Radio Transmitter* cuyas características las podemos encontrar en la siguiente página: https://www.asturmodel.es/p-282/Emisoras,-Receptores-y-Accesorios/Emisoras-y-Accesorios/Emisora-Turnigy-9XR-PRO--24-canales---Modo-2. Además, en esta página se puede encontrar un manual en inglés.

En nuestro caso usamos estos módulos para el control de nuestro dron [15]. El módulo *OPLink Mini Air* lo conectamos a la aeronave mientras que el *OPLink Mini Ground* lo conectamos a un sistema de radio control. En la figura 3.9 se ilustra la conexión citada anteriormente.

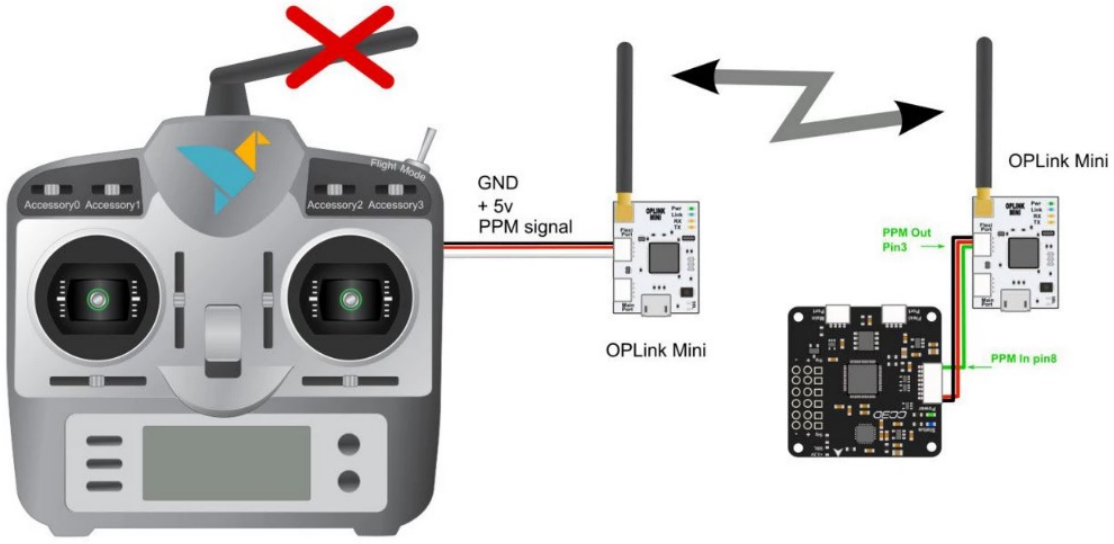


Figura 3.9: Conexión módulos OPLink.

Los módulos *OPLink* están programados para aceptar la modulación PPM[2] proveniente de un sistema de radio-control con entre 4 y 8 canales.

[2]Modulación por posición de pulso. La amplitud y el ancho son fijos y la posición es variable. Usa un mismo cable para enviar la información de todos los canales, uno tras otro.

En la figura 3.10 se describen las especificaciones de los dos módulos.

Specs: Oplink Mini Air
Dimensions:**34mm x19.5mm x10.5mm (included case)**
Frequency:**433mhz**
Input Voltage: **+5v**

Specs: Oplink Mini Ground
Modem: **100mw Standalone Radio Modem**
3 IO Ports: **Micro-B USB, Mainport & Flexiport**
Dimensions: **38mm x23mm x10mm (included case)**
Input Voltage: **+5v**
Frequency:**433mhz**

Figura 3.10: Especificaciones módulos OPLink.

3.1.2.7 Controladora de vuelo

La controladora de vuelo es el elemento más importante de cualquier dron, ya que es el cerebro de éste. Se encarga por ejemplo de recopilar y gestionar los datos que recibe de los sensores o realiza ajustes en la velocidad de giro de los motores para mantener el sistema nivelado y bajo control. En la mayoría de los casos llevan incorporados giróscopos y acelerómetros que se utilizan para controlar la orientación. Además, se les pueden conectar otros dispositivos externos como unidades GPS, sistemas de telemetría y sistemas OSD [3] entre otros.

En este trabajo usamos la controladora denominada *CC3D Mini Revolution* (figura 3.11) [16] que usa la serie STM32F4 de micro-controladores ARM. Contiene una unidad de coma flotante (FPU), que es un paso más en la escala de rendimiento de estas controladoras.

Figura 3.11: CC3D Mini Revolution.

Esta placa es un pequeño ordenador de control de vuelo con piloto automático que está destinada a multirrotores, helicópteros y alas fijas. Lleva incorporado los sensores esenciales para realizar el vuelo como giróscopo, acelerómetro y magnetómetro. Además, incluye un sensor de presión con el que mide la presión atmosférica y con ello, conocer la altura a la que se encuentra la aeronave.

A continuación se realizará una descripción técnica de la placa donde se describirá la CPU, los puertos y los sensores que tiene incorporados.

[3] OSD significa On Screen Display y muestra información útil para el piloto por pantalla como puede ser el estado de la batería o calidad de señal RSSI entre otros.

La CPU es el chip STM32F405RGT6, con núcleo ARM Cortex-M4 en 210MIPS, funciones FPU, y la saturación de la aritmética DSP. El chip cuenta con una gama de módulos de hardware integradas que pueden ser programados una vez y funcionar de forma independiente, lo que requiere poca o ninguna sobrecarga de la CPU. Entre éstos podemos citar: 14 temporizadores multicanal, 3 ADC con muestreo síncrono, 2 DAC, controlador de memoria matriz con DMA, módulos de comunicación como USB 2.0, 3 I2Cs, 3 SPIs, 4 USARTs, 2 CANs y SIDO. Todos estos módulos se pueden activar usando una matriz de conmutación o desactivar para ahorrar energía. El software y la configuración se cargan gracias a un conector USB a través de la función de actualización en el GCS (*Ground Control Station*).

En las siguiente líneas se realizará una breve descripción de cada uno de los puertos que tiene esta placa:

- ***Servo 1-6***: Salidas PWM que van a servos o controladores electrónicos de velocidad (ESC). La alimentación para la controladora de vuelo le llega por este puerto a partir de uno de los ESC que incluya BEC.
- ***Flexi-IO*** : Este puerto puede utilizarse tanto para entrada como para salida, depende de la configuración que se establezca en el GCS.
- ***MainPort***: Se trata de un puerto serie cuya velocidad de transmisión se puede ajustar a través del GCS. Opcionalmente se pueden asignar a este puerto algunos receptores. La configuración que tiene por defecto es de telemetría para la conexión de un módem de RF.
- **FlexiPort**: Este puerto también puede ser configurado y se puede escoger entre comunicación I2C[4] o Serial[5].
- **Pwr Sen/Sonar**: Este puerto puede ser configurado para acomodar a un sensor de corriente o un sonar como el HC-SR04. También se puede utilizar como puerto de entrada/salida de propósito general o como puerto de entrada analógica.

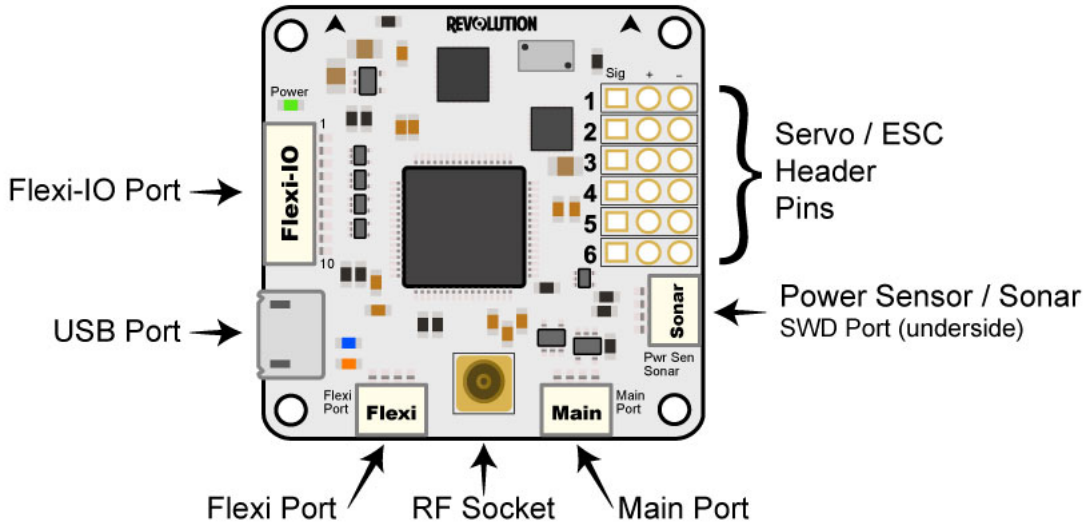


Figura 3.12: Puertos de CC3D Revolution.

[4]Bus bidireccional que utiliza dos líneas, una de datos serie (SDA) y otra de reloj serie (SCL). SCL es la línea de reloj, se utiliza para sincronizar todos los datos. SDA es la línea de datos.

[5]La comunicación serial consiste en el envío de un bit de información de manera secuencial, ésto es, un bit a la vez y a un ritmo acordado entre el emisor y el receptor.

CC3D Mini Revolution incluye un conjunto de sensores que se describen a continuación:

- ***MPU-6000*** (figura 3.13): combina un acelerómetro y un giróscopo de 3 ejes en un mismo encapsulado. El acelerómetro mide la aceleración de las coordenadas (cambio de la velocidad del dispositivo en el espacio). Mide todas las aceleraciones excepto la causada por la gravedad. El giróscopo mide la orientación basándose en los principios del momento angular. Se trata de un objeto que gira sobre sí mismo y cuyo eje es libre de cambiar su orientación. Cuando el giróscopo cambia la orientación del eje de rotación, el objeto que giraba sobre sí mismo cambiará de orientación para intentar seguir en su dirección intuitiva.

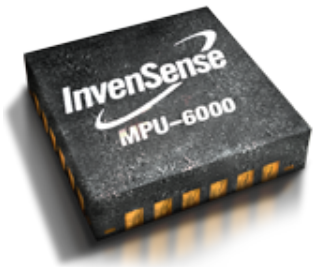


Figura 3.13: MPU-6000.

- **Sensor de presión/altímetro MS5611** (figura 3.14): sensor de presión barométrica que está optimizado para altímetros y barómetros con una resolución de altitud de 10 cm. El módulo incluye un sensor de presión de alta linealidad y un ADC de 24 bits. Su consumo de energía es bastante bajo ($1\mu A$). Proporciona un preciso valor de temperatura y de presión digital de 24 bits y diferentes modos de funcionamiento que permiten al usuario optimizar la velocidad de conversión y el consumo de corriente.

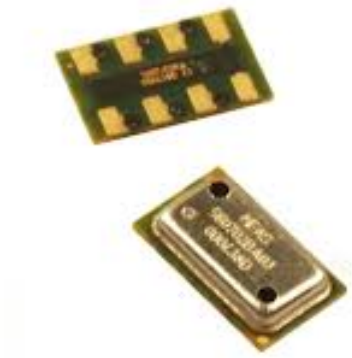


Figura 3.14: Sensor de presión/Altímetro.

- **Magnetómetro HMC5883L** (figura 3.15): Este módulo es un magnetómetro de 3-ejes que se puede utilizar como brújula o compás digital. Su interfaz de comunicación es de tipo I2C por lo que puede conectarse a una amplia gama de sistemas digitales. El sensor detecta el campo magnético más fuerte, que comúnmente es el campo magnético de la Tierra, a partir del cual se puede encontrar el Norte Magnético. Tiene una precisión de 1 a 2 grados sexagesimales, un ADC de 12 bits y una resolución de 5 mili-Gauss.

Figura 3.15: Magnetómetro.

3.1.2.8 Sensor HC-SR04

El sensor HC-SR04 [17] utiliza señales ultrasónicas para medir la distancia existente hacia objetos presentes, su rango de detección es de 2cm a 4m y tiene una precisión de 3mm. En la figura 3.16 se especifican las características técnicas de este sensor.

Características eléctricas.

- Voltaje de operación: 5V DC
- Consumo de corriente: 15mA
- Frecuencia de operación: 40Hz
- Rango de medición: 2cm – 4m
- Angulo de medición: 15°
- Señal de entrada Trigger: 10us TTL
- Señal de salida Echo: TTL, correspondiente a distancia medida

Figura 3.16: Características eléctricas de HC-SR04.

En la figura 3.17 se muestran los pines de conexión de este módulo.

Figura 3.17: Pines de consexión HC-SR04.

Su funcionamiento es el siguiente: Enviamos una señal a nivel alto de al menos $10\mu s$ al pin *Trigger* del módulo. El emisor del sensor envía automáticamente una ráfaga de impulsos (8 ciclos de 40 KHz) al recibir el pulso de disparo (*trigger*). Los impulsos emitidos viajan a la velocidad del sonido. A continuación empieza a contar el tiempo que tarda en llegar el eco. Este tiempo se traduce en un pulso de salida, en

el pin *Echo*, cuya duración es proporcional a la distancia a la que se encuentra el objeto. Después se lee el pulso de salida de *Echo* y en caso de que no se produzca ningún eco, este pulso tendrá una longitud aproximada de 36 ms. Una vez medido el ancho de pulso de la señal *Echo*, se puede calcular la distancia de acuerdo a las siguientes fórmulas[6] proporcionadas por la hoja de datos:

$$Distancia(cm) = \frac{\mu s}{58}$$

En la figura 3.18 se representa lo explicado en el párrafo anterior.

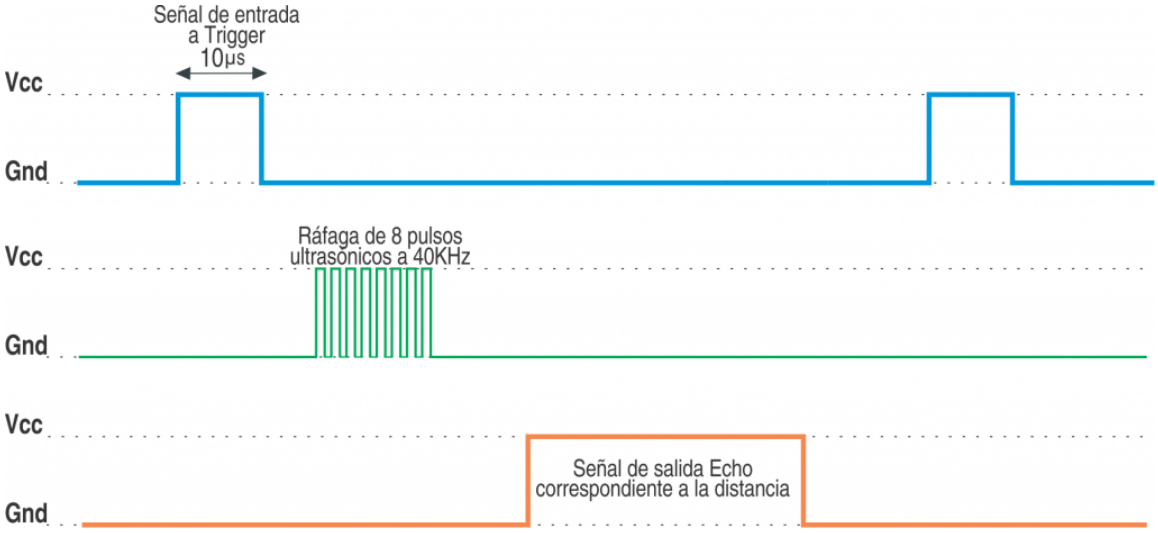


Figura 3.18: Diagrama de tiempo de HC-SR04.

3.1.2.9 NanoPi NEO Air

NanoPi Neo Air [18] es una tarjeta de ARM de código abierto. Utiliza un procesador *Allwinder H3 Quad Core A7* a 1.2GHz. Cuenta con 512 MB de RAM, 8 GB eMMC y una ranura *MicroSD*. Tiene *WiFi*, *Bluetooth* e interfaz de cámara DVP (YUV422). La interfaz de cámara DVP puede soportar el módulo de cámara de 5 Megapíxeles de *friendlyarm*. Además, tiene un diseño de circuito de potencia mejorado y una mejor disipación del calor.

Figura 3.19: NanoPi Neo Air.

En la figura 3.20 se definen las especificaciones hardware de esta tarjeta.

[6]distancia = (tiempo a nivel alto * velocidad del sonido (340m/s)) / 2

- CPU: Allwinner H3, Quad-core Cortex-A7 Up to 1.2GHz
- RAM: 512MB DDR3 RAM
- Storage: 8GB eMMC
- WiFi: 802.11b/g/n
- Bluetooth: 4.0 dual mode
- DVP Camera: 0.5mm pitch 24 pin FPC seat
- MicroUSB: OTG and power input
- MicroSD Slot x 1
- Debug Serial Port: 4Pin,2.54mm pitch pin header
- GPIO1: 2.54mm spacing 24pin,It includes UART,SPI,I2C,GPIO
- GPIO2: 2.54mm spacing 12pin,It includes USBx2,IR,SPDIF,I2S
- PCB Size: 40 x 40mm
- PCB layer: 6
- Power Supply: DC 5V/2A
- Working Temperature: -40℃ to 80℃
- OS/Software: u-boot, UbuntuCore, eflasher
- Weight: 7.5g(WITHOUT Pin-headers)

Figura 3.20: Especificaciones Hardware NanoPi Neo Air.

En las figuras 3.21 y 3.22 se muestra el diseño de esta tarjeta y el diagrama de pines respectivamente.

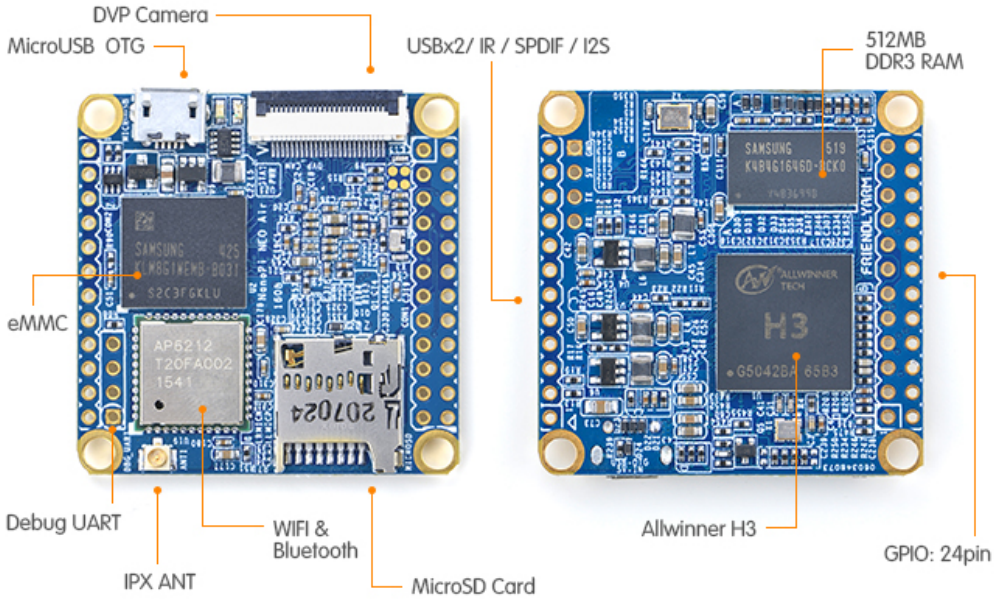


Figura 3.21: Diseño NanoPi Neo Air.

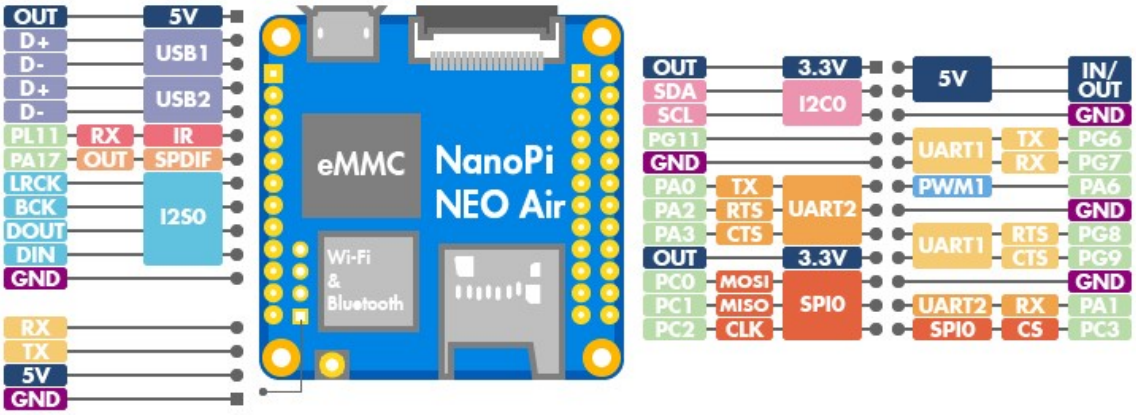


Figura 3.22: Diagrama de pines NanoPi Neo Air.

3.1.2.10 CAM500B

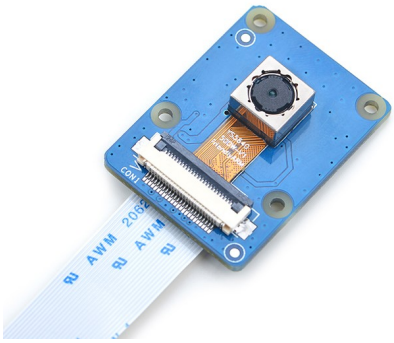


Figura 3.23: Cámara CAM500B.

Esta cámara utiliza el módulo QSXGA CMOS OV5640 de *Omni Vision*. Tiene una interfaz de salida DVP. Soporta imágenes de tamaño 2592x1944 y grabaciones de vídeo 720p a 30fps. Tiene funciones automáticas de control de imagen como AFC, AWB y AEC, etc. La longitud focal de esta cámara es de 2,7 mm, su abertura es 2.8 y su ángulo de visión es de 66 grados.

La conexión con *NanoPi Neo Air* se realiza como se puede ver en la figura 3.24 .

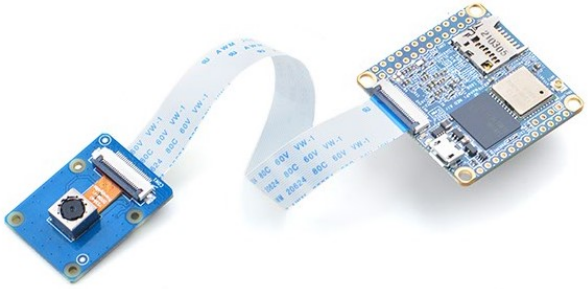


Figura 3.24: Conexión CAM500B a NanoPi Neo Air.

3.1.3 Estudio de la propulsión

En este apartado se estimará la carga que deben de mover los motores. En la tabla se relacionan los componentes con el peso de cada uno de ellos y, la suma será la carga que deben levantar los motores.

Tabla 3.1: Tabla que relaciona los componentes con sus pesos.

Componente	**Unidad**	Peso	Total
Frame	1	110g	110g
Motor	4	15.5g	62g
Hélices	4	2g	8g
Batería	1	188g	188g
ESC	4	10g	40g
Mini CC3D Revolution	1	5g	5g
OPLink Mini Air	1	5g	5g
HC-SR04	1	5g	5g
NanoPi Neo Air	1	20g	20g
Total			443g

Sabemos que el peso es una fuerza, y las unidades de la fuerza son los newtons(N), que son $Kg * \frac{m}{s^2}$. Los Kg es la masa que hemos obtenido de la aeronave, pero los $\frac{m}{s^2}$ son la aceleración de la gravedad ($9{,}81Kg * \frac{m}{s2}$), lo que significa que con ese empuje lo único que conseguiremos es compensar la aceleración de la gravedad, pero no mover el aparato arriba o abajo. Nuestro dron debe poder ascender con una aceleración de igual valor a la gravedad, es decir, que pueda sustentarse en el aire llevando una carga igual a su peso.

El empuje que debe realizar cada uno de los motores se consigue a partir de la siguiente fórmula:

$$E_{motor} = \frac{P_{aeronave}}{4} * 2 = \frac{443}{4} * 2 = 221g$$

El resultado anterior es el empuje en gramos que debe crear cada uno de los motores para poder mover la aeronave. Según el fabricante, los motores elegidos tienen un empuje de 263.28 a su máxima potencia. Como vemos, estos motores nos dan el empuje necesario para mantener en el aire la aeronave.

3.2 Estructura de control

El proyecto *LibrePilot* utiliza la estructura de control que se muestra en el siguiente diagrama:

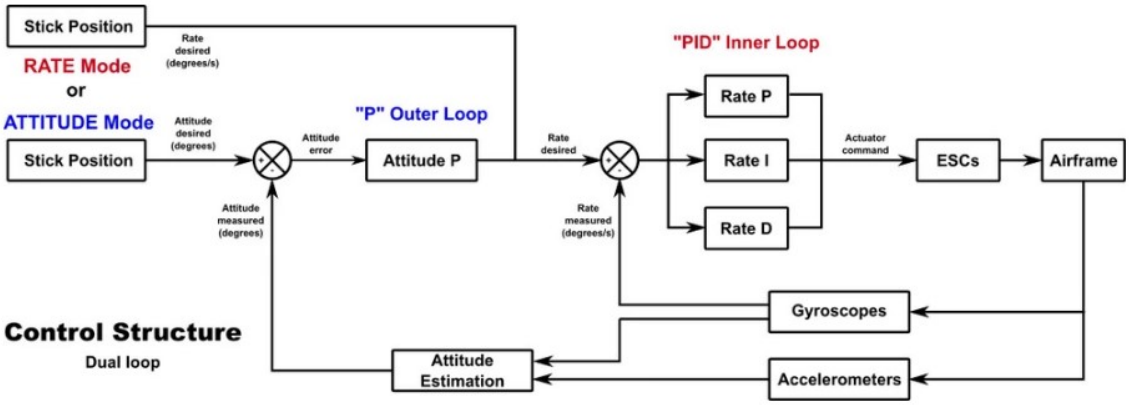


Figura 3.25: Estructura de control.

El **bucle de estabilización de la posición** (el bucle externo) compara el ángulo de inclinación de un eje con la posición deseada (posición del stick) del transmisor y envía una señal al bucle de estabilización de la velocidad siendo ésta la diferencia entre los dos.

El **bucle de estabilización de la velocidad** (el bucle interno) compara la velocidad de rotación medida por los giróscopos de ese eje con una señal de control.

- Si se selecciona el modo de vuelo *Rate* para algún eje, la señal de control para el modo *Rate* proviene directamente del transmisor, sin pasar por el bucle externo. Cuando los sticks se encuentran en la posición neutral el vehículo mantendrá el ángulo actual mientras que si se encuentra en su rango completo, el vehículo rotará los grados definidos en el modo *Rate*. Es responsabilidad del propio piloto de volver a nivelar la aeronave.

- Si se selecciona el modo de vuelo *Attitude* para algún eje, la señal de control para el bucle de velocidad proviene del bucle de estabilización de la posición. Cuando los sticks se encuentran en la posición neutral, el vehículo se mantendrá en un vuelo nivelado.

3.3 Desarrollo Software

Este apartado se divide en 4 subapartados. En el primero de ellos se describen las modificaciones realizadas al proyecto *LibrePilot* para conseguir la inclusión de un sensor de ultrasonidos y usarlo conjuntamente con el barómetro. En el segundo se describe el proceso necesario para realizar el seguimiento de un objeto mediante funciones de la librería *OpenCV*. En tercer lugar, se describe la comunicación que se realiza entre la controladora de vuelo y la tarjeta *NanoPi Neo Air* para conseguir pasar los datos necesarios de una a otra. Y por último, se explica el procesamiento que se realiza de los datos obtenidos.

3.3.1 Modo *AltitudeHold*

El software de *LibrePilot* dispone de un modo de vuelo denominado *AltitudeHold*. Cuando la aeronave entra en este modo de vuelo, tomará el valor de la altura que mide en ese instante y mediante un control PID intentará mantenerse a esa altura. Ésta es tomada gracias a un barómetro que medirá la presión atmosférica y la temperatura y, gracias a una conversión, obtendrá la altura en la que se encuentra la aeronave en cada instante. Este sensor es realmente sensible a la luz y a los cambios de iluminación, lo que se traduce en grandes saltos en la medición de la altitud.

El software de *LibrePilot* dispone de una gráfica donde se representan los valores de la altitud obtenidos por este sensor. En la figura 3.26 se muestra un ejemplo de esta representación.

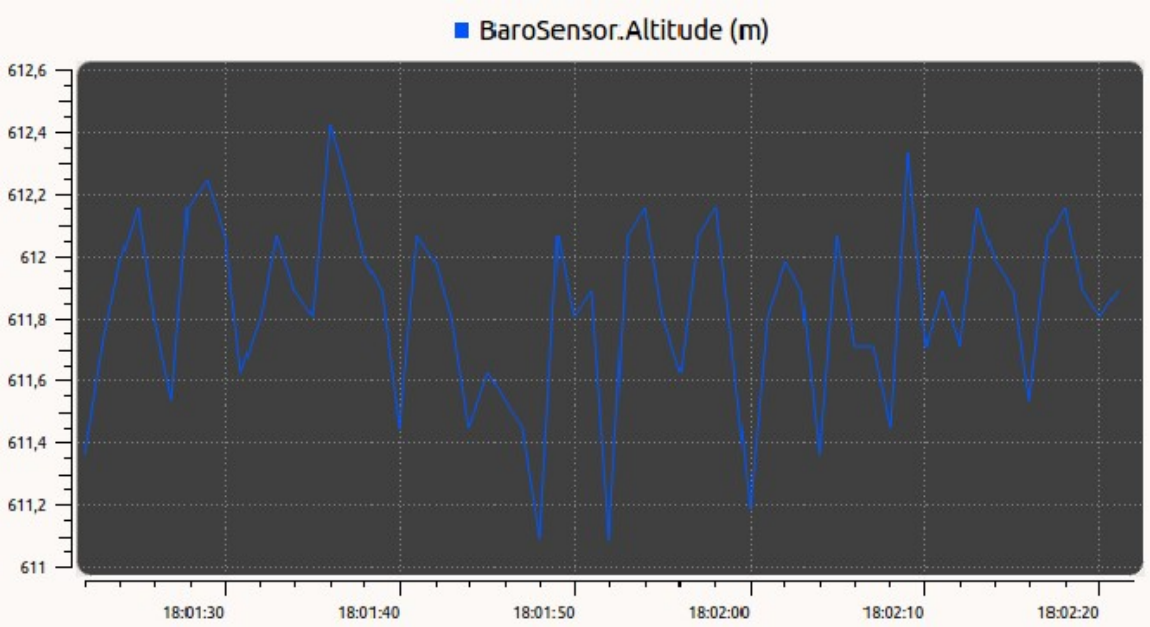


Figura 3.26: Gráfica en la que se muestran los valores de altitud obtenidos por el barómetro.

Como se puede observar en la gráfica, no es un valor demasiado estable y como resultado, veremos ciertos saltos en el vuelo. Para resolver este problema se decide implantar un sensor de ultrasonidos (sonar) que mida la distancia existente hacia la superficie sobre la que se encuentre el dron. Este sensor tiene un rango de funcionamiento y por ese motivo cuando nos encontramos fuera de este rango, el sensor no realizará ningún aporte al control de la altura.

El barómetro no conoce el relieve de la superficie y por ello, no le indica al dron que debe de aumentar la altitud cuando la superficie se eleva. El sonar realiza esa función y le indica al dron que debe elevar su altitud para evitar una colisión con el terreno. Por otro lado, el sonar le indicará al dron que debe disminuir su altitud cuando el terreno disminuye su nivel.

Para conseguir obtener las medidas de la altura a partir del sensor de ultrasonidos se deben realizar unas modificaciones al proyecto *LibrePilot* ya que éste no tiene el soporte para dicho sensor. En primer lugar, se deben definir el puerto y los pines que se van a utilizar de la controladora. En nuestro caso usamos el puerto Flexi-io y los pines 8 y 9. El primero de ellos lo definimos como entrada para el pin *Echo* del sonar, mientras que el segundo lo definimos como salida para enviar el *Trigger* necesario para iniciar el proceso. Ésto se define en el fichero *board-hw-defs.c* que se encuentra en la dirección *librepilot/flight/targets/board/revolution* del proyecto. En este fichero se encuentran definidos los puertos y pines que se utilizan en este proyecto y para esta controladora en concreto. La modificación que realizamos se muestra a continuación:

Listado 3.1: Definición de pines para el sonar.

```
#if defined(PIOS_INCLUDE_HCSR04)
#include <pios_hcsr04_priv.h>
#include <hwsettings.h>
#include <pios_tim_priv.h>

static const struct pios_hcsr04_cfg pios_hcsr04_cfg = {
    .tim_ic_init = {
        .TIM_ICPolarity  = TIM_ICPolarity_Rising,
        .TIM_ICSelection = TIM_ICSelection_DirectTI,
        .TIM_ICPrescaler = TIM_ICPSC_DIV1,
        .TIM_ICFilter    = 0x0,
         },

    .channel = {
        .timer = TIM8,
        .timer_chan = TIM_Channel_3,
        .pin  = {
           .gpio = GPIOC,
           .init = {
           .GPIO_Pin  = GPIO_Pin_8,
           .GPIO_Mode  = GPIO_Mode_AF,
           .GPIO_Speed = GPIO_Speed_25MHz,
           .GPIO_PuPd  = GPIO_PuPd_DOWN
        },
        .pin_source = GPIO_PinSource8,
        },

    .remap = GPIO_AF_TIM8,
     },

    .gpio = GPIOC,
    .gpioInit = {
        .GPIO_Pin = GPIO_Pin_9,
        .GPIO_Speed = GPIO_Speed_2MHz,
        .GPIO_Mode = GPIO_Mode_OUT,
        .GPIO_PuPd = GPIO_PuPd_UP,
        .GPIO_OType = GPIO_OType_PP,
        },
};
#endif /* if defined(PIOS_INCLUDE_HCSR04) */
```

Tras tener definidos los pines que vamos a usar, se crea una función en la que se lance el *Trigger* que consiste en poner un pin a nivel alto durante 15 microsegundos. Ésto lo realizamos en la siguiente función que se encuentra en el fichero *pios-hcsr04.c* de la carpeta *librepilot/flight/pios/common*

Listado 3.2: Función en la que se lanza el Trigger.

```
bool PIOS_HCSR04_driver_poll(uintptr_t context)
{
    struct pios_hcsr04_dev *hcsr04_dev = (struct pios_hcsr04_dev *)context;

    if (!PIOS_HCSR04_validate(hcsr04_dev)) {
        /* Invalid device specified */
        return false;
```

```
    }

    bool done = ( hcsr04_dev->CaptureState == HCSR04_CAPTURE_Done );

    if( done )
    {
        hcsr04_dev->Timeout = HCSR04_TIMEOUT_RTC_TICKS;
        PIOS_HCSR04_set_capture_state(hcsr04_dev, HCSR04_CAPTURE_Rising);

        GPIO_SetBits(hcsr04_dev->pios_hcsr04_cfg->gpio, hcsr04_dev->pios_hcsr04_cfg->
            gpioInit.GPIO_Pin);
        PIOS_DELAY_WaituS(15);
        GPIO_ResetBits(hcsr04_dev->pios_hcsr04_cfg->gpio, hcsr04_dev->pios_hcsr04_cfg
            ->gpioInit.GPIO_Pin);
    }
    return done;
}
```

Para verificar si la función anterior realiza el disparo que debemos obtener en dicho pin, realizamos una prueba encendiendo y apagando un led.

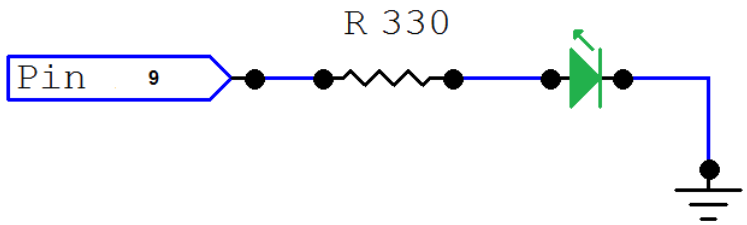


Figura 3.27: Esquema de conexión de un led.

La conexión que se realizó para dicha prueba es la que se observa en la figura 3.27. Como podemos observar, conectamos el led a través de una resistencia de 330 ohmios al pin 9 del puerto *Flexi I/O* de la controladora de vuelo. Para realizar la prueba, se dejó un tiempo de conmutación de un segundo para así, conseguir ver a simple vista que el led se encendía y se apagaba.

Una vez verificado y lanzado el *Trigger* se pone el pin del *Echo* a nivel alto y se inicia un temporizador. Una vez que llegue el eco del disparo, el pin anterior se pone a nivel bajo y con ello se interrumpe el temporizador. Éste nos indicará el tiempo que ha tardado el eco en ir y volver, gracias a ésto se obtendrá la distancia a la que se encuentra el objeto con el que ha chocado la onda. Si por alguna razón no se recibe el eco pasado un tiempo se pone el pin de *Echo* a nivel bajo para esperar una nueva recepción.

Todo lo explicado en el párrafo anterior se encuentra desarrollado en las funciones que mostramos a continuación que están ubicadas en el fichero mencionado con anterioridad.

Listado 3.3: Función en la que si pasado un tiempo no llega el eco da por finalizada la captura.

```
static void PIOS_HCSR04_Supervisor(uint32_t context)
{
    struct pios_hcsr04_dev *hcsr04_dev = (struct pios_hcsr04_dev *)context;


    if (!PIOS_HCSR04_validate(hcsr04_dev)) {
        /* Invalid device specified */
        return;
```

```
    }

    if( hcsr04_dev->CaptureState != HCSR04_CAPTURE_Done ) {

        //hcsr04_dev->Timeout--;
        if(hcsr04_dev->Timeout > 0){
                hcsr04_dev->Timeout--;
        }

        if( hcsr04_dev->Timeout == 0 ) {
            hcsr04_dev->CaptureState    = HCSR04_CAPTURE_Done;
            hcsr04_dev->RiseValue       = 0;
            hcsr04_dev->CaptureValue    = PIOS_RCVR_TIMEOUT;
        }
    }
}
```

Listado 3.4: Función en la que configura el estado de captura.

```
static void PIOS_HCSR04_set_capture_state(struct pios_hcsr04_dev *hcsr04_dev,
    HCSR04_CAPTURE_State state)
{
    hcsr04_dev->CaptureState = state;

    const struct pios_tim_channel *chan   = &hcsr04_dev->pios_hcsr04_cfg->channel;

    TIM_ICInitTypeDef TIM_ICInitStructure = hcsr04_dev->pios_hcsr04_cfg->tim_ic_init;

    TIM_ICInitStructure.TIM_ICPolarity = state == HCSR04_CAPTURE_Falling ?
        TIM_ICPolarity_Falling : TIM_ICPolarity_Rising;
    TIM_ICInitStructure.TIM_Channel    = chan->timer_chan;
    TIM_ICInit(chan->timer, &TIM_ICInitStructure);
}}
```

Listado 3.5: Función en la que se calcula el ancho del flanco del pin *Echo*

```
static void PIOS_HCSR04_tim_edge_cb(__attribute__((unused)) uint32_t tim_id, uint32_t
    context, uint8_t chan_idx, uint16_t count){
    /* Recover our device context */
    struct pios_hcsr04_dev *hcsr04_dev = (struct pios_hcsr04_dev *)context;

    if (!PIOS_HCSR04_validate(hcsr04_dev)) {
        /* Invalid device specified */
        return;
    }
    if (chan_idx >= 1) {
        /* Channel out of range */
        return;
    }

    if(hcsr04_dev->CaptureState == HCSR04_CAPTURE_Done) {
            return;
    }

        const struct pios_tim_channel *chan = &hcsr04_dev->pios_hcsr04_cfg->channel;
```

```
            if (hcsr04_dev->CaptureState == HCSR04_CAPTURE_Rising) {
                hcsr04_dev->RiseValue = count;
                /* Switch states */
                PIOS_HCSR04_set_capture_state(hcsr04_dev, HCSR04_CAPTURE_Falling);
            } else { // HCSR04_CAPTURE_Falling

                /* Capture computation */
                if (count > hcsr04_dev->RiseValue) {
                    hcsr04_dev->CaptureValue = (count - hcsr04_dev->RiseValue);
                } else {
                    hcsr04_dev->CaptureValue = ((chan->timer->ARR - hcsr04_dev->
                        RiseValue) + count);
                }
                /* Switch states */
                PIOS_HCSR04_set_capture_state(hcsr04_dev, HCSR04_CAPTURE_Done);
            }
}
```

Para finalizar se describirán las dos rutinas de interrupción: la del sonar y la del barómetro que se encuentran en el fichero *sensors.c* de la carpeta *librepilot/flight/module/sensors*.

En la rutina de interrupción del sonar se verifica que la lectura se encuentre dentro del rango de medidas del sensor. Una vez verificado, actualiza el valor del *UAVObject* denominado *currentSonar* que después, utilizará en la rutina de interrupción del barómetro. A continuación se muestra el código de la rutina de interrupción del sonar.

Listado 3.6: Rutina de interrupción del Sonar.

```
static float currentSonar = 0;
static int rango = 0;

#if defined(PIOS_INCLUDE_HCSR04)
static void handleSonar(float sample)
{

    // 4 meters equals 23200 uS, and we will consider usable range of slightly over
        60% of that
    // filtering for invalid / settling is done in driver if/as needed for the type of
         hardware
    if ((sample > 100) && (sample < 15000)) {

        rango = 1;

        // convert the sample from usecs to meters
        currentSonar = sample * 0.00034f / 2.0f;

        // update sonar altitude by the factor that occurs due to longer sonar
            distances at high bank angles
        // to produce the corrected altitude use: angle = acosf(cos_lookup_deg(
            attitude->Roll) * cos_lookup_deg(attitude->Pitch));
        // then: factor = cos(angle)
         //or use: quaternion to axis angle

        AttitudeStateData attitudeState;
        AttitudeStateGet(&attitudeState);

        // it might be good to subtract something from the angle because a wide beam
            could still work at over 90 degrees bank angle
```

```
        if (fabsf(attitudeState.Roll) < 89.0f && fabsf(attitudeState.Pitch) < 89.0f) {
            // for a given ping length the actual altitude goes to zero as the bank
                angle goes to 90
            currentSonar *= cos_lookup_deg(attitudeState.Roll) * cos_lookup_deg(
                attitudeState.Pitch);

        } else {
            currentSonar = 0.0f;
        }
    }else {
        rango = 0;
        currentSonar = 0;
    }
    // Update the SonarAltitude UAVObject
    SonarAltitudeAltitudeSet(&currentSonar);
}
#endif /* defined(PIOS_INCLUDE_HCSR04) */
```

En la rutina de interrupción del barómetro se lleva a cabo la conversión de la presión atmosférica, que se obtiene con el barómetro, a valores de altitud sobre el nivel del mar a la que se encuentra nuestra aeronave en cada instante. Se guarda el valor inicial de la altitud en una variable y sumándole la medida del sonar, obtenemos la altitud que va a usar la aeronave para poder mantener el vuelo a una cierta altura. Si en algún momento el sonar no estuviese tomando medidas dentro de su rango efectivo, la aeronave usaría únicamente las medidas tomadas por el barómetro para mantener la altura deseada. A continuación se muestra el código de la rutina de interrupción del barómetro.

Listado 3.7: Rutina de interrupción del barómetro.

```
static uint32_t baroNeedsInit = 500;
static float initialSample = -1.0f;

static void handleBaro(float sample, float temperature)
{
         if (baroNeedsInit > 1) {
             --baroNeedsInit;
         }
     updateBaroTempBias(temperature);
     sample -= baro_temp_bias;
     float currentBaro = 44330.0f * (1.0f - powf((sample) /
         PIOS_CONST_MKS_STD_ATMOSPHERE_F, (1.0f / 5.255f)));

    if (baroNeedsInit == 1){
        if (initialSample < 0.0f){
                initialSample = sample;
        }
    }

    if (!isnan(currentBaro)) {
        BaroSensorData data;

        if (rango == 1){
                updateBaroTempBias(temperature);
                initialSample -= baro_temp_bias;
                float initialBaro = 44330.0f * (1.0f - powf((initialSample) /
                    PIOS_CONST_MKS_STD_ATMOSPHERE_F, (1.0f / 5.255f)));
                data.Altitude = initialBaro + currentSonar;
```

```
        }else{
                data.Altitude = currentBaro;
        }
        data.AltitudeRaw = currentBaro;
        data.Temperature = temperature;
        data.Pressure    = sample;
        // Update the BaroSensor UAVObject
        BaroSensorSet(&data);
    }
}
```

En la figura 3.28 se puede observar la gráfica donde se representan los valores de la altitud que se obtienen con el uso de los dos sensores conjuntamente. En este caso los datos obtenidos son más estables y con esto conseguimos mejores resultados.

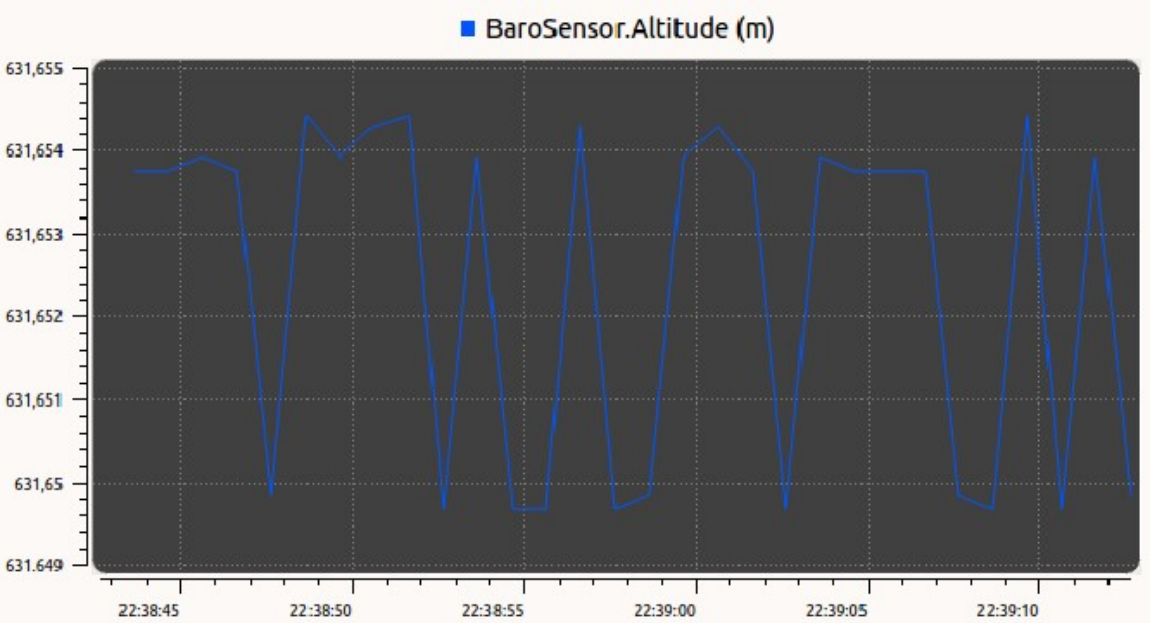


Figura 3.28: Gráfica en la que se muestran los valores de altitud obtenidos a partir de uso conjunto de los dos sensores.

3.3.2 Seguimiento de un objeto con *OpenCV*

En este apartado se explica cómo se puede detectar un objeto basándonos en su color usando la librería *OpenCv*. Esta técnica se usa para detectar objetos con un color uniforme y con ello, extraer las coordenadas en las que se encuentre para posteriormente resaltarlo.

La detección y segmentación de objetos en una imagen es la tarea más importante y desafiante de la visión por computador. El método que se ha escogido en este trabajo para detectar y segmentar un objeto es el basado en la segmentación por color. El objeto y el fondo deben de tener una diferencia significativa de color para poder separarlos.

3.3.2.1 Captación de imágenes en RGB

Para poder llevar a cabo la segmentación de un objeto debemos de obtener las imágenes a partir de una cámara, en nuestro caso, el dispositivo *CAM500B*. Dicha cámara toma cuatro muestras y las guarda

temporalmente en el fichero */dev/shm* de la *NanoPI Neo Air*. Para evitar quedarnos sin espacio en la memoria y ralentizar los procesos, se propone sobrescribir las muestras, con lo que accediendo a dicho fichero siempre obtendremos las últimas cuatro muestras que se hayan captado con la cámara. Dicha captación se realizará con la ayuda del proyecto *MJPG-streamer*

MJPG-Streamer es una aplicación que se ejecuta por línea de comandos. A grandes rasgos, se encarga de obtener *frames JPG* (imágenes) capturadas desde una cámara compatible y transmitirlas como *M-JPEG* (secuencia de vídeo) mediante el protocolo *HTTP* para poder visualizarlo en navegadores, *VLC* y otras herramientas. Además, posee la posibilidad de guardar las imágenes en el fichero que nosotros elijamos. Su funcionamiento se basa en unos *plugins* de entrada y de salida, es decir, un *plugin* (de entrada) copia las imágenes *JPEG* a un directorio de acceso global, mientras que otro *plugin* (de salida) procesa las imágenes, sirviéndolas como un simple fichero de imagen, o bien, emite las mismas de acuerdo a los estándares existentes.

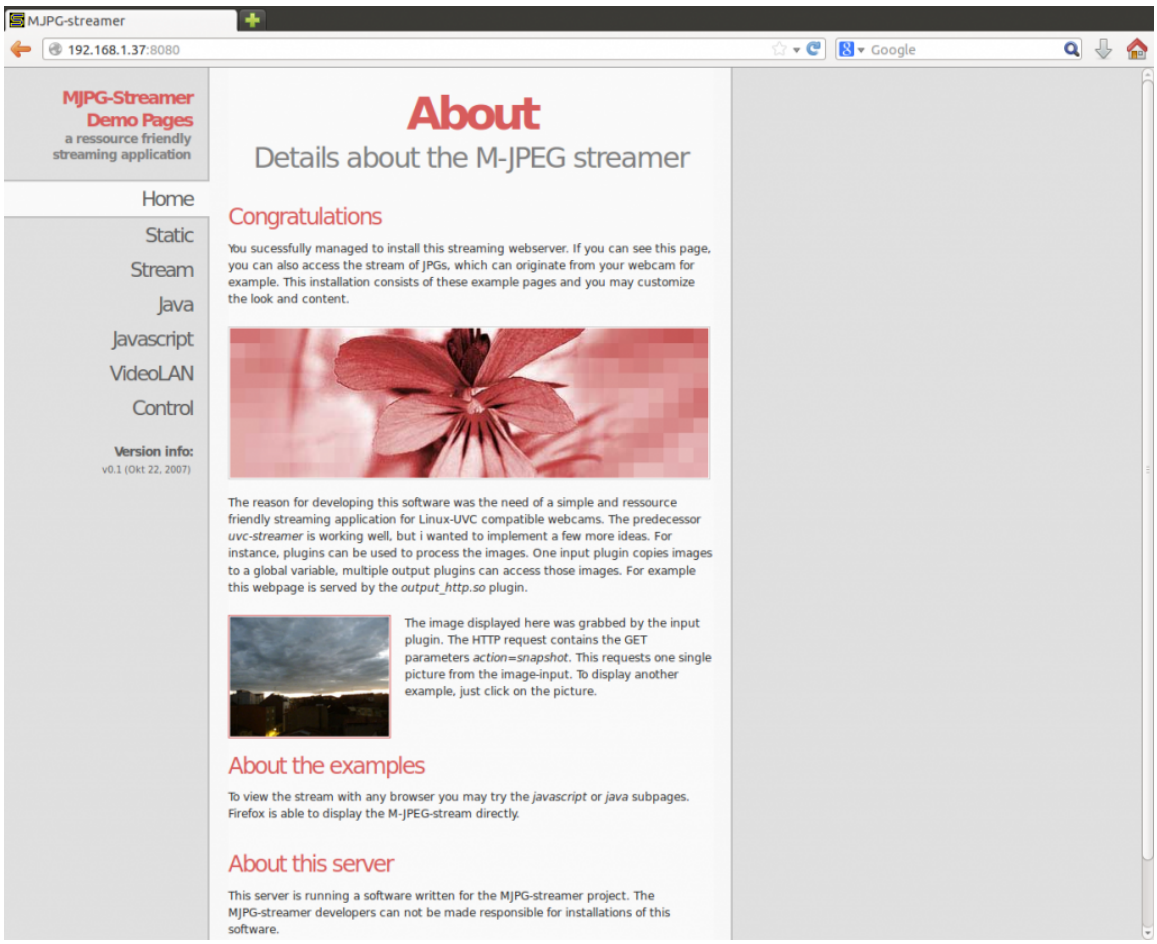


Figura 3.29: Aplicación web de la herramienta MJPG-Streamer.

3.3.2.2 Proceso de la segmentación

Tras obtener las muestras captadas por la cámara, lo primero que se hace es cambiar el espacio de color de RGB[7] a HSV, de esta manera se puede establecer un filtro para obtener sólo aquellas secciones de la imagen cuyo color se encuentre en el rango de color definido. La diferencia entre un espacio de color y otro es que, RGB representa un color mediante la mezcla por adición de los tres colores de luz

[7]RGB (sigla en inglés de *red*, *green*, *blue*, en español «rojo, verde y azul»). Se trata de un modelo de color basado en la síntesis aditiva, con el que es posible representar un color mediante la mezcla por adición de los tres colores de luz primarios.

primarios (rojo,verde y azul), mientras que el modelo HSV, define un modelo de color en términos de sus componentes y por lo que se puede obtener fácilmente el color que se desea.

El espacio de color HSV se compone de 3 matrices, *HUE, SATURATION y VALUE* con rango de valores de 0-179, 0-255 y 0-255 respectivamente. *HUE* representa el color, *SATURATION* representa diferencia del color respecto a un gris con la misma intensidad, cuanto más diferente, más saturado, y por último, *VALUE* que representa la cualidad de ser más claro o más oscuro.

Para realizar la conversión usamos la función *cvtColor(imgOriginal, imgHSV, COLOR-BGR2HSV)* pasándole como parámetros la imagen de origen, la imagen destino y el tipo de conversión que se desee, en este caso *COLOR-BGR2HSV*. Para filtrar sólo aquel color que deseamos obtener en la segmentación usamos la función *inRange*, que en nuestro caso se filtrará el color rojo con rangos superior e inferior definidos por los siguientes valores:

```
iLowH = 134;
iHighH = 179;
iLowS = 124;
iHighS = 244;
iLowV = 113;
iHighV =255;
```

Estos son los valores aproximados. *SATURATION Y VALUE* van a depender de la condición de iluminación del entorno, así como de la superficie del objeto a detectar.

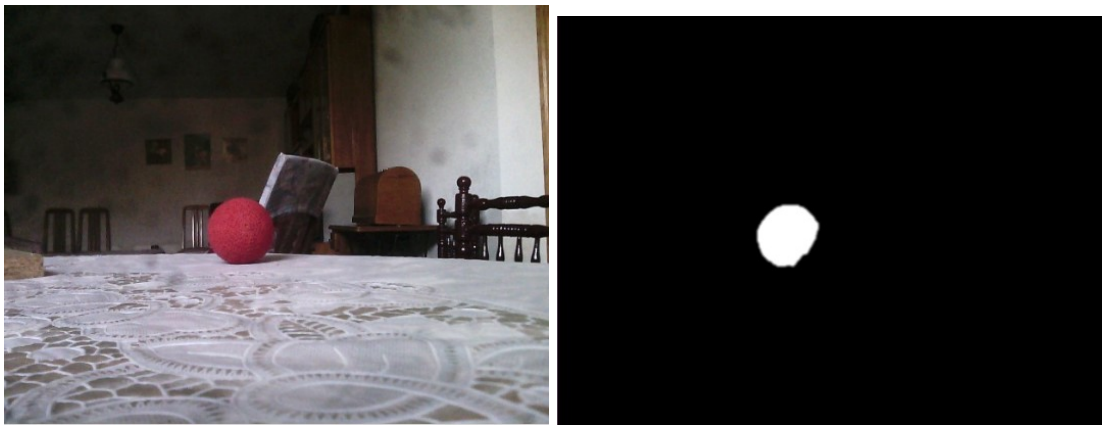
Para descartar aquellos segmentos no deseados se usan las transformaciones morfológicas *erode y dilate*, y se configuran para buscar elementos circulares en nuestra imagen binaria. Estos pequeños segmentos que pueden aparecer son debidos a ruido en la imagen u objetos pequeños reales que tienen el mismo color que nuestro objeto. Para eliminar los pequeños objetos que se detecten se usa la aplicación de cierre morfológico que se puede lograr mediante una erosión seguido de una dilatación con el mismo elemento estructurante. Por otro lado, también se deben eliminar los pequeños agujeros que aparezcan mediante la aplicación de cierre morfológico que se consigue mediante una dilatación seguida de una erosión.

Listado 3.8: Segmentación de un objeto

```
//Obtener frame de la cam
cap.open("/dev/shm/picture_%d.jpg");

if (!cap.isOpened()){  // Si no se consigue, salir del programa
        cout << "Cannot open the web cam" << endl;
        return -1;
}

// convertir imagen RGB a HSV
cvtColor(imgOriginal, imgHSV, COLOR_BGR2HSV);

// Aplica el filtro para color deseado
inRange(imgHSV,Scalar(iLowH, iLowS, iLowV), Scalar(iHighH, iHighS, iHighV),
    imgThresholded);


// Aplicar transformaciones morfológicas
//Apertura mofológica (eleimina pequeños objetos del fondo)
erode(imgThresholded, imgThresholded, getStructuringElement(MORPH_ELLIPSE,Size(2,2)));
```

```
dilate(imgThresholded,imgThresholded, getStructuringElement(MORPH_ELLIPSE, Size(2,2)));


//Cierre morfológico (elimina pequeños agujeros del fondo)
dilate(imgThresholded,imgThresholded, getStructuringElement(MORPH_ELLIPSE, Size(2,2)));

erode(imgThresholded, imgThresholded, getStructuringElement(MORPH_ELLIPSE, Size(2,2)));
```

Una vez realizadas estas transformaciones tendremos a nuestro objeto (en nuestro caso una pelota) segmentado del resto. En la nueva imagen veremos a nuestro objeto de color blanco y el resto de color negro. En la figura 3.30 podemos ver la transformación citada anteriormente.



(a) Imagen captada por la cámara. (b) Imagen segmentada.

Figura 3.30: Resultado de aplicar, a una imagen captada por la cámara del dron, las funciones descritas anteriormente.

3.3.2.3 Obtención de parámetros

Lo siguiente que se debe hacer es encontrar las coordenadas donde se sitúa nuestro objeto de interés. Para ello se usan los momentos, con los que se calcula la posición central de éste. *OpenCv* dispone de una función denominada *moments* con la que se calcula todos los momentos espaciales de hasta tercer orden. Se utiliza el momento espacial de primer orden para calcular las coordenadas del eje X e Y en las que se encuentra el objeto deseado, y los momentos centrales de orden 0 para el área. En nuestra aplicación se ha acordado que si el área blanca de nuestra imagen binaria es inferior o igual a los 100000 píxeles, no se considera que se esté detectando nuestro objeto.

Para finalizar, se imprimen por pantalla los parámetros de cada uno de los círculos detectados (figura 3.31).

```
Ball Position x = 342.2 y = 72.4 r = 92.8609 d = 36.1831
Ball Position x = 347.2 y = 114.2 r = 86.8915 d = 38.6689
Ball Position x = 349 y = 107.4 r = 83.2496 d = 40.3605
Ball Position x = 356.4 y = 95.4 r = 84.8086 d = 39.6186
Ball Position x = 348.4 y = 114.8 r = 94.8979 d = 35.4065
Ball Position x = 354.8 y = 100 r = 85.1915 d = 39.4406
Ball Position x = 362.6 y = 94.2 r = 90.1061 d = 37.2894
Ball Position x = 334 y = 109.4 r = 83.2274 d = 40.3713
Ball Position x = 354 y = 76 r = 80.9365 d = 41.514
Ball Position x = 334.4 y = 104.4 r = 79.1527 d = 42.4496
Ball Position x = 356.2 y = 63.8 r = 87.2014 d = 38.5315
Ball Position x = 352.6 y = 95.2 r = 90.984 d = 36.9296
Ball Position x = 368.6 y = 115.6 r = 82.9731 d = 40.495
Ball Position x = 361.8 y = 108 r = 96.1924 d = 34.93
Ball Position x = 335.6 y = 100.2 r = 84.3719 d = 39.8237
Ball Position x = 368.4 y = 101.8 r = 90.9803 d = 36.9311
Ball Position x = 352 y = 112.4 r = 85.4898 d = 39.3029
Ball Position x = 358 y = 103.8 r = 93.5211 d = 35.9277
Ball Position x = 370.8 y = 87.4 r = 83.8784 d = 40.058
Ball Position x = 344.4 y = 101.8 r = 97.8884 d = 34.3248
Ball Position x = 349 y = 78.6 r = 82.9481 d = 40.5073
Ball Position x = 379.2 y = 99.6 r = 92.8963 d = 36.1694
Ball Position x = 366.2 y = 84.4 r = 83.2056 d = 40.3819
```

Figura 3.31: Parámetros de los círculos que se detectan.

El código que se utiliza para realizar lo anteriormente descrito se muestra en el Listado 3.9.

Listado 3.9: Obtención de los parámetros que definen a los círculos

```
//Calculate the moments of the thresholded image
Moments oMoments = moments(imgThresholded);

double dM01 = oMoments.m01;
double dM10 = oMoments.m10;
double dArea = oMoments.m00;

if (dArea > 100000){
        //calculate the position of the ball
        int posX = dM10 / dArea;
        int posY = dM01 / dArea;

        float distance = 10884.94434*3/ sqrt(dArea/3.1411592654);
        medida_fil = distance*alfa + medida_fil*(1-alfa);
        cout <<"Ball Pos =" <<  posX << " distance = "<< distance <<" medida_fil = "<<
            medida_fil << "\n ";

        if (fd != -1 ) {
                char ball_position[12];
                sprintf(ball_position, "x%dr%.1f\r", posX, medida_fil);
     int count = write(fd, ball_position, sizeof(ball_position)-1);
    if (count < 0){
        cout <<"uart tx error" << "\n";
      }
   }
```

```
    iLastX = posX;
    iLastY = posY;

}else{

        int posX=1000;
        dArea=1000;
        cout <<"x = "<< posX << " Area ball = "<< dArea << " \n ";

        if (fd != -1 ){
                char ball_position[14];
                sprintf(ball_position, "x%dr%.1f\r", posX, dArea);
                int count = write(fd, ball_position, sizeof(ball_position)-1);
                if (count < 0){
                        cout <<"uart tx error "<< "\n");
                }
        }
}
```

Como podemos observar en la figura 3.31 aparece un datos más aparte de las coordenadas y el radio de cada círculo. Este dato es la distancia existente entre la cámara y el objeto, y se calcula a partir del radio detectado. A continuación se hace una breve descripción de cómo se ha calculado esta distancia.

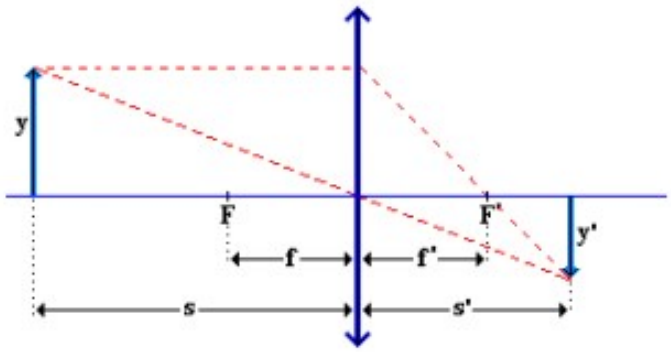


Figura 3.32: Diagrama de rayos de una lente.

Para conseguir las medidas de la distancia usamos la siguiente propiedad de los triángulos semejantes[8]:

$$\frac{y}{S} = \frac{y'}{S'}$$

En nuestro caso, y sería el radio de nuestra pelota, S sería la distancia entre la cámara y la pelota, S' sería la distancia hasta la imagen virtual, y y' sería el tamaño del radio virtual de la pelota.

En primer lugar, calibramos con unas medidas de la pelota y de la distancia conocidas. Con esta calibración obtenemos el valor de la distancia hasta la imagen virtual que, al no cambiar de lente, será siempre la misma. Con ésto presente, calculamos la distancia existente en el resto de casos a partir de la anterior propiedad.

[8] http://www.mathopenref.com/similartriangles.html

3.3.2.4 Errores estadísticos

Una vez que hemos obtenido los datos de la distancia a la que se encuentra nuestro objeto, debemos verificar si éstos son correctos o si existe algún error en las medidas que se toma. En la tabla 3.2 se relaciona el valor de la distancia real (medida con un telémetro) con la distancia virtual (medida a través de la cámara).

Tabla 3.2: Tabla que relaciona la distancia real con la virtual.

Distancia real (cm)	**Distancia virtual (cm)**
5.8	3.2126
11.1	9.1281
15.4	13.8128
22.3	20.9762
25.7	24.1213
30.5	29.6287
34.8	33.9428
40.5	40.0012
46.4	46.2348
49.8	50.1223
57.2	57.4239
62.3	62.9834
66.1	66.9283
70.6	71.4459
77.2	78.1248
77.2	78.2246
80.9	81.4895
85.2	85.8843
90.4	91.2386
95.6	96.3245
99.8	100.9724
105.2	106.4289
111.3	112.5432
115.1	116.2436
120.4	122.2133
124.6	125.8892
129.7	131.3895
136.5	139.1113
140.0	142.8912
144.5	146.7892
150.1	153.2247

La calibración para la medida de la distancia a la que se encuentra nuestro objeto, se realizó a una distancia de unos 50 centímetros. Como podemos observar en la tabla 3.2, los valores obtenidos en la medida a la que calibramos son muy similares, en cambio, si nos alejamos de esta medida, los valores van tomando una diferencia mayor y no siguen una recta lineal. Para evitar esta no-linealidad de los datos usamos el método de la no-linealidad de los puntos finales de nuestro rango de medidas. Con este método construiremos una función lineal que se asemeja a los valores reales y conseguimos reducir los errores que se puedan cometer en la medición

3.3.3 Comunicación UART

Para transmitir los datos provenientes de la *NanoPi Neo Air* a la controladora de vuelo se utiliza la comunicación *UART*[9]. Para esta comunicación se utiliza el pin etiquetado como UART1-Tx de la tarjeta *NanoPi Neo Air* que se configura con la función de transmisor y el pin 4 del puerto *Main* de la controladora de vuelo que se configura con la función de receptor. Ambos se deben configurar con la misma tasa de transmisión (*baudrate*)

En primer lugar configuramos la UART de la *NanoPi Neo Air* con la siguiente función:

Listado 3.10: Inicialización UART en *NanoPi Neo Air*

```
void uart_ini(int &fd){

        struct termios options;
   fd = open("/dev/ttyS1", O_RDWR | O_NOCTTY | O_NDELAY);
   if(fd == -1){
      perror("open_port: Unable to open:");
    }
    options.c_cflag = B9600 | CS8 | CLOCAL | CREAD;
    options.c_iflag = IGNPAR;
    options.c_oflag = 0;
    options.c_lflag = 0;
    tcsetattr(fd, TCSANOW, &options);
}
```

Como se puede observar en el código, la UART se ha configurado para una velocidad de 9600 baudios, 8 bits de datos, un bit de Stop y sin paridad.

Una vez configurada la UART transmitimos los datos necesarios hacia la controladora como se muestra en las siguientes líneas de código:

Listado 3.11: Transmisión datos UART

```
char ball_position[12];
sprintf(ball_position, "x%dr%.1f\r", posX, medida_fil);
int count = write(fd, ball_position, sizeof(ball_position)-1);
if (count < 0){
     printf("uart tx error\n");
}
```

Para verificar que se está realizando correctamente el envío de dichos datos, se utiliza el módulo *HC-05*. Se trata de un módulo *Bluetooth* con el que se puede establecer una conexión inalámbrica segura usando dicha tecnología.

[9] universal asynchronous receiver/transmitter. Más información: https://learn.sparkfun.com/tutorials/serial-communication

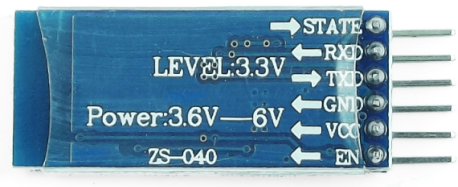


Figura 3.33: Módulo Bluetooth HC-05.

Tras configurar dicho módulo (ver: http://www.naylampmechatronics.com/blog/24_configuracion-del-modulo-bluetooth-hc-05-usa.html) con las mismas especificaciones que la *UART* de la controladora *NanoPI Neo Air*, podemos conectarlo inalámbicamente a nuestro celular y, mediante una aplicación (Blueterm), observar los datos provenientes de la controladora. Gracias a esto, podremos verificar que los datos que se estén enviando son los correctos.

Una vez que nos cercioramos de que obtenemos los datos correctamente, se configura la UART de la controladora de vuelo con las mismas características que la *NanoPi Neo Air* para así conseguir la comunicación entre ellas. En la siguiente función se inicializa la UART con los valores deseados.

Listado 3.12: Inicialización UART en *CC3D Mini Revolution*

```
static int32_t comUsbBridgeInitialize(void)
{
    usart_port = PIOS_COM_BRIDGE;
    vcp_port   = PIOS_COM_VCP;

    // Register the call back handler for USB control line changes to simply
    // pass these onto any handler registered on the USART

    if (usart_port) {
        PIOS_COM_RegisterCtrlLineCallback(usart_port,
                                          usb2ComBridgeSetCtrlLine,
                                          usart_port);
     }

     PIOS_COM_ChangeBaud(usart_port, 9600);


#ifdef MODULE_COMUSBBRIDGE_BUILTIN
    bridge_enabled = true;
#else
    if (usart_port) {
        bridge_enabled = true;
    } else {
        bridge_enabled = false;
    }
#endif

    if (bridge_enabled) {
        com2usb_buf = pios_malloc(BRIDGE_BUF_LEN);
        PIOS_Assert(com2usb_buf);
        usb2com_buf = pios_malloc(BRIDGE_BUF_LEN);
        PIOS_Assert(usb2com_buf);
    }
    return 0;
}
```

3.3.4 Procesamiento de los datos obtenidos

Una vez inicializada la UART y obtenidos los datos en la controladora pasamos al procesamiento de éstos. En la siguiente función se muestra el procesamiento realizado.

Listado 3.13: Función que se encarga del procesamiento de los datos obtenidos.

```
static void com2UsbBridgeTask(__attribute__((unused)) void *parameters)
{
    /* Handle usart -> vcp direction */

    int state = 0;

    while (1) {
      uint32_t rx_bytes;
      rx_bytes = PIOS_COM_ReceiveBuffer(usart_port, com2usb_buf, BRIDGE_BUF_LEN, 500);

      if (rx_bytes > 0) {

              // PITCH Y ROLL
              switch (state) {

              case 0:
                if(*com2usb_buf++==120) // cuando se detecta la x en el buffer
                {
                        *com2usb_buf=0;
                        com2usb_buf=(uint8_t*)buffer;
                        state=1;
                }
                else{
                        state=0;
                }
              break;

              case 1:

                if(*com2usb_buf++==114) // si se detecta y
                {
                        *com2usb_buf=0;
                        com2usb_buf=(uint8_t*)buffer;

                        if(atoi(buffer) == 1000){
                                pelota_x = 1000;
                        }else{
                                float error_x = atoi(buffer) - 325;
                                float pid_out = pid_apply(&ejex, error_x, dT);
                                pelota_x = (pid_out* 0.000230769f);
                        }
                        state=2;
               }
                break;

                case 2:

                  if(*com2usb_buf++==13)          // si se detecta \r
                  {
                          *com2usb_buf=0;
                          com2usb_buf=(uint8_t*)buffer;
```

```
                        if(atoi(buffer) == 1000){
                            pelota_y = 1000;

                            if (count >= 10){
                                pid_zero(&ejey);
                                pid_zero(&ejex);
                            }else{
                                count = count +1;
                            }

                        }else{
                                count = 0;
                            float error_y = 75 - atoi(buffer);
                            float pid_out = pid_apply(&ejey, error_y, dT);
                            pelota_y = (pid_out* 0.0005f);
                        }
                        state=0;
                    }
                break;
                }
            }
        }
}
```

En esta función se identifican cada uno de los parámetros que definen la posición de la pelota. La diferencia entre el valor del parámetro obtenido y un punto de consigna definido nos da el error que se debe de utilizar en el PID con el que corregiremos la posición de nuestra aeronave. El punto de consigna que usaremos es un valor intermedio del rango de medidas que exista para cada uno de los ejes. En el caso del eje X, obtendremos valores en el rango de 0 y 650, por ello, el punto de consigna que usaremos es el 325. Por otro lado, para el eje Y tendremos valores en el rango de 0 y 150 y por esa razón, usaremos el punto de consigna 75. La razón de usar estos valores intermedios es la de obtener el mismo peso de error para cada uno de los sentidos del movimiento.

Para corregir la posición de nuestra aeronave hacemos que la salida de los PIDs modifiquen las variables de control de vuelo con las que se llevan a cabo los movimiento necesarios del dron, y así, cumplir con las especificaciones citadas anteriormente.

En los siguientes diagramas se muestran las estructuras de control que se utilizan para que la aeronave consiga realizar los movimientos necesarios y así, seguir el objeto que esté detectando.

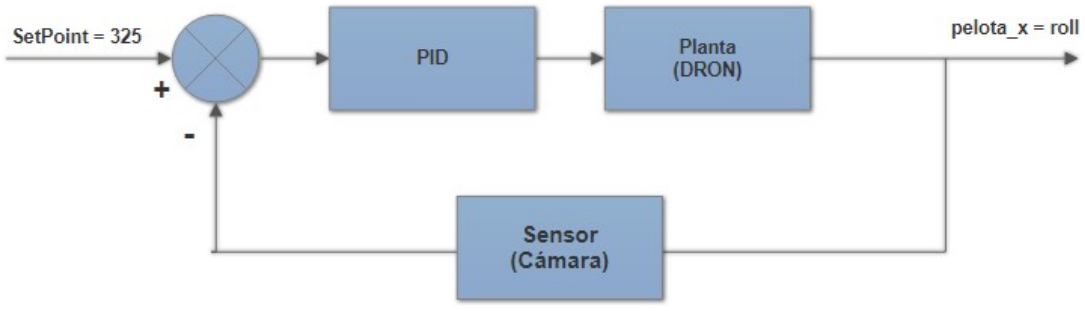


Figura 3.34: Diagrama de la estructura de control del desplazamiento en el eje X.

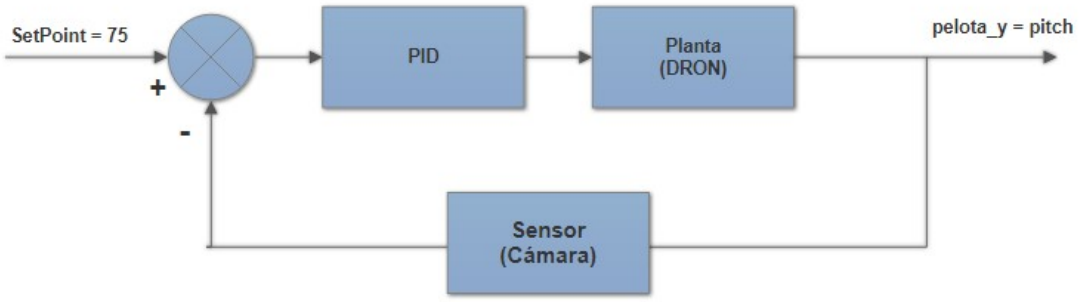


Figura 3.35: Diagrama de la estructura de control del desplazamiento en el eje Y.

Todo el código utilizado en la tarjeta *NanoPi Neo Air* se encuentra en la carpeta denominada *NanoPi Neo Air* de nuestro proyecto y el código usado en *CC3D Mini Revolution* en la carpeta *librepilot*.

Capítulo 4

Conclusiones y líneas futuras

En este último apartado se procede a realizar un breve resumen de las principales conclusiones a las que se ha llegado y comprobar si se han conseguido los objetivos marcados al inicio del presente documento. Después, se explicarán algunas de las líneas futuras que se pueden desarrollar como continuidad de este proyecto.

4.1 Conclusiones

En esta sección se analizan si se han conseguido los objetivos que se marcaron para este proyecto en la sección 1.1. Como ya se comentó en dicho apartado, en este proyecto se debían cumplir los siguientes objetivos: Estudio de las características básicas de los drones y de su funcionamiento, elección de los materiales necesarios, diseño y construcción de la aeronave, estudio del proyecto *LibrePilot* para posteriormente realizar las oportunas modificaciones, diseño de los algoritmos de control, configuración de la controladora de vuelo, integración y configuración del sensor de ultrasonidos en la controladora, estudio de la librería *OpenCV* para realizar el procesamiento de las imágenes que se capten con la cámara, integración de la librería *OpenCV* en la controladora *NanoPi Neo Air*, integración y configuración de la cámara en la controladora, procesamiento de las imágenes captadas y obtención de los parámetros que definen la posición del objeto a seguir en cada instante, habilitación de una comunicación para transmitir datos de un microprocesador a otro, y por último, realización de pruebas finales:

En apartados anteriores se ha llevado a cabo el estudio del proyecto *LibrePilot* del que se han obtenido los conocimientos necesarios para conseguir realizar este proyecto. Una vez conocido este proyecto se dio paso al estudio de la librería *OpenCV* que, gracias a algunos ejemplos de aplicación en otros proyectos, se pudo conocer mejor su funcionamiento y poder así, conseguir la segmentación del color necesaria para nuestra aplicación. Por lo tanto, los primeros objetivos de este proyecto se han logrado.

Tras conocer bien las características del proyecto *LibrePilot* y su arquitectura, se procede al estudio de los materiales necesarios para el dron y su consiguiente construcción. Una vez montado, se pasa a la implementación de *LibrePilot* en la controladora. Gracias al manual de usuario del proyecto se consigue configurar la aeronave para realizar los primeros vuelos. En éstos no se conseguía que el dron se estabilizase y se decidió cambiar los valores de los PIDs del bucle de estabilización de la posición para así, conseguir que la aeronave se estabilizase y garantizar que se moviese lo menos posible en los momentos en que no se diese ninguna orden. Una vez conseguida esta estabilización, se procede a mantener una cierta altura con la ayuda del barómetro. El dron vuela correctamente y mantiene la altura con pequeñas variaciones. Estas

variaciones se consiguen evitar incorporando al proyecto un sensor de ultrasonidos, con ello se consigue mantener la altura sin las variaciones que se producían anteriormente.

Una vez que se ha conseguido el objetivo de mantener la altura, se pasa a la programación del código necesario para obtener los parámetros de la pelota que se desea que siga nuestro dron. Para llevar a cabo esta acción usamos la controladora *NanoPI Neo Air*, que alimentamos mediante un regulador de tensión *LM2596* con salida 5V y 2 amperios de amperaje máximo. En un principio esta placa se alimentaba con un BEC de 0.5 amperios, pero nos daba problemas debido a que el amperaje que necesitaba dicha placa era mayor que lo que permitía el BEC.

Tras la obtención de estos parámetros se procede a abrir una comunicación UART entre los dos microcontroladores y transmitir los datos de uno al otro. Después, se procesan dichos datos y se modifican las variables de control de la aeronave a partir de esos datos. El procesamiento se basa en un control PID que se configura con valores pequeños para obtener pequeñas variaciones en los movimientos de la aeronave y así, conseguir el seguimiento de nuestro objeto.

Por último, y conseguidos los objetivos anteriores, se procede a realizar las pruebas finales. Debido a la limitación de tiempo de uso de las baterías se decide utilizar una fuente de alimentación que suministra hasta 12 amperios que, aunque no es suficiente para que los motores den su máxima potencia, nos sirve para llevar a cabo ciertas pruebas a una altura mínima. Conseguimos ajustar los PIDs para que el dron siga nuestro objeto en esta altura mínima a la que nos limitaba la fuente. Otro inconveniente que se presenta es que cada vez que cambiemos alguna característica de nuestra aeronave se deben volver a ajustar los PIDs para que ésta cumpla con los objetivos que se habían marcado anteriormente.

4.2 Líneas futuras

El proyecto *LibrePilot* tiene muchas posibilidades de modificación y por esta razón, se pueden proponer múltiples trabajos futuros. Algunos de ellos pueden ser:

- Inclusión de sensores de ultrasonidos adicionales para evitar obstáculos en el recorrido del dron.
- Inclusión de un sensor de infrarrojos para poder mantener la aeronave en una posición fija.
- Creación de un modo de vuelo automático con un dispositivo GPS para los vuelos en exterior.
- Implementar otro modo de vuelo en el que se pueda escoger el objeto que se desee reconocer y seguir.
- Construcción mediante impresión 3D de una estructura para recubrir el dron.
- Construcción mediante impresión 3D de unas protecciones para las hélices.
- Programación de una rutina de despegue.
- Programación de una rutina de aterrizaje.

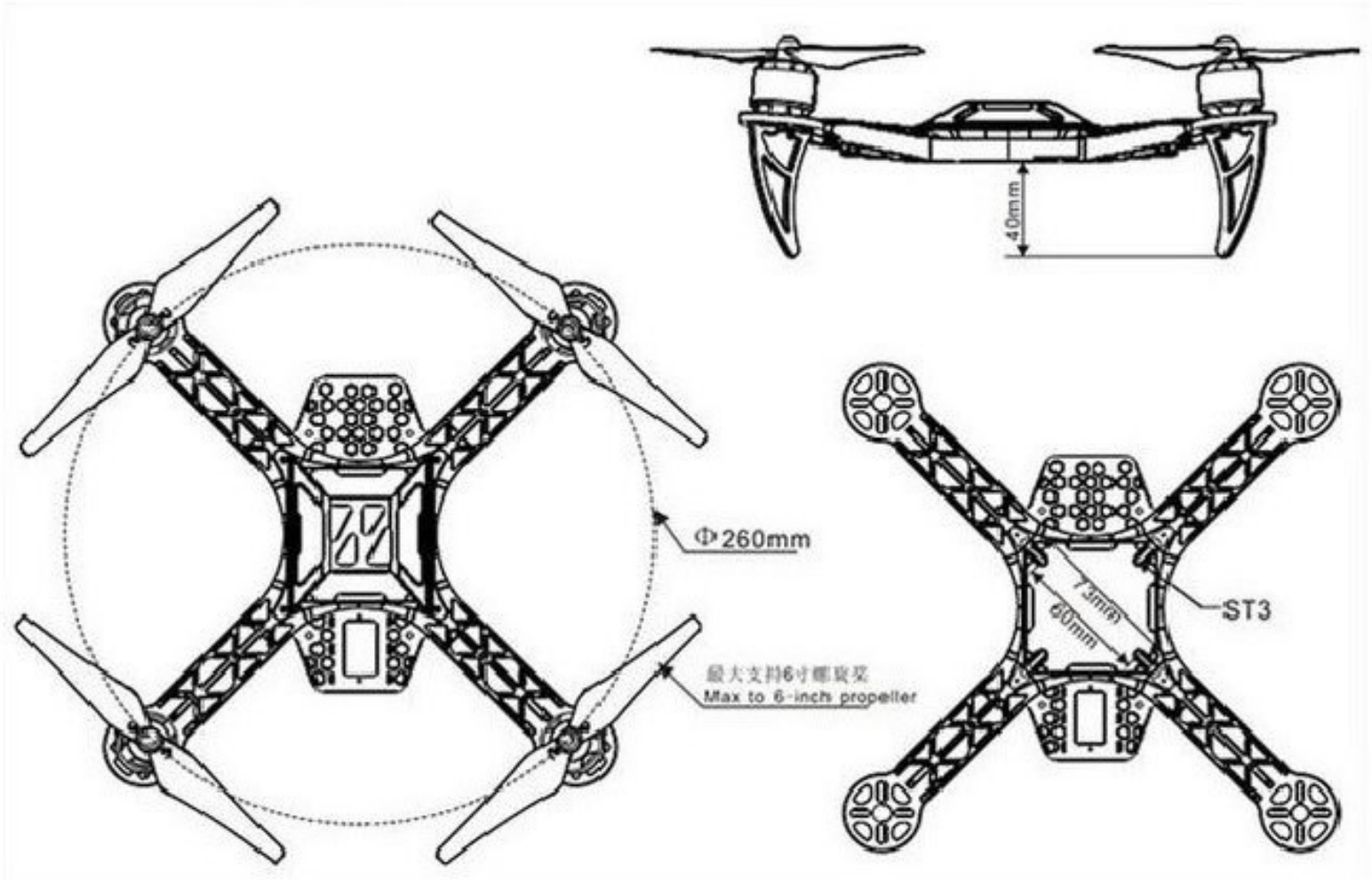
Planos y diagramas

Capítulo 5

Planos y diagramas

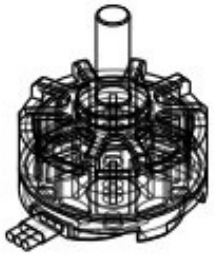
5.1 Índice de planos

1. Plano del frame (1).
2. Plano del motor Turnigy (2).
3. Plano CC3D Revolution (3).
4. Plano NanoPi Neo Air (4).
5. Esquema eléctrico CC3D Revoution (5).
6. Esquemas eléctricos NanoPi Neo Air (6-17).
7. Esquema eléctrico de conexionado del dron (18).
8. Diagrama de flujo general (19).

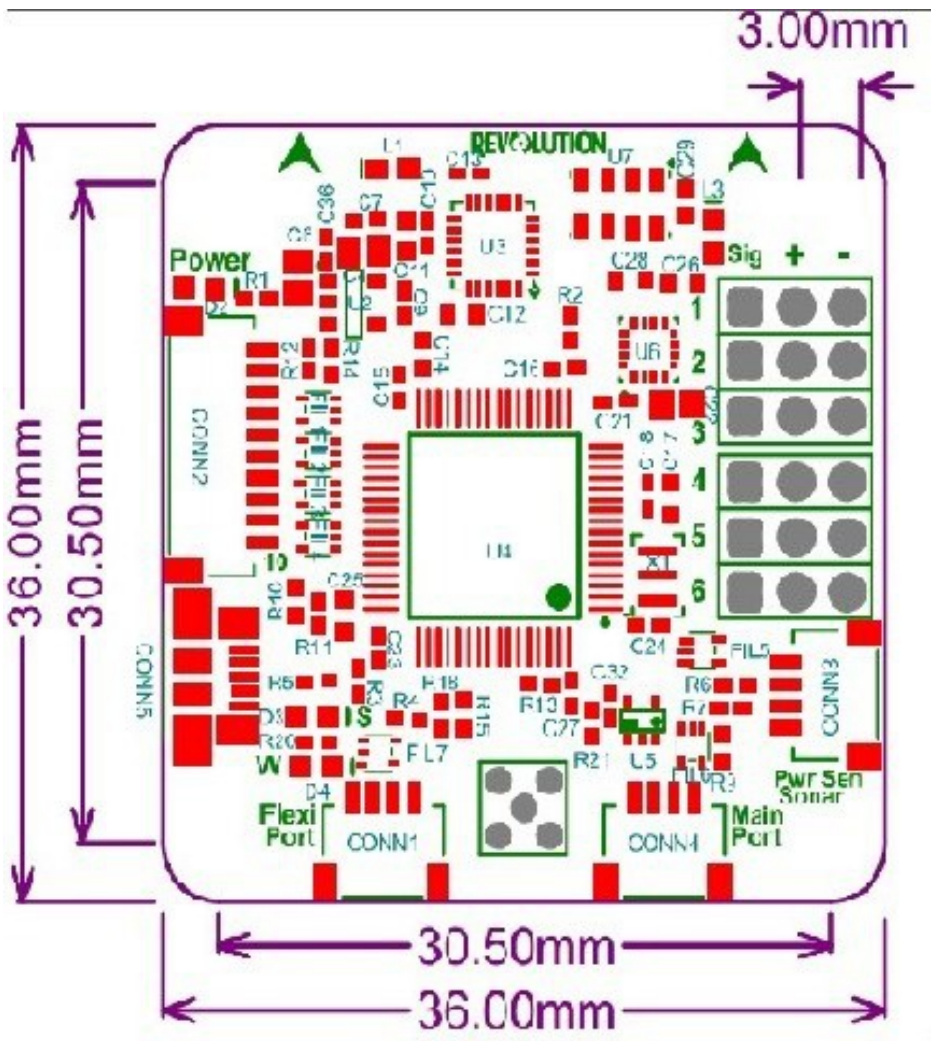


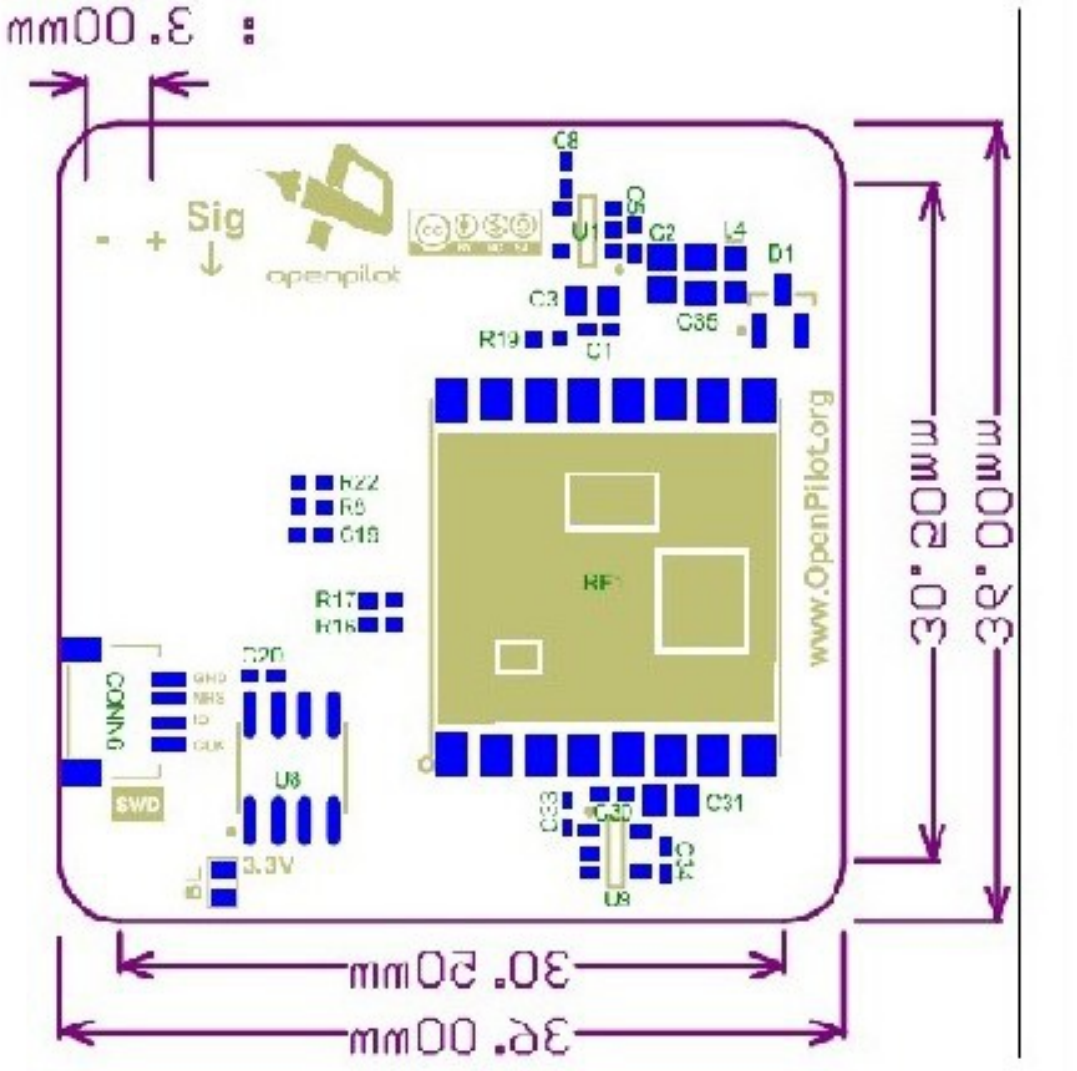
Autor: Francisco Ciudad Fernández	Archivo: 1
Fecha: 21/09/2017	Folio: 1/19

Frame FPV 250

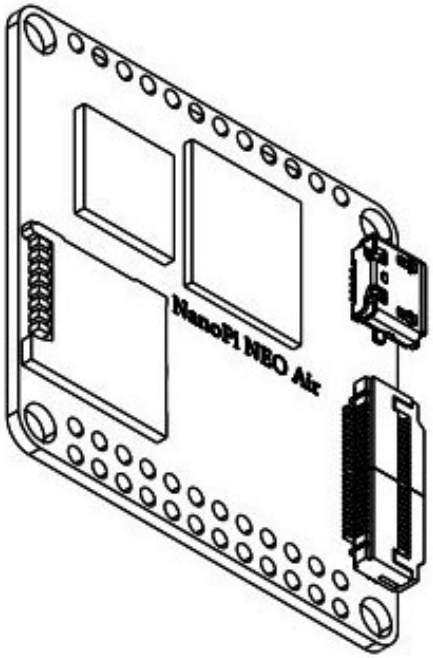


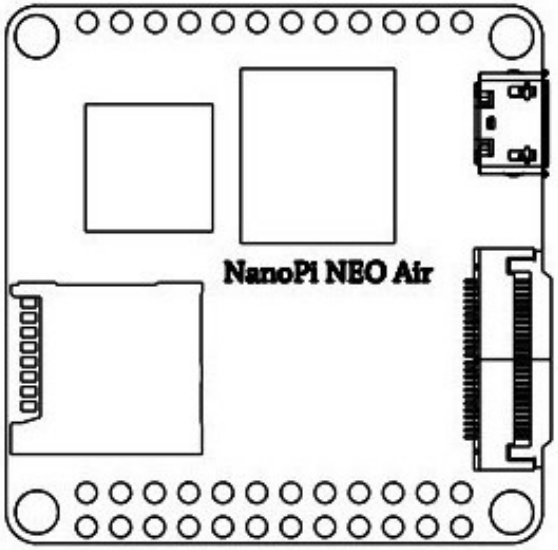
Autor: Francisco Ciudad Fernández	Motor Turnigy	Archivo: 1
Fecha: 21/09/2017		Folio: 2/19





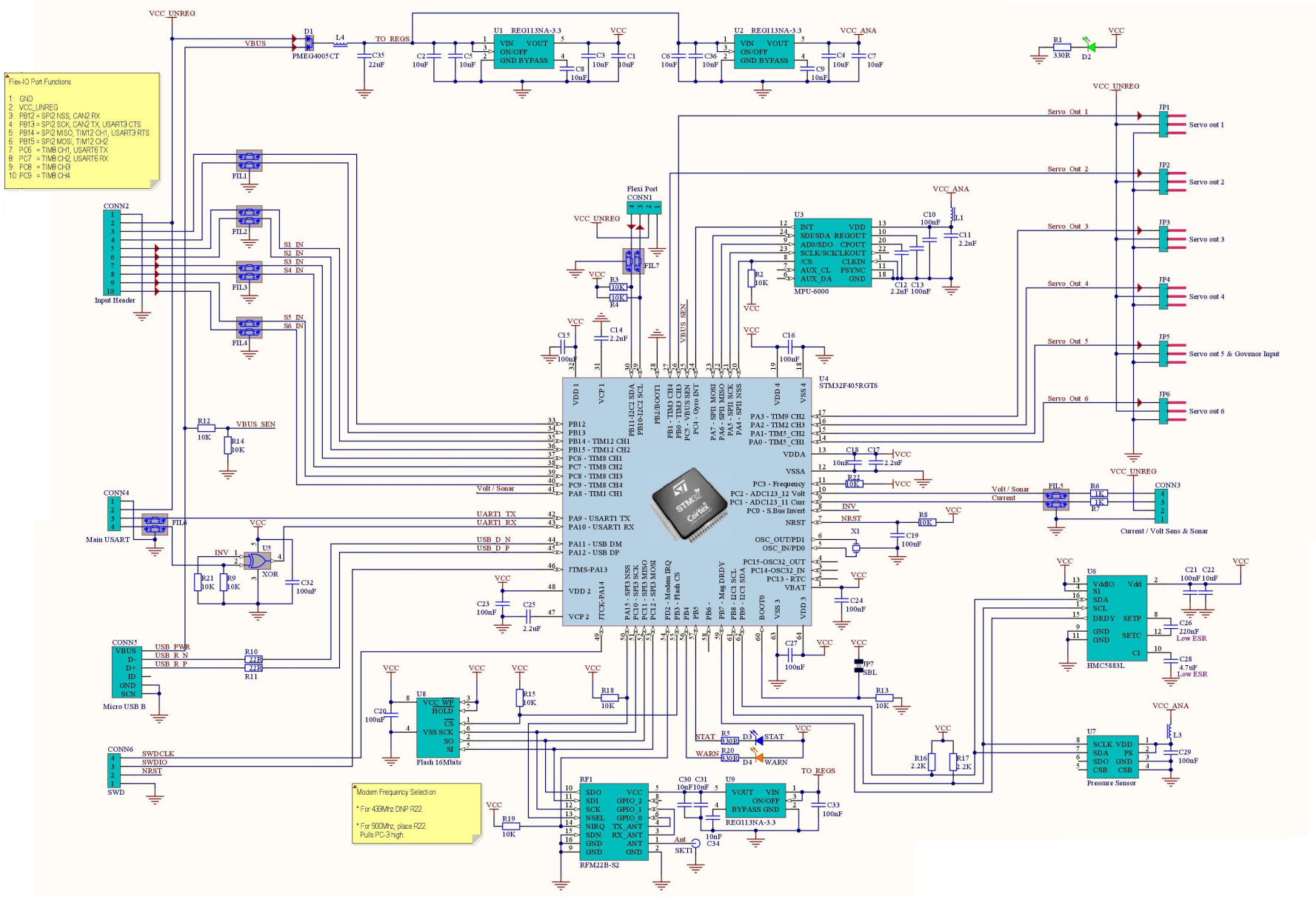
Autor: Francisco Ciudad Fernández	CC3D MINI REVOLUTION (Controladora de vuelo)	Archivo: 1
Fecha: 21/09/2017		Folio: 3/19





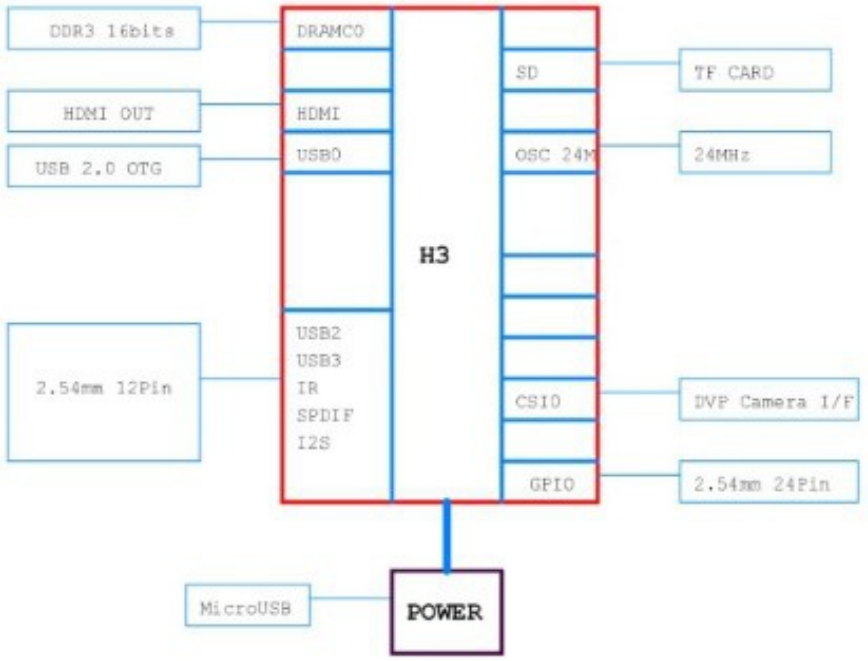
Autor: Francisco Ciudad Fernández	Archivo: 1
Fecha: 21/09/2017	Folio: 4/19

NanoPi NEO Air (Procesamiento de imágenes)



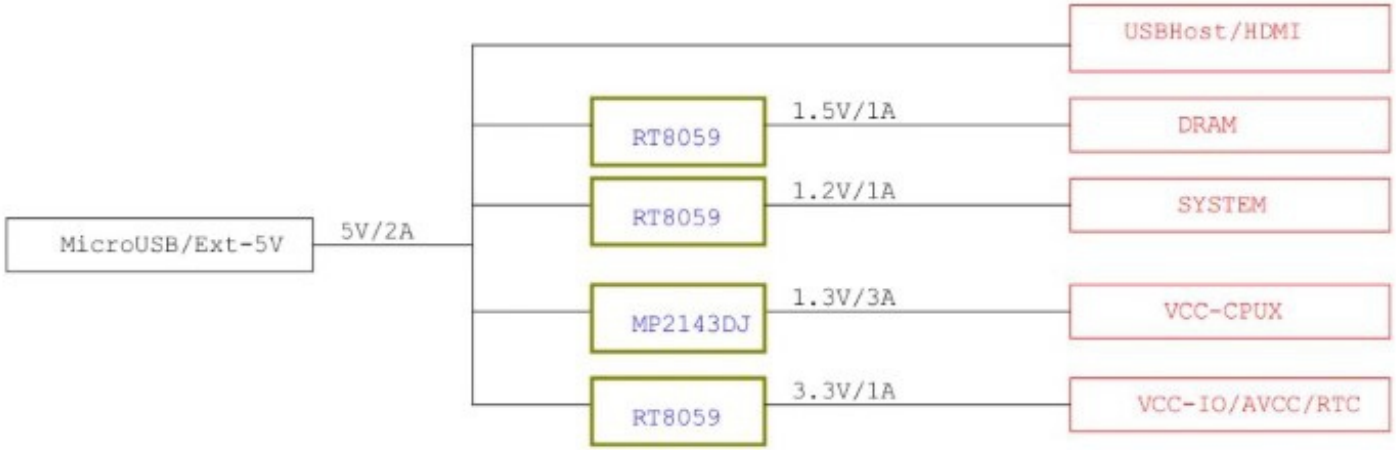
Autor: Francisco Ciudad Fernández	CC3D Mini Revolution (Esquema eléctrico)	Archivo: 1
Fecha: 21/09/2017		Folio: 5/19

BLOCK



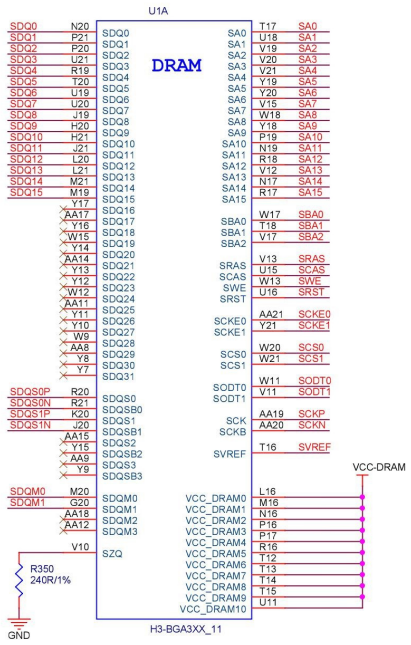
Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 6/19

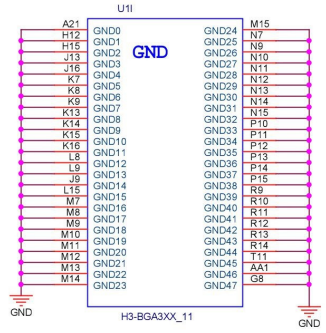
POWER TREE

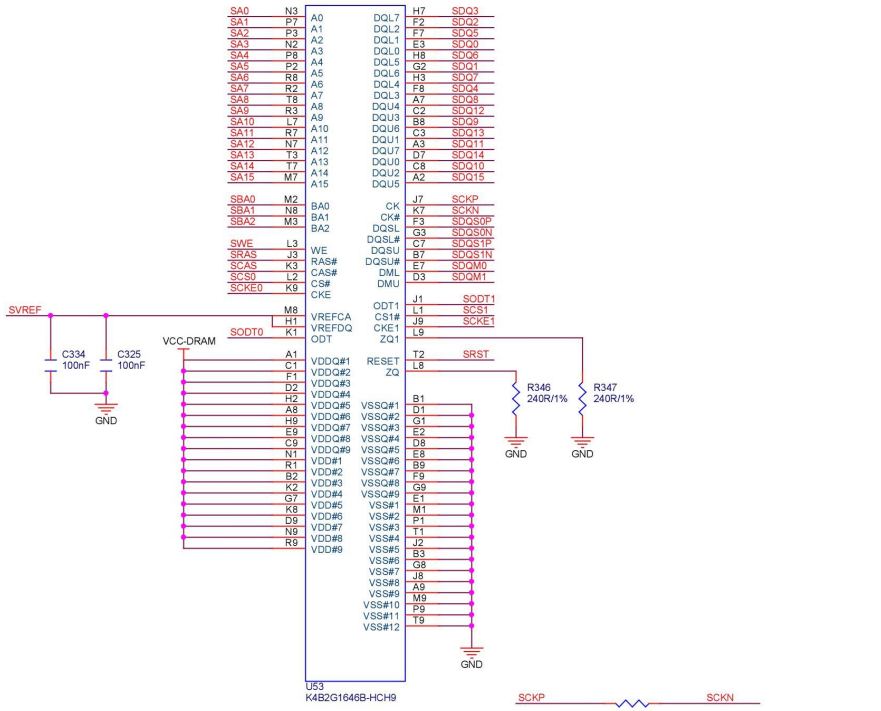


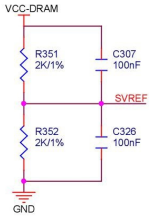
Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 7/19

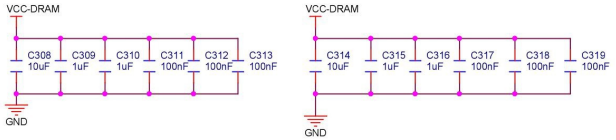
DDR3 16x1

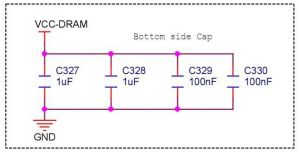




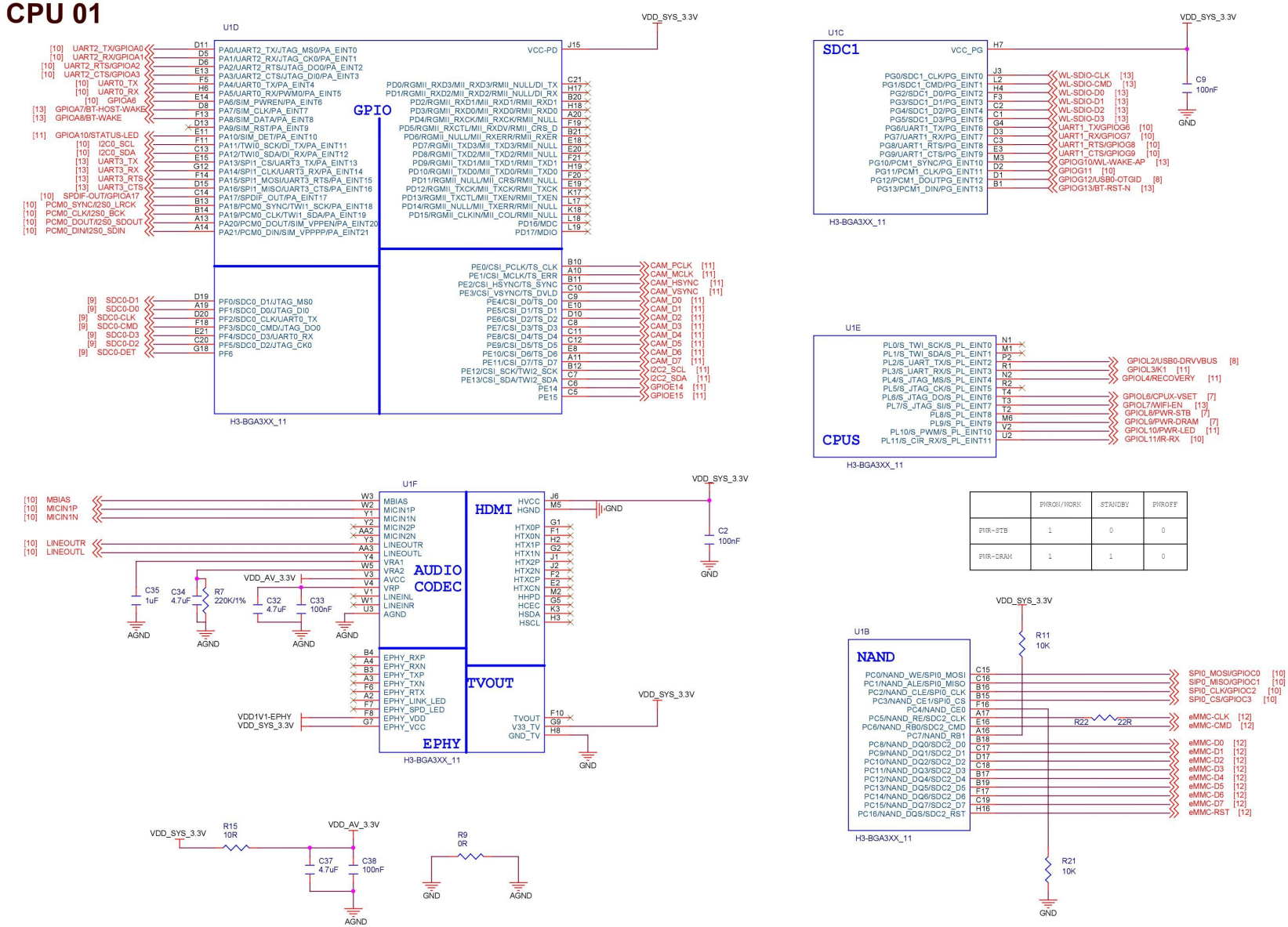






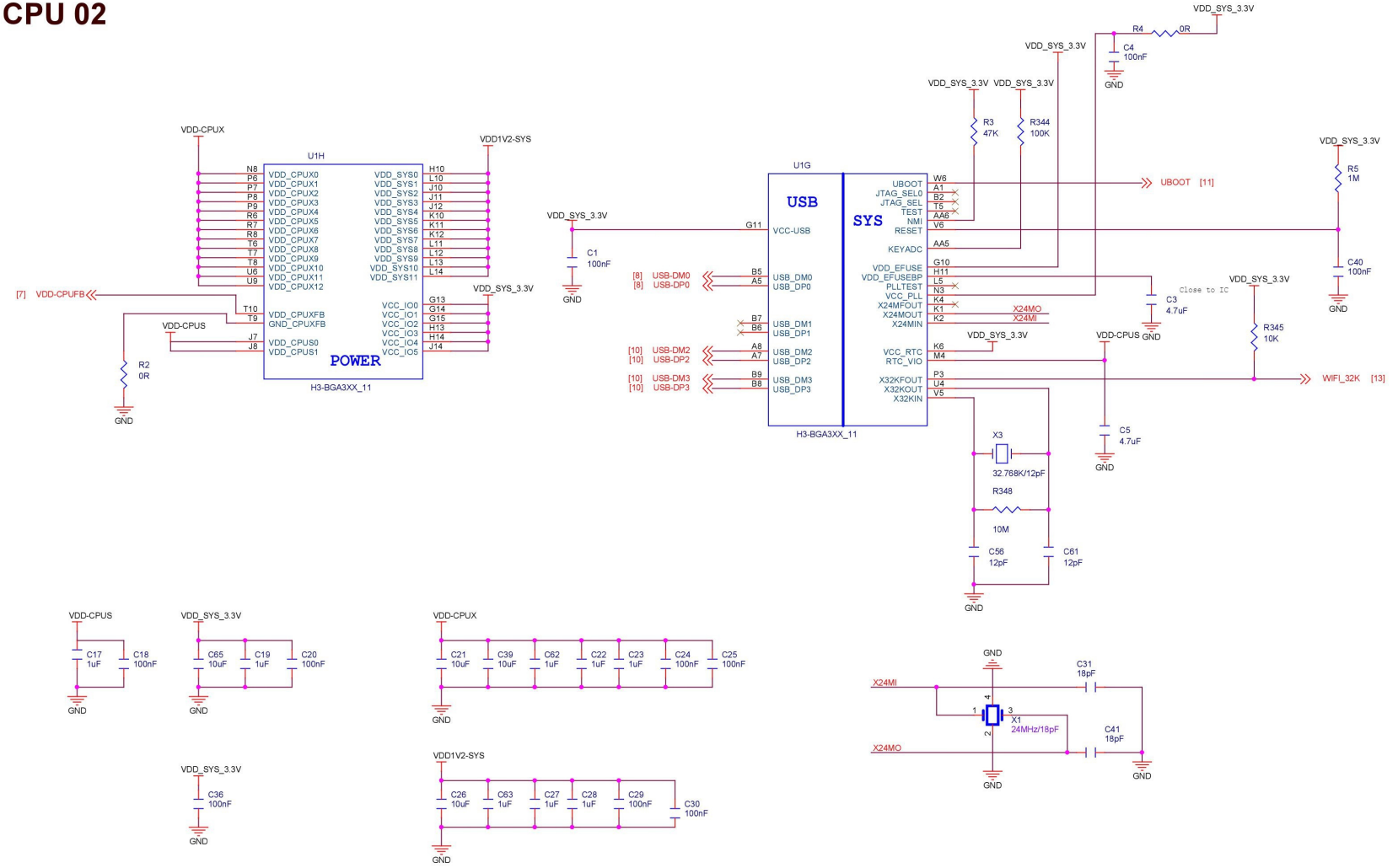


Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 8/19



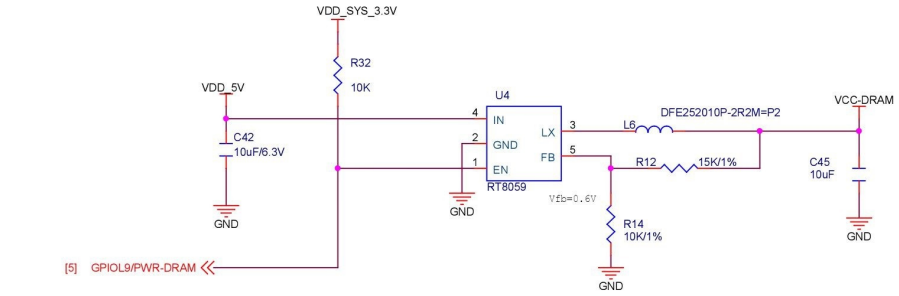
Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 9/19

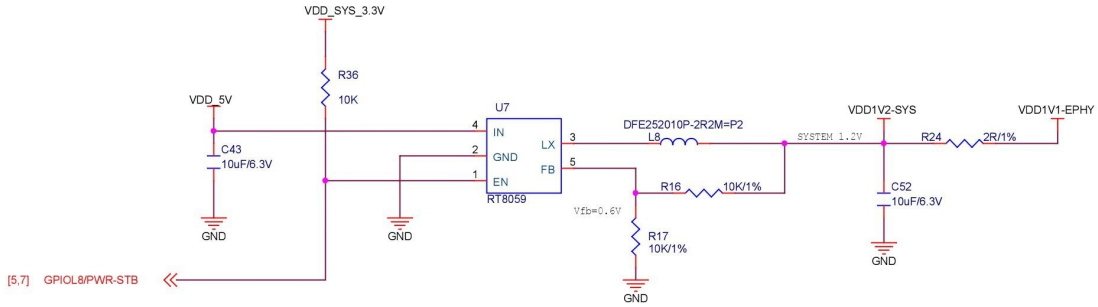
CPU 02

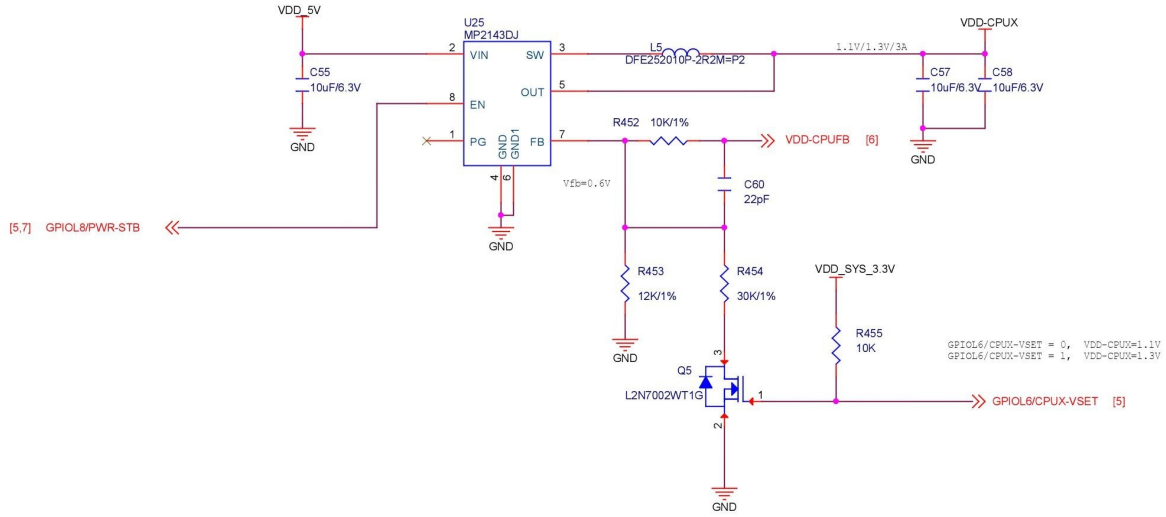


Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 10/19

POWER 01

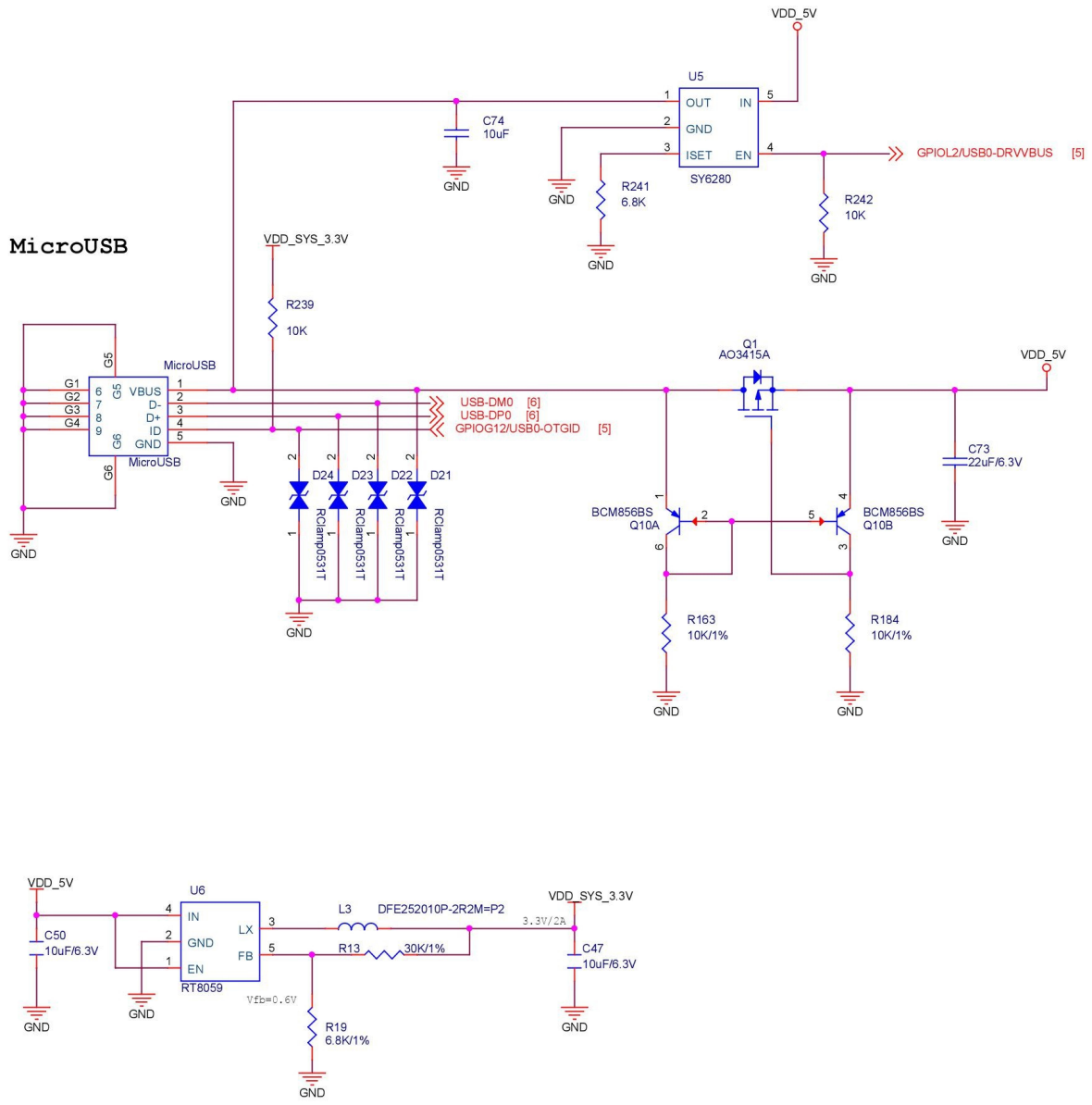






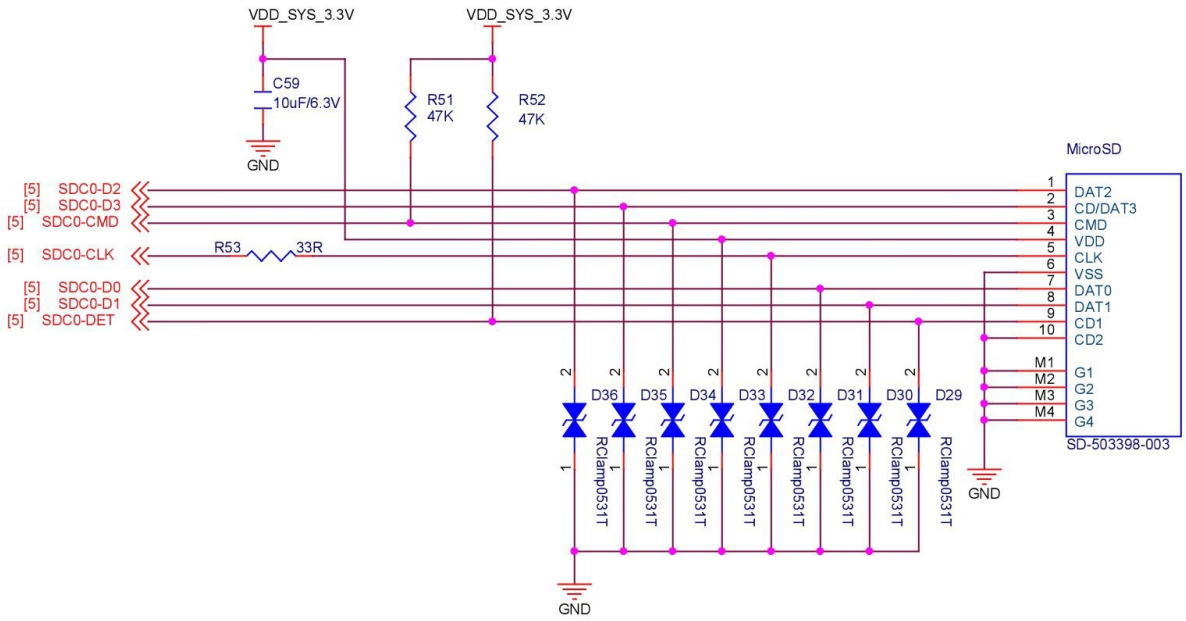
Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 11/19

POWER 02



Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air
Fecha: 18/05/2017	Archivo: 1
	Folio: 12/19

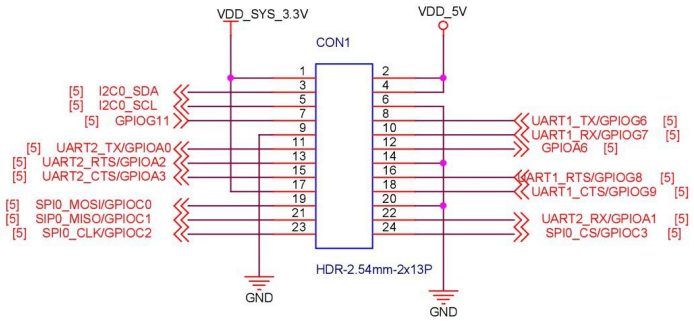
MicroSD

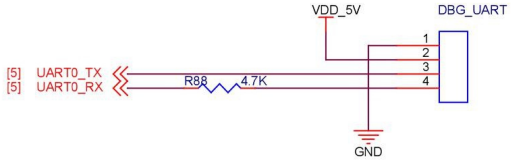


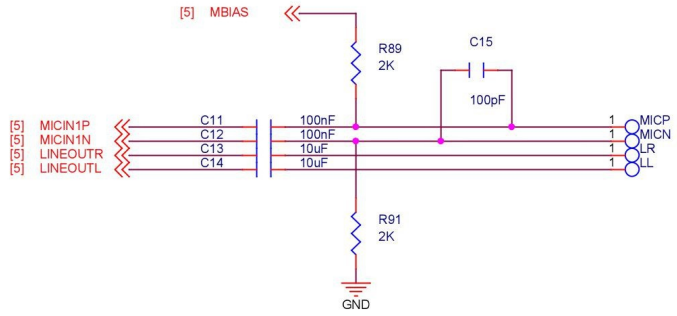
Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 13/19

GPIO

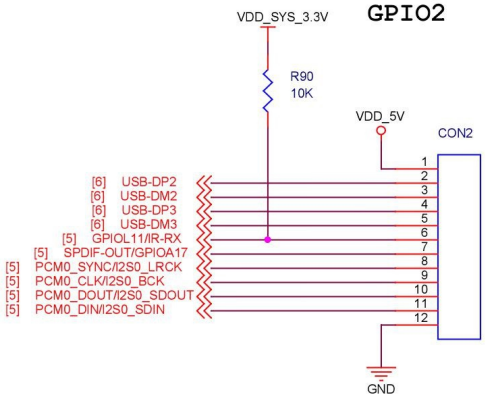
GPIO1





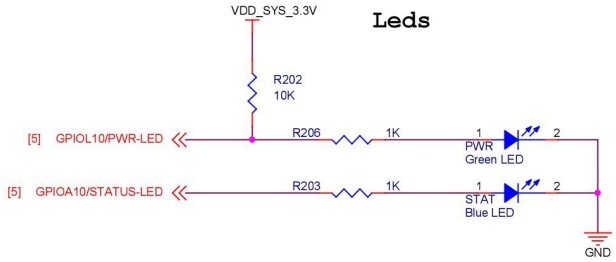


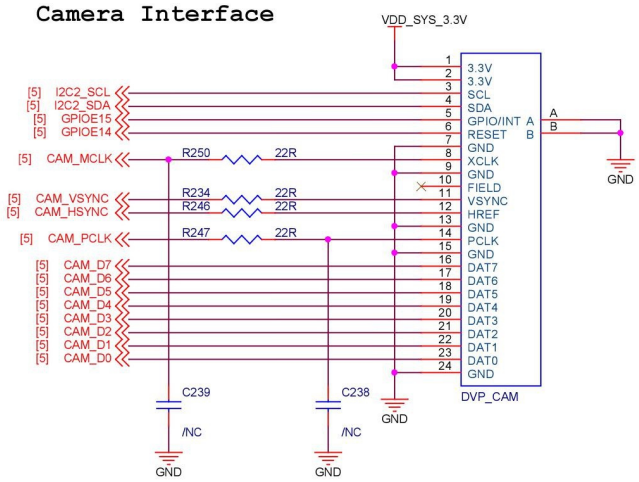
GPIO2

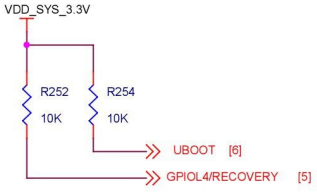


Autor: Francisco Ciudad Fernández	Esquema eleéctrico NanoPi NEO Air	Archivo: 1
Fecha: 18/05/2017		Folio: 14/19

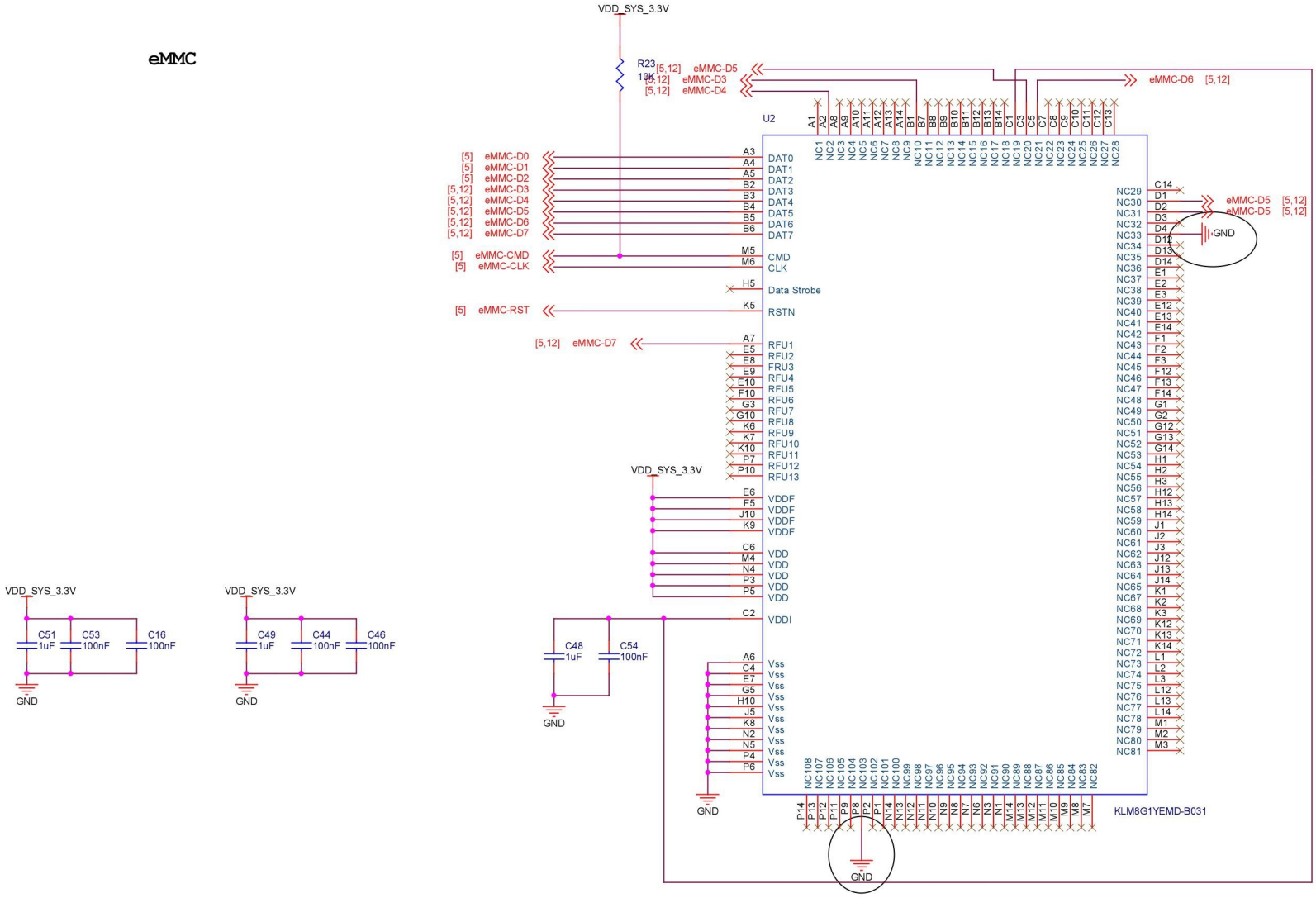
LEDs, LAN, Keys





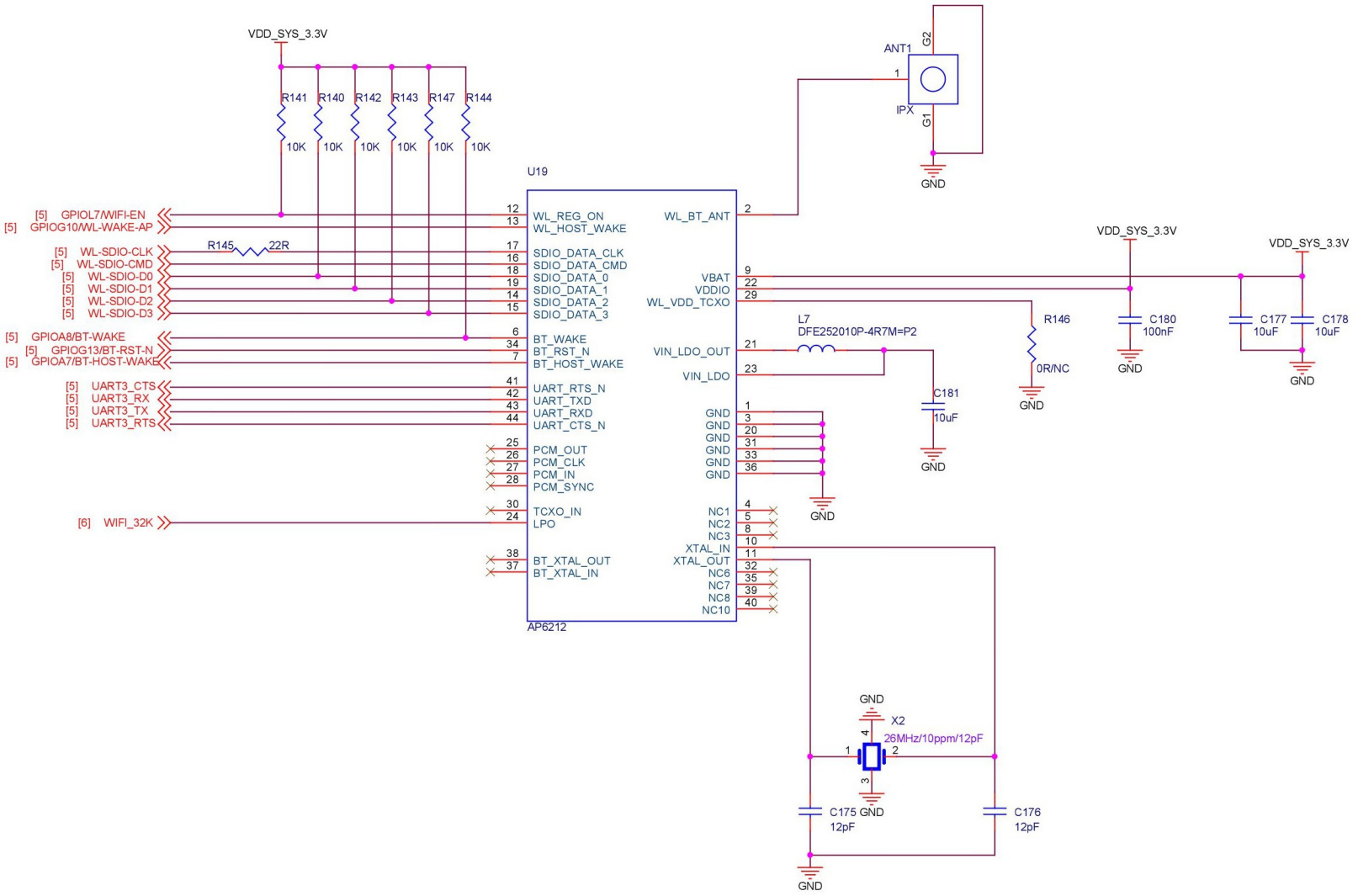


Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air	Archivo: 1
Fecha:		Folio: 15/19

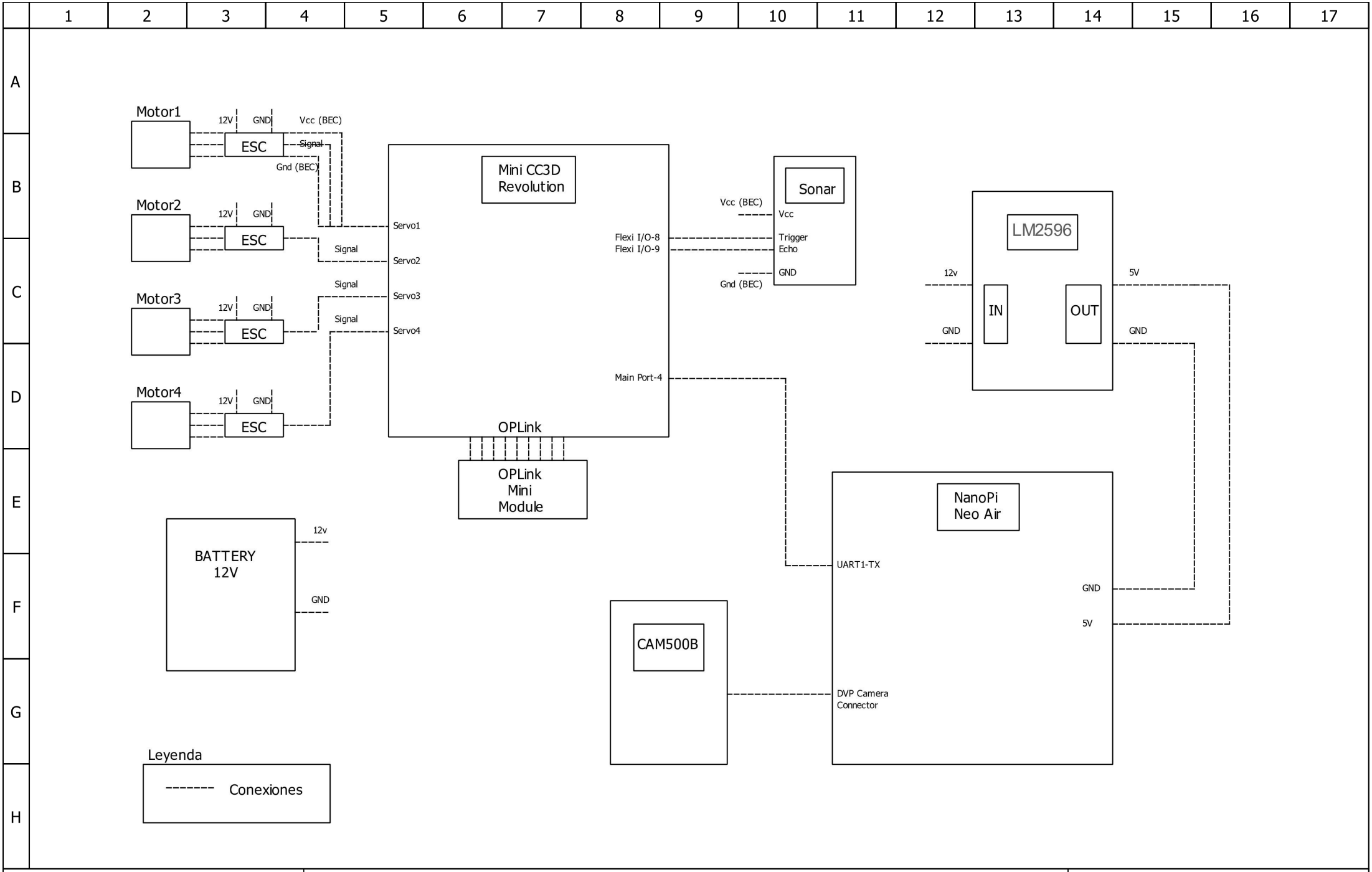


Autor: Francisco Ciudad Fernández	Archivo: 1
Fecha: 18/05/2017	Folio: 16/19

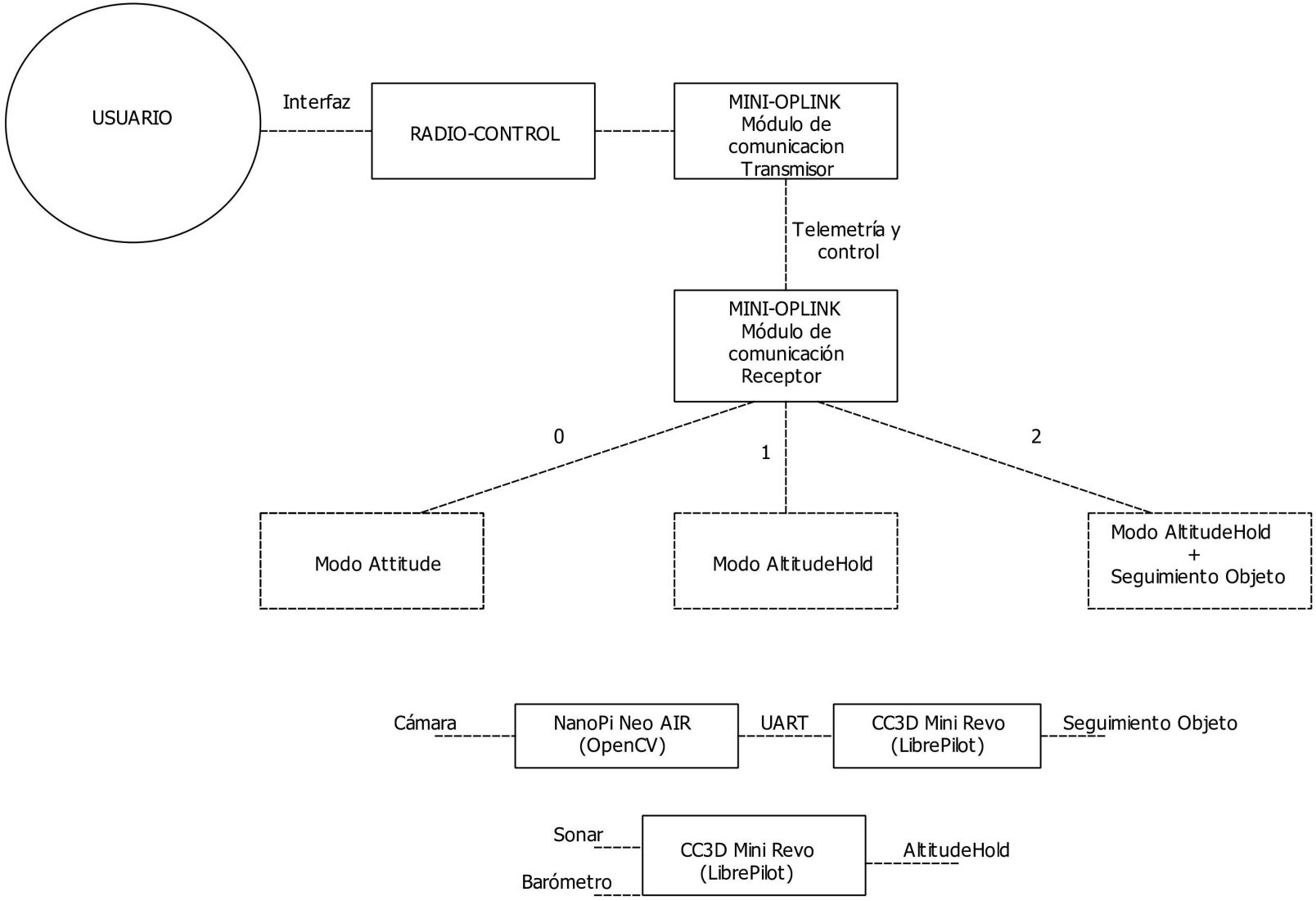
Esquema eléctrico NanoPi NEO Air



Autor: Francisco Ciudad Fernández	Esquema eléctrico NanoPi NEO Air
Fecha: 18/05/2017	Archivo: 1
	Folio: 17/19



Autor: Francisco Ciudad Fernández	Esquema eléctrico conexiones	Archivo: 1
Fecha: 23/01/2018		Folio: 18/19



Autor: Francisco Ciudad Fernández	Diagrama de flujo general	Archivo: 1
Fecha: 02/02/2018		Folio: 19/19

Pliego de condiciones

Capítulo 6

Pliego de condiciones

6.1 Requisitos Hardware

El hardware utilizado en este proyecto ha sido el siguiente:

- Para los vuelos autónomos es necesario un ordenador *AMD Quad-Core A10* con los siguientes requisitos mínimos:
  - Puerto USB 2.0 ó 3.0.
  - Memoria RAM 4 GB.
  - Disco duro de 500 GB.
- Pelota a la que deba seguir la aeronave.
- Módulo radio-control con frecuencia de transmisión de 433 MHz que se usa como interfaz entre el usuario y el dron.

6.2 Requisitos Software

El Software utilizado en este proyecto ha sido el siguiente:

- Sistema operativo *Ubuntu Xenial* versión 16.04 LTS
- Librería *OpenCV* verrsión 3.0.0.
- Proyecto *LibrePilot* versión 16.09 que se carga en la controladora de vuelo.
- Editor de texto inteligente para cualquier lenguaje de programación (*Eclipse* versión 3.8).

Presupuesto

Capítulo 7

Presupuesto

En este capítulo se incluye el presupuesto del proyecto [19], que consta de: presupuesto de ejecución material (materiales y mano de obra), presupuesto de ejecución por contrata y presupuesto total.

7.1 Presupuesto de ejecución material

El presupuesto de ejecución material es la suma del presupuesto de las distintas partidas que componen el proyecto, sin incluir: gastos generales, beneficio industrial, honorarios ni impuestos.

En estas partidas se incluyen: El coste por uso de equipos y software, coste de personal y coste de materiales.

El coste por uso de equipos y software se compone de:

- Ordenador Portátil Acer Aspire E15 con procesador AMD Quad Core A10-7300 with Turbo CORE Technology up to 3.20 GHz, con 16 GB RAM, de 64 bits y 500 GB de memoria ROM.
- Impresora HP Laserjet 400.
- Software de libre dsitribución (*LibrePilot*, OpenCV, GNU/LINUX).
- Sistema operativo *windows 8.1* y *Ubunto Xenial*.

Tabla 7.1: Presupuesto de Costes de Equipos y Software.

Equipos	**Precio Unitario(€)**	**Amortización**	**Uso**	**Total (€)**
Ordenador Acer	504.4	48 meses	7 meses	73.6
HP Laserjet 400	400.0	48 meses	1 mes	112.5
LibrePilot y OpenCV	0.0	48 meses	7 meses	0.0
SO: Windows 8.1	18.9	48 meses	48 meses	18.9
SO: Ubuntu Xenial	0.0	48 meses	4 meses	0.0
TOTAL:				205.0

En el coste de personal, se incluyen los sueldos base por hora trabajada de las personas que trabajarían en este proyecto. En este caso se ha trabajado 25 horas semanales a 4 semanas cada mes en 7 meses (700 horas en total).

Tabla 7.2: Coste de personal.

Concepto	Horas	Precio por hora (€)	Total
Ingeniero	700	25	17500

Tabla 7.3: Coste de materiales.

Concepto	Cantidad (Ud)	Precio unidad(€)	Total
Motor Brushless Multistar 1704	4	6.6	26.4
Hélices Tripala	4	2.2	8.6
ESC o variador	4	8.2	32.8
Chasis o Frame	1	9.9	9.9
Batería	1	13.0	13.0
OPLink mini	1	22.9	22.9
CC3D Mini Revolution	1	36.8	36.8
Sensor HC-SR04	1	5.4	5.4
NanoPi NEO Air	1	19.9	19.9
CAM500B	1	19.9	19.9
TOTAL:			196.1

En el coste de materiales hardware se incluye todo el material que se ha utilizado para la construcción del dron. En la tabla 7.3 se muestra la relación componente precio.

En la tabla 7.4 se muestra el coste total de los equipos informáticos, personal y materiales:

Tabla 7.4: Presupuesto de ejecución de material.

Concepto de costes	Precio (€)
Equipos informáticos	205.0
Personal	17500.0
Materiales	196.1
TOTAL:	17901.1

7.2 Presupuesto de ejecución por contrata.

El presupuesto de ejecución por contrata se refiere a la suma del presupuesto de ejecución material, gastos generales, beneficio industrial e IVA.

Tabla 7.5: Presupuesto de ejecución por contrata.

Concepto	**Valor (%)**	**Precio (€)**
Presupuesto de ejecución de material	100	17901.1
Gastos generales y beneficio industrial	19	3401.2
IVA	21	3759.2
TOTAL:		25061.5

7.3 Presupuesto total

El presupuesto total es la suma del presupuesto de contrata y los honorarios de redacción y de dirección.

Tabla 7.6: Presupuesto total.

Concepto	**Precio (€)**
Presupuesto por ejecución de contrata	25061.5
Honorarios de redacción	1253.1
Honorarios de dirección	1253.1
Total honorarios con IVA	3032.5
TOTAL:	28094.2

Manual de usuario

Capítulo 8

Manual de usuario

En este capítulo se explican los pasos que cualquier usuario debe de repetir para conseguir los objetivos que se detallan en este proyecto.

8.1 Descarga del código de *LibrePilot* versión 16.09

En primer lugar debemos obtener el código del proyecto *LibrePilot* para poder trabajar con él. Para ello seguimos los siguientes pasos:

1. Instalamos todas las herramientas y dependencias para poder compilar y depurar el código.
2. Obtenemos el código del repositorio usando *Git*.
3. Instalamos las librerías y herramientas de desarrollo, que se descargarán e instalarán usando *Makefile*.
4. Por último, depuramos el código y corremos el software localmente.

El proyecto se encuentra disponible para diferentes sistemas operativos. En nuestro caso, usamos la versión para *Ubuntu Xenial* debido a que la versión de *Windows* daba problemas a la hora de compilar las modificaciones que se realizaban. Los pasos que hemos especificado anteriormente para obtener el código se detallan en la siguiente página:

Tabla 8.1: Página en la que se detalla la obtención y compilación del código que se usa.

https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/57671696/Linux+-+Building+and+Packaging

8.2 Flujo de trabajo con *Git*

Git es un sistema de control de versiones distribuido pensado para proyectos donde existe un gran número de archivos de código fuente. Los sistemas de Control de Versiones Distribuidos se basan en que cada miembro del grupo tiene una copia local del control de versiones, es decir, una copia del historial de

cambios que se han ido realizando. Cada vez que realizamos una modificación en nuestra copia ésta se reporta a la copia principal, donde es gestionada convenientemente por el administrador del proyecto.

Mientras que en otros sistemas de control de versiones lo que se guarda en la base de datos son los cambios que se producen, *Git* guarda instantáneas. Es decir, cada vez que guardas el estado de tu proyecto, *Git* realiza una fotografía de éste, de cada uno de sus archivos. Esta instantánea, refleja el estado de ese archivo. En el caso de que un archivo determinado no haya sufrido ninguna modificación, no hace una copia de él, sino que guarda un enlace hacia el último archivo guardado. De esta manera *Git* trabaja con tu proyecto como una secuencia de instantáneas. Cada plano de la película está constituido por cada una de las instantáneas que se tomaron de cada uno de los archivos que constituyen el proyecto.

El flujo de trabajo con *Git* es el siguiente:

- Realizas un cambio en un archivo (estado modificado).
- Le indicas a *Git* que has modificado ese archivo (estado preparado).
- Confirmas todos los cambios realizados en los diferentes archivos (estado confirmado).

En el proyecto *LibrePilot* se hace uso de un repositorio donde se almacena el código de dicho proyecto. Todo aquel que quiera contribuir debe crearse una cuenta en *Bitbucket* y clonar una rama del proyecto usando el protocolo *http* o *ssh*:

Tabla 8.2: Clonación con el protocolo http

git clone https://YOUR_BB_USERNAME@bitbucket.org/YOUR_BB_USERNAME/librepilot.git

El siguiente paso es el de añadir el repositorio remoto (upstream) del proyecto *LibrePilot* de la siguiente forma:

Tabla 8.3: Adición del repositorio remoto

git remote add upstream https://bitbucket.org/librepilot/librepilot.git

Una vez que tenemos añadido el repositorio remoto podemos empezar a trabajar con nuestra versión local y en cualquier momento obtener la última actualización publicada en el repositorio remoto. Para ello hacemos lo siguiente:

Tabla 8.4: Obtención de la última versión publicada en el repositorio remoto

git fetch upstream

Una vez se hayan realizado las modificaciones oportunas se deben añadir al control de versiones para saber cuales son los ficheros que se han modificado. Para ello usamos "git add filesz "git commit".

Para finalizar se actualiza el repositorio remoto con todo aquello que se haya cambiado en nuestra máquina local. Para esto usamos:

Tabla 8.5: Actualización del repositorio remoto con los cambios realizados.

git push -u origin HEAD

8.3 Instalación de *LibrePilot* versión 16.09 modificada

Para comenzar se debe instalar *LibrePilot* que se puede encontrar en el siguiente archivo que se adjunta en el disco del proyecto:

Tabla 8.6: dirección carpeta *build* del *librepilot*

librepilot/build/archivo.deb

Esta versión del proyecto es la que se ha obtenido al compilar las diferentes modificaciones que se le han realizado a la versión 16.09. Con dichas modificaciones se ha conseguido cumplir con las especificaciones que se enuncian al inicio de este documento.

Una vez instalado *LibrePilot*, conectamos la controladora de vuelo al ordenador, actualizamos la versión y configuramos nuestro dron.

8.4 Configuración básica

En este apartado se explica cómo realizar la configuración básica de nuestra aeronave, es decir, lo básico para que pueda realizar un vuelo.

Esta configuración se puede realizar siguiendo los pasos descritos en la siguiente dirección:

Tabla 8.7: Configuración básica de un dron

https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/5669335/Multirotor+Wizard

En la anterior dirección se describe el proceso que se debe de realizar para configurar nuestro multirrotor. Además, se debe configurar la comunicación entre éste y un sistema de radio-control. Para realizarlo se pueden seguir los pasos descritos en la siguiente dirección:

Tabla 8.8: Configuración básica de un dron

https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/5669344/RC+Transmitter+Wizard

8.5 Modo *AltitudeHold*

Una vez realizada la configuración básica de nuestra aeronave, se pueden realizar vuelos estables con nuestro sistema de radio-control. Para cumplir con la especificación de mantener el dron a una cierta altura incluimos el modo de vuelo *AltitudeHold* como se explica en la siguiente dirección:

Tabla 8.9: Configuración *AltitudeHold*

https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/12058671/Altitude+Hold

Realizando esta configuración conseguiremos vuelos estables a la altura que se desee al entrar en el modo de vuelo *AltitudeHoLd*.

8.6 Comunicación UART

Para permitir la comunicación con la tarjeta *NanoPi Neo Air* se necesita configurar el puerto de la controladora que se utiliza para esta función. Para ello, nos vamos a la pestaña *Configuration* del GCS y en ésta a la configuración del Hardware. En esta pestaña aparece representada la controladora de vuelo que tengamos conectada en el puerto USB con sus respectivos puertos y la configuración posible de cada uno de ellos.

Para habilitar la comunicación UART, se selecciona la función *ComBridge* en el *MainPort* como podemos observar en la figura 8.1.

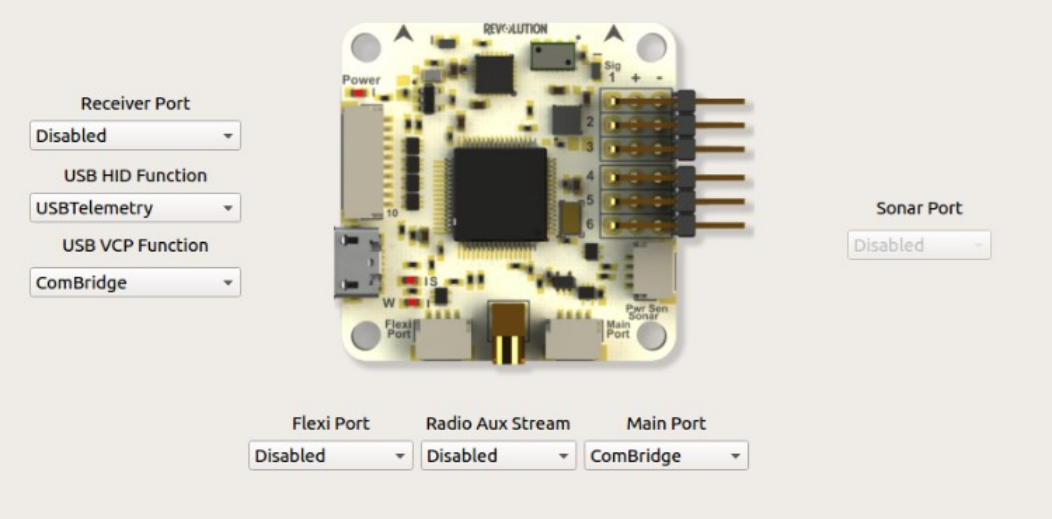


Figura 8.1: Configuración de los puertos.

8.7 Obtención de los parámetros del objeto a seguir

Lo último que falta para que el dron pueda seguir un objeto, es el reconocimiento del objeto y el enviar los datos correspondientes, a la controladora de vuelo mediante la comunicación que se ha citado anteriormente.

Para conseguir este objetivo, se debe cargar el proyecto del procesamiento de imágenes en la placa *NanoPi Neo Air*. En primer lugar, se debe configurar la placa y para ello, se siguen los pasos citados en el siguiente link:

Tabla 8.10: Configuración de la placa *NanoPi Neo Air*.

http://wiki.friendlyarm.com/wiki/index.php/NanoPi_NEO_Air

Tras tener configurada la conexión vía *wi-fi*, abrimos un terminal y cargamos los ficheros de esta parte del proyecto en nuestra tarjeta mediante SCP[1]. Los ficheros los podemos encontrar en la dirección */NanoPi Neo Air* del disco adjunto.

Una vez que tenemos el proyecto cargado en la placa, procedemos a lanzar la tarea con la que se capturan imágenes captadas con la cámara. Para realizar ésto, nos dirigimos a la dirección */NanoPi Neo Air/mjpg-streamer/mjpg-streamer* y arrancamos la tarea tecleando en el terminal:

Tabla 8.11: Tarea de captura de imágenes

./start.sh

Con esta tarea se capturan las imágenes provenientes de la cámara y se guardan en la memoria RAM de la placa para obtener escrituras y lecturas de la memoria más rápidas. Para que estas capturas no ocupen demasiado espacio en la memoria y saturen la placa, se decide guardar únicamente 5 capturas y sobreescribir el resto sobre éstas.

Una vez obtenidas las capturas se procede al procesamiento de las mismas. Para ello, abrimos otro terminal y nos dirigimos a la dirección */NanoPi Neo Air/bgsubtract/procesamiento* y arrancamos la tarea tecleando en el terminal:

Tabla 8.12: Tarea de procesamiento de las imágenes.

./main

Con esta tarea se consigue obtener los parámetros que definen la posición de la pelota en cada momento. Éstos llegan a la controladora de vuelo mediante la comunicación que se ha descrito anteriormente y una vez allí, se procesan para modificar las variables de control de manera proporcional.

En la siguiente referencia [20] podremos encontrar un enlace a un video en el que se demuestra el funcionamiento de los diferentes modos de vuelo que se han creado en nuestro proyecto.

[1] https://desarrolloweb.com/articulos/transferir-archivos-scp-ssh.html

Bibliografía

Bibliografía

[1] G. INTELLIGENIA. Aplicaciones y usos de drones. (Último acceso 10-Abril-2017). [Online]. Available: http://www.iuavs.com/pages/aplicaciones_y_usos

[2] J. Montero. (2016) Estos son los mejores drones profesionales de 2016. (Último acceso 20-Abril-2017). [Online]. Available: http://www.todrone.com/mejores-drones-profesionales/

[3] Scolton. (2010, Diciembre) Pcb quadrotor (brushless). (Último acceso 15-Septiembre-2017). [Online]. Available: http://www.instructables.com/id/PCB-Quadrotor-Brushless/

[4] DroneCode. Dronecode. (Último acceso 15-Septiembre-2017). [Online]. Available: https://www.dronecode.org/about/

[5] C. Escura. (2016, Enero) Principios básicos de vuelo. (Último acceso 20-Mayo-2017). [Online]. Available: https://vueloartificial.com/introduccion/toma-de-contacto/principios-basicos-de-vuelo/

[6] flybai. Controles de vuelo. (Último acceso 22-Mayo-2017). [Online]. Available: https://cursopilotodedrones.net/leccion/4-5-controles-de-vuelo/

[7] L. Laurent. (2017, Marzo) Librepilot system architecture. (Último acceso 2-Mayo-2017). [Online]. Available: https://www.librepilot.org/site/index.html

[8] G. Bradski, *Learning OpenCV: Computer Vision with the OpenCV library.* O'Reilly Media Inc, 2008.

[9] THEDRONEINFO. (2015, Junio) What parts do you need to build a drone? (Último acceso 5-Marzo-2017). [Online]. Available: http://www.thedroneinfo.com/2015/06/06/what-parts-do-you-need-to-build-a-drone/

[10] mobus. (2016, Noviembre) Motores brushless para drones, todo lo que necesitas saber. (Último acceso 19-Marzo-2017). [Online]. Available: https://mobus.es/blog/motores-brushless-para-drones-todo-lo-que-necesitas-saber/

[11] RCTECNIC. Motor brushless multistar 1704 - 2300kv v2 para multicópteros. (Último acceso 19-Marzo-2017). [Online]. Available: http://www.rctecnic.com/motores/3803-motor-brushless-multistar-1704-2300kv-v2-para-multicopteros

[12] FvpMax. (2017, Febrero) Hélices para drones: Tipos y tamaños. (Último acceso 22-Marzo-2017). [Online]. Available: http://fpvmax.com/2017/02/10/helices-drones-tipos-tamanos/

[13] FpvMax. (2017, Agosto) Variador electrónico (esc): Qué es y cómo funciona. (Último acceso 10-Abril-2017). [Online]. Available: http://fpvmax.com/2016/12/21/variador-electronico-esc-funciona/

[14] T. Hobby. (2013, Mayo) Baterías de lipo, características y consejos. [Online]. Available: http://blogturbohobby.blogspot.com.es/2013/05/bateriaslipo.html

[15] L. Community. Oplm vehicle control link. (Último acceso 23-Junio-2017). [Online]. Available: http://opwiki.readthedocs.io/en/latest/user_manual/oplink/control.html

[16] ——. Revolution board setup. (Último acceso 10-Mayo-2017). [Online]. Available: http://opwiki.readthedocs.io/en/latest/user_manual/revo/revo.html

[17] elecfreaks. Hc-sr04 user guide. (Último acceso 29-Abril-2017). [Online]. Available: http://www.elecfreaks.com/store/download/product/Sensor/HC-SR04/HC-SR04_Ultrasonic_Module_User_Guide.pdf

[18] FriendlyElec. Nanopi neo air. (Último acceso 5-Mayo-2017). [Online]. Available: http://www.friendlyarm.com/index.php?route=product/product&product_id=151

[19] CIFP. Documentos de un proyecto: Las mediciones y el presupuesto. (Último acceso 20-Septiembre-2017). [Online]. Available: https://es.slideshare.net/ACADENAS?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

[20] F. C. Fernández, "Vídeo demostración del tfg," http://www.youtube.com/demostracion_TFG.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá