Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Diseño e implementación de un quadrotor mediante el empleo de Raspberry Pi

Autor: Alejandro Jiménez Becerra
Tutor: Ignacio Alvarado Aldea

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Grado
Ingeniería de las Tecnologías Industriales

Diseño e implementación de un quadrotor mediante el empleo de Raspberry Pi

Autor:
Alejandro Jiménez Becerra

Tutor:
Ignacio Alvarado Aldea
Profesor titular

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Proyecto Fin de Grado: Diseño e implementación de un quadrotor mediante el empleo de Raspberry Pi

Autor: Alejandro Jiménez Becerra

Tutor: Ignacio Alvarado Aldea

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia, por estar siempre para apoyarme en mis caídas.

A mi padre en especial, por todos los problemas solucionados, ideas propuestas y horas dedicadas.

Agradecimientos

Quisiera agradecer a todas aquellas personas que me han acompañado en esta bonita andadura, familia, amigos, compañeros y profesores. A mis padres por apoyarme en todo momento y por mostrarme que el esfuerzo nunca es en vano. A mi tío por enseñarme lo bonito que es crear a través de las muchísimas charlas que hemos compartido. A mi hermano por su apoyo desinteresado, su amistad y sobre todo por su paciencia.

No podría olvidarme de aquellas personas que, aunque no se encuentran aquí, nunca dejaron de estar conmigo dándome fuerzas en todos mis proyectos, venidos y por venir. Abuelo, ¡lo conseguimos!

Por último, agradecer a Ignacio que aceptara la idea de un alumno desconocido y un tanto chiflado. Gracias por tu paciencia y por toda la ayuda que me has brindado. Espero que hayas disfrutado de este proyecto como yo lo he hecho pese a sus dificultades.

Resumen

Los vehículos aéreos están tomando una importancia cada vez mayor en nuestro día a día. Esta industria está experimentando una revolución que está permitiendo obtener avances hasta hace unos años inimaginables, avances tales como: seguimiento de personas, detección de incendios, transporte de mensajería, y un largo etcétera que tiene como factor común la automatización de la tarea liberando a las personas del riesgo intrínseco de la actividad. Con esta mira, el trabajo aquí expuesto consistirá en el diseño, fabricación e implementación de un vehículo aéreo no tripulado de tipo quadrotor.

Abstract

Aerial vehicles are becoming more important in our day everyday life. This industry is submerged in a revolution that is allowing us to obtain technological improvements which were unconceivable some years ago. People tracking, fire detection or packets delivery are just a few examples of this improvements. Although they seem to be absolutely different, they all have in common the fact that these tasks are being automated. According to this, the aim of this project is to design, fabricate and implement an unmanned aerial vehicle.

Índice

Agradecimientos .... **ix**
Resumen .... **xi**
Abstract .... **xii**
Índice .... **xiii**
1 Índice de Tablas .... **i**
2 Índice de Figuras .... **i**
3 Notación .... **i**
4 Introducción **1**
5 Diseño del quadrotor **3**
5.1 Análisis Previo *3*
5.1.1 Requisitos iniciales 3
5.1.2 Forma de fabricación 3
5.2 Diseño *3*
5.2.1 Piezas principales 4
5.2.2 Patas 5
5.2.3 Cajón central 5
5.2.4 Fijador 6
5.3 Cálculos de fuerzas y deformaciones *6*
5.3.1 Cálculos previos 6
5.3.2 Análisis de fuerzas y deformaciones 7
5.4 Montaje final *8*
6 Sistemas electrónicos **11**
6.1 Microcontrolador *11*
6.1.1 Características 11
6.1.2 Sistema Operativo 12
6.1.2.1 Clasificaciones de las tareas 12
6.1.2.2 Real-Time Operating System 13
6.1.3 Pines GPIO 13
6.1.4 Comunicaciones inalámbricas: WiFi y Bluetooth 14
6.2 Medidas inerciales *14*
6.2.1 Comunicaciones con la IMU 15
6.2.2 Configuración inicial 17
6.2.3 Lectura de parámetros 21
6.3 Control de altitud *22*
6.3.1 Sónar 23
6.3.2 Barómetro 23
6.4 Motores Brushless y Variadores ESC *26*
6.4.1 Motores Brushless 26
6.4.2 Variadores ESC 27
6.4.3 Conexionado 28
6.4.3.1 Conexión ESC 28

6.4.3.2 Conexionado motores brushless 29
6.4.4 Control por software 30
6.4.5 Determinación de parámetros de funcionamiento 30
6.4.6 Disposición de los motores 31
6.5 Batería LiPo 32
6.6 Cableado de potencia 33
6.7 PCB 35
6.7.1 Salidas a los variadores 35
6.7.2 Comunicaciones con los diferentes sensores 36
6.7.2.1 Sónar 36
6.7.2.2 Barómetro 37
6.7.2.3 IMU 38
6.7.2.4 GPS 38
6.7.3 Circuito de alimentación de la RPi 38
6.7.3.1 Etapa de regulación de tensión 38
6.7.3.2 Etapa de filtrado 40
6.7.4 Circuitos auxiliares 42
6.7.5 Placa resultante 44
6.7.6 Errores placa diseñada 47
7 Algoritmo de control, sintonización pid y comunicaciones 51
7.1 Algoritmo de control 51
7.1.1 Función de arranque 51
7.1.2 Tarea de lectura IMU y conversión a ángulos de Euler 52
7.1.3 Tarea de estimación de altitud 54
7.1.4 Tarea implementación controladores PID 55
7.1.5 Tarea de comunicación PWM con los motores 55
7.1.6 Tarea para la comunicación entre el drone y el usuario 55
7.1.7 Comunicación entre tareas 55
7.2 Comunicación entre el drone y el usuario 56
7.2.1 Formas de comunicación 56
7.2.2 Impementación del protocolo UDP 57
7.2.3 App de comunicación 57
7.3 Control de los grados de libertad 58
8 Solución alternativa 65
8.1 Interrupción externa del DMP 67
9 Anexo I 71
10 Anexo II 73
10.1 Instalación completa 73
10.1.1 Instalación de Raspbian 73
10.1.2 Instalación de Xenomai 74
10.1.3 Configuración perifericos 75
10.2 Instalación desde imagen 76
11 Anexo III 77
11.1 Hilos 77
11.2 Mútex 78
11.3 Variables de condición 78
11.4 Código de ejemplo 79
12 Anexo IV 81
12.1 BMP.h y BMP.c 81
12.2 Brushless.h y Brushless.c 85

12.3 Socket.c 86
12.4 Imu.h e Imu.c 86
12.5 Main.c 91
13 Anexo V 105
13.1 Implementación del DMP 105
14 Referencias 115
15 Glosario 118

1 ÍNDICE DE TABLAS

Tabla 6-1 Clasificación tareas 12
Tabla 6-2 Frecuencias protocolos IMU 16
Tabla 6-3 Especificaciones HC-SR04 23
Tabla 6-4 Especificaciones BMP-180 24
Tabla 6-5 Especificaciones BL2210/30 27
Tabla 6-6 Especificaciones ESC BUDGET 28
Tabla 6-7 Valores de funcionamiento de los motores 31
Tabla 6-8 Especificaciones batería ZIPPY Flightmax 32
Tabla 6-9 Especificaciones LM2576-ADJ 39
Tabla 6-10 Componentes etapa regulación de tensión 40
Tabla 6-11 Especificaciones diodos LED 43
Tabla 6-12 Especificaciones transistor NPN 43

2 ÍNDICE DE FIGURAS

Figura 5-1 Vista principal base quadrotor. 4
Figura 5-2 Vista posterior base quadrotor. 4
Figura 5-3 Pata del quadrotor.... 5
Figura 5-4 Cajón central porta-batería.... 6
Figura 5-5 Plano del soporte-fijador.... 6
Figura 5-6 Tensiones en el quadrotor (II). 7
Figura 5-7 Tensiones en el quadrotor (II). 7
Figura 5-8 Deformaciones en el quadrotor. 8
Figura 5-9 Detalle ensamblaje. 8
Figura 5-10 Conjunto montado. 9
Figura 6-1 Pines GPIO Raspberry Pi 3B 13
Figura 6-2 IMU MPU-9250 15
Figura 6-3 Trama paquete I2C 15
Figura 6-4 Comunicación SPI 16
Figura 6-5 Registro 27 - Gyroscope Configuration.... 17
Figura 6-6 Registro 28 - Accelerometer Configuration.... 18
Figura 6-7 Registro 29 - Accelerometer Configuration 2.... 18
Figura 6-8 Registro 26 - Configuración.... 18
Figura 6-9 Registro 106 – User Control 19
Figura 6-10 Registro 36 – I2C Master Control 20
Figura 6-11 Registro 37 – I2C_SLV0_ADDR 20
Figura 6-12 Registro 38 – I2C_SLV0_REG.... 20
Figura 6-13 Registro 39 – I2C_SLV0_CONTROL I 21
Figura 6-14 Registro 39 – I2C_SLV0_CONTROL II.... 21
Figura 6-15 Registro 0A – Control 1.... 21
Figura 6-16 Registros medidas magnetómetro 22
Figura 6-17 Registros medidas acelerómetro/giroscopio 22
Figura 6-18 Esquema funcionamiento sensor ultrasónico 23
Figura 6-19 Sensor barométrico BMP-180.... 24
Figura 6-20 Valores calibración barómetro 25
Figura 6-21 Ecuaciones cálculo presión atmosférica 25
Figura 6-22 Ecuación cálculo altitud.... 25
Figura 6-23 Esquema interno motor brushless.... 26
Figura 6-24 Motor brushless BL2210/30.... 27
Figura 6-25 Ancho pulso onda de control del variador 27
Figura 6-26 ESC BUGET 18A 28
Figura 6-27 Conexionado RPi - ESC 29
Figura 6-28 Conexión ESC - Motor 29
Figura 6-29 Disposiciones rotores en un drone.... 31
Figura 6-30 Batería ZIPPY Flightmax 32
Figura 6-31 Placa de distribución de potencia 33
Figura 6-32 Esquema conexionado 34
Figura 6-33 Conector motores - LiPo.... 35
Figura 6-34 Divisor de tensión salida sónar 37

Figura 6-35 Valores normalizados de resistencias ... 37
Figura 6-36 Esquema conexión LM2576-ADJ ... 39
Figura 6-37 Esquema filtro paso bajo pasivo ... 41
Figura 6-38 Atenuación del reuido según R ... 42
Figura 6-39 Ganancia en frecuencia para R = 1K y C = 100 µF ... 42
Figura 6-40 Esquema general circuitos auxiliares ... 43
Figura 6-41 Esquemático PCB ... 45
Figura 6-42 Top Layer PCB ... 45
Figura 6-43 Bottom Layer PCB ... 46
Figura 6-44 Primera versión RaspiPilot PCB ... 46
Figura 6-45 Esquemático PCB ... 47
Figura 6-46 Top Layer PCB ... 48
Figura 6-47 Bottom Layer PCB ... 48
Figura 7-1 Detalle efecto filtroo paso bajo en medidas IMU ... 53
Figura 7-2 Efecto filtro paso bajo en medidas IMU ... 54
Figura 7-3 Esquema algoritmo estimación altitud ... 55
Figura 7-4 Esquema conexión WiFi ... 56
Figura 7-5 Pantalla principal app ... 58
Figura 7-6 Pantalla de configuración app ... 58
Figura 7-7 Esquema general PID ... 59
Figura 7-8 Nuevo soporte batería ... 60
Figura 7-9 Gráficas control pitch ... 60
Figura 7-10 Gráficas control roll ... 61
Figura 7-11 Esquema comercial RPi ... 61
Figura 7-12 Controladora de vuelo Multiwii ... 63
Figura 7-13 Controladora de vuelo PXFmini ... 63
Figura 7-14 Placa Adafruit PCA968 ... 64
Figura 8-1 Placa Arduino Uno ... 65
Figura 8-2 Módulo Bluetooth HC-06 ... 66
Figura 8-3 Diagrama de flujo algoritmo Arduino ... 67
Figura 8-4 Control eje X ... 68
Figura 8-5 Control eje Y ... 68
Figura 8-6 Control ejes X e Y ... 69
Figura 9-1 Fuerzas sobre un quadrotor. ... 71
Figura 10-1 Pantalla usuario BalenaEtcher ... 73
Figura 10-2 Latencia Xenomai ... 75
Figura 10-3 Sistema operativo actual ... 75
Figura 11-1 Esquema básico mútex ... 78

3 NOTACIÓN

A	Amperio
Ah	Amperio-hora
A_V	Ganancia en tensión
F_{NETA}	Fuerza Neta
GB	Gigabyte
hPa	Hectopascal
I_{MAX}	Intensidad máxima admisible
I_N	Intensidad nominal
Kp	Kilopondio o kilogramo-fuerza
N	Newtons
mm	Milímetro
MPa	Megapascal
mAh	Miliamperio-hora
ms	Milisegundo
T_{SAMPLE}	Periodo de muestreo
uF	Microfaradio
V_{BE}	Tensión base-emisor
V_{CE}	Tensión colector-emisor
V_N	Tensión nominal
V_{REV}	Tensión inversa
Ω	Ohmio

4 INTRODUCCIÓN

Lo difícil se consigue, lo imposible se intenta.
- Napoleón Bonaparte -

Actualmente nos encontramos sumergidos en una vorágine de avances tecnológicos en la que nuevos productos se abren paso en el mercado día tras día. Lo que ayer era impensable, hoy es una realidad y mañana estará obsoleto. Dentro de este ímpetu innovador hay multitud de dispositivos que se han visto afectados: smartphones, vehículos eléctricos y ordenadores entre otros dispositivos.

Uno de estos avances son los denominados "drones". Estos han sufrido una gran evolución en un periodo de tiempo muy corto gracias al desarrollo tecnológico y a la inmensa cantidad de posibilidades que ofrecen. Gracias a este avance, vemos constantemente estos sistemas en nuestro día a día realizando operaciones muy diversas: detección de incendios, seguimiento de personas, detección de personas en tragedias, controles de velocidad… Cada vez son más las finalidades de estos y supondrán una revolución en diferentes aspectos de nuestra sociedad.

Por este motivo, el proyecto aquí expuesto consiste en el diseño, fabricación e implementación de todos los elementos necesarios para obtener un vehículo aéreo quadrotor totalmente operativo. Esta empresa supone un verdadero reto académico: abordar un problema muy candente en la investigación actual empleando los conocimientos de un estudiante.

Para poder lograr el objetivo deberemos abordar diferentes problemas. En primer lugar, se realizará el diseño de nuestro quadrotor para su posterior impresión en 3D. Deberemos realizar un diseño que asegure la integridad del sistema, además, deberá ser suficientemente grande para poder albergar todos los componentes que se necesitan, pero, a su vez, deberá ser ligero en pro de autonomía de este.

En una segunda etapa, abordaremos los diferentes componentes eléctricos/electrónicos. Aquí analizaremos las diferentes formas de comunicación posible entre los sensores, qué medidas toman, cómo comunicarnos con los actuadores y desarrollaremos una manera de comunicación entre el usuario y el quadrotor. Además, se realizará el diseño de una PCB en la cual se encontrarán todos los sistemas electrónicos del drone.

Finalmente, se abordará en una tercera etapa el diseño y prueba de algoritmos de control que aseguren la estabilidad del sistema. En este apartado se desarrollará la metodología para sintonizar los controladores que como ya adelanto, serán unos PID.

A modo de avance, cabe decir que la complejidad técnica que este tipo de sistemas tienen es muy elevada, hecho que ocasionará tener que desviar el camino del rumbo inicial.

5 DISEÑO DEL QUADROTOR

En este apartado se abordará el diseño físico del drone. Para ello emplearemos software de diseño asistido en 3D dada su comodidad y grandes posibilidades. En esta etapa se comprende desde el primer momento en el que tenemos un boceto mental de cómo ha de ser hasta que empezamos a fabricar las piezas.

5.1 Análisis Previo

5.1.1 Requisitos iniciales

Para comenzar a diseñar el quadrotor en primer lugar se debe tener una idea básica de qué tenemos que representar en ayuda del software. En nuestro caso estamos persiguiendo el diseño de un drone por lo que de partida ya conocemos algunos de los elementos que va a tener este sistema: patas, brazos donde se colocarán los motores y una zona central suficientemente amplia para poder albergar los diferentes componentes electrónicos del sistema, así como la batería.

La elección del tipo de drone se realizará en base a simplicidad: un quadrotor. Este consta de cuatro rotores: número suficiente para conseguir una estabilidad relativamente buena, pero sin complicar en exceso la etapa de control.

5.1.2 Forma de fabricación

Otro aspecto para tener en cuenta para el diseño del quadrotor es cómo va a ser fabricado. Este procedimiento impondrá los límites dimensionales de nuestro modelo.

Dado que buscamos que el conjunto sea ligero pero flexible, vamos a optar por la impresión en 3D. Esta técnica se encuentra en pleno auge y consiste en la formación de piezas en 3D mediante la impresión de un polímero. Esta técnica nos permite obtener una buena calidad en las piezas además de emplear un material ligero pero resistente. La única desventaja que notaremos en este proyecto es que el tamaño de las piezas no puede ser muy grande por limitaciones de la propia impresora 3D.

Para solventar este contra, optaremos por dividir nuestro diseño en piezas pequeñas que puedan ensamblarse posteriormente.

5.2 Diseño

Una vez tenemos claro qué tenemos que hacer y cuáles son las limitaciones que disponemos procedemos a el diseño del quadrotor. Este estará compuesto por los siguientes componentes:

- *4 piezas principales:* contiene el brazo sobre el que se apoya el motor y una fracción de la cavidad central (aquella en la que se apoyan los elementos electrónicos).
- *Patas:* sirven para que el quadrotor se apoye en el suelo.
- *Cojón central:* cajón que irá bajo la cavidad central del drone y que servirá para sostener a la batería y bajar el centro de gravedad.
- *Fijador:* componente metálico que une las piezas principales y que le da rigidez al conjunto.

En los siguientes subapartados procederemos a explicar componente a componente.

5.2.1 Piezas principales

Como antes se ha descrito, estas piezas constan de una fracción del cuerpo central del drone a la vez que un brazo sobre el que se colocará un motor.

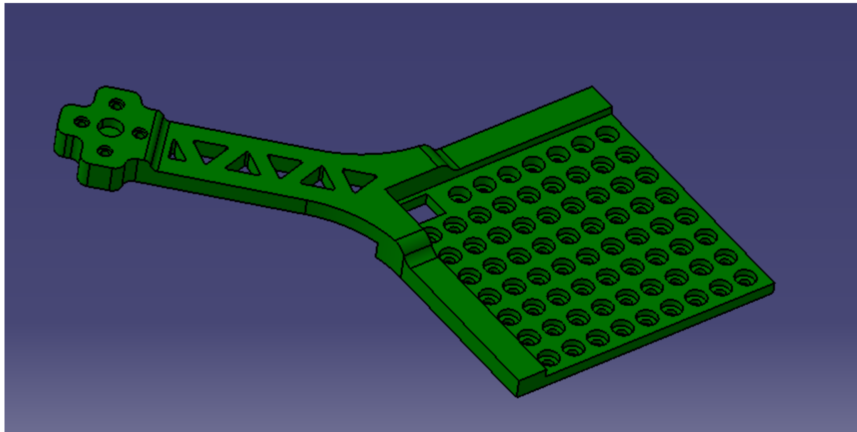


Figura 5-1 Vista principal base quadrotor.

Como se puede apreciar en la imagen, la zona central está totalmente agujereada y esto se debe a que permite aligerar peso a la vez que la podemos emplear para fijar los componentes a esta. Además, si se observa, se aprecia un rectángulo en la esquina de la base central. Este sirve para introducir las patas.

Por otra parte, el diseño de los brazos tiene un par de peculiaridades. La primera es que, para poder aligerar peso se ha optado por la triangulación del mismo (recordemos que el triángulo es la figura que cuesta más deformar). Además, el extremo de este tiene una forma en cruz para que apoye el motor en su totalidad y se reduzcan así posibles vibraciones, a la vez de una serie de agujeros pasantes para introducir los tornillos que fijarán el motor y un agujero central para que no choque el eje saliente del motor.

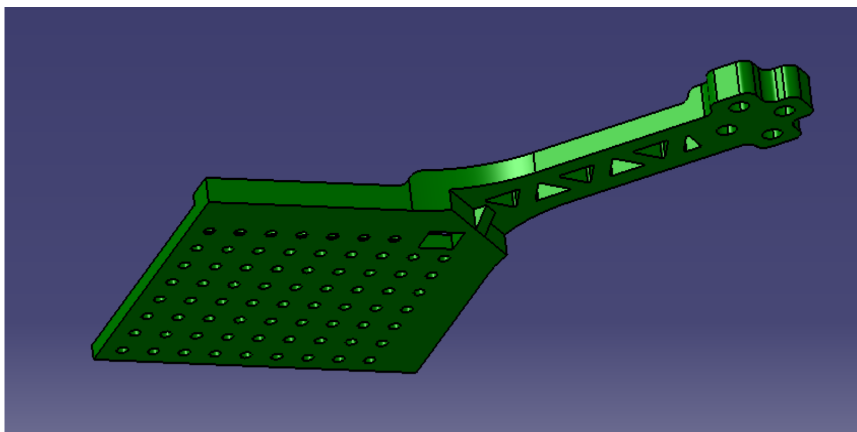


Figura 5-2 Vista posterior base quadrotor.

Finalmente, para aportar un mayor grado de rigidez al brazo, se ha reforzado el punto más débil del mismo: la unión entre el cuerpo central y el brazo.

5.2.2 Patas

Dada la finalidad de estos componentes y la previsión de que puedan sufrir impactos, deberán ser lo más pequeñas y lo más anchas posibles para evitar su ruptura. De nuevo, para reducir peso hemos triangulado la pieza.

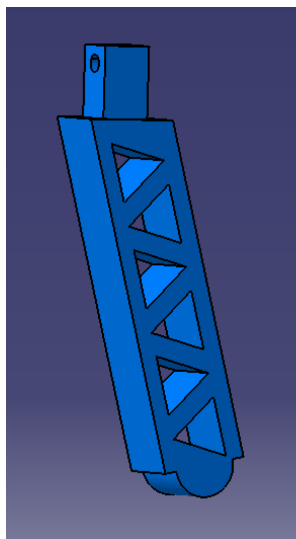


Figura 5-3 Pata del quadrotor

Como se aprecia en la imagen, junto al cuerpo central hay un pequeño cuerpo muy largo y delgado. Este sirve para fijar las patas a la base central. Para reforzar este elemento colocaremos espuma en aquellos puntos más críticos de manera que esta absorba parte de los impactos que pudiera recibir la pata.

5.2.3 Cajón central

Dado que la batería es muy voluminosa y pesada, esta no puede ir colocada en la base central por dos motivos: dado el elevado peso desplazaría el centro de gravedad del sistema y dificultaría su control y al ser tan voluminosa impediría que cupieran todos los elementos electrónicos que deben de ir en la base central. Por este motivo, se ha diseñado un cajón[1] que va atornillado al cuerpo central del quadrotor y que alojará a la batería. Al situar la batería en un punto bajo, el centro de gravedad se verá desplazado hacia el suelo y facilitará la estabilidad del sistema.

[1] En el apartado 7 se sustituirá el cajón por motivos de estabilidad.

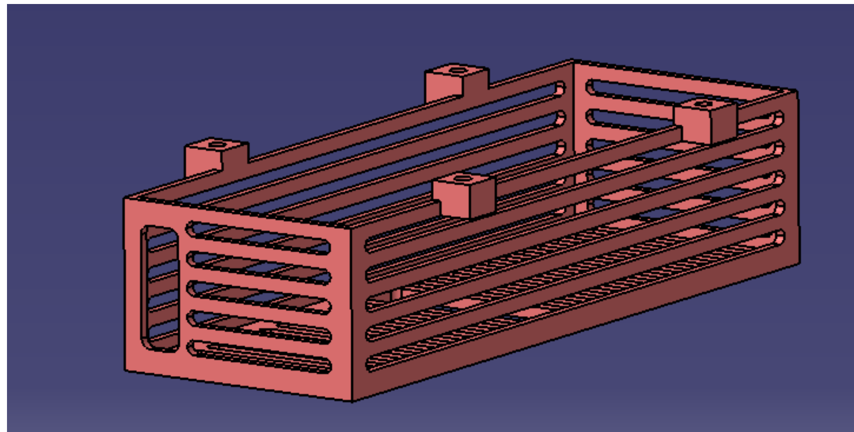


Figura 5-4 Cajón central porta-batería

5.2.4 Fijador

La finalidad última de este elemento es unir las cuatro partes principales y darle rigidez al conjunto. Por este motivo, esta pieza, será fabricada en aluminio (muy ligero a la vez que resistente) en vez de mediante impresión 3D empleando PLA.

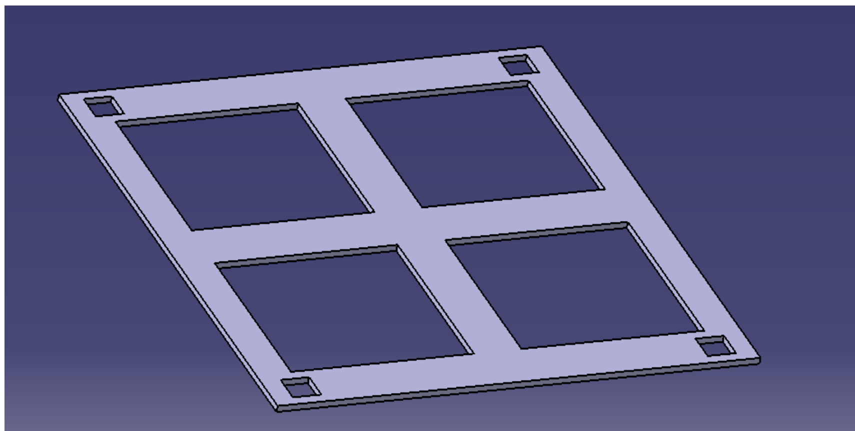


Figura 5-5 Plano del soporte-fijador.

5.3 Cálculos de fuerzas y deformaciones

Para comprobar si la estructura soporta las fuerzas que podemos estimar que vayan a ocurrir, emplearemos una herramienta de cálculo de tensiones y deformaciones que incluye el software de diseño.

5.3.1 Cálculos previos

En primer lugar, debemos estimar las fuerzas que van a ocurrir. Para realizar el caso más desfavorable para la estructura, es decir, aquel caso en el que las fuerzas que esta ha de soportar son las mayores. Esta previsiblemente será cuando los motores estén ejerciendo el empuje vertical máximo:

$$F_{NETA} = Empuje - Peso$$

La fuerza neta que deberá soportar cada brazo del quadrotor es de 4.80 N (los cálculos realizados se justifican en el Anexo I). Analizamos mediante el método los elementos finitos las fuerzas y deformaciones soportadas:

5.3.2 Análisis de fuerzas y deformaciones

El propio software de diseño en 3D nos permite realizar un cálculo de las tensiones y deformaciones que experimentará la pieza empleando para ello el método de elementos finitos. Recordemos que el análisis que se va a realizar es el de un brazo del quadrotor ya que estos componentes son los que previsiblemente van a soportar mayores tensiones, las cuales van a ser iguales en los cuatro por la simetría del propio quadrotor.

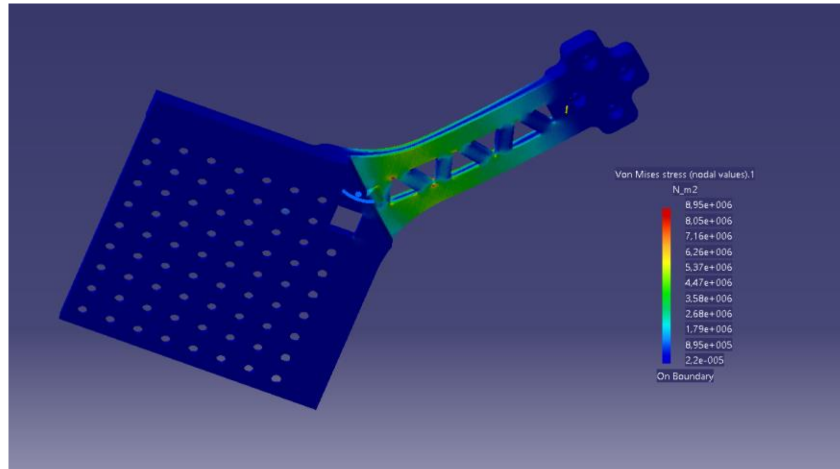


Figura 5-6 Tensiones en el quadrotor (II).

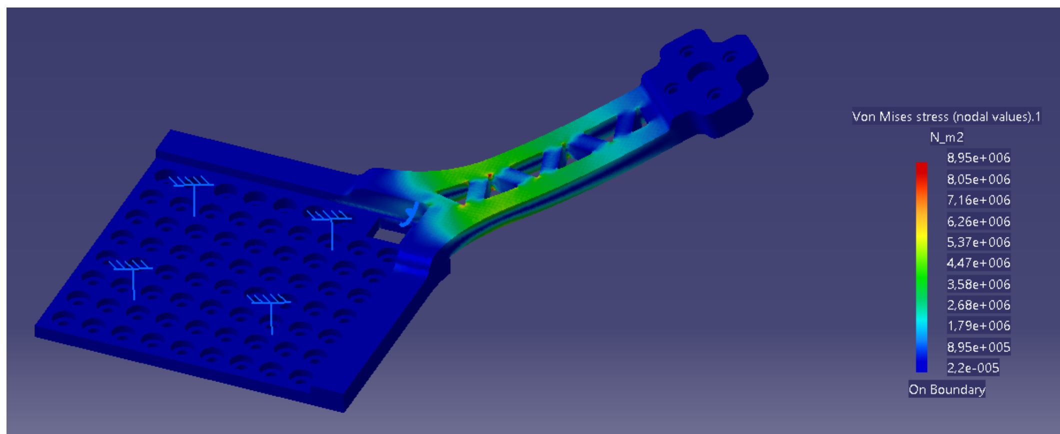


Figura 5-7 Tensiones en el quadrotor (II).

En las imágenes anteriores pueden verse las diferentes tensiones que sufre el brazo en el caso más adverso. De estas, pueden sacarse las siguientes conclusiones:

- La tensión máxima que experimenta el brazo es del orden de 10 MPa y se da en las esquinas de los triángulos del brazo. Sin embargo, el límite elástico del PLA está alrededor de los 50 MPa por lo que podemos asegurar que el material no dejará de tener un comportamiento elástico durante su funcionamiento.

- El pequeño nervio colocado en la unión brazo-cuerpo central de las piezas ha aliviado la tensión de manera que esta zona, que por el diseño es crítica, no sufre grandes tensiones y por tanto no se fracturará al usar el quadrotor.

En cuanto a las deformaciones, el análisis nos arroja los siguientes resultados:

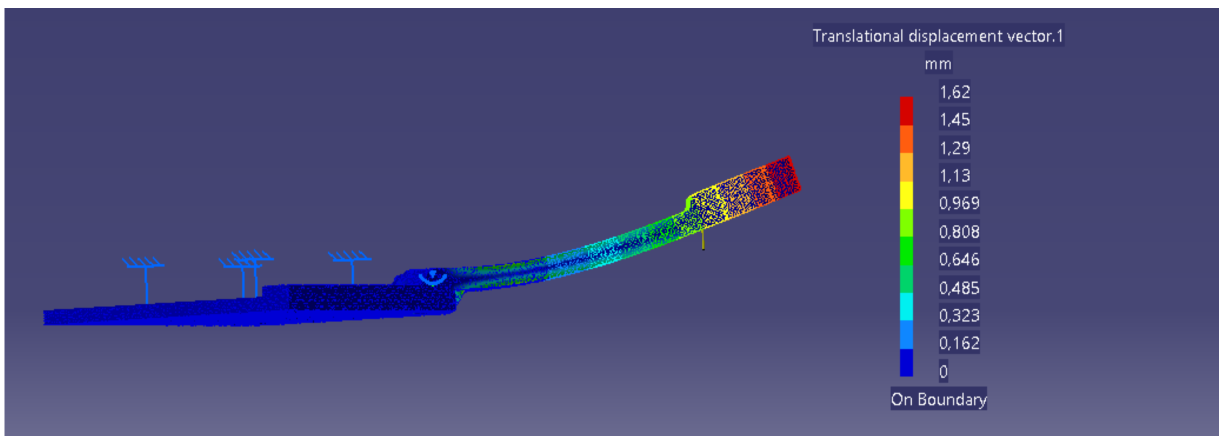


Figura 5-8 Deformaciones en el quadrotor.

Como es previsible, la máxima deformación se dará en el extremo del brazo y será de 1.6 mm, un valor que consideraremos admisible.

5.4 Montaje final

Tras las simulaciones podemos dar el visto bueno al diseño. Ahora es turno de ensamblar todas las piezas del quadrotor. Estas uniones han de ser firmes para que asegure la integridad del conjunto en pleno vuelo, pero a su vez tiene que ser fácil de cambiar en caso de rotura. Por este motivo, para todas las uniones emplearemos tornillos y tuercas. Además, el propio diseño del drone contaba con multitud de taladros en los que se pueden insertar los tornillos. La única excepción en este sentido serían las patas que, dado su reducido tamaño, será más conveniente el empleo de unos pequeños pasadores que bloqueen el movimiento de estas.

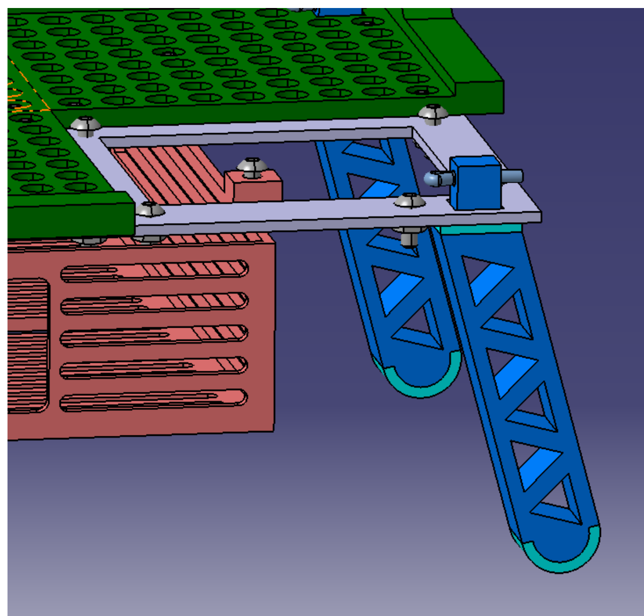


Figura 5-9 Detalle ensamblaje.

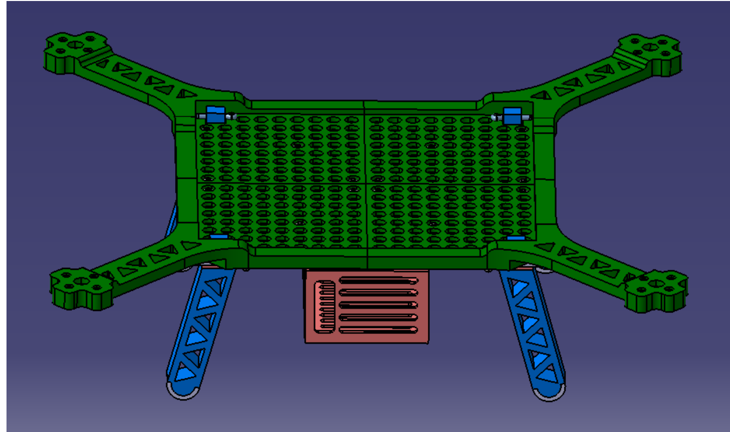


Figura 5-10 Conjunto montado.

6 SISTEMAS ELECTRÓNICOS

En las sucesivas secciones se va a detallar el funcionamiento de los diferentes componentes electrónicos que constituyen el quadrotor.

6.1 Microcontrolador

El primer componente que vamos a explicar el el microprocesador. Este componente va a ser "el cerebro" de nuestro drone, es decir, aquel que va a realizar las tareas de cálculo, va a interactuar con los sensores y actuadores y se va a comunicar con el usuario. Actualmente en el mercado existen multitud de microcontroladores posibles, pero entre ellos, suelen destacar:

- *Arduino:* corresponde a toda una familia de placas de desarrollo muy extendida por su simplicidad. Dependiendo de la función que realizar y su coste computacional existen placas que incluyen procesadores más complejos y potentes.
- *Raspberry Pi:* pequeño PC de prestaciones limitadas que ofrece posibilidad de comunicación con el entorno físico mediante entradas y salidas GPIO. Incluye diferentes protocolos de comunicación tanto cableada (UART, SPI, I2C y Ethernet) como inalámbrica (WiFi y Bluetooth).
- *Placas de desarrollo Texas Instruments:* pequeñas placas de desarrollo muy parecidas a las de Arduino, pero del fabricante Texas Instruments.
- *Otros sistemas más complejos como las placas Intel Galileo*.

Estas son algunas de las diferentes opciones que podemos encontrar en el mercado. En nuestro caso emplearemos la Raspberry Pi. Los motivos para elegir este sistema son su versatilidad, la multitud de prestaciones que tienen y el reto pues es la única de las placas arriba nombradas con las que no se trabaja durante el grado por lo que la elección de esta supone un plus en el aprendiaje que este proyecto supone. En concreto, el departamento nos ha facilitado la Raspberry Pi 3B.

6.1.1 Características

Dentro de la gama Raspberry hay diferentes modelos y cada uno tiene prestaciones diferentes. Como ya se ha indicado en el apartado anterior, en nuestro proyecto emplearemos la Raspberry Pi 3B cuyas especificaciones son las siguientes:

- Quad Core CPU BCM2837 de 64 bits.
- 1 GB de RAM.
- Conexiones inalámbricas Bluetooth y WiFi.
- 40 pines GPIO.
- Puertos USB, HDMI y Jack.
- Alimentación mediante Micro USB.

El empleo de esta placa nos permite tener a nuestra disposición un procesador relativamente potente para la aplicación que nosotros le vamos a dar y la incorporación de manera nativa de diferentes formas de comunicación con periféricos de la placa (tanto de manera inalámbrica como cableada).

6.1.2 Sistema Operativo

Cualquier Raspberry Pi está pensada para ser utilizada con su propio sistema operativo: Raspbian. Este es una adaptación de Debian para las placas RPi. Sin embargo, por los motivos que ahora se expondrán, su uso no resulta adecuado para nuestra aplicación.

6.1.2.1 Clasificaciones de las tareas

Dada la importancia de cumplir exigencias temporales los sistemas pueden clasificarse en:

- *Sistemas críticos:* las tareas han de ejecutarse cada un periodo T y no es admisible un retraso en las tareas.
- *Sistemas no críticos:* se permite un retraso en la ejecución de la tarea. Dentro de estos tenemos:
    - *Sistemas firmes:* si se produce el retraso se ignoran los resultados de la tarea.
    - *Sistemas no firmes:* si se produce el retraso, el resultado de la tarea es válido, pero no óptimo.

Ahora necesitaríamos saber qué tareas ha de llevar a cabo nuestro drone durante su funcionamiento para así poder clasificarlas y saber las restricciones que vamos a tener:

- *Lectura IMU:* tarea encargada de leer los valores de la IMU y traducirlos en los ángulos de Euler.
- *Lectura altitud:* tarea cuya función es calcular a la altitud con respecto al suelo a la que se encuentra el quadrotor.
- *Cálculo PID:* tarea que implementa los controladores PID. Dados los valores de referencia de los ángulos de Euler y la altitud y los valores reales calcula las acciones a aplicar sobre cada motor.
- *Escritura PWM:* comunicación con los motores para indicarle el nuevo régimen de giro.
- *Lectura usuario:* tarea encargada de recepcionar las acciones deseadas por el usuario.

Atendiendo a la lista anterior, podemos prever que en el drone vamos a tener tareas críticas y no críticas. Las primeras son aquellas cuya demora podría suponer la inestabilización del sistema. Las no críticas son aquellas cuya importancia en la estabilidad del sistema no es vital y, por tanto, una demora temporal en su ejecución no tiene un efecto indeseado sobre este.

A continuación, se muestra una tabla resumen de las diferentes tareas del sistema y su clasificación:

Tabla 6-1 Clasificación tareas

Tarea	Clasificación
Lectura IMU	Crítica
Lectura altitud	Crítica
Cálculo PID	Crítica
Escritura PWM	Crítica
Lectura usuario	No crítica

6.1.2.2 Real-Time Operating System

Como ya hemos visto, nuestro sistema posee tareas críticas, es decir, no es admisible demoras en su ejecución. Esto plantea un problema a resolver ya que deberemos elegir de manera adecuada el sistema operativo que correrá en nuestro micro. Hay dos tipos principales de sistemas operativos:

- Sistemas operativos convencionales: son aquellos que están orientados a la eficiencia. El tiempo que transcurre entre tareas no es vital. Algunos ejemplos de este tipo de sistemas son Windows, MacOs, Ubuntu…
- Real-Time Operating System: estos sistemas operativos (RTOS de aquí en adelante) priman la predictibilidad frente a la eficiencia. Es decir, no buscan que el sistema sea lo más eficiente posible, sino que el tiempo entre tareas sea predecible y controlable. Algunos ejemplos son Xenomai o FreeRTOS.

Dado que nuestro sistema contiene restricciones críticas, es necesario el empleo de un RTOS que asegure que dichas imposiciones temporales se van a cumplir. Raspbian proviene de Debian, el cual no es un RTOS, por lo que deberemos buscar alguna alternativa. En nuestro caso emplearemos Xenomai. El proceso seguido en la instalación de Xenomai se detalla en el Anexo II. Podríamos haber elegido la opción de emplear diferentes normas POSIX en Raspbian para tener un pseudo RTOS, pero pudiendo emplear Xenomai con total normalidad, se ha optado por esta opción.

6.1.3 Pines GPIO

GPIO proviene de General Purpose Input/Output y hace referencia a una serie de pines de la Raspberry que son empleados como salidas y entradas digitales. Nótese que son digitales no analógicos, por lo que si quisiésemos emplear un sensor analógico necesitaríamos un convertidor ADC o el empleo de otra placa que permita la lectura analógica (un Arduino por ejemplo). Además de pines de carácter general, algunos implementan funcionalidades específicas como pueden ser soportar los protocolos de comunicaciones o servir de referencias de tensión. En la figura adjunta se muestran los diferentes pines GPIO y sus funcionalidades:

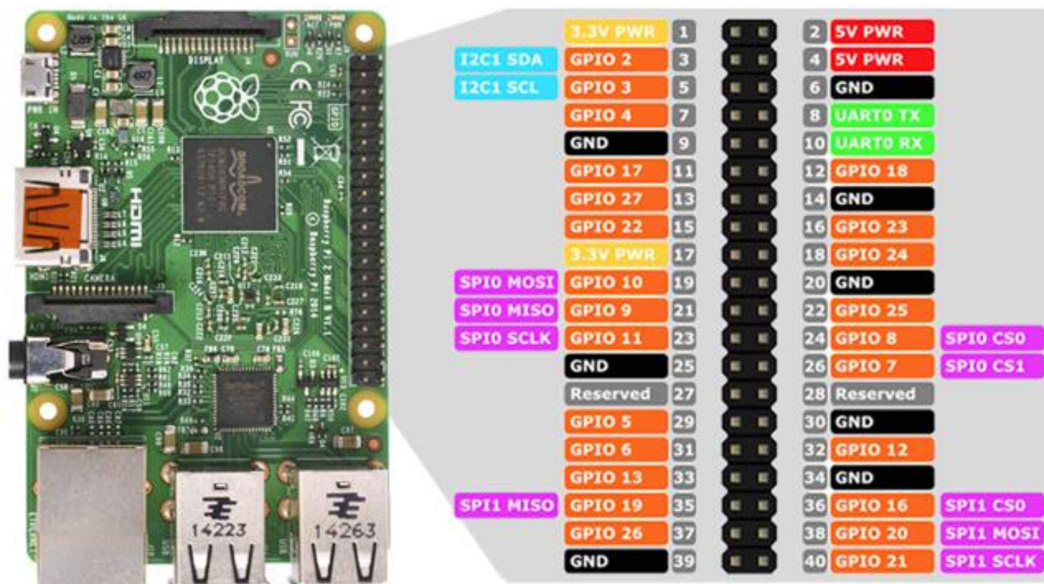


Figura 6-1 Pines GPIO Raspberry Pi 3B

Las funcionalidades de los pines pueden explotarse mediante librerías las cuales ya implementa el protocolo de comunicación a bajo nivel, haciendo que el desarrollo de la aplicación sea

notablemente más sencillo. En los diferentes sensores y actuadores se explicará qué protocolo de comunicación se ha empleado, en qué consiste y cómo se implementa.

6.1.4 Comunicaciones inalámbricas: WiFi y Bluetooth

El ultimo aspecto a mencionar es la integración de forma nativa de comunicaciones WiFi y Bluetooth. En caso de querer usar cualquiera de estas dos formas de comunicaciones inalámbricas para interactuar con el quadrotor ya no sería necesario el empleo de módulos que implementen antenas WiFi o Bluetooth (algo que sí sería necesario en otras tarjetas microcontroladoras como en algunos Arduino). Al igual que pasaba con los pines GPIO, para poder emplear estos periféricos es necesario el empleo de librerías.

6.2 Medidas inerciales

En este apartado nos vamos a centrar en el análisis de uno de los componentes vitales del sistema que aquí estamos diseñando: la IMU. Este despositivo toma su nombre del inglés "Inertial Measurement Unit" y su finalidad es proporcionar diferentes medidas inerciales para poder calcular la orientación del sistema y/o estimar la posición del mismo.

Existen diferentes tipos de IMUs atendiendo a los sensores que esta contenga en su interior y, por lo tanto, a los grados de libertad que sea capaz de medir:

- *3 DOF:* integra un acelerómetro o un giroscopio para cada eje.
- *6 DOF:* integra acelerómetro y giroscopio por cada eje. Por ello es capaz de medir orientaciones en los ejes X e Y del dispositivo.
- *9 DOF:* integra acelerómetro, giroscopio y magnetómetro. Por ello puede proporcionar la orientación en las tres coordenadas cartesianas del sistema.
- *10 DOF:* IMU 9DOF + sensor de altitud. Junto a la orientación puede estimar la altura del sistema.

Con esta clasificación arriba expuesta comienza la toma de decisiones, ¿qué tipo de IMU deberemos emplear? Para nuestra aplicación deberemos medir (al menos) la orientación en los tres ejes coordenados [2]por lo que requeriremos una imu de 9 ó 10 DOF. Esta elección dependerá si será también la IMU quien realice las medidas de altitud o no. En nuestro sistema la IMU no tendrá que desempeñar esa tarea por lo que una de 9DOF proporcionará la información necesaria a nuestro sistema sobre la orientación de este.

En concreto, emplearemos la imu MPU-9250. Algunas características de este sensor es que permite la comunicación mediante SPI o I2C, así como la existencia de un procesador interno el cual permite la lectura de los ángulos directamente para así ahorrar coste computacional al elemento exterior.

---

[2] El ángulo en el eje Z puede estimarse mediante las lecturas de una IMU de 6DOF pero inicialmente se prefirió obtener una medida exacta de este.

Figura 6-2 IMU MPU-9250

6.2.1 Comunicaciones con la IMU

Este modelo de IMU como ya se ha comentado soporta dos tipos de comunicaciones: SPI e I2C.

El protocolo I2C es una forma de comunicación síncrona en la que los diferentes elementos están interconectados mediante dos líneas: una para transmitir la señal de reloj y otra para transmitir los datos. Cada dispositivo posee una dirección física que ha de ser única pues es la manera de la que el maestro (quien comienza la comunicación) intercambia los datos con los esclavos.

La trama de datos enviada por los diferentes dispositivos consta de los siguientes campos:

- 7 bits para definir la dirección física del dispositivo con el que nos vamos a comunicar.
- 1 bit para indicar si enviamos o recibimos información.
- 1 bit de validación.
- 1 o más bytes de datos.
- 1 bit de validación/stop.

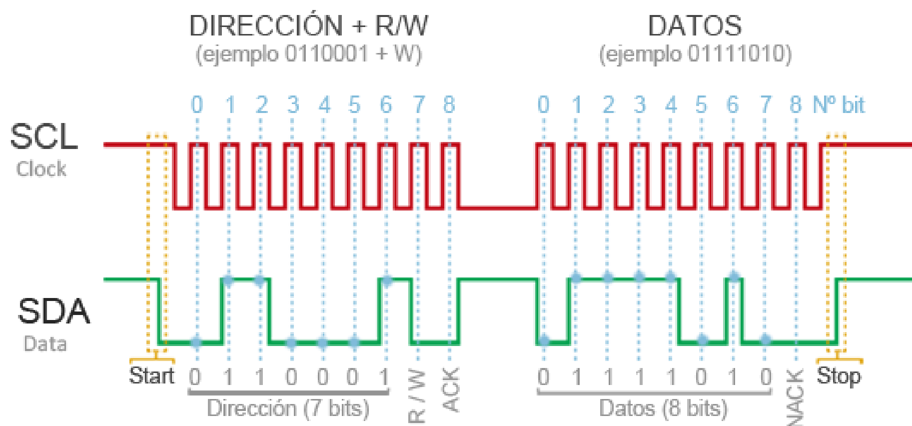


Figura 6-3 Trama paquete I2C

El protocolo SPI también es una forma de comunicación síncrona entre diferentes dispositivos. Un maestro es quien comienza la comunicación y los esclavos le responden a esta. Este tipo de comunicación es Full-Duplex, es decir, cualquier dispositivo puede leer y escribir datos en los buses a la vez. Esto se debe a la mayor complejidad del bus SPI pues consta de al menos 4 líneas: una para la señal de reloj, una para la lectura del los datos por parte de los esclavos (MOSI), otra para la lectura de los datos por parte del maestro (MISO) y una línea de habilitación por cado esclavo.

El funcionamiento es el siguiente: todas las líneas de habilitación están a nivel alto pero, cuando el maestro quiere empezar comunicación con algún esclavo pone su línea a nivel bajo. Es entonces cuando comienzan a transmitirse datos. A diferencia del bus I2C no hay una trama de datos predefinida por lo que los diferentes dispositivos han debido acordar la longitud de los paquetes de datos.

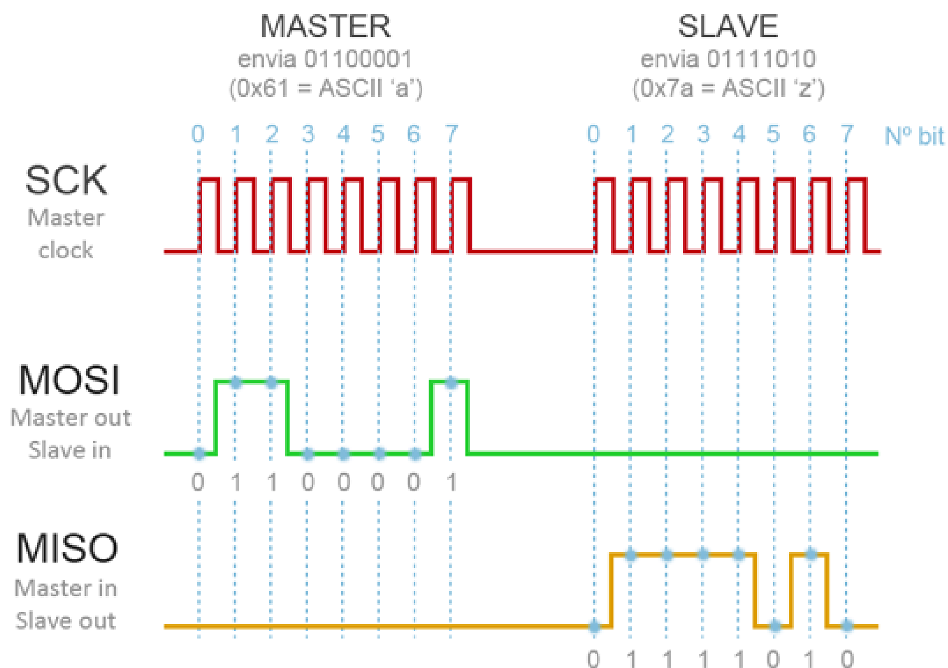


Figura 6-4 Comunicación SPI

Cada uno de los protocolos tienen sus ventajas. El bus I2C necesita menos cables y posee mecanismos para conocer si ha llegado la trama. Sin embargo, el bus SPI es Full-Duplex y consigue velocidades mucho mayores que el bus I2C.

En nuestro caso se decidió en un primer momento el protocolo I2C pues como se verá más adelante, el sensor barométrico se comunica con este mismo protocolo con la RPi. Sin embargo, la experimentación arrojó que esta opción no era la más adecuada. Esto se debe a las velocidades de transimisión de datos de cada protocolo:

Tabla 6-2 Frecuencias protocolos IMU

Protocolo	f_{max}	Unidad
SPI	1	MHz
I2C	400	kHz

El protocolo SPI es más del doble de rápido que el I2C y, dada la gran cantidad de medidas de todos los sensores que necesitaremos realizar, es necesaria una comunicación rápida. El esquema de conexionado de la IMU al bus SPI se analiza en el apartado 6.7.2 Comunicaciones con los diferentes sensores.

6.2.2 Configuración inicial

A continuación, analizaremos aquellos registros que nos son de especial utilidad para la configuración de la IMU.

- *Registro 107 – Power Management 1:* permite despertar al dispositivo, hibernarlo, establecer que se despierte periódicamentes.... Inicialmente mantiene al dispositivo hibernado por lo que deberemos cambiar el modo de funcionamiento para asegurarnos que la imu realice medidas.
- *Registro 108 – Power Management 2:* permite deshabilitar alguna medida. Deberemos asegurarnos de que todas estén habilitadas.

Una vez despertada la IMU configuraremos el giroscopio y el acelerómetro:

- *Registro 27 – Gyroscope Configuration:* nos permite elegir la precisión que deseamos en el giroscopio. Los valores admitidos son 250, 500, 1000 y 2000 º/s donde si elegimos 250 º/s tendremos mucha precisión en la medida a consta de perder rango de medición, y con 2000 º/s sería lo opuesto. En nuestro caso buscamos precisión por lo que estableceremos una escala de 250 º/s.

BIT	NAME	FUNCTION
[7]	XGYRO_Cten	X Gyro self-test
[6]	YGYRO_Cten	Y Gyro self-test
[5]	ZGYRO_Cten	Z Gyro self-test
[4:3]	GYRO_FS_SEL[1:0]	Gyro Full Scale Select: 00 = +250dps 01= +500 dps 10 = +1000 dps 11 = +2000 dps
[2]	-	Reserved
[1:0]	Fchoice_b[1:0]	Used to bypass DLPF as shown in table 1 above. NOTE: Register is Fchoice_b (inverted version of Fchoice), table 1 uses Fchoice (which is the inverted version of this register).

Figura 6-5 Registro 27 - Gyroscope Configuration

- *Registro 28 – Accelerometer Configuration:* es el encargado de elegir la precisión del acelerómetro. En este caso, las escalas son 2, 4, 8 y 16 g. De nuevo, seguimos primando la precisión a un mayor rango, por lo que elegiremos escala 2g.

BIT	NAME	FUNCTION
[7]	ax_st_en	X Accel self-test
[6]	ay_st_en	Y Accel self-test
[5]	az_st_en	Z Accel self-test
[4:3]	ACCEL_FS_SEL[1:0]	Accel Full Scale Select: ±2g (00), ±4g (01), ±8g (10), ±16g (11)
[2:0]	-	Reserved

Figura 6-6 Registro 28 - Accelerometer Configuration

- *Registro 29 – Accelerometer Configuration 2:* implementa un filtro paso bajo para el acelerómetro. Este registro es muy importante pues la propia IMU eliminará gran parte del ruido producido por las vibraciones de los motores (en caso de no configurarla el filtrado y tratamiento de la información que nosotros realicemos será más duro y con peores resultados).

BIT	NAME	FUNCTION
[7:6]	Reserved	
[5:4]	Reserved	
[3]	accel_fchoice_b	Used to bypass DLPF as shown in table 2 below. NOTE: This register contains accel_fchoice_b (the inverted version of accel_fchoice as described in the table below).
[2:0]	A_DLPFCFG	Accelerometer low pass filter setting as shown in table 2 below.

Accelerometer Data Rates and Bandwidths (Normal Mode)

ACCEL_FCHOICE	A_DLPF_CFG	Output 3dB BW (Hz)	Rate (kHz)	Filter Block	Delay (ms)	Noise Density (μg/rtHz)
0	x	1,046	4	Dec1	0.503	300
1	0	218.1	1	DLPF	1.88	300
1	1	218.1	1	DLPF	1.88	300
1	2	99	1	DLPF	2.88	300
1	3	44.8	1	DLPF	4.88	300
1	4	21.2	1	DLPF	8.87	300
1	5	10.2	1	DLPF	16.83	300
1	6	5.05	1	DLPF	32.48	300
1	7	420	1	Dec2	1.38	300

Figura 6-7 Registro 29 - Accelerometer Configuration

- *Registro 26 – Configuration:* este registro es el encargado de aplicar un filtro paso bajo al giroscopio. De nuevo, al igual que pasaba con el acelerómetro, esta tarea es vital para obtener posteriormente buenos resultados. Se ha de tener cuidado pues dicho filtro introduce un delay en las medidas.

FCHOICE		DLPF_CFG	Gyroscope			Temperature Sensor	
<1>	<0>		Bandwidth (Hz)	Delay (ms)	Fs (kHz)	Bandwidth (Hz)	Delay (ms)
x	0	x	8800	0.064	32	4000	0.04
0	1	x	3600	0.11	32	4000	0.04
1	1	0	250	0.97	8	4000	0.04
1	1	1	184	2.9	1	188	1.9
1	1	2	92	3.9	1	98	2.8
1	1	3	41	5.9	1	42	4.8
1	1	4	20	9.9	1	20	8.3
1	1	5	10	17.85	1	10	13.4
1	1	6	5	33.48	1	5	18.6
1	1	7	3600	0.17	8	4000	0.04

Figura 6-8 Registro 26 - Configuración

NOTA: en nuestro caso, tanto para el acelerómetro como para el giroscopio elegiremos una frecuencia de corte de 40 Hz aprox. Para esta frecuencia de corte, la frecuencia de muestreo es de 1 KHz.

- *Registro 25 – Sample Rate Divider:* permite seleccionar la frecuencia de muestreo a la que la imu leerá los valores del acelerómetro y del giroscopio. En nuestro caso dejaremos la frecuencia de muestreo en 1 KHz.

La configuración del magnetómetro es relativamente más compleja pues la comunicación no es directa. Esto se debe a que nosotros nos comunicamos con la imu mediante SPI pero el magnetómetro admite I2C por lo que, las órdenes que nosotros le demos a la imu esta las deberá comunicar mediante un protocolo disitinto. Para habilitar esta comunicación deberemos modificar lo siguiente:

- *Registro 106 – User Control:* permite habilitar el modo multimaster para la comunicación con el magnetómetro.

BIT	NAME	FUNCTION
[7]	Reserved	
[6]	FIFO_EN	1 – Enable FIFO operation mode. 0 – Disable FIFO access from serial interface. To disable FIFO writes by dma, use FIFO_EN register. To disable possible FIFO writes from DMP, disable the DMP.
[5]	I2C_MST_EN	1 – Enable the I2C Master I/F module; pins ES_DA and ES_SCL are isolated from pins SDA/SDI and SCL/ SCLK. 0 – Disable I2C Master I/F module; pins ES_DA and ES_SCL are logically driven by pins SDA/SDI and SCL/ SCLK. NOTE: DMP will run when enabled, even if all internal sensors are disabled, except when the sample rate is set to 8Khz.
[4]	I2C_IF_DIS	1 – Disable I2C Slave module and put the serial interface in SPI mode only.
[3]	Reserved	
[2]	FIFO_RST	1 – Reset FIFO module. Reset is asynchronous. This bit auto clears after one clock cycle.
[1]	I2C_MST_RST	1 – Reset I2C Master module. Reset is asynchronous. This bit auto clears after one clock cycle. NOTE: This bit should only be set when the I2C master has hung. If this bit is set during an active I2C master transaction, the I2C slave will hang, which will require the host to reset the slave.
[0]	SIG_COND_RST	1 – Reset all gyro digital signal path, accel digital signal path, and temp digital signal path. This bit also clears all the sensor registers. SIG_COND_RST is a pulse of one clk8M wide.

Figura 6-9 Registro 106 – User Control

- *Registro 36 – I2C Master Control:* es el registro encargado de establecer la frecuencia de reloj en la comunicación I2C entre la IMU y el magnetómetro. En nuestro caso la hemos escogido de 400 kHz.

BIT	NAME	FUNCTION
[7]	MULT_MST_EN	Enables multi-master capability. When disabled, clocking to the I2C_MST_IF can be disabled when not in use and the logic to detect lost arbitration is disabled.
[6]	WAIT_FOR_ES	Delays the data ready interrupt until external sensor data is loaded. If I2C_MST_IF is disabled, the interrupt will still occur.
[5]	SLV_3_FIFO_EN	1 – write EXT_SENS_DATA registers associated to SLV_3 (as determined by I2C_SLV0_CTRL and I2C_SLV1_CTRL and I2C_SLV2_CTRL) to the FIFO at the sample rate; 0 – function is disabled
[4]	I2C_MST_P_NSR	This bit controls the I2C Master's transition from one slave read to the next slave read. If 0, there is a restart between reads. If 1, there is a stop between reads.
[3:0]	I2C_MST_CLK [3:0]	*I2C_MST_CLK* is a 4 bit unsigned value which configures a divider on the MPU-9250 internal 8MHz clock. It sets the I²C master clock speed according to the following table:

I2C_MST_CLK	I²C Master Clock Speed	8MHz Clock Divider
0	348 kHz	23
1	333 kHz	24
2	320 kHz	25
3	308 kHz	26
4	296 kHz	27
5	286 kHz	28
6	276 kHz	29
7	267 kHz	30
8	258 kHz	31
9	500 kHz	16
10	471 kHz	17
11	444 kHz	18
12	421 kHz	19
13	400 kHz	20
14	381 kHz	21
15	364 kHz	22

Figura 6-10 Registro 36 – I2C Master Control

Como se ha comentado, la comunicación con el magnetómetro es diferente a la realizada con los otros sensores. Esta se basa en el empleo de tres registros:

- *Registro 37 – I2C_SLV0_ADDR:* este registro permite establecer la comunicación con el dispositivo esclavo (en nuestro caso el magnetómetro). Se le ha de indicar la dirección física del dispositivo y el modo de comunicación (lectura o escritura).

BIT	NAME	FUNCTION
[7]	I2C_SLV0_RNW	1 – Transfer is a read 0 – Transfer is a write
[6:0]	I2C_ID_0[6:0]	Physical address of I2C slave 0

Figura 6-11 Registro 37 – I2C_SLV0_ADDR

- *Registro 38 – I2C_SLV0_REG:* permite seleccionar el registro que se desea leer/escribir del dispositivo esclavo.

BIT	NAME	FUNCTION
[7:0]	I2C_SLV0_REG[7:0]	I2C slave 0 register address from where to begin data transfer

Figura 6-12 Registro 38 – I2C_SLV0_REG

- *Registro 39 – I2C_SLV0_CONTROL:* registro que ejecuta la acción de leer/escribir el numero de bytes definidos al dispositivo esclavo (en nuestro caso el magnetómetro).

BIT	NAME	FUNCTION
[7]	I2C_SLV0_EN	1 – Enable reading data from this slave at the sample rate and storing data at the first available EXT_SENS_DATA register, which is always EXT_SENS_DATA_00 for I2C slave 0. 0 – function is disabled for this slave

Figura 6-13 Registro 39 – I2C_SLV0_CONTROL I

[3:0]	I2C_SLV0_LENG[3:0]	Number of bytes to be read from I2C slave 0

Figura 6-14 Registro 39 – I2C_SLV0_CONTROL II

Conocida ya la dinámica de cómo debermos comunicarnos con el magnetométro, deberemos elegir su modo de funcionamiento. Para ello escribiremos en el *Registro 0A – Control 1* el modo de funcionamiento que deseamos. En nuestro caso, la imu estará constantemente tomando medidas por lo que será medida contínua de 16 bits a 100 Hz (máxima tasa de muestreo del magnetómetro).

5.8 CNTL1: Control 1

Addr	Register name	D7	D6	D5	D4	D3	D2	D1	D0
Read-only register									
0AH	CNTL1	0	0	0	BIT	MODE3	MODE2	MODE1	MODE0
Reset		0	0	0	0	0	0	0	0

MODE[3:0]: Operation mode setting

"0000": Power-down mode
"0001": Single measurement mode
"0010": Continuous measurement mode 1
"0110": Continuous measurement mode 2
"0100": External trigger measurement mode
"1000": Self-test mode
"1111": Fuse ROM access mode
Other code settings are prohibited

BIT: Output bit setting

"0": 14-bit output
"1": 16-bit output

Figura 6-15 Registro 0A – Control 1

NOTA: el magnetómetro tiene una escala de 4800 uT fija por lo que no se puede elegir tal y como hicimos con el acelerómetro y el giroscopio.

6.2.3 Lectura de parámetros

Ahora nos centraremos en leer los datos proporcionados por los sensores. Cada uno de los sensores posee la medida de los sensores en 6 registros distintos (2 registros por cada eje). Para obtener todas las medidas deberemos leer cada uno de los registros y combinarlos dos a dos convirtiendo el valor resultante a complemento a dos. De esta manera obtendremos las nueve mediciones (3 del acelerómetro, 3 del giroscopio y 3 del magnetómetro) en formato raw.

49	73	EXT_SENS_DATA_00	R	EXT_SENS_DATA_00[7:0]
4A	74	EXT_SENS_DATA_01	R	EXT_SENS_DATA_01[7:0]
4B	75	EXT_SENS_DATA_02	R	EXT_SENS_DATA_02[7:0]
4C	76	EXT_SENS_DATA_03	R	EXT_SENS_DATA_03[7:0]
4D	77	EXT_SENS_DATA_04	R	EXT_SENS_DATA_04[7:0]
4E	78	EXT_SENS_DATA_05	R	EXT_SENS_DATA_05[7:0]
4F	79	EXT_SENS_DATA_06	R	EXT_SENS_DATA_06[7:0]

Figura 6-16 Registros medidas magnetómetro

3B	59	ACCEL_XOUT_H	R	ACCEL_XOUT_H[15:8]
3C	60	ACCEL_XOUT_L	R	ACCEL_XOUT_L[7:0]
3D	61	ACCEL_YOUT_H	R	ACCEL_YOUT_H[15:8]
3E	62	ACCEL_YOUT_L	R	ACCEL_YOUT_L[7:0]
3F	63	ACCEL_ZOUT_H	R	ACCEL_ZOUT_H[15:8]
40	64	ACCEL_ZOUT_L	R	ACCEL_ZOUT_L[7:0]
41	65	TEMP_OUT_H	R	TEMP_OUT_H[15:8]
42	66	TEMP_OUT_L	R	TEMP_OUT_L[7:0]
43	67	GYRO_XOUT_H	R	GYRO_XOUT_H[15:8]
44	68	GYRO_XOUT_L	R	GYRO_XOUT_L[7:0]
45	69	GYRO_YOUT_H	R	GYRO_YOUT_H[15:8]
46	70	GYRO_YOUT_L	R	GYRO_YOUT_L[7:0]
47	71	GYRO_ZOUT_H	R	GYRO_ZOUT_H[15:8]
48	72	GYRO_ZOUT_L	R	GYRO_ZOUT_L[7:0]

Figura 6-17 Registros medidas acelerómetro/giroscopio

Resulta conveniente realizar unas mediciones del acelerómetro y del giroscopio en reposo para calcular el offset que estos poseen y así poder tener una lectura más fina de la realidad.

Tras obtener los valores raw sin el offset de cada eje, deberemos transformar este valor en unidades de ingeniería (m/s^2 para el acelerómetro, °/s para el giroscopio y uT para el magnetómetro):

$$m/s^2 = (Raw_{accel} - offset_{accel}) * \frac{1}{factor\ de\ sensibilidad\ accel} * 9.8$$

$$°/s = \left(Raw_{gyro} - offset_{gyro}\right) * \frac{1}{factor\ de\ sensibilidad\ gyro}$$

$$uT = Raw_{magnet} * factor\ sensibilidad\ magnet$$

NOTA: los factores de sensibilidad dependen de la escala que hayamos elegido. Estos valores vienen dados por el fabricante en el datasheet del dispositivo.

6.3 Control de altitud

Junto a la IMU, uno de los aspectos más importantes del drone es el control de la altitud de este ya que no deseamos que ascienda sin control causando un peligro potencial o que se estrelle con el suelo. Para esta tarea, necesitaremos sensores que nos proporcionen los datos suficientes para el conocer la altitud a la que se encuetra nuestro quadrotor. Estos son dos: un sónar y un módulo de presión barométrica.

6.3.1 Sónar

Este dispositivo emplea el sonido como método de medida de la distancia. El esquema de funcionamiento de este dispositivo es el siguiente:

1. Un emisor emite una onda de pulsos de alta frecuencia que viaja por el aire.
2. Dicha onda se reflejará en algún objeto y será interceptada por un receptor.
3. Se calcula el tiempo que tarda desde que se emite la onda hasta que se recibe y conociendo la velocidad del sonido en el medio, sonido podemos conocer la distancia del objeto.

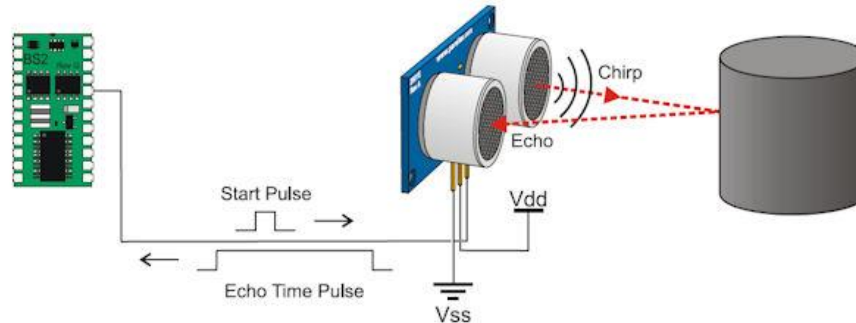


Figura 6-18 Esquema funcionamiento sensor ultrasónico

Para nuestro proyecto emplearemos el HC-SR04, el cual tiene estos datos de funcionamiento:

Tabla 6-3 Especificaciones HC-SR04

Medida	Valor	Unidad
Tensión alimentación	5	V
Tensiones salida	5	V
Resolución	1	cm
Distancia máx.	300	cm
Distancia min.	3	cm

De la siguiente tabla se aprecia que deberemos adaptar la salida del sensor para que sea compatible con los valores de tensión de los pines GPIO de la RPi (ver PCB) y que no podremos emplear este sensor para medir la altitud en cualquier instante, deberemos combinarlo con otro para aquellos rangos de medida que se encuentren fuera del alcance de este.

6.3.2 Barómetro

Para aquellos valores de altitud fuera del rango del sónar emplearemos un sensor de presión barométrica. En nuestro caso emplearemos el sensor BMP-180 del fabricante BOSCH.

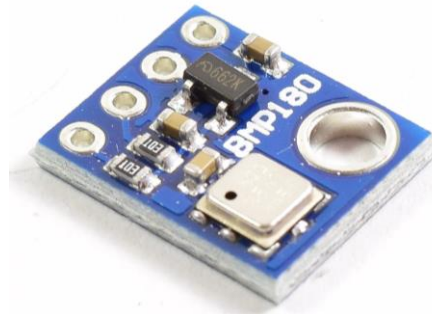


Figura 6-19 Sensor barométrico BMP-180

Las características del datasheet del dispositivo son las siguientes:

Tabla 6-4 Especificaciones BMP-180

Parametro	Valor	Unidad
Presión	300 - 11000	hPa
Precisión	±1	m
T_{sample}	4.5 – 76.5	ms
Voltaje	1.8 – 3.6	V

Además, este sensor tiene como medio de cominicación el protocolo I2C con una frecuencia máxima de 3.4 MHz.

Este sensor tiene 4 modos de obtención de la presión barométrica: ultra low power mode, standard mode, high resolution mode y ultra high resolution mode. Estos tienen como principal diferencia el ruido de la medida y el tiempo de conversión donde, el ultra low power mode es el más rápido, pero más ruidoso y el ultra high resolution mode el más lento, pero menos ruidoso.

El funcionamiento del sensor es el siguiente: se solicita medida de la temperatura y se ha de esperar un periodo igual al tiempo de conversión. Tras esto se lee el valor medido y se pide al sensor la medida de la presión. Se vuelve a esperar el tiempo de conversión y se lee este valor. Una vez tomados estos datos, se aplican las ecuaciones proporcionadas por el fabricante (Figura 6-20) y se obtiene la difencia de altura con respecto a la inicial. Desde el inicio del ciclo hasta la obtención de la altura se ha debido esperar dos veces el tiempo de conversión por lo que, para nuestro sistema, se hace vital que este sea el menor posible. Por ello, el modo de funcionamiento ha de ser el ultra low power mode cuyo tiempo de conversión es de 4.5 ms.

Read calibration data from the E²PROM of the BMP180	
read out E²PROM registers, 16 bit, MSB first	
AC1 (0xAA, 0xAB)	(16 bit)
AC2 (0xAC, 0xAD)	(16 bit)
AC3 (0xAE, 0xAF)	(16 bit)
AC4 (0xB0, 0xB1)	(16 bit)
AC5 (0xB2, 0xB3)	(16 bit)
AC6 (0xB4, 0xB5)	(16 bit)
B1 (0xB6, 0xB7)	(16 bit)
B2 (0xB8, 0xB9)	(16 bit)
MB (0xBa, 0xBB)	(16 bit)
MC (0xBC, 0xBD)	(16 bit)
MD (0xBE, 0xBF)	(16 bit)

Figura 6-20 Valores calibración barómetro

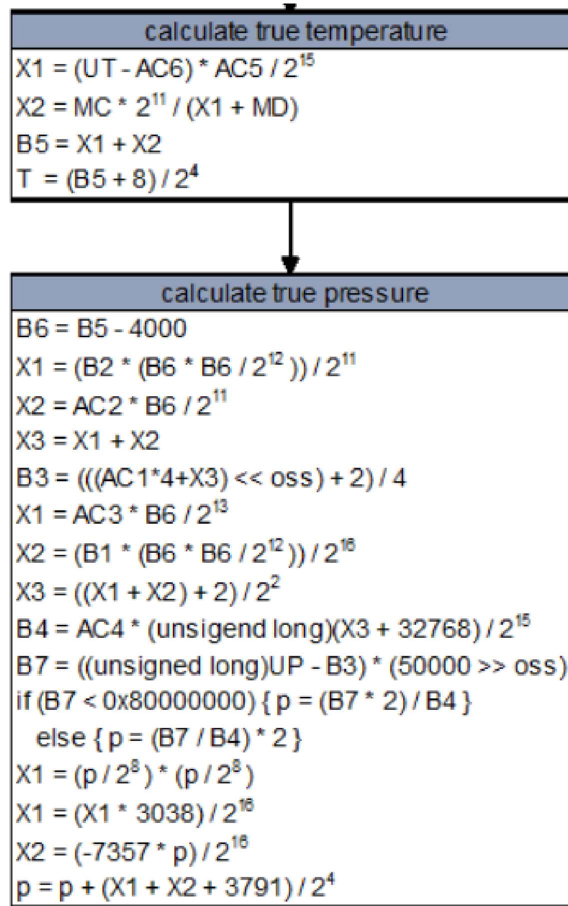


Figura 6-21 Ecuaciones cálculo presión atmosférica

$$\text{altitude} = 44330 * \left(1 - \left(\frac{p}{p_0}\right)^{\frac{1}{5.255}}\right)$$

Figura 6-22 Ecuación cálculo altitud

6.4 Motores Brushless y Variadores ESC

En este apartado trataremos uno de los componentes fundamentales del drone: los motores. Además, explicaremos cómo controlarlos empleando los variadores.

6.4.1 Motores Brushless

Los motores brushless son un tipo de transductor eléctrico/mecánico, es decir, unos dispositivos que transforman la energía eléctrica en energía mecánica. En concreto, estos motores son alimentados con corriente continua a diferencia de otros que han de ser excitados con alterna.

Este tipo de motores están constituidos por un estator central, en el cuál se enrollan bobinas correspondientes a diferentes fases que haya en el motor y un rotor formado por una carcasa giratoria en la que se encuentran fijados los imanes permanentes. Al excitar las bobinas el estator, el rotor tenderá a alinearse con el campo magnético producida por estas logrando así un giro.

El control el régimen de giro del motor se realiza variando la velocidad con la que se excitan las bobinas que componen el estator. Esta tarea la realiza el ESC, que más adelante se explicará.

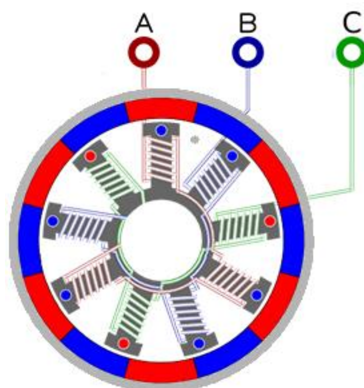


Figura 6-23 Esquema interno motor brushless

Visto el funcionamiento, veamos las ventajas que este nos aporta al emplearlo frente a los motores convencionales con escobillas:

- *Más eficiente y duradero:* al no realizarse la conmutación entre las bobinas mediante escobillas sino eléctricamente, se produce una menor pérdida por rozamiento en las escobillas al igual que el mantenimiento es menor por ese mismo motivo.
- *Mayor relación par/motor:* al no tener escobillas, al aumentar la velocidad no se produce fricción y por lo tanto el par no se ve disminuido.
- *Mayores rangos de velocidad.*

Por el contrario, estos motores son más caros que los motores con escobillas.

Nuestro drone constará de 4 motores BL2210/30 del fabricante EMAX. Este modelo tiene las siguientes especificaciones:

Tabla 6-5 Especificaciones BL2210/30

Parametro	Valor	Unidad
V_n	7.4 - 11	V
I_{max}	16.5	A
R. giro	1300[3]	KV
Peso	45	g

Figura 6-24 Motor brushless BL2210/30

6.4.2 Variadores ESC

La palabra ESC proviene de “Electronic Speed Controller”, es decir, “Controlador de velocidad electrónico”. Estos dispositivos tienen la función de cambiar la excitación de las distintas fases de la bobina de manera que, como ya se explicó en los motores, al cambiar la excitación de las bobinas, se produce un campo magnético inducido y esto provoca el giro del rotor. Además, permiten cambiar la frecuencia de excitación y por tanto el régimen de giro del motor.

El control de estos componentes se realiza mediante ondas cuadradas de periodo T (este tipo de regulación de llama PWM):

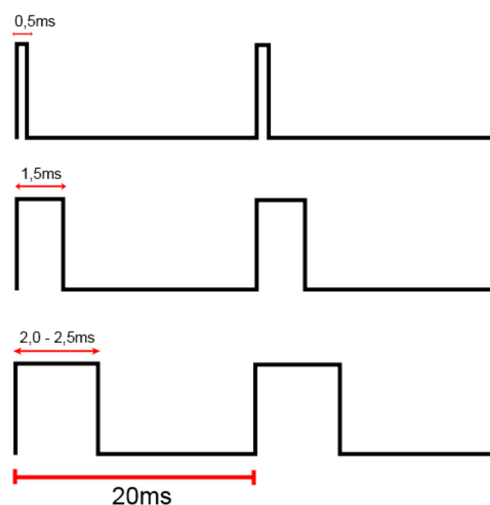


Figura 6-25 Ancho pulso onda de control del variador

[3] A mayor KV el motor será más rápido, pero proporcionará un par menor, mientras que si es un valor bajo es muy lento pero con un gran par. Es por ello por lo que hemos elegido un motor con un valor intermedio.

En particular el periodo T es de 20 ms (tal y como aparece en la imagen) y la regulación del ancho de pulso ha de ser la siguiente:

- La señal debe inicializarse a nivel alto durante 1 ms.
- A continuación, si se quiere una potencia de giro del 20% se deberá mantener activa 0.2 ms más, si se quiere al 70% debería mantenerse 0.7 ms más y así con el valor deseado.
- El resto del ciclo la señal debe mantenerse a nivel bajo.

Antes de probar deberemos fijarnos en que las especificaciones de tensión y de corriente máxima admisible sean compatibles con los motores. En nuestro caso hemos empleado variadores BUDGET con las siguientes características:

Tabla 6-6 Especificaciones ESC BUDGET

Parametro	Valor	Unidad
V_n	2-4S	-
I_{max}	18	A
BEC	Sí	-

Figura 6-26 ESC BUGET 18A

6.4.3 Conexionado

En este apartado procederé a explicar cómo han de conectarse los variadores y los motores para que puedan ser gobernados desde la RPi.

6.4.3.1 Conexión ESC

Los variadores que nosotros vamos a emplear se denominan BEC o “Battery Eliminator Circuit”. Estos son un tipo de ESC que no requieren de alimentación extra para su funcionamiento, sino que tienen una etapa de regulación de tensión de manera que cogen la energía necesaria para funcionar de la propia corriente que se les inyecta a los motores. En caso de que no fueran BEC, deberíamos emplear otra batería de menor tensión nominal que la empleada para los motores y así poder alimentar al variador.

El esquema del conexionado sería el siguiente:

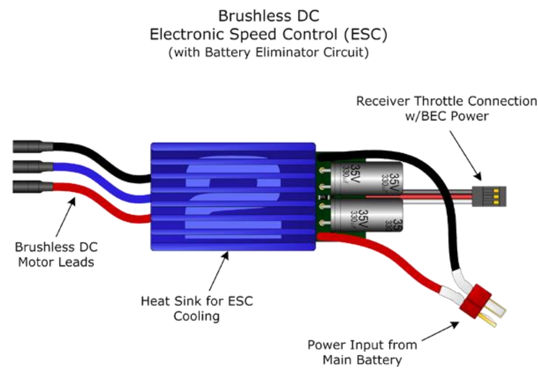


Figura 6-27 Conexionado RPi - ESC

En este apartado nos encargaremos del conexionado de "la parte derecha de la imagen", es decir, de la conexión con la Raspberry. En primer lugar, deberemos conectar el ESC a la batería[4] mediante unos conectores en T. Se ha de tener especial cuidado a la hora de conectarlo pues de hacerlo con la polaridad cambiada es muy probable que se dañe el variador. En segundo y último lugar, deberemos conectar los pines de control llamados en la imagen como "Throttle Connection" a la RPi de la siguiente manera:

- Cable negro del ESC conectado a tierra de la RPi. Este hará la función de masa.
- Cable rojo del ESC se deja al aire.
- Cable blanco del ESC conectado a cualquier pin GPIO (preferiblemente que no tenga ninguna funcionalidad especial). A través de este pin comunicaremos la señal PWM para el control del brushless.

6.4.3.2 Conexionado motores brushless

Ahora nos centraremos en la conexión entre la salida del variador y de los motores. Esta es realmente sencilla, basta con conectar cada una de las tres salidas del variador a los cables de alimentación del motor brushless. De esta manera es cierto que si activamos el motor este rotará, pero ¿en qué sentido? El sentido de giro dependerá de cómo hayamos unidos los cables. Si deseáramos cambiar el giro sería tan sencillo como intercambiar dos de los tres cables de alimentación como se ve en la imagen siguiente:

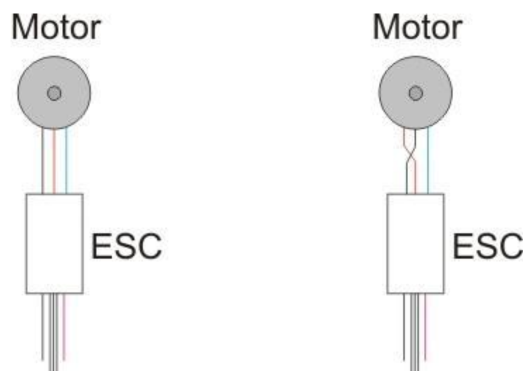


Figura 6-28 Conexión ESC - Motor

Determinar el sentido de giro no es un asunto menor ya que para que el drone se mantenga estable en el aire, cada motor ha de girar en un sentido determinado (dependiendo de la disposición y

[4] Los variadores han de ser compatibles al tipo de batería que vayamos a usar.

cantidad de motores que tengamos) y dicho sentido ha de ser coherente con la hélice que se le coloque.

6.4.4 Control por software

Una vez que ya tenemos conexionado los motores y los variadores es hora de explicar cómo se controlan estos empleando la RPi. En primer lugar, he de aclarar que hay multitud de formas posibles y la que aquí expongo es la opción que he elegido para este proyecto, pero no es la única.

Para realizar el control por software es muy cómodo emplear alguna librería ya que esta implementa la comunicación a bajo nivel de la CPU con los periféricos. De esta manera, nosotros nos ahorramos la tarea de tener que interactuar con los periféricos reduciendo así la posibilidad de errores en el código. En nuestro caso, como se indica en el Anexo II, hemos empleado la librería pigpio.

Esta librería incluye la función gpioServo(unsigned gpio, unsigned ancho_pulso). Esta es una función que inicialmente estaba pensada para controlar servomotores pero que puede ser utilizada para el control de los variadores ya que la diferencia de control entre uno y otro es mínima. Así, para poder controlar mediante la RPi lo motores deberemos:

- Crear una tarea periódica de periodo 20 ms.
- En la tarea anterior llamar a la función gpioServo donde el primer argumento de entrada corresponde al pin GPIO al que se ha conectado el variador, y el segundo corresponde al ancho del pulso que le queremos mandar (en microsegundos).

Si queremos parar el motor, el ancho del pulso debería ser 1000 + 0 = 1000 µs y la orden quedaría:

```
gpioServo(17,1000) // Para este ejemplo se conecta el ESC al pin GPIO 17
```

Si quisiéramos el motor a máxima potencia, el ancho del pulso sería 1000 + 1000 = 2000 µs:

```
gpioServo(17,2000) // Para este ejemplo se conecta el ESC al pin GPIO 17
```

NOTA: antes de poner en movimiento los motores deberemos enviarle pulsos de 1000 us para que adquieran la referencia.

6.4.5 Determinación de parámetros de funcionamiento

Cuando estamos trabajando con sistema reales la realidad difiere de la teoría. Aunque esta establece que el motor adquiere su valor máximo cuando el ancho de pulso es de 2000 µs y se para cuando el ancho es de 1000 µs en la realidad puede que no sea así (y de hecho no lo es). En los motores existen dos valores que son muy importantes conocer para poder actuar de manera eficaz sobre ellos:

- *Zona muerta:* aquel rango de valores en los que el motor debería rotar, pero debido a la no idealidad del sistema no lo hace.
- *Zona de saturación:* valor a partir del cual el motor no es capaz de rotar a mayor velocidad.

Para poder diseñar unos controladores que sean realmente eficaces deberemos determinar estos valores para cada uno de los motores. Con el fin de determinarlos realizaremos el siguiente ensayo: partiendo de un ancho de pulso de 1000 µs se irá aumentando este hasta que el motor comience a rotar. Este valor determinará la zona muerta. A continuación, se aumentará el ancho del pulso hasta que perceptiblemente el rotor no gire más rápido por más que aumentemos el ancho del pulso. Este último valor corresponderá con la zona muerta del motor. Para nuestros motores, los valores obtenidos son:

Tabla 6-7 Valores de funcionamiento de los motores

Motor	Zona muerta	Zona de saturación
Motor 1	1045	>1900
Motor 2	1045	>1900
Motor 3	1045	>1900
Motor 4	1045	>1900

Como era previsible, los valores mínimos y máximos reales de funcionamiento difieren de la teoría.

6.4.6 Disposición de los motores

En la práctica, la disposición en la que se pueden colocar los motores no es única, sino que hay multitud de maneras. Sin embargo, muchas ellas tienen algo en común, que hay un número par de motores:

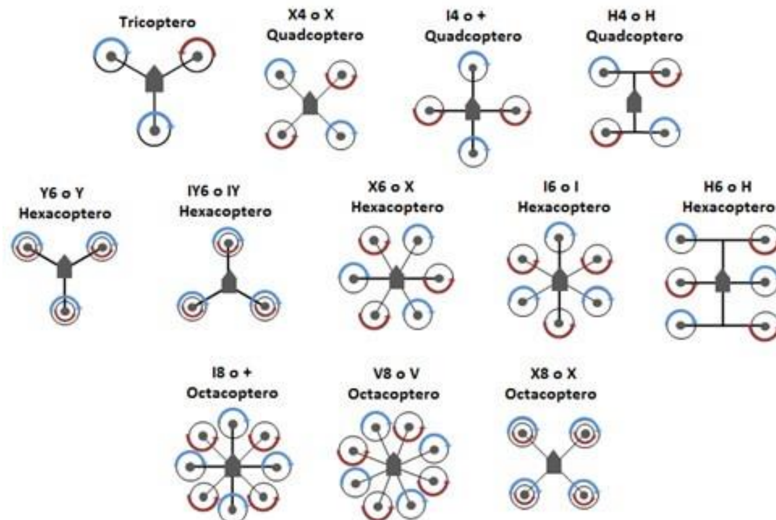


Figura 6-29 Disposiciones rotores en un drone

Esto se debe a que, en la mayoría de los casos por cada motor que gira en un sentido hay otro que gira en el sentido opuesto. De esta manera se consigue una cierta regulación del spin del propio drone sin haber aplicado ningún controlador y, por lo tanto, simplificando el diseño de los controladores. Ahora bien, la elección de la configuración depende de muchos factores como:

- La carga que se desee levantar ya que, a mayor cantidad de motores mayor será el empuje del quadrotor, y, por tanto, mayor será el peso que pueda levantar.

- A mayor cantidad de motores más seguro es el drone en caso de que falle un motor.
- Un mayor número de motores se reduce en un mayor desembolso.
- Cuantos más motores funcionando haya menor será el tiempo de autonomía del sistema.

En nuestro caso, dado que no deseamos levantar un gran peso optaremos por un diseño en el que haya pocos motores y sea un número par. Esto nos facilitará el diseño del drone, el diseño de los controladores y reducirá el coste del proyecto. En concreto, la configuración elegida para nuestro drone será “4H[5]”.

6.5 Batería LiPo

La batería es un elemento indispensable en el drone. Es el componente encargado de suministrar la energía eléctrica a los diferentes sistemas del quadrotor (motores, sensores y microcontrolador). La elección de esta no es tarea trivial ya que hay que tener en cuenta diferentes aspectos:

- *Capacidad:* se mide en mAh. A mayor capacidad el drone podrá mantenerse en el aire un mayor tiempo. Sin embargo, a mayor capacidad también aumenta el peso de la batería.
- *Tensión nominal:* depende del número de celdas que posea la batería (donde cada celda posee una tensión nominal de 3.7 V. Así, una batería 3S tendrá una tensión de 11.1 V.
- *Tasa de descarga:* corriente de descarga que es capaz de soportar en funcionamiento normal. Una batería de 1000 mAh (ó 1 Ah) con una tasa de descarga de 20C podrá suministrar 20 A de manera nominal.

En nuestro caso, el fabricante nos recomienda el empley de una batería 2-3S. En cuanto a la capacidad de descarga, la corriente máxima de los motores es de 16.5 A por lo que los cuatro consumirán 66 A en el caso de mayor consumo posible. Para cumplir estas especificaciones hemos optado por la siguiente batería:

Tabla 6-8 Especificaciones batería ZIPPY Flightmax

Parametro	Valor	Unidad
V_n	3S (11.1 V)	-
Capacidad	5000	mAh
Tasa desc.	20C (100 A)	-

Figura 6-30 Batería ZIPPY Flightmax

[5] Podría haber sido cualquier otra disposición con 4 motores.

6.6 Cableado de potencia

En el apartado anterior se mostraba cómo se debía conectar un motor, tanto eléctrica como electrónicamente. Sin embargo, cuando queremos conectar más de un motor nos surge un problema: la batería sólo tiene un conector y nosotros queremos conectar a la batería 4 motores, ¿cómo lo hacemos?

Deberemos buscar algún sistema que distribuya la energía entre todos los motores. Para ello hay varias alternativas, unas más válidas que otras:

- *Empleo de una batería por motor:* esta opción no resulta viable ya que a la vez que ocuparía mucho espacio incrementaría el peso del drone de una manera muy significativa.
- *Empleo de una placa de distribución de potencia:* eesta podría haber sido una buena opción pues a la vez de tener salida para 4 motores posee una salida a 5V que podría haberse empleado para alimentar la RPi. Sin embargo, como se verá en el siguiente punto, necesitamos emplear una PCB para nuestros sistemas por lo que ya no queda espacio en nuestro drone para colocar la placa de potencia.

Figura 6-31 Placa de distribución de potencia

- *Creación de cableado propio:* optaremos por realizar una multiplexación de cables de tal manera que, un solo cable se conecte a la batería y este se subdivida en 4 (uno para cada motor).

Para crear nuestros cables vamos a necesitar los siguientes componentes:

- *Cable rojo y negro de 14 AWG [6]con recubrimiento de silicona.*
- *Conectores especiales de tipo XT-60.*
- *Conectores especiales de tipo T-Deans.*
- *Tubos termo retráctiles.*

---

[6] AWG es la medida de calibre americano. Correspondería con una sección de 2 mm².

Analicemos los diferentes componentes:

- El cable empleado es un tipo de cable flexible que soporta hasta 24 A en la configuración unipolar. Si quisiéramos que aguantasen una mayor carga deberían tener una sección mayor (12 AWG, 10 AWG…), pero sería más difícil de trabajar con ellos en la aplicación que les queremos dar.
- Los conectores que vamos a emplear son especiales para aeromodelismo. Estos aseguran la conexión de tal manera que en mitad del vuelo no sorprenda un motor parándose por haberse desconectado.
    - Los XT-60 pertenecen a la familia de los XT. Están pensados para la conexión a las baterías y son capaces de soportar hasta 60 A.
    - Los T-Dean están también pensados para la conexión a las baterías, pero sólo son capaces de soportar 35 A.

Entonces, si los XT-60 y los T-Dean tienen la misma funcionalidad, ¿por qué no empleamos sólo un tipo? La respuesta es por seguridad y por criterio económico. Por seguridad podríamos realizar todas las conexiones con los conectores XT-60 pero estos tienen el inconveniente de que son más caros que los T-Dean. Por ese motivo, emplearemos los XT-60 en aquellas conexiones en las que previsiblemente pueda circular mayor corriente y los T-Dean para aquellas en las que la corriente sea menor. Con todo esto, el esquema de del cableado sería:

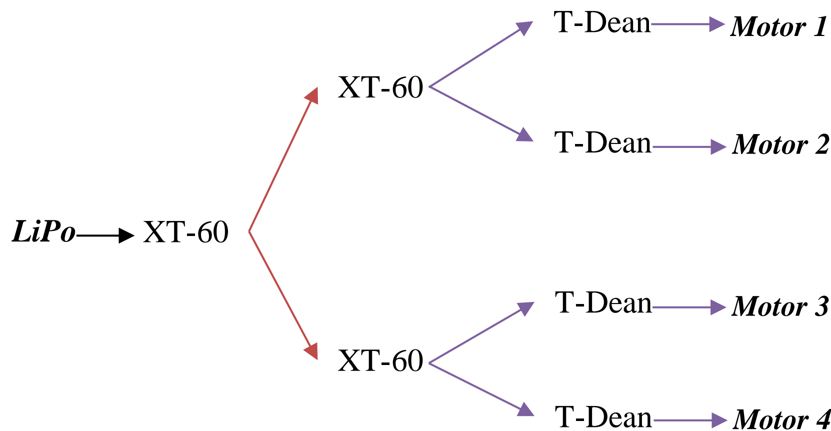


Figura 6-32 Esquema conexionado

El resultado sería el siguiente:

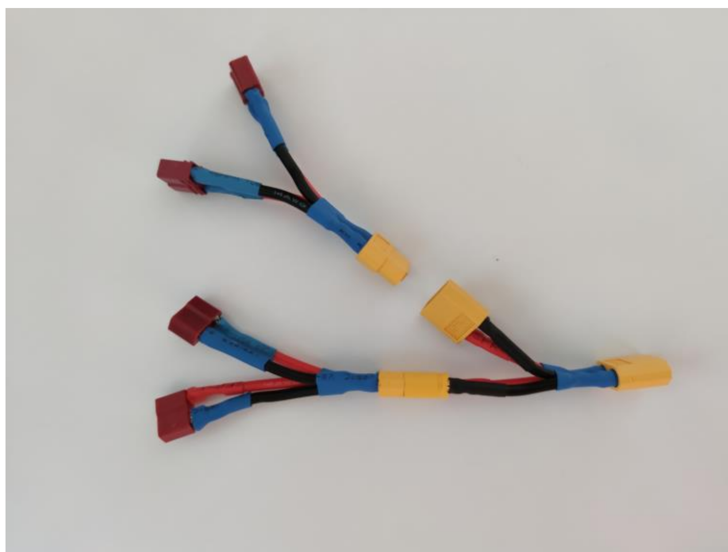


Figura 6-33 Conector motores - LiPo

6.7 PCB

PCB son las siglas "Printed Circuit Board" cuya traducción al castellano sería "Placa de Circuito Impreso". Como su propio nombre indica, es una placa (por lo general de fibra de vidrio) sobre la que se imprimen una serie de pistas de cobre con el fin de cerrar circuitos electrónicos. Además, permite la soldadura de todo tipo de componentes sobre las pistas. Es por esto por lo que es muy útil su empleo ya que:

- Asegura los componentes mediante una soldadura.
- Permite la implementación de varios sistemas electrónicos independientes dentro de la misma placa.
- Asegura una buena conexión entre los componentes de la placa.
- Reduce el efecto de la no idealidad de los cables de conexión.
- Puede diseñarse para que tenga un tamaño reducido.

Es por todo lo expuesto anteriormente por lo que se decidió emplear una PCB de diseño propio para los siguientes circuitos:

- Señales de salidas a los variadores de los motores.
- Comunicaciones con los diferentes sensores (IMU, BMP y sónar).
- Circuito de alimentación de la RPi.
- Circuitos auxiliares.

A continuación, se detallará cada circuito uno por uno.

6.7.1 Salidas a los variadores

Este conjunto de circuitos tiene la función de permitir una conexión rápida, fácil y segura de los pines de control de los variadores con la RPi. Recordemos que, para cada variador las conexiones a realizar eran dos:

- Cable negro del ESC conectado a tierra de la RPi.

- Cable blanco del ESC conectado al pin GPIO específico para el control de ese motor.

6.7.2 Comunicaciones con los diferentes sensores

La finalidad de este conjunto de circuitos es la interconexión entre los diferentes sensores y la RPi. En nuestro caso, se ha diseñado la PCB para que se puedan conectar a la RPi los siguientes dispositivos:

- Un sónar HC-SR04.
- Un sensor de presión barométrica comunicado mediante I2C.
- Una IMU empleando la conexión SPI.
- Un módulo GPS que se comunique mediante UART.

En todos los sensores menos en el sónar la conexión que se realiza es directa. En este se requiere una etapa de adaptación para no dañar la Pi.

6.7.2.1 Sónar

En este apartado no se va a explicar el funcionamiento de un sónar sino una justificación de la necesidad de emplear una etapa de adaptación.

Recordemos que el sensor empleado para este proyecto es el HC-SR04, el cual tiene los siguientes pines:

- *VCC:* alimentación del sensor a 5 V.
- *Trigger:* disparo de la señal.
- *Echo:* señal de salida del sensor. Salida a 5 V.
- *GND.*

El problema está en que la salida del sensor (pin Echo) es a 5 V mientras que los pines GPIO de la RPi son a 3.3 V. Por lo tanto, es necesario realizar un divisor de tensiones de tal manera que se asegure un valor igual o inferior a 3.3 V en el pin GPIO. El esquema sería el siguiente:

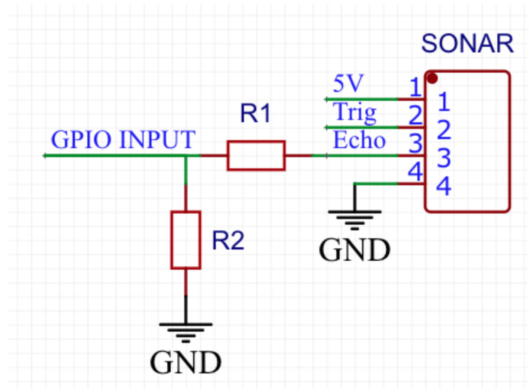


Figura 6-34 Divisor de tensión salida sónar

La tensión que recibiría el pin GPIO sería:

$$V_{GPIO} = V_{ECHO} * \left(\frac{R_2}{R_1 + R_2}\right)$$

Particularizando para nuestro caso (V_{GPIO} = 3.3 V y V_{ECHO} = 5V) quedaría:

$$3.3 = 5 * \left(\frac{R_2}{R_1 + R_2}\right) \rightarrow \frac{3.3}{5} = \frac{R_2}{R_1 + R_2} = 0.66$$

Ya sólo quedaría buscar un valor normalizado para R1 y otro para R2 que se aproxime a la relación que se ha calculado. Los valores normalizados para las resistencias son los siguientes:

x 1	x 10	x 100	x 1.000 (K)	x 10.000 (10K)	x 100.000 (100K)	x 1.000.000 (M)
1 Ω	10 Ω	100 Ω	1 KΩ	10 KΩ	100 KΩ	1 M Ω
1,2 Ω	12 Ω	120 Ω	1K2 Ω	12 KΩ	120 KΩ	1M2 Ω
1,5 Ω	15 Ω	150 Ω	1K5 Ω	15 KΩ	150 KΩ	1M5 Ω
1,8 Ω	18 Ω	180 Ω	1K8 Ω	18 KΩ	180 KΩ	1M8 Ω
2,2 Ω	22 Ω	220 Ω	2K2 Ω	22 KΩ	220 KΩ	2M2 Ω
2,7 Ω	27 Ω	270 Ω	2K7 Ω	27 KΩ	270 KΩ	2M7 Ω
3,3 Ω	33 Ω	330 Ω	3K3 Ω	33 KΩ	330 KΩ	3M3 Ω
3,9 Ω	39 Ω	390 Ω	3K9 Ω	39 KΩ	390 KΩ	3M9 Ω
4,7 Ω	47 Ω	470 Ω	4K7 Ω	47 KΩ	470 KΩ	4M7 Ω
5,1 Ω	51 Ω	510 Ω	5K1 Ω	51 KΩ	510 KΩ	5M1 Ω
5,6 Ω	56 Ω	560 Ω	5K6 Ω	56 KΩ	560 KΩ	5M6 Ω
6,8 Ω	68 Ω	680 Ω	6K8 Ω	68 KΩ	680 KΩ	6M8 Ω
8,2 Ω	82 Ω	820 Ω	8K2 Ω	82 KΩ	820 KΩ	8M2 Ω
						10M Ω

Figura 6-35 Valores normalizados de resistencias

Los valores que más se aproximan a esta relación son R1 = 0.27k y R2 = 0.47k. Estos darían una relación de 0.64.

6.7.2.2 Barómetro

Como se comentó en las especificaciones, el rango de voltajes del barómetro es de 1.8 – 3.6 V por lo que no es necesario adaptar ni las salidas ni las entradas del sensor en tensión.

Este sensor tiene 4 pines que se han de conectar a la RPi:

- *Alimentación (VCC) y tierra (GND).*
- *Pines de comunicación: Rx y Tx.*

Donde Rx y Tx son los pines de lectura y trasnmisión del bus I2C entre el sensor y la RPi respectivamente. Dada la finalidad específica de los mismos no es posible conectarlos a cualquier pin de la Raspberry, sino que ha emplearse pines que soporten dichas funcionalidades.

6.7.2.3 IMU

La única peculiaridad que posee la IMU es su protocolo de comunicación, que era SPI. Es por ello por lo que la precacución que deberemos tomar es conectar este sensor a pines que soporten implementen este tipo de comunicación. Además, se debe recordar que la alimentación se realiza a 5 V y no a 3.3 V.

6.7.2.4 GPS

Al igual que pasaba con la IMU, el GPS implementa un tipo específico de comunicación: protocolo UART. Es por ello por lo que se ha de conectar a los pines de la RPi que soporten comunicación UART. La alimentación en este caso se realiza a 3.3 V.

Aunque en este proyecto no se empleará el módulo GPS, por futuras ampliaciones se ha decidido colocar el circuito para así evitar tener que rediseñar la placa por completo.

6.7.3 Circuito de alimentación de la RPi

Uno de los aspectos más importantes de la PCB es la implementación de una serie de etapas en la alimentación de la Raspberry. Según los datos del fabricante, la RPi se ha de alimentar a 5 V y tiene un consumo máximo de 2.5 A. Esto ya plantea un conflicto pues, la única fuente de alimentación que tenemos es de 11.1 V y posee ruido debido a las conmutaciones de fase a alta frecuencia en los variadores de los motores. Para poder adaptar la fuente de alimentación de la Raspberry deberemos diseñar un circuito que conste de estas etapas:

- *Una primera etapa de regulación de tensión:* ha de proporcionar una salida estable de 5 V sea cual sea la entrada (la tensión de la batería oscila entre 11.1 V y 12.6 cuando está completamente cargada). La corriente nominal de la salida ha de ser entre 2.5 y 3 A.
- *Una segunda etapa de filtrado:* con el fin de eliminar el ruido de alta frecuencia de los motores se diseñará un filtro que proporciene una salida lo más limpia de ruido posible.

Se detallan las dos etapas a continuación.

6.7.3.1 Etapa de regulación de tensión

La finalidad de esta etapa es proporcionar una salida constante de 5 V y, al menos, 2.5 A. Para lograr ello, hay dos maneras principales de conseguirlo:

- *Módulos ya fabricados:* estos módulos ya implementan todos los elementos necesarios para la regulación de tensión. Son recomendables para aquellos casos en los que no es obligatorio el uso de una PCB. Dado que nosotros estamos realizando una PCB en la que se aglutinan diferentes subsistemas electrónicos no es adecuado el empleo de este tipo de reguladores.
- *Componentes discretos:* pequeños dispositivos electrónicos que actúan como reguladores de tensión junto con un esquema de resistencias, bobinas y condensadores (especificado por el fabricante). Están pensados para su empleo en PCBs por lo que será el método de regulación que usaremos.

En nuestro caso emplearemos el Step-Down Regulator LM2576-ADJ de Texas Instruments. Algunas de sus especificaciones son:

Tabla 6-9 Especificaciones LM2576-ADJ

Parámetro	Valor	Unidad
Máx. V_{in}	45	V
Máx. I_{out}	3	A
V_{out}	Ajustable	V
$V_{Feedback}$	1.18 - 1.28	V

En el propio datasheet del fabricante aparece un circuito general de conexión con los diferentes componentes que se han de colocar:

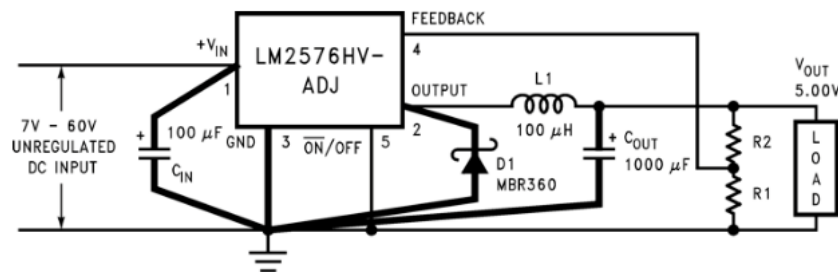


Figura 6-36 Esquema conexión LM2576-ADJ

Además del propio regulador de tensión deberemos colocar en el circuito:

- *Condensador de entrada (C_{in}):* asegura la estabilidad del flujo.
- *Bobina:* establece el modo de operación del regulador (contínuo o discontínuo). Para el caso de altas corrientes (mayores que 0.3 A) se recomienda el modo de funcionamiento contínuo.
- *Condensador de salida:* filtra la señal de salida y asegura la estabilidad de la alimentación.
- *Diodo de retorno:* provee un camino de descarga para la bobina.

- *Resistencias para feedback:* reducen la tensión de salida del sistema a valores admisibles para el feedback del dispositivo (valores expuestos en la Tabla 6-8 Especificaciones LM2576-ADJ).

Siguiendo el procedimiento descrito en el datasheet del dispositivo para calcular los valores de impedancias, los componentes a colocar en el circuito son:

Tabla 6-10 Componentes etapa regulación de tensión

Componente	Valor	Unidad
C_{in}	100	μF
C_{out}	3	μF
R1	2.7	kΩ
R2	8.2	kΩ
I_l	220-150	μH
Diodo Schottky de al menos 7 V de V_{rev}		

Es notorio comentar que este dispositivo ya proporciona un filtrado de la tensión de salida por lo que no es estrictamente necesaria la segunda etapa que vamos a desarrollar.

6.7.3.2 Etapa de filtrado

Esta etapa va a servir para eliminar el posible ruido que quedase en la salida de la etapa de regulación de tensión. ¿De dónde puede proceder este ruido? En nuestro sistema tenemos dos posibles fuentes:

- *Conmutación de los motores:* la conmutación en las fases de los motores por parte de los variadores puede introducir un ruido de alta frecuencia en la tensión de la batería.
- *Conmutación del regulador:* la regulación de tensión se realiza mediante conmutaciones del nivel de tensión. Esta conmutación puede introducir una componente de ruido en la salida.

En las dos posibles fuentes de ruido vemos que la perturbación que introducen son componentes de alta frecuencia mientras que nuestra señal deseada es de baja frecuencia (contínua), por lo que es una buena opción el empleo de un filtro paso bajo el cual permitirá el paso de la componente de contínua y bloqueará el ruido de alta frecuencia. Este tipo de filtros puede ser de dos tipos:

- *Filtro pasivo:* la etapa no aporta ninguna ganancia al sistema. En todo caso podría atenuar la salida.
- *Filtro activo:* la etapa proporciona una amplificación de la tensión de entrada.

Dado que la tensión de salida es la deseada, emplearemos un filtro paso bajo pasivo intentando que la ganancia total sea $A_v = 1$, es decir, que ni amplifique (imposible por su naturaleza) ni reduzca la tensión de salida.

El esquema de un filtro pasivo es:

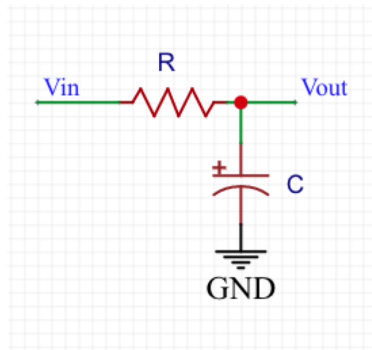


Figura 6-37 Esquema filtro paso bajo pasivo

Donde las ecuaciones que describen el comportamiento del filtro son:

$$g_{max} = \frac{R_L}{R + R_L}$$

$$w_c = \frac{1}{(R||R_L) * C}$$

Donde R_L es la resistencia de la carga, R la resistencia del filtro y C la capacidad del filtro. En nuestro caso podemos hacer la simplificación siguiente $R_L >> R$, es decir, la resistencia de entrada de la RPi es mucho mayor que la resistencia del filtro. De este modo quedarían las siguientes ecuaciones simplificadas:

$$g_{max} = \frac{R_L}{R_L} \approx 1$$

$$w_c = \frac{1}{R * C}$$

Eligiendo bien los valores de R y C podemos establecer una frecuencia de corte relativamente baja, impidiendo así que llegue el ruido a la alimentación de la RPi. Para un valor de C = 100 µF, la reducción del ruido según la resistencia sería la siguiente:

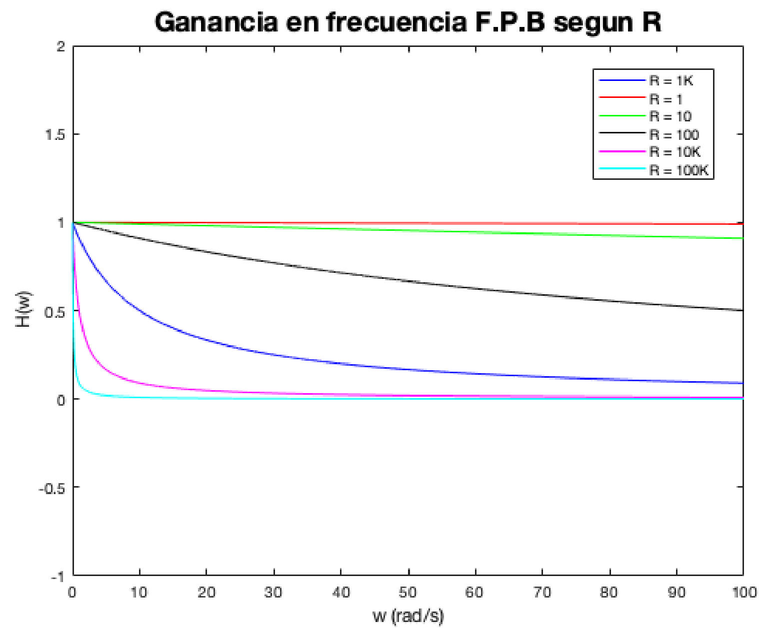


Figura 6-38 Atenuación del reuido según R

Como se aprecia, a mayor R la atenuación es mayor, es decir, la frecuencia de corte es menor. En nuestro caso elegiremos un valor de R = 1 k Ω cuya frecuencia de corte sería w = 10 rad/s ó f = 2 Hz, valor suficientemente bajo como para filtrar el ruido que tendremos en nuestro caso.

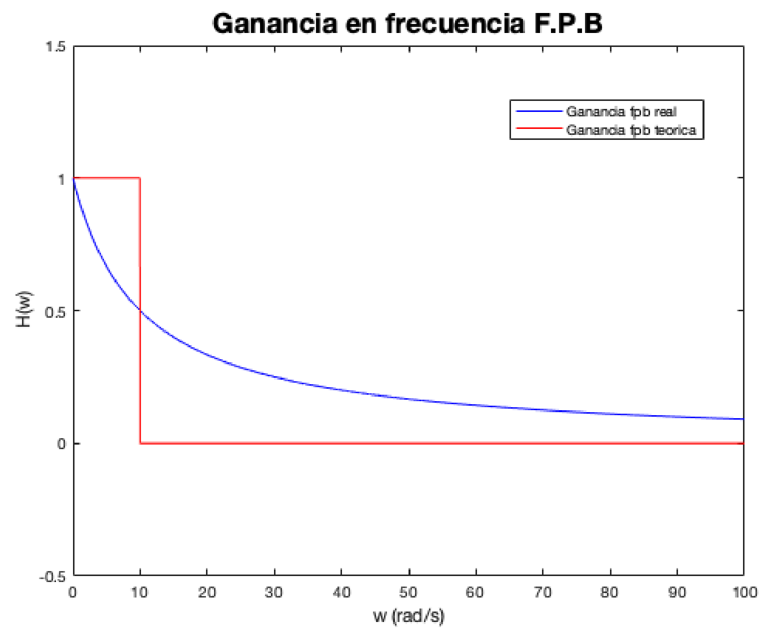


Figura 6-39 Ganancia en frecuencia para R = 1K y C = 100 µF

6.7.4 Circuitos auxiliares

Estos circuitos sirven para depuración de errores y por seguridad. Se trata de dos indicadores luminosos que advierten al usuario cuándo está alimentada la RPi y cuándo está activada la salida GPIO de la RPi y por tanto hay riesgo de que se enciendan los motores.

El esquema de los circuitos son los siguientes

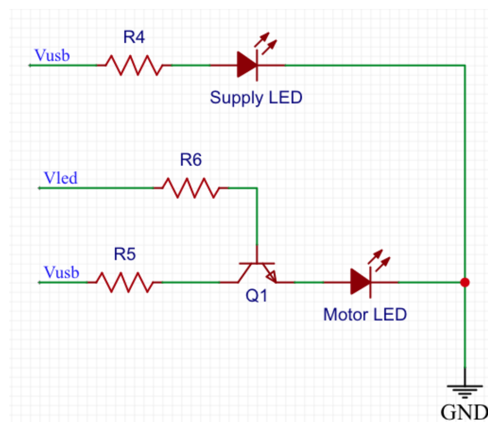


Figura 6-40 Esquema general circuitos auxiliares

Donde:

- *Vusb:* hace referencia a la tensión de alimentación de la RPi.
- *Vled:* corresponde con una salida GPIO de la RPi a través de la cual se controla el estado del transistor. Para no confundir nomenclatura, en el cálculo se nombrará como V_{GPIO}.
- *Q1*: hace referencia a un transistor NPN.

Los datos proporcionados por los fabricantes son:

Tabla 6-11 Especificaciones diodos LED

Especificación	Valor	Unidad
Caida tensión	2.4	V
I_n motor led	20	mA
I_n supply led	10	mA

Tabla 6-12 Especificaciones transistor NPN

Especificación	Valor	Unidad
Ganancia DC	200	-
V_{BE}	0.7	V
V_{CE}	0.25	V

Con estos datos podemos calcular los valores de las resistencias que necesitaremos. Para el circuito del LED de alimentación:

$$V_{usb} = V_{led} + V_{R4} \rightarrow V_{R4} = V_{usb} - V_{led} = 5 - 2.4 = 2.6\,V$$

$$V_{R4} = I_n * R_4 \rightarrow R_4 = \frac{V_{R4}}{I_n} = \frac{2.6}{0.01} = 260\,\Omega \approx 270\,\Omega$$

Para el circuito del LED de estado de los motores tenemos las siguientes ecuaciones:

$$I_{led} = (1+\beta) * I_B \rightarrow I_B = \frac{I_{led}}{1+\beta} = \frac{20}{200+1} \approx 0.1\ mA$$

$$I_c = \beta * I_B = 200 * 0.1 = 20\ mA$$

$$V_{usb} = V_{CE} + V_{R5} + V_{led} \rightarrow V_{R5} = V_{usb} - V_{led} - V_{CE} = 5 - 2.4 - 0.25 = 2.35\ V$$

$$V_{R5} = I_n * R_5 \rightarrow R_5 = \frac{V_{R5}}{I_n} = \frac{2.35}{0.02} = 117.5\ \Omega \approx 120\ \Omega$$

$$V_{GPIO} = V_{BE} + V_{R6} + V_{led} \rightarrow V_{R6} = V_{GPIO} - V_{led} - V_{BE} = 3.3 - 2.4 - 0.7 = 0.2\ V$$

$$V_{R6} = I_n * R_6 \rightarrow R_6 = \frac{V_{R6}}{I_n} = \frac{0.2}{0.01} = 2000\ \Omega\ \approx 2K2\ \Omega$$

6.7.5 Placa resultante

Se procede a mostrar la PCB diseñada atendiendo a los criterios y necesidades expuestos en los apartados anteriores:

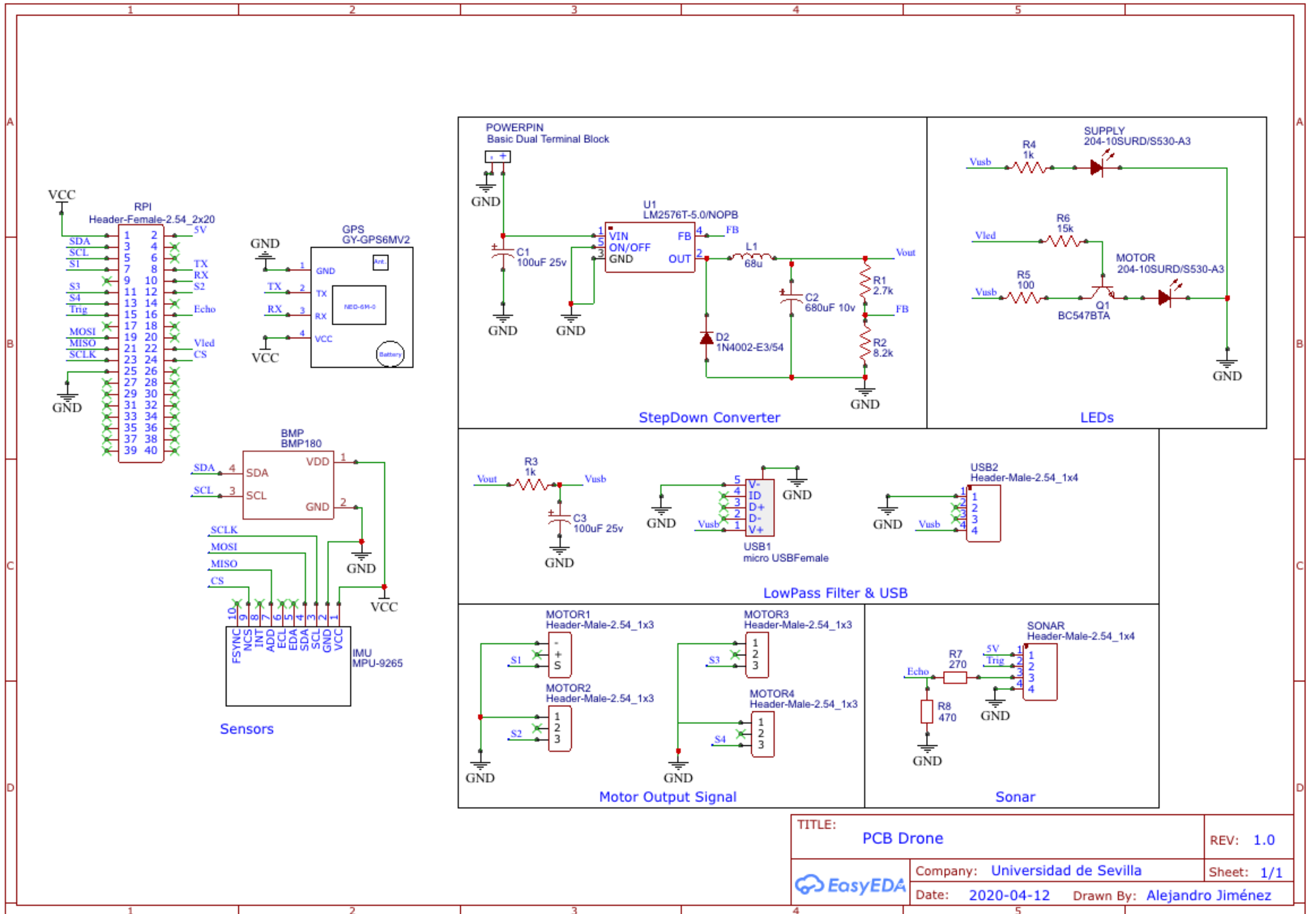


Figura 6-41 Esquemático PCB

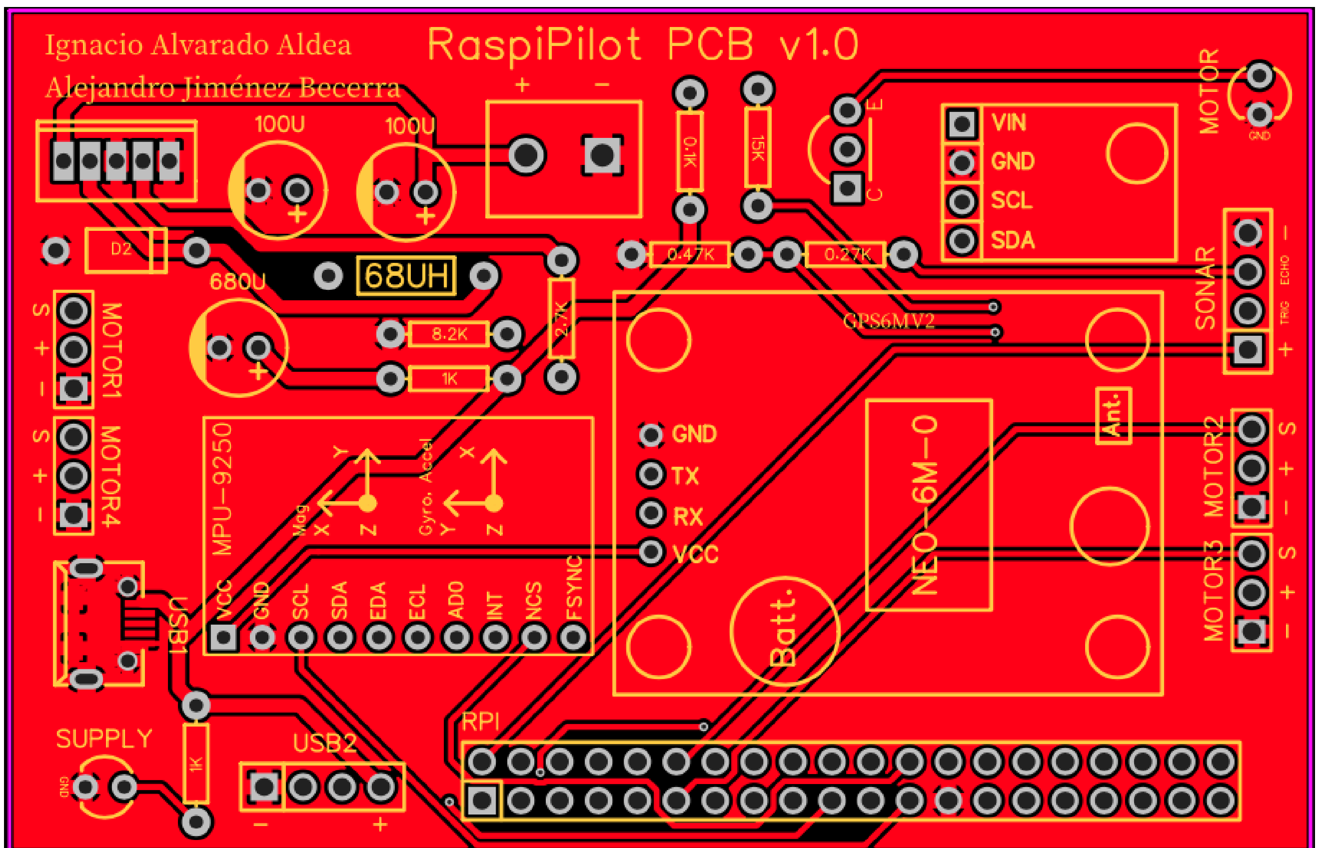


Figura 6-42 Top Layer PCB

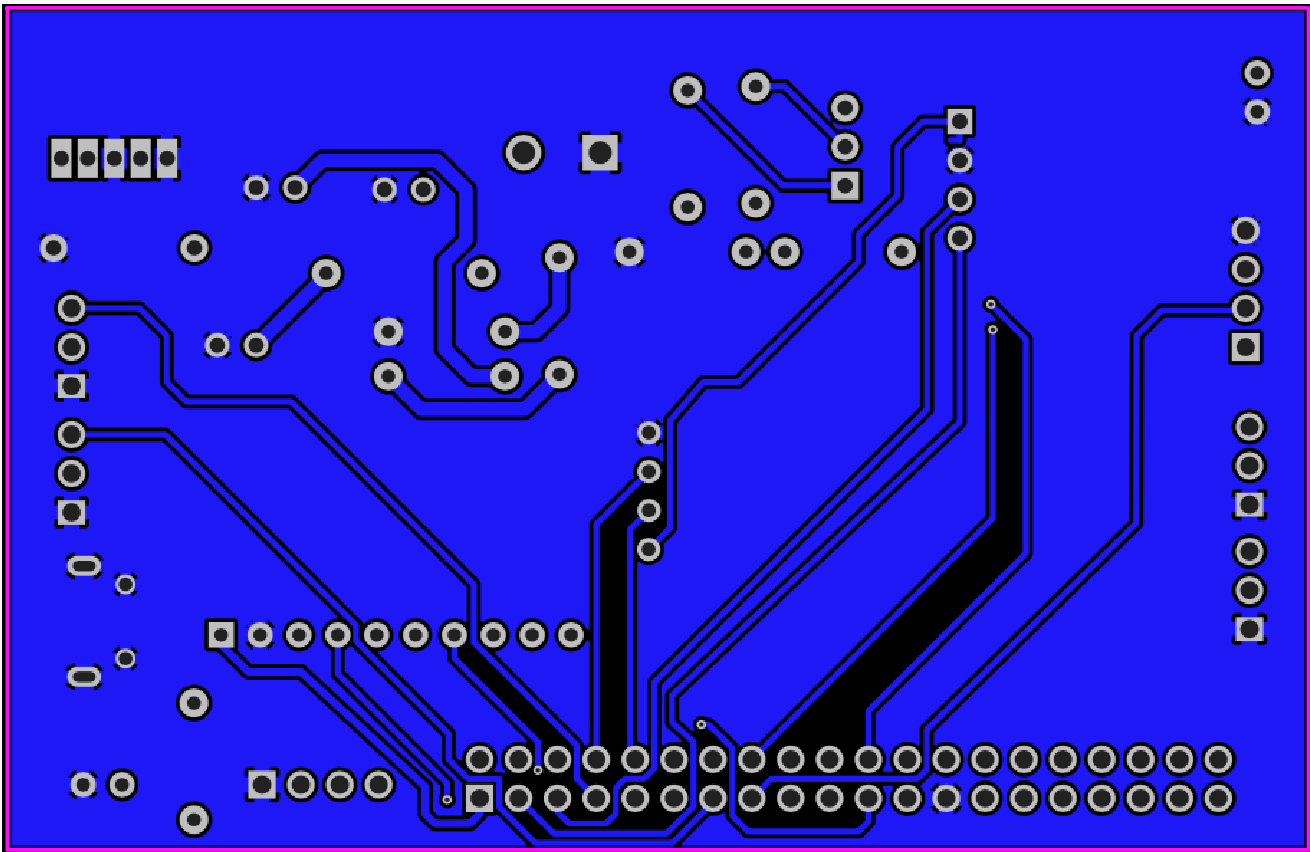


Figura 6-43 Bottom Layer PCB

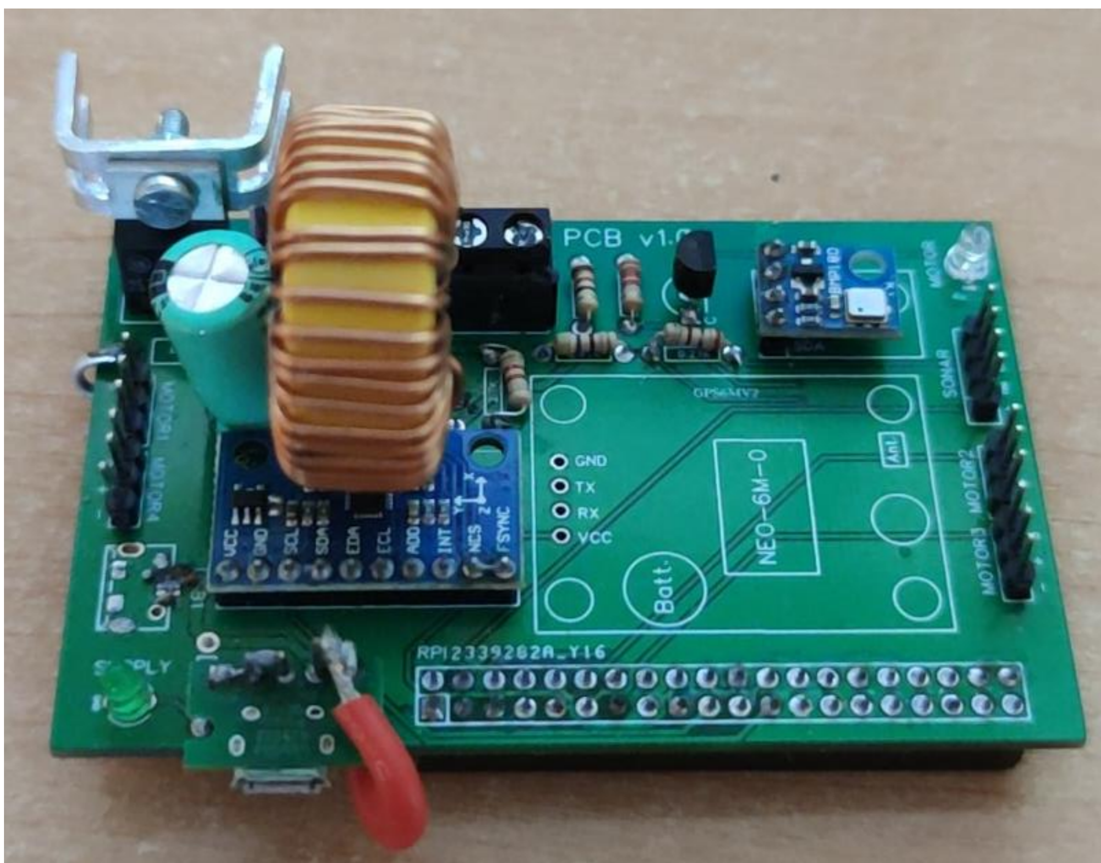


Figura 6-44 Primera versión RaspiPilot PCB

6.7.6 Errores placa diseñada

Una vez recibida la placa nos decidimos a soldar los componentes y comprobar el funcionamiento de los diferentes elementos de la placa.

En un primer lugar se pudo apreciar que el tamaño de la huella del diodo y del condensador de 680 uF no coincide con el tamaño del componente (siendo este último más grande). Es por ello por lo que se debería rectificar la huella en la PCB. Además, los pads del diodo deberán ser de un diámetro mayor también.

Otro aspecto que mejorar es la configuración de la placa con respecto a la RPi. En la versión inicial al insertarse la PCB queda en un extremo de la Raspberry mientras que para lograr un aprovechamiento mejor del espacio, así como para centrar los ejes de la IMU convendría que fiera estilo "shield", en la que la PCB quedase tapando la RPi. Esto útimo tendría un impacto sobre la refrigeración del micro, la cual se vería muy reducida. Por ello, se hace también necesario instalar un zócalo para un ventilador, así como pines para conectarlo para su alimentación.

Finalmente, experimentalmente se pudo apreciar que en el filtro paso se prodiucía una caída de tensión no contemplada inicialmente que hacía que la Raspberry no se alimentase de manera correcta. Es por ello por lo que se decidió eliminar esta etapa de la alimentación de la RPi.

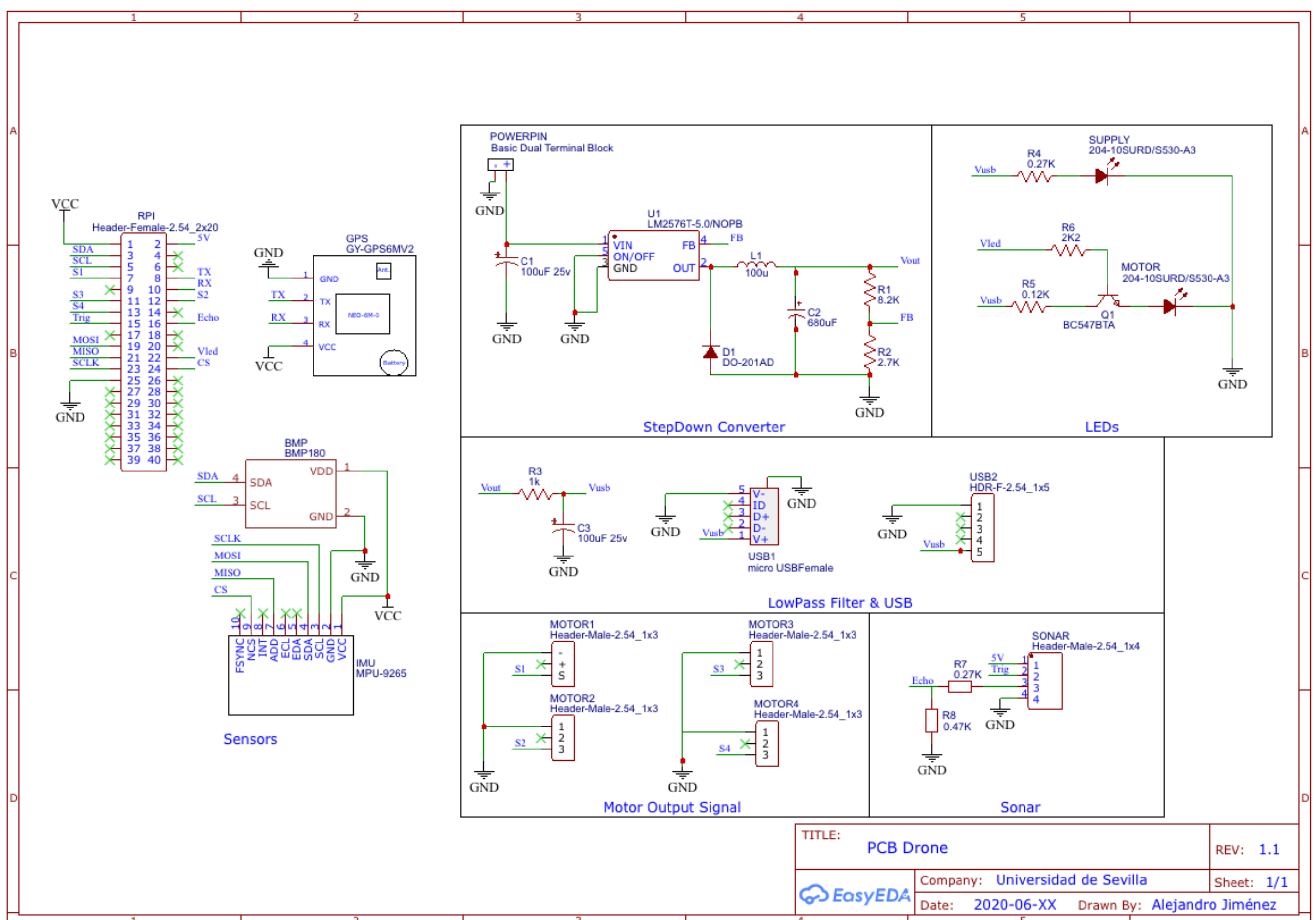


Figura 6-45 Esquemático PCB

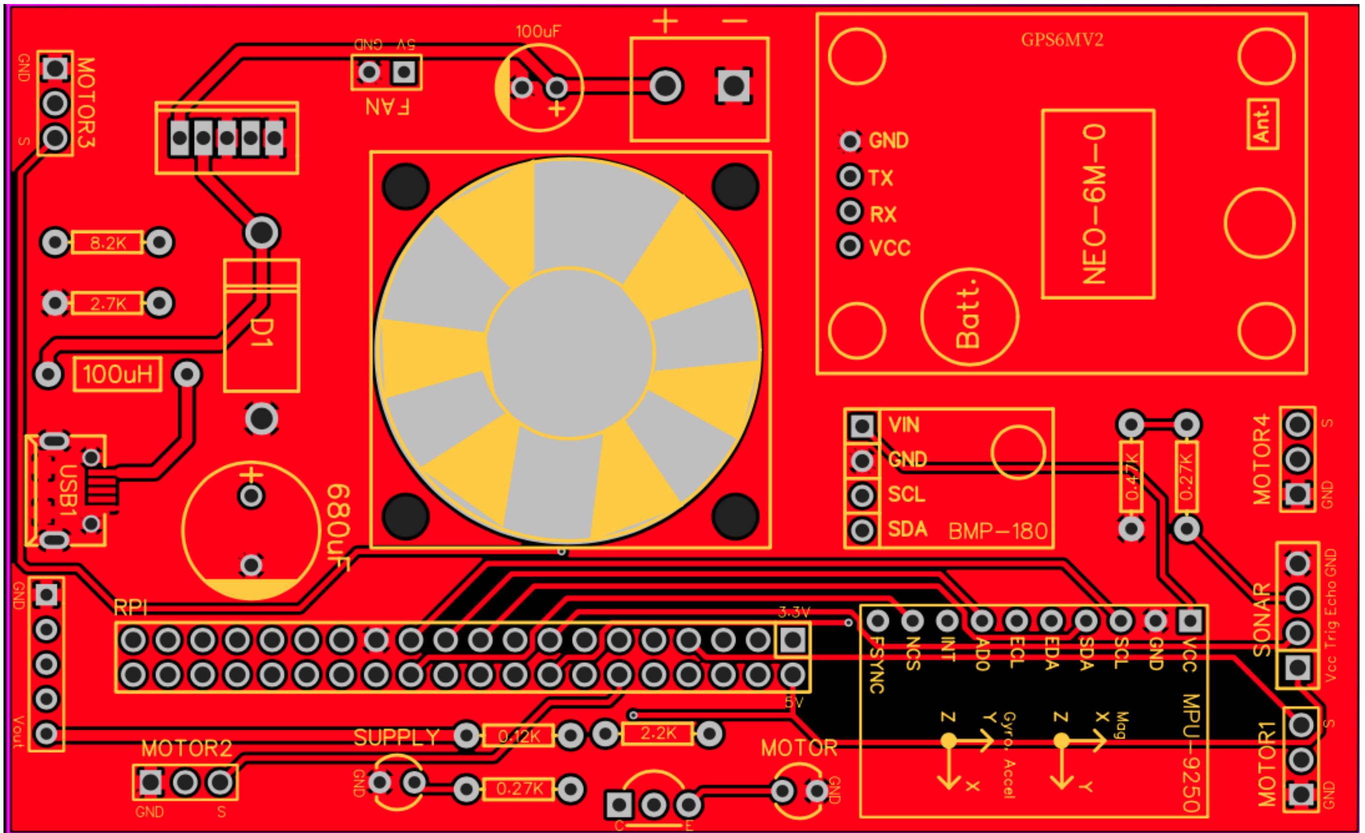


Figura 6-46 Top Layer PCB

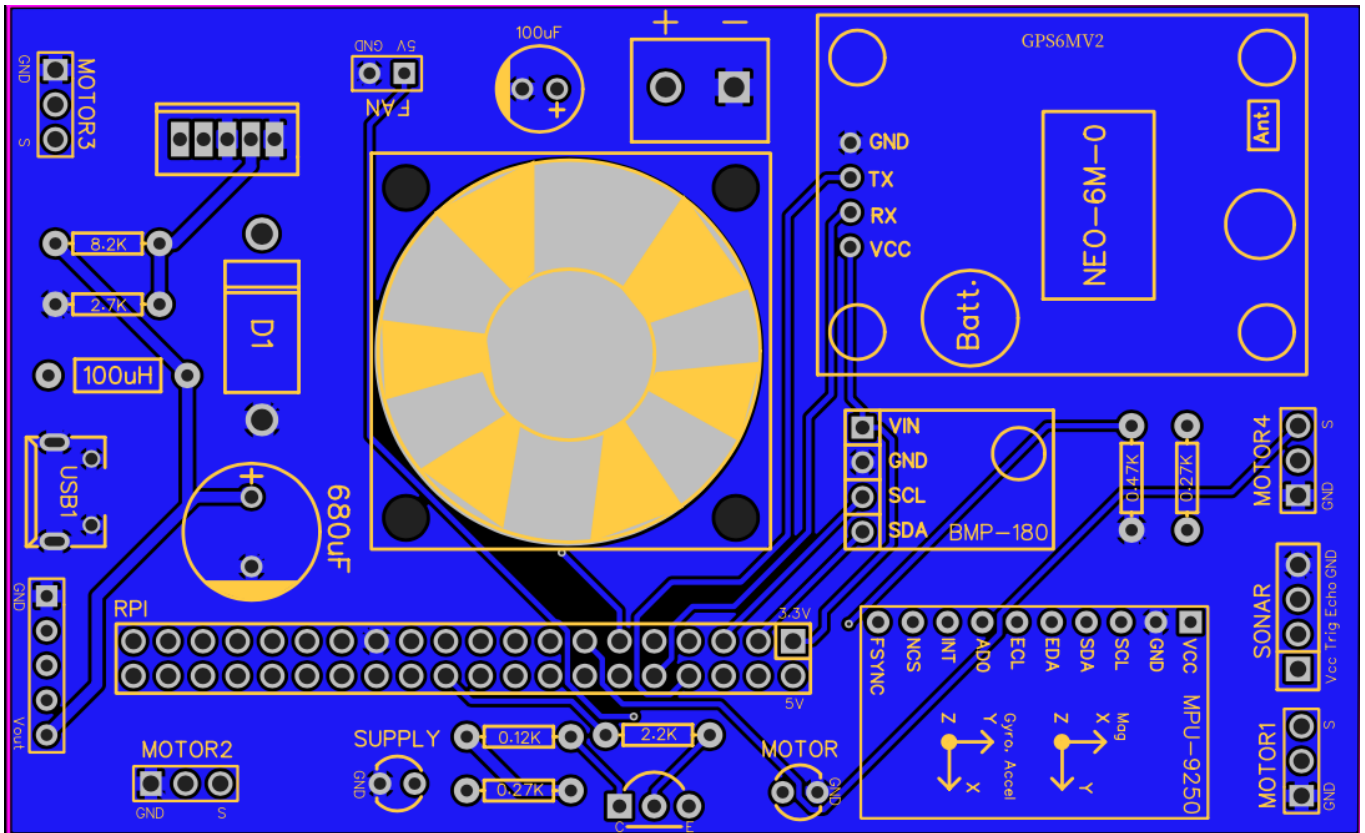


Figura 6-47 Bottom Layer PCB

7 ALGORITMO DE CONTROL, SINTONIZACIÓN PID Y COMUNICACIONES

A continuación se abordará el diseño de los algoritmos de control que correrán en la Raspberry así como la sintonización de los controladores de los motores y el protocolo empleado para las comunicacciones entre el usuario y el drone.

7.1 Algoritmo de control

Se hace evidente la necesidad de diseñar un algoritmo que corra en el microprocesador de la RPi y que sea capaz de gobernar, comunicar y sincronizar los diferentes elementos del drone con el fin de que trabajen de una manera coordinada, robusta y fiable. Este algortimo lo llamaremos algoritmo de control y está formado por diferentes tareas periódicas que se ejecutan cíclicamente y que cada una tiene una función dentro del drone. ¿Pero, por qué fragmentamos el algoritmo? Los motivos son varios:

- *Sencillez:* desde el punto de vista práctico, es más sencillo programar diferentes funciones interconectadas que una macro función encargada de controlar todos los sitemas.
- *Timing:* debido a aspectos físicos, técnicos o prácticos, no todas las funciones deberán ejecutarse cada un mismo periodo de tiempo por lo que deberemos separar aquellas cuyo ciclo es diferente.
- *Depuración:* en un código fragmentado es infinitamente más sencillo depurar y encontrar errores que en un código no dividido en pequeñas funciones.

Por estos motivos el algoritmo de control se encuentra dividido en las siguientes tareas:

- *Una tarea para la lectura de la IMU y su conversión a ángulos de Euler.*
- *Una tarea para la estimación de la altitud del drone.*
- *Una tarea para la implementación de los diferentes PIDs que controlan los motores.*
- *Una tarea para el control de los motores.*
- *Una tarea para la comunicación entre el drone y el usuario.*
- *Una función de arranque.*

A continuación, se explicarán detalladamente cada una de las tareas anteriores.

7.1.1 Función de arranque

Corresponde con la función main de cualquier script en C. Esta función tiene como finaliadad las siguientes tareas:

- *Configuración de las entradas y salidas de propósito general (pines PWM para los motores y pines para la lectura del sónar).*
- *Inicialización y comprobación de los canales de comunicación con los periféricos: SPI e I2C.*
- *Creación de un socket para la comunicación con el usuario de manera remota.*
- *Inicialización de la comunicación con los motores mediante pulsos PWM.*
- *Declaración de mútex y variables de control. Estos conceptos son explicados en el Anexo III.*
- *Declaración y ejecución del resto de tareas del drone.*

En resumen, esta función en la encargada de la configuración inicial del drone y, una vez realizada esta, de la llamada a las demás funciones necesarias para el control del sistema.

7.1.2 Tarea de lectura IMU y conversión a ángulos de Euler

Esta tarea tiene como finalidad la obtención de los datos proporcionados por la IMU y su conversión a ángulos de Euler. Es por ello por lo que esta tarea ha de ser periódica (deseamos que se ejecute cada un periodo determinado, que en nuestro caso es de 1 ms).

Es aquí donde se plantea el primer problema: el acelerómetro y giroscopio son capaces de proporcionar datos cada 1 ms pero el magnetómetro no, requiere al menos 10 ms. Este porblema se solventa de la siguiente manera: cada ciclo se leerá el accelerómetro y el giroscopio, pero sólo uno de cada diez ciclos se leerá el magnetómetro.

Los datos extraídos deberemos preprocesarlos para adaptarlos a las unidades adecuadas y después se convertirán a ángulos de Euler mediante las siguientes ecuaciones:

$$Pitch_{accel} = arc\tan\frac{accel_x}{\sqrt{{accel_y}^2 + {accel_z}^2}}$$

$$Roll_{accel} = arc\tan\frac{accel_y}{\sqrt{{accel_x}^2 + {accel_z}^2}}$$

$$Yaw_{magnet} = arc\ tan - \frac{mag_y * \cos(roll) - mag_z * \sin(roll)}{mag_x * \cos(pitch) + mag_y * \sin(roll) * \sin(pitch) + mag_z * \cos(\text{roll}) * \cos(\text{pitch})}$$

En estas medidas estamos empleando únicamente las medidas proporcionadas por acelerómetro y, como se explicó en el apartado de la IMU, estas, aunque presentan poca deriva temporal son muy ruidosas. Es por ello por lo que se le deberán aplicar a las medidas anteriores un filtro complementario:

$$Pitch = A * (Pitch_{anterior} + gyro_x + \Delta t) + (1 - A) * Pitch_{acccel}$$

$$Roll = B * \left(Roll_{anterior} + gyro_y + \Delta t\right) + (1 - B) * Roll_{acccel}$$

$$Yaw = C * (Yaw_{anterior} + gyro_z + \Delta t) + (1 - C) * Yaw_{magnet}$$

Donde A, B y C son parámetros de diseño del filtro. A mayor C se le da una mayor importancia al giroscopio con respecto a al acelerómetro, por lo que el ruido se ve reducido. Este valor suele estar alrededor de 0.98.

Experimentalmente se apreció que debido a las vibraciones producidas por los motores los valores oscilaban continuamente por lo que de cara al control de la estabilidad podría aportar muchos problemas. Es por esto por lo que se optó a aplicar un filtro paso bajo para reducir el ruido de la medida:

$$Pitch_{filtrado} = \mathrm{A} * Pitch_{\sin filtrar} + (1 - A) * Pitch_{filtrado\ anterior}$$

$$Roll_{filtrado} = \mathrm{B} * Roll_{\sin filtrar} + (1 - B) * Roll_{filtrado\ anterior}$$

$$Yaw_{filtrado} = \mathrm{C} * Yaw_{\sin filtrar} + (1 - C) * Yaw_{filtrado\ anterior}$$

Donde de nuevo, A, B y C son parámetros de diseño del filtro. Estos han de oscilar entre 0, donde el nuevo ángulo será igual al anterior, y 1, valor para el que el ángulo anterior no será tenido en cuenta. Tras ensayos prueba-error se determinó que los valores óptimos, es decir, aquellos en los que la señal estaba más limpia de ruido, eran A = B = 0.90 y C = 0.99. El resultado es el siguiente:

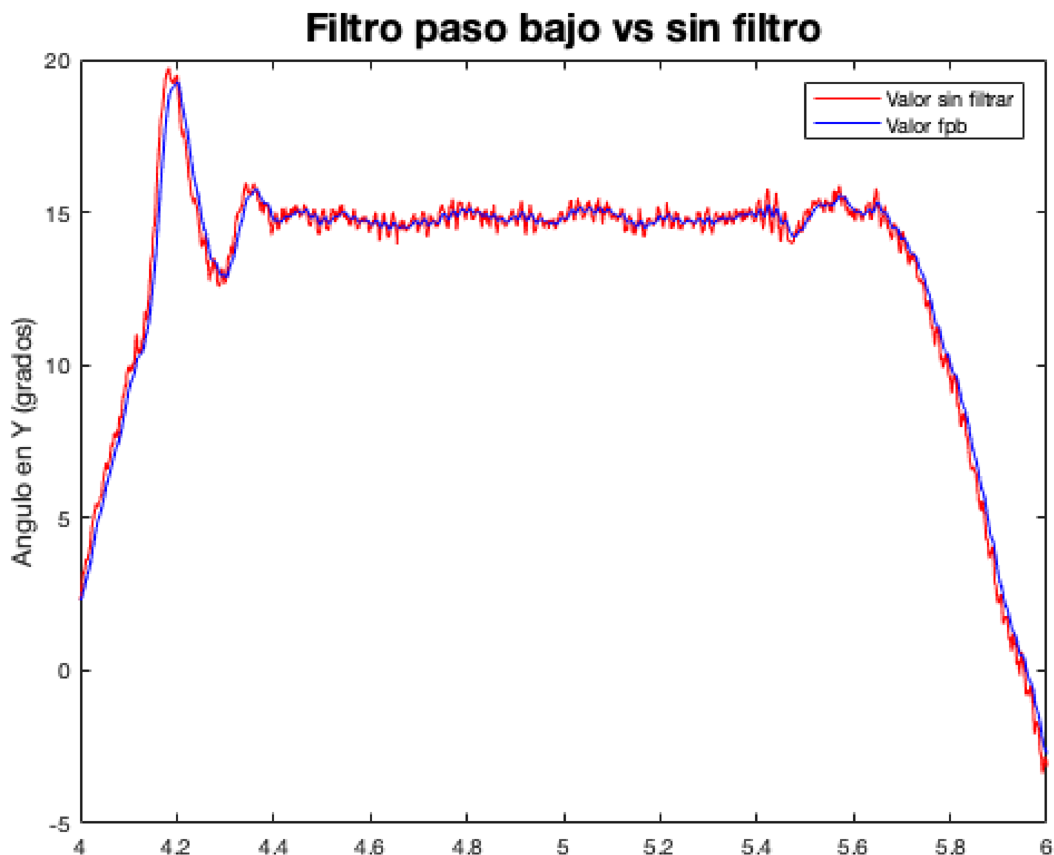


Figura 7-1 Detalle efecto filtroo paso bajo en medidas IMU

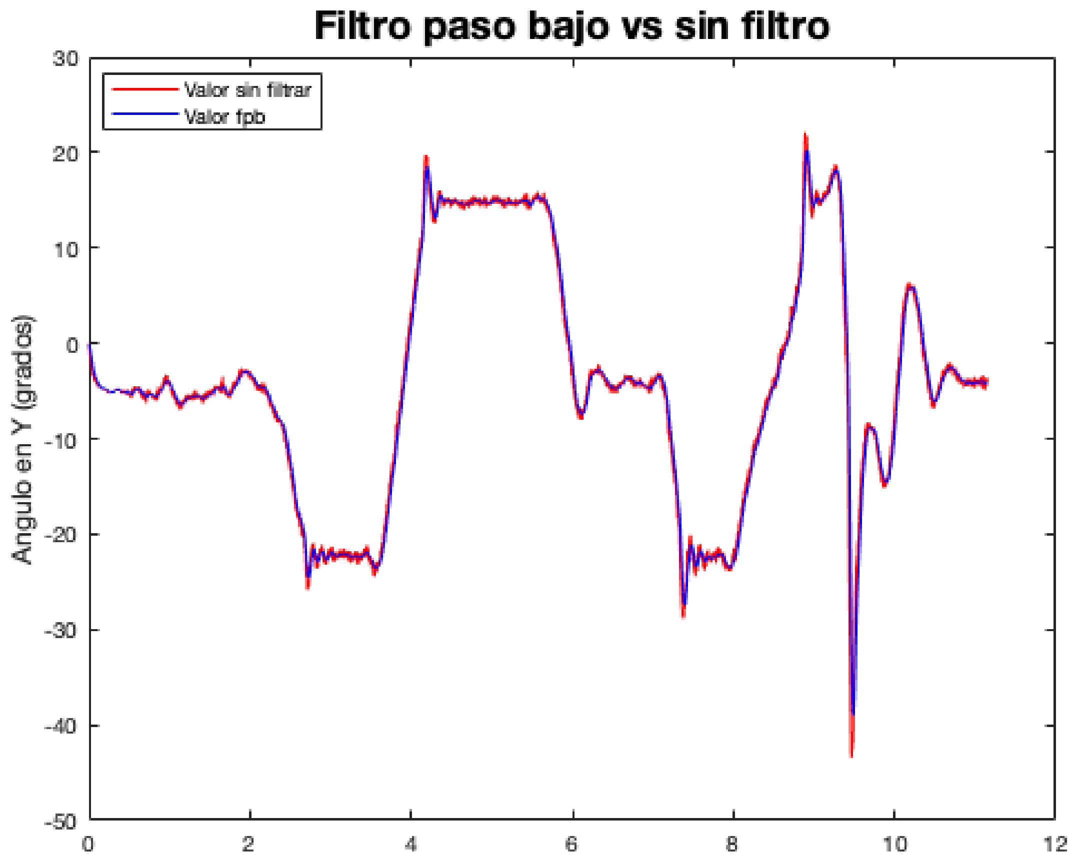


Figura 7-2 Efecto filtro paso bajo en medidas IMU

Se aprecia como el filtro complementario no es capaz de eliminiar ruido que el filtro paso bajo sí es capaz (hasta cierto nivel) a cambio de introducir un pequeño retraso en las medidas.

Tras haber obtenido los ángulos, la función la almacenará en variables compartidas entre las diferentes tareas y esperará hasta que vuelva a comenzar el ciclo.

7.1.3 Tarea de estimación de altitud

Esta tarea se encargará de combinar la lectura del sónar con la del barómetro de tal manera que se logre estimar la altitud del drone de manera rápida y precisa.

A diferencia de la tarea anterior, la dinámica del sistema permite que el periodo de ejecución de la tarea sea mucho mayor. Esto se debe a que mientras que para obtener los ángulos de Euler de la posición del drone requeríamos muchas medidas para así filtrarlas y eliminar el ruido, en la estimación de la altitud se produce menos ruido a la vez que la dinámica de cambio de altitud es más lenta. Por ello, el periodo de esta tarea se establecerá en torno a los 100 ms.

Para optimizar el tiempo de ejecución de la función debemos recordar las ventajas e inconvenientes de los dos sistemas empleados para estimar la altitud:

- *Sónar:* tiene una mayor precisión (±1 cm) y es más rápido a la hora de estimar la altitud. Sin embargo, su rango de operación es más limitado (de 5 a 300 cm).
- *Barómetro:* tiene un rango de operación muy superior al requerido. Sin embargo, es más lento (una lectura requiere en torno a 15 ms) y su precisión es mucho menos (±1 m).

Es por esto por lo que para optimizar el tiempo de CPU empleado se seguirá este esquema:

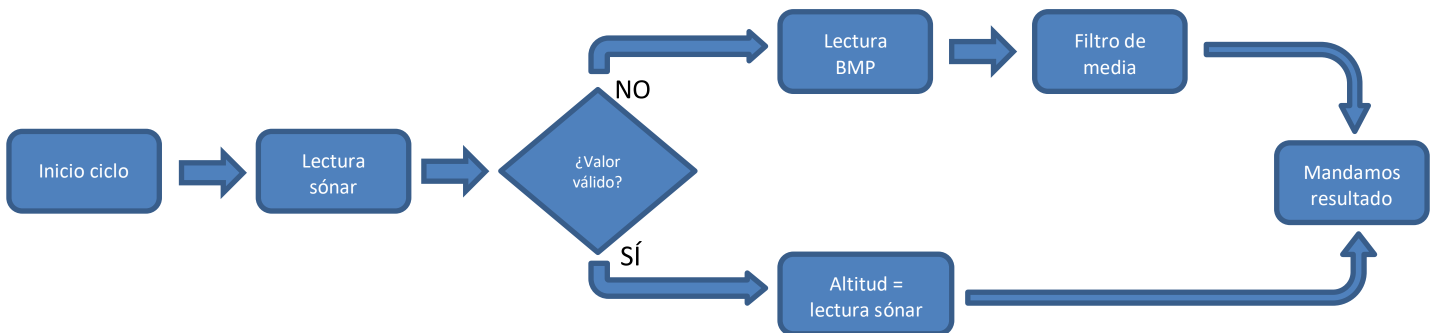


Figura 7-3 Esquema algoritmo estimación altitud

En este algortimo se intenta medir en primer lugar la altitud mediante el sónar por ser más rápido y preciso. Sin embargo, si la medida tomada está fuera de unos valores de seguridad no se puede asegurar que el resultado obtenido sea fidedigno a la realidad por lo que se procede a medir con el sensor barométrico y se le aplica un filtro de media para reducir el ruido del sensor. Tras esto comunica el valor de la altitud a las demás funciones para su empleo en otros procesos.

7.1.4 Tarea implementación controladores PID

En esta tarea a partir de los datos de entrada de los sensores y los datos procedentes del ciclo anterior, se procede a calcular las acciones de control para el control del sistema.

7.1.5 Tarea de comunicación PWM con los motores

La finalidad de esta tarea es comunicarles a los motores el régimen de giro. Es por ello por lo que el periodo de esta tarea viene determinado por la forma de comunicación entre la RPi y los motores: anchos de pulsos de periodo igual a 20 ms (más adelante se modificará este periodo).

Para llevar a cabo su finalidad, esta función accede a las acciones de control calculadas por los diferentes controladores PID y tras esto se las comunica a los motores. Finalmente, vuelve a esperar al siguiente ciclo.

7.1.6 Tarea para la comunicación entre el drone y el usuario

Esta función implementa la comunicación entre el usuario y el drone descrita en el siguiente punto. Para ello crea un socket y espera a que le envíen el mensaje de arrancar y, al recibirlo, se lo comunica a las demás tareas. De esta manera logramos un arranque seguro que nos porporcione cierta certeza de cuándo comenzarán a girar los motores.

Tras esto, esperará a que el usuario le comunique a través del socket una órden y entonces este la procesa. En caso de que la orden sea apagar, el drone realizará un cierre ordenado de las diferentes tareas que corren en ese mismo momento.

7.1.7 Comunicación entre tareas

Dada la modularidad de nuestro código, se hace evidente la necesidad de emplear algún algoritmo de comunicación seguro entre las tareas. Esto mecanismo son las variables de condición y los

mútex[7]. Mediante estos sistemas nos aseguramos de que ninguna tarea acceda a los datos sensibles (ángulos de Euler, altura, referencias…) mientras otra está leyéndolos o escribiendo en ellos y por lo tanto evitamos que se corrompan.

7.2 Comunicación entre el drone y el usuario

7.2.1 Formas de comunicación

Uno de los elementos que son imprescindibles a la hora de implementar un drone es la comunicación entre este dispositivo y el humano encargado de controlarlo. Esta puede ser implementada de diferentes maneras y todas con características diferentes. Veamos algunas opciones:

- *Bluetooth:* la comunicación mediante Bluetooth permite una implementación sencilla y sin necesidad de nuevos sensores y/o receptores pues la propia placa RPi incorpora una antena. Sin embargo, es de muy corto alcance la señal.
- *Radiofrecuencia:* este método de comunicación solventa el inconveniente anterior permitiendo la comunicación a decenas de metros. Por el contrario, complicaría el sistema de comunicación ya que se requerirían tanto un emisor como un receptor RF.
- *WiFi:* la comunicación mediante el empleo WiFi tiene las ventajas de poder implementarse de manera sencilla ya que hay diferentes librerías para abstraernos de las comunicaciones a bajo nivel y no requiere material adicional. Pese a que el alcance de la señal es menor que para la RF sigue siendo superior al Bluetooth. Es por esto por lo que esta forma de comunicación se postula como la más acorde para este proyecto.

El esquema de conexiones sería el siguiente:

Figura 7-4 Esquema conexión WiFi

Nuestro dispositivo móvil creará una red WiFi a la que se conectará el drone y a través de la cual estos compartirán información. De esta manera evitamos tener que añadir otros elementos que se encarguen de crear la red WiFi que interconexione al drone y al usuario.

El siguiente aspecto que contemplar es qué protocolo de comunicación vamos a emplear. Se podrían considerar dos diferentes:

- *TCP:* consiste en un protocolo de transmisión de mensajes con asentimiento. El dispositivo 1 envía un mensaje al dispositivo 2 y espera su mensaje de asentimiento. En

[7] Ver Anexo III.

caso de que no llegue el mensaje al dispositivo 2 o que llegue corrupto el dispositivo 1 reenviará el mensaje.
- *UDP:* es un protocolo de comunicación de mensajes sin asentimiento. El dispositivo 1 envía un mensaje al dispositivo 2 sin esperar respuesta alguna por parte de este.

La principal ventaja del protocolo TCP es la fiabilidad de la comunicación ya que el propio protocolo se encarga de asegurarse que el mensaje llegue al destinatario. Sin embargo, esto hace que los tiempos de transmisión de datos sean mayores en el TCP frente al UDP. En nuestro caso hemos elegido el protocolo UDP ya que no se consideran que los mensajes que vamos a comunicarle al drone son vitales para su integridad y, por lo tanto, reducimos tiempos de comunicaciones. Sin embargo, podría haberse empleado el protocolo TCP sin problema alguno.

7.2.2 Impementación del protocolo UDP

Una vez decidida la forma de comunicación que vamos a emplear en el drone debemos implementarla. Esto se lleva a cabo mediante sockets los cuales son interfaces de interconexión adaptables a diferentes arquitecturas de comunicaciones. Para nuestro caso deberemos:

1. *Crear un socket.*
2. *Vincularlo a una dirección IP y a un puerto.* La dirección IP será la del dispositivo móvil que al actuar de enrutador será de la siguiente forma 192.168.XX.1 mientras que el puerto puede ser el que deseemos. Para evitar conflictos con otras aplicaciones elegiremos valores altos como por ejemplo el puerto 20000.

De esta manera ya podremos comunicarnos entre el drone y el usuario de una manera bidireccional.

7.2.3 App de comunicación

El procedimiento anterior es válido para el drone, sin embrago, deberemos crear otro socket en el dispositivo Android que además deberá contar con una interfaz gráfica de tal manera que permita al usuario una interacción con el quadrotor intuitiva y sencilla. Hay dos opciones[8]:

- *Emplear una aplicación móvil ya existente.* Deberemos asegurarnos de que nos permita elegir el puerto de comunicación y que el protocolo implementado sea UDP.
- *Crearnos nuestra propia aplicación.* Esta opción permitirá crear la interfaz a nuestro gusto y personalizar los mensajes de comunicación.

En nuestro caso hemos decidido diseñar mediante el software AppInventor del MIT una aplicación que permitiese comunicarnos con el drone.

Esta consta de una pantalla con una botonera de tal manera que al pulsar un botón se comunica el mensaje por el canal (cada botón tiene un mensaje asociado para que el algoritmo de recepción del drone sea capaz de diferenciar qué botón se ha pulsado):

[8] El empleo de una opción no es incompatible con la otra si se realiza de manera adecuada.

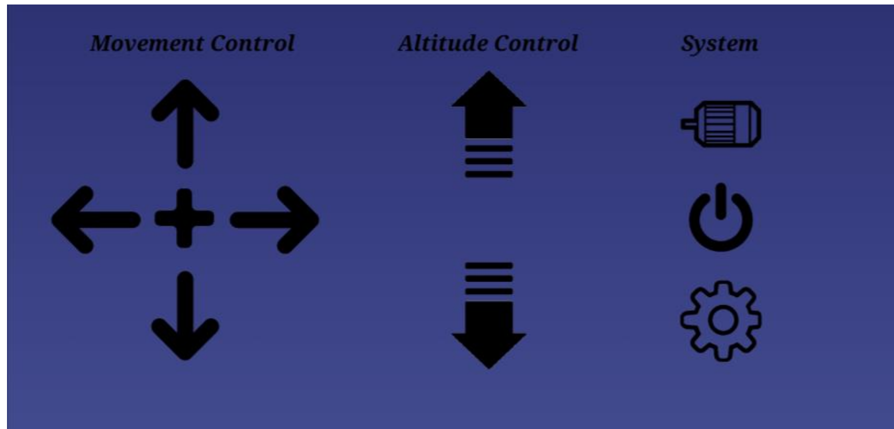


Figura 7-5 Pantalla principal app

La primera pantalla posee además de los botones de control de movimiento y de altitud otros botones que permiten arrancar los motores (figura de un motor), pararlos (símbolo de apagado) y acceder a la configuración de la app (figura de un engranaje). Al pulsar en este botón nos desplazará a la pantalla secundaria en la que podremos introducir la dirección IP y el puerto con el que comunicarnos:

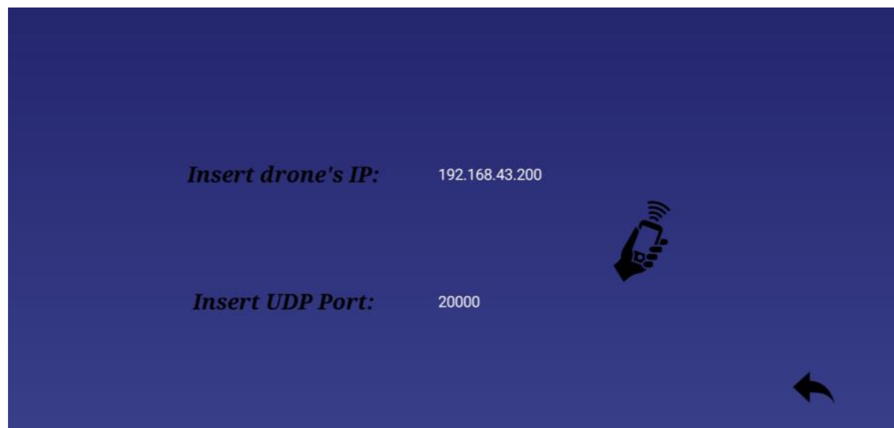


Figura 7-6 Pantalla de configuración app

Para hacer efectiva la comunicación, una vez introducida la dirección IP y el puerto UDP se ha de pulsar el botón de comunicación.

7.3 Control de los grados de libertad

Por último, vamos a detallar el procedimiento para controlar los diferentes grados de libertad del drone, así como la problemática que nos ha surgido.

En primer lugar, hay que establecer qué tipo de controlador se ha de sintonizar. Existen distintos tipos, pero nosotros hemos optado por uno de los más sencillos si no el que más, el PID. Este tipo de controlador es una composición de tres acciones:

- *Acción proporcional P:* genera una salida en función del error del sistema, a mayor error mayor será la acción generada.
- *Acción integral I:* genera una salida atendiendo al error acumulado en los ciclos anteriores. Este término es el encargado de eliminar el error en régimen permanente del sistema.

- *Acción derivativa D:* genera la acción de control atendiendo a la evolución temporal del error, es decir, según la derivada temporal de este. Es el término encargado de corregir inestabilidades del sistema ocasionadas por los términos P e I.

Estas acciones se suman para generar una única acción de control cuya ecuación es:

$$u_k = K_p * [e_t + K_i * T * I_k + \frac{K_d}{T} * (e_k - e_{k-1})]$$

Donde Kp, Ki y Kd son las constantes que nosotros deberemos sintonizar y T es el periodo de muestreo.

Un esquema visual es:

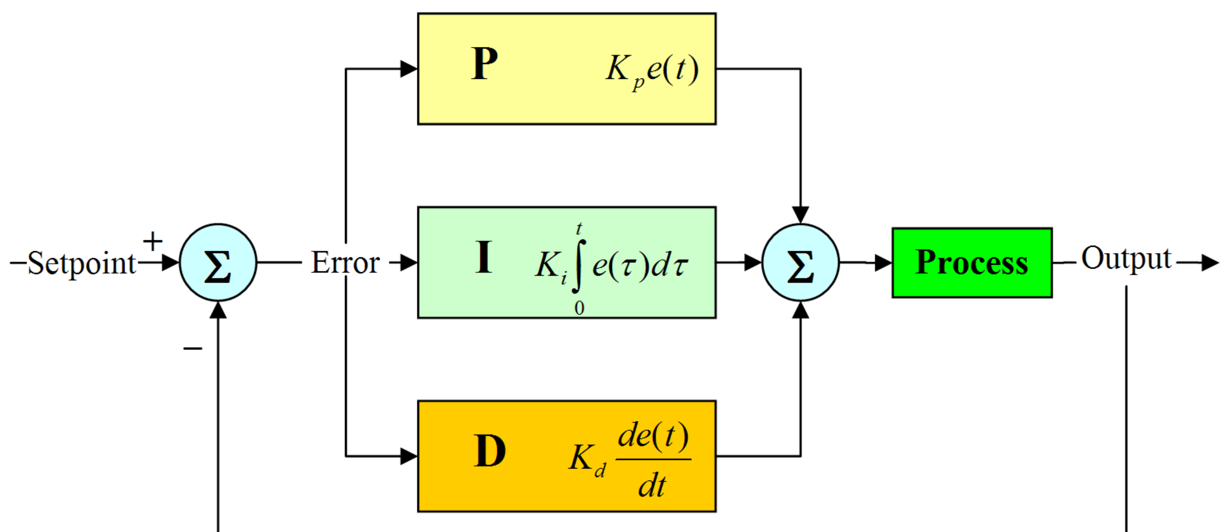


Figura 7-7 Esquema general PID

El motivo de la elección de este controlador es que para intentar sintonizarlo no es obligatorio obtener el modelo matemático del sistema por lo que se simplifica el proceso de sintonización.

Antes de comenzar con la sintonización deberemos modificar unos aspectos del drone para facilitar el control, el soporte de la batería y el tiempo de muestreo.

Inicialmente era un cajón impreso en 3D, pero si se sustituye este por una placa de aluminio estaremos desplazando verticalmente el centro de gravedad hacia abajo y por lo tanto, propiciaremos que el drone se comporte como un péndulo facilitando su control. Esta placa que hemos colocado tiene unas dimensiones de 90x145x5 mm y un peso aproximado de 180 gramos.

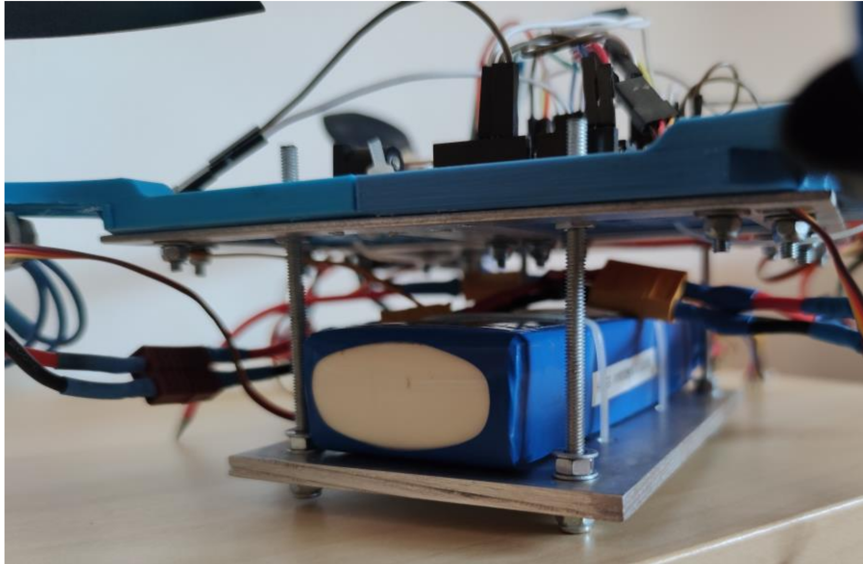


Figura 7-8 Nuevo soporte batería

En cuanto al tiempo de muestreo hago referencia al periodo en el que se manda una señal a los ESC. Inicialmente este era de 20 ms pues el fabricante del ESC indicaba que ese era el periodo cada cuanto deberíamos enviar una señal a los variadores. Sin embargo, en la práctica vimos que a altas potencia de los motores el drone se desestabilizaba debido al gran periodo de muestreo, por lo que tuvimos que reducirlo a la mitad. Con ello pudimos mejorar un poco el control.

Una vez realizadas estas pequeñas modificaciónes vamos a intentar controlar el sistema. Para ello iremos probando diferentes valores de las constantes y veremos cómo se comporta el sistema para esos valores. El mejor comportamiento que obtuvimos con la RPi fue:

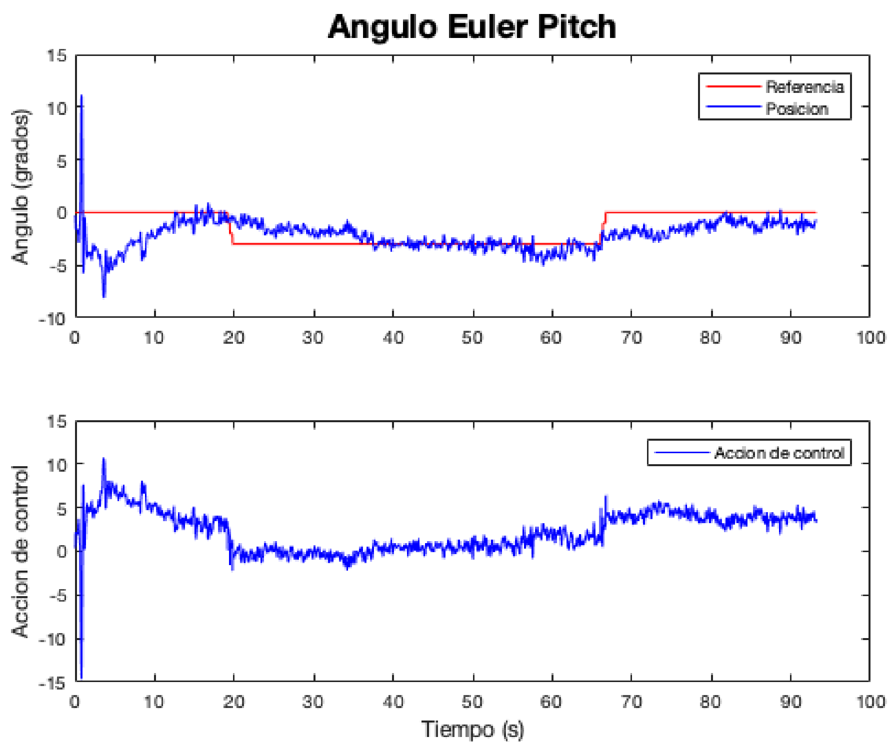


Figura 7-9 Gráficas control pitch

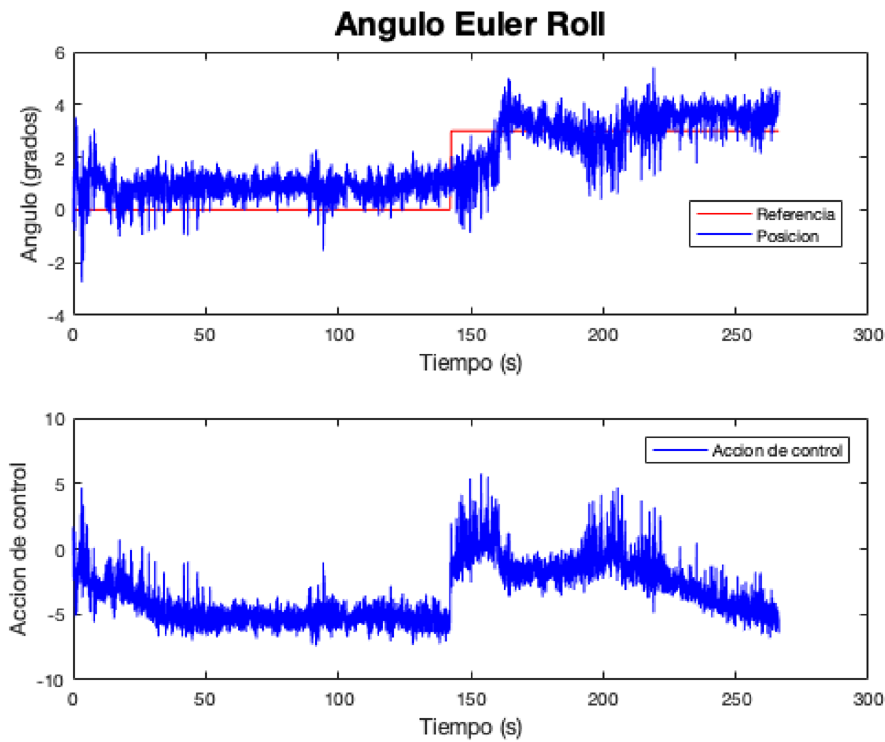


Figura 7-10 Gráficas control roll

Como se puede apreciar en la imagen se obtienen comportamientos muy lentos y exageradamente ruidosos. Aún así nos aventuramos a sacar al drone del soporte e intentar que se mantuviese estático a unos centímetros del suelo, pero esto nunca ocurrió. Dado el ruido del sistema este se volvía loco y nunca controlaba siendo un sistema totalmente inestable.

Se intentó con multitud de controladores distintos pero los resultados eran siempre los mismos: teníamos un sistema altamente inestable. ¿Era razonable lo que nos estaba ocurriendo? ¿El fallo era entero nuestro? Esto hizo que redirigiéramos los esfuerzos al por qué no controlaba en vez de a cómo controlarlo.

Tras una búsqueda por internet de drones basados en RPi nos percatamos de que había un factor común en todos ellos: entre la RPi y los motores no había comunicación directa.

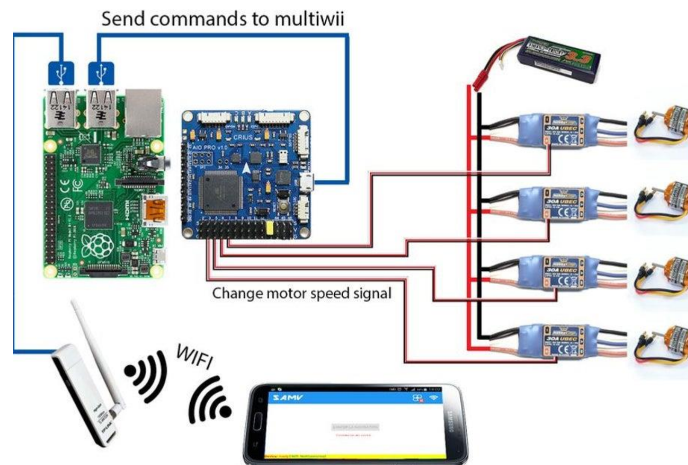


Figura 7-11 Esquema comercial RPi

En la mayoría de los casos la Raspberry no es el componente que realiza los cálculos e interactúa con los sensores y actuadores, sino que es el nexo entre la controladora de vuelo y el usuario, pero… ¿Por qué?

Todo se debe a la comunicación con los motores. Como se explicó en un apartado anterior, esta se realiza mediante modulación de ancho de pulso y esta se puede hacer de 3 maneras disitintas:

- *Mediante hardware:* es una circuitería interna quien genera las señales. En la Raspberry sólo hay dos canales asociados a los pines 12, 32, 33 y 35. Estos pines tienen una resolución de 1us.
- *Mediante software:* en caso de requerir más de dos señales PWM podremos generarlas mediante software a través de librerías. Estas señales tienen una precisión de 100 us la cual empeora a mayor carga computacional de la RPi.
- *Mediante DMA:* es generada por el periférico DMA. Esta es un paso intermedio entre la modulación hardware y software. Es mucho más precisa que la generada mediante librerías, pero no llega a la resolución de las señales PWM hardware. Está disponible en cualquier pin GPIO de la Pi.

En nuestro script empleábamos la librería pigpio la cual emplea moodulación de ancho de pulso mediante DMA (explicada arriba) en aquellos pines que no es posible la modulación hardware (casi todos los de la RPi). Esta señal recordemos que no tiene tanta precisión como una señal PWM generada por hardware. Además, esta señal se hace más imprecisa conforme requiramos mayor uso de la CPU, en procesos multihilo y si hay un tráfico de datos a los USB y las SD (cumplimos todos los requisitos para que el sistema no genere buenas señales). Por estos motivos, la señal que mandábamos a los variadores para controlar los motores no eran exactamente del ancho de pulso que nuestros controladores calculaban para intentar estabilizar el sistema y por tanto este nunca llegaba a estabilizarse.

Aparentemente este resulta ser el por qué de nuestro problema. Una manera sencilla de comprobarlo sería medir con un osciloscopio las señales PWM generadas por la Raspberry y comprobarla con la teórica. Dado que no podemos acceder a los laboratorios no será posible realizar dicha comprobación por lo que asumiremos que este es el fallo. La siguiente pregunta que nos planteamos es, ¿hay soluciones o alternativas? La respuesta es sí, claro que hay alternativas y muy variopintas:

- *Cambio de controladora:* dado que la Raspberry no asegura que la señal PWM generada sea exactamente la deseada, una opción es cambiar a otra placa que sí la asegure como por el ejemplo el Arduino. Este microcontrolador genera suficientes PWM por hardware para controlar los motores, pero veremos mermada nuestra capacidad de cálculo.
- *Empleo de controladora intermedia:* seguir el esquema de la figura 7-11 y emplear una controladora de tal manera que la RPi no sea quien gobierne los motores. Algunos ejemplos de controladoras de vuelo son PXFmini o Multiwii.

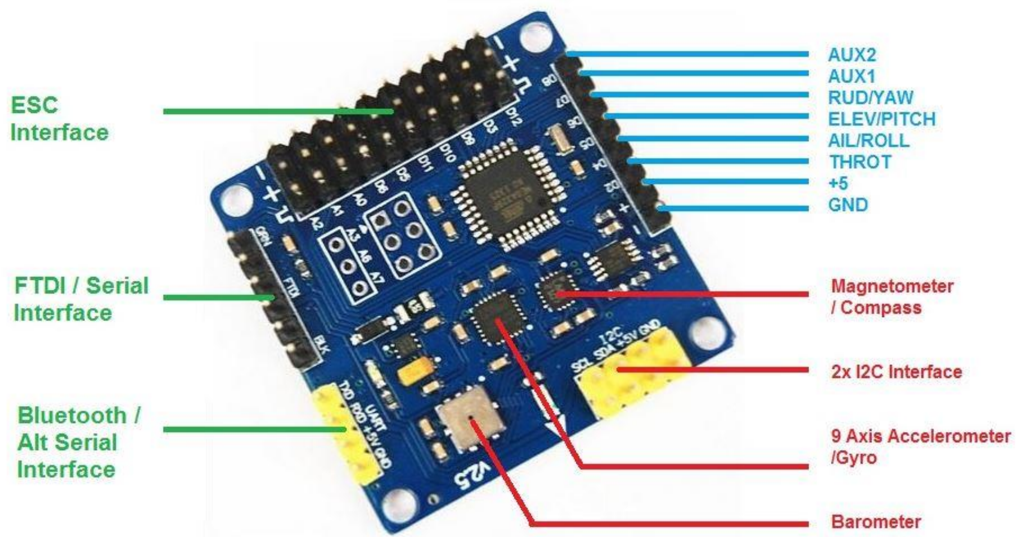


Figura 7-13 Controladora de vuelo PXFmini

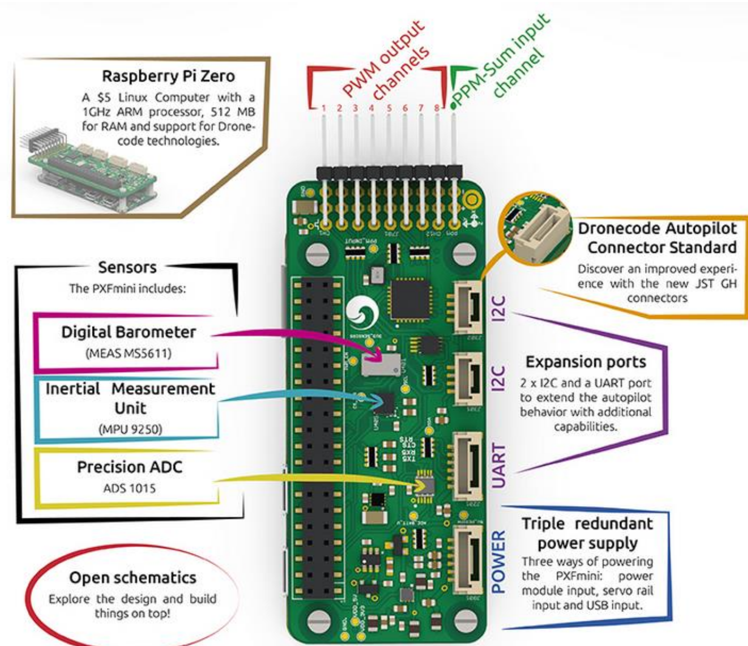


Figura 7-12 Controladora de vuelo Multiwii

- *Empleo de generadora de PWM:* emplear una placa que, conectada la RPi, reciba mediante I2C o SPI el ancho de la señal y esta lo genere. Por ejemplo, la placa Adafruit PCA9685.

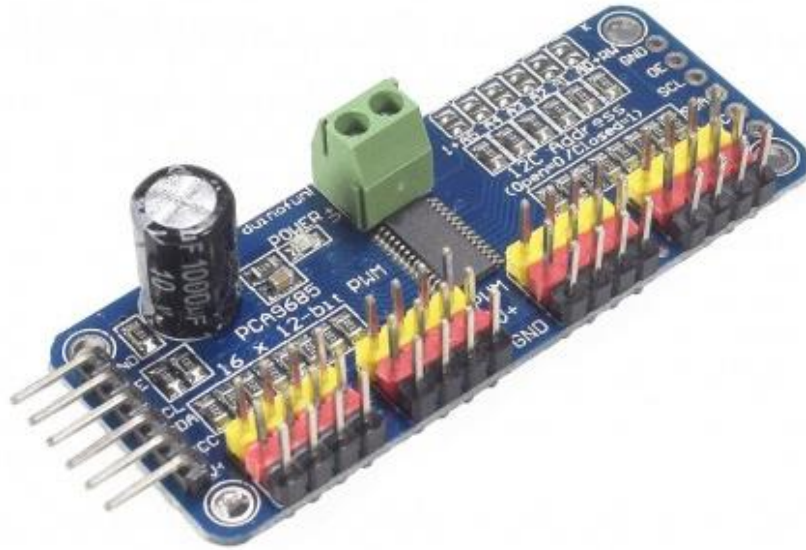


Figura 7-14 Placa Adafruit PCA968

8 SOLUCIÓN ALTERNATIVA

Como se vio en el apartado anterior, el empleo en exclusiva de la Raspberry no es viable para el diseño de una controladora de vuelo para un quadrotor. Por este motivo se mostaron diferentes opciones para solventar el problema. De todas las opciones dadas aquella que más nos convence es emplear un Arduino, en concreto un Arduino Uno. Esto se debe a que poseemos conocimientos previos sobre Arduino y a la gran cantidad de documentación existente en Internet acerca del tema.

Las especificaciones del modelo que vamos a emplear son las siguientes:

- Microcontrolador ATmega328P.
- 2 KB de memoria SRAM.
- 1 KB de memoria EEPROM y 32 KB de FLASH.
- 3 Timers de dos canales cada uno.
- 16 pines digitales a 5 V, de los cuales 6 implementan PWM por hardware.
- 5 entradas analógicas.
- Soporte para comunicaciones SPI, I2C y UART.
- Sin conexiones inalámbricas de manera nativa.

Figura 8-1 Placa Arduino Uno

Con tal de simplificar la solución, optaremos por que sea directamente el propio Arduino quien controle los motores y ejecute el software de control en vez de emplearlo como un elemento intermedio entre la RPi y los variadores. Sin embargo, hay que tener en cuenta algunos aspectos a la hora de realizar la migración del código de RPi a Arduino:

- *Capcidad de cálculo.* El ATmega328 es un procesador mucho más limitado que el BCM2837 por lo que se deberá buscar la simplicidad del código para así lograr una mayor eficiencia.
- *Imposibilidad de hilo.:* El ATmega328 no permite la concurrencia mediante hilos. Sin embargo, podremos emplear interrupciones temporales o externas de una manera sencilla para el control del flujo del programa.
- *No hay comunicaciones inalámbricas:* Arduino Uno no implementa ningún tipo de comunicación inalámbrica, por lo que deberemos recurrir a módulos que habiliten este tipo de comuniaciones.

Por estos motivos se tuvieron que hacer una serie de modificaciones al código de la Pi mientras se adaptaba para el lenguaje Arduino:

- *Supresión de los hilos.* Ya nuestro sistema de control no estaría compuesto por hilos con una función específica cada uno. Ahora sería un conjunto de tareas consecutivas cuya duración total debe ser menor al periodo de muestreo. El inicio de la primera tarea será activado por una interrupción en el micro.
- *Eliminación del control de altitud.* el control de altitud requiere mucho tiempo de la CPU por lo que se decidió eliminar tanto el sónar como el barómetro.
- *Cambio de medio de comunicaciones.* La forma de comunicarnos con la RPi era mediante un WiFi hotspot creado por nuestro propio celular. Ahora esto ya no es posible dado que el Arduino no implementa de forma nativa comunicaciones WiFi y tampoco disponíamos de un módulo receptor. Es por ello por lo que se decidió a cambiar las comunicaciones a Bluetooth pues sí poseíamos antenas de este tipo (módulo HC-06).

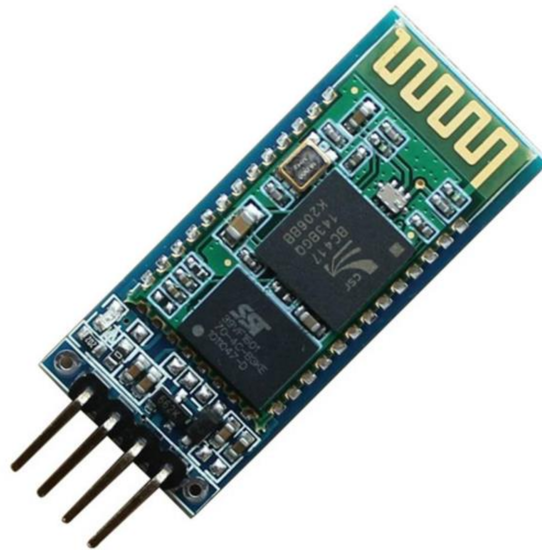


Figura 8-2 Módulo Bluetooth HC-06

Por todo lo anterior, el diagrama de flujo de nuestro algoritmo sería el siguiente:

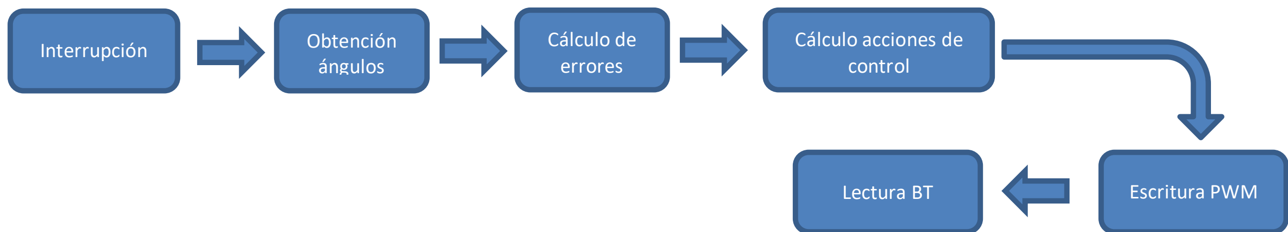


Figura 8-3 Diagrama de flujo algoritmo Arduino

La interrupción y la obtención del ángulo puede hacerse de diferentes maneras.

8.1 Interrupción externa del DMP

La primera opción consiste en emplear el "Digital Motion Processor" o DMP. Este es un algoritmo que contiene la IMU en el cual, mediante la aplicación de diferentes filtros, es el sensor quien calcula los ángulos de Euler liberando así al microprocesador de esta tarea.

El empleo de este sistema posee tanto ventajas como inconvenientes. Veamos en primer lugar los aspectos positivos de usar el DMP:

- *Libera al ATmega de cálculos tediosos.* Dado que es la propia IMU quien realiza los cálculos, liberamos de esta tarea al Arduino con la consiguiente mejora en el rendimiento (recordemos que el poder computacional del Arduino es limitado, y más operando con números flotantes).
- *Gran precisión.* El algoritmo empleado por el DMP implementa diversos filtros a los valores obtenidos de tal manera que, los valores de los ángulos poseen una gran precisión.
- *Interrupción externa asociada.* Cada dato nuevo que posee el DMP este genera una interrupción mediante el pin INT de la IMU. Si este pin se asocia a uno del Arduino y habilitamos las interrupciones de dicho pin ya tendríamos la interrupción que da comienzo al periodo de muestreo.

Algunos de los inconvenientes de emplear el DMP son:

- *Tiempos de muestreo definidos.* El tiempo mínimo de generación de un dato nuevo es de 6 ms. Si queremos un tiempo mayor deberemos aplicar preescaladores de reloj, por lo que no podremos elegir el periodo que nosotros deseemos, sino que deberemos ajustarnos a los valores posibles.
- *Inutilidad del magnetómetro.* Aunque existen librerías que implementan el DMP con los 9 grados de libertad de nuestro sensor, estas presentan errores por lo que nos vemos obligados a emplear una librería para una IMU de 6 DOF. Esta librería estima el ángulo Z como la integración temporal de las velocidades angulares en dicho eje.

Conociendo estas limitaciones procedemos a probar nuestro sistema de control. Una aclaración que debo realizar es que se ha empleado un periodo de muestreo de 10 ms para que de tiempo a la ejecución de todas las tareas antes de que vuelva a ocurrir otra interrupción.

A la hora de intentar controlar el ángulo en X y el ángulo en Y obtenemos los siguientes resultados:

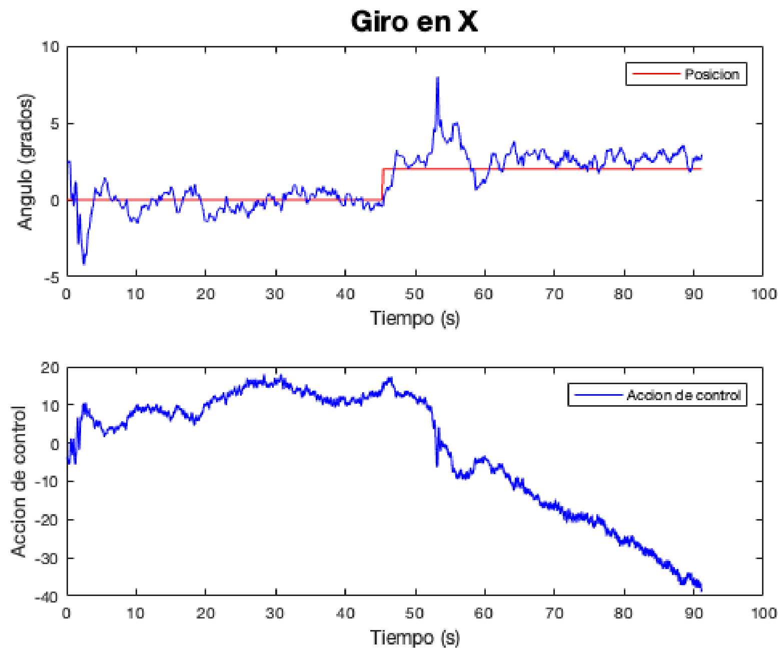


Figura 8-4 Control eje X

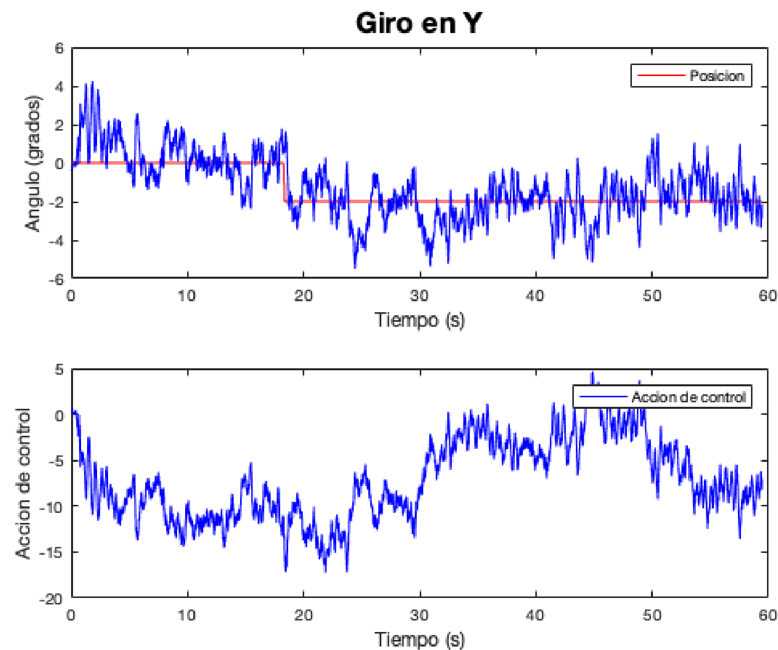


Figura 8-5 Control eje Y

Como podemos apreciar, los resultados que obtenemos no son muy finos. Aunque logramos que el sistema funcione en torno a la referencia y que si hay un cambio de esta el drone reaccione de manera rápida, la salida sigue siendo muy ruidosa, hecho que puede inestabilizar el sistema.

Si probamos el control de los dos ejes a la vez obtenemos las siguientes salidas:

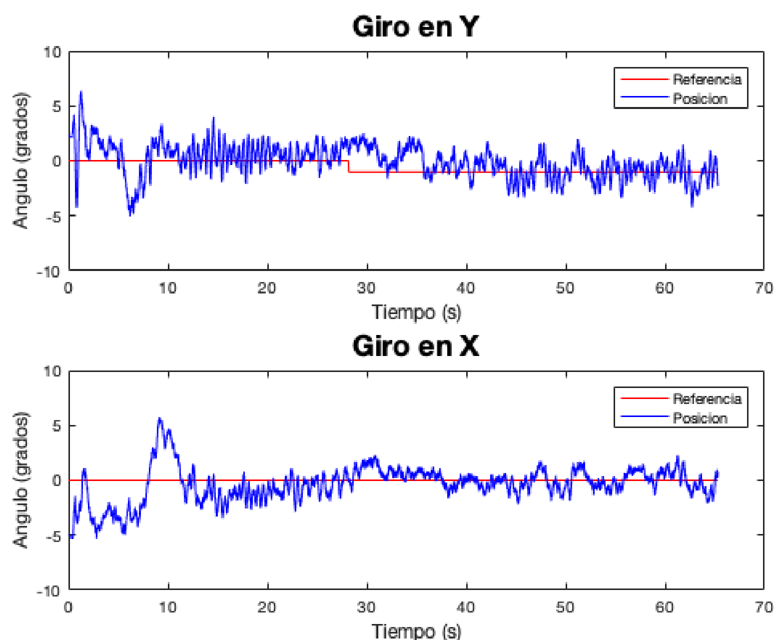


Figura 8-6 Control ejes X e Y

El comportamiento ruidoso en el eje Y resulta un gran problema para la estabilidad del drone. Pruebas realizadas sin soporte ni fijación alguno del quadrotor resultaron en un auténtico fracaso por dicho ruido. Por motivos de tiempo, el desarrollo de este TFG acaba con estos resultados. Aunque no son resultados satisfactorios, se ha podido justificar la imposibilidad de controlar un drone con la placa comercial Raspberry Pi 3B y se han propuesto posibles soluciones para solventar la problemática que esta lleva aparejada.

9 ANEXO I

A continuación se detarallá el cálculo de las estimaciones de las fuerzas que soportará el quadrotor en el caso más desfavorable.

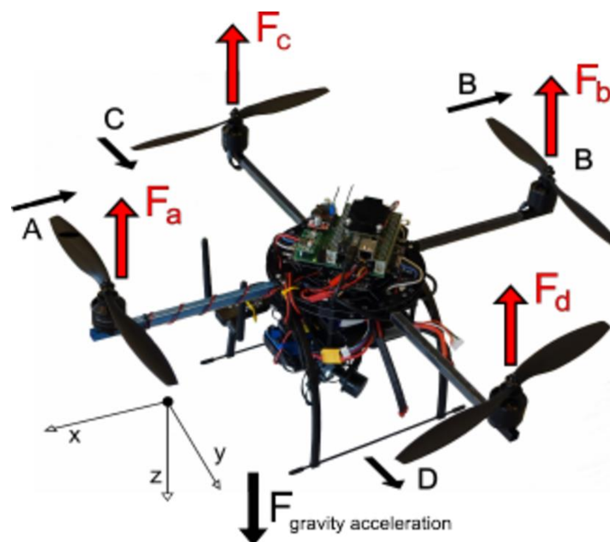


Figura 9-1 Fuerzas sobre un quadrotor.

Como ya se ha indicado previamente, la situación más desfavorable es aquella en el que todos los motores están ejerciendo el máximo empuje, es decir, aquella situación en la que los todos motores ejercen su máximo empuje en la dirección del eje Z y en el sentido opuesto al peso. Esto se traduce matemáticamente en la siguiente expresión:

$$F_{NETA} = Empuje - Peso$$

El empuje que ejerce cada motor es de 0.8 kp por lo que el empuje total que experimenta el quadrotor será de 3.2 kp. A continuación, deberemos realizar una estimación del peso del drone. Para ello, enumeraremos los distintos componentes de este y estimaremos el peso que este pueda tener:

- *Estructura + fijador:* 0.24 kg. Esta estimación te la facilita el software de diseño 3D.
- *Motor + variador:* cada conjunto motor-variador posee una masa aproximada de 0.1kg. Por lo que el total de los motores y variadores serán 0.4kg.
- *Batería:* la masa aproximada de la batería es de 0.4kg.
- *Raspberry Pi:* la Raspberry posee una masa de 0.05kg.
- *Componentes electrónicos + cableado eléctrico:* este es el conjunto más difícil de estimar su masa ya que está compuesto por muchos elementos. Estimaremos la masa total del conjunto en 0.15 kg.

La masa total estimada del quadrotor sería 1.24 kg que equivaldría a un peso de 1.24 kp. De esta manera, la fuerza neta que experimentaría el quadrotor completo sería F_{NETA} = 1.96 kp = 19.21 N.

Para facilitar el análisis nosotros analizaremos los efectos sobre un solo brazo del drone. Esto se debe a que, al analizar un solo componente requerimos menor coste computacional y los resultados

son los mismos que obtendríamos si analizásemos el sistema por completo ya que el quadrotor es totalmente simétrico. Por lo tanto, la fuerza que experimentará cada brazo será 4.80 N.

10 ANEXO II

El siguiente anexo contiene los pasos a seguir para instalar Xenomai y configurar los diferentes periféricos de la Raspberry a fin de que esté lista para usarse en la aplicación que a nosotros nos concierne.

Aunque puedan encontrarse multitud de versiones de Raspbian por internet en las que ya viene instalado Xenomai, esto no resulta adecuado en algunos casos (como en el nuestro) ya que puedes encontrar problemas a la hora de emplear algunos periféricos de la RPi (como el bus I2C). Por este motivo, se recomienda seguir esta guía de instalación.

Existen dos maneras de instalar Xenomai y preparar el RTOS. La primera es una instalación completa y paso a paso en la que podremos elegir la configuración que deseamos. Por el contrario, tenemos la opción de una instalación rápida a partir de una imagen del sistema. Se procede a explicar ambas opciones.

10.1 Instalación completa

A continuación, se explicará la instalación completa paso a paso.

10.1.1 Instalación de Raspbian

En primer lugar, deberemos instalar el sistema Raspbian. Para ello, descargaremos la imagen[9] desde la web oficial. Una vez tengamos descargada la imagen, se deberá grabar la imagen en una tarjeta microSD[10]. Para ello, nos ayudaremos de alguna herramienta de software como puede ser BalenaEtcher:

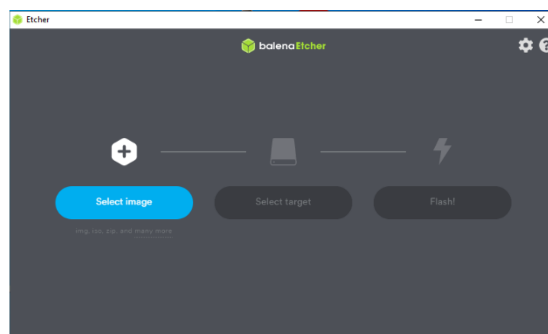


Figura 10-1 Pantalla usuario BalenaEtcher

Cuando se haya completado la instalación, se extrae la microSD, se conecta en la RPi y la arrancamos. Al arrancarla nos pedirá que realicemos una configuración inicial, al terminar esta ya estará montado el sistema Raspbian en la tarjeta.

Ahora deberemos actualizar el software pues puede que haya algún paquete del sistema que esté obsoleto. Para ello deberemos configurar la red WiFi y cuando tengamos acceso a internet

[9] Se recomienda descargar la versión sin software recomendado.
[10] Se recomienda que la capacidad sea de al menos 8 GB.

teclearemos en el terminal el siguiente comando: "sudo apt-get upgrade & sudo apt-get update". Al acabar de actualizar los paquetes, ya tendremos el SO totalmente operativo.

10.1.2 Instalación de Xenomai

En primer lugar, deberemos descargar el kernel. Para ello, tecleamos en el terminal:

```
sudo apt-get install subversion
svn checkout https://github.com/thanhtam-h/rpi23-4.9.80-xeno3/trunk/prebuilt
```

Tras esto, abrimos la carpeta en la que se ha descargado el kernel y lo instalamos:

```
cd prebuilt
chmod +x deploy.sh
./deploy.sh
```

Una vez se haya instalado, deberemos fijar las cabeceras de Linux:

```
cd /usr/src/linux-headers-4.9.80-v7-xeno3+/
 sudo make -i modules_prepare
```

En último lugar, deberemos configurar la CPU de nuestra RPi para que todos los núcleos corran Xenomai. Para ello teclearemos:

```
sudo nano /boot/cmdline.txt
```

Esto nos abrirá un editor de texto. En este fichero que se nos ha abierto deberemos añadir al final la siguiente línea:

```
isolcpus=0,1,2,3 xenomai.supported_cpus=0xF
```

Finalmente, para que se apliquen todos los cambios deberemos reiniciar el sistema:

```
Sudo reboot now
```

Una vez que se reinicie la Raspberry ya tendremos instalado el RTOS Xenomai. Si se desea, se puede comprobar que esto haya sido así mediante los siguientes comandos:

```
sudo /usr/xenomai/bin/latency
uname -a
```

La primera instrucción se encarga de medir la latencia del sistema. Dado que ahora estamos trabajando en un RTOS sin entorno gráfico, esta ha de ser muy pequeña:

Figura 10-2 Latencia Xenomai

Mientras que la segunda indica qué sistema operativo estamos corriendo. Si todo ha salido bien debería ser una versión xeno.

Figura 10-3 Sistema operativo actual

10.1.3 Configuración perifericos

En último lugar, deberemos configurar los periféricos para que podamos emplearlos correctamente. En concreto, habilitaremos las comunicaciones I2C, SPI y SSH. Como ya se explicó en otros apartados, tanto las comunicaciones SPI como I2C se emplearán para comunicarnos con los diferentes sensores del sistema. Sin embargo, la comunicación SSH la emplearemos para poder programar el software que correrá en el drone en nuestro PC y pasarlo a la RPi de una manera sencilla empleando algún entorno de programación.

Para habilitar las conexiones por SPI e I2C teclearemos en el terminal:

```
sudo raspi-config
```

Se abrirá una pestaña y seleccionaremos: Interfaces > I2C > Accept, SPI > Accept y SSH > Accept > Finish. Tras esto nos pedirá que reiniciemos la Raspberry. Una vez que haya arrancado, ya tendremos habilitadas las tres interfaces de comunicación. Ya sólo faltaría instalar los drivers y librerías necesarias para poder emplear los periféricos desde un script programado en C. Tecleamos en el terminal:

```
sudo apt-get install i2c-tools
sudo apt-get install pigpio python-pigpio python3-pigpio
sudo reboot now
```

Por último, deberíamos configurar la frecuencia de reloj del bus I2C (la estableceremos en 400kbits/s). Para ello tecleamos en el terminal:

```
sudo nano /boot/config.txt
```

Se nos abrirá un fichero de texto, buscaremos y cambiaremos lo siguiente:

```
i2c_arm_baudrate = <> -> i2c_arm_baudrate = 400000
```

¡Ya estaría la RPi lista para emplearse!

10.2 Instalación desde imagen

Dadas las dificultades encontradas para poder instalar Xenomai en la RPi y que funcionen todos los periféricos, he creado una imagen que contiene el RTOS Xenomai con la configuración[11] de los periféricos. Para instalar esta versión, bastará con emplear un programa que introduzca en una tarjeta microSD la imagen (p.e. BalenaEtcher). A diferencia del método anterior, en el que se podía usar una tarjeta de cualquier tamaño, en este caso ha de ser de 16 GB obligatoriamente.

[11] Esta versión no contiene ninguna librería instalada por lo que el usuario debería ser quien escogiese cuál usar y la instalase manualmente.

11 ANEXO III

En el siguiente anexo se procederá a explicar las funciones básicas del núcleo de Alchemy que nos servirán para implementar nuestro sistema en tiempo real.

En primer lugar, veamos algunos de los elementos que suelen componer un sistema de tiempo real:

- *Hilos:* son recursos informáticos que permiten la ejecución de manera concurrente de diferentes tareas.
- *Mútex:* son elementos informáticos que aseguran la integridad de las variables compartidas entre diferentes funciones. Para ello, impiden que ninguna función acceda a la variable si otra ya la está usando.
- *Variables de condición:* son un mecanismo de sincronización entre los diferentes procesos o tareas del sistema. Sirven para notificar que se ha producido un evento, como sería la disponibilidad de un dato, por ejemplo. Estas están asociadas a los mútex.

11.1 Hilos

Estos recursos serán los empleados para implementar las diferentes tareas que necesitemos llevar a cabo en nuestro sistema. Estas tareas pueden ser de dos maneras:

- *Periódicas:* se repiten cada un periodo T hasta un determinado evento.
- *De ejecución única:* sólo se ejecuta la tarea una vez.

En nuestro caso, queremos que las tareas que debe desempeñar la CPU del drone se ejecuten indefinidamente, por lo que optaremos por tareas periódicas. El procedimiento de creación de un hilo periódico es el siguiente:

- *Creación de la tarea:* emplearemos la función rt_task_create ().
- *Inicialización de la tarea:* mediante la función rt_task_start ().
- *Esperar a la función (opcional):* se empleará rt_task_join ().

Al crear la tarea, esta se asociará con una función del tipo: *void name (void *arg).* Es imperativo que la función tenga la estructura de entrada y salidas de argumentos anterior o en caso contrario no se implementará de manera adecuada el hilo. Para que esta sea periódica deberemos:

- *Indicar que se trata de una tarea periódica:* emplearemos la función rt_task_set_periodic (). Esto se colocará antes del bucle principal (el cuál se ejecutará periódicamente) de la función.
- *Esperar al siguiente ciclo:* mediante la función rt_task_wait_period (). Deberá ser colocado al final del bucle para que, cuando se haya ejecutado este la función libere la CPU hasta que empiece otro ciclo.

Las funciones anteriores son las más básicas y sencillas de usar. Sin embargo, se pueden realizar más operaciones con los hilos como eliminarlos desde otros hilos, suspenderlos, reanudarlos…

11.2 Mútex

Para proteger la integridad de las variables compartidas emplearemos un mútex. Para crear un mútex se han de seguir los siguientes pasos:

- *Creación del mutex:* emplearemos la función rt_mutex_create ().
- *Eliminación del mútex:* mediante la función rt_mutex_delete ().

Una vez tenemos el mútex creado, deberemos acceder a él para operar con las variables compartidas y, una vez se haya acabado con estas, se debe liberar para que otra función pueda acceder al mútex. Esto se implementa de la siguiente manera:

- *Acceso al mútex:* puede accederse de dos maneras principales
    - *Sin límite de tiempo:* la función espera indefinidamente al mútex. Se emplea la función *rt_mutex_acquire ().*
    - *Con límite de tiempo:* si la función no accede en un determinado tiempo, se produce un error por timeout. Se emplea la función *rt_mutex_acquire_timed ().*
- *Liberación del mútex:* mediante la función rt_mutex_release ().

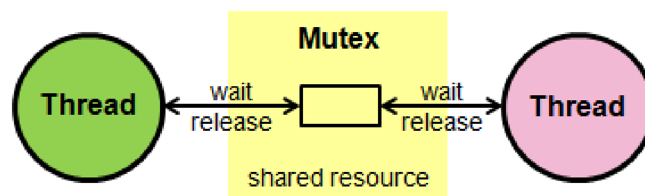


Figura 11-1 Esquema básico mútex

11.3 Variables de condición

Como se explicó anteriormente, sirven para sincronizar diferentes tareas. Esta consiste en una variable asociada a un mútex de tal manera que, si una tarea desea informar a otras de algún suceso, envía una señal por medio de dicha variable y así, aquellas tareas que estaban a la espera de algún cambio en la variable de condición son notificados del cambio ocurrido. De nuevo, el procedimiento para emplear una variable de condición es:

- *Creación de la variable:* emplearemos la función rt_cond_create ().
- *Eliminación de la variable:* mediante la función rt_cond_delete ().

Una tarea puede esperar una señal o puede enviar una señal. Esto se realiza de la siguiente manera:

- *Envío de la señal:* puede accederse de dos maneras principales
    - *Envío con un solo destinatario:* la señal se envía a una sola tarea que esté esperando la señal. Se emplea la función *rt_cond_signal ().*

- *Envío mediante broadcasting:* la señal se envía a todas las tareas que estén esperando la señal. Se emplea la función *rt_cond_broadcast ()*.
- *Espera de la señal:* hay dos tipos de espera principales
  - *Sin límite de tiempo:* la función espera a recibir la señal indefinidamente. Se emplea la función *rt_cond_wait ()*.
  - *Con límite de tiempo:* la función espera recibir la señal durante un periodo determinado de tiempo. Si en ese periodo no recibe la señal devuelve un error de timeout. Se emplea la función *rt_cond_wait_timed ()*.

11.4 Código de ejemplo

Se adjunta un pequeño código de ejemplo en el que una función se encarga de contar hasta 4. En ese momento, esa señal envía una señal a otra función que se encarga de mostrar por pantalla que ha recibido una señal. En este sencillo ejemplo se puede observar el modo de empleo de las funciones antes descritas.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <pigpio.h>
#include <alchemy/task.h>
#include <alchemy/mutex.h>
#include <alchemy/cond.h>

// Tasks, mutex and cond. var declaration //
RT_TASK T1, T2;
RT_MUTEX excl;
RT_COND sign;

// Shared variable //
int shared = 0;

// Function that counts to 5 //
void Task1(void *arg) {

        // Iniializa variables //
        int i = -1, signals = 0;

        // Function loop //
        rt_task_set_periodic(NULL,TM_NOW,rt_timer_ns2ticks(1000000000)); // 1s
period

        while(signals<5){

                // Updates values //
                i++;
                if(i==4) signals++;

                // Requests mutex //
                rt_mutex_acquire(&excl, TM_INFINITE);
                        shared = signals;
                        if(i==4){
                                rt_cond_signal(&sign);
                        }
                rt_mutex_release(&excl);

                // Show data and updates values //
                printf("Task1: counter value %d\n",i);
```

```
		if(i==4){
			i = 0;
			printf("Task1:  signal  sent.  Signal  number  %d\n”,
signals);
		}

		// Waits period //
		rt_task_wait_period(NULL);
	}
}

// Function to send PWM signal to motors //
void Task2(void *arg){

	// Initializes variables //
	int aux = 0, received = 0;

	// Fucntion loop //
	while(received<5){

		// Waits for signal //

		rt_mutex_acquire(&excl, TM_INFINITE);
			rt_cond_wait(&sign, &excl,TM_INFINITE);
				aux = shared;
					received++;
		rt_mutex_release(&excl);

		// Show data //
		printf("Task2: new signal. Total received: %d\n”, received);
		printf("Task2: Supposed received signals: %d\n", aux);
	}
}

// Main function //
int main(int argc, char* argv[]) {

	// Creates mutex, tasks and condition vars //
	rt_task_create(&T1,"SendSignalFunc",0, 50, T_JOINABLE);
	rt_task_create(&T2,"ReadSignalFunc",0, 50, T_JOINABLE);
	rt_mutex_create(&excl,"excl");
	rt_cond_create(&sign,"signal");

	// Starts tasks //
	rt_task_start(&T1, &Task1, NULL);
	rt_task_start(&T2, &Task2, NULL);

	// Waits for tasks //
	rt_task_join(&T1);
	rt_task_join(&T2);

	// Deletes mutex and cond var //
	rt_cond_delete(&sign);
	rt_mutex_delete(&excl);

}
```

12 ANEXO IV

En el siguiente anexo se procederá a mostrar el código desarrollado para el control de la Raspberry.

Este se ha dividido en un fichero principal y una serie de ficheros anexos que añaden funcionalidades y definiciones. Veamos detalladamente cada uno de los ficheros.

12.1 BMP.h y BMP.c

Ficheros para implementar el sensor barométrico. Incluye funciones para la calibración del sensor, la lectura de la presión barométrica y la implementación del sónar así como definiciones de los pines, registros y clases para que serán empleados en algún momento del código.

BMP.h

```
// BMP register map values //

#define BMP_180 0x77        // BMP i2c bus register
#define REG_AC1 0xAA        // Calibration data registers
#define REG_AC4 0xB0
#define REG_B1 0xB6
#define REG_M_C 0xF4        // Measurements control register
#define REG_MEDIDAS 0xF6    // Temperature and pressure measurements register

// Sonar gpio pins

#define Trig 22              // Sonar trigger pin
#define Echo 23              // Sonar echo pin

// Function input struct //

typedef struct input{

        int id_device;    // Variable to store BMP handle
        int cal_bmp[8];   // Variable to store int calibration data
        unsigned int uncal_bmp[3]; // Variable to store uint calibration data
        int error;        // Variable to store if an error has occured

} bmp;
```

BMP.c

```
// BMP/Sonar setting up function

bmp Calibration(int id_BMP){

        // Local variables initialization //

        int error = 0;  //  Variable  to  show  weather  calibration  data  was
received successfully
        unsigned int temp;
        int cal_bmp[8] = {0,0,0,0,0,0,0,0};    // Int-typed calibration data
        unsigned int uncal_bmp[3] = {0,0,0};  // Uint-typed calibration data
        char aux_cal[10];  // Auxiliary variable to store i2c received data
        bmp output;                            // Output function variable
        int i;

        // Calibration data request //

        printf("\n\nReading calibration data from BMP...");

        i2cReadI2CBlockData(id_BMP, REG_AC1, aux_cal, 6);

        for(i=0;i<3;i++){
                cal_bmp[i]=aux_cal[2*i]*256+aux_cal[2*i+1];
                if (cal_bmp[i]>32767) cal_bmp[i] -= 65536;
        }

        i2cReadI2CBlockData(id_BMP, REG_AC4, aux_cal, 6);

        for(i=0;i<3;i++){
                uncal_bmp[i]=aux_cal[2*i]*256+aux_cal[2*i+1];
        }

        i2cReadI2CBlockData(id_BMP, REG_B1, aux_cal, 10);

        for(i=0;i<5;i++){
                cal_bmp[3+i]=aux_cal[2*i]*256+aux_cal[2*i+1];
                if (cal_bmp[3+i]>32767) cal_bmp[3+i] -= 65536;
        }

        printf(" Received.\n");

        // Calibration data checkout //

        output.error = 0;

        printf("Checking out weather data has been received successfully...\n");

        for(i=0; i<8; i++){  // If any signed data is 0 error has occurred
                if(cal_bmp[i] == 0){
                        output.error = 1;
                }
        }

        for(i=0; i<3; i++){  // If any signed data is 0 error has occurred
                if(uncal_bmp[i] == 0){
                        output.error = 1;
                }
        }

        // We show up if this calibration has been successful or not //
```

```
        if(output.error == 1){
                printf("\n***** ERROR *****\n");
        } else {
                printf("BMP has been calibrated successfully.\n");
        }

        // Update output //

        for(i=0;i<9;i++){
                output.cal_bmp[i] = cal_bmp[i];
        }

        for(i=0;i<3;i++){
                output.uncal_bmp[i] = uncal_bmp[i];
        }

        output.id_device = id_BMP;

        return output;
}


// Preassure acquisition //

float Pressure(bmp input){

        // Local variables initialization //

        long int UT = 0, UP = 0;     // Variables to store data from sensor
        char storage[3];             // Variable to store i2c data block
        long int X1,X2,B5;           // Variable to get processed temperature
        long int B6,X3,B3;           // Variable to get processed pressure
        unsigned long int B4,B7;     // Variable to get processed pressure
        long int T;                  // Variable to store processed temperature
        float P;                     // Variable to store processed pressure
        float aux;                   // Auxiliary variable
        unsigned long int auxl;
        int i;


        // Temperature data acquisition //

        i2cWriteByteData(input.id_device,REG_M_C,0x2E);
        rt_task_sleep(5000000);
        i2cReadI2CBlockData(input.id_device,REG_MEDIDAS,storage,2);

        UT = storage[0]*256+storage[1];

        // Pressure data acquisition //

        i2cWriteByteData(input.id_device,REG_M_C,0x34);
        rt_task_sleep(5000000);
        i2cReadI2CBlockData(input.id_device,REG_MEDIDAS,storage,3);

        UP = (long)(storage[0]*256.0)+storage[1]+(storage[2]/256.0);

        // Temperature data processing //

        X1 = (long)(UT-input.uncal_bmp[2])*0.77713;

        aux = (input.cal_bmp[6]*2048.0)/((X1+input.cal_bmp[7])*1.0);
        X2 = (long)aux;
```

```
        B5 = X1+X2;

        T = (B5+8)/16;

        // Pressure data precessing //

        B6 = B5-4000;

        aux = ((B6*B6)/4096.0)*(input.cal_bmp[4]/2048.0);
        X1 = (long)aux;

        X2 = input.cal_bmp[1]*B6/2048,

        X3 = X1+X2;

        B3 = (input.cal_bmp[0]*4+X3+2)/4;

        X1 = (input.cal_bmp[2]*B6)/8192;

        X2 = (long)((B6*B6)/4096.0)*(input.cal_bmp[3]/65536.0);

        X3 = (X1+X2+2)/4;

        auxl = (unsigned long)(X3+32768);
        B4 = (auxl*input.uncal_bmp[0])/32768;

        B7 = (UP-B3)*50000;

        if(B7<0x80000000){
                P = (B7*2)/B4;
        } else {
                P = (B7/B4)*2;
        }

        X1 = (P/256.0)*(P/256.0);
        X1 = (X1*3038)/65536;

        X2 = (-7357*P)/65536;

        P = P+((X1+X2+3791)/16.0);

        return P;
}


// Sonar distance acquisition //

int Sonar(){

        // Local variables //

        float init,end,time_up;         // Variable to store time
        int distance;                   // Variable to store distance

        // Send pulse of 10us //

        gpioTrigger(Trig,10,1);         // Trigger signal

        // Read time while ECHO pin is high //

        while(gpioRead(Echo)==0);

        init = (double)rt_timer_read();
```

```
        while(gpioRead(Echo)==1);

        end = (double)rt_timer_read();

        time_up = end-init;

        // Convert time to distance //

        distance = (time_up*17150)/1000000000;

        return distance;
```

12.2 Brushless.h y Brushless.c

Ficheros para implementar el control de los PWM de los motores. Contienen la función de arranque inicial del los motores y los pines en los cuales están conectados los variadores así como la potencia base de cada motor.

Brushless.h

```
#define brushless_h

// Brushless ESC gpio pins

#define Motor1 4
#define Motor2 18
#define Motor3 17
#define Motor4 27

// Brushless U0 values

#define MotorBase1 1320
#define MotorBase2 1335
#define MotorBase3 1350
#define MotorBase4 1322
```

Brushless.c

```
// Starts up motors //

void InitMotors(void *arg){

    // Local variables initialization //

    int i=0;

    // Set task period //

    rt_task_set_periodic(NULL,TM_NOW,rt_timer_ns2ticks(5000000));

    // Sends PWM pulses //

    while(i<200){
        gpioServo(Motor1,1000);
        gpioServo(Motor2,1000);
        gpioServo(Motor3,1000);
        gpioServo(Motor4,1000);
        i++;
```

```
        rt_task_wait_period(NULL);
    }

    printf("\n\n***** MOTORS ARE READY TO ROTATE *****\n");

}
```

12.3 Socket.c

Ficheros en el cual se encuentra la función que permite inicializar el socket de comunicación con el usuario.

```
// Socket initialization function //

int sock_create(int port, int addr){

        // Local variables initialization //

        int sock, check;
        struct sockaddr_in sock1;

        // Socket creation //

        sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);

        // Socket definition //

        sock1.sin_family = AF_INET;
        sock1.sin_port = htons(port);
        sock1.sin_addr.s_addr = htonl(addr);

        check = bind(sock,(struct sockaddr *)&sock1, sizeof(sock1));

        if (check<0){
                sock = -1;
        }

        return sock;
}
```

12.4 Imu.h e Imu.c

En el fichero “.h” se encuentra la defínción de los diferentes registros que vamos a requerir a la hora de realizar la configuración o de extraer los datos del sensor. Por otra parte, en el fichero “.c” tenemos la configuración inicial necesaria para poder emplear la imu en con las condiciones que nosotros deseamos para esta aplicación.

IMU.h

```
#define imu_h

// Accel/Gyro/Mag register map values //

#define IMU_RATE_DIV 25            // MCU-6050 sample rate divider register
#define IMU_CONFIG 26              // MCU-6050 configuration register
#define IMU_GYR_CONF 27            // MCU-6050 gyro configuration register
#define IMU_ACCEL_CONF 28          // MCU-6050 accel configuration register
#define IMU_ACCEL_CONF_2 29        // MCU-6050 accel configuration 2 register
#define IMU_I2C_MASTER_CONTROL 36  // MCU-6050 i2c master control register
#define IMU_REG_BYPASS 55           // MCU-6050 bypass enable register
#define IMU_AC_X 59                 // MCU-6050 X axis accel register
#define IMU_AC_Y 61                // MCU-6050 Y axis accel register
#define IMU_AC_Z 63                // MCU-6050 Z axis accel register
#define IMU_REG_TEMP 65            // MCU-6050 temperature register
#define IMU_GY_X 67                // MCU-6050 X axis gyro register
#define IMU_GY_Y 69                // MCU-6050 Y axis gyro register
#define IMU_GY_Z 71                // MCU-6050 Z axis gyro register
#define IMU_USER_CONTROL 106      // MCU-6050 user control register
#define IMU_REG_POWER 107          // MCU-6050 power management register

#define IMU_SLV0_ADDR 37            // MCU-6050 slave 0 address register
#define IMU_SLV0_REG 38             // MCU-6050 slave 0 begin register
#define IMU_SLV0_CONTROL 39         // MCU-6050 slave 0 begin register
#define IMU_SLV0_DATAOUT 99         // MCU-6050 slave 0 data out register

#define IMU_EXT_SENS_DATA 73      // MCU-6050 slave 0 received data register

#define AK8963_ADDR 0x0C             // AK8963 I2C bus address
#define AK8963_MX 0x03               // AK8963 X axis compass register
#define AK8963_St2 0x09              // AK8963 read process register
#define AK8963_CONTROL 0x0A          // AK8963 control 1 register
#define AK8963_CONTROL2 0x0B         // AK8963 control 2 register
```

IMU.c

```
// Start-up imu function //

int AccelSetUp(int id_accel){

        // Local variables initialization //

        unsigned char buf[7];  // Variable to store sent message
        int counter = 0,j;     // Variable to count connection failures
        unsigned int aux,aux2;

        // Checks SPI interface //

        do {
                buf[0] = 245;                      // Read WhoIAm register
                aux = spiXfer(id_accel,buf,buf,2);
                aux2 = (unsigned int)buf[1];
                counter++;
        } while((aux2!=113)&&(counter<3));

        if(aux2 != 113){
                printf("Conenction lost. Connection attempts: %d\n",counter);
                return -1;
        }
```

```
printf("SPI interface is OK!. Received value: %u\n",aux2);

// Start IMU //

printf("\nStarting the device up: ");

buf[0] = IMU_REG_POWER;
buf[1] = 0;
aux = spiXfer(id_accel,buf,buf, 2);

buf[0] = IMU_REG_POWER+1;
buf[1] = 0;
aux = spiXfer(id_accel,buf,buf, 2);

rt_task_sleep(1000000);

printf("Device is already connected.\n");

// Setting device parameters up //

printf("Changing device accuracy: 2g y 250°/s \n");

buf[0] = IMU_GYR_CONF;
buf[1] = 0;
aux = spiXfer(id_accel,buf,buf, 2);

buf[0] = IMU_ACCEL_CONF;
buf[1] = 0;
aux = spiXfer(id_accel,buf,buf, 2);

rt_task_sleep(1000000);

printf("Setting low-pass filter: Hz\n");

buf[0] = IMU_ACCEL_CONF_2;
buf[1] = 3;
aux = spiXfer(id_accel,buf,buf, 2);

rt_task_sleep(1000000);

printf("Changing sample frequency: 1000 Hz \n");

buf[0] = IMU_CONFIG;
buf[1] = 3;
aux = spiXfer(id_accel,buf,buf, 2);

buf[0] = IMU_RATE_DIV;
buf[1] = 0;
aux = spiXfer(id_accel,buf,buf, 2);

rt_task_sleep(1000000);

// Disable AK8963 bypass //

printf("\n\nStarting up magnetometer: \n");

// Configure I2C Master //

buf[0] = IMU_USER_CONTROL;
buf[1] = 32;
aux = spiXfer(id_accel,buf,buf, 2);     // Enables multimaster

rt_task_sleep(1000000);
```

```
        buf[0] = IMU_I2C_MASTER_CONTROL;
        buf[1] = 13;
        aux = spiXfer(id_accel,buf,buf, 2);   // Set 400kHz master clock

        rt_task_sleep(1000000);

        // Reset AK8963 //

        buf[0] = IMU_SLV0_ADDR;
        buf[1] = AK8963_ADDR;                 // AK8963 I2C address
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_REG;
        buf[1] = AK8963_CONTROL2;             // Pointing at AK8963 Control2
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_DATAOUT;
        buf[1] = 1;
// Data to send to AK8963 Control2 register
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_CONTROL;
        buf[1] = 0x81;
        aux = spiXfer(id_accel,buf,buf, 2);

        rt_task_sleep(1000000);

        // Sleep AK8963 //

        buf[0] = IMU_SLV0_ADDR;
        buf[1] = AK8963_ADDR;                 // AK8963 I2C address
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_REG;
        buf[1] = AK8963_CONTROL;              // Pointing at AK8963 Control
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_DATAOUT;
        buf[1] = 0;                           // Data to send
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_CONTROL;
        buf[1] = 0x81;
        aux = spiXfer(id_accel,buf,buf, 2);

        rt_task_sleep(1000000);

        // Set AK8963 mode //

        buf[0] = IMU_SLV0_ADDR;
        buf[1] = AK8963_ADDR;                 // AK8963 I2C address
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_REG;
        buf[1] = AK8963_CONTROL;              // Pointing at AK8963 Control1
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_DATAOUT;
        buf[1] = 22;                          // Data to send: 6bits continuous
mode 100Hz
        aux = spiXfer(id_accel,buf,buf, 2);
```

```
        buf[0] = IMU_SLV0_CONTROL;
        buf[1] = 0x81;
        aux = spiXfer(id_accel,buf,buf, 2);

        rt_task_sleep(1000000);

        // Check whether configuration has been done successfully //

        counter = 0;

        do {

                buf[0] = IMU_SLV0_ADDR;
                buf[1] = 140;            // AK8963 I2C address as reading mode
                aux = spiXfer(id_accel,buf,buf, 2);

                buf[0] = IMU_SLV0_REG;
                buf[1] = 0x00;           // Register to read: WhoAmI register
                aux = spiXfer(id_accel,buf,buf, 2);

                buf[0] = IMU_SLV0_CONTROL;
                buf[1] = 0x81;           // Read 1 byte from magnetometer
                aux = spiXfer(id_accel,buf,buf, 2);

                rt_task_sleep(10000000);

                buf[0] = IMU_EXT_SENS_DATA+128;    // Read EXT_SENS_DATA_0
                aux = spiXfer(id_accel,buf,buf,2);
                aux2 = (unsigned int)buf[1];

                counter++;

        } while((aux2!=72)&&(counter<3));

        if(aux2 != 72){
                printf("Conenction lost. Connection attempts: %d\n",counter);
                return -1;
        }
        printf("SPI interface is OK!. Received value: %u\n",aux2);

        // Prepare magnet to be read //

        buf[0] = IMU_SLV0_ADDR;
        buf[1] = AK8963_ADDR+128;        // AK8963 I2C address as reading mode
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_REG;
        buf[1] = 0x00;                   // Register to read: WhoAmI register
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_CONTROL;
        buf[1] = 0x83;                   //  Read  3  byte  from  magnetometer:
WhoIAM, Info and Status1 registers
        aux = spiXfer(id_accel,buf,buf, 2);

        rt_task_sleep(1000000);

        buf[0] = IMU_SLV0_ADDR;
        buf[1] = AK8963_ADDR+128;       // AK8963 I2C address as reading mode
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_REG;
        buf[1] = 0x09;                 // Register to read: Status2
```

```
        aux = spiXfer(id_accel,buf,buf, 2);

        buf[0] = IMU_SLV0_CONTROL;
        buf[1] = 0x81;                  // Read 1 byte from magnetometer
        aux = spiXfer(id_accel,buf,buf, 2);

        rt_task_sleep(1000000);

        buf[0] = IMU_EXT_SENS_DATA+128; // Read EXT_SENS_DATA_0 register
        spiXfer(id_accel,buf,buf,8);

        rt_task_sleep(1000000);

        return 0;
}
```

12.5 Main.c

En este último fichero se encuentra la definición de las variables globales que serán requeridas a lo largo de la ejecución del código, así como la función principal y las funciones que de desarrollaran los diferentes hilos que componen nuestro software.

Las definiciones iniciales son:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <math.h>
#include <pigpio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <alchemy/timer.h>
#include <alchemy/task.h>
#include <alchemy/mutex.h>
#include <alchemy/cond.h>
#include "imu.h"
#include "imu.c"
#include "brushless.h"
#include "brushless.c"
#include "bmp.h"
#include "bmp.c"
#include "sockets.c"

// Tasks frequency //

#define T_accel 1000000                // 1ms period for task
#define T_height  100000000            // 100ms period
#define T_PWM  5000000                 // 5ms period

// UDP parameters //

#define Port 20000                     // Socket port
#define IP INADDR_ANY                  // Socket IP to connect

// Tasks declaration //

RT_TASK UDP_Read;
```

```
RT_TASK StartMotor;
RT_TASK AccelData;
RT_TASK RelHeight;
RT_TASK PIDcontrol;
RT_TASK EscPwm;

// Mutex declaration //

RT_MUTEX excl_inp;
RT_MUTEX excl_pwm;

// Condition variable declaration //

RT_COND accel_signal;

// Shared variables controlled by mutex exc_inp //

double yaw = 0, pitch = 0, roll = 0;  // Actual drone position & inclination
double reference[4] = {0,0,0,6.0};    // Drone position & height reference
int actual_height = 0;                // Actual quadrotor height
int received_end = 0;                 // Variable to stop in order all tasks
int received_start = 0;               // Variable to start system

// Shared variables controlled by mutex exc_pwm //

int control_action[4] = {0,0,0,0};    // Control action for brushless motors
int end_flag = 0;                     // Variable to stop in order all tasks

// Global non-shared variables //

bmp bmp_params;
float init_press;                     // Variable to store initial pressure
```

El código correspondiente al hilo cuya función se encarga de comunicarse con la IMU y operar con los datos que esta le facilita es el siguiente:

```
// Accel/Gyro/Compass data acquisition //

void AccelAngles(void *arg){

        // Local variables initialization //

        const float alpha = 0.90;      // Low pass filter value
        const float alpha_yaw = 0.99;  // Low pass filter value for yaw
        int id_accel = (int)arg;       // Unsigned handle for accel
        unsigned char buf[15];         // Variable to store sent message
        unsigned int sensor[15];       // Variable to store accel/gyro data
        unsigned int sensor2[7];       // Variable to store compass data
        int raw_values[9];             // Variable to store raw values
        float prepro_data[9];          // Variable to store pre-processed data
        float angles[3]={0,0,0};       // Variable to store actual angles
        float filtered[3] = {0,0,0};   // Variable to store lpf angles
        float ant_angles[3]={0,0,0};   // Variable to store old angles
        float time,time_ant,time_aux;  // Variable to store time
        int end = 0;                   // Variable to end process
        int reference_ok = 0;          // Variable to check Z-axis reference
        int j = 0,k = 0, counter = 0;  // Auxiliary variables
        float aux,aux2,angles_rad[2];

        // Set Magnet_addr to read //

        buf[0] = IMU_SLV0_ADDR;
```

```
	buf[1] = 140;                         // AK8963 I2C address
	aux = spiXfer(id_accel,buf,buf, 2);

	// Sets task frequency //

	rt_task_set_periodic(NULL,TM_NOW,rt_timer_ns2ticks(T_accel));

	// Initialize time_ant variable //

	time_ant = (double)rt_timer_read();

	// Task loop //

	while(end == 0){

		// Read system time //

		time = (double)rt_timer_read();

		// Accel/Gyro data request //

		buf[0] = 128+IMU_AC_X;
		spiXfer(id_accel,buf,buf,15);


		for(j=0;j<14;j++){
			sensor[j] = (int)buf[j+1];
		}

		// Magnet data request //

		if (k==0) {

			buf[0] = IMU_SLV0_ADDR;
			buf[1] = 140;  // AK8963 address as reading mode
			spiXfer(id_accel,buf,buf, 2);

			buf[0] = IMU_SLV0_REG;
			buf[1] = 0x03; // Register to read: WhoAmI register
			spiXfer(id_accel,buf,buf, 2);

			buf[0] = IMU_SLV0_CONTROL;
			buf[1] = 0x87; // Read 1 byte from magnetometer
			spiXfer(id_accel,buf,buf, 2);

		} else if (k==1) {

			buf[0] = IMU_EXT_SENS_DATA+128; //Read EXT_SENS_DATA_0
			spiXfer(id_accel,buf,buf,8);

			for(j=0;j<7;j++){
				sensor2[j] = (int)buf[j+1];
			}

		}

		// Adapt data info //

		raw_values[0] = sensor[0]*256+sensor[1]; // Reads X accel
		if (raw_values[0]>32767) raw_values[0] -= 65536;

		raw_values[1] = sensor[2]*256+sensor[3]; // Reads Y accel
		if (raw_values[1]>32767) raw_values[1] -= 65536;
```

```
                raw_values[2] = sensor[4]*256+sensor[5]; // Reads Z accel
                if (raw_values[2]>32767) raw_values[2] -= 65536;

                raw_values[3] = sensor[8]*256+sensor[9]; // Reads X gyro
                if (raw_values[3]>32767) raw_values[3] -= 65536;

                raw_values[4] = sensor[10]*256+sensor[11]; // Reads Y gyro
                if (raw_values[4]>32767) raw_values[4] -= 65536;

                raw_values[5] = sensor[12]*256+sensor[13]; // Reads Z gyro
                if (raw_values[5]>32767) raw_values[5] -= 65536;

                raw_values[6] = sensor2[1]*256+sensor2[0]; // Reads X mag
                if (raw_values[6]>32767) raw_values[6] -= 65536;

                raw_values[7] = sensor2[3]*256+sensor2[2]; // Reads Y mag
                if (raw_values[7]>32767) raw_values[7] -= 65536;

                raw_values[8] = sensor2[5]*256+sensor2[4]; // Reads Z mag
                if (raw_values[8]>32767) raw_values[8] -= 65536;

                // Data adaptation //

                prepro_data[0] = (raw_values[0]-1404)*0.000598;
                prepro_data[1] = (raw_values[1]-108)*0.000598;
                prepro_data[2] = (raw_values[2]+2611)*0.000598;
                prepro_data[3] = (raw_values[3]-222)/131.0;
                prepro_data[4] = (raw_values[4]+148)/131.0;
                prepro_data[5] = (raw_values[5]-351)/131.0;
                prepro_data[6] = raw_values[6];
                prepro_data[7] = raw_values[7];
                prepro_data[8] = raw_values[8];

                // Pitch angle //

                aux = 0;
                aux = pow(prepro_data[1],2)+pow(prepro_data[2],2);
                aux = atan2(prepro_data[0],sqrt(aux));
                aux = aux*(180/3.1415);

                angles[0]        =        0.98*(angles[0]+prepro_data[3]*((time-
time_ant)/1000000000.0)) + 0.02*aux;
                angles_rad[0] = angles[0]*(3.1415/180);

                // Roll angle //

                aux = 0;
                aux = pow(prepro_data[0],2)+pow(prepro_data[2],2);
                aux = atan2(prepro_data[1],sqrt(aux));
                aux = aux*(180/3.1415);

                angles[1]        =        0.98*(angles[1]+prepro_data[4]*((time-
time_ant)/1000000000.0)) + 0.02*aux;
                angles_rad[1] = angles[1]*(3.1415/180);

                // Yaw angle //

                aux = 0;
                aux2 = 0;
                aux           =           (prepro_data[7]*cos(angles_rad[0]))-
(prepro_data[8]*sin(angles_rad[0]));
```

```
                aux2                                                          =
((prepro_data[6]*cos(angles_rad[1]))+(prepro_data[7]*sin(angles_rad[0])*sin(a
ngles_rad[1]))+(prepro_data[8]*cos(angles_rad[0])*sin(angles_rad[1])));
                aux = atan2(aux,aux2);
                aux = aux*(180/3.1415);

                angles[2]       =       0.99*(angles[2]+prepro_data[5]*((time-
time_ant)/1000000000.0)) + 0.01*aux;

                // Low pass filter to reduce noise //

                filtered[0] = (1-alpha)*angles[0]+alpha*ant_angles[0];
                filtered[1] = (1-alpha)*angles[1]+alpha*ant_angles[1];
                filtered[2] = (1-alpha_yaw)*angles[2]+alpha_yaw*ant_angles[2];

                // Requests mutex to update variables //

                rt_mutex_acquire(&excl_inp,TM_INFINITE);
                pitch = filtered[0];
                roll = filtered[1];
                yaw = filtered[2];
                end = received_end;
                if((k==2)&&(reference_ok==1)){
                        rt_cond_signal(&accel_signal);
                }
                rt_mutex_release(&excl_inp);

                // When get  position set as reference //

                if (counter == 1000){
                        rt_mutex_acquire(&excl_inp,TM_INFINITE);
                        reference[0] = filtered[0];
                        reference[1] = filtered[1];
                        reference[2] = filtered[2];
                        rt_mutex_release(&excl_inp);
                        reference_ok = 1;
                }

                // Update variables //

                ant_angles[0] = filtered[0];
                ant_angles[1] = filtered[1];
                ant_angles[2] = filtered[2];
                time_ant = time;

                if (counter<1001) counter++;

                k++;
                if (k>9) k=0;          // Wait 10ms to read compass again

                // Waits till next cycle //

                rt_task_wait_period(NULL);

        }

        // Send signals for ending process //

        rt_mutex_acquire(&excl_inp,TM_INFINITE);
        rt_cond_signal(&accel_signal);  // Sends signal to release PID cycle
        rt_mutex_release(&excl_inp);

}
```

El siguiente hilo es el encargado de obtener la altitud del drone:

```
// Relative height acquisition //

void Height(void *arg){

        // Local variables initialization //

        int id_BMP = (int)arg;
        float actual_press[5];       // Variable to store actual pressure
        float actual_press_mean;     // Variable to store actual pressure mean
        int h;                       // Variable to store relative height
        int floor_sonar;             // Variable to store sonar distance
        int end = 0;                 // Variable to end process
        int aux;

        // Initialize actual_press data //

        for(aux=0;aux++;aux<5){
                actual_press[aux] = init_press;
        }

        // Set periodic task //

        rt_task_set_periodic(NULL,TM_NOW,rt_timer_ns2ticks(T_height));

        while(end == 0){

                // Distance to the floor according to sonar //

                floor_sonar = Sonar();

                // If floor is far away (max. reliable measurement = 3m) //

                if((floor_sonar>275)||(floor_sonar<4)){

                        for(aux=0;aux++;aux<4){
                                actual_press[aux+1] = actual_press[aux];
                        }

                        actual_press[0] = Pressure(bmp_params)/100.0;
                        actual_press_mean = 0;

                        for(aux=0;aux++;aux<5){ // Mean filter to reduce noise
                                actual_press_mean += actual_press[aux];
                        }
                        actual_press_mean = actual_press_mean/5;

                        h                      =                      44330*(1-
pow((actual_press_mean/init_press),1/5.255));      // Actual relative height
                        h                                                     =
h*100;
                                // Get height in cm instead of m

                } else {                  // If floor is close to the sonar

                        h = floor_sonar;

                }
```

```
                // Requests mutex to update variables //

                rt_mutex_acquire(&excl_inp,TM_INFINITE);
                actual_height = h;
                end = received_end;
                rt_mutex_release(&excl_inp);

                // Waits till next cycle //

                rt_task_wait_period(NULL);

        }

}
```

A continuación, mostraremos el hilo encargado de calcular las acciones de control que se ejercerán sobre cada motor:

```
// PID controller function //

void PID(void *arg){

        // Initializes variables //

        double inp_ref[4] = {0,0,0,0};     //  Variable  to  store  reference
[pitch,roll,yaw,height]
        double inp_state[4] = {0,0,0,0}; // variable to store actual state
[pitch,roll,yaw,height]
        double error[4] = {0,0,0,0};       //   Variable   to   store   error
[pitch,roll,yaw,height]
        double error_ant[4] = {0,0,0,0};   // Variable to store previous error
        double derivada[4] = {0,0,0,0};    // Variable to store derivatives
        double derivada_ant[4] = {0,0,0,0}; // Variable to store previous
derivatives
        double control_act[4] = {0,0,0,0}; // Variable to store control actions
[pitch,roll,yaw,height]
        double I[4] = {0,0,0,0};                // Variable to store error Ik
[pitch,roll,yaw,height]
        const double alpha = 0.9800000;         // Low-pass filter parameter
        float T,time_ant,time;                  // Variable to store time
        int end = 0,k;                          // Variable to end the task
        double aux1, aux2, aux3;
        int i;


        // Creates file //

        pf =  fopen("Angles_yaw", "w");

        // Initializes time data //

        time_ant = (double)rt_timer_read();

        // Repeat till need to finish task //

        while(end == 0){

// Requests mutex to update angle/height variables and waits for data to be
ready//

                rt_mutex_acquire(&excl_inp,TM_INFINITE);
```

```
		rt_cond_wait(&accel_signal,&excl_inp,TM_INFINITE);
		inp_ref[0] = reference[0];
		inp_ref[1] = reference[1];
		inp_ref[2] = reference[2];
		inp_ref[3] = reference[3];
		inp_state[0] = pitch;
		inp_state[1] = roll;
		inp_state[2] = yaw;
		inp_state[3] = actual_height;
		end = received_end;
		rt_mutex_release(&excl_inp);

		// Read current time //

		time = (double)rt_timer_read();
		T = (time-time_ant)/1000000;

		// Round to 2 decimal, calculates error and updates Ik//

		for(i=0;i<4;i++){
			inp_state[i] = round((inp_state[i]*100))/100;
			error[i] = (int)inp_ref[i]- inp_state[i];
			derivada[i] = error[i] - error_ant[i];
			if ((I[i]+error[i]<3000) && (I[i]+error[i]>-3000)){
				I[i] += error[i];
			}
		}

		// Get shortest rotation wise //

		if (error[2]<-180){
			error[2] += 360;
		} else if (error[2]>180){
			error[2] -=360;
		}

		// Calculates Diferential PI (Euler II) control action //

		control_act[0]                                          =
1.200000000000*(error[0]+0.00007*T*I[0]+(20/T)*(derivada[0]*(1-
alpha)+derivada_ant[0]*alpha));
		control_act[1]                                          =
1.350000000000*(error[1]+0.0001*T*I[1]+(20/T)*(derivada[1]*(1-
alpha)+derivada_ant[1]*alpha));
		control_act[2] = 1.350000000000*(error[2]+0.0001*T*I[2]);
		control_act[3] = 1.500000000000*(error[3]+0.0001*T*I[3]);


		// Round to 2 decimal control actions //

		for(i=0;i<4;i++){
			control_act[i] = round((control_act[i]*100))/100;
		}

		// Updates control action variable control //

		rt_mutex_acquire(&excl_pwm,TM_INFINITE);

		control_action[0]                                       =
(int)(MotorBase1+control_act[0]+control_act[3]);    //  Motor  1  (pitch  +
height)
	       control_action[1]            =               (int)(MotorBase2-
control_act[1]+control_act[3]);             // Motor 2 (roll + height)
```

```
              control_action[2]                =              (int)(MotorBase3-
control_act[0]+control_act[3]);       // Motor 3 (pitch + height)
              control_action[3]                                                =
(int)(MotorBase4+control_act[1]+control_act[3]);            // Motor 4 (roll
+ height)

                end_flag = end;
                rt_mutex_release(&excl_pwm);

                // Updates value //

                if (k<2005) k++;
                time_ant = time;
                for (i=0;i<4;i++){
                        error_ant[i] = error[i];
                        derivada_ant[i] = derivada[i];
                }
        }
}
```

El código encargado de mandar las señales PWM a los ESC es:

```
// Send PWM pulses to ESC //

void WritePWM(void *arg){

        // Initializes variables //

        int PWM_Value[4] = {0,0,0,0};  // Variable to store control action
        int end = 0;                   // Variable to end up task iteration
        int i;

        // Sets task as periodic //

        rt_task_set_periodic(NULL,TM_NOW,rt_timer_ns2ticks(T_PWM));

        // Repeat till need to finish task //

        while(end == 0){

                // Requests mutex to update variables //

                rt_mutex_acquire(&excl_pwm,TM_INFINITE);
                PWM_Value[0] = control_action[0];
                PWM_Value[1] = control_action[1];
                PWM_Value[2] = control_action[2];
                PWM_Value[3] = control_action[3];
                end = end_flag;
                rt_mutex_release(&excl_pwm);

                // Saturation values //

                for(i=0;i<4;i++){
                        if (PWM_Value[i]<1000){
                                PWM_Value[i] = 1000;
                        } else if (PWM_Value[i]>1800){
                                PWM_Value[i] = 1800;
                        }
                }

                // Writes PWM pulses //
```

```
                gpioServo(Motor1,PWM_Value[0]);
                gpioServo(Motor2,PWM_Value[1]);
                gpioServo(Motor3,PWM_Value[2]);
                gpioServo(Motor4,PWM_Value[3]);

                // Waits cycle //

                rt_task_wait_period(NULL);
        }
}
```

El último hilo es el encargado de la comunicación con el usuario:

```
// Function to read data from socket //

void ReadSock(void *arg){

        // Local variables initialization //

        int sock = (int)arg;
        int lecture = 0;               // Variable to store received data
        struct sockaddr_in sock1;      // Socket addr to respond
        double local_ref[4] = {-3.0,0,0,6.0};         // Variable to store
        int flag = 0,i;

        // Socket definition //

        sock1.sin_family = AF_INET;
        sock1.sin_port = htons(20000);
        sock1.sin_addr.s_addr = inet_addr("192.168.43.1");

        // Wait to receive start system order //

        while (lecture!=9){
                recv(sock,&lecture,sizeof(lecture),MSG_WAITALL);
                for(i=0;i<3;i++){
                        sendto(sock,&lecture,sizeof(lecture),0,(struct sockaddr
*)&sock1,sizeof(sock1));
                }
                lecture = lecture-65;
        }

        printf("SYSTEM STARTED!\n");

        // Communicates start requirement to other tasks //

        rt_mutex_acquire(&excl_inp,TM_INFINITE);
        received_start = 1;
        rt_mutex_release(&excl_inp);

        // Repeat indefinitely //

        while (flag==0){

                recv(sock,&lecture,sizeof(lecture),MSG_WAITALL);
                for(i=0;i<3;i++){
                        sendto(sock,&lecture,sizeof(lecture),0,(struct sockaddr
*)&sock1,sizeof(sock1));
                }
                lecture = lecture-65;
```

```
                switch(lecture){
                        case 0:
                                for(i=0;i<2;i++) local_ref[i] = 0;
                                break;
                        case 1:
                                if (local_ref[0]<3) {
                                        local_ref[0] += 1;
                                }
                                break;
                        case 3:
                                if (local_ref[1]<3) {
                                        local_ref[1] += 1;
                                }
                                break;
                        case 4:
                                if (local_ref[0]>-3) {
                                        local_ref[0] -= 1;
                                }
                                break;
                        case 5:
                                if (local_ref[1]>-3) {
                                        local_ref[1] -= 1;
                                }
                                break;
                        case 7:
                                if (local_ref[3]<30) {
                                        local_ref[3] += 5;
                                }
                                break;
                        case 8:
                                if (local_ref[3]>-30) {
                                        local_ref[3] -= 5;
                                }
                                break;
                        case 10:
                                flag = 1;
                                break;
                }

                // Communicates for very last time //

                rt_mutex_acquire(&excl_inp,TM_INFINITE);
                reference[0] = local_ref[0];
                reference[1] = local_ref[1];
                reference[2] = local_ref[2];
                reference[3] = local_ref[3];
                received_end = flag;
                rt_mutex_release(&excl_inp);
        }
}
```

Por último, el código de la función principal es:

```
// Main function. Starts-up system and creates tasks //

int main(int argc, char* argv[]){

  // Local variables initialization //

  int id_accel,id_bmp;                        // Connections identifications
  int sock_android;                           // Socket to receive data
```

```
  char error = 0;      // Variable to store calibration() success or not
  int aux = 0;


  // Initialize GPIO //

  gpioInitialise();

  // Set up i2c connection to BMP //

  id_bmp = i2cOpen(1,BMP_180,0);

  // Set up SPI connection to IMU //

  gpioSetMode(19, PI_OUTPUT);
  gpioSetMode(21, PI_INPUT);
  gpioSetMode(23, PI_OUTPUT);
  gpioSetMode(24, PI_OUTPUT);
  gpioSetMode(25, PI_OUTPUT);


  id_accel = spiOpen(0, 1000000, 0);          // Channel 0, 1Mhz

  if(id_accel < 0){
          printf("\nConnection    failure.    Unable    to    connect.    Error:
%d\n",id_accel);
          return 0;
  }

  printf("\nConnection established. Accel/gyro id: %d\n",id_accel);

  // Set up GPIO connections //

  gpioSetMode(Trig,PI_OUTPUT);        // Trigger pin
  gpioSetMode(Echo,PI_INPUT);         // Echo pin

  // Create WiFi socket //

  sock_android = sock_create(Port,IP);

  // Check whether connection has failed //

  if(sock_android<0){
          printf("\nUPS...Something went wrong creating UDP socket.\n\n");
          return 0;
  } else {
          printf("\nSuccessful socket creation attempt.\n\n");
  }

  // Mutex declaration //

  rt_mutex_create(&excl_inp, "MutInput"); // Creates mutex for angles & height
  rt_mutex_create(&excl_pwm, "MutPWM");  // Creates mutex for brushless control
action

  // Condition variable declaration //

  rt_cond_create(&accel_signal, "AccelAvailable");

  // Devices initialization before running tasks //

  aux = AccelSetUp(id_accel);
```

```
  if(aux==-1){
          return 0;
  }

  do {          // Tries 5 time calibration if it is detected as wrongly done
          bmp_params = Calibration(id_bmp);
          aux++;
  } while((bmp_params.error==1)&&(aux<5));

  if (aux == 5) { // If calibration data has failed 5 times it stops running.
          printf("\n***** MAX CALIBRATION INTERATIONS REACHED *****\n");
          return 0;
  }

  init_press = Pressure(bmp_params)/100.0;    // We read very first pressure to
obtaining relative pressure differences then

  // Tasks creation //

  rt_task_create(&StartMotor,"InitMotors", 0, 50, T_JOINABLE);
  rt_task_create(&UDP_Read,"ReadWiFi", 0, 50, T_JOINABLE);
  rt_task_create(&AccelData,"AccelData", 0, 50, T_JOINABLE);
  rt_task_create(&RelHeight,"BMPData", 0, 50, T_JOINABLE);
  rt_task_create(&PIDcontrol,"PID", 0, 50, T_JOINABLE);
  rt_task_create(&EscPwm,"ESCPwm", 0, 50, T_JOINABLE);

  // Brushless motors initialization //

  rt_task_start(&StartMotor, &InitMotors, NULL);
  rt_task_join(&StartMotor);

  // Activates motor led //

  gpioWrite(25, 1);

  // WiFi data reception task initialization //

  rt_task_start(&UDP_Read, &ReadSock, (void *)sock_android);

  // Wait until start system order //

  aux = 0;

  do {

          rt_mutex_acquire(&excl_inp,TM_INFINITE);
          aux = received_start;
          rt_mutex_release(&excl_inp);
          rt_task_sleep(10000000);

  } while(aux==0);

  // Periodic tasks //

  rt_task_start(&AccelData, &AccelAngles, (void *)id_accel);
  rt_task_start(&RelHeight, &Height, (void *)id_bmp);
  rt_task_start(&PIDcontrol, &PID, NULL);
  rt_task_start(&EscPwm, &WritePWM, NULL);

  rt_task_join(&AccelData);
  rt_task_join(&RelHeight);
  rt_task_join(&PIDcontrol);
  rt_task_join(&EscPwm);
```

```
  rt_task_join(&UDP_Read);

  // Shut down brushless //

  for (aux = 0; aux < 50; aux++){
          gpioServo(Motor1,1000);
          gpioServo(Motor2,1000);
          gpioServo(Motor3,1000);
          gpioServo(Motor4,1000);
          rt_task_sleep(20000000);
  }

  // Desactivates motor led //

  gpioWrite(25, 0);

  // Shut down interfaces //

  i2cClose(id_bmp);
  spiClose(id_accel);
  close(sock_android);
  gpioTerminate();

}
```

13 ANEXO V

En el siguiente anexo se procederá a mostrar el código desarrollado para el control del Arduino. Haremos distinción entre el código que implementa el DMP y aquel que no.

13.1 Implementación del DMP

```
/*********************************************/
/*      Incluimos las librerias externas      */
/*********************************************/

#include "I2Cdev.h"
#include "MPU6050_9Axis_MotionApps41.h"
#include "Wire.h"
#include "SoftwareSerial.h"

/*********************************************/
/*   Realizamos las definiciones de obtetos   */
/*********************************************/

MPU6050 mpu(0x68);
SoftwareSerial BT(13,4);

/*********************************************/
/*          Definiciones de nombres           */
/*********************************************/

#define MPUInterrupt_PIN      2                // Pin de interrupcion del DMP
#define MOTOR1                9                // Pin del motor 1
#define MOTOR2                10               // Pin del motor 2
#define MOTOR3                11               // Pin del motor 3
#define MOTOR4                3                // Pin del motor 4

/*********************************************/
/*          Definiciones de vaiables          */
/*********************************************/

/* Variables relacionadas con la IMU */

bool dmpCorrecta = false;   // Variable para indicar si el DMP se inicio bien
uint16_t paqueteDatos;      // Variable para almacenar tamaño de un paquete
uint8_t fifoBuffer[64];     // Variable para almacenar la lectura de la FIFO
uint8_t mpuIntStatus;       // Variable para almacenar la interrupcion
uint16_t iteraciones = 0;   // Variable para el numero de muestras del eje Z

/* Variables relacioandas con las interrupciones */

volatile bool mpuInterrupt = false;  // Variable para indicar interrupción

/* Variable para el control de los motores */

bool flag_m1, flag_m2, flag_m3, flag_m4;
```

```
/* Variables para el calculo de controladores */

#define T 500

/* Variables para el control del sistema */

bool fin_recibido = false;
bool imu_lista = false;
bool init_recibido = false;

/*********************************************/
/*         Gestion de interrupciones         */
/*********************************************/

void datoDMP() {
    mpuInterrupt = true;
}

/*********************************************/
/*                   Setup                   */
/*********************************************/

void setup()  {

  /* Inicializamos las variables */

  bool conexion;              // Variable para comprobar la conexion

  /* Iniciamos las comunicación serial */

  Serial.begin(2000000);      // Transmision de datos al PC a 2000000 bits/s

  /* Iniciamos comunicacion BT para recibir mensajes */

  BT.begin(57600);

  /* Configuramos el bus I2C */

  Wire.begin();
  Wire.setClock(400000);      // Frecuencia de reloj: 400 kHz

  /* Inicializamos la IMU y comprobamos la conexion */

  mpu.initialize();

  if (mpu.testConnection() == 0){
    Serial.println("IMU Iniciada correctamente");
  } else {
    Serial.println("Error al iniciar la IMU");
  }

  /* Inicializamos el pin de interrupciones */

  pinMode(MPUInterrupt_PIN, INPUT);

  /* Inicializamos el DMP */

  Serial.println("Inicializando DMP...");
  conexion = mpu.dmpInitialize();

  /* Activamos DMP y le asociamos una interrupcion */

  if (conexion == 0) {
```

```
    mpu.setDMPEnabled(true);            // Indicamos que el DMP está activado

    attachInterrupt(digitalPinToInterrupt(MPUInterrupt_PIN), datoDMP, RISING);
// Asociamos la interrupcion externa a una funcion

    mpuIntStatus = mpu.getIntStatus();

    dmpCorrecta = true;               // Variable para indicar DMP disponible

    paqueteDatos = mpu.dmpGetFIFOPacketSize();    // Leemos tamaño paquete

    Serial.println("DMP inicializada correctamente");

  } else {

    Serial.print("Error al inicializar el DMP");

  }

  /* Configuramos las salidas para los motores */

  pinMode(MOTOR1,OUTPUT);
  pinMode(MOTOR2,OUTPUT);
  pinMode(MOTOR3,OUTPUT);
  pinMode(MOTOR4,OUTPUT);

  /* Apagamos los motores por seguridad */

  digitalWrite(MOTOR1, LOW);
  digitalWrite(MOTOR2, LOW);
  digitalWrite(MOTOR3, LOW);
  digitalWrite(MOTOR4, LOW);

}

/**********************************************/
/*                Software Drone              */
/**********************************************/

void loop() {

  /*    Inicializamos    las    variables    para    la    imu.    Para    los
angulos/errores/referencias el orden es [Z,Y,X] */

  Quaternion q;              // Variable para almacenar el quaternion del DMP
  VectorFloat gravedad;      // Variable intermedia calculo angulos
  float ypr[3];              // Variable para almacenar angulos
  static float ypr_ant[3];   // Variable para almacenar angulos anteriores
  uint16_t fifoCount;        // Variable para numero de bytes en la FIFO

  /* Inicializamos las variables para los controladores */

  static float referencias[3] = {0.0,0.0,0.0};   // Referencias
  float errores[4] = {0.0,0.0,0.0};              // Errores
  static float errores_ant[4] = {0.0,0.0,0.0};   // Errores del ciclo ant.
  float derivada[4] = {0.0,0.0,0.0};             // Derivadas de los errores
  float derivada_ant[4] = {0.0,0.0,0.0};     // Derivadas de los errores ciclo
ant.
  static float I[4] = {0.0,0.0,0.0};             // Errores acumulados
  float accion_control[4] = {0,0,0};             // Acciones de control

  /* Inicializamos las variables para los pulsos PWM */
```

```
  float accion_m1, accion_m2, accion_m3, accion_m4;     // Acciones motores
  bool aM1, aM2, aM3, aM4 = false;

  /* Inicializamos las variables para la comunicacion BT */

  char mensaje = 0;

  /* Variable para el cierre ordenado */

  bool fin_listo = 0;

  /* Variables auxiliares */

  static int i,j = 5;

  /* Variables para el timing */
  float loop_timer, esc_loop_timer;

  /* Si la imu falla detenemos el programa */

  if (!dmpCorrecta) return;

  /* En caso de que no falle continuamos con la ejecucion normal. Comprobamos
la interrupción que haya habido. */

  if (mpuInterrupt && !fin_recibido) {

    mpuInterrupt = false;      // Actualizamos el valor de la interrupcion
    mpuIntStatus = mpu.getIntStatus(); // Obtenemos la interrupcion producida

    /* Comprobamos que no haya habido overflow de la FIFO */

    fifoCount = mpu.getFIFOCount();  // Leemos numero de bytes en la FIFO

    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {       // Si ha habido
overflow: por flag o por lectura de registro:

      mpu.resetFIFO();            // reseteamos la FIFO
      Serial.println("FIFO overflow!");

    } else if (mpuIntStatus & 0x02) {

      /* Esperamos a que haya un paquete entero disponible */

      while (fifoCount < paqueteDatos) fifoCount = mpu.getFIFOCount();

      /* Una vez obtenido el paquete se lee */

      mpu.getFIFOBytes(fifoBuffer, paqueteDatos);
      mpu.resetFIFO();

      /* Actualizamos el valor del buffer de la FIFO */

      fifoCount -= paqueteDatos;

      /* Obtenemos los angulos de Euler */

      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravedad, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravedad);

     /* Cambiamos a grados los ángulos */
```

```
     ypr[0] = ypr[0]*(180/M_PI);
     ypr[1] = ypr[1]*(180/M_PI);
     ypr[2] = ypr[2]*(180/M_PI);

     if (init_recibido && !imu_lista){

      if(iteraciones>2000){
        referencias[0] = ypr[0];
        imu_lista = true;
      }

      iteraciones++;

     } else {

       /* Calculamos errores y acciones de control */

       for(i=0;i<3;i++){
         errores[i] = referencias[i]-ypr[i];
         derivada[i] = errores[i]-errores_ant[i];
         if (imu_lista) I[i] += errores[i];
       }

       //accion_control[0]                                                =
1.000000000000*(errores[0]+0.00001*T*I[0]+(1/T)*(errores[0]-errores_ant[0]));
       accion_control[0] = 0;
       accion_control[1]                                                 =
1.75000000000*(errores[1]+0.0075*I[1]+0.0005*(0.02*derivada[1]-
0.98*derivada_ant[1]));
       accion_control[2]                                                 =
1.5000000000*(errores[2]+0.005*I[2]+0.0005*(0.1*derivada[2]-
0.90*derivada_ant[2]));

       for (i = 0; i<3; i++){
        errores_ant[i] = errores[i];
        derivada_ant[i] = derivada[i];
       }

      /* Enviamos señal PWM */

      loop_timer = micros();

      digitalWrite(MOTOR1, HIGH); //Motor 1 HIGH
      digitalWrite(MOTOR2, HIGH); //Motor 2 HIGH
      digitalWrite(MOTOR3, HIGH); //Motor 3 HIGH
      digitalWrite(MOTOR4, HIGH); //Motor 4 HIGH

      if (imu_lista){

        accion_m1 =  1300 + accion_control[2] + accion_control[1] + loop_timer;
// Calculamos el tiempo que han de estar a nivel alto las señales PWM
        accion_m2 =  1300 - accion_control[2] + accion_control[1] + loop_timer;
        accion_m3 =  1300 - accion_control[2] - accion_control[1] + loop_timer;
        accion_m4 =  1300 + accion_control[2] - accion_control[1] + loop_timer;

      } else {

        accion_m1 =  1000 + loop_timer;   // Mantenemos apagados los motores
        accion_m2 =  1000 + loop_timer;
        accion_m3 =  1000 + loop_timer;
        accion_m4 =  1000 + loop_timer;
```

```
    }

    aM1 = true; aM2 = true; aM3 = true; aM4 = true;

    while (aM1 || aM2 || aM3 || aM4 == true) {

      esc_loop_timer = micros();
      if (accion_m1 <= esc_loop_timer) { // Motor LOW
        aM1 = false;
        digitalWrite(MOTOR1, LOW);
      }

      esc_loop_timer = micros();
      if (accion_m2 <= esc_loop_timer) { // Motor 2 LOW
        aM2 = false;
        digitalWrite(MOTOR2, LOW);
      }

      esc_loop_timer = micros();
      if (accion_m3 <= esc_loop_timer) { // Motor 3 LOW
        aM3 = false;
        digitalWrite(MOTOR3, LOW);
      }

      esc_loop_timer = micros();
      if (accion_m4 <= esc_loop_timer) { // Motor 4 LOW
        aM4 = false;
        digitalWrite(MOTOR4, LOW);
      }

    }

  }

  /* Leemeos el dato recibido por BT (si lo hubiera) */

  if (BT.available()){
    mensaje = BT.read();
    switch(mensaje){
      case 'A':
        referencias[1] = 0;
        referencias[2] = 0;
        break;
      case 'B':
        referencias[2]+= 1;
        break;
      case 'D':
        referencias[1]-= 1;
        break;
      case 'E':
        referencias[2]-= 1;
        break;
      case 'F':
       referencias[1]+= 1;
        break;
      case 'H':
        if (altitud<200) altitud+=50;
        break;
      case 'I':
        if (altitud>0) altitud-=50;
        break;
       case 'J':
        init_recibido = true;
```

```
          break;
        case 'K':
          fin_recibido = true;
          break;
      }
    }

    /* Enseñamos resultados para debugging */

    if (imu_lista){
       Serial.print(ypr[0],2);
       Serial.print("\t");
       Serial.println(referencias[0],2);
    }

  }

}

if (mpuInterrupt && fin_recibido && j<500) {

    /* Enviamos señal PWM */

    loop_timer = micros();

    digitalWrite(MOTOR1, HIGH); //Motor 1 HIGH
    digitalWrite(MOTOR2, HIGH); //Motor 2 HIGH
    digitalWrite(MOTOR3, HIGH); //Motor 3 HIGH
    digitalWrite(MOTOR4, HIGH); //Motor 4 HIGH

    accion_m1 =  1000 + loop_timer;        // Mantenemos apagados los motores
    accion_m2 =  1000 + loop_timer;
    accion_m3 =  1000 + loop_timer;
    accion_m4 =  1000 + loop_timer;

    aM1 = true; aM2 = true; aM3 = true; aM4 = true;

    while (aM1 || aM2 || aM3 || aM4 == true) {

      esc_loop_timer = micros();
      if (accion_m1 <= esc_loop_timer) { // Motor LOW
        aM1 = false;
        digitalWrite(MOTOR1, LOW);
      }

      esc_loop_timer = micros();
      if (accion_m2 <= esc_loop_timer) { // Motor 2 LOW
        aM2 = false;
        digitalWrite(MOTOR2, LOW);
      }

      esc_loop_timer = micros();
      if (accion_m3 <= esc_loop_timer) { // Motor 3 LOW
        aM3 = false;
        digitalWrite(MOTOR3, LOW);
      }

      esc_loop_timer = micros();
      if (accion_m4 <= esc_loop_timer) { // Motor 4 LOW
        aM4 = false;
        digitalWrite(MOTOR4, LOW);
      }
```

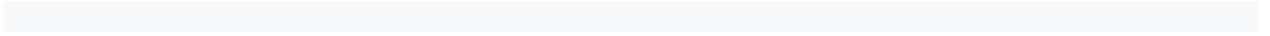
```
    }

    j++;

  }

  if (mpuInterrupt && fin_recibido && j>=500) {
    Serial.println("FIN");
    exit(0);
  }

}
```



14 REFERENCIAS

[1] 23 said: D. 18, bonsitm said: D. 18, jestrada said: A. 22, bonsitm said: A. 22, amba said: J. 7, bonsitm said: J. 7, amba said: J. 8, bonsitm said: J. 8, S. said: J. 14, bonsitm said: J. 15, gabriel said: J. 29, N. P. said: J. 19, jamessmithjs said: J. 28, john said: F. 12, bonsitm said: F. 13, madan said: M. 7, A.-I. said: M. 21, madan said: M. 22, maddy said: J. 19, N. said: M. 21, miera said: M. 23, and M Dio Khairunnas (@mdiokha) said: April 21, "RPi: HC-SR04 Ultrasonic Sensor mini-project," try { work(); } finally { code(); }, 16-Jul-2013. [Online]. Available: https://ninedof.wordpress.com/2013/07/16/rpi-hc-sr04-ultrasonic-sensor-mini-project/. [Accessed: 31-Aug-2020].

[2] "Arduino Uno Rev3," Arduino Uno Rev3 | Arduino Official Store. [Online]. Available: https://store.arduino.cc/arduino-uno-rev3. [Accessed: 31-Aug-2020].

[3] "Beginner's Guide to IMU," Beginner's Guide to IMU | Robotics Club. [Online]. Available: http://students.iitk.ac.in/roboclub/2017/12/21/Beginners-Guide-to-IMU.html. [Accessed: 31-Aug-2020].

[4] "berry120berry120 10.2k1010 gold badges4848 silver badges6262 bronze badges, Mark BoothMark Booth 4, Alex ChamberlainAlex Chamberlain 14.4k1111 gold badges6161 silver badges111111 bronze badges, joanjoan 59k44 gold badges5555 silver badges8989 bronze badges, JohnJohn 2111 bronze badge, and GlantucanGlantucan 12122 bronze badges, "Can I use the GPIO for pulse width modulation (PWM)?," Raspberry Pi Stack Exchange, 01-Nov-1961. [Online]. Available: https://raspberrypi.stackexchange.com/questions/298/can-i-use-the-gpio-for-pulse-width-modulation-pwm. [Accessed: 31-Aug-2020].

[5] "BL2210/30 - 1300KV," Emax BL2210/30. [Online]. Available: http://www.hiperhobby.com/BL2210-30.html&osCsid=s1hbbr84fdefbp1sk9ndabcf50. [Accessed: 31-Aug-2020].

[6] "BMP180 Digital pressure sensor." [Online]. Available: https://media.digikey.com/pdf/Data Sheets/Bosch/BMP180.pdf. [Accessed: 31-Aug-2020].

[7] "Configuración del módulo bluetooth HC-06 usando comandos AT," Inicio. [Online]. Available: https://naylampmechatronics.com/blog/15_Configuración--del-módulo-bluetooth-HC-06-usa.html. [Accessed: 31-Aug-2020].

[8] A. Cotes, "[Free] UDP client extension," Community, 27-Jul-2017. [Online]. Available: https://community.thunkable.com/t/free-udp-client-extension/5831. [Accessed: 31-Aug-2020].

[9] "Download Raspberry Pi OS for Raspberry Pi," Raspberry Pi, 24-Aug-2020. [Online]. Available: https://www.raspberrypi.org/downloads/raspbian/. [Accessed: 31-Aug-2020].

[10] Fivdi, "Hardware PWM · Issue #9 · fivdi/pigpio," GitHub. [Online]. Available: https://github.com/fivdi/pigpio/issues/9. [Accessed: 31-Aug-2020].

[11] "Guangzhou HC Information Technology Co., Ltd. Product Data ..." [Online]. Available: https://www.olimex.com/Products/Components/RF/BLUETOOTH-SERIAL-HC-06/resources/hc06.pdf. [Accessed: 31-Aug-2020].

[12] Jav, Ricardo, P. P. Morales, Comunicacionclr, J. Montes, and Anthony, "Diferencias entre motores con escobillas y brushless," Blog CLR. [Online]. Available:

https://clr.es/blog/es/diferencias-motores-con-escobillas-brushless/. [Accessed: 31-Aug-2020].

[13] “LM2574/LM2574HV SIMPLE SWITCHER® 0.5-A Step-Down Voltage ...” [Online]. Available: https://www.ti.com/lit/ds/symlink/lm2574.pdf. [Accessed: 31-Aug-2020].

[14] Luis, “El bus SPI en Arduino,” Luis Llamas, 30-Oct-2017. [Online]. Available: https://www.luisllamas.es/arduino-spi/. [Accessed: 31-Aug-2020].

[15] Luis, “Medir la inclinación con IMU, Arduino y filtro complementario,” Luis Llamas, 30-Oct-2017. [Online]. Available: https://www.luisllamas.es/medir-la-inclinacion-imu-arduino-filtro-complementario/. [Accessed: 31-Aug-2020].

[16] Luis, “Cómo usar un acelerómetro en nuestros proyectos de Arduino,” Luis Llamas, 30-Oct-2017. [Online]. Available: https://www.luisllamas.es/como-usar-un-acelerometro-arduino/. [Accessed: 31-Aug-2020].

[17] Luis, “El bus I2C en Arduino,” Luis Llamas, 02-May-2020. [Online]. Available: https://www.luisllamas.es/arduino-i2c/. [Accessed: 31-Aug-2020].

[18] Luis, “Determinar la orientación con Arduino y el IMU MPU-6050,” Luis Llamas, 08-Mar-2020. [Online]. Available: https://www.luisllamas.es/arduino-orientacion-imu-mpu-6050/. [Accessed: 31-Aug-2020].

[19] Luise, “Configurar I2C en la Raspberry Pi,” Norsip Soluciones I D, 12-Jan-2015. [Online]. Available: https://blog.norsip.com/2015/01/12/configurar-i2c-en-la-raspberry-pi/. [Accessed: 31-Aug-2020].

[20] “Main Page,” Xenomai. [Online]. Available: https://xenomai.org/documentation/xenomai-3/html/xeno3prm/index.html. [Accessed: 31-Aug-2020].

[21] Ming DingMing Ding 9711 silver badge66 bronze badges and joanjoan 59k44 gold badges5555 silver badges8989 bronze badges, “What's the difference between soft PWM and PWM,” Raspberry Pi Stack Exchange, 01-Dec-1968. [Online]. Available: https://raspberrypi.stackexchange.com/questions/100641/whats-the-difference-between-soft-pwm-and-pwm. [Accessed: 31-Aug-2020].

[22] J. Montero, “PXFmini: cómo crear un dron autónomo con RaspBerry Pi 3,” ToDrone. [Online]. Available: https://www.todrone.com/pxfmini-como-crear-dron-raspberry-pi-3/. [Accessed: 31-Aug-2020].

[23] F. Moya, “Comunicaciones I2C,” Comunicaciones I2C · Taller de Raspberry Pi. [Online]. Available: https://franciscomoya.gitbooks.io/taller-de-raspberry-pi/content/es/elems/i2c.html. [Accessed: 31-Aug-2020].

[24] “MPU6050 (Accelerometer Gyroscope) Interfacing with Raspberry Pi: ..,” ElectronicWings. [Online]. Available: https://www.electronicwings.com/raspberry-pi/mpu6050-accelerometergyroscope-interfacing-with-raspberry-pi. [Accessed: 31-Aug-2020].

[25] pigpio library. [Online]. Available: http://abyz.me.uk/rpi/pigpio/. [Accessed: 31-Aug-2020].

[26] “Raspberry pi 2,3 - Deploy xenomai kernel,” Simple Robot. [Online]. Available: http://www.simplerobot.net/2018/06/rpi23-deploy-xenomai-kernel.html. [Accessed: 31-Aug-2020].

[27] “Raspberry Pi PWM Generation using Python and C: Raspberry Pi,” ElectronicWings. [Online]. Available: https://www.electronicwings.com/raspberry-pi/raspberry-pi-pwm-generation-using-python-and-c. [Accessed: 31-Aug-2020].

[28] reglisse44 and Instructables, "The Drone Pi," Instructables, 05-Oct-2017. [Online]. Available: https://www.instructables.com/id/The-Drone-Pi/. [Accessed: 31-Aug-2020].

[29] Shawn and S. Blake, "Change Raspberry Pi I2C Bus Speed," Raspberry Pi Spy, 01-May-2020. [Online]. Available: https://www.raspberrypi-spy.co.uk/2018/02/change-raspberry-pi-i2c-bus-speed/. [Accessed: 31-Aug-2020].

[30] Teach, Learn, and Make with Raspberry Pi – Raspberry Pi. [Online]. Available: https://www.raspberrypi.org/. [Accessed: 31-Aug-2020].

[31] Tr4nsduc7or, "Tutorial de Arduino y MPU-6050," robologs, 15-Oct-2014. [Online]. Available: https://robologs.net/2014/10/15/tutorial-de-arduino-y-mpu-6050/. [Accessed: 31-Aug-2020].

[32] user79387user79387, iman ansariiman ansari 2122 bronze badges, larskslarsks 56911 gold badge33 silver badges1616 bronze badges, and John SJohn S 33911 silver badge66 bronze badges, "How to generate accurate PWM signals," Raspberry Pi Stack Exchange, 01-Jun-1967. [Online]. Available: https://raspberrypi.stackexchange.com/questions/77774/how-to-generate-accurate-pwm-signals. [Accessed: 31-Aug-2020].

[33] Walter, Arduproject, Joto, Lourdes, Diego, Daniel, Abde, Johan, and Jordi, "Drone Arduino: Conceptos generales sobre drones," ArduProject.es, 06-Jan-2019. [Online]. Available: https://arduproject.es/conceptos-generales-sobre-drones/. [Accessed: 31-Aug-2020].

[34] With MIT App Inventor, anyone can build apps with global impact," MIT App Inventor | Explore MIT App Inventor. [Online]. Available: http://appinventor.mit.edu/. [Accessed: 31-Aug-2020].

[35] R. Iglesias e I. Alvarado, «Control de posición y trayectoria de una pelota en un sistema tipo Ball And Plate», TFM, Departamento de Sistemas y Automática, US, Sevilla, 2019.

[36] D. Puertas y R. J. Calom, «DronePi: Construccion de un dron basado en Raspberry Pi», TFG, UPV, Valencia, 2016.

15 GLOSARIO

ADC: Analog to Digital Converter	13
BEC: Battery Eliminator Circuit	17
DMP: Digital Motion Processor	67
ESC: Electronic Speed Controller	16
GPIO: General Purpose Input/Output	13
I2C: Inter-integrated Circuit	13
IMU: Inertial Measurement Unit	12
PC: Personal Computer	45
PCB: Printed Circuit Board	23
PI: Proporcional Integral	1
PLA: Ácido Poliláctico	6
RF: Radiofrecuencia	37
RPi: Raspberry Pi	17
RTOS: Real Time Operating System	13
SO: Sistema Operativo	44
SPI: Serial Peripheral Interface	13
UART: Universal Asynchronous Receiver-Transmitter	13