UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Procesado y almacenamiento en servidor remoto en tiempo real de los datos de vuelo para aeronaves radiocontrol

Trabajo Fin de Grado

Grado en Ingeniería Aeroespacial

AUTOR/A: Cantarino Barberá, Carlos

Tutor/a: Alcañiz Fillol, Miguel

Cotutor/a: Masot Peris, Rafael

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Procesado y almacenamiento en servidor remoto en tiempo real de datos de vuelo para aeronaves radiocontrol

Trabajo de Fin de Grado

Grado en Ingeniería Aeroespacial

Autor: Carlos Cantarino Barberá

Tutor: Miguel Alcañiz Fillol

Cotutor: Rafael Masot Peris

Curso 2022/23

Resumen

Al operar una aeronave de control remoto, es de gran utilidad e incluso indispensable conocer su posición, velocidad y altitud, entre otros. Dependiendo de la misión, esta información puede ser requerida tanto durante como después del vuelo, es decir, es necesario no solo procesar y proveer al piloto de un continuo de información sino también almacenarlos.

En este Trabajo de Fin de Grado se estudian alternativas para lograr desarrollar un sistema de procesado, monitorizado y guardado de datos en tiempo real que sea apto en diversos escenarios. Junto a ofrecer medios de visualizado en teléfono móvil y PC, un pilar fundamental es implementar el almacenamiento y visualización de los datos en servidor remoto por sus beneficios únicos: independencia de almacenamiento local, disposición inmediata global o análisis de datos en paralelo y en remoto, entre otros.

Estas ventajas se traducen en no tener que lidiar con unidades de memorias físicas, poder trabajar de forma más eficiente, cumplir con diferentes responsabilidades sin estar presente en el lugar del vuelo del UAV o verificar (por ejemplo, un cliente) que la misión o servicio se está realizando. Las plataformas de móvil y PC, por su parte, completan la accesibilidad de los operadores a los datos.

Con el fin de demostrar sus capacidades, la explicación detallada del sistema se pone en práctica con un prototipo real compuesto por microcontroladores, sensores y comunicaciones propias. El resultado final muestra el amplio rango de su aplicabilidad al tener funciones como simulación de instrumentos aeronáuticos, seguimiento de la posición de la aeronave, compartición de los datos en la red o almacenamiento local y remoto.

Abstract

When operating a remote-controlled aircraft, it is of great utility and even essential to know its position, speed, and altitude, among other parameters. Depending on the mission, this information may be required both during and after the flight, meaning that it is necessary not only to process and provide the pilot with a continuous stream of information but also to store it.

In this Final Degree Project, alternatives are studied to develop a real-time data processing, monitoring, and storage system that is suitable for various scenarios. In addition to providing visualization capabilities on mobile phones and PCs, a fundamental pillar is the implementation of remote server storage and visualization of the data due to its unique benefits: independence from local saves, immediate global availability, parallel and remote data analysis, among others.

These advantages translate into not having to deal with physical memory units, working more efficiently, fulfilling different responsibilities without being present at the UAV's flight location, or verifying (for example, by a client) that the mission or service is being carried out. Mobile and PC platforms, on the other hand, complete the operators' accessibility to the data.

To demonstrate its capabilities, a detailed explanation of the system is put into practice with a real prototype consisting of microcontrollers, sensors, and proprietary communications. The final result shows the wide range of its applicability, with functions such as aeronautical instrument simulation, aircraft position tracking, data sharing on the network, and local and remote storage.

Índice

MEMORIA **9**
1. Objeto 10
2. Estudio de necesidades 12
3. Estudio de soluciones 16
   3.1 Microcontrolador 17
   3.2 Sensores 19
   3.3 Comunicación emisor-receptor 21
   3.4 Plataformas de monitorizado y almacenamiento 23
   3.5 Alimentación 26
4. Solución adoptada 28
   4.1 Emisor 29
      4.1.1 Hardware 29
      4.1.2 Software 32
   4.2 Receptor 36
      4.2.1 Hardware 36
      4.2.2 Software 36
   4.3 Servidor remoto 37
      4.3.1 Código en ESP32 37
      4.3.2 Visualizado con ThingSpeak 40
      4.3.3 Guardado en local con MATLAB 43
   4.4 Aplicación MATLAB 46
      4.4.1 Código en ESP32 46
      4.4.2 Aplicación de MATLAB con App Designer 48
   4.5 Aplicación móvil 55
      4.5.1 Código en ESP32 55
      4.5.2 Aplicación móvil con Kodular 59
5. Validación 72
      5.1.1 Condiciones de la prueba 72
      5.1.2 Resultados 73
6. Conclusiones 80
7. Bibliografía 83

PRESUPUESTO **85**
1. Coste material 86
2. Coste de software 86
3. Coste humano 87
4. Resumen 88

OBJETIVOS DE DESARROLLO SOSTENIBLE 90

PLIEGO DE CONDICIONES 93

1. Normas y procedimientos de UAVs 94
2. Derechos y obligaciones del trabajador 100

ANEXOS 107

A. Código completo 108

A.1 Microcontrolador emisor 108

A.2 Microcontrolador receptor 114

A.3 Aplicación de MATLAB 121

A.4 Visualizador de ThingSpeak: Mapa 125

A.5 Visualizador de ThingSpeak: Rumbo 126

A.6 Guardado local con MATLAB de datos de ThingSpeak 127

A.7 Guardado local con MATLAB de vuelos de ThingSpeak 128

A.8 Mapeado 2D y 3D con MATLAB de vuelos guardados 130

B. Fichas técnicas 133

MEMORIA

1. OBJETO

El uso de aeronaves no tripuladas o UAVs [1] (*unmanned aerial vehicles*) tales como cuadricópteros o aviones radiocontrol se encuentra en un crecimiento continuo en el ámbito civil. Con incontables aplicaciones en áreas como transporte de mercancías, agricultura o cartografía, su empleo cada vez más apartado del ocio hace necesario garantizar que estas máquinas cumplen su función, es decir, que son fiables.

En aviación comercial, durante el vuelo, la tripulación es la encargada de realizar un seguimiento continuo de los parámetros relevantes con el objetivo de detectar cualquier anomalía. Indirectamente, la fiabilidad se logra también antes y después de los vuelos con el análisis de estos parámetros y otros muchos. Gracias a ello, se puede estimar la probabilidad de fallo a lo largo de la vida de casi cualquier elemento y actuar en consecuencia a través del mantenimiento.

De este modo, los datos de vuelo hacen falta no solo en tiempo real pero también a posteriori. En cuanto a aeronaves radiocontrol, existen varios modelos comerciales que permiten algo similar, almacenando y enviando al piloto los datos que recogen sus sensores integrados. Sin embargo, estos suelen ser productos costosos y, sobre todo, poco personalizables.

De hecho, normalmente es la libertad de personalización la que inclina a particulares y empresas a fabricar diseños propios que se adecúen al propósito del UAV. Es posible que termine habiendo poca diferencia en el coste, pero el dinero se invierte de una forma mucho más eficiente ya que solo se incluye lo imprescindible y necesario. Por tanto, la pregunta que se ha de formular es cómo de importante es el manejo de datos en cada caso.

La respuesta sencilla es que, idealmente, los operarios de un UAV deberían de disponer de toda la información posible siempre, lo que hace indispensable un sistema de almacenamiento y envío de datos. A pesar de ello, muchas de estas aeronaves no lo poseen y se acaba dependiendo en gran medida de la experiencia personal. Variables como velocidad y altitud se han de estimar visualmente durante el vuelo, mientras que el mantenimiento será en su gran mayoría correctivo, es decir, se actúa tras el fallo.

Adicionalmente, la idea de disponer de la máxima información se puede extender a todos los involucrados en la misión desempeñada por el UAV. No obstante, dicho grupo puede ser muy amplio, lo que facilita que haya múltiples individuos que no se encuentren ubicados en el lugar del vuelo. Estas personas, incluso con la aeronave siendo capaz de transmitir datos, no tienen posibilidad de recibirlos en tiempo real.

El problema nace de que, en el contexto descrito, la comunicación entre aeronave y piloto es parte de un sistema cerrado o aislado. Como consecuencia, no solo la información está restringida a unos pocos, sino que fallos en la integridad de la misma pueden ser irremediables al guardarse localmente. Afortunadamente, la solución es conocida por todos: Internet. Con conexión a la red, la mejor opción es compartir los datos globalmente subiéndolos y almacenándolos en un servidor remoto, comúnmente conocida como la nube.

Frente a la inexistencia de un sistema que monitorice en tiempo real el UAV a la vez que guarde la información en la nube, el presente documento estudia en profundidad las principales causas de dicha realidad, así como sus posibles soluciones. Además, con el fin de demostrar la aplicabilidad real de las propuestas, se ha desarrollado un conjunto de software y hardware, cuya composición y funcionamiento también se detallan.

[1] Un UAV abarca tanto aeronaves autónomas como remotamente pilotadas. Este último grupo se conoce como RPA (*remotely controlled aircraft*) y es en el que se centra el documento, pero por extensión y popularidad del término se usará la denominación “UAV”. Además, este trabajo no cierra las puertas a aeronaves autónomas.

2. ESTUDIO DE NECESIDADES

En esta sección, se analizarán los requisitos de un sistema de monitorización y almacenamiento de datos de vuelo para un UAV. Como punto de partida, se ha de tratar de entender cuáles son los principales inconvenientes de incluir sistemas así. De una forma amplia, se pueden identificar tres: peso, coste y accesibilidad.

El peso y el coste son problemas conocidos en la aviación, y al referirnos a aeronaves radiocontrol estos cobran especial importancia debido a su naturaleza. En primer lugar, su menor tamaño reduce las cargas aerodinámicas a soportar, permitiendo emplear materiales menos resistentes y más ligeros, pero obliga en la mayoría de los casos a usar fuentes de energía eléctricas gracias a su sencillez, facilidad de uso, mantenimiento mínimo, seguridad…

La cantidad de energía que pueden proveer respecto a su masa es el problema: baterías muy comunes en este campo como las LiPo tienen una densidad energética de alrededor de 200 Wh/kg, mientras que la de combustibles de aviación como Jet-A (reacción) o AVGAS (combustión interna) es de un orden de magnitud 100 veces mayor. Por tanto, si se busca una autonomía de vuelo o carga útil elevadas, el peso que hay que dedicar en baterías puede hacer indeseable implementar el sistema de datos de vuelo si no es lo suficientemente ligero.

Asimismo, es crucial tener en cuenta el coste tanto energético (consumo) como económico de un sistema así. La importancia de este último es debida a la baja inversión económica que por lo general suponen los UAVs, sobre todo gracias al ahorro en materia prima y fabricación. Como consecuencia, cualquier añadido con un precio aparentemente asequible puede ser demasiado caro en comparación con el coste de la aeronave en sí.

El último aspecto clave es la accesibilidad de la solución, la cual se puede resumir en dos características: sencillez y versatilidad. Por un lado, el sistema se ha de poder aplicar e implementar sin la necesidad de unos conocimientos demasiado avanzados en ámbitos como electrónica o programación. Por otro lado, tiene que ser fácilmente modificable, además de apto para una amplia diversidad de condiciones de uso, es decir, que se considere por ejemplo la conectividad de la zona o los dispositivos disponibles (móvil, ordenador portátil…).

Una vez expuestos globalmente los requisitos principales, es necesario concretar las funcionalidades que el sistema de manejo de datos de vuelo tiene que incluir. No cabe duda de que una de las decisiones fundamentales a tomar es qué datos recoger y enviar. Los parámetros más adecuados variarán de acuerdo con la misión del UAV; sin embargo, existen una serie de datos que se podrían determinar básicos y que son útiles en todo caso:

- Velocidad: admite múltiples definiciones y todas son igualmente válidas y útiles siempre y cuando no haya duda de con cuál se está trabajando. No hay que confundir:
  - Velocidad terrestre, que es la componente horizontal de la velocidad respecto del suelo y la proporciona un GPS[2]. Es análoga a, por ejemplo, la velocidad de un coche, y sirve para conocer el tiempo que se tardará en hacer un recorrido.
  - Velocidad aerodinámica, que es respecto al aire y se mide normalmente con uno o varios tubos de Pitot (su orientación determina la dirección de la velocidad medida) y puertos estáticos. En aeronaves de ala fija es clave pues es una medida del flujo de aire alrededor del ala, sirviendo de indicativo para evitar una entrada en pérdida o exceso de velocidad. Por relatividad, un viento suficientemente

[2] A lo largo del documento, "GPS" denotará cualquier sistema de posicionamiento por satélite. Técnicamente, el GPS (*Global Positioning System*) está patentado en Estados Unidos y solo es el más conocido de los existentes, siendo el Galileo europeo o el GLONASS ruso otros equivalentes.

fuerte sería capaz de generar sustentación sin que la aeronave se desplace, por lo que esta estaría aparentemente suspendida en el aire.

- Velocidad inercial, que se obtiene a partir de las aceleraciones medidas por una unidad de medición inercial o IMU. Normalmente, la IMU se alinea con los ejes de rotación (cabeceo, alabeo y guiñada) de la aeronave, obteniendo así las aceleraciones en un sistema de referencia consistente. La desventaja es que la velocidad se calcula integrando, lo que da lugar a errores acumulativos.

- Altitud: se define como la distancia vertical entre la aeronave y el nivel del mar. Para aeronaves UAV, las cuales suelen volar bajo, la altura (distancia al suelo) puede ser más conveniente, por lo que ser capaz de enviarla es indispensable.

- Temperatura: un sensor de temperatura es capaz de obtener información diferente dependiendo de dónde se coloque. Puede ser especialmente útil cerca del motor u otros componentes eléctricos y electrónicos para detectar sobrecalentamiento.

- Alabeo, cabeceo y guiñada (ver Figura 2.1): definen la orientación de la aeronave y se pueden conocer de forma precisa con una IMU, pues esta no solo incluye acelerómetros sino también giróscopos. Hacen falta en dos instrumentos básicos: horizonte artificial (alabeo y cabeceo) e indicador de rumbo (guiñada). Este último provee la misma información que una brújula, pero es más fiable por no verse afectado por fluctuaciones magnéticas y ser más robusto en condiciones adversas.

- Latitud y longitud: vienen dadas por un GPS y marcan la posición de la aeronave. Tomando las coordenadas con seis decimales, la precisión es de aproximadamente 10 centímetros, aunque los GPS más comunes tienen una precisión de escasos metros.

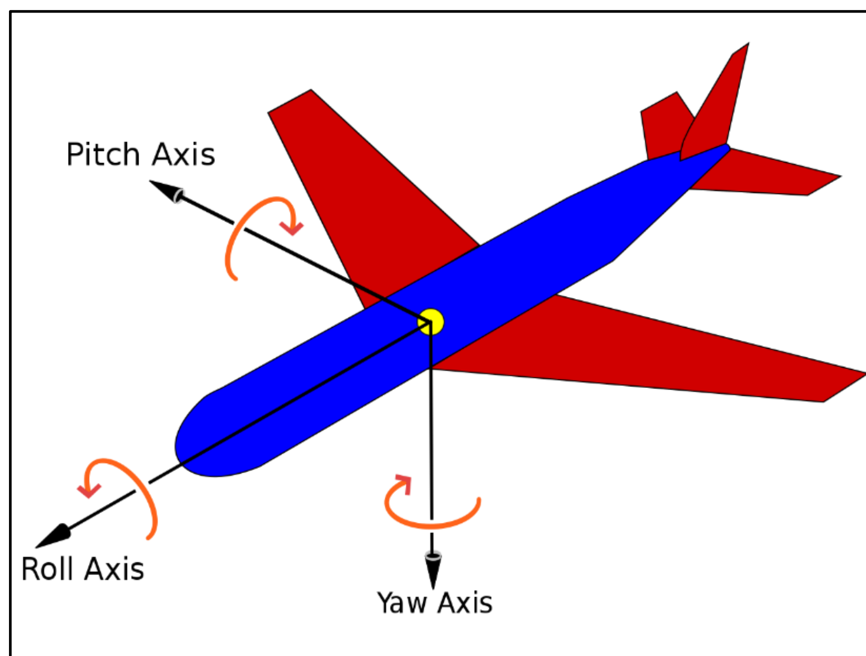


Figura 2.1: Ejes de rotación de un avión: alabeo (roll), cabeceo (pitch) y guiñada (yaw). Por Auawise, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9441238

Es recomendable también disponer de cierta redundancia sobre todo en parámetros básicos como velocidad y altitud, es decir, que dos sensores diferentes proporcionen la misma información. Los sensores que recojan todos los datos se discutirán en la siguiente sección, donde se valorarán diversos factores de acuerdo con las necesidades presentadas. Estos dispositivos tendrán que conectarse a un "cerebro" que reciba, interprete y procese los datos, cumpliendo con los requisitos de peso, coste y consumo. Dicho "cerebro" se conoce como microcontrolador y está compuesto de un procesador o CPU, una memoria y periféricos de entrada y salida, es decir, es básicamente un ordenador en miniatura ya que es programable.

La siguiente función a desempeñar por el sistema es la de permitir comunicación en tiempo real, la cual es una de las bases del proyecto. Lo más sencillo es añadir una memoria externa al microcontrolador y que guarde todos los datos; sin embargo, estos son inaccesibles durante el vuelo y pueden ser tediosos de extraer dependiendo del diseño. Enviar los datos desde el UAV a un receptor en tierra a medida que se recogen es la alternativa, la cual es compleja, pero de mayor utilidad por la disponibilidad inmediata: visualizado de instrumentos de vuelo, seguimiento del recorrido por GPS, posibilidad de trabajar con los datos en paralelo…

Por tanto, se ha de conseguir establecer una comunicación personalizada de larga distancia (cientos de metros o más) entre un receptor y el UAV, cuyo microprocesador actúe de transmisor. El receptor será también un microcontrolador con las mismas funciones: recibir, interpretar, procesar y enviar. Este último envío es sinónimo de poner los datos a disposición del usuario que, según su situación, preferirá una opción u otra. A continuación, se plantean tres situaciones o contextos con las que tratar de cubrir la gran mayoría de posibilidades.

En primer lugar, el caso ideal es que el usuario complete el sistema con un ordenador portátil ya que esta es la herramienta más potente. El microcontrolador receptor podría entonces conectarse directamente al ordenador y comunicarle toda la información recibida. Si la opción del portátil no fuera viable, siempre queda el teléfono móvil, un dispositivo que actualmente podemos suponer que siempre está a mano, pero mucho más limitado en hardware y software.

El inconveniente de estas opciones es que requieren de dispositivos externos y son locales, es decir, lo que se envía al portátil o al móvil no se mueve de allí a menos que se guarden y exporten manualmente. Es aquí cuando la disponibilidad y accesibilidad de un servidor remoto es imbatible gracias a que los datos se comparten globalmente en la red, maximizando la eficiencia y posibilidades de quienes trabajan con ellos. El objetivo es que se pueda acceder a este servidor a través del microcontrolador exclusivamente.

En la siguiente sección, se explorarán posibles métodos de visualizado y almacenamiento en las tres plataformas mencionadas: PC, *smartphone* y servidor remoto.

3. ESTUDIO DE SOLUCIONES

Como se ha visto en el estudio de necesidades, la solución ha de cubrir una serie de objetivos que se listan resumidamente a continuación:

- Minimizar coste, peso y complejidad
- Maximizar versatilidad y sencillez
- Recoger datos básicos de vuelo, en este caso velocidad, altitud, alabeo, cabeceo, guiñada, posición y temperatura
- Enviar datos según se toman para que el usuario disponga de ellos en tiempo real
- Hacer llegar los datos al usuario en tres plataformas: ordenador o PC, teléfono móvil y servidor remoto.
- Crear algún método de almacenamiento apropiado

En esta sección y sus subapartados se desarrollarán todos los elementos que, potencialmente, son capaces de lograr cumplir las mencionadas metas.

3.1 MICROCONTROLADOR

Para empezar, abarcaremos la selección del microcontrolador, para la cual existe un enorme abanico de donde elegir. Un elemento que ayudará a tomar la decisión es elegir primero el entorno de desarrollo integrado o IDE (*Integrated Development Environment*), es decir, qué programa usar para editar código, depurarlo y compilarlo, de forma que podamos introducir la lógica deseada en el microcontrolador.

El más extendido entre programadores es Visual Studio, de Microsoft, acumulando casi un 30% del total de búsquedas de IDEs en 2023 [1]. Sin embargo, este y otros similares están orientados a aplicaciones web y de escritorio, y para trabajar con microcontroladores se necesitan extensiones en el caso de Visual Studio.

Una opción más adecuada y que maximiza versatilidad y sencillez es el IDE de Arduino, compañía de software y hardware que diseña microcontroladores. Gracias a ser una plataforma de código abierto (diseños, firmware y software, entre otros, están disponibles de forma gratuita al público, y pueden ser manipulados), su popularidad no ha hecho más que crecer con usuarios que continuamente añaden contenido.

Gracias a esta comunidad, es casi imposible encontrar algún microcontrolador o periférico que no tenga una librería (conjunto de código y funciones predefinidas) que lo haga compatible con Arduino. Aun así, la plataforma incluye librerías oficiales que hacen conveniente elegir ciertos microcontroladores para tener facilidades, como es el caso de las placas de la familia Arduino o los ESP32 y ESP8266.

Para continuar descartando opciones, hay que adelantar que en la Sección 3.4 se establece la necesidad de comunicarse por Bluetooth y WiFi. Ambas se pueden generar con módulos externos, pero el objetivo es que estén integrados para reducir la complejidad e incluso el precio dependiendo del caso. Esto nos obliga a descartar placas Arduino como el Arduino Nano 33 BLE, que solo tiene Bluetooth, o el ESP8266, que solo tiene WiFi.

Dos placas que sí cumplen con todos los requisitos son el ESP32 y el Arduino Nano 33 IoT (ver Figura 3.1.1 y 3.1.2), siendo ambas opciones interesantes ya que presentan sus propias ventajas únicas. Por un lado, el Arduino Nano 33 IoT tiene un menor tamaño (18x45 mm respecto a 28x56 mm) y una IMU integrada, que en la próxima sección se comprobará que es algo muy útil. [2]

Por otro lado, el ESP32 es técnicamente superior con una mayor velocidad de procesamiento (hasta 240 MHz respecto a 48 MHz) y más memoria flash (4 MB respecto a 1 MB) entre otros. Además, permite conectar más periféricos con sus pines de entrada y salida visibles en la Figura 3.1.1 (34 respecto a 11) y presenta una variedad mayor de comunicaciones cableadas: UART (3 canales), SPI (2 canales), I2C (2 canales), I2S y CAN respecto a un único canal para UART, SPI e I2C. [3]

Un punto intermedio entre estas placas es el Arduino Nano ESP32 (ver Figura 3.1.3), que con el tamaño del Nano 33 IoT posee un canal UART adicional y las características técnicas de un ESP32, aunque no incluye la IMU. En cuanto a coste económico, es más barato que el Nano 33 IoT (18€ respecto a 22,80€ en la página oficial de Arduino), pero no tanto como el ESP32, que tiene un precio de 11,49€ en el distribuidor reconocido AZ-Delivery. Si consideramos la IMU, entonces el Nano 33 IoT es el más barato por tenerla integrada, pero en la sección siguiente se verá que este no es el único factor.

Finalmente, se puede concluir que por rendimiento, comunicaciones y coste el ESP32 es la mejor opción, aunque haya que sacrificar tamaño. El Arduino Nano 33 IoT sigue siendo una opción válida y compatible con la solución adoptada, aunque sería complicado añadirle otros sensores por sus comunicaciones limitadas y habría que estudiar el efecto que tiene sus menores capacidades técnicas. En cambio, el Arduino Nano ESP32 no es completamente compatible por el motivo que se explica en el apartado sobre sensores, donde también se propondrá una alternativa para poder seguir usándolo.

Nótese que hasta ahora se ha hablado de un único microcontrolador como solución, pero emisor y receptor cumplen funciones distintas. Por ejemplo, el problema del Nano ESP32 con los canales de comunicación no aplica al receptor ya que no lleva sensores conectados, y tampoco es necesario un microcontrolador con WiFi y Bluetooth para el emisor.

No obstante, emplear un solo tipo de dispositivo facilita la familiarización con el mismo, además de que un microcontrolador tan completo con el ESP32 admite mejoras y modificaciones futuras que en un principio podían no estar contempladas. Esto último ha sido aprovechado para añadir en la solución adoptada la posibilidad de cambiar ajustes por Bluetooth al emisor.

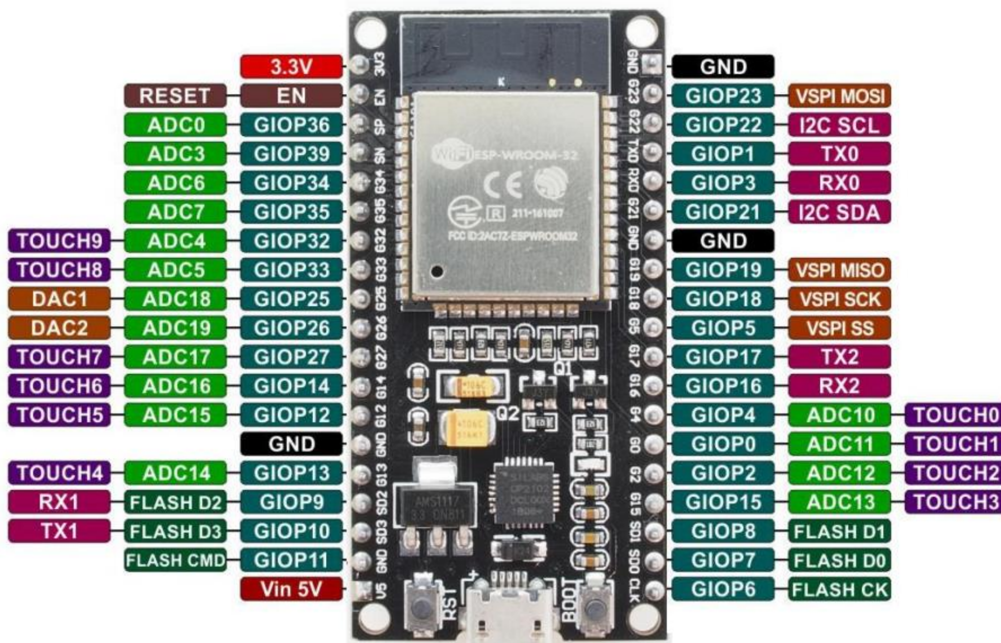


Figura 3.1.1: ESP32 con diagrama de pines entrada/salida.
Imagen del microcontrolador de AZ-delivery

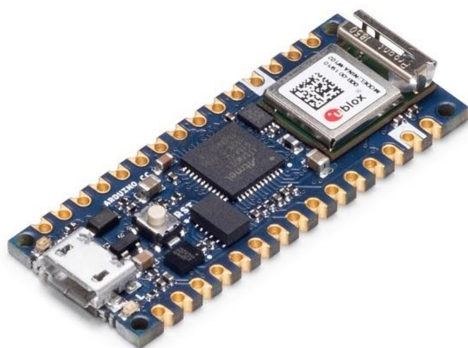


Figura 3.1.2: Arduino Nano 33 IoT ESP32.
Imagen de la tienda oficial de Arduino

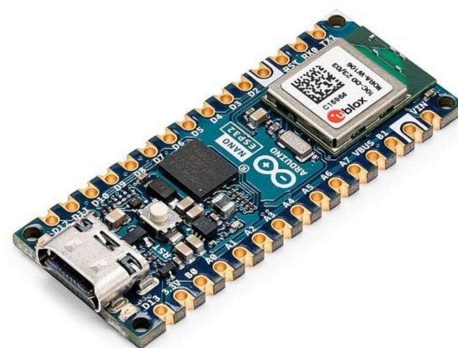


Figura 3.1.3: Arduino Nano ESP32.
Imagen de la tienda oficial de Arduino

3.2 SENSORES

En este apartado, se discutirán qué sensores llevar a bordo para recoger los datos deseados. Son muchas las combinaciones posibles de dispositivos que, conjuntamente, logran proporcionarlos, por lo que habrá que buscar la más adecuada. En un principio, el mínimo número de sensores parece la mejor opción, pero entonces aparecen inconvenientes como pérdida de redundancia.

Dos dispositivos imprescindibles son, como se adelantó previamente, una IMU y un GPS. Con sus funcionalidades más básicas obtenemos la orientación de la aeronave con el giróscopo de la IMU y la latitud, longitud y altitud del GPS. Además, con cálculos sencillos, se puede obtener velocidad tanto con la IMU (inercial) como con el GPS (terrestre), otorgando redundancia y haciendo innecesario instalar un sensor especializado como un tubo de Pitot.

Consecuentemente, el único parámetro sin sensor asociado es la temperatura, aunque en el estudio de necesidades se comentó la posibilidad de hacer redundante también la altitud. La forma más sencilla de conocer esta última sin perder fiabilidad es con un altímetro barométrico, el cual se basa en las diferencias de presión respecto a una presión de referencia. Para calibrarla y calcular altura en lugar de altitud, será necesario que se envíe también la presión medida para permitir al usuario establecerla como referencia.

Indirectamente, emplear un sensor de presión tiene la ventaja de que es extremadamente común que incluya un sensor de temperatura, la variable que falta. Esto es debido a que la presión es sensible a este valor, por lo que el propio sensor puede corregir desviaciones si dispone de él, como hacen el MS5611, el MPL3115A2, el DPS310 o los BMP180/280/388.

Afortunadamente, la existencia de tantos modelos y fabricantes es ideal para el consumidor, pues la inmensa mayoría son igual de competentes y de precio similar. En este caso, se han escogido el BMP280 (presión, temperatura y altitud barométrica), BNO055 (IMU: cabeceo, alabeo, rumbo y velocidad), y BN-880 (GPS: latitud, longitud, altitud y velocidad), visibles en las Figuras 3.2.1, 3.2.2 y 3.3.3, respectivamente. La disponibilidad ha sido la principal razón de la elección, pero antes se ha comprobado que sean compatibles con Arduino, aunque su gran comunidad hace improbable que no exista alguna librería.

Con respecto al BNO055 como IMU, aunque su precio es elevado y provoca que el ESP32 sea una opción más cara que el Arduino Nano 33 IoT, lo compensa con sus características técnicas. La orientación no solo es crítica para la aplicación, sino que es un problema muy difícil de resolver, y el BNO055 lo logra a través de un giroscopio, magnetómetro y acelerómetro interpretados por un procesador de alta velocidad. Esto justifica su coste y lo hace una de las mejores, si no la mejor, opción del mercado. [4]

Cabe mencionar que el BNO055 incluye un sensor de temperatura, por lo que si no se deseara redundancia en altitud se podría prescindir del BMP280. No obstante, este último es el más simple y de menor consumo, precio, tamaño y peso, por lo que no incluirlo tiene una ínfima aportación positiva mientras que incluirlo proporciona datos muy valiosos.

Retomando otro aspecto de la sección anterior, prescindir de redundancia sería obligatorio en caso de disponer de un Arduino Nano ESP32 como microcontrolador transmisor. Esto se debe a que tanto el BNO055 como el BMP280 usan comunicación SPI, de la cual este microcontrolador solo incluye un canal. Esto también le ocurre al Arduino Nano 33 IoT, pero al tener la IMU integrada, un único canal SPI para el BMP280 es suficiente.

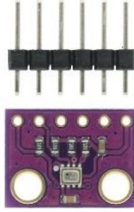


Figura 3.2.1: Sensor BMP280
Imagen de Fruugo

Figura 3.2.2: IMU BNO055
Imagen de Adafruit

Figura 3.2.3: GPS BN-880
Imagen de Beitian

3.3 COMUNICACIÓN EMISOR-RECEPTOR

El siguiente paso es decidir cómo se van a comunicar los datos entre emisor y receptor. La comunicación de la aeronave a tierra es la más crítica y ha de cumplir con los requisitos impuestos en la sección previa, principalmente personalizable, de largo alcance y con bajo consumo y coste.

Una solución es aprovecharse de que esta comunicación ya existe en todo UAV y se utiliza para pilotar: el transmisor está en tierra (los mandos) y el receptor se encuentra en la aeronave. Sin embargo, hay que recordar que el sentido del envío de datos que nos interesa es el opuesto: receptor en tierra y emisor en la aeronave. Por tanto, para que esta solución funcione se depende de que los mandos puedan actuar de receptores.

Esta capacidad de recepción es muy habitual no solo en aplicaciones complejas como vuelos FPV (*First Person View*, para los cuales el dron envía vídeo de una cámara en tiempo real, siendo esta cámara los ojos del piloto) sino también en usos simples como informar de que la conexión se ha establecido (esto es algo que un transmisor por sí solo no puede saber). El verdadero problema es las posibilidades de recepción que ofrece cada mando.

Por consiguiente, en el caso de que el mando disponible no sea suficiente, la solución deja de cumplir varios requisitos: un nuevo mando es costoso (ha de tener pantalla para el visualizado de datos) y va en contra de la sencillez (lo sencillo es poder seguir usando el mismo mando), además de que es poco versátil. Por ejemplo, la fabricante de mandos FlySky comercializa algunos sensores como el FS-CAT01 de altitud, pero son incompatibles con otros mandos y viceversa: sensores de otras marcas no funcionan con mandos FlySky.

Por último, aunque se posea el mejor mando posible y todos los sensores necesarios, sigue habiendo funcionalidades muy difíciles o directamente imposibles de implementar por las limitaciones de hardware y software, como es el almacenamiento o la comunicación con otras plataformas, en especial un servidor remoto. Tras analizar esta realidad, construir un transmisor a partir de sensores individuales y un microcontrolador se consolida como la mejor opción, pero hay que crear una comunicación propia.

En el campo de las telecomunicaciones, las redes inalámbricas de bajo consumo y largo alcance son relativamente recientes y se encuentran en un continuo desarrollo, recibiendo un nombre específico: LPWAN (*Low-Power Wide-Area Network*, o red de área amplia de bajo consumo). La Figura 3.3.1 muestra que esta es la más indicada para los objetivos de rango, consumo y coste con respecto a otras telecomunicaciones conocidas.

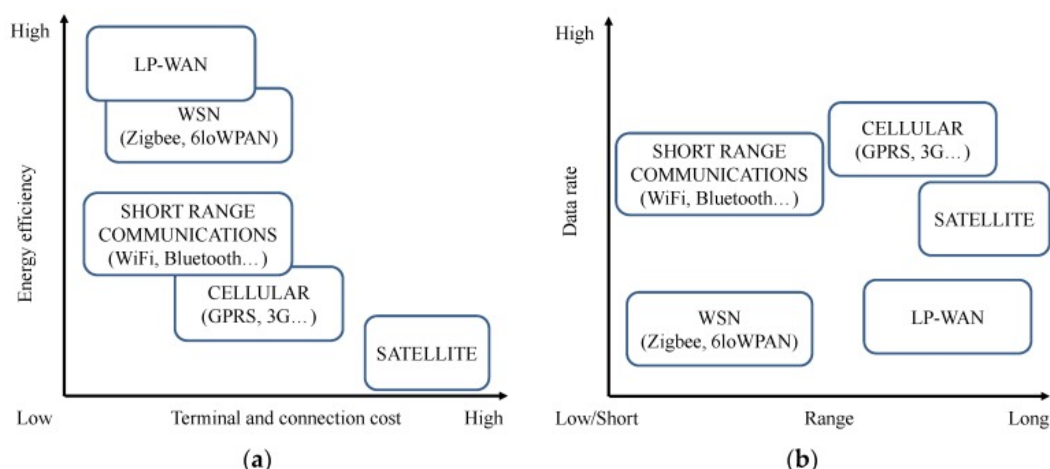


Figura 3.3.1: Comparación entre eficiencia energética y coste (a) y velocidad de datos y rango (b) de distintas telecomunicaciones [5]

Dentro de las plataformas LPWAN, existen multitud de opciones de donde elegir. No obstante, por la novedad de la tecnología, unas se encuentran en un estado de madurez superior que otras. Principalmente, son tres las que han conseguido un mayor impacto y crecimiento, destacando por encima de otras: LoRa, Sigfox y RPMA. Sus características fundamentales se recogen en la Tabla 3.3.1. [5] [6]

	LoRa	**Sigfox**	**RPMA**
Rango máximo rural (km)	10-15	30-50	~15
Rango máximo urbano (km)	3-5	3-10	
Velocidad máx. de subida (bps)	37500	100	8000
Tamaño máx. de paquete (bytes)	256	12	10000

Tabla 3.3.1: Comparativa de las características principales de tres plataformas LPWAN

Como puede observarse en la Tabla 3.3.1, cada plataforma presenta puntos fuertes y débiles en distintas áreas (ver colores: verde mejor, amarillo intermedio, rojo peor), por lo que la elección dependerá de la aplicación. Tratándose de UAVs que han de permanecer en el campo de visión del piloto, un rango relativamente bajo no supone un problema porque la limitación es la vista: para un objeto de 30 centímetros, la resolución angular del ojo (~0.017º) no permite verlo más allá de un kilómetro [7].

Para la comunicación en tiempo real, la velocidad de subida es un parámetro mucho más importante teniendo en cuenta que una aeronave es capaz de realizar movimientos a gran velocidad, y poder visualizar estos cambios en los instrumentos de forma inmediata es esencial. Asimismo, los datos se envían en paquetes, por lo que también es crítico estudiar el tamaño máximo de estos.

El método más indicado para enviar múltiples datos de naturaleza variable es unirlos en una sola cadena de texto (paquete) usando un carácter de separador, por ejemplo "&". Un carácter (número, separador decimal, signo negativo...) ocupa un byte, por lo que si suponemos que cada dato junto con su separador tiene una media de ocho caracteres (por ejemplo, "-12.345&"), con LoRa podemos enviar 32 parámetros, con RPMA, 1250, y con Sigfox no llegaría a 2.

Hay que tener en cuenta que no se pueden maximizar rango, velocidad de subida y tamaño de paquete a la vez. Sin embargo, las características estudiadas y sus valores máximos son una buena estimación, permitiendo descartar con confianza Sigfox por su velocidad de subida y tamaño de paquete incompatible con la aplicación.

A la hora de elegir entre LoRa y RPMA, este último parece mejor opción a primera vista basándonos en la Tabla 3.3.1. De hecho, en su página web se compara directamente con LoRaWan y SigFox, destacando sus fortalezas como una banda de frecuencia única y mayor cobertura global [8]. Sin embargo, estas son características de la comunicación en sí, la cual puede ser mejor técnicamente, pero la aplicabilidad en este caso es igual o más importante.

Es en este aspecto en el que LoRa es la ganadora indiscutible: la comunicación se compone de dos módulos (transmisor y receptor), los cuales están muy extendidos, son económicos, pequeños y ligeros, y no hay ninguna placa Arduino que sea incompatible con ellos. La tecnología RPMA está diseñada para empresas y la instauración de la red así como la adquisición de software se hace contactando a la empresa desarrolladora, Ingenu, por lo que su uso no tiene sentido en esta aplicación pensada especialmente para particulares.

3.4 PLATAFORMAS DE MONITORIZADO Y ALMACENAMIENTO

El último asunto por cubrir es la creación de la interfaz de usuario, en la cual los datos sean visibles y puedan ser almacenados. Respecto a la aplicación móvil y la de PC, lo relevante son las funcionalidades y aspectos técnicos que afectan al usuario. El entorno de desarrollo de estas aplicaciones es una decisión basada principalmente en preferencias y conocimientos previos, pues cualquiera de los posibles permite más o menos fácilmente llegar al mismo resultado final.

En este caso, para la aplicación móvil se eligió Kodular y para la de PC, App Designer de MATLAB. En cuanto a Kodular, su principal ventaja es la sencillez a su programación en bloques, opción muy visual y cómoda para desarrolladores con poca experiencia, además de ser completamente gratuita. Una alternativa equivalente y mucho más conocida es App Inventor MIT, pero la interfaz está muy anticuada y no recibe actualizaciones, haciendo trabajar en ella mucho menos deseable.

La elección de App Designer para desarrollar la aplicación de PC se debe en gran medida a una preferencia personal en base a la experiencia con el programa y el lenguaje de MATLAB. No obstante, en ella se encuentran opciones muy útiles para este proyecto, como es una sección de instrumentos aeronáuticos ya creados como altímetro u horizonte artificial, los cuales se pueden usar para simular el panel de instrumentos de la supuesta cabina del UAV.

Para lograr este mismo propósito en la app móvil, otros lenguajes de programación son más apropiados que el de Kodular, pero su curva de aprendizaje es mucho más lenta, haciéndolos inviables para alguien con tiempo limitado y escasos conocimientos en el campo. Por otro lado, también hay componentes predefinidos que agilizan mucho el proceso y que se tratarán en la solución adoptada (ver Sección 4.4.2).

De estas dos plataformas falta por discutir con qué método conectarlas al receptor. Debido a que, en principio, el receptor estará cerca del móvil o de PC, aparecen dos opciones interesantes: por cable o con telecomunicación de corto alcance (ver Figura 3.3.1). Para el móvil, sería necesario unir un puerto tipo micro-B (típico de microcontroladores como el ESP32) con el del móvil, que es muy variable (USB micro-B, USB-C, Lightning…) lo que supone un cable demasiado concreto, perdiendo varios objetivos propuestos como simplicidad, coste o versatilidad.

Eso nos deja con la telecomunicación de corto alcance, la cual en un dispositivo móvil convencional se limita a Bluetooth o WiFi. En este caso, el Bluetooth es la opción indiscutible ya que está mucho más orientado a este tipo de aplicaciones, haciéndolo fácil de configurar y conectar comparado con el WiFi. Lo mismo aplicaría para el PC si no fuera porque con él desaparecen los problemas del cable: con un extremo micro-B y que admita transmisión de datos sería suficiente, y estos son extremadamente comunes. Además, es de las conexiones más sencillas, así como veloces y robustas.

La última plataforma que queda por definir es el servidor remoto, para el cual existe una oferta muy amplia. De hecho, se han publicado multitud de estudios y artículos dedicados únicamente a explorar y comparar estas posibilidades, de donde destacan las plataformas desarrolladas por gigantes tecnológicos como Amazon (Amazon Web Services, AWS), Microsoft (Microsoft Azure), Google (Google Cloud) o IBM (IBM Watson IoT Cloud). [9] [10] [11]

Sin embargo, estas opciones disponen de tantos recursos que se vuelven difíciles de navegar, están muy orientados a empresas, y los precios son confusos, poco transparentes y sujetos a muchas variables. Esto último es de gran importancia ya que todas las plataformas cobran por sus servicios a distintos niveles, haciendo necesario poder conocer a fondo las limitaciones que impone cada precio pues los requisitos que tenemos son muy concretos.

Antes de continuar, hay que ser consciente de que un servidor remoto es de gran utilidad, pero tiene limitaciones intrínsecas, como es el tiempo que se tarda en subir un dato a la nube. Esto impide usar el servidor como panel de instrumentos (algo que sí se puede conseguir en aplicaciones de PC y móvil) ya que, a pesar de seguir siendo en tiempo real, el retraso en la recepción de la nueva información es crítico al pilotar una aeronave.

Como consecuencia, el requisito de "en tiempo real" es ligeramente diferente para esta plataforma y, aunque la frecuencia de datos ha de ser alta para poder visualizar cambios de forma inmediata y fluida (PC y móvil), la del servidor remoto puede reducirse. Concretamente, la sensación de fluidez se puede lograr con actualizaciones alrededor de cada 100 milisegundos, pero si esto no es un factor y se busca solo recoger suficientes datos para definir con exactitud el vuelo del UAV, entonces un dato por segundo es aceptable.

Por otro lado, los servidores remotos controlan no solo la frecuencia de subida sino la cantidad de datos por envío. Esto se mide comúnmente en canales, cuyo tamaño y definición varían según la plataforma, pero sí sabemos que se necesita una capacidad total de ocho parámetros de acuerdo con lo descrito en el estudio de necesidades. También hay otros factores con cierta relevancia como si los datos se eliminan pasado un tiempo, si se pueden exportar o si los canales se pueden compartir.

En general, la opción gratuita es inviable por alguna limitación en alguno de los aspectos mencionados (tiempo entre envíos alto, canales pequeños…). Ante esto, se decidió desarrollar el servidor remoto en la plataforma ThingSpeak por una ventaja única y diferenciadora: el entorno de programación MATLAB. Este lenguaje es extremadamente potente en procesamiento y análisis, sin el cual el almacenamiento de datos no tiene sentido. En cuanto a la comunicación, se requerirá WiFi para subir los datos a la nube.

La Figura 3.4.2 es una captura de pantalla de un canal público en ThingSpeak, el cual recoge datos de una estación meteorológica en San Diego, Uruguay. Este es un claro ejemplo de las posibilidades que ofrece la plataforma, permitiendo toda clase de visualizadores y aplicar cualquier herramienta de análisis que sean parte de MATLAB. Asimismo, tiene una gran compatibilidad con Arduino, no borra los canales tras un tiempo y es posible exportarlos.

En cuanto a sus limitaciones, un canal admite hasta ocho parámetros, haciéndolo idóneo pues la versión gratuita incluye hasta cuatro canales, pero su velocidad de subida máxima es de un dato cada 15 segundos. Para solucionar esto, existe el plan de pago que por menos de 5 euros al mes (55 euros al año para estudiantes) reduce este tiempo a un dato por segundo, suficiente para la aplicación.

Una alternativa similar, Adafruit IO (empresa reconocida por fabricar microcontroladores y sensores), también permite un dato por segundo con su plan de pago, pero este es de 99 euros al año y los datos se eliminan tras 60 días, con el añadido de no tener un entorno de programación como MATLAB con ThingSpeak. En definitiva, la plataforma ThingSpeak es una opción excelente, además de que se puede usar para almacenar datos de vuelo que, en un análisis posterior, muestras cada 15 segundos sea suficiente, como puede ser la posición.

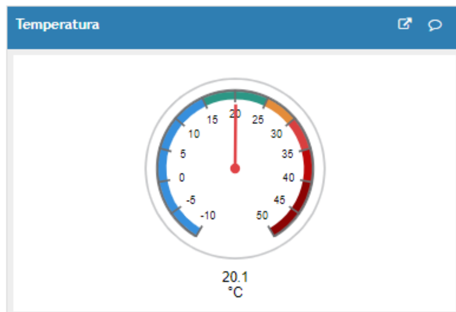
ThingSpeak™ Channels Apps Support Commercial Use How to Buy

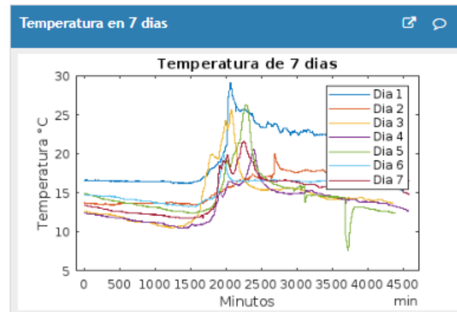
San Diego - Estación Meteorológica

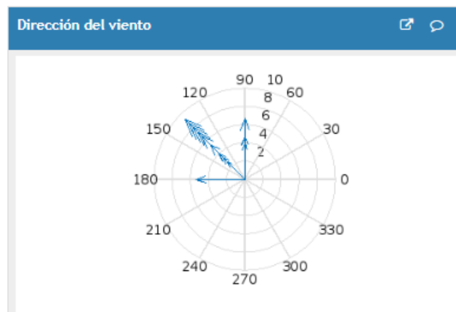
Channel ID: **1293177**
Author: santiago
Access: Public

San Diego, Cerro Largo, Uruguay Estación Meteorológica Solar (Temp, Hum, Presion, Lluvia, Viento). ESP8266, UNO R3, DTH 22, BMP 280 Update Interval - 15 seg https://clima.santiago.ovh/
uruguay, cerro largo, san diego, estacion solar, solar, cerro largo uy, uy

Export recent data | More Information | GitHub | MATLAB Analysis | MATLAB Visualization







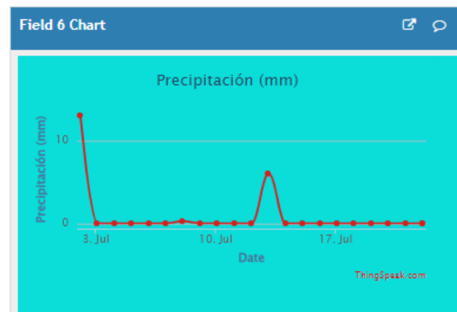


Figura 3.4.2: Canal de ThingSpeak aplicado a una estación meteorológica

3.5 ALIMENTACIÓN

Un aspecto que, en un principio, parece haber sido ignorado es la alimentación del sistema. Esto se debe a que, en la mayoría de casos, el propio UAV puede cumplir esta función gracias a que incluye baterías por ser eléctrico o tener componentes eléctricos, permitiendo realizar una conexión sencilla al pin de alimentación del ESP32.

Sin embargo, esta solución no es universal ya que es posible que sea conveniente instalar el sistema de forma que desmontarlo sea sencillo y rápido o en una zona de la aeronave de difícil acceso para el cableado eléctrico. Ante estas situaciones, la única alternativa es una batería externa que, de hecho, es lo más indicado para hacer pruebas con el prototipo fabricado pues se realizarán en un UAV de terceros y se busca evitar modificarlo.

El primer requisito que se impondrá es que sea recargable con el fin de mitigar la inversión económico a través de múltiples usos. La segunda característica a tener en cuenta es el voltaje: el microcontrolador ESP32 trabaja a 3.3 voltios (V) e integra un convertidor de 5V. El inconveniente de estos voltajes es que son poco comunes, dificultando encontrar una batería que, además, posea otras características relevantes.

Por ejemplo, los principales resultados de buscar baterías de 3.3V y 5V son la ANR 26650 y la ICR18650, que pesan 76 y 45 gramos respectivamente. Por otro lado, opciones mucho más comunes como la batería LiPo de 3.7V de la Figura 3.5.1 pesa 15 gramos, un factor muy importante a considerar. La incompatibilidad de voltaje tiene, afortunadamente, una sencilla solución, y es emplear un convertidor DC-DC *boost*.

La ventaja es que aumentar el abanico de baterías posibles hace más probable que no sea imprescindible comprar una nueva porque alguna previamente adquirida sirve. Esta es la razón por la que para el prototipo se empleará una LiPo de 3.7V y 820 miliamperios-hora (mAh) de EEMB, visible en la Figura 3.5.1. Baterías de amperaje y voltaje mucho mayor son innecesarias pues suelen tener asociado mayor peso y coste, y el consumo del sistema está diseñado para ser bajo (en la prueba de vuelo de la Sección 4.5 se hará un comentario al respecto).

En cuanto al DC-DC *boost*, también por disponibilidad se usará el PowerBoost 500C de Adafruit (ver Figura 3.5.2). Al mismo tiempo, esta opción es adecuada por su bajo peso y tamaño y amplia versatilidad, ya que a través de su entrada USB-A se puede conectar un cable que se una directamente con el ESP32, simplificando mucho la alimentación.

Figura 3.5.1: Batería LiPo de 3.7V de EEMB (Imagen de Amazon)

Figura 3.5.2: PowerBoost 500C de Adafruit (Imagen de Adafruit)

4. SOLUCIÓN ADOPTADA

El sistema de monitorizado y procesado de datos de vuelo que se propone lo conforma principalmente un dispositivo emisor y otro receptor. Ambos se encuentran comunicados por medio de módulos LoRa SX1278 y están basados en un microcontrolador ESP32 al que se le conectan los componentes necesarios.

Por otro lado, para la accesibilidad de los datos se pone a disposición del usuario tres métodos distintos: servidor remoto (ThingSpeak), aplicación móvil (Kodular) y aplicación de ordenador (App Designer de MATLAB). Todos ellos presentan sus ventajas e inconvenientes, los cuales se discuten más adelante junto a la explicación detallada de su uso y funcionamiento.

Antes de profundizar en cada una de las partes que forman el sistema completo, se adjunta el esquema de la Figura 4.1 con el objetivo de ayudar a comprender globalmente el conjunto.

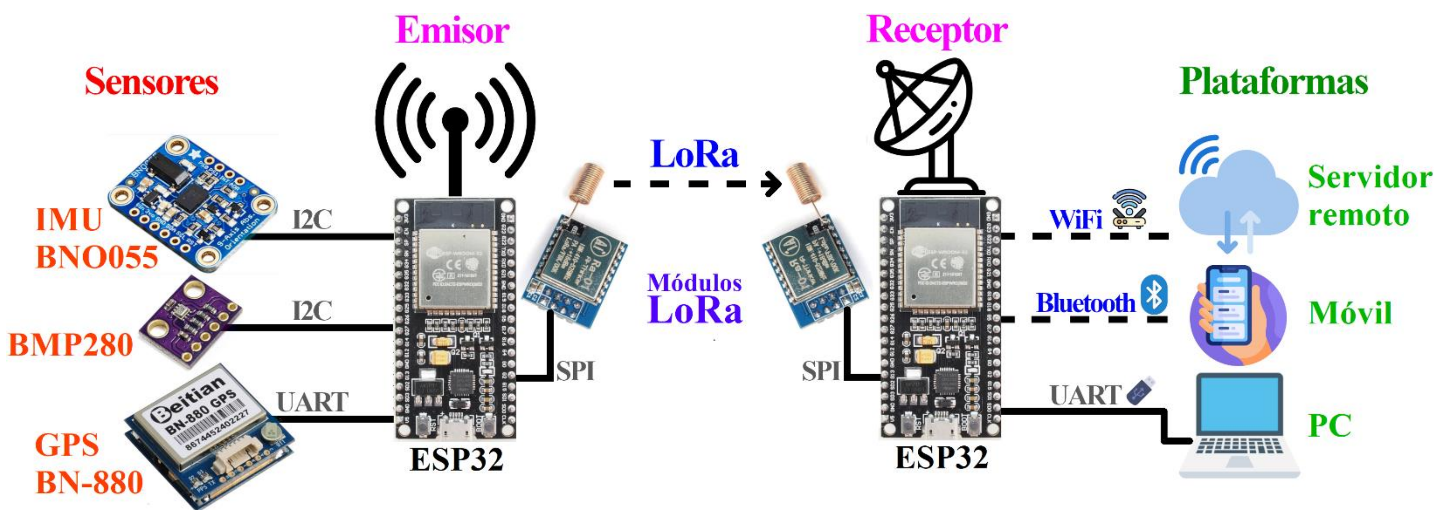


Figura 4.1: Esquema del sistema de monitorizado, procesado y visualizado de datos de vuelo

4.1 EMISOR

4.1.1 Hardware

El dispositivo emisor es un ESP32 encargado de recoger datos de los sensores y enviarlos. Dichos sensores, junto a los parámetros que mide cada uno, son:

- BMP280: presión y temperatura (calcula altitud barométrica).
- BNO055 (IMU): velocidad, cabeceo, alabeo y guiñada.
- BN-880 (GPS): velocidad, altitud, latitud y longitud.

Puesto que el BNO055 y BN-880 tienen nombres similares, para evitar confusiones de aquí en adelante nos referiremos a ellos como IMU y GPS, respectivamente. Todos los componentes añadidos se comunican en serie con el ESP32, pero lo hacen con distintos protocolos que son conveniente conocer:

- I2C (*Inter Integrated Circuit*): comunicación realizada a través de dos líneas, una de datos (SDA, *Serial Data*) y otra de sincronización (SCL, *Serial Clock*). Utilizada por el BMP280 y la IMU.
- UART (*Universal Asynchronous Receiver Transmitter*): comunicación también de dos líneas pero asíncrona, con una línea de transmisión (Tx) y otra de recepción (Rx) de datos. Es la conocida como "Serial" en Arduino y la utiliza el GPS.
- SPI (*Serial Peripheral Interface*): comunicación de cuatro líneas basada en maestro-esclavo. Emplea una línea para la transmisión de datos (MOSI, *Master Out Slave In*), una para la recepción (MISO, *Master In Slave Out*), una para la sincronización (SCL) y otra de selección de dispositivo (SS, *Slave Select*). Utilizada por el módulo LoRa junto a otras dos líneas, una de reinicio (RST, *reset*) y otra de notificación de eventos específicos como por ejemplo fin de transmisión (DIO0, *Digital Input/Output 0*).

Con estos conceptos básicos, antes de realizar las conexiones solo falta saber a qué pines del ESP32 corresponde cada línea. Para ello, volvemos a la Figura 3.1.1 en la que se pueden observar que muchos de los pines tienen asignadas líneas concretas de las comunicaciones en serie que nos interesan. Sin embargo, el ESP32 permite una gran customización, lo que quiere decir que es posible reasignar líneas si lo deseamos. Por simplicidad, para los sensores usaremos líneas predeterminadas. En el caso del BMP280 y la IMU, solo existe una opción: pin G21 para SDA y G22 para SCL. En las Figuras 4.1.1 y 4.1.2 se adjunta un esquema del cableado necesario.

Para la comunicación UART del GPS hay tres alternativas, 0, 1 y 2, entre las que escogeremos la última de ellas. El motivo de esto se debe a que, por un lado, el 0 se usa para la aplicación en MATLAB (y el "Monitor Serie" de Arduino IDE) y, por otro lado, el 1 tiene asignadas también memorias FLASH que, aunque no tiene por qué ser un problema, podría llegar a ocasionar conflictos fácilmente evitables eligiendo el UART 2. El cableado resultante se muestra en el diagrama de la Figura 4.1.3.

Por último, aunque para la comunicación SPI también hay líneas dedicadas, se han modificado de acuerdo con la Figura 4.1.4 para mayor comodidad y accesibilidad teniendo en cuenta los sensores a conectar. Naturalmente, todos los pines pueden modificarse en caso de que el usuario lo desee.

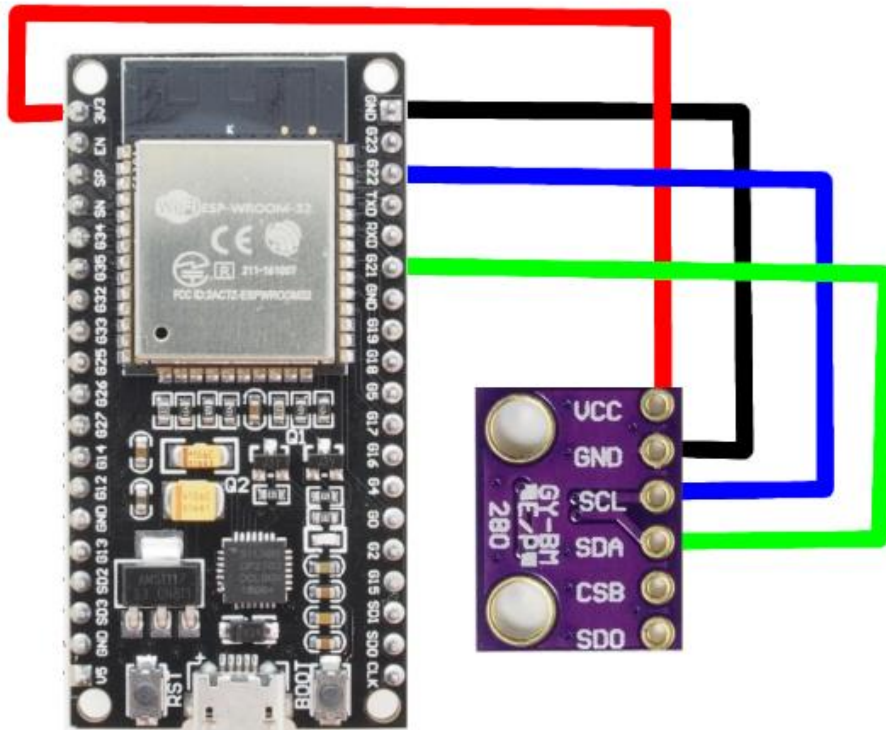


Figura 4.1.1: Diagrama para el cableado del sensor BMP280

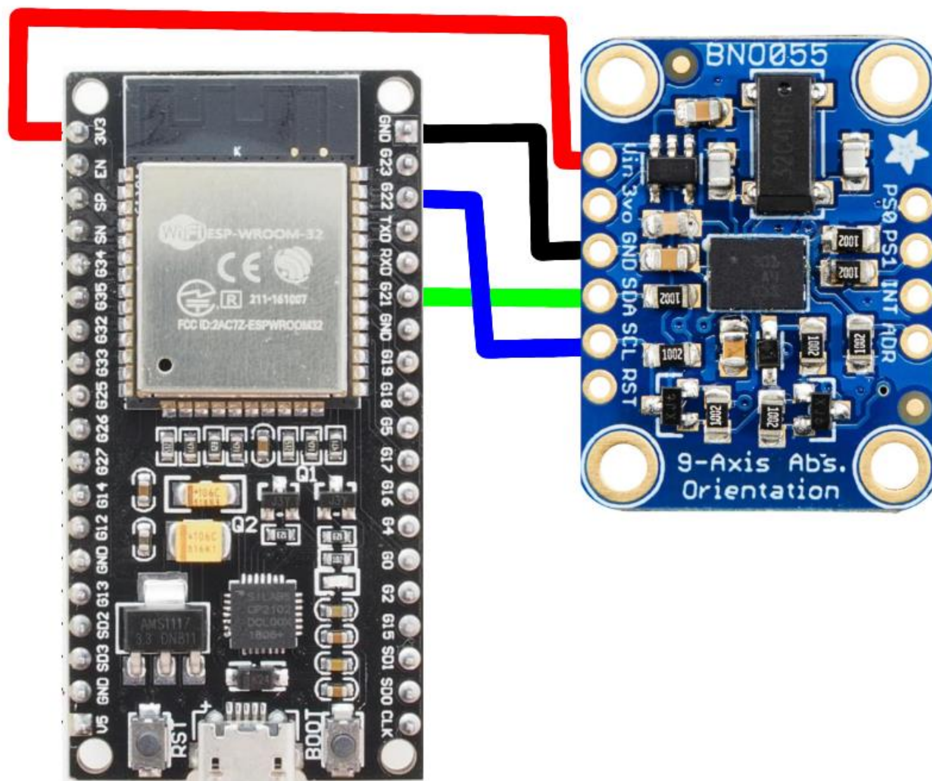


Figura 4.1.1: Diagrama para el cableado del sensor BNO055

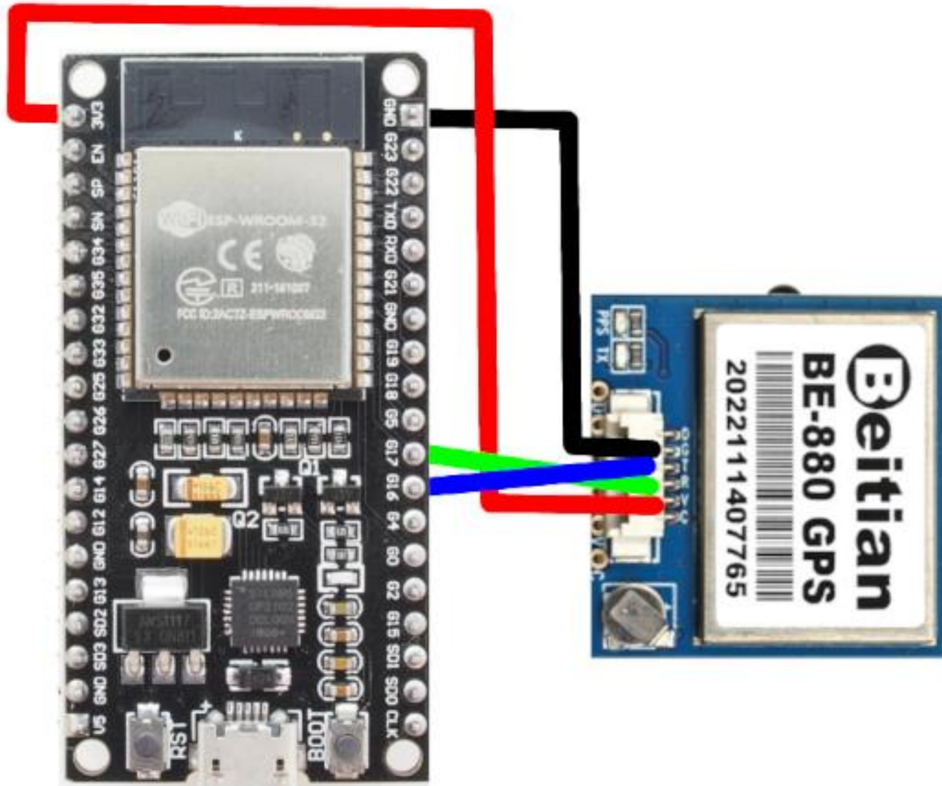


Figura 4.1.3: Diagrama para el cableado del sensor BN-880 (en la imagen BE-880, versión similar)

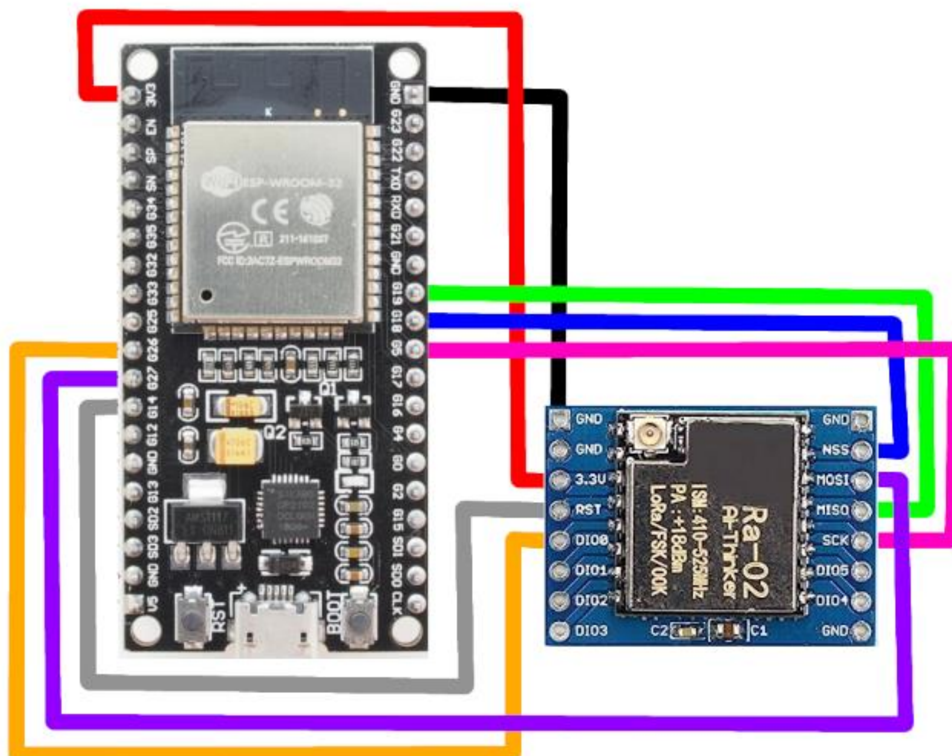


Figura 4.1.4: Diagrama para el cableado del módulo LoRa

4.1.2 Software

Respecto a la programación del ESP32 realizada en Arduino, en primer lugar, se han de introducir las librerías necesarias que se muestran en el Código 4.1.1. Además, se crean los objetos bmp, imu y gps a los cuales se acceden más adelante. Nótese que para el objeto bno hemos de acudir al manual del fabricante del sensor y comprobar la dirección I2C. En este caso, la predeterminada es la 0x28 (la ID 55 es arbitraria).

```
// LoRa
  #include <SPI.h>
  #include <LoRa.h>

// BMP280
  #include <Adafruit_BMP280.h>
  Adafruit_BMP280 bmp;

// BNO055
  #include <Adafruit_BNO055.h>
  Adafruit_BNO055 imu = Adafruit_BNO055(55, 0x28);
  // El dispositivo BNO055 corresponde con el puerto 0x28

// GPS (BN-880)
  #include <TinyGPSPlus.h>
  TinyGPSPlus gps;
```

Código 4.1.1: Inicialización de librerías

La inicialización de los sensores se incluye en el Código 4.1.2, donde destaca que es necesario comprobar la dirección del BMP280 al igual que con la IMU, siendo la 0x76 de forma predeterminada en este caso. Además, si el BMP280 o la IMU no se inicializan, se avisa del error y el ESP32 permanece en bucle infinito. Esto no es necesario para la comunicación UART del GPS pues su correcta inicialización no depende del dispositivo conectado. Será necesario observar que se enciende el LED correspondiente para saber si el GPS se está comunicando. También es importante seleccionar 9600 bits/s como velocidad de baudios pues es la que mejor funciona para este GPS (puede variar dependiendo del modelo). El uso de "Serial.print" para avisos se discutirá más adelante.

```
void startBMP() {
  if (!bmp.begin(0x76)) { // El BMP280 corresponde con el puerto 0x76
    Serial.println("No se encontró el sensor BMP280.");
    while (1);
  }
}

void startIMU() {
  if (!imu.begin()) {
    Serial.println("No se encontró el sensor BNO055.");
    while (1);
  }
```

```
}

void startGPS(){ // Comunicación UART con Serial2: pines 16 (RX) y 17 (TX)
  Serial2.begin(9600);
  while (!Serial2) delay(10);  // Esperar a que Serial se abra
  Serial.println("¡Sensores correctamente inicializados!");
}
```

Código 4.1.2: Inicialización de los sensores

La comunicación LoRa se inicializa con el Código 4.1.3., donde definimos los pines de cada línea y establecemos la frecuencia de transmisión, configurando los parámetros necesarios para tener un buen rango y tamaño de datos sin sacrificar demasiado la integridad y capacidad de propagación de la onda. Incluye unas líneas de código destinadas a monitorizar si el LoRa se ha inicializado correctamente.

```
#define SCK 5    // BLANCO
#define MISO 19 // VERDE
#define MOSI 27 // MORADO
#define SS 18    // MARRON
#define RST 14   // GRIS
#define DIO0 26 // AMARILLO

#define BAND 433E6 // Transmisión a 433 MHz (LoRa SX1278)

void startLoRA(){
  // Establecer pines
  SPI.begin(SCK, MISO, MOSI, SS);
  LoRa.setPins(SS, RST, DIO0);

  // Establecer propiedades de la comunicación
  LoRa.setSpreadingFactor(7);
  LoRa.setSignalBandwidth(125E3);
  LoRa.setCodingRate4(5);
  LoRa.enableCrc();

  Serial.print("Conectando...");
  while (!LoRa.begin(BAND)) { // Mientras no se inicialice LoRa...
    Serial.print("."); delay(500);
  }
  Serial.println(" LoRa OK!");
}
```

Código 4.1.3: Inicialización de LoRa

Todas estas funciones de los Códigos 4.1.2 y 4.1.3 se incluyen finalmente en el *void setup*. Si no hay problemas en la inicialización, se empezará a ejecutar el bucle principal del Código 4.1.4. Este se caracteriza por dos condicionales *if*, uno de lectura y otro de envío de datos. Ambos se ejecutan con cierta frecuencia contabilizada por la función *micros*, que toma del propio ESP32 la cantidad de microsegundos que el código lleva ejecutándose.

```
void loop() {

  if ((micros() - t0_read) > (ReadFreq_MS * 1000)){  // Leer datos
    t0_read = micros();
    readIMU();
    readGPS();
    readBMP();
  }

  if ((micros() - t0_send) > (SendFreq_MS * 1000)){  // Enviar datos
    t0_send = micros();
    sendData();
  }
}
```

Código 4.1.4: Bucle principal (void loop)

En la parte de lectura se encuentra en primer lugar el Código 4.1.5, que toma datos de la IMU. Este sensor es el que hace necesaria una frecuencia de lectura y envío diferente, pues la velocidad inercial se calcula indirectamente de los acelerómetros: se obtiene la aceleración en la componente de avance (x) y se multiplica por el tiempo entre tomas. Por tanto, cuanto menor sea este tiempo, más precisión tendremos. Las variables de orientación de rumbo, cabeceo y alabeo (negativo para coincidir con la convención de signos) se obtienen directamente.

```
  void readIMU(){
    […]

    // Componente X de la velocidad (V = accel * t)        (m/s -> km/h)
    V2 += linearAccelData.acceleration.x * ReadFreq_MS * 3.6f / 1000;


    // Orientación
    H = orientationData.orientation.x - H0;  // Heading o rumbo
    if (H<0) { H += 360.0f;} // Rumbo entre 0º y 360º
    p = orientationData.orientation.y - p0;  // Pitch o cabeceo
    r = -orientationData.orientation.z - r0; // Roll o alabeo
  }
```

Código 4.1.5: Función de lectura del sensor IMU

Las lecturas de los sensores restantes son directas y pueden consultarse en el Código 4.1.6. En cuanto a las unidades, cabe destacar que la velocidad es en km/h y la presión en hPa. Además, la altitud del GPS es respecto al nivel medio del mar, mientras que la del BMP280 es barométrica y necesita una presión de referencia que, por defecto, es 1 atmósfera (1013.25 hPa).

```
void readGPS(){
  while (Serial2.available() > 0) {
    if (gps.encode(Serial2.read())) {
      if (gps.location.isValid()) {
        Y = gps.location.lat();         // Latitud [º]
        X = gps.location.lng();         // Longitud [º]
        h = gps.altitude.meters() - h0; // Altitud [m]
        V = gps.speed.kmph();           // Velocidad [km/h]
      }
    }
  }
}


void readBMP(){
  T = bmp.readTemperature();            // Temperatura [ºC]
  P = bmp.readPressure() / 100.0f;      // Presión [hPa]
  h2 = bmp.readAltitude(PresRef) - h20; // Altitud [m]
}
```

Código 4.1.6: Función de lectura de los sensores GPS y BMP280

Por último, cada segundo se realizan 10 envíos de las 100 muestras tomadas por medio del Código 4.1.7. En total, son 11 los datos que se envían como paquete por LoRa, concatenados en una sola variable de texto y delimitados por el carácter “&”. Con una precisión de dos decimales (seis en el caso de latitud y longitud), esta variable no supera los 80 bytes, algo razonable para mantener el rango de comunicación LoRa deseado.

```
void sendData(){
  // LoRa
  String DataString = "";
  // Combinar todos los valores en una sola variable de texto
  //          Velocidad              […]         Longitud
  DataString = String(V,prec) + "&" + […] + "&" + String(X,6);

  LoRa.beginPacket();
  LoRa.print(DataString); // Enviar datos por LoRa
  LoRa.endPacket();
}
```

Código 4.1.7: Función de envío de datos

4.2 Receptor

El dispositivo receptor es un ESP32 encargado de recibir los datos del emisor y comunicárselos a las plataformas disponibles. En esta sección nos centraremos exclusivamente en la parte de recepción ya que los siguientes apartados se centran específicamente en la comunicación con las plataformas y su funcionamiento.

4.2.1 Hardware

El receptor está conformado únicamente por el microcontrolador y un módulo LoRa. Su hardware es, por tanto, muy simple, y es equivalente al del emisor sin sensores, por lo que el diagrama de conexiones es equivalente a la Figura 4.1.4.

4.2.2 Software

Como consecuencia de ahora enfocarnos solo en la recepción, la explicación necesaria del software es breve partiendo de la inicialización del Código 4.1.3 (las configuraciones de emisor y receptor han de ser idénticas). En el bucle principal ejecutamos continuamente la función *readData* visible en el Código 4.2.1, en el cual se comprueba si hay algún paquete para recibir por LoRa y, tras esto, se lee carácter a carácter hasta recibir el indicador de fin de paquete.

```
void readData(){
  if (LoRa.parsePacket()) { // Comprobar si se reciben datos por LoRa
    Data = "";
    while (LoRa.available()) { // Leer paquete
      Data += (char)LoRa.read(); // Leer carácter
    }
  }
}
```

Código 4.2.1: Función de lectura de datos por LoRa

4.3 SERVIDOR REMOTO

4.3.1 Código en ESP32

La comunicación con el servidor remoto ThingSpeak necesita la inicialización del Código 4.3.1, donde simplemente se ha de introducir el código de identificación y la llave de escritura de la API del canal deseado. No obstante, la función *ThingSpeak.begin* requiere del objeto *client*, para lo cual habremos de conectarnos a una red WiFi.

```
// ThingSpeak
  #include <ThingSpeak.h>

  const char* server = "api.ThingSpeak.com";
  const unsigned long channelID = 2149521;
  const char* writeAPIKey = "PC9XIEVM2EBD2Y3H";


  void startThingSpeak(){
    ThingSpeak.begin(client);
  }
```

Código 4.3.1: Inicialización de ThingSpeak

El Código 4.3.2 muestra cómo realizar dicha conexión, en la que se usan dos librerías que vienen integradas en el ESP32, *WiFi* y *WiFiClient*. Con esta última se crea el objeto *client*, mientras que con la primera se realiza la conexión usando el nombre y la contraseña de la red. La función *startWiFi* inicializa la conexión y busca la red WiFi establecida durante 30 segundos. En caso de que no se conecte, se devuelve un aviso pero no impide que el resto del código se ejecute.

```
// WiFi
  #include <WiFi.h>
  #include <WiFiClient.h>
  WiFiClient client;

  const char* ssid = " NombreWiFi";
  const char* password = " ContraseñaWiFi";


  void startWiFi(){
    WiFi.begin(ssid, password);

    Serial.print(F("Conectando a red WiFi..."));
    byte conectar = 0;
    while ( (WiFi.status()!= WL_CONNECTED) || (conectar > 30) ) {
      Serial.print(F("."));
      delay(500);
      conectar++;
    }
```

```
    if (conectar <= 30){
      Serial.println(F(" ¡Conectado a red WiFi!"));
    }
    else{
      Serial.println(F(" Conexión a red WiFi fallida."));
    }
  }
```

Código 4.3.2: Inicialización de WiFi

La subida de datos se efectúa con la función *ThingSpeakUpload* del Código 4.3.3 con la frecuencia que establezca el usuario (variable *ThingSpeakFreq_S*), principalmente dependiendo de su cuenta de ThingSpeak. Se puede establecer una frecuencia menor de la contratada (por ejemplo, cada segundo con una cuenta gratuita de un dato cada 15 segundos) y el código funcionará.

Sin embargo, la función *ThingSpeak.writeFields* tarda alrededor de 0.5 segundos en ejecutarse, por lo que no se recomienda usar una frecuencia menor para evitar un consumo de tiempo innecesario que otras funciones podrían usar. De hecho, el tiempo que ocupa dicha función es problemático por ser irregular: si una vez tarda 0.55 segundos y la siguiente 0.45 segundos, aunque entre ejecuciones de la función *ThingSpeakUpload* haya pasado la frecuencia establecida, entre subidas de datos a ThingSpeak ha pasado menos tiempo, y el servidor lo rechazará (ver ejemplo ilustrativo de la Figura 4.3.1).

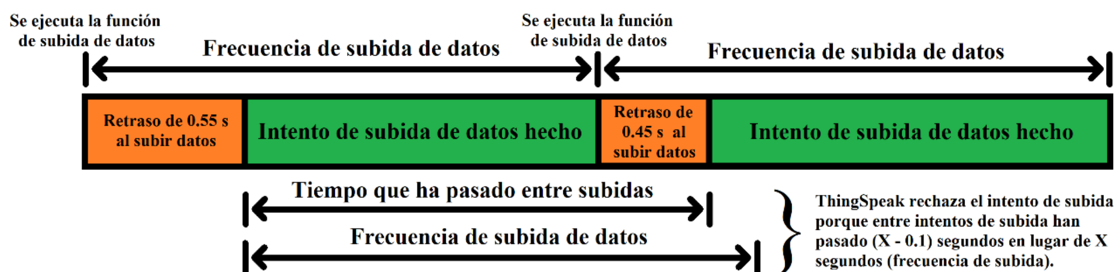


Figura 4.3.1: Esquema ilustrativo del problema de sincronización al subir datos a ThingSpeak

Para solucionarlo, a la frecuencia de subida se le añaden 0.25 segundos (variable *TScorr*) de margen, un valor obtenido experimentalmente. Otra solución es la de establecer una frecuencia de subida muy baja, pero con la desventaja explicada. Esto solo se recomienda si se modifica el código para que la única función del ESP32 receptor sea la de enviar datos al servidor remoto.

Otro inconveniente se trata de que el paquete recibido es una única cadena de texto que hay que separar en variables antes de subirlas a ThingSpeak. En el lenguaje de Arduino, se usa la función *strtok*, que extrae subpartes de texto denominadas "tokens" usando un delimitador ("&" en este caso). Esto se realiza en un bucle "while" hasta que no queden más "tokens", donde hay que abordar el último problema: el canal de ThingSpeak tiene 8 canales y tenemos 11 variables.

Con este motivo se integra en el código la variable *DMS*, siglas de *Data Main Sensor*, que puede tomar los valores 1, 2, 3 y 4:

1. El sensor principal de velocidad y altitud se establece como el GPS
2. El sensor principal de velocidad se establece como la IMU y el de altitud, el BMP280
3. El sensor principal de velocidad se establece como el GPS y el de altitud, el BMP280
4. El sensor principal de velocidad se establece como la IMU y el de altitud, el GPS

El DMS es decisión del usuario de acuerdo con la experiencia y la fiabilidad de sus sensores, siendo el GPS (1) la opción por defecto. No subir los datos redundantes nos deja con 9 variables, por lo que la otra variable descartada es la de presión por su menor relevancia. En el código, esto se consigue dentro del bucle "while", en el que cuando se dan las condiciones de DMS e identificador de variable, se salta la función *ThingSpeak.setField*.

Finalmente, ante la posibilidad de recibir datos corruptos, se comprueba la integridad del paquete recibido de dos formas: viendo que exista algún carácter "&" en el texto y verificando que todos los "tokens" sean variables numéricas válidas. En caso contrario, se establece la variable "corr" como 1 (*true*) con la que, en lugar de subir los datos, se vuelve a llamar las funciones *ReadData* y, después, *ThingSpeakUpload*.

```
void loop() {
  […]
  if  ((micros() - t_TS) >= ThingSpeakFreq_S){
    t_TS = micros();
    ThingSpeakUpload();
  }
}


  void ThingSpeakUpload(){
    byte corr = 0;
    if (Data.indexOf("&") == -1) { // Comprobar que haya delimitador "&"
      corr= 1;
      readData();
      ThingSpeakUpload();
      Serial.println(F("Datos corruptos no enviados a ThingSpeak."));
    }
    else{
      char *var; // Trozos de texto a extraer de "Data"
      const char *delimiter = "&"; // Delimitador
      var = strtok(const_cast<char*>(Data.c_str()), delimiter);

      byte var_i = 1; // Indicador de la variable actual
      byte field = 1; // Indicador del campo del canal

      while (var != NULL) { // Separar el dato recibido en variables

        // Comprobar que las variables sean números válidos
        float varcheck = atof(var);
        if (isnan(varcheck)) {corr= 1; break;}

        // Escribir dato en el campo correspondiente
        if (var_i == 5) {} // Saltar dato de presión
        else if ( (DMS == 1) && ((var_i == 2)||(var_i == 4)) ){}
        else if ( (DMS == 2) && ((var_i == 1)||(var_i == 3)) ){}
        else if ( (DMS == 3) && ((var_i == 2)||(var_i == 3)) ){}
        else if ( (DMS == 4) && ((var_i == 1)||(var_i == 4)) ){}
```

```
        else {ThingSpeak.setField(field, var); field++;}

        var=strtok(NULL, delimiter); // Siguiente token
        var_i++; // Siguiente ID de variable
      }

      if (!corr){
        int httpCode = ThingSpeak.writeFields(channelID, writeAPIKey);
        […]
      }
      else {
        Serial.println(F("Datos corruptos no enviados a ThingSpeak."));
        readData();
        ThingSpeakUpload();
      }
    }
  }
```

Código 4.3.3: Subida de datos a ThingSpeak

4.3.2 Visualizado con ThingSpeak

Con el código listo y funcionando, es momento de acudir a la página web de ThingSpeak para visualizar los datos. Por defecto, el canal muestra gráficas de la variable en función del tiempo, como las de la Figura 4.3.2. Sin embargo, estas gráficas se pueden modificar para que su apariencia sea las de un instrumento reloj, útiles para velocidad y altitud como se muestra en la Figura 4.3.3. Los límites e intervalos se pueden modificar según la aeronave.

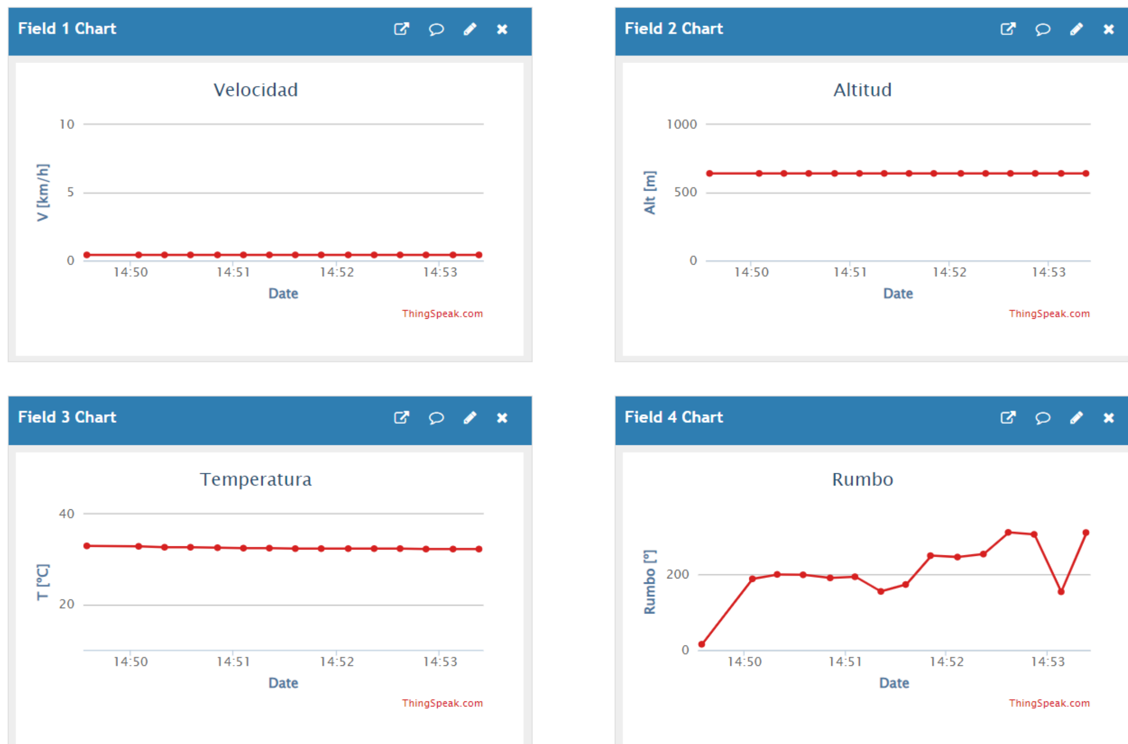


Figura 4.3.2: Gráficas en base a las predeterminadas de ThingSpeak

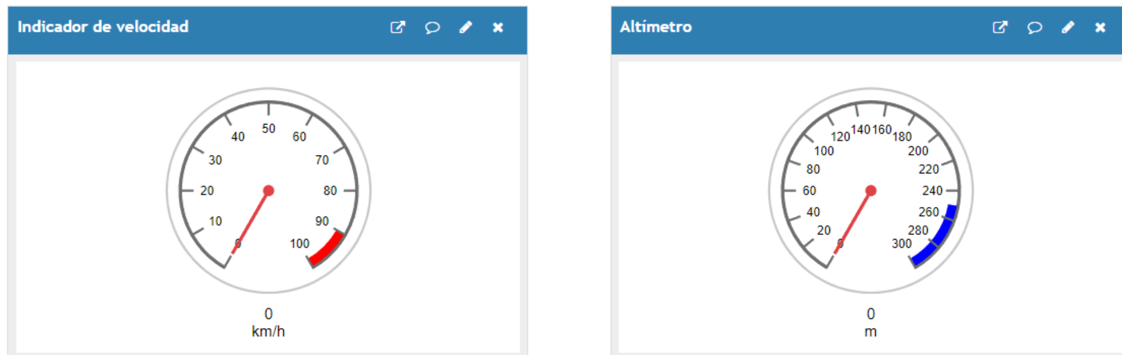


Figura 4.3.3: Instrumentos de velocidad y altitud creados con ThingSpeak

Por otro lado, con MATLAB Visualizations podemos crear un script destinado a graficar por medio de *plot* o similares. En este caso, se han creado un mapa con el recorrido trazado y un indicador de rumbo (ver Figura 4.3.4).

Figura 4.3.4: Ubicación y rumbo creados con MATLAB Visualization de ThingSpeak

Para generar la figura del mapa se usa el Código 4.3.4, donde en primer lugar se leen con la función *ThingSpeakRead* los datos de latitud y longitud necesarios. Dicha función emplea cuatro inputs: llave e ID de lectura (como el Código 4.3.1 pero de lectura en lugar de escritura), variable (*field*, 7 y 8 en este caso) y número de datos. Este último valor se calcula a partir del tiempo de vuelo que se desea mostrar y la frecuencia de subida.

Tras esto, solo queda ejecutar la función *geoscatter*, que dibuja puntos discretos en un mapa. El resto del código son ajustes: *colormap* y *colors* para colorear los puntos (origen azul, intermedios verde, final rojo), *geobasemap* para el tipo de mapa (por satélite) y *geolimits* para establecer los límites del mapa alrededor de la ubicación más reciente. Este último se basa también en la variable *res*, que es la resolución.

```
% Channel Read API Key
readAPIKey = 'ABCDEFGHIJKLMNOP';
% Channel ID to read data from
readChannelID = 1234567;

% Latitude Field ID
LatFieldID = 7;
% Longitude Field ID
LonFieldID = 8;
```

```
% Variables
tiempo  = 2; % Cuánto tiempo de vuelo mostrar en minutos
tsfreq  = 15; % Frecuencia en segundos de subida de datos a ThingSpeak
res     = 0.005; % Resolución (más pequeño, más zoom)

% Obtener latitud y longitud
n = tiempo*60/tsfreq;
lat = ThingSpeakRead(readChannelID,'Fields',LatFieldID,...
    'NumPoints',n,'ReadKey',readAPIKey);
lon = ThingSpeakRead(readChannelID,'Fields',LonFieldID,...
    'NumPoints',n,'ReadKey',readAPIKey);

% Colores para los marcadores: azul punto inicial, verde intermedios,
% rojo final.
colormap([0 0 1; 0 1 0; 1 0 0]);
if n == 1
    colors = 1;
elseif n == 2
    colors = [1 3];
else
    colors = 2 * ones(1,n-2);
    colors = [1 colors 3];
end

% Crear geoscatter plot
%          Coordenadas  Tamaño   Colores    Relleno
geoscatter(lat, lon,    50,      colors,   'filled');

geobasemap satellite % Tipo de mapa (satelital en este caso)
geolimits([lat(n)-res lat(n)+res],[lon(n)-res lon(n)+res]) % Límites
```

Código 4.3.4: Creación de mapa con MATLAB Visualizations de ThingSpeak

El indicador de rumbo es más sencillo pues solo usa el último dato de la variable rumbo. Se observa en el Código 4.3.5 que, tras obtenerla, se convierte a radianes y se crean las coordenadas polares, ángulo y radio, para el *polarplot*. Aunque solo son necesarios dos puntos, se añaden algunos más para dibujar la flecha de la Figura 4.3.4. Finalmente, se cambian algunas propiedades para que se asemeje al instrumento real (el "0" en la parte superior, sentido horario…).

```
% Channel Read API Key
readAPIKey = 'ABCDEFGHIJKLMNOP';
% Channel ID to read data from
readChannelID = 1234567;

% Obtener último valor de rumbo
hdn = ThingSpeakRead(readChannelID,'Fields',4,...
    'NumPoints',1,'ReadKey',readAPIKey);

% Convertir a radianes
rad = hdn*2*pi/360;

% Coordenadas polares (puntos extra para crear flecha)
theta = [0, rad, rad+0.1, rad, rad-0.1, rad];
r     = [0, 0.8,     0.8,   1,     0.8, 0.8];

% Generar brújula
polarplot(theta, r,'LineWidth',3,'Color','r');
```

```
% Propiedades
ax = gca;
ax.ThetaZeroLocation = 'top'; % Etiqueta de "0º" en la parte superior
ax.ThetaDir = 'clockwise';    % Números en sentido horario
ax.ThetaTick = [0:15:345];    % Resolución
ax.RTickLabel = {};           % Sin etiquetas en la dirección r
```

Código 4.3.5: Creación de indicador de rumbo con MATLAB Visualizations de ThingSpeak

4.3.3 Guardado en local con MATLAB

A la hora de manejar los datos, la mayoría de veces es conveniente exportarlos y trabajar con ellos con otras herramientas como Excel. Desde la página de ThingSpeak existe la posibilidad de hacerlo con el botón "*Export recent data*", pero esto tiene inconvenientes como no poder especificar la cantidad de datos, que son siempre los últimos 100.

Esto se puede solventar con el Código 4.3.6, un script llamado *ThingSpeakDataSave* que permite especificar el tiempo de guardado, generando un archivo CSV, formato común y sencillo de valores separados por comas. Es visible que el comienzo del código es equivalente al Código 4.3.4, donde introducimos lo necesario para realizar la lectura del servidor remoto y calcular las muestras a guardar. Estas muestras pueden ser 8000 como máximo, por lo que con el menor tiempo de subida de 1 segundo podemos guardar 133 minutos de vuelo.

Tras esto, se recogen tanto las variables como la etiqueta temporal asociada y se comprueba que no haya datos no numéricos. Nótese la cantidad de filtros existentes para evitar acabar con datos corruptos: en el envío por LoRa, al recibirlos y durante el guardado. El último paso antes del guardado es unir los datos y el tiempo en una matriz del tipo celda, añadiendo una primera fila de nombres. Asimismo, se crea un nombre de archivo donde se incluye la plataforma de origen ("TS" por ThingSpeak) y la etiqueta temporal del primer dato.

El proceso de guardado no tiene misterios: se abre el archivo y se recorre la matriz fila por fila, escribiendo el texto extraído de cada celda y añadiendo una coma o un "\n" (nueva línea) si hemos llegado a la última columna. Cuando se ejecuta la función *fclose*, el archivo CSV se añade al directorio en uso.

```
% Channel Read API Key and Channel ID to read data from
readAPIKey = 'ABCDEFGHIJKLMNOP';
readChannelID = 1234567;

% Muestras a guardar
tiempo  = 10; % Cuánto tiempo de vuelo mostrar en minutos
tsfreq  = 15; % Frecuencia en segundos de subida de datos a ThingSpeak

% Leer de ThingSpeak
n = tiempo*60/tsfreq;
[data, time] = ThingSpeakRead(readChannelID,'ReadKey',readAPIKey,...
    'Fields',1:8,'NumPoints',n);

% Encontrar y eliminar filas con valores NaN
filas_nan = any(isnan(data), 2);
data = data(~filas_nan, :);
time = time(~filas_nan, :);

% Unir datos en formato cell
datos_cell = num2cell(data);
```

```
timestamps = cellstr(datestr(time));
nombres_variables = {'Tiempo', 'Velocidad', 'Altitud', […]}

AirData = [nombres_variables; timestamps datos_cell];

% Obtener el nombre del archivo CSV de salida
fecha = strrep(strrep(datestr(time(1)),' ','_'),':','');
nombre_archivo = join(['AirData_TS_',fecha,'.csv']);

% Abrir el archivo CSV en modo de escritura
archivo_csv = fopen(nombre_archivo, 'w');

% Guardar los datos de la matriz de celdas en el archivo CSV
[n_rows, n_col] = size(AirData); % Obtener el tamaño de la matriz
for row = 1:n_rows
    for col = 1:n_col
        % Convertir el contenido de cada celda a texto
        contenido_celda = mat2str(AirData{row, col});

        % Escribir el contenido de la celda en el archivo CSV
        fprintf(archivo_csv, '%s,', contenido_celda);
    end
    fprintf(archivo_csv, '\n'); % Siguiente fila
end

% Cerrar el archivo CSV
fclose(archivo_csv);
```

Código 4.3.6: Script de MATLAB para guardado de datos de ThingSpeak en local

Otra versión de este guardado es el script *ThingSpeakFlightsSave*, el cual lee los 8000 datos disponibles en busca de vuelos concretos. En las Secciones 4.3 y 4.4 veremos que las aplicaciones de PC y móvil tienen una opción de guardado, con la cual hacen que el ESP32 receptor suba a ThingSpeak un -100 (inicio) o un -200 (fin) al primer campo del canal y un 0 al resto. Cómo se hace esto se explica en la Subsección 4.3.1.

En cualquier caso, lo que busca *ThingSpeakFlightsSave* son los indicadores de inicio de guardado con la función *find* que muestra el Código 4.3.7. Dicha función devuelve un vector con el número de fila de cada -100 y que usaremos en el bucle "for", el cual se ejecuta tantas veces como inicios de guardado existan.

Tras esto, se recorre la matriz de datos empezando por la fila siguiente a la del -100 actual hasta que aparezca un -200 (fin de guardado), otro inicio de guardado o se llegue al final de la matriz. De esta forma evitamos que, tras enviar un inicio de guardado, el vuelo se guarde sí o sí, sin necesidad de que se haya recibido un fin de guardado.

El resto del script es la creación del archivo CSV vista en el Código 4.3.6. Únicamente se destaca el nombre del archivo, al que se le añade el número de vuelo y la fecha se recorta para que solo sea el día sin la hora (ej.: AirData_TS_01-Jan-2023_Vuelo-1.csv).

```
[…]
% Leer de ThingSpeak
[data, time] = ThingSpeakRead(readChannelID,'ReadKey',readAPIKey,...
    'Fields',1:8,'NumPoints',8000);
[…]
% Buscar marcadores de vuelo
[N, ~] = find(data(:,1) == -100);
```

```
for i=1:length(N) % Crear un archivo csv para cada vuelo

    n = N(i) + 1; % Punto de inicio

    dataFlight = [];% Nueva matriz de datos con solo el vuelo guardado
    timeFlight = [];% Nuevas etiquetas de tiempo con solo el vuelo guardado

    while (data(n,1) ~= -100 && data(n,1) ~= -200)

        dataFlight = [dataFlight; data(n,:)];
        timeFlight = [timeFlight; time(n)];

        n = n + 1;
        if n > size(data,1) % ¿Se ha llegado al fin de la matriz?
            break;
        end
    end
    […]
    fecha = datestr(timeFlight(1));
    fecha = fecha(1:11); % Recortamos la fecha para quitar la hora

    nombre_archivo = join(['AirData_TS_',fecha,'_Vuelo',num2str(i),'.csv']);
    […]
end
```

Código 4.3.7: Script de MATLAB para guardado de vuelos de ThingSpeak en local

4.4 Aplicación PC

4.4.1 Código en ESP32

La aplicación en MATLAB recibe los datos del ESP32 receptor por UART, es decir, ha de estar conectado por cable. Para ello se usa el UART0, denominado simplemente "Serial" en Arduino. Su uso no requiere ninguna conexión especial adicional. A continuación, veremos las líneas de código que hay que añadir a lo mostrado en la Sección 4.1.

En primer lugar, se ha de incluir en el *void setup* la función inicializadora *startSerial*, en la que abrimos el puerto comentado a 115200 baudios (bits por segundo). Lo importante de esta velocidad es que sea consistente con la del App Designer de MATLAB, pero se puede establecer todo lo rápido que se desee siempre y cuando el cable USB lo soporte.

```
void startSerial(){
  Serial.begin(115200);
  while (!Serial) delay(10);  // Espera a que se abra el puerto Serial
}
```

Código 4.4.1: Inicialización del puerto Serial (UART0)

Con esta inicialización, podemos ir directamente al bucle principal donde encontraremos la función *AppData* del Código 4.4.2. Esta se ejecutará cada cierto tiempo y enviará por UART0 la cadena de texto recibida por LoRa.

```
[…]
  if ((micros() - t_App) >= AppFreq_MS){
    t_App = micros();
    AppData();
  }
[…]


  void AppData(){
    Serial.println(Data);
  }
```

Código 4.4.2: Función AppData

En el siguiente apartado veremos que la aplicación de PC tiene la capacidad de aplicar ciertos cambios, para lo cual no solo lee datos, sino que también escribe. Por esta razón, hemos de adelantarnos y mostrar la función *readSerial* del Código 4.4.2, la cual se ejecuta continuamente en el bucle principal comprobando si hay algún byte para recibir por Serial.

En caso afirmativo, se comprueba si es un 0, un 1, un 7 o un 8 (en formato texto, no numérico, de ahí la notación hexadecimal: 0x30 = '0'); en cualquier otro caso, se ignora. El significado de cada valor es el siguiente:

- 0: desactiva el envío de datos a ThingSpeak con TS_ON = 0 (false)
- 1: activa el envío de datos a ThingSpeak con TS_ON = 1 (true), opción por defecto
- 7: inicio de guardado con SaveStart = 1
- 8: fin de guardado con SaveStop = 1

El encendido o apagado de la subida de datos se aplica añadiendo un “if” que comprueba el valor de la variable TS_ON antes de ejecutar la función *ThingSpeakUpload* del Código 4.3.3. Además, si se desactiva la subida a ThingSpeak entonces el envío de datos con AppData pasa a ejecutarse cada vez que se recibe un dato por LoRa. Para ello, a la función *ReadData* le añadimos:

```
if (!TS_ON){AppData();} // Ejecutar AppData con ThingSpeak desactivado
```

Con esto aprovechamos la máxima velocidad de envío de datos a otras plataformas que disponemos, que es de 100 milisegundos (frecuencia de envío del ESP32 emisor). Esto mismo con *ThingSpeakUpload* activa es una posibilidad; sin embargo, al tardar alrededor de 0.5 segundos en ejecutarse causa parones en bucle principal. Si la frecuencia de subida es baja, esto puede no ser molesto, pero a la frecuencia ideal de 1 Hz se produce una irregularidad en el recibo de datos muy notoria.

Aplicar el guardado no es tan simple, por lo que se adjunta el Código 4.4.3, el cual también se ha de añadir a la función *ThingSpeakUpload*. En él, primero se verifica si hay algún marcador de guardado activo para asignar el valor -100 o -200 al primer campo del canal de ThingSpeak según corresponda. Después, se asigna un 0 al resto de campos y se tratan de subir con *ThingSpeak.writeFields*. Esta función devuelve un 200 si la subida fue exitosa, y solo en caso de que lo sea se asignará 0 a los indicadores SaveStart o SaveStop. Si no lo es, el Código 4.3.3 volverá a ejecutarse en el siguiente ciclo.

```
void readSerial(){
  if (Serial.available()) { // Comprobar si se reciben datos por Serial

    byte i = Serial.read(); // Leemos el byte recibido

    while (Serial.available()) {Serial.read();} // Descartamos el resto

    // Si recibimos un '0', desactivamos el envío de datos a ThingSpeak
    if (i == 0x30) {
      TS_ON = 0;
      Serial.println(F("ThingSpeak: OFF"));
    }
    // Si recibimos un '1', activamos el envío de datos a ThingSpeak
    else if (i == 0x31) {
      TS_ON = 1;
      Serial.println(F("ThingSpeak: ON"));
    }

    // '7': enviamos a ThingSpeak un marcador de inicio de guardado
    else if (i == 0x37) {
      SaveStart = 1;
      Serial.println(F("Inicio de guardado enviado"));
    }
    // '8': enviamos a ThingSpeak un marcador de fin de guardado
    else if (i == 0x38) {
      SaveStop  = 1;
      Serial.println(F("Fin de guardado enviado"));
    }
```

```
  }
}
```

Código 4.4.2: Función de lectura por UART

```
  else if ((SaveStart)||(SaveStop)){
    int save = 0;
    if (SaveStart){save = -100;}
    else if (SaveStop){save = -200;}

    ThingSpeak.setField(1, save);
    for (byte i = 2; i <= 8; i++) {
      ThingSpeak.setField(i, 0);
    }

    int httpCode = ThingSpeak.writeFields(channelID, writeAPIKey);
    if (httpCode == 200) {
      if      (SaveStart) {SaveStart = 0;
        Serial.println(F("Guardado comenzado en ThingSpeak."));
      }
      else if (SaveStop)  {SaveStop  = 0;
        Serial.println(F("Guardado terminado en ThingSpeak."));
      }
    }
  }
```

Código 4.4.3: Código adicional para guardado de ThingSpeak

Como se vio en la Subsección 4.1.1, el código del ESP32 emisor también contiene llamadas al objeto Serial (y un Código 4.4.1 equivalente). Esto no está pensado para la aplicación de MATLAB sino para el "Monitor Serie" del Arduino IDE, un visualizador sencillo de todo aquello que el microcontrolador envía por el puerto Serial (UART0) pensado para detectar algunos errores (por ejemplo, un BMP280 o IMU mal conectado).

4.4.2 Aplicación con App Designer

En esta subsección se explicará la aplicación desarrollada con App Designer de MATLAB. Respecto al diseño, esta tiene tres pantallas o ventanas:

- Panel de control (Figura 4.4.1): muestra todos los datos recibidos de forma directa, incluyendo texto informativo. Incluye tres botones:
    - Start/reset: permite empezar a visualizar los datos recibidos o detener este proceso
    - Comenzar/detener guardado: envía un 7 o un 8 al ESP32 receptor para manipular las variables de guardado como se vio en la función readSerial del Código 4.4.2. También realiza un guardado local.

   - Activar/desactivar la subida de datos a ThingSpeak: envía un 0 o un 1 al ESP32 receptor para manipular la variable TS_ON como se vio en la función readSerial del Código 4.4.2.
- Panel de instrumentos (Figura 4.4.2): simula el panel de instrumentos de una aeronave de forma simple. Incluye:
   - Indicador de velocidad (velocidad de GPS)
   - Indicador de velocidad inercial (velocidad de IMU)
   - Altímetro (altitud de GPS)
   - Baroaltímetro (altitud por presión atmosférica de BMP280)
   - Horizonte artificial (cabeceo y alabeo de IMU)
   - Indicador de rumbo (rumbo de IMU)
   - Termómetro (temperatura de BMP280)
- Ubicación (Figura 4.4.3): crea un mapa que se actualiza con la posición recibida, dejando constancia del recorrido.

En las líneas de código que se presentarán en adelante será útil saber en qué pantalla se encuentra el usuario, para lo cual se crean funciones que se ejecutan al clicar sobre las pestañas superiores. Redefiniendo con 0 o 1 las variables *ControlTab*, *InstrumentTab* y *MapaTab* se lleva la cuenta de cuál se ha pulsado.

Todas estas ventanas se basan en la información recibida por UART, por lo que empezaremos explicando cómo se realiza esta conexión por medio del Código 4.4.4. Este se ejecuta al iniciar la aplicación (función integrada *startUpFnc*) y comienza cerrando puertos activos y buscando si hay alguno disponible, devolviendo un error en caso contrario. Tras esto, crea el objeto Puerto con la función *serialport*, que tiene como argumentos el nombre del puerto (COM3 en este caso) y la velocidad de baudios (115200 bits por segundo al igual que en el ESP32 receptor). Este objeto será el que se use para leer y escribir además de otros ajustes.

También hemos de crear la función de retorno de llamada o "callback" vinculada a la recepción de datos por el puerto COM3, la cual llamaremos *Funcion_Recepcion*. Sus argumentos son el puerto, el nombre de la función y el tipo de desencadenante. Puesto que solo recibiremos cadenas de texto, se establece en modo "terminator", el cual de forma predeterminada llama a la función cuando hay un "\n" (indicador de siguiente línea) disponible para leer. Esto es lo deseado en nuestro caso pues es el tipo de "terminator" que la función *Serial.println* añade al final del texto enviado en Arduino.

```
function startupFcn(app)
    global Puerto […]
    clc;

    % Cierra los puertos que se hayan podido quedar abiertos
    Puertos_Activos=instrfind; % Lee los puertos activos
    if isempty(Puertos_Activos)==0 % Comprueba si hay puertos activos
        fclose(Puertos_Activos); % Cierra los puertos activos
        delete(Puertos_Activos); % Borra la variable Puertos_Activos
        clear Puertos_Activos; % Destruye la variable Puertos_Activos
    end

    % Puertos disponibles
    if isempty(serialportlist("available"))
        error("Conexión por USB no disponible")
    end

    % Se crea el objeto serial asociado al COMX
    Puerto = serialport("COM3",115200);
```

```
    % Se crea la llamada de recepción de datos
    configureCallback(Puerto,'terminator',@app.Funcion_Recepcion);
    % Se abre el puerto (comunicación UART)
    fopen(Puerto);
[…]
end
```

Código 4.4.4: Conexión UART en App Designer

La función de recepción de datos se muestra sintetizada en el Código 4.4.5, el cual solo se ejecuta en caso de que el "ControlButton" (botón de abajo a la izquierda en la Figura 4.4.1) esté activo, es decir, el usuario haya hecho clic en "Start". Si es así, se leen todas las cadenas de texto recibidas por el puerto y se procesan de la siguiente forma:

1) ¿Es numérico el primer carácter y el texto contiene 10 caracteres "&"?
2) Si no, ¿es al menos el primer carácter numérico?
3) Resto de casos

Con este método, se sabe si el texto recibido es un conjunto de variables válido (1), si son variables no válidas (2) o si es cualquier otro tipo de texto (3), normalmente avisos que envía el ESP32 receptor (por ejemplo, "datos enviados a ThingSpeak"). Adicionalmente, a los casos 2 y 3 se les añade un condicional para que solo se ejecuten si el usuario está en el panel de control, evitando así gasto de recursos innecesarios.

Dentro del caso 1 se lleva a cabo un procesado más extenso, donde se empieza contabilizando la muestra recibida (variable *cont*) y separándola con *split* para poder añadirlas al vector de la variable correspondiente. Si el botón "SaveFlight" está pulsado, el guardado está activo y, por tanto, los datos recibidos se añaden al vector *DatosGuardados* junto a una etiqueta temporal.

Por último, se comprueba si el usuario está en el panel de control o de instrumentos para actualizar las etiquetas de cada variable o los instrumentos según corresponda. El panel de control tiene además un LED verde arriba a la izquierda que parpadea con la recepción de datos, efecto logrado con la variable *lamp*.

```
function Funcion_Recepcion(app,~,~)
    global Puerto […]

    if (app.ControlButton.Value == 1)

        datos_str = fgetl(Puerto); % Leer datos del puerto

        if isstrprop(datos_str(1),'digit') && count(datos_str,'&') == 10

            cont=cont+1; % Se incrementa en contador de muestras

            datos = str2double(split(datos_str,"&")); % Separar variables

            % Variables
            vel(cont)      = datos(1);
            vel2(cont)     = datos(2);
            […]
            lat(cont)      = datos(10);
            lon(cont)      = datos(11);

            % Guardar datos
            if app.SaveFlight.Value == 1
                DatosGuardados = [DatosGuardados; datos'];
                TimeStamp = [TimeStamp; datetime('now')];
```

```
            end

            % DATOS
            if ControlTab == 1
                app.DatosRecibidos.Value = datos_str;

                app.VelocidadGPS.Value = string(vel(cont));
                […]
                app.Longitud.Value = string(lon(cont));

                if lamp == 0
                    app.Lamp.Color = [0,1,0]; lamp = 1;
                elseif lamp == 1
                    app.Lamp.Color = [1,1,1]; lamp = 0;
                    app.Info.Value="";
                end

            % INSTRUMENTOS
            elseif PanelTab == 1
                app.AirspeedIndicator.Airspeed = vel(cont);
                app.AirspeedInercial.Value = vel2(cont);
                app.AltimetroGPS.Altitude = alt(cont);
                app.Baroaltimetro.Value = alt2(cont);
                app.HorizonteArtificial.Pitch = pitch(cont);
                app.HorizonteArtificial.Roll = roll(cont);
                app.IndicadordeRumbo.Heading = hdng(cont);
                app.Termometro.Value = temp(cont);
            end

        elseif ControlTab == 1 && isstrprop(datos_str(1),'digit')
            app.Info.Value = "Datos recibidos corruptos";

        elseif ControlTab == 1
            app.Info.Value = datos_str;
        end
    end

end
```

Código 4.4.5: Función recepción de datos por UART en App Designer

La programación detrás de los botones del panel de control es simple de explicar. En primer lugar, al darle a “Reset” no solo se deja de ejecutar el Código 4.4.5 sino que todas las variables guardadas se reinician a 0. Por otro lado, el botón de guardado y el interruptor de ThingSpeak se comunican con el ESP32 receptor con la función *fwrite(Puerto,'X')*, enviando el valor como texto de acuerdo con lo establecido en el Código 4.4.2.

Asimismo, cuando se hace clic sobre “Detener guardado”, no solo se envía un “8” sino que también se ejecuta el Código 4.4.6. Como se puede ver, es una versión del Código 4.3.6 pero independiente de ThingSpeak y con más variables. Los datos guardados son los recogidos con el Código 4.4.5, y el archivo se nombra con el indicativo “APP” junto a la fecha completa de inicio de guardado.

```
% Generar nombres de variables
nombres_variables = {'Tiempo','Velocidad GPS', 'Velocidad IMU','Altitud
GPS','Alt. barom.','Presion', 'Temperatura', 'Rumbo', 'Cabeceo', 'Alabeo',
'Latitud','Longitud'};
```

```
% Unir datos en formato cell
datos_cell = num2cell(DatosGuardados);
timestamps = cellstr(datestr(TimeStamp));

AirData = [nombres_variables; timestamps datos_cell];

% Obtener el nombre del archivo CSV de salida
nombre_archivo = join(['AirData_APP_',fecha,'.csv']);

% Abrir el archivo CSV en modo escritura
archivo_csv = fopen(nombre_archivo, 'w');

[…]

% Cerrar el archivo CSV
fclose(archivo_csv);
```

Código 4.4.6: Guardado de datos en local con App Designer

Respecto a la ventana de Ubicación, en ella se aplica la principal utilidad de poder diferenciar en qué pantalla se encuentra el usuario ya que el visualizado del mapa consume gran cantidad de recursos y puede ralentizar la aplicación. Para optimizarla, el mapa no se actualiza con la recepción de datos sino con un temporizador aparte limitado a un segundo (recordemos se pueden recibir datos hasta cada 100 milisegundos).

El Código 4.4.7 se encuentra en la función inicializadora *startUpFnc*, creándose así el temporizador al abrir la app. A este lo llamamos *GenerarMapa* y le asignamos tanto el periodo de 1 segundo como la función asociada. Para que la pantalla Ubicación no esté vacía al inicio, se genera un primer mapa de todo el mundo (ver Figura 4.4.4).

```
GenerarMapa = timer; % Se crea el objeto timer
GenerarMapa.Period = 1; % Se establece el periodo en segundos
GenerarMapa.TimerFcn = @app.Funcion_GenerarMapa;
% Cuando se dispare el evento del timer se ejecutará el Generar Mapa

gx = geoaxes(app.MapaPanel); % Asignamos dónde generar el mapa
geoplot(gx,0,0); % Mapa creado en las coordenadas (0,0)
gx.Basemap = 'satellite'; % Mapa satelital
gx.ZoomLevel = 2; % Zoom pequeño para que se vea todo el mundo
```

Código 4.4.7: Inicialización para la ventana Ubicación en App Designer

Como puede verse, la generación del mapa se basa en la función *geoplot*. Para que este plot se genere en la ventana deseada, definimos el objeto *gx* usando *geoaxes*, asignando a estos ejes el panel del mapa. Este objeto permite muchas opciones de configuración del *geoplot*, como es el mapa base (satelital en este caso) o el zoom.

La función de generar mapa del Código 4.4.8 es más completa y solo se ejecuta si la variable contador (*cont*) es mayor de 0, es decir, si el usuario ha hecho clic en “Start” y se ha recibido algún dato válido por UART. Puesto que la frecuencia de ejecución puede ser diferente a la de recepción, se han de crear vectores de latitud y longitud diferentes, a los que se le va añadiendo el último dato recibido de estas variables. Dicho dato se añade a la etiqueta de “Posición actual”.

Antes de generar el mapa, el objeto *gx* es borrado y recreado con el fin de evitar tener una gran cantidad de mapas superpuestos, que ocasionan problemas gráficos y de rendimiento. Acto seguido, se llama a la función *geoplot* con tres inputs de coordenadas:

1) Punto inicial: marcador de estrella de color rojo
2) Puntos intermedios: líneas discontinuas azules
3) Punto final: marcador de círculo de color rojo

Finalmente, establecemos algunas opciones de *geoplot* para una correcta visualización: centrado de mapa alrededor del último punto, mapa satelital y zoom a elección del usuario. Este zoom utiliza el valor seleccionado entre 10% y 100% (elemento abajo a la derecha de las Figuras 4.4.3 y 4.4.4) y lo transforma en un valor de 10 a 20 para que la característica *ZoomLevel* la procese correctamente.

```
function Funcion_GenerarMapa(app,~,~)
    global lat lon latarray lonarray cont gx
    if cont > 0
        latarray = [latarray, lat(cont)];
        lonarray = [lonarray, lon(cont)];

        app.PosLabel.Text = join([…], num2str(lat(cont),'%.6f'),[…]);

        delete(gx); % Borrar mapa anterior

        % Generamos mapa:
        gx = geoaxes(app.MapaPanel);

        geoplot(gx, ...
          latarray(1),lonarray(1),'p', ... % Marcador de estrella
          latarray,lonarray,'b--',... % Línea discontinua azul
          latarray(end),lonarray(end),'o',... % Marcador de círculo
          […],"MarkerFaceColor","r","MarkerEdgeColor","r"); % Color rojo
        gx.MapCenter = [latarray(end) lonarray(end)]; % Centrado
        gx.ZoomLevel = round(10 + (app.ZoomSpinner.Value - 10)* 10/90, 1);
        gx.Basemap = 'satellite';

    end
end
```

Código 4.4.8: Actualización de mapa de la ventana Ubicación en App Designer

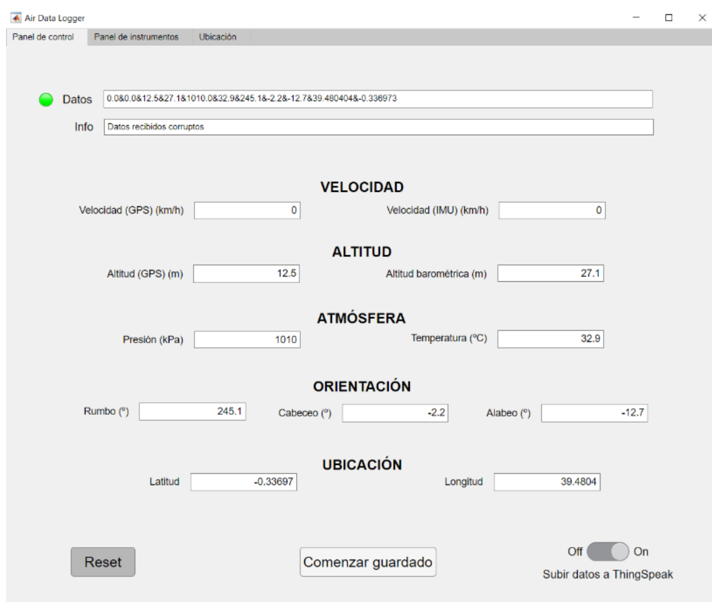


Figura 4.4.1: Ventana "Panel de control" en App Designer

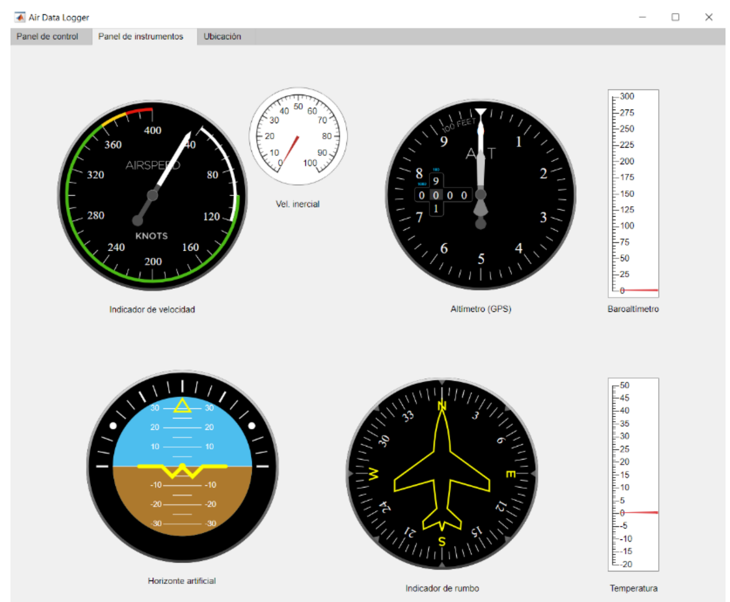


Figura 4.4.2: Ventana "Panel de instrumentos" en App Designer

Figura 4.4.3: Ventana "Ubicación" en App Designer

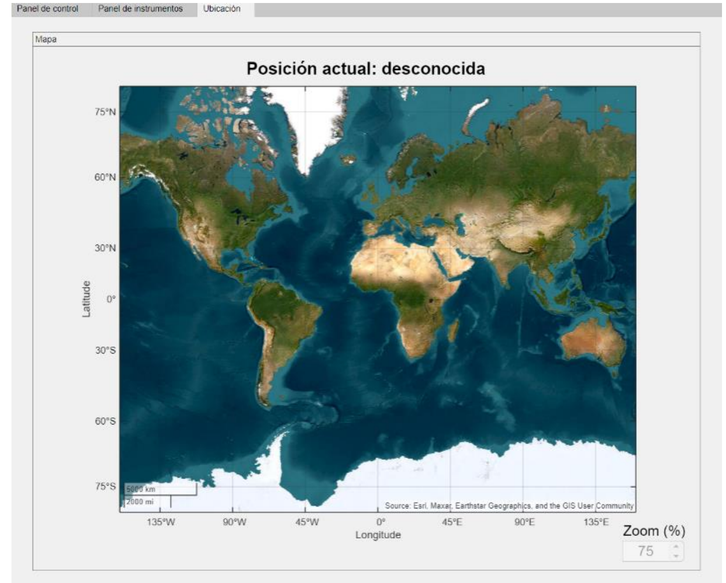


Figura 4.4.4: Ventana "Ubicación" al inicio en App Designer

4.5 APLICACIÓN MÓVIL

4.5.1 Código en ESP32

La comunicación de la aplicación móvil se basa en Bluetooth, y con ella no solo recibe datos del receptor sino que aplica cambios tanto al receptor como al emisor. Por tanto, es necesario introducir en ambos ESP32 el Código 4.5.1, con el que se inicializa el Bluetooth.

A través de él definimos el nombre con el que aparecerá en la lista de dispositivos Bluetooth del móvil, por lo que cada ESP32 ha de tener nombres distintos (por ejemplo, “EmisorBT” y “ReceptorBT”). Por otro lado, aparece la función *clearBluetoothBuffer* a la que llamaremos en muchas ocasiones, permitiendo borrar cualquier byte indeseado que interfiera en la comunicación.

```
// Bluetooth
  #include <BluetoothSerial.h>
  BluetoothSerial SerialBT;

  void startBluetooth(){
    SerialBT.begin(F("NombreESP32")); // Nombre del dispositivo
    clearBluetoothBuffer();
  }

  void clearBluetoothBuffer() {
    while (SerialBT.available()) {
      SerialBT.read(); // Leer y descartar byte
    }
  }
```

Código 4.5.1: Inicialización Bluetooth para ESP32 emisor y receptor

La segunda y última igualdad entre el emisor y el receptor es el uso de la función *readBT* dentro del bucle principal, la cual comprueba continuamente si hay bytes disponibles para recibir por Bluetooth. Sin embargo, la información que se espera recibir no es la misma, por lo que el contenido de esta función difiere.

Empezando por el ESP32 emisor, el Código 4.5.2 muestra el procedimiento dependiendo del valor del primer byte recibido:

- Si es un 1, se van a recibir otros 7 bytes con los siguientes ajustes:
  - Calibración de alabeo y cabeceo, altitud y/o rumbo (3 bytes de true/false)
  - Precisión (1 byte con el número de decimales: 0, 1 o 2)
  - Presión de referencia (número que ocupa 3 bytes, ej: 101325)
- Si es un 3, el usuario ha preguntado por la calidad de la conexión (WiFi y LoRa). Como el ESP32 emisor la desconoce (ha de preguntarse al receptor), devuelve un 1 (valor imposible) que la app móvil se encargará de procesar para avisar del error.
- En cualquier otro caso, se limpia el búfer del Bluetooth.

De estos ajustes, la calibración requiere cierta programación que cabe explicar. En primer lugar, cuando la variable *cal* del Código 4.5.2 toma el valor de 1, el bucle principal llama a la función *calibrar* del Código 4.5.3. Esta función reestablece la variable *cal* a 0 para que solo se ejecute una vez, y comprueba si algún indicador de calibración está activo. De este modo, se definen las variables *x0* que crean el nuevo punto de equilibrio.

```
void readBT(){

  if (SerialBT.available()){

    byte i = SerialBT.read();
    delay(1); // Se añaden delays de 1 ms para evitar interferencias

    if (i == 1){ // Calibración
      sIMU = SerialBT.read(); // Indicador de cal. de alabeo y cabeceo
      delay(1);

      sALT = SerialBT.read(); // Indicador de calibración de altitud
      delay(1);

      sHDN = SerialBT.read(); // Indicador de calibración de rumbo
      delay(1);

      prec = SerialBT.read(); // Precisión
      delay(1);

      // Presión de referencia (3 bytes)
      byte byte1 = SerialBT.read(); // Parte alta
      delay(1);
      byte byte2 = SerialBT.read(); // Parte media
      delay(1);
      byte byte3 = SerialBT.read(); // Parte baja

      PresRef = ((byte1 << 16) | (byte2 << 8) | byte3 ) / 100.0f;

      // Indicador de cambios recibidos
      cal = 1;
    }

    if (i == 3){
      SerialBT.write(1);
      SerialBT.write(1);
    }

    else {clearBluetoothBuffer();}
  }
}
```

Código 4.5.2: Lectura de datos Bluetooth para ESP32 emisor

```
void calibrar(){
  cal = 0;

  // Calibración
  if (sIMU){
    sIMU = 0;
    p0 = p;
    r0 = r;
  }

  if (sHDN){
    sHDN = 0;
    H0 = H;
  }

  if (sALT){
    sALT = 0;
    h0 = h;
    h20 = h2;
  }
}
```

Código 4.5.3: Función de calibración para ESP32 emisor

En cuanto a la función *readBT* del receptor, visible en el Código 4.5.4, el procedimiento en función del primer byte recibido es el siguiente:

- Si es un 2, se van a recibir otros 5 bytes con los siguientes ajustes:
  - Data Main Sensor (DMS), explicado en la Sección 4.2.1 (1 byte: 1, 2, 3 o 4)
  - Subida de datos a ThingSpeak activada o desactivada (1 byte de true/false)
  - Frecuencia de subida a ThingSpeak (1 byte: de 1 a 60 segundos)
  - Frecuencia de envío a las aplicaciones (2 bytes: de 100 ms en adelante)
- Si es un 3, se ha solicitado la calidad de las conexiones LoRa y WiFi. Si alguna está desconectada, se envía un 0; en caso contrario, se envía el indicador de fuerza de la señal recibida (RSSI, medido en decibelios y con un rango típico de entre -30 y -100) en valor absoluto pues es un número negativo.
- Si es un 7, se establece *SaveStart* como verdadero (comienzo de guardado)
- Si es un 8, se establece *SaveStop* como verdadero (fin de guardado)
- En cualquier otro caso, se limpia el búfer del Bluetooth.

Para terminar con la programación del ESP32 receptor, a la función *AppData* del Código 4.4.2 simplemente hemos de añadirle que ejecute:

```
SerialBT.println(Data);
```

Como consecuencia, los datos se envían sin separar, al igual que en la aplicación de MATLAB, y será necesario procesarlos en la app móvil.

```
void readBT(){
  if (SerialBT.available()) { // Comprobar si se reciben datos por BT
    byte i = SerialBT.read();
    delay(1); // Se añaden delays para evitar interferencias

    if (i == 2){
      DMS = SerialBT.read();
      delay(1);

      TS_ON = SerialBT.read();
      delay(1);

      ThingSpeakFreq_S = SerialBT.read() * 1000000 + TScorr;
      delay(1);

      byte byte1 = SerialBT.read();
      delay(1);
      byte byte2 = SerialBT.read();

      AppFreq_MS = ((byte1 << 8) | byte2) * 1000;
    }

    else if (i == 3) {
      if (WiFi.status() == WL_CONNECTED){RSSI_WiFi = abs(WiFi.RSSI());}
      else {RSSI_WiFi = 0;}

      if (LoRaOK) {RSSI_LoRa = abs(LoRa.packetRssi());}
      else {RSSI_LoRa = 0;}

      SerialBT.write(RSSI_WiFi);
      SerialBT.write(RSSI_LoRa);
    }

    else if (i == 7) {SaveStart = SerialBT.read();}

    else if (i == 8) {SaveStop  = SerialBT.read();}

    else {clearBluetoothBuffer();}
  }
}
```

Código 4.5.4: Lectura de datos Bluetooth para ESP32 receptor

Nótese que los bytes indicativos, con los cuales el ESP32 sabe qué se espera de él, no coinciden entre dispositivo emisor y receptor a excepción del “3” (calidad de conexión). Gracias a esto se impide que un ESP32 procese datos no destinados a él, lo que daría lugar a errores.

4.5.2 Aplicación móvil con Kodular

En este apartado se tratará el funcionamiento, la composición y la programación más relevante de la aplicación móvil diseñada en Kodular, la cual se ha bautizado con el nombre de Air Data App o ADA. Al igual que en la aplicación MATLAB, podemos dividir la Air Data App en las siguientes pantallas:

1. Inicio: muestra los datos de vuelo. Incluye una opción para cambiar las variables visibles.
2. Mapa: muestra un mapa con la ubicación más reciente recibida
3. Guardar vuelo: permite guardar vuelos mediante ThingSpeak
4. Conexión: permite conocer la calidad de la conexión LoRa y WiFi
5. Ajustes: permite cambiar los ajustes vistos anteriormente para los ESP32 emisor (i=1 en el Código 4.5.2) y receptor (i=2 en el Código 4.5.4).

Para moverse entre pantallas, se usa el componente "Side Menu Layout" con el que se crea el menú lateral de la Figura 4.5.1. A este se puede acceder pulsando en el botón de arriba a la izquierda, representado por tres rayas (ver Figura 4.5.7), o deslizando en alguno de los extremos laterales.

Los bloques que constituyen la programación de la ADA se muestran en la Figura 4.5.2. A pesar de ser un tipo de programación muy visual, la alta densidad de elementos puede dificultar la comprensión en un primer momento. Por este motivo, las figuras con bloques en las que se apoyarán las explicaciones solo incluirán aquellos esenciales.

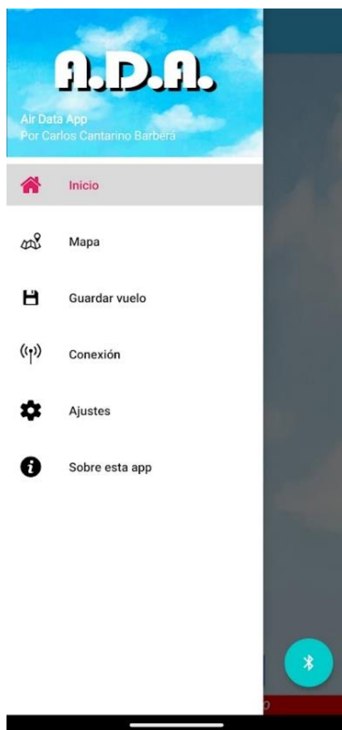


Figura 4.5.1: Menú para moverse entre pantallas

Figura 4.5.2: Visualizado de todos los bloques que componen la Air Data App

La funcionalidad más relevante es la menos visible, pero es en la que se fundamenta la aplicación, y es la comunicación Bluetooth. Para crearla, Kodular dispone del componente "Bluetooth Client" que usaremos para conectarnos a los ESP32 emisor y receptor. Dentro de ADA, esto se hace con el botón flotante que aparece en todas las pantallas abajo a la derecha (ver, por ejemplo, Figura 4.5.1 o Figura 4.5.7).

Cuando se pulsa este botón, se ejecutan los bloques de la Figura 4.5.3:

- Se comprueba que el Bluetooth esté conectado
- Si no lo está, se crea y abre la lista de los dispositivos que el móvil tiene emparejados por Bluetooth. Después de que el usuario elija a qué dispositivo conectarse, no solo se produce el intento de conexión (si falla se devuelve un error) sino que se guarda el nombre del dispositivo (se borran los 19 primeros caracteres que contienen la dirección).
- Si el móvil ya estaba conectado, se desconecta

Al mismo tiempo, los bloques de la Figura 4.5.4 se están continuamente ejecutando pues son parte de un temporizador (componente "Timer") con periodo de 100 milisegundos. Con este se verifica tanto la conexión Bluetooth como si hay datos disponibles para recibir (más de 30 bytes para garantizar que sean datos de variables), en cuyo caso llama a la función "GetData". La asignación de nombres y colores que se observa logra mostrar en la app la Figura 4.5.5, con la que el usuario está informado del estado de la conexión WiFi.

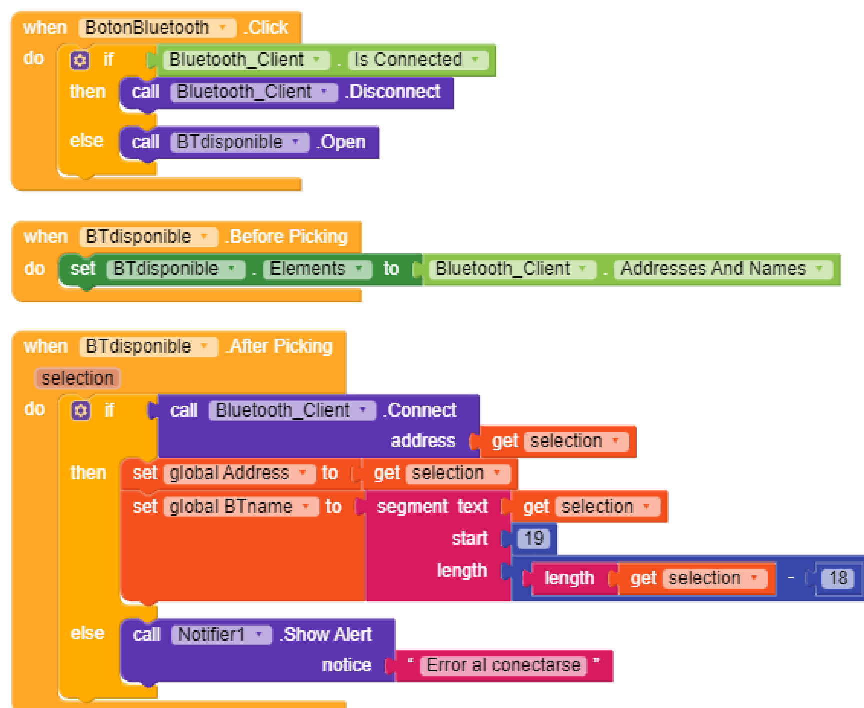


Figura 4.5.3: Bloques ejecutados al pulsar el botón Bluetooth

```
when clockBT .Timer
do  if  Bluetooth_Client . Is Connected
    then  set EstadoBT . Text to  join  " CONECTADO a "
                                        get global BTname
          set BluetoothConnection . Background Color to
          set BotonBluetooth . Background Color to
          set BotonBluetooth . Material Icon Name to " bluetooth_disabled "
          if  call Bluetooth_Client .Bytes Available To Receive > 30
          then  call GetData
    else  set EstadoBT . Text to " DESCONECTADO "
          set BluetoothConnection . Background Color to
          set BotonBluetooth . Background Color to
          set BotonBluetooth . Material Icon Name to " bluetooth "
```

Figura 4.5.4: Bloques ejecutados cada 100 ms por el temporizador Bluetooth

Figura 4.5.5: Componentes Bluetooth visibles para el usuario

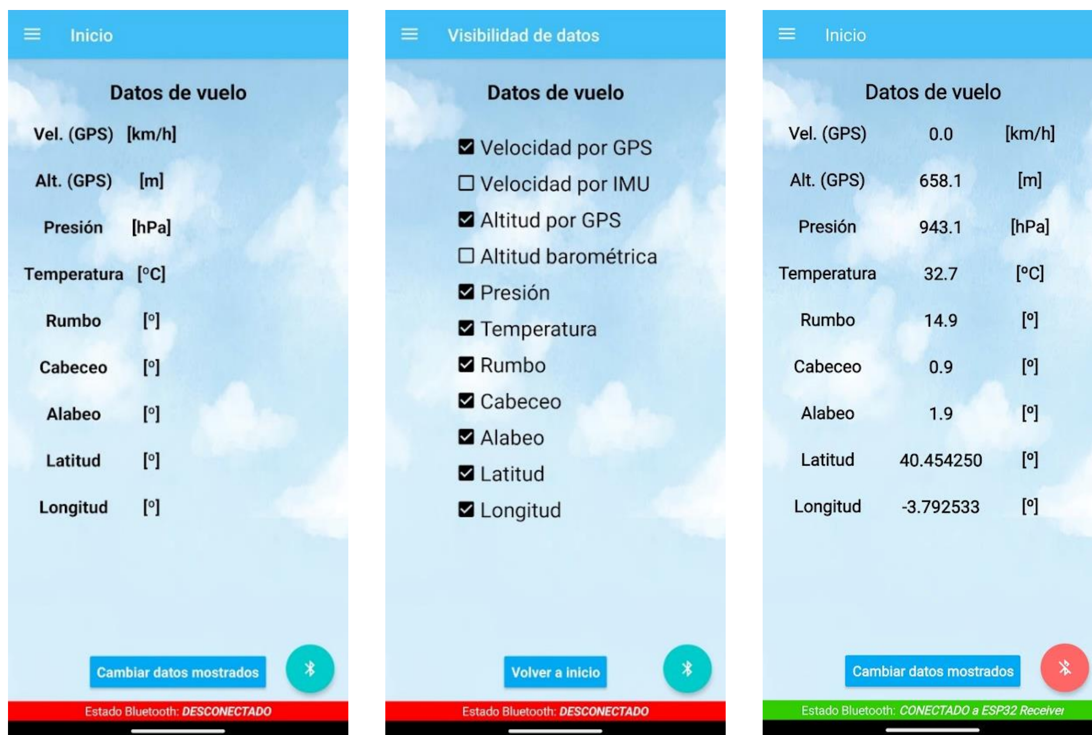
Respecto a la función "GetData" (ver Figura 4.5.6), la lógica que sigue es la siguiente:

1) Obtener cadena de texto (la misma que envía el emisor por LoRa)
2) Separar el texto en una lista de las variables
3) Recorrer la lista verificando que las 11 variables son numéricas. Este paso es muy importante pues la comunicación Bluetooth es muy susceptible a pérdidas de datos
4) Establecer el valor de cada variable (en la figura se muestran la primera y la última)

```
to GetData
do  set global Data to  call Bluetooth_Client .Receive Text
                            number Of Bytes  call Bluetooth_Client .Bytes Available To Receive
    set global DataList to  split text  get global Data
                                  at  " & "
    initialize local checkCorrupt to 0
    in  for each number from 1
                         to  length of list list  get global DataList
                         by 1
        do  if  is number?  select list item list  get global DataList
                                               index  get number
            then  set checkCorrupt to  get checkCorrupt + 1
        if  get checkCorrupt = length of list list  get global DataList
        then  set global V_GPS to  select list item list  get global DataList
                                                  index 1
              set global lon to  select list item list  get global DataList
                                                index 11
```

Figura 4.5.6: Bloques ejecutados cuando hay más de 30 bytes disponibles por Bluetooth

Los datos recibidos son en los que se basan las pantallas 1 y 2. Al abrir la aplicación, se muestra la pantalla Inicio de la Figura 4.5.7a, la cual es una cuadrícula de etiquetas (objetos "Label"): una columna del nombre de la variable y otra con las unidades. Entre ambas está la columna de datos, no visible hasta recibir el primer dato y que se actualiza según se reciben más (ver Figura 4.5.7c). Además, el botón "Cambiar datos mostrados" oculta la cuadrícula y muestra las casillas de selección de la Figura 4.5.7b que, según se marquen o no, activan o deshabilitan la visibilidad del dato en cuestión (toda la fila de la cuadrícula).



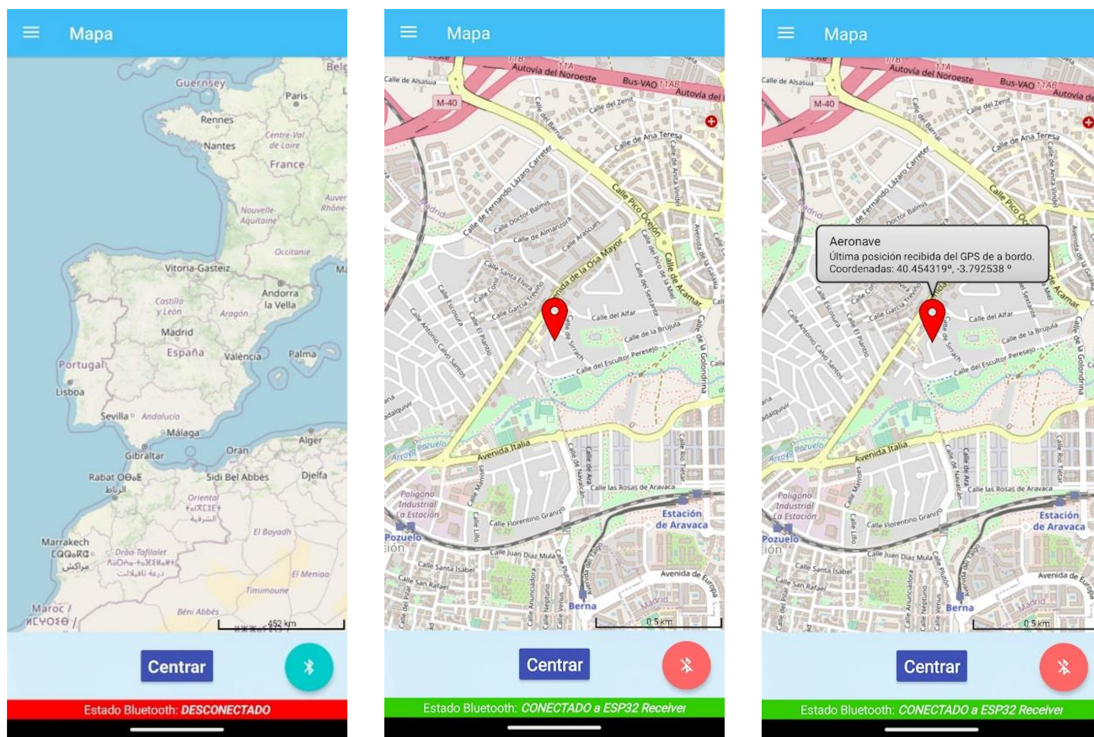
a) Datos de vuelo *b) Visibilidad de datos* *c) Recibiendo datos*

Figura 4.5.7: Pantalla "Inicio"

Respecto a la segunda pantalla, esta se basa en el componente "Maps" que genera un mapa como el de la Figura 4.5.8a. Aunque hay otros diseños disponibles, el que se muestra por defecto es el más detallado y permite ampliarlo a distancias pequeñas, algo necesario para ubicar la aeronave. Además, es más sencillo de hacer que funcione en la aplicación que la alternativa de Google Maps, ya que no requiere una clave API (el acceso a Google Maps Platform está restringido sin una API).

El mapa de la Figura 4.5.8a solo se muestra hasta recibir el primer dato con coordenadas válidas, momento en el cual la imagen se amplia y centra alrededor de la ubicación actual, indicada por un marcador rojo como se ve en la Figura 4.5.8b. Pulsando en el marcador se hace visible el bocadillo informativo de la Figura 4.5.8c, que incluye la latitud y longitud exactas.

Los nuevos datos recibidos recentrarán el mapa manteniendo el zoom, ya sea el predeterminado o el realizado por el usuario. Además, el botón "Centrar" de la parte inferior se puede pulsar en cualquier momento para realizar un nuevo centrado, pues el usuario es libre de mover el mapa.



a) Vista sin ubicación *b) Marcador de posición* *c) Información de marcador*

Figura 4.5.8: Pantalla "Mapa"

La programación por bloques de la pantalla Mapa que logra lo descrito anteriormente se incluye en la Figura 4.5.9. El condicional "if" de la izquierda es, en realidad, el último paso de la función "GetData" (Figura 4.5.6), de forma que la posición se actualice tras recibir el nuevo dato (solo si la pantalla Mapa está visible ya que, de lo contrario, la aplicación se ralentiza por ser un componente que gasta muchos recursos).

En la figura se observa cómo se establecen las coordenadas del marcador y su descripción, tras lo cual se comprueba si la variable "mapa" es nula, es decir, si hasta ahora no se había recibido ningún dato. En ese caso, se hace visible el marcador, se amplía el mapa (de la Figura 4.5.7a a la 4.4.7b) y se llama a la función de retorno "CentrarMapaBoton", que también se ejecuta cuando se presiona el botón "Centrar".

Como podemos ver en esta función (bloques de la derecha), si no se ha recibido ningún dato el centrado lleva al mapa de la Figura 4.5.8a. En caso contrario, se hace alrededor de las últimas latitud y longitud recibidas con el zoom actual.

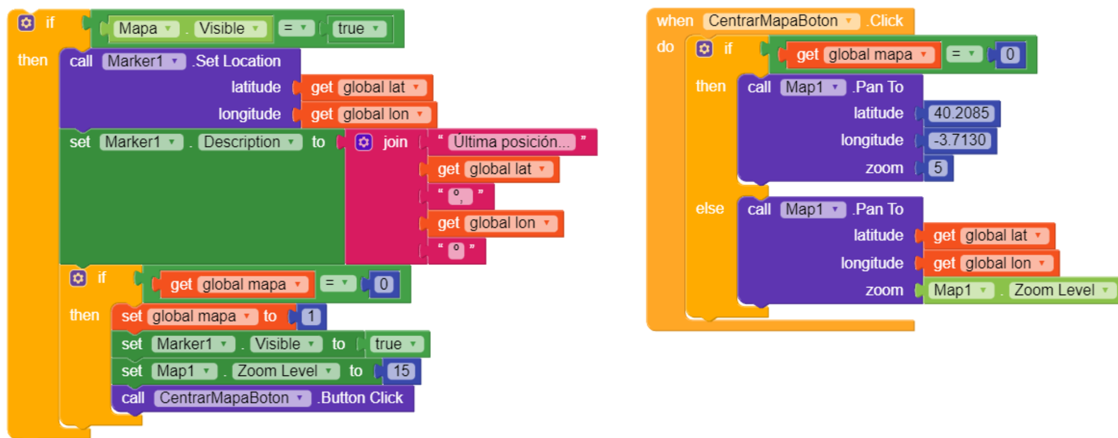


Figura 4.5.9: Bloques de la pantalla "Mapa"

Las dos pantallas explicadas hasta ahora eran puramente receptoras; sin embargo, las restantes se comunican con los ESP32 por medio de los Códigos 4.5.2 y 4.5.4. Por ello, todos los bloques de componentes con los que el usuario puede interactuar (botones, interruptores…) tienen integrado el condicional de la Figura 4.5.10. Este verifica que exista una comunicación Bluetooth activa antes de ejecutar el resto de los bloques; si no, devuelve un error.

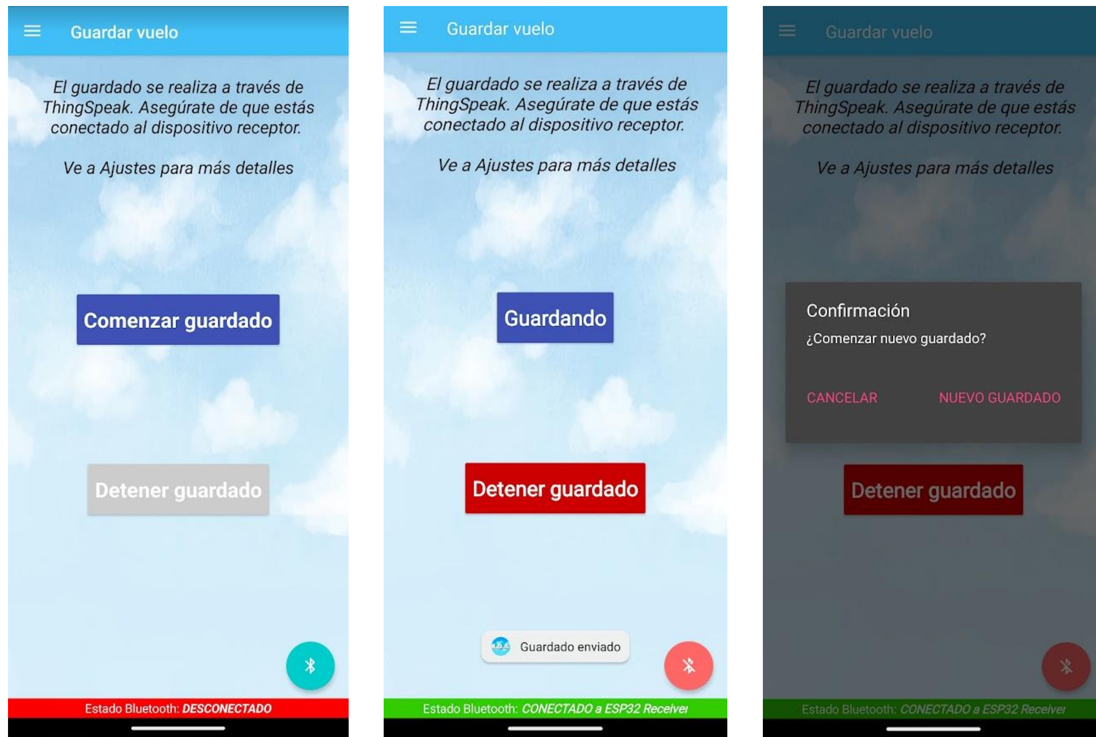
Figura 4.5.10: Comprobación de conexión Bluetooth en pantallas que se comunican con ESP32

La siguiente pantalla es la de "Guardar vuelo", visible en la Figura 4.5.11a, y que funciona con el método de guardado de ThingSpeak por lo que solo se comunica con el ESP32 receptor. De sus bloques asociados de la Figura 4.5.12, los más relevantes son los de envío de un byte (el número 7 para inicio de guardado o el 8 para fin de guardado). El resto tiene las siguientes funcionalidades:

- Cuando se pulse el botón "Comenzar guardado", su texto se cambie a "Guardando" y el color del botón "Detener guardado" se cambie a rojo (ver Figura 4.5.11b).
- Impedir que el botón "Comenzar guardado" envíe un 7 directamente si en su texto pone "Guardando". En cambio, muestra un cuadro de diálogo de confirmación (ver Figura 4.5.11c) ya que puede ser que el usuario desee enviar otro "inicio de guardado". Como se vio en el script de MATLAB de guardado de vuelos (Código 4.3.7), puede haber varios inicios de guardado seguidos y el programa lo procesará sin problema.
- Impedir que el botón "Detener guardado" envíe un 8 directamente si no está rojo. En cambio, muestra un cuadro de diálogo de confirmación (puede ser que el usuario desee

enviar un “fin de guardado” aunque la Air Data App no tenga constancia de que se ha enviado un “inicio de guardado”).

- Al enviarse un “Fin de guardado”, se reestablece la pantalla de la Figura 4.5.11a.



a) Vista inicial *b) Guardado comenzado* *c) Cuadro de confirmación*

Figura 4.5.11: Pantalla “Guardar vuelo”

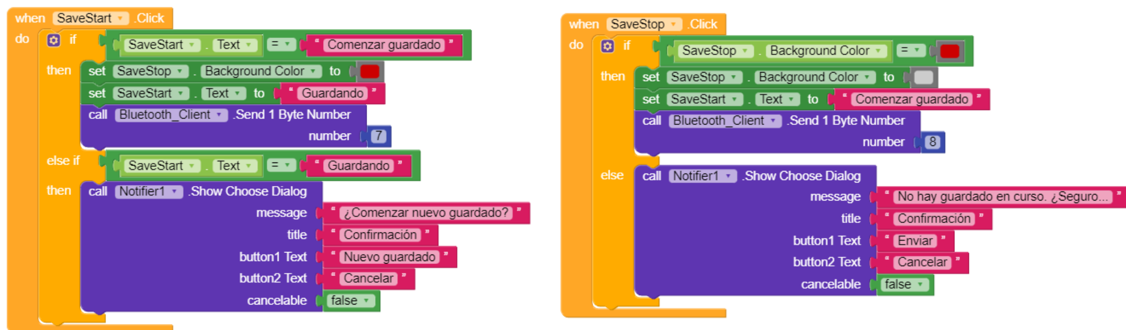
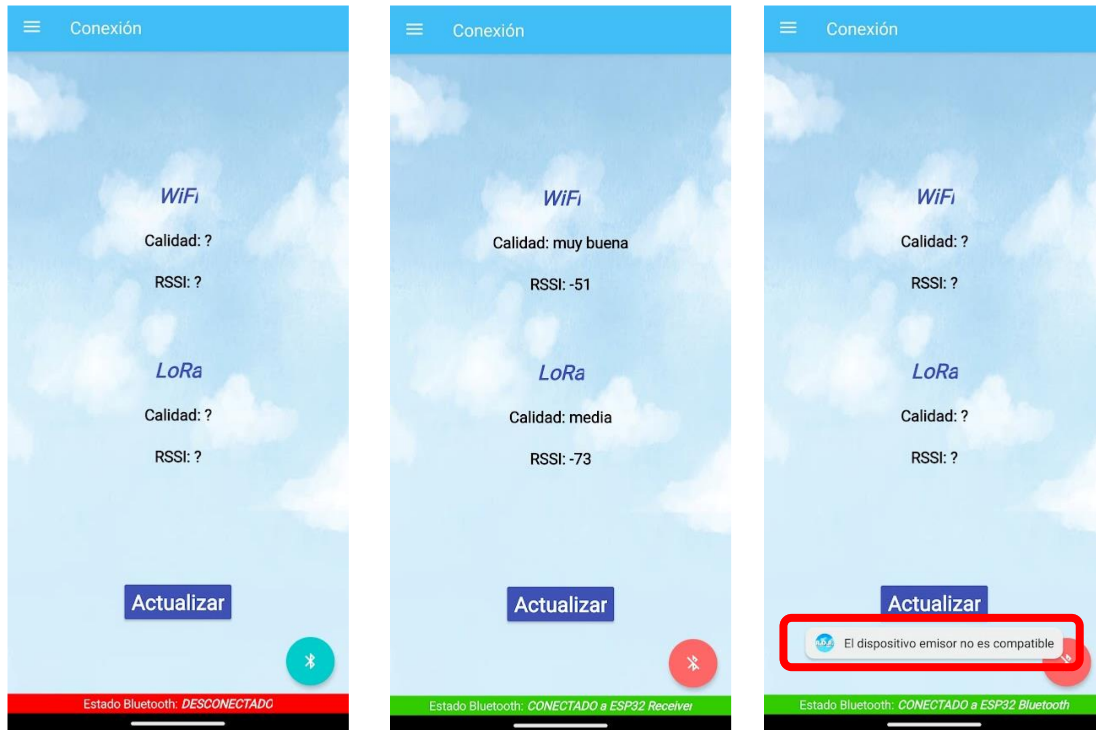


Figura 4.5.12: Bloques de la pantalla “Guardar vuelo”

La penúltima pantalla es la de “Conexión”, Figura 4.5.13, la única que se comunica con un ESP32 para recibir respuesta. Cuando se accede a la pantalla desde el menú lateral o se pulsa el botón “Actualizar”, se ejecutan los bloques de la Figura 4.5.14 donde:

1) Se envía un 3 por Bluetooth
2) Se reciben los valores de RSSI enviados por el ESP32 al que la app esté conectada
3) Si ambos son 1, le hemos preguntado la conexión al emisor, y se devuelve el error destacado en la Figura 4.5.13c.
4) Si alguno es 0, no hay conexión WiFi y/o LoRa (Figura 4.5.13a)

5) En cualquier otro caso, se define con palabras la calidad según el RSSI tanto para WiFi como para LoRa, pudiéndose observar un ejemplo en la Figura 4.5.13b (en la Figura 4.5.14 solo se muestra para WiFi, y el RSSI se procesa en valor absoluto):
    a. Más de -40 dBm: excelente
    b. Entre -40 y -60 dBm: muy buena
    c. Entre -60 y -70 dBm: buena
    d. Entre -70 y -80 dBm: media
    e. Menos de -80 dBm: baja



a) Sin conexión *b) Conexión del receptor* *c) Dispositivo incompatible*

Figura 4.5.13: Pantalla "Conexión"

Figura 4.5.14: Bloques de la pantalla "Conexión"

La última pantalla, “Ajustes”, tiene dos ventanas, una para configurar el receptor (Figura 4.5.15a) y otra para configurar el emisor/aeronave (Figura 4.5.15b). Ambas constan de una serie de botones y cuadros de texto con los que elegir los ajustes deseados y que, al modificar, cambian el color de los botones “Aplicar” a rojo (ver Figura 4.5.15c). Cuando se pulsa alguno de los dos botones “Aplicar”, se ejecutan los bloques correspondientes (izquierda o derecha) de la Figura 4.5.16. Los bloques tanto del emisor como del receptor comparten la misma estructura lógica:

1) Se comprueba que el botón “Aplicar” sea de color rojo. Si no lo es, se avisa de que no hay cambios que aplicar.
2) Se muestra un cuadro de diálogo de confirmación como el de la Figura 4.5.15d. El objetivo es cerciorarse de que el usuario tiene claro los cambios que desea (por ejemplo, se podría haber pulsado el botón por error), recordándole además el dispositivo al que está conectado.
3) Al confirmar, se envía el byte indicador de cambios en la configuración (un 1 para el emisor y un 2 para el receptor) seguido de los bytes

En el momento de envío de datos al ESP32 aparece una complicación: a pesar de que el microcontrolador puede enviar cadenas de texto como ya hemos visto, no puede recibirlas. De hecho, solo puede recibir bytes de uno en uno, lo cual supone un problema para números más grandes que 255 debido a que ocupan más de un byte (1 byte = 8 bits = 2^8 = 256 representaciones numéricas posibles, incluyendo el 0).

Por suerte, este problema es bien conocido, así como su solución, la cual explicaremos para el dato más grande: la presión de referencia. Esta puede tomar valores alrededor de 1000 hPa (nunca menos de 900 o más de 1100), por lo que debería ocupar dos bytes (entre 0 y 2^{16}-1 o 65535). Sin embargo, se permite una precisión de dos decimales, así que multiplicaremos por 100 y tendremos un número como 101325 que el ESP32 se encargará de volver a convertir a hPa. Por tanto, al ser mayor de 65535, el número ocupa 3 bytes, y se puede separar para su envío de la siguiente forma:

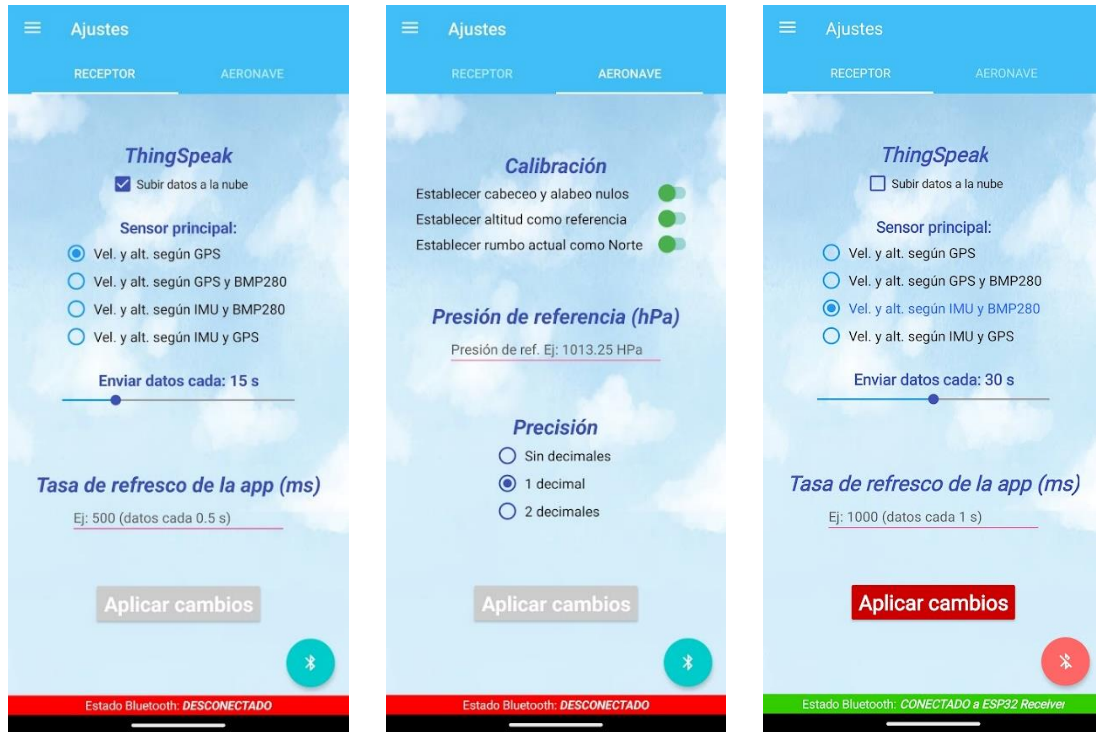
1) Dividimos la presión de referencia entre 65535 y obtenemos su cociente y su resto. El cociente es el primer byte.

   Ej: 101325 = 65536 × 1 + 35789 → cociente = 1, resto = 35790 → byte1 = 1

2) Dividimos el resto anterior entre 256 y obtenemos su cociente y su resto. Este nuevo cociente será el segundo byte, mientras que el resto será el tercer byte.

   Ej: 35790 = 256 × 139 + 205 → cociente = 139, resto = 206 → byte2 = 139, byte3 = 205

Como se puede ver, los tres bytes son menores de 255, y cuando el ESP32 los reciba solo tiene que hacer los cálculos opuestos (65536 × 1 + 256 × 139 + 205 = 101325) o el equivalente del Código 4.5.2. Para enviar la frecuencia de refresco de la aplicación, que es un número de dos bytes (entre 100 y 10000 milisegundos) se hace lo mismo solo que empezando por el segundo paso (se podría hacer el primer paso y el resultado sería el mismo ya que el cociente daría nulo).



a) Ajustes para receptor b) Ajustes para emisor c) Cambios en ajustes

d) Cuadro de confirmación

Figura 4.5.15: Pantalla "Ajustes"

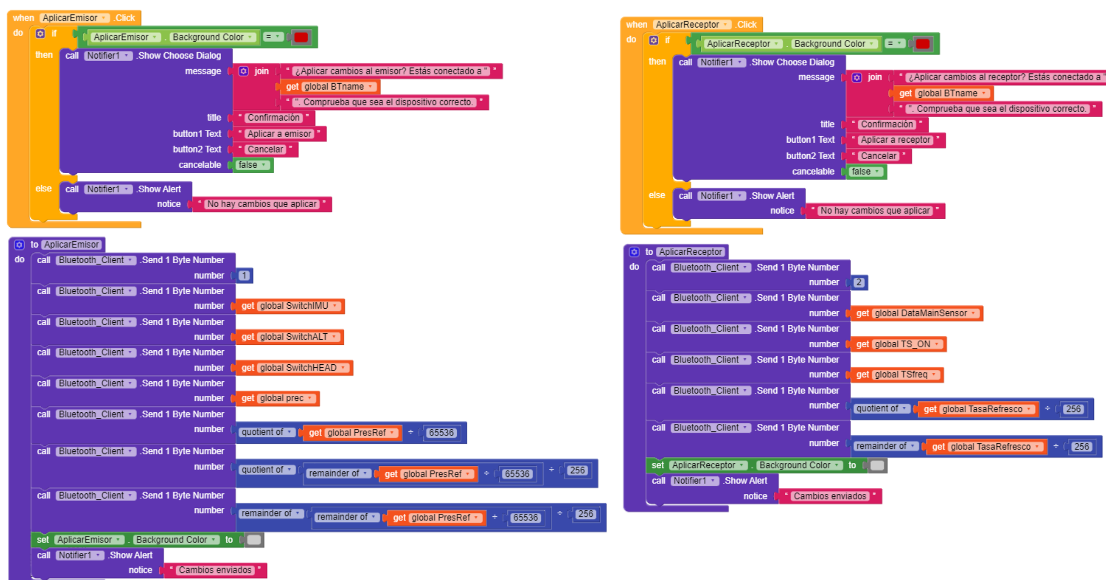


Figura 4.5.16: Bloques de la pantalla "Ajustes"

En cuanto al funcionamiento de la interacción con los botones de los ajustes, algunos son sencillos como los interruptores del apartado de “Calibración”. Los bloques de este caso son simplemente una función de retorno del evento “Switch.clicked”, que comprueba si se ha activado o desactivado y establece el valor de la variable a enviar (0 o 1) según corresponda.

En cambio, los cuadros de texto tienen cierta complejidad causada por el hecho de que el usuario puede introducir cualquier número. Como primera medida para evitar errores, las Figuras 4.5.15a y 4.5.15b muestran que, sin introducir ningún número, aparece un texto con un ejemplo de qué poner. Además, para impedir que se envíe a los ESP32 (receptor y emisor) un valor imposible, los bloques de la Figura 4.5.17 se ejecutan cuando hay un cambio en el texto del cuadro y obedecen la siguiente lógica:

1) Se comprueba que el cuadro de texto no esté vacío. Si lo está, se establece la presión de referencia a la predeterminada de 1 atmósfera.
2) Se comprueba que el valor introducido esté en los límites establecidos:
    a. Para la presión de referencia, entre 870 y 1080 hPa. Estos son récords de presión mínima (medida en un tifón, [12]) y máxima (por debajo de 750 metros, [13]).
    b. Para la frecuencia de refresco, un período de entre 100 y 15000 milisegundos. Estos son la frecuencia a la que el emisor envía datos (mínima) y la de ThingSpeak para una cuenta gratuita (15 segundos).
3) Si está fuera de límites, se devuelve un error. Si no lo está, se cambia el color del botón “Aplicar” a rojo para indicar el cambio y se guarda el valor procesándolo antes:
    a. Para la presión de referencia, se toman los dos primeros decimales (para el caso en el que el usuario ponga tres o más) y se multiplica por 100.
    b. Para la frecuencia de refresco, se redondea para eliminar cualquier decimal que el usuario haya podido introducir.

Figura 4.5.17: Bloques asociados a los cuadros de texto de la pantalla “Ajustes”

Para terminar con la explicación de la Air Data App, en el menú lateral de la Figura 4.5.1 es visible una última opción llamada “Sobre esta app”. Al pulsarla, simplemente se muestra una breve descripción de la aplicación en el cuadro de diálogo de la Figura 4.5.18.

Figura 4.5.17: Cuadro de diálogo "Sobre esta app"

5. VALIDACIÓN

Con el objetivo de validar el sistema de monitorizado y almacenado de datos de vuelo, se ha realizado una prueba de vuelo. Para completarla, se acopló el emisor a un dron cuadricóptero de acrobacias para hacer un recorrido de diferentes movimientos, velocidades y altitudes. Dicho vuelo tuvo lugar en el club de aeromodelismo de L'Abella, Valencia (ver Figura 5.1), una zona rural y abierta.

Figura 5.1: UAV con prototipo despegando en el Club de Aeromodelismo de L'Abella

5.1.1 Condiciones de la prueba

El hardware del emisor (sensores, microcontrolador y LoRa) se instaló en una placa de pruebas pues sus dimensiones se ajustaban al UAV y facilita arreglos y modificaciones en el prototipo gracias a que las soldaduras son innecesarias. La Figura 5.2 muestra la distribución cada uno de los componentes del emisor, donde es visible que el espacio empleado se puede reducir en gran medida. El receptor, en cambio, es mucho más sencillo ya que se compone del microcontrolador y el módulo LoRa, y se puede ver en la Figura 5.3.

Para la prueba, se analizaron las condiciones del vuelo y se decidió que, debido a la velocidad y agilidad del dron en cuestión, la mejor opción era incrementar el muestreo al máximo (cada 100 milisegundos). Por este motivo, se desactivó la opción de guardado en nube, la cual recordemos que ralentiza la ejecución del código alrededor de medio segundo, para centrarse en el funcionamiento de las aplicaciones móvil y de PC.

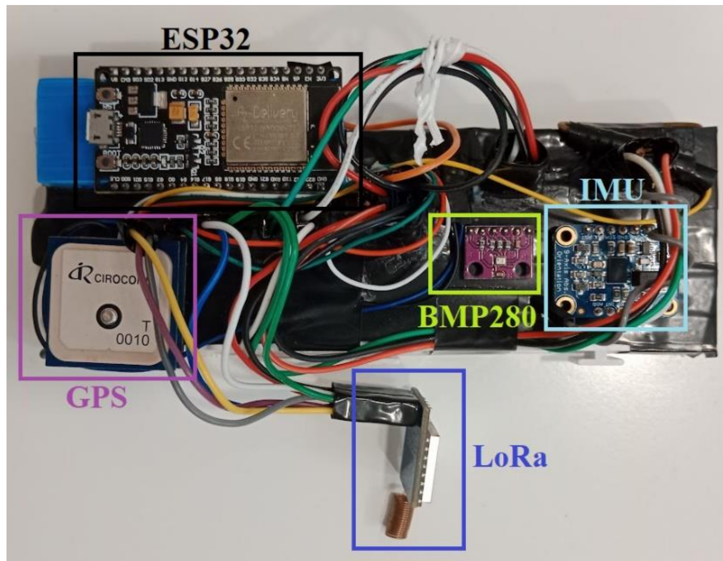


Figura 5.2: Emisor

Figura 5.3: Receptor

Respecto al servidor remoto, este no era el objetivo en la prueba de vuelo puesto que su funcionamiento ya se había comprobado en otras condiciones. Por otro lado, al estar empleando la versión gratuita de ThingSpeak, la calidad de los resultados con una frecuencia de un envío de datos por segundo ha de estudiarse a través del almacenamiento local. No obstante, a continuación se aportarán los datos recogidos durante un breve tiempo mientras se configuraba el sistema antes de la prueba de vuelo.

5.1.2 Resultados

En total, el vuelo duró 5 minutos y 32 segundos, durante el cual se recogieron 2417 muestras, es decir, 7.28 muestras por segundo. Eso implica una reducción del 28.2% en la velocidad de envío, la cual está establecida en 10 muestras por segundo. Asimismo, de estas muestras, 50 estaban corruptas, lo que significa que se perdieron el 2.07% de los datos enviados por el emisor.

Como consecuencia, se puede concluir que el envío de datos tiene una alta robustez y fiabilidad por el bajo número de muestras corruptas, pero existen retrasos arraigados en el software y hardware que ralentizan la frecuencia de envío. Ante este problema, una acción posible es un análisis exhaustivo del código para optimizarlo, aunque no es garantía de solución. Una alternativa es bajar el período de muestreo para que el retraso inherente se compense y dé lugar a un promedio más cercano a 100 milisegundos.

Los resultados obtenidos, al almacenarse por la aplicación de PC, aparecen registrados en una tabla como la de la Figura 5.4, que es un extracto de parte del vuelo. Se puede observar que las columnas incluyen todos los datos que el emisor envía, además de información adicional acerca de qué instante se tomó la medida (con precisión de segundos). Aquí destacan dos medidas por sus valores defectuosos: la velocidad inercial de la IMU y la altitud del GPS.

	A	B	C	D	E	F	G	H	I	J	K	L	M
	AirDataAPP09Jul2023154448												
	Tiempo	VelocidadG...	VelocidadI...	AltitudGPS	AltBarom	Presion	Temperatura	Rumbo	Cabeceo	Alabeo	Latitud	Longitud	VarName13
	Categorical	Number	Number	Number	Number	Number	Number	Number	Number	Number	Number	Number	Text
1257	'09-Jul-202...	58.1	72.5	-13	34.9	1019	33.1	152.1	-60.1	-12.2	39.176128	-0.53409	
1258	'09-Jul-202...	58.1	72.6	-13	35.3	1019	33.1	151.9	-60.2	-12.2	39.176128	-0.53409	
1259	'09-Jul-202...	58.1	72.8	-13	35.4	1019	33.1	151.9	-59.9	-11.8	39.176128	-0.53409	
1260	'09-Jul-202...	76.2	73	-13.2	35.9	1018.9	33.1	151	-60.2	-12.5	39.175949	-0.534058	
1261	'09-Jul-202...	76.2	73.1	-13.2	35.3	1019	33.1	149.4	-60.3	-12.1	39.175949	-0.534058	
1262	'09-Jul-202...	76.2	73.3	-13.2	36.2	1018.9	33.1	147.9	-60.6	-11.9	39.175949	-0.534058	
1263	'09-Jul-202...	76.2	73.4	-13.2	35.9	1018.9	33.1	146.7	-60.8	-11.7	39.175949	-0.534058	
1264	'09-Jul-202...	76.2	73.5	-13.2	34.6	1019	33.1	144.8	-61.8	-10.5	39.175949	-0.534058	
1265	'09-Jul-202...	76.2	73.6	-13.2	33.6	1019.2	33.1	140.3	-62.6	-4.3	39.175949	-0.534058	
1266	'09-Jul-202...	76.2	73.7	-13.2	32.5	1019.3	33.1	131.9	-63.4	5.6	39.175949	-0.534058	
1267	'09-Jul-202...	88.5	73.8	-13.2	22.5	1020.5	33.1	115.3	-60.9	14.5	39.175732	-0.534052	
1268	'09-Jul-202...	88.5	73.9	-15.3	33.1	1019.2	33.1	95.6	-57.6	12.3	39.175732	-0.534052	
1269	'09-Jul-202...	88.5	74	-15.3	58.1	1016.2	33.1	79.2	-53.1	-3.7	39.175732	-0.534052	
1270	'09-Jul-202...	88.5	74.1	-15.3	64.4	1015.5	33.1	67.2	-52.2	-16.3	39.175732	-0.534052	
1271	'09-Jul-202...	88.5	74.1	-15.3	62.1	1015.7	33	63.4	-52.4	-18.6	39.175732	-0.534052	
1272	'09-Jul-202...	88.5	74.2	-15.3	58.2	1016.2	33	63.6	-53.1	-17.2	39.175732	-0.534052	
1273	'09-Jul-202...	88.5	74.3	-15.3	55.2	1016.5	33	64.1	-54	-15.6	39.175732	-0.534052	
1274	'09-Jul-202...	88.5	74.4	-15.3	53	1016.9	33	61.6	-54.9	-16.9	39.175732	-0.534052	
1275	'09-Jul-202...	73.5	74.5	-15.3	48.5	1017.2	33	57.4	-55.8	-21.5	39.175549	-0.53398	
1276	'09-Jul-202...	73.5	74.6	-17	50.6	1017.1	33	51.4	-54.9	-29.8	39.175549	-0.53398	

Figura 5.4: Segmento de tabla de datos recogidos en la prueba de vuelo

En primer lugar, la velocidad inercial la calcula el acelerómetro del eje *x* que, idealmente, ha de estar alineado con la dirección de avance. En una aeronave de ala fija esto es aproximadamente cierto, en especial en crucero, pero este dron vuela normalmente con 60 grados de cabeceo negativo ("mirando" hacia el suelo, ver columna I de la Figura 5.4), lo cual conduce a error.

De todas formas, aunque la magnitud difiera, la tendencia (aceleraciones y deceleraciones) debería mantenerse. No obstante, esto no ocurre, observándose una mayor sensibilidad a las aceleraciones que las deceleraciones, lo que causa que la velocidad crezca continuamente. Este problema es habitual e incluso esperable porque este método necesita gran precisión y se suele combinar con sistemas de control y filtros especializados. La mejor solución es aprovechar la redundancia en este valor y emplear la alternativa del GPS.

En cuanto a la altitud del GPS, la redundancia también ha resultado clave ya que esta medición presenta una inexactitud y variabilidad que no es tolerable. Si se crea un mapa 3D con estas alturas, el resultado es el de la Figura 5.5 que, comparado con el de la Figura 5.6, basado en la altitud barométrica, se concluye que hay un claro error. La Figura 5.6 tampoco es perfecta por la menor precisión del instrumento, lo que da lugar a escalones de altitud, pero es visiblemente más realista y entendible.

A causa de la duración del vuelo y la pequeña área cubierta, la trayectoria completa se solapa y puede confundir. Para facilitar su visionado, las Figuras 5.7 y 5.8 muestran el recorrido en dos y tres dimensiones, respectivamente, de un tramo del vuelo. Para crearla se hace uso de la función *geoplot3*, visible en el Código 5.1. De esta forma, es posible comprobar que el guardado de datos en local con la aplicación de MATLAB proporciona una trayectoria de calidad, permitiendo estudiar el vuelo del UAV con exactitud.

A pesar de que dicho guardado fue satisfactorio, durante el visualizado con la aplicación de PC se produjeron problemas de rendimiento. Concretamente, la pestaña del mapa con la ubicación de la aeronave en teoría se actualiza de forma continua para dibujar la trayectoria tal y como se comprobó durante su desarrollo, pero en las condiciones de la prueba de vuelo la aplicación se ralentizaba y colgaba.

Esto se debe, por un lado, a una pobre optimización de la función *geoplot*, que consume muchos recursos, y, por otro lado, a un ordenador portátil muy solicitado en esos momentos (batería baja, programas en segundo plano…) y con especificaciones técnicas no demasiado altas. En comparación, las otras dos pestañas (datos y panel de instrumentos) junto a la aplicación móvil se desenvolvieron de la forma esperada y su funcionamiento resultó ser un éxito.

```
% Mapa 3D
uif = uifigure;
g = geoglobe(uif);
geoplot3(g,lat,lon,alt,"r-o","HeightReference","terrain")
```
Código 5.1: Generación de trayectoria 3D en MATLAB

Figura 5.5: Trayectoria 3D con la altitud del GPS

Figura 5.6: Trayectoria 3D con la altitud del altímetro barométrico

Figura 5.7: Vista de pájaro de la trayectoria de un tramo del vuelo

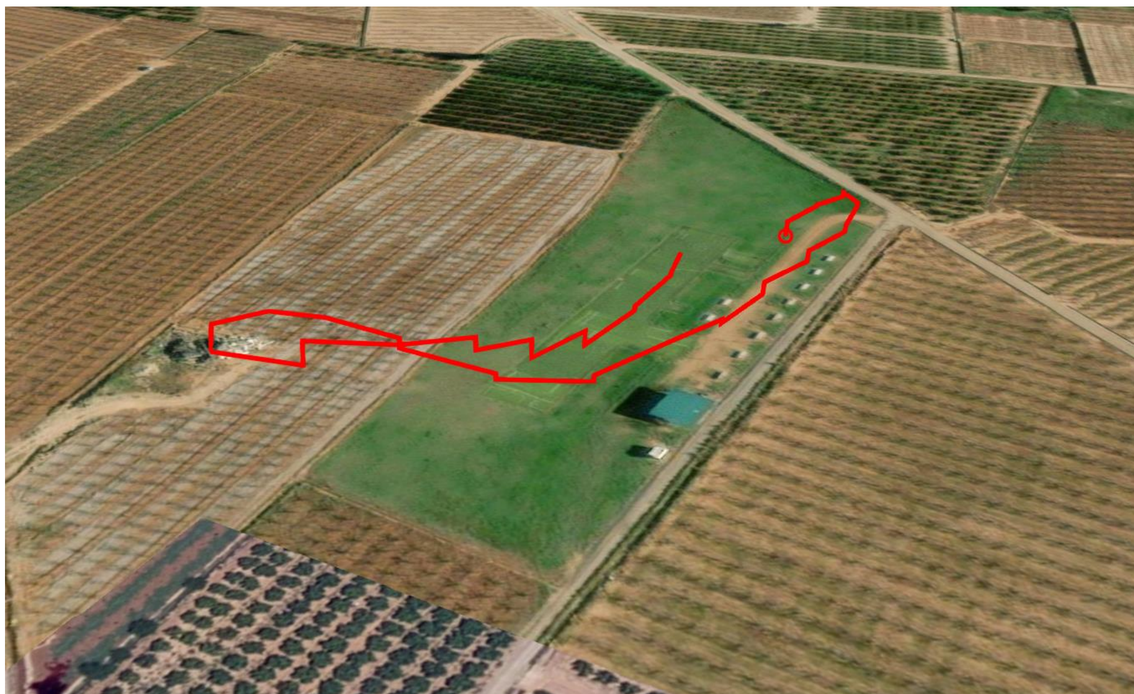


Figura 5.8: Trayectoria 3D de un tramo del vuelo

A continuación, trataremos la aplicabilidad del servidor remoto en estas condiciones. Antes de la prueba de vuelo, mientras se configuraba el sistema, se dejó activado ThingSpeak para validar la conexión WiFi y su funcionamiento en la zona. Estos datos se subieron perfectamente a la plataforma como se aprecia en las interfaces gráficas de la Figura 5.9, que registran las altitudes erróneas del altímetro (segunda fila, segunda columna).

Figura 5.9: Visualizado de datos subidos a ThingSpeak

El siguiente paso es analizar el resultado que tendría en el registro si pasamos de diez muestras por segundo a una, es decir, qué datos tendríamos almacenando en nube con ThingSpeak usando la versión de pago. Para ello, la Figura 5.10 incluye todos los datos recibidos mientras que la Figura 5.11 presenta una décima parte.

Es posible observar comparando ambas figuras que se pierde información, notable en las dos vueltas realizadas hacia el norte. Al mismo tiempo, la realidad es que el UAV de estudio es un caso extremo debido a su alta maniobrabilidad y velocidad (en dichas vueltas se alcanzaron velocidades de 80 km/h girando). A pesar de esto, la forma general de la trayectoria se conserva, y es de esperar que, para recorridos más lentos o largos con giros menos pronunciados, el muestreo de un dato por segundo proporcione resultados mucho más exactos.

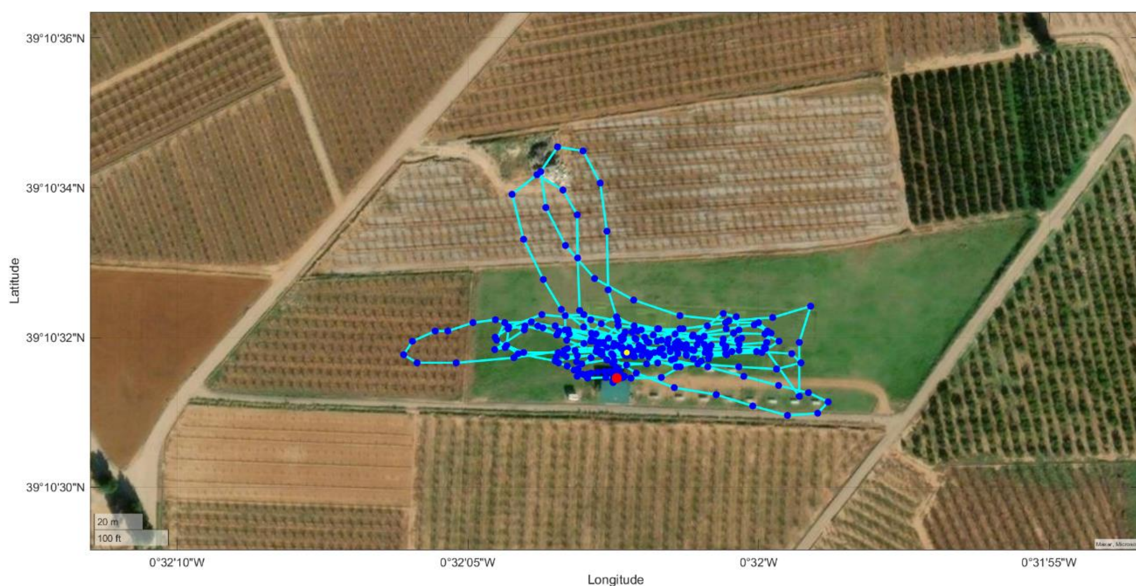


Figura 5.10: Vuelo completo a vista de pájaro con un muestreo de ~10Hz

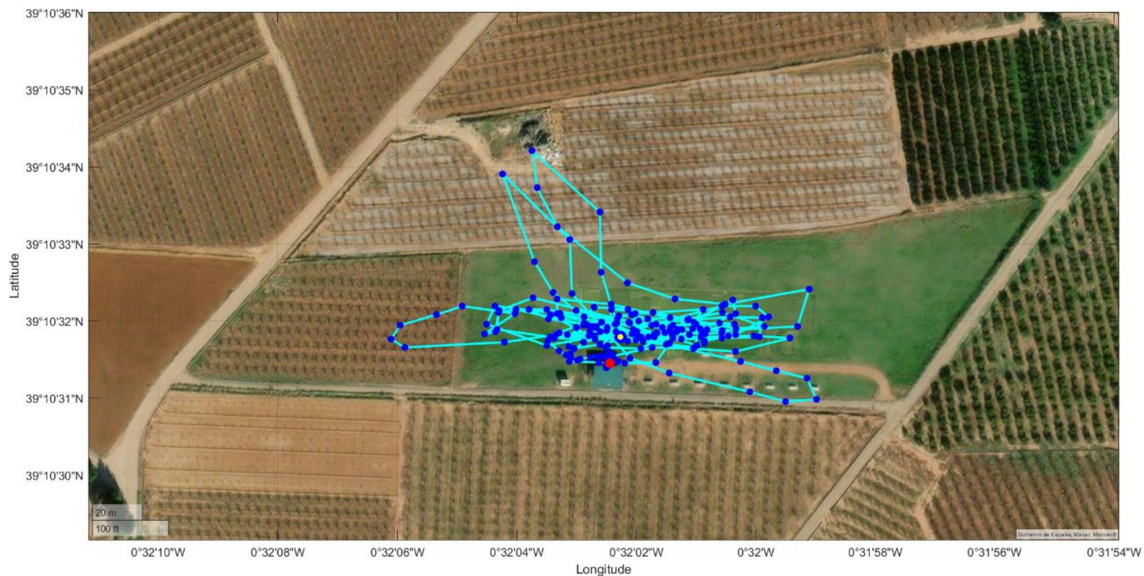


Figura 5.11: Vuelo completo a vista de pájaro con un muestreo de ~1Hz

Por último, cabe destacar que también fue posible estudiar el consumo energético del sistema. Aun con una temperatura más alta que en otras pruebas, no hubo notables diferencias en la duración de la batería, que fue de 2.5 horas. Recordando que dicha batería es de 3.7V y 820 mAh, la potencia (voltaje × intensidad) requerida por el prototipo se calcula como:

$$P = V \cdot I = \frac{3.7\ [\mathrm{V}] \cdot 0.820\ [\mathrm{A} \cdot \mathrm{h}]}{2.5\ [\mathrm{h}]} \approx 1.2\ [\mathrm{W}]$$

Que la potencia sea de 1.2 vatios (W) es equivalente a decir que el prototipo usa 1.2 julios por segundo. En comparación, este es el consumo más bajo que podemos encontrar en bombillas para testigos de coche en la página web de AutoBulbsDirect, un especialista de iluminación para automoción en Reino Unido. Un ejemplo de estas es la R509TDBK de 1.2W, mostrada en la Figura 5.12. En definitiva, es posible afirmar que se ha logrado el objetivo de minimizar el consumo energético del sistema desarrollado.

Figura 5.12: Bombilla R509TDBK de 1.2W para indicador de panel de coche
Imagen de Ring, el fabricante

6. CONCLUSIONES

A lo largo de este documento, se ha planteado el problema que afecta a un gran número de UAVs con respecto a la ausencia de una comunicación de datos de vuelo entre aeronave y operadores. Esto implica un impedimento no solo para el piloto, quien depende de estimaciones visuales para volar, sino también de cualquier interesado en tener constancia de la evolución de la misión en tiempo real.

Ante esta situación, se expusieron una serie de necesidades con el foco en aplicabilidad para el público general, obligando a la solución a ser sencilla, fiable, versátil y económica, entre otros. Estas cualidades debían estar maximizadas en todos los componentes del sistema desarrollado: microcontroladores, sensores, telecomunicaciones y plataformas. Tras un análisis detallado de las distintas opciones, se llegó a las siguientes propuestas:

- ESP32 como microcontrolador: aunque de mayor tamaño que sus alternativas, este modelo es superior en cuanto a conexiones posibles (más posibilidades para incorporar nuevos sensores) y coste. En las pruebas ha demostrado un rendimiento excelente a pesar de las exigencias de procesamiento (sensores y envío en el emisor, recepción y reenvío a plataformas en el receptor).

- BN-880 como GPS: elegido por estar disponible en el Departamento de Electrónica de la ETSID (UPV), su desempeño recogiendo latitud, longitud y velocidad fue muy bueno, pero la obtención de la altitud resultó defectuosa, proveyendo de valores prácticamente aleatorios.

- BNO055 como IMU: a pesar de ser el dispositivo más caro, los datos de orientación (cabeceo, alabeo, guiñada) son críticos en una aeronave y el hardware y software de este sensor inteligente proporciona una fiabilidad de la más alta calidad. No obstante, emplearlo para conocer la velocidad no fue posible por la naturaleza compleja del cálculo, que es muy susceptible a errores.

- BMP280 como sensor de presión y temperatura: aunque en un principio innecesaria y de precisión mejorable, la altitud barométrica resultó ser indispensable para sustituir la errónea del GPS. Tanto la temperatura como la presión (y la altitud) probaron ser medidas útiles y fiables.

- LoRa como telecomunicación: este tipo de LPWAN suple su menor alcance (suficiente como mínimo de todos modos) con unos protocolos de comunicación extremadamente robustos, alta compatibilidad con el ESP32, elevada velocidad de envío y módulos pequeños y ligeros.

- App Designer para aplicación de PC: este entorno de desarrollo tiene integrado instrumentos de vuelo que facilitan simular un panel de instrumentos y usa el lenguaje de programación MATLAB con el que el autor está familiarizado. La aplicación resultante es la opción más potente ya que permite visualizar los datos de vuelo tanto brutos como visualmente con la mayor fluidez posible (10 Hz), seguir el recorrido del UAV en el mapa en tiempo real, guardar datos en local y activar o desactivar la subida de datos a la nube. La desventaja es que requiere disponer de un PC en el lugar del vuelo, preferiblemente con buen rendimiento.

- Kodular para aplicación móvil: esta desarrolladora de aplicaciones móvil se seleccionó principalmente por la facilidad que supone su programación en bloques y la sencillez de su interfaz, conveniente para alguien con poca experiencia. La aplicación creada es similar pero más limitada que la de PC pues no incluye guardado en local (aunque podría implementarse con más conocimientos) ni panel de instrumentos. Por otro lado, presenta ventajas únicas como ser para móvil y tener conexión Bluetooth, por lo que su disponibilidad es inmejorable, y permite cambiar inalámbricamente varios ajustes de ambos microcontroladores.

- ThingSpeak como servidor remoto: esta plataforma usa convenientemente MATLAB como lenguaje de programación, admitiendo el uso de sus potentes herramientas de visualizado y análisis de datos, su complejidad es baja, acepta hasta ocho variables en un canal y con un pago relativamente bajo admite una subida (ocho datos) por segundo. Por medio del servidor se puede acceder remotamente en tiempo real, aunque con un retraso de segundos que restringe su uso en pleno pilotaje.

En definitiva, la solución adoptada es el resultado de una serie de meticulosas decisiones en cuanto a sus componentes, consiguiéndose así un sistema de monitorizado, procesado y almacenamiento de datos que cumple con los objetivos propuestos. Entre ellos, destaca la versatilidad que las tres plataformas logran, dificultando enormemente que exista algún escenario en que ninguna de ellas sea adecuada.

En concreto, el almacenamiento en servidor remoto en tiempo real abre las puertas a la maximización de la eficiencia y calidad del monitorizado. Casi instantáneamente, todos los datos que recoge la aeronave están a disposición de cualquier persona alrededor del mundo con conexión a Internet. Esto permite a un analista de datos trabajar en paralelo sin esperar a que le envíen nada, un responsable dirigir las operaciones y hacer correcciones sin estar presente, un cliente verificar que el servicio del UAV que ha contratado se está realizando, etcétera.

Aun con el gran impacto positivo del sistema propuesto, no deja de ser un prototipo en el que se vieron defectos como las medidas de sensores erróneas, el bajo rendimiento del mapa de la aplicación de PC y la velocidad de envío reducida. Sin embargo, estos han de ser objetivo de mejora a través de análisis técnicos y de optimización. A su vez, aspectos negativos son oportunidades de expansión, como es la incapacidad de guardar en local con la aplicación móvil o cambiar ajustes con la aplicación de PC.

Para concluir, este estudio ha derivado no solo en una solución realista, generalizada y modificable según las necesidades, sino también en descubrir el potencial que herramientas ya existentes tienen en la aeronáutica. Junto al avance tecnológico y pragmático de los UAVs, las posibilidades de un sistema así crecerán, tiempo durante el cual se progresará a versiones superiores con los problemas anteriores resueltos.

7. BIBLIOGRAFÍA

[1] Carbonnelle, P. *Top IDE Index.* Accedido el 19 de mayo de 2023 desde: https://pypl.github.io/IDE.html

[2] Arduino. *Arduino Nano 33 IoT Product Reference Manual*, 2023. Disponible en: https://docs.arduino.cc/resources/datasheets/ABX00027-datasheet.pdf

[3] Espressif. *ESP32 Series Datasheet*, 2023. Disponible en: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[4] Adafruit. *BNO055 Absolute Orientation Sensor's Overview*, 2015. Accedido desde: https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/overview

[5] Sanchez-Iborra R., Cano M. D. *State of the Art in LP-WAN Solutions for Industrial IoT*, 2016. Disponible en: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4883399/

[6] Centenaro M., Vangelista L., Zanella A., Zorzi M. *Long-range communications in unlicensed bands: The rising stars in the IoT and smart city scenarios*, 2016. Disponible en: http://arxiv.org/pdf/1510.00620v1.pdf

[7] Yanoff, Myron; Duker, Jay S. *Ophtalmology 3rd Edition,* 2009. Publicado por MOSBY Elsevier, p. 54. ISBN 978-0444511416.

[8] Ingenu Inc. *RPMA vs. Competition.* Accedido el 23 de mayo de 2023 desde: https://www.ingenu.com/technology/rpma/competition/?doing_wp_cron=1689784976.7305018901824951171875

[9] Pais, N. *IoT Platforms Comparison: AWS, Azure, Google, IBM, Cisco*. Accedido el 24 de mayo de 2023 desde: https://www.31west.net/blog/iot-platforms-comparison-aws-azure-google-ibm-cisco

[10] Moreno Castro, M y Martinez Hernandez, P. *Estudio comparativo de plataformas cloud que ofrecen servicios IoT*, 2021. Disponible en: https://repository.usta.edu.co/handle/11634/33347

[11] Martínez Jacobson, R. *Comparativa y estudio de plataformas IoT*, 2017. Disponible en: https://upcommons.upc.edu/bitstream/handle/2117/113622/TFG%2dRodrigoMartinezJacobson.pdf?sequence=1&isAllowed=y

[12] Landsea, C. *Hurricanes: frequently asked questions*, 2010. Disponible a través del Atlantic Oceanographic and Meteorological Laboratory en: https://www.aoml.noaa.gov/hrd-faq/#1569507388495-a5aa91bb-254c

[13] World Meteorological Organization. *World: Highest Sea Level Air Pressure Below 700m*, 2016. Disponible en: https://wmo.asu.edu/content/world-highest-sea-level-air-pressure-below-700m

PRESUPUESTO

PRESUPUESTO

A lo largo de este documento, se ha hecho mención en numerosas ocasiones al coste, siendo denominado un factor fundamental. En este bloque, se tratará el valor económico con exactitud, diferenciando entre los gastos de los componentes del sistema, los relacionados con software y los de desarrollo (mano de obra).

1. COSTE MATERIAL

En la Tabla 1.1 se presenta el desglose del coste del material que componen el sistema de monitorizado y almacenado. Estos no incluyen cableado y placa de pruebas, que se asocian al prototipo de pruebas y fueron proporcionados por el Departamento de Electrónica de la ETSID (UPV), ni tampoco gastos de envío por ser dependientes de factores como ubicación o cantidad de compra.

Asimismo, se ha decidido separar el material indispensable del sistema con el opcional para una mejor comprensión de los costes. Los componentes opcionales son los de la alimentación externa (la fuente energética puede ser la del UAV) y un cable USB con extremos tipo micro-B y A con transmisión de datos (es muy común y solo es indispensable para la aplicación de PC).

Componente	Unidades	Coste unitario	Distribuidor	Importe
Material indispensable				
Microcontrolador ESP32	2	11.49€	AZ-Delivery	22.98€
Módulo LoRa SX1278	2	3.84€	AliExpress	7.68€
Sensor BMP280	1	2.90€	Satkit	2.90€
IMU BNO055	1	13.89€	DigiKey	13.89€
GPS BN-880	1	14.18€	AliExpress	14.18€
Subtotal:				**61.63€**
Material opcional				
Batería LiPo 3.7V 820 mAh	1	11.29€	Amazon	11.29€
PowerBoost 500C	1	13.60€	Adafruit	13.60€
Cable USB-A a micro B	1	1.18€	SuperParts	1.18€
Subtotal:				**26.07€**
Material completo				
TOTAL:				**87.70€**

Tabla 1.1: Presupuesto de material

2. COSTE DE SOFTWARE

Durante la elaboración de la parte de software del proyecto, para reducir costes se han empleado plataformas sin cargo como Kodular (aplicación móvil) o Arduino (microcontrolador), mientras que MATLAB está disponible a través de la universidad. Si además se usa la versión gratuita de ThingSpeak, entonces el coste de software es nulo.

No obstante, como se ha argumentado anteriormente, realizar una subida de datos cada segundo es lo indicado en un UAV, por lo que se ha de considerar este caso. El precio que permite esto es de 55 euros al año, el cual es conveniente relativizar con respecto a cuánto se vuela. La Tabla 2.1 recoge algunos casos realistas, donde se observa que para un UAV que vuela ocasionalmente (una hora a la semana), la hora de vuelo almacenada en ThingSpeak sale a poco más de un euro. Además, hay que tener en cuenta que se pueden configurar varios UAVs para que suban sus datos a una misma cuenta de ThingSpeak (con la limitación de que no sea al mismo tiempo), lo que aumentarían las horas de vuelo y reduciría el coste relativo.

Plan de ThingSpeak	Coste anual	Horas de vuelo	Coste/hora
Hasta 8 variables Una subida cada 15 segundos	Gratuito	-	Gratuito
Hasta 8 variables Una subida por segundo	55€	1 a la semana	1.06€
		1 al día	0.15€

Tabla 2.1: Presupuesto de ThingSpeak

3. COSTE HUMANO

Por último, se han de considerar las horas de trabajo en el desarrollo del proyecto. El coste está asociado al campo de la ingeniería aeronáutica que, aunque se lleven a cabo distintas tareas de programación y electrónica, entre otras, necesitan del criterio de una persona con conocimientos aeronáuticos para un enfoque y resultado correctos.

El gráfico de la Figura 3.1 refleja el primer salario de los graduados del curso 2013/14 en Ingeniería Aeronáutica según el Instituto Nacional de Estadística (INE). De aquí podemos extraer que el rango salarial más común es de entre 1500 y 1999 euros, aunque la mayoría se encuentra por debajo. Por este motivo, tomaremos 1500 euros netos mensuales como referencia que, a jornada completa (40 horas por semana, 160 al mes), son 9.4€ la hora.

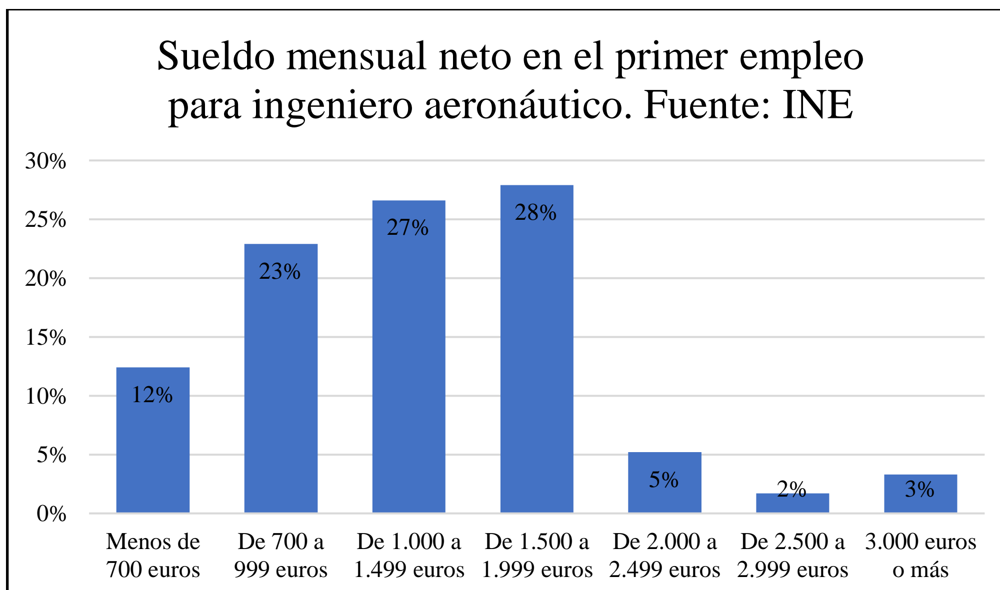


Figura 3.1: Sueldo mensual neto en el primer empleo de un ingeniero aeronáutico según INE

La Tabla 3.1 incluye un resumen de lo anterior junto a una estimación de 300 horas dedicadas en estudio de soluciones, desarrollo de las aplicaciones, códigos de almacenamiento y visualizado, fabricación del prototipo, pruebas de funcionamiento y redacción. Esto conlleva un importe total de 2820€.

Función desempeñada	Horas dedicadas	Coste/hora	Importe
Técnico en Ingeniería Aeroespacial	300	9.4€	2820€

Tabla 3.1: Coste de mano de obra

4. RESUMEN

Como resumen de los costes expuestos previamente, se adjunta la Tabla 4.1. Esta considera la compra del material opcional así como un año de ThingSpeak. El resultado es que el trabajo realizado supone un coste total de 2962.70€.

Tipo de coste	Importe
Material (indispensable y opcional)	87.70€
Software (1 año de ThingSpeak)	55€
Humano (ingeniero aeronáutico)	2820€
TOTAL	**2962.70€**

Tabla 4.1: Presupuesto completo

OBJETIVOS DE DESARROLLO SOSTENIBLE

OBJETIVOS DE DESARROLLO SOSTENIBLE

Esta sección trata la aportación que el trabajo realizado tiene en los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030. En primer lugar, introduciendo este sistema de monitorizado y almacenado de datos de vuelo, la aplicabilidad del UAV en cuestión crece enormemente, haciendo posible su uso en muchos más ámbitos, condiciones y contextos. Aunque la misión específica (hecha posible por el sistema) pueda afectar a otros ODS, en adelante se comentarán aquellos que el impacto es más directo y generalizado.

ODS 7: ENERGÍA ASEQUIBLE Y NO CONTAMINANTE

Al aumentar la aplicabilidad de los UAVs, crece la probabilidad de que estos sean la mejor opción a elegir para cumplir una misión. Este medio es de bajo consumo y emplea en la mayoría de los casos energía eléctrica, siendo capaz de sustituir a transportes tan contaminantes como aviones o helicópteros.

ODS 8: TRABAJO DECENTE Y CRECIMIENTO ECONÓMICO

La implementación de un sistema con almacenamiento remoto reduce la necesidad de desplazarse hasta la zona de vuelo del UAV, permitiendo a más personas trabajar en el monitorizado y análisis de datos de forma más segura, incluso posibilitando el teletrabajo. La siguiente ODS también guarda relación con el crecimiento económico.

ODS 9: INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURAS

Encontrar nuevas aplicaciones a tecnologías en continuo desarrollo como las telecomunicaciones LPWAN o los servidores remotos fomenta que las industrias implicadas crezcan al crearse más demanda. Además, el sistema de monitorizado y almacenamiento es muy demandante y lleva a estas tecnologías al límite de sus capacidades (velocidad, rango, consumo, coste…), provocando que mejoras aparentemente ínfimas puedan marcar la diferencia. Por tanto, dedicar recursos a la investigación e innovación supone una inversión con altas probabilidades de éxito, lo cual incentiva la competitividad entre empresas.

ODS 15: VIDA DE ECOSISTEMAS TERRESTRES

El medio rural es el ideal para el uso de UAVs por la ausencia de edificios y telecomunicaciones que causan inferencias y menos impedimentos legales para volar. La reducción de personas presentes en la zona y la evitación de alternativas como aeronaves o vehículos a motor son beneficiosas para el ecosistema, pues se minimiza el impacto tanto físico como acústico del humano.

PLIEGO DE CONDICIONES

1. NORMAS Y PROCEDIMIENTOS DE UAVS

La operación de un UAV (en esta sección se usará UAS, *unmanned aircraft system*, para clarificar que es el UAV y el equipo para controlarla de forma remota) está legislada de forma que se garantice la seguridad para todas las personas en tierra y en aire. El marco legal competente y aplicable es el desarrollado y aprobado por la Comisión de la Unión Europea en el Reglamento de Ejecución (UE) 2019/947. Puesto que la validación del prototipo consistió en realizar una prueba de vuelo, durante la misma se tuvo en cuenta este reglamento de donde destacan los siguientes artículos.

Artículo 4. Categoría «abierta» de operaciones de UAS

1. Las operaciones se clasificarán como operaciones de UAS en la categoría «abierta» únicamente cuando se cumplan los requisitos siguientes:

    a) el UAS pertenece a una de las clases establecidas en el Reglamento Delegado (UE) 2019/945, es de construcción privada o cumple las condiciones definidas en el artículo 20;

    b) la masa máxima de despegue de la aeronave no tripulada es inferior a 25 kg;

    c) el piloto a distancia garantiza que la aeronave no tripulada se mantiene a una distancia segura de las personas y que no vuela sobre concentraciones de personas;

    d) el piloto a distancia mantiene en todo momento la aeronave no tripulada dentro del alcance visual, salvo cuando vuele en modo sígueme o cuando se utilice un observador de aeronave no tripulada, tal como se especifica en la parte A del anexo;

    e) durante el vuelo, la aeronave no tripulada no se alejará más de 120 m del punto más próximo de la superficie terrestre, salvo cuando sobrevuele un obstáculo, tal como se especifica en la parte A del anexo;

    f) durante el vuelo, la aeronave no tripulada no transportará mercancías peligrosas ni dejará caer ningún material;

Artículo 7. Normas y procedimientos aplicables a la utilización de UAS

1. Las operaciones de UAS en la categoría «abierta» deberán respetar las limitaciones operacionales establecidas en la parte A del anexo.

Artículo 11. Normas para efectuar evaluación del riesgo operacional

1. En una evaluación del riesgo operacional:

    a) se describirán las características de la operación del UAS;

    b) se propondrán objetivos adecuados de seguridad operacional;

c) se determinarán los riesgos de la operación en tierra y en el aire, teniendo en cuenta las consideraciones siguientes:

   i. la medida en que la actividad podría poner en peligro a terceros o bienes en tierra;

   ii. la complejidad, el rendimiento y las características operacionales de la aeronave no tripulada utilizada;

   iii. la finalidad del vuelo, el tipo de UAS, la probabilidad de colisión con otras aeronaves y la clase de espacio aéreo utilizado;

   iv. el tipo, la escala y la complejidad de la operación o actividad del UAS, incluidos, cuando proceda, el tamaño y el tipo de tráfico gestionado por la organización o persona responsable;

   v. la medida en que las personas afectadas por los riesgos de la operación del UAS pueden evaluar y controlar tales riesgos;

d) se determinarán posibles medidas de atenuación del riesgo;

e) se determinará el nivel de solidez que deben tener las medidas de atenuación seleccionadas de tal manera que la operación pueda llevarse a cabo de forma segura.

2. La descripción de la operación del UAS deberá incluir, como mínimo, la información siguiente:

   a) la naturaleza de las actividades realizadas;

   b) el entorno operacional y la zona geográfica de la operación prevista, en particular la población sobrevolada, la orografía, los tipos de espacio aéreo, el volumen de espacio aéreo en el que se llevará a cabo la operación y el volumen de espacio aéreo previsto como tampón de seguridad necesario, así como los requisitos operacionales respecto a las zonas geográficas;

   c) la complejidad de la operación, en particular la planificación y ejecución, las competencias, la experiencia y la composición del personal y los medios técnicos necesarios que se prevén para llevar a cabo la operación;

   d) las características técnicas del UAS, incluyendo su rendimiento en vista de las condiciones de la operación prevista y, si procede, su número de registro;

   e) la competencia del personal para realizar la operación, en particular su composición, su función, sus responsabilidades, su formación y su experiencia reciente.

3. La evaluación propondrá un objetivo de seguridad, que será equivalente al nivel de seguridad de la aviación tripulada, teniendo en cuenta las características específicas de la operación del UAS.

4. La determinación de los riesgos incluirá la determinación de todos los elementos siguientes:

a) el riesgo en tierra no atenuado de la operación, teniendo en cuenta el tipo de operación y las condiciones en las que se lleva a cabo, y en particular, como mínimo, los elementos siguientes:

    i. VLOS o BVLOS;

    ii. la densidad de población de las zonas sobrevoladas;

    iii. el vuelo sobre una concentración de personas;

    iv. las características en cuanto a dimensiones de la aeronave no tripulada;

b) el riesgo aéreo no atenuado de la operación, teniendo en cuenta todos los elementos siguientes:

    i. el volumen exacto del espacio aéreo en el que se llevará a cabo la operación, ampliado en un volumen de espacio aéreo necesario para procedimientos de contingencia;

    ii. la clase del espacio aéreo;

    iii. el impacto sobre otros tipos de tráfico aéreo y la gestión del tráfico aéreo (GTA), en particular:

        - altitud de la operación;

        - espacio aéreo controlado frente a espacio aéreo no controlado;

        - entorno de un aeródromo frente a entorno distinto de un aeródromo;

        - espacio aéreo sobre un entorno urbano frente a espacio aéreo sobre un entorno rural;

        - separación del resto del tráfico.

5. Al determinar las posibles medidas de atenuación que deban aplicarse para alcanzar el nivel de seguridad propuesto se tendrán en cuenta las posibilidades siguientes:

    a) medidas de contención para las personas en tierra;

    b) limitaciones operacionales estratégicas de la operación del UAS, en particular:

        i. la restricción de los volúmenes geográficos en los que se lleva a cabo la operación;

        ii. la restricción de la duración o la programación de la franja horaria en la que se lleva a cabo la operación;

c) la atenuación estratégica mediante normas de vuelo comunes o una estructura y servicios del espacio aéreo comunes;

d) la capacidad para hacer frente a posibles condiciones operativas adversas;

e) factores de organización como los procedimientos operacionales y de mantenimiento elaborados por el operador de UAS y procedimientos de mantenimiento conformes con el manual del usuario facilitado por el fabricante;

f) el nivel de competencia y experiencia del personal responsable de la seguridad del vuelo;

g) el riesgo de error humano en la aplicación de los procedimientos operacionales;

h) las características de diseño y el rendimiento del UAS, en particular:

   i. la existencia de medios para atenuar los riesgos de colisión;

   ii. la existencia de sistemas que limiten la energía en el impacto o la frangibilidad de la aeronave no tripulada;

   iii. el diseño del UAS según normas reconocidas y el diseño a prueba de fallos.

6. Se evaluará la solidez de las medidas de atenuación propuestas para determinar si son proporcionales a los objetivos de seguridad y los riesgos de la operación prevista, en particular para asegurarse de que todas las fases de la operación sean seguras.

Artículo 16. Operaciones de UAS en el marco de clubes y asociaciones de aeromodelismo

1. A petición de un club o asociación de aeromodelismo, la autoridad competente podrá expedir una autorización de operaciones de UAS en el marco de clubes y asociaciones de aeromodelismo.

2. La autorización mencionada en el apartado 1 se expedirá de conformidad con cualquiera de las opciones siguientes:

   a) las normas nacionales pertinentes;

   b) los procedimientos, la estructura organizativa y el sistema de gestión establecidos del club o asociación de aeromodelismo, asegurándose de que:

      i. los pilotos a distancia que operen en el marco de clubes o asociaciones de aeromodelismo estén informados de las condiciones y las limitaciones definidas en la autorización expedida por la autoridad competente;

      ii. los pilotos a distancia que operen en el marco de clubes o asociaciones de aeromodelismo reciban asistencia para alcanzar la competencia mínima

necesaria para pilotar los UAS de forma segura y de conformidad con las condiciones y las limitaciones definidas en la autorización;

iii. el club o asociación de aeromodelismo adopte las medidas adecuadas cuando es alertado de que un piloto a distancia que opera en el marco de clubes o asociaciones de aeromodelismo no cumple las condiciones y las limitaciones definidas en la autorización, y, en caso necesario, informe a la autoridad competente;

iv. el club o asociación de aeromodelismo proporcione, a petición de la autoridad competente, la documentación necesaria con fines de supervisión y seguimiento.

3. En la autorización mencionada en el apartado 1 se especificarán las condiciones en las que podrán efectuarse operaciones en el marco de clubes o asociaciones de aeromodelismo, y dicha autorización se limitará al territorio del Estado miembro en el que se expida.

4. Los Estados miembros podrán permitir que los clubes y asociaciones de aeromodelismo registren a sus miembros, en su nombre, en los sistemas de registro establecidos de conformidad con el artículo 14. En caso contrario, los miembros de clubes y asociaciones de aeromodelismo se registrarán de conformidad con el artículo 14.

UAS.OPEN.060. Responsabilidades del piloto a distancia

1) Antes de iniciar una operación de UAS, el piloto a distancia:

a) tendrá las competencias apropiadas correspondientes a la subcategoría de operaciones de UAS previstas, de conformidad con las secciones UAS.OPEN.020, UAS.OPEN.030 o UAS.OPEN.040, para desempeñar sus tareas y portará una prueba de dichas competencias durante la utilización del UAS, salvo cuando utilice una aeronave no tripulada de conformidad con la sección UAS.OPEN.020, punto 5, letras a), b) o c);

b) obtendrá información actualizada pertinente para la operación de UAS prevista, acerca de toda zona geográfica, publicada por el Estado miembro de la operación de conformidad con el artículo 15;

c) observará el entorno operativo, comprobará la presencia de obstáculos y, salvo que opere en la categoría A1 con una aeronave no tripulada a la que se hace referencia en la sección UAS.OPEN.020, apartado 5, letras a), b) o c), comprobará si está presente alguna persona no participante;

d) se asegurará de que el UAS está en condiciones de realizar el vuelo previsto con seguridad y, si procede, comprobará el correcto funcionamiento de la identificación directa a distancia;

e) si el UAS lleva una carga útil adicional, verificará que su masa no supera la MTOM determinada por el fabricante o el límite de MTOM de su clase.

2) Durante el vuelo, el piloto a distancia:

   a) no desempeñará sus tareas bajo los efectos de sustancias psicoactivas o alcohol o si no está en condiciones de desempeñarlas debido a lesiones, cansancio, medicación, enfermedad u otras causas;

   b) mantendrá la aeronave no tripulada en modo VLOS y mantendrá un riguroso control visual del espacio aéreo que rodea a la aeronave no tripulada con el fin de evitar cualquier riesgo de colisión con una aeronave tripulada; el piloto a distancia interrumpirá el vuelo si la operación supone un riesgo para otras aeronaves, personas, animales, el medio ambiente o bienes;

   c) respetará las limitaciones operacionales en las zonas geográficas determinadas de conformidad con el artículo 15;

   d) será capaz de mantener el control de la aeronave no tripulada, salvo en caso de pérdida de conexión o si utiliza una aeronave no tripulada de vuelo libre;

   e) utilizará el UAS de acuerdo con el manual del usuario facilitado por el fabricante, incluida toda limitación aplicable;

   f) respetará los procedimientos del operador, si están disponibles.

3) Durante el vuelo, los pilotos a distancia y los operadores de UAS no los harán volar cerca o dentro de zonas en las que se estén llevando a cabo operaciones de emergencia, salvo que los servicios de emergencia responsables les hayan dado permiso para hacerlo.

4) A los fines del punto 2, letra b), los pilotos a distancia podrán estar asistidos por un observador de la aeronave no tripulada situado junto a ellos que haga una observación visual de la aeronave sin la ayuda de instrumentos y les ayude a efectuar el vuelo de forma segura. Se establecerá una comunicación clara y eficaz entre el piloto a distancia y el observador de la aeronave no tripulada.

2. DERECHOS Y OBLIGACIONES DEL TRABAJADOR

A continuación, con el fin de reflejar el marco legal que ampara a los trabajadores en términos de seguridad e higiene dentro del territorio español, se incluye en este documento la presente normativa. Esta se recoge en el BOE número 64 del 16 de marzo de 1971, de donde se han extraído aquellos artículos que atañen al proyecto.

Su relevancia es consecuencia de haber llevado a cabo el desarrollo del sistema de monitorizado y almacenamiento en el Departamento de Electrónica de la ETSID, ubicado en las instalaciones de la UPV, mientras que la prueba de vuelo se realizó en el Club de Aeromodelismo de L'Abella. Por tanto, con el fin de garantizar el bienestar y la salud, se han tenido en cuenta las siguientes disposiciones.

Artículo 11. Obligaciones y derechos de los trabajadores

Incumbe a los trabajadores la obligación de cooperar en la prevención de riesgos profesionales en la Empresa y el mantenimiento de la máxima higiene en la misma, a cuyos fines deberán cumplir fielmente los preceptos de esta Ordenanza y sus instrucciones complementarias, así como las órdenes e instrucciones que a tales efectos les sean dados por sus superiores.

Los trabajadores, expresamente, están obligados a:

A) Recibir las enseñanzas sobre Seguridad e Higiene y sobre salvamento y socorrismo en los centros de trabajo que les sean facilitadas por la Empresa o en las Instituciones del Plan Nacional.

B) Usar correctamente los medios de protección personal y cuidar de su perfecto estado y conservación.

C) Dar cuenta inmediata a sus superiores de las averías y deficiencias que puedan ocasionar peligros en cualquier centro o puesto de trabajo.

D) Cuidar y mantener su higiene personal, en evitación de enfermedades contagiosas o de molestias a sus compañeros de trabajo.

E) Someterse a los reconocimientos médicos preceptivos y a las vacunaciones o inmunizaciones ordenadas por las Autoridades Sanitarias competentes o por el Servicio Médico de Empresa.

F) No introducir bebidas u otras sustancias no autorizadas en los centros de trabajo, ni presentarse o permanecer en los mismos en estado de embriaguez o de cualquier otro género de intoxicación.

G) Cooperar en la extinción de siniestros y en el salvamento de las víctimas de accidentes de trabajo en las condiciones que, en cada caso, fueren racionalmente exigibles.

Artículo 13. Seguridad estructural.

1. Todos los edificios, permanentes o provisionales, serán de construcción segura y firme para evitar riesgos de desplome y los derivados de los agentes atmosféricos.

2. Los cimientos, pisos y demás elementos de los edificios ofrecerán resistencia suficiente para sostener y suspender con seguridad las cargas para los que han sido calculados.

3. Se indicará por medio de rótulos o inscripciones las cargas que los locales puedan soportar o suspender, quedando prohibido sobrecargar los pisos y plantas de los edificios.

Artículo 14. Superficie y cubicación.

1. Los locales de trabajo reunirán las siguientes condiciones mínimas:
   a) Tres metros de altura desde el piso al techo.

   b) Dos metros cuadrados de superficie por cada trabajador.

   c) Diez metros cúbicos por cada trabajador.

2. No obstante, en los establecimientos comerciales, de servicios y locales destinados a oficinas y despachos la altura a que se refiere el apartado a) del número anterior podrá quedar reducida hasta 2,50 metros, pero respetando la cubicación por trabajador que se establece en el apartado c), y siempre que se renueve el aire suficientemente.

3. Para el cálculo de la superficie y volumen no se tendrán en cuenta los espacios ocupados por máquinas, aparatos, instalaciones y materiales.

Artículo 15. Suelos, techos y paredes.

1. El pavimento constituirá un conjunto homogéneo, llano y liso sin soluciones de continuidad; será de material consistente, no resbaladizo o susceptible de serlo con el uso y de fácil limpieza. Estará al mismo nivel, y de no ser así, se salvarán las diferencias de altura por rampas de pendiente no superior al 10 por 100.

2. Las paredes serán lisas, guarnecidas o pintadas en tonos claros y susceptibles de ser lavadas o blanqueadas.

3. Los techos deberán reunir las condiciones suficientes para resguardar a los trabajadores de las inclemencias del tiempo. Si han de soportar o suspender cargas deberán reunir las condiciones que se establecen para los pisos en el artículo 13.

Artículo 25. Iluminación. Disposiciones generales.

1. Todos los lugares de trabajo o tránsito tendrán iluminación natural, artificial o mixta apropiada a las operaciones que se ejecuten.

2. Siempre que sea posible se empleará la iluminación natural.

3. Se intensificará la iluminación de máquinas peligrosas, lugares de tránsito con riesgo de caídas, escaleras y salidas de urgencia.

4. Se deberá graduar la luz en los lugares de acceso a zonas de distinta intensidad luminosa.

Artículo 30. Ventilación, temperatura y humedad.

1. En los locales de trabajo y sus anexos se mantendrán, por medios naturales o artificiales, condiciones atmosféricas adecuadas, evitando el aire viciado, exceso de calor y frío, humedad o sequía y los olores desagradables.

2. Las emanaciones de polvo, fibras, humos, gases, vapores o neblinas, desprendidos en locales de trabajo, serán extraídos, en lo posible, en su lugar de origen, evitando su difusión por la atmósfera.

3. En ningún caso el anhídrido carbónico o ambiental podrá sobrepasar la proporción de 50/10.000, y el monóxido de carbono, la de 1/10.000. Se prohíbe emplear braseros, salamandras, sistemas de calor por fuego libre, salvo a la intemperie y siempre que no impliquen riesgo de incendio o explosión.

4. En los locales de trabajo cerrados, el suministro de aire fresco y limpio por hora y trabajador será, al menos, de 30 a 50 metros cúbicos, salvo que se efectúe una renovación total del aire varias veces por hora, no inferior a seis veces para trabajos sedentarios ni diez veces para trabajos que exijan esfuerzo físico superior al normal.

5. La circulación de aire en locales cerrados se acondicionará de modo que los trabajadores no estén expuestos a corrientes molestas y que la velocidad del aire no exceda de 15 metros por minuto con temperatura normal, ni de 45 metros por minuto en ambientes muy calurosos.

6. En los centros de trabajo expuestos a altas y bajas temperaturas serán evitadas las variaciones bruscas por el medio más eficaz. Cuando la temperatura sea extremadamente distinta entre los lugares de trabajo, deberán existir locales de paso para que los operarios se adapten gradualmente a unas y otras.

7. Se fijan como límites normales de temperatura y humedad en locales y para los distintos trabajos, siempre que el procedimiento de fabricación lo permita, los siguientes:
   - Para trabajos sedentarios: De 17 a 22 grados centígrados.
   - Para trabajos ordinarios: De 15 a 18 grados centígrados.
   - Para trabajos que exijan acusado esfuerzo muscular: De 12 a 15 grados centígrados.

   La humedad relativa de la atmósfera oscilará del 40 al 60 por 100, salvo en instalaciones en que haya peligro por generarse electricidad estática, que deberá estar por encima del 50 por 100.

8. Las instalaciones generadoras de calor o frío se situarán con la debida separación de los locales de trabajo para evitar en ellos peligros de incendio o explosión, el desprendimiento de gases nocivos, irradiaciones directas de calor o frío y las corrientes de aire perjudiciales al trabajador.

9. Todos los trabajadores estarán debidamente protegidos contra las irradiaciones directas y excesivas de calor.

10. En los trabajos que hayan de realizarse en locales cerrados con extremado frío o calor se limitará la permanencia de los operarios estableciendo, en su caso, los turnos adecuados.

Artículo 51. Protección contra contactos en las instalaciones y equipos eléctricos.

1. En las instalaciones y equipos eléctricos, para la protección de las personas contra los contactos con partes habitualmente en tensión se adoptarán algunas de las siguientes prevenciones:

   a) Se alejarán las partes activas de la instalación a distancia suficiente del lugar donde las personas habitualmente se encuentran o circulen, para evitar un contacto fortuito o por la manipulación de objetos conductores, cuando éstos puedan ser utilizados cerca de la instalación.

   b) Se recubrirán las partes activas con aislamiento apropiado, que conserven sus propiedades indefinidamente y que limiten la corriente de contacto a un valor inocuo.

   c) Se interpondrán obstáculos que impidan todo contacto accidental con las partes activas de la instalación. Los obstáculos de protección deben estar fijados en forma segura y resistir a los esfuerzos mecánicos usuales.

2. Para la protección contra los riesgos de contacto con las masas de las instalaciones que puedan quedar accidentalmente con tensión, se adoptarán, en corriente alterna, uno o varios de los siguientes dispositivos de seguridad:

   a) Puesta a tierra de las masas. Las masas deben estar unidas eléctricamente a una toma de tierra o a un conjunto de tomas de tierra interconectadas, que tengan una resistencia apropiada. Las instalaciones, tanto con neutro aislado de tierra como con neutro unido a tierra, deben estar permanentemente controladas por un dispositivo que indique automáticamente la existencia de cualquier defecto de aislamiento, o que separe automáticamente la instalación o parte de la misma, en la que esté el defecto de la fuente de menta que la alimenta.

   b) De corte automático o de aviso, sensibles a la corriente de defecto (interruptores diferenciales), o a la tensión de defecto (relés de tierra).

   c) Unión equipotencial o por superficie aislada de tierra o de las masas (conexiones equipotenciales).

   d) Separación de los circuitos de utilización de las fuentes de energía, por medio de transformadores o grupos convertidores, manteniendo aislados de tierra todos los conductores del circuito de utilización, incluido el neutro.

   e) Por doble aislamiento de los equipos y máquinas eléctricas.

3. En corriente continua, se adoptarán sistemas de protección adecuados para cada caso, similares a los referidos para la alterna.

Artículo 82. Medios de prevención y extinción.

En los centros de trabajo que ofrezcan peligro de incendios, con o sin explosión, se adoptarán las prevenciones que se indican a continuación, combinando su empleo, en su caso, con la protección general más próxima que puedan prestar los servicios públicos contra incendios.

1. Uso del agua:
   - Donde existan conducciones de agua a presión, se instalarán suficientes tomas o bocas de agua a distancia conveniente entre sí y cercanas a los puestos fijos de trabajo y lugares de paso del personal, colocando junto a tales tomas las correspondientes mangueras, que tendrán la sección y resistencia adecuada.
   - Cuando se carezca normalmente de agua a presión o ésta sea insuficiente, se instalarán depósitos con agua suficiente para combatir los posibles incendios.
   - En los incendios provocados por líquidos, grasas o pinturas inflamables o polvos orgánicos, sólo deberá emplearse agua muy pulverizada.
   - No se empleará agua para extinguir fuegos en polvos de aluminio o magnesio o en presencia de carburo de calcio u otras sustancias que al contacto con el agua produzcan explosiones, gases inflamables o nocivos.
   - En incendios que afecten a instalaciones eléctricas con tensión, se prohibirá el empleo de extintores de espuma química, soda ácida o agua.

2. Extintores portátiles:
   - En proximidad a los puestos de trabajo con mayor riesgo de incendio, colocados en sitio visible y accesible fácilmente, se dispondrán extintores portátiles o móviles sobre ruedas, de espuma física o química, mezcla de ambas o polvos secos, anhídrido carbónico o agua, según convenga a la causa determinante del fuego a extinguir.
   - Cuando se empleen distintos tipos de extintores serán rotulados con carteles indicadores del lugar y clase de incendio en que deban emplearse.
   - Se instruirá al personal, cuando sea necesario, del peligro que presenta el empleo de tetracloruro de carbono, y cloruro de metilo en atmósferas cerradas y de las reacciones químicas peligrosas que puedan producirse en los locales de trabajo entre los líquidos extintores y las materias sobre las que puedan proyectarse.
   - Los extintores serán revisados periódicamente y cargados según las normas de las casas constructoras inmediatamente después de usarlos.

3. Empleo de arenas finas: para extinguir los fuegos que se produzcan en polvos o virutas de magnesio y aluminio, se dispondrá en lugares próximos a los de trabajo, de cajones o retenes suficientes de arena fina seca, de polvo de piedra u otras materias inertes semejantes.

4. Detectores automáticos: en las industrias o lugares de trabajo de gran peligrosidad en que el riesgo de incendio afecte a grupos de trabajadores, la Delegación Provincial de Trabajo podrá imponer la obligación de instalar aparatos de fuego o detectores de incendios, del tipo más adecuado: aerotérmico, termoeléctrico, químico, fotoeléctrico, radiactivo, por ultrasonidos, etc.

5. Prohibiciones personales:
   - En las dependencias con alto riesgo de incendio, queda terminantemente prohibido fumar o introducir cerillas, mecheros o útiles de ignición. Está prohibición se indicará con carteles visibles a la entrada y en los espacios libres de las paredes de tales dependencias.

- Se prohíbe igualmente al personal introducir o emplear útiles de trabajo, no autorizados por la Empresa, que puedan ocasionar chispas por contacto o proximidad a sustancias inflamables.
- Es obligatorio el uso de guantes, manoplas, mandiles o trajes ignífugos, y de calzado especial contra incendios que las Empresas faciliten a los trabajadores para uso individual.

6. Equipos contra incendios:
   - En las industrias o centros de trabajo con grave riesgo de incendio se instruirá y entrenará especialmente al personal integrado en el equipo o brigada contra incendios, sobre el manejo y conservación de las instalaciones y material extintor, señales de alarma, evacuación de los trabajadores y socorro inmediato a los accidentados.
   - El personal de los equipos contra incendios dispondrá de cascos, trajes aislantes, botas y guantes de amianto y cinturones de seguridad; asimismo dispondrá, si fuera preciso, para evitar específicas intoxicaciones o sofocación, de máscaras y equipos de respiración autónoma.
   - El material asignado a los equipos de extinción de incendios: escalas, cubiertas de lona o tejidos ignífugos, hachas, picos, palas, etc., no podrá ser usado para otros fines y su emplazamiento será conocido por las personas que deban emplearlo.
   - La empresa designará el Jefe de equipo o brigada contra incendios, que cumplirá estrictamente las instrucciones técnicas dictadas por el Comité de Seguridad para la extinción del fuego y las del Servicio Médico de Empresa para el socorro de los accidentados.

7. Alarmas y simulacros de incendios: para comprobar el buen funcionamiento de los sistemas de prevención, el entrenamiento de los equipos contra incendios y que los trabajadores, en general, conocen y participan con aquéllos, se efectuarán periódicamente alarmas y simulacros de incendios, por orden de la Empresa y bajo la dirección del Jefe del equipo o brigada contra incendios, que sólo advertirá de los mismos a las personas que deban ser informadas en evitación de daños o riesgos innecesarios.

ANEXOS

A. CÓDIGO COMPLETO

A.1 Microcontrolador emisor

```
/*
"LoRa_SEND"
Por Carlos Cantarino Barberá, 2023

El siguiente código para ESP32 recoge datos de tres sensores diferentes
(BMP280, BNO055 y BN-880) y los envía a otro microcontrolador por LoRa.

Se recomienda no modificar nada sin pleno conocimiento.
*/

////////////////////////////////////////////////////////////////////////////

//-----------//
// PRE SETUP //
//-----------//


// Variables controladas por app
  // Calibración
  byte cal = 0;

  byte sIMU = 0;
  float p0 = 0;
  float r0 = 0;

  byte sALT = 0;
  float h0 = 0;
  float h20 = 0;

  byte sHDN = 0;
  float H0 = 0;

  // Precisión
  byte prec = 1;

  // Presión de referencia
  float PresRef = 1013.25;


// Variables generales

  unsigned int ReadFreq_MS = 10; // Tiempo en ¡MILISEGUNDOS! entre toma de
datos (ha de ser pequeño para mayor precisión)
  unsigned int SendFreq_MS = 100;  // Tiempo en ¡MILISEGUNDOS! entre envío de
datos por LoRa y Serial.

  unsigned long long t0_read;
  unsigned long long t0_send;


// Bluetooth
  #include <BluetoothSerial.h>
```

```
  BluetoothSerial SerialBT;


// LoRa
  #include <SPI.h>
  #include <LoRa.h>

  #define SCK 5    // BLANCO
  #define MISO 19 // VERDE
  #define MOSI 2 //27 // MORADO
  #define SS 18    // MARRON
  #define RST  0//14  // GRIS
  #define DIO0 15 //26 // AMARILLO

  #define BAND 433E6 // Transmisión a 433 MHz (LoRa SX1278)

  unsigned long readingID = 0; // Contador de número de lecturas realizadas


// BMP280
  #include <Adafruit_BMP280.h>
  Adafruit_BMP280 bmp;


// BNO055
  #include <Adafruit_BNO055.h>
  Adafruit_BNO055 IMU = Adafruit_BNO055(55, 0x28); // El dispositivo BNO055
corresponde con el puerto 0x28


// GPS (BN-880)
  #include <TinyGPSPlus.h>
  TinyGPSPlus gps;


// Definir variables de datos
  float V = 0;       // Airspeed from GPS
  float V2 = 0;      // Airspeed from IMU
  float h = 0;       // Height from GPS
  float h2 = 0;      // Height from pressure
  float P = 0;       // Pressure
  float T = 0;       // Temperature
  float H = 0;       // Heading
  float p = 0;       // Pitch angle
  float r = 0;       // Roll angle
  float X = 0;       // Position X, longitude
  float Y = 0;       // Position Y, latitude


//////////////////////////////////////////////////////////////////////////////

//-------//
// SETUP //
//-------//

void setup() {
  startSerial();
  startBluetooth();
  startBMP();
```

```
  startIMU();
  startGPS();
  startLoRA();

  delay(100);
  t0_read = micros();
  t0_send = micros();
}

  void startSerial(){
    Serial.begin(115200);
    while (!Serial) delay(10);  // Espera a que se abra el puerto
  }

  void startBluetooth(){
    SerialBT.begin("ESP32 Bluetooth"); // Nombre del dispositivo
    delay(100);  // Espera a que se inicie el bluetooth
    clearBluetoothBuffer();
  }

  void startBMP() {
    if (!bmp.begin(0x76)) { // El dispositivo BMP280 corresponde con el
puerto 0x76
      Serial.println("No se encontró el sensor BMP280.");
      while (1);
    }
  }

  void startIMU() {
    if (!IMU.begin()) {
      Serial.println("No se encontró el sensor BNO055.");
      while (1);
    }
  }

  void startGPS(){ // Usamos comunicación UART con Serial2, que son los pines
16 (RX) y 17 (TX) en el ESP32
    Serial2.begin(9600);
    while (!Serial2) delay(10);  // Esperar a que Serial se abra
    Serial.println("¡Sensores correctamente inicializados!");
  }

  void startLoRA(){
    // Establecer pines
    SPI.begin(SCK, MISO, MOSI, SS);
    LoRa.setPins(SS, RST, DIO0);

    // Establecer propiedades de la comunicación
    LoRa.setSpreadingFactor(7);
    LoRa.setSignalBandwidth(125E3);
    LoRa.setCodingRate4(5);
    LoRa.enableCrc();

    Serial.println();
    Serial.print("Conectando...");
    while (!LoRa.begin(BAND)) {
      Serial.print(".");
      delay(500);
    }
```

```
    Serial.print(" LoRa OK!");
    Serial.println();
  }


/////////////////////////////////////////////////////////////////////////////

//------//
// LOOP //
//------//

void loop() {
  readBT();

  if ((micros() - t0_read) > (ReadFreq_MS * 1000)){  // Leer datos
    t0_read = micros();
    if (cal){
      calibrar();
    }
    readIMU();
    readGPS();
    readBMP();
  }

  if ((micros() - t0_send) > (SendFreq_MS * 1000)){  // enviar datos
    t0_send = micros();
    sendData();
  }
}


  void readBT(){

    if (SerialBT.available()){

      byte i = SerialBT.read();
      delay(1); // Se añaden delays para evitar interferencias entre los
datos recibidos

      if (i == 1){
        sIMU = SerialBT.read(); // Indicador de calibración de alabeo y
cabeceo
        delay(1);

        sALT = SerialBT.read(); // Indicador de calibración de altitud, las
lecturas serán por tanto de altura
        delay(1);

        sHDN = SerialBT.read(); // Indicador de calibración de rumbo o
heading
        delay(1);

        prec = SerialBT.read(); // Precisión
        delay(1);

        // Presión de referencia (3 bytes)
        byte byte1 = SerialBT.read(); // Parte alta
        delay(1);
        byte byte2 = SerialBT.read(); // Parte media
```

```
      delay(1);
      byte byte3 = SerialBT.read(); // Parte baja

      PresRef = ((byte1 << 16) | (byte2 << 8) | byte3 ) / 100.0f;

      // Indicador de calibración aplicable disponible
      cal = 1;
    }

    if (i == 3){
      SerialBT.write(1);
      SerialBT.write(1);
    }

    else {clearBluetoothBuffer();}
  }
}

void clearBluetoothBuffer() {
  while (SerialBT.available()) {
    SerialBT.read(); // Leer y descartar byte
  }
}


void calibrar(){
  cal = 0;

  // Calibración
  if (sIMU){
    sIMU = 0;
    p0 = p;
    r0 = r;
  }

  if (sHDN){
    sHDN = 0;
    H0 = H;
  }

  if (sALT){
    sALT = 0;
    h0 = h;
    h20 = h2;
  }
}


void readIMU(){
  sensors_event_t orientationData , linearAccelData;
  IMU.getEvent(&orientationData, Adafruit_BNO055::VECTOR_EULER);
  IMU.getEvent(&linearAccelData, Adafruit_BNO055::VECTOR_LINEARACCEL);

  // Componente X de la velocidad (V = accel * t)
  V2 += linearAccelData.acceleration.x * ReadFreq_MS * 3.6f / 1000;

  // Orientación
  H = orientationData.orientation.x - H0;  // Heading o rumbo
```

```
    if (H<0) { H += 360.0f;} // Convertimos valores negativos a positivos (H
entre 0º y 360º)
    p = orientationData.orientation.y - p0;  // Pitch o cabeceo
    r = -orientationData.orientation.z - r0; // Roll o alabeo
  }


  void readGPS(){
    while (Serial2.available() > 0) {
      if (gps.encode(Serial2.read())) {
        if (gps.location.isValid()) {
          Y = gps.location.lat();         // Latitud [º]
          X = gps.location.lng();         // Longitud [º]
          h = gps.altitude.meters() - h0; // Altitud [m]
          V = gps.speed.kmph();           // Velocidad [km/h]
        }
      }
    }
  }


  void readBMP(){
    T = bmp.readTemperature();            // Temperatura [ºC]
    P = bmp.readPressure() / 100.0f;      // Presión [hPa]
    h2 = bmp.readAltitude(PresRef) - h20; // Altitud [m]
  }


  void sendData(){
    // LoRa
    String DataString = "";
    // Combinar todos los valores en una sola variable de texto para enviar
    //           Velocidad          Velocidad2          Altitud            Al
titud2            Presión            Temperatura         Rumbo              Ca
beceo            Alabeo             Latitud          Longitud
    DataString =
String(V,prec)+"&"+String(V2,prec)+"&"+String(h,prec)+"&"+String(h2,prec)+"&"
+String(P,prec)+"&"+String(T,prec)+"&"+String(H,prec)+"&"+String(p,prec)+"&"+
String(r,prec)+"&"+String(Y,6)+"&"+String(X,6);

    LoRa.beginPacket();
    LoRa.print(DataString); // Enviar datos por LoRa
    LoRa.endPacket();

    // Enviar por UART0
    readingID++;
    if (Serial){
      Serial.print("Sending packet #");
      Serial.print(readingID);Serial.print(": ");
      Serial.println(DataString);
      Serial.println();
    }
  }
```

A.2 Microcontrolador receptor

```
/*
"LoRa_RECEIVE"
Por Carlos Cantarino Barberá, 2023

El siguiente código para ESP32 recibe datos del dispositivo emisor y los
envía por Bluetooth (app para móvil), Serial (MATLAB app) y WiFi
(ThingSpeak).

Se recomienda no modificar nada sin pleno conocimiento.

*/

/////////////////////////////////////////////////////////////////////////////

//-----------//
// PRE SETUP //
//-----------//


// Variables controladas por App
                //
 vel.  alt.       vel.  alt.       vel.  alt.
  byte DMS = 1; // Data Main Sensor: velocidad y altitud basada en 1) GPS; 2)
GPS y BMP280; 3) IMU y BMP280; 4) IMU y GPS.

  byte TS_ON = 1; // ThingSpeak ON o OFF (controla el envío de datos a
ThingSpeak)

  byte SaveStart = 0; // Marcador de inicio de guardado
  byte SaveStop = 0;  // Marcador de fin de guardado

  unsigned int ThingSpeakFreq_S = 15; // Cada cuántos SEGUNDOS enviar datos
por ThingSpeak

  unsigned int AppFreq_MS = 1000; // Cada cuántos MILISEGUNDOS enviar datos
por Bluetooth y UART


// Variables generales

  byte RSSI_WiFi = 0;
  byte RSSI_LoRa = 0;
  byte LoRaOK = 0;

  unsigned int TScorr = 250000; // Corrección de envío de datos a ThingSpeak
en microsegundos (0.25 segundos por defecto)

  unsigned long long t_App;
  unsigned long long t_TS;

  String Data = "";


// LoRa
  #include <SPI.h>
  #include <LoRa.h>
```

```
  #define SCK 5   // BLANCO
  #define MISO 19 // VERDE
  #define MOSI 27 // MORADO
  #define NSS 18  // AZUL/MARRON
  #define RST 14  // GRIS
  #define DIO0 26 // AMARILLO

  #define BAND 433E6 // Transmisión a 433 MHz (LoRa SX1278)


// Bluetooth
  #include <BluetoothSerial.h>
  BluetoothSerial SerialBT;


// WiFi
  #include <WiFi.h>
  #include <WiFiClient.h>
  WiFiClient client;

  const char* ssid = "RedCCB";
  const char* password = "gity2716";


// ThingSpeak
  #include <ThingSpeak.h>

  const char* server = "api.ThingSpeak.com";
  const unsigned long channelID = 2149521;
  const char* writeAPIKey = "PC9XIEVM2EBD2Y3H";


/////////////////////////////////////////////////////////////////////////////

//-------//
// SETUP //
//-------//

void setup(){
  startSerial();
  startBluetooth();
  startLoRA();
  startWiFi();
  startThingSpeak();

  delay(100);
  t_App = micros();
  t_TS = micros();

  // Tiempos de espera en microsegundos:
  AppFreq_MS = AppFreq_MS * 1000;
  ThingSpeakFreq_S = ThingSpeakFreq_S * 1000000 + TScorr;
}

  void startSerial(){
    Serial.begin(115200);
    while (!Serial) delay(10);  // Espera a que se abra el puerto
  }
```

```
void startBluetooth(){
  SerialBT.begin(F("ESP32 Receiver")); // Nombre del dispositivo
  clearBluetoothBuffer();
}

void startLoRA(){
  //SPI LoRa pins
  SPI.begin(SCK, MISO, MOSI, NSS);
  //setup LoRa transceiver module
  LoRa.setPins(NSS, RST, DIO0);

  LoRa.setSpreadingFactor(7);
  LoRa.setSignalBandwidth(125E3);
  LoRa.setCodingRate4(5);
  LoRa.enableCrc();

  Serial.print(F("Inicializando LoRa..."));
  byte conectar = 0;
  while ( (!LoRa.begin(BAND)) || (conectar > 20) ){
    Serial.print(F("."));
    delay(1000);
    conectar++;
  }

  LoRaOK = (conectar <= 20);
  if (LoRaOK){
    Serial.println(F(" ¡Inicialización de LoRa correcta!"));
  }
  else{
    Serial.println(F(" Error al inicializar LoRa."));
  }
}

void startWiFi(){
  WiFi.begin(ssid, password);

  Serial.print(F("Conectando a red WiFi..."));
  byte conectar = 0;
  while ( (WiFi.status()!= WL_CONNECTED) || (conectar > 30) ) {
    Serial.print(F("."));
    delay(500);
    conectar++;
  }

  if (conectar <= 30){
    Serial.println(F(" ¡Conectado a red WiFi!"));
  }
  else{
    Serial.println(F(" Conexión a red WiFi fallida."));
  }
}

void startThingSpeak(){
  ThingSpeak.begin(client);
}

//////////////////////////////////////////////////////////////////////////////
```

```
//------//
// LOOP //
//------//

void loop() {

  readData();

  readSerial();

  readBT();

  if (TS_ON){

    if ((micros() - t_App) >= AppFreq_MS){
      t_App = micros();
      AppData();
    }

    if  ((micros() - t_TS) >= ThingSpeakFreq_S){
      t_TS = micros();
      ThingSpeakUpload();
    }

  }
}


  void readData(){
    if (LoRa.parsePacket()) { // Comprobar si se reciben datos por LoRa
      Data = "";
      while (LoRa.available()) { // Leer paquete
        Data += (char)LoRa.read(); // Leer carácter
      }

      if (!TS_ON){AppData();} // Ejecutar AppData con ThingSpeak desactivado
    }
  }

  void readSerial(){
    if (Serial.available()) { // Comprobar si se reciben datos por Serial

      byte i = Serial.read(); // Leemos el valor recibido

      while (Serial.available()) {Serial.read();} // Descartamos todos los
bytes de más

      // Si recibimos un '0', desactivamos el envío de datos a ThingSpeak
      if (i == 0x30) {
        TS_ON = 0;
        Serial.println(F("ThingSpeak: OFF"));
      }
      // Si recibimos un '1', activamos el envío de datos a ThingSpeak
      else if (i == 0x31) {
        TS_ON = 1;
        Serial.println(F("ThingSpeak: ON"));
      }
```

```
      // Si recibimos un '7', enviamos a ThingSpeak un marcador de inicio de
guardado
      else if (i == 0x37) {
        SaveStart = 1;
        Serial.println(F("Inicio de guardado enviado"));
      }
      // Si recibimos un '8', enviamos a ThingSpeak un marcador de fin de
guardado
      else if (i == 0x38) {
        SaveStop  = 1;
        Serial.println(F("Fin de guardado enviado"));
      }

    }
  }

  void readBT(){
    if (SerialBT.available()) { // Comprobar si se reciben datos por
Bluetooth
      byte i = SerialBT.read();
      delay(1); // Se añaden delays para evitar interferencias entre los
datos recibidos

      if (i == 2){
        DMS = SerialBT.read();
        delay(1);

        TS_ON = SerialBT.read();
        delay(1);

        ThingSpeakFreq_S = SerialBT.read() * 1000000 + TScorr;
        delay(1);

        byte byte1 = SerialBT.read();
        delay(1);
        byte byte2 = SerialBT.read();

        AppFreq_MS = ((byte1 << 8) | byte2) * 1000;
      }

      else if (i == 3) {
        if (WiFi.status() == WL_CONNECTED) {RSSI_WiFi = abs(WiFi.RSSI());}
        else                               {RSSI_WiFi = 0;}

        if (LoRaOK)                        {RSSI_LoRa =
abs(LoRa.packetRssi());}
        else                               {RSSI_LoRa = 0;}

        SerialBT.write(RSSI_WiFi);
        SerialBT.write(RSSI_LoRa);
      }

      else if (i == 7) {
        SaveStart = SerialBT.read();
      }

      else if (i == 8) {
        SaveStop  = SerialBT.read();
      }
```

```
      else {clearBluetoothBuffer();}
    }
  }

  void clearBluetoothBuffer() {
    while (SerialBT.available()) {
      SerialBT.read(); // Leer y descartar byte
    }
  }


  void AppData(){
    SerialBT.println(Data);
    Serial.println(Data);
  }


  void ThingSpeakUpload(){
    byte corr = 0;
    if (Data.indexOf("&") == -1) { // Comprobar que haya algún delimitador
"&"
      corr= 1;
      readData();
      ThingSpeakUpload();
      Serial.println(F("Datos corruptos no enviados a ThingSpeak."));
    }

    else if ((SaveStart)||(SaveStop)){
      int save = 0;
      if (SaveStart){save = -100;}
      else if (SaveStop){save = -200;}

      ThingSpeak.setField(1, save);
      for (byte i = 2; i <= 8; i++) {
        ThingSpeak.setField(i, 0);
      }

      int httpCode = ThingSpeak.writeFields(channelID, writeAPIKey);
      if (httpCode == 200) {
        if      (SaveStart) {SaveStart = 0; Serial.println(F("Guardado
comenzado en ThingSpeak."));}
        else if (SaveStop)  {SaveStop  = 0; Serial.println(F("Guardado
terminado en ThingSpeak."));}
      }
    }

    else{
      char *var; // Trozos de texto a extraer de “Data”
      const char *delimiter = "&"; // Delimitador
      var = strtok(const_cast<char*>(Data.c_str()), delimiter);

      byte var_i = 1; // Indicador de la variable actual
      byte field = 1; // Indicador del campo del canal

      while (var != NULL) { // Separar el dato recibido en variables
separadas para enviar a ThingSpeak

        // Comprobar que las variables sean números válidos
```

```
        float varcheck = atof(var);
        if (isnan(varcheck)) {corr= 1; break;}

        // Escribir dato en el campo correspondiente
        if (var_i == 5) {} // Saltar dato de presión
        else if ( (DMS == 1) && ((var_i == 2)||(var_i == 4)) ){} // Saltar
datos de
        else if ( (DMS == 2) && ((var_i == 2)||(var_i == 3)) ){} // velocidad
y altitud
        else if ( (DMS == 3) && ((var_i == 1)||(var_i == 3)) ){} // según el
Data Main
        else if ( (DMS == 4) && ((var_i == 1)||(var_i == 4)) ){} // Sensor
deseado
        else {ThingSpeak.setField(field, var); field++;} // Establecer campo
y variable; después siguiente campo

        var=strtok(NULL, delimiter); // Siguiente token
        var_i++; // Siguiente ID de variable
      }

      if (!corr){
        int httpCode = ThingSpeak.writeFields(channelID, writeAPIKey);
          if (httpCode == 200) {
            Serial.println(F("¡Datos enviados correctamente a ThingSpeak!"));
          }
          else {
            Serial.println(F("No se pudieron enviar datos a ThingSpeak."));
          }
      }
      else {
        Serial.println(F("Datos corruptos no enviados a ThingSpeak."));
        readData();
        ThingSpeakUpload();
      }

    }

  }
```

A.3 Aplicación de MATLAB

```
methods (Access = private)

    %%% RECEPCION por SERIAL
    function Funcion_Recepcion(app,~,~)
        global Puerto lamp x DatosGuardados TimeStamp ControlTab InstrumentTab
        global vel vel2 alt alt2 pres temp hdng pitch roll lat lon cont

        if (app.ControlButton.Value == 1)

            datos_str = fgetl(Puerto);

            if isstrprop(datos_str(1),'digit') && count(datos_str,'&') == 10

                cont=cont+1; % Se incrementa en contador de muestras

                datos = str2double(split(datos_str,"&")); % Separar variables

                x(cont)=cont; % Se escribe en el vector de la abcisas el contador de muestras

                % Variables
                vel(cont)      = datos(1);
                vel2(cont)     = datos(2);
                alt(cont)      = datos(3);
                alt2(cont)     = datos(4);
                pres(cont)     = datos(5);
                temp(cont)     = datos(6);
                hdng(cont)     = datos(7);
                pitch(cont)    = datos(8);
                roll(cont)     = datos(9);
                lat(cont)      = datos(10);
                lon(cont)      = datos(11);

                % Guardar datos
                if app.SaveFlight.Value == 1
                    DatosGuardados = [DatosGuardados; datos'];
                    TimeStamp = [TimeStamp; datetime('now')];
                end

                % DATOS
                if ControlTab == 1
                    app.DatosRecibidos.Value = datos_str;

                    app.VelocidadGPS.Value = string(vel(cont));
                    app.VelocidadIMU.Value = string(vel2(cont));
                    app.AltitudGPS.Value = string(alt(cont));
                    app.Baroaltitud.Value = string(alt2(cont));
                    app.Presion.Value = string(pres(cont));
                    app.Temperatura.Value = string(temp(cont));
                    app.Rumbo.Value = string(hdng(cont));
                    app.Cabeceo.Value = string(pitch(cont));
                    app.Alabeo.Value = string(roll(cont));
                    app.Longitud.Value = string(lat(cont));
                    app.Latitud.Value = string(lon(cont));

                    if lamp == 0
                        app.Lamp.Color = [0,1,0]; lamp = 1;
                    elseif lamp == 1
                        app.Lamp.Color = [1,1,1]; lamp = 0;
                        app.Info.Value="";
                    end

                % INSTRUMENTOS
                elseif InstrumentTab == 1
                    app.AirspeedIndicator.Airspeed = vel(cont);
                    app.AirspeedInercial.Value = vel2(cont);
                    app.AltimetroGPS.Altitude = alt(cont);
                    app.Baroaltimetro.Value = alt2(cont);
                    app.HorizonteArtificial.Pitch = pitch(cont);
                    app.HorizonteArtificial.Roll = roll(cont);
                    app.IndicadordeRumbo.Heading = hdng(cont);
                    app.Termometro.Value = temp(cont);
                end

            elseif ControlTab == 1 && isstrprop(datos_str(1),'digit')
                app.Info.Value="Datos recibidos corruptos";

            elseif ControlTab == 1
                app.Info.Value=erase(datos_str,"Â");
```

```
            end
        end

    end

    %%% TEMPORIZADOR
    function Funcion_GenerarMapa(app,~,~)
        global lat lon latarray lonarray cont gx
        if cont ~= 0
            latarray = [latarray, lat(cont)];
            lonarray = [lonarray, lon(cont)];

            app.PosLabel.Text = join(['Posición actual: (', num2str(lat(end),'%.6f'), ',',
num2str(lon(end),'%.6f'),')']);

            delete(gx);

            gx = geoaxes(app.MapaPanel);

            geoplot(gx,latarray(1),lonarray(1),'p', ...
            latarray,lonarray,'b--',...
            latarray(end),lonarray(end),'o',...
            "Linewidth",2,"MarkerSize",5,"MarkerFaceColor","r","MarkerEdgeColor","r");

            gx.MapCenter = [latarray(end) lonarray(end)];
            gx.ZoomLevel = round(10 + (app.ZoomSpinner.Value - 10) * (20 - 10) / (100 - 10), 1);
            gx.Basemap = 'satellite';

        end
    end
end


    % Callbacks that handle component events
    methods (Access = private)

        % Code that executes after component creation
        function startupFcn(app)
            global Puerto GenerarMapa Start lamp ControlTab InstrumentTab MapaTab cont gx DatosGuardados
TimeStamp

            clc;

            % Cierra los puertos que se hayan podido quedar abiertos
            Puertos_Activos=instrfind; % Lee los puertos activos
            if isempty(Puertos_Activos)==0 % Comprueba si hay puertos activos
                fclose(Puertos_Activos); % Cierra los puertos activos
                delete(Puertos_Activos); % Borra la variable Puertos_Activos
                clear Puertos_Activos; % Destruye la variable Puertos_Activos
            end

            % Configuración del puerto
            if isempty(serialportlist("available"))
                error("Conexión por USB no disponible")
            end
            Puerto = serialport("COM3",115200); % Se crea el objeto serial asociado al COMX
            configureCallback(Puerto,'terminator',@app.Funcion_Recepcion);
            fopen(Puerto);

            GenerarMapa=timer; % Se crea el objeto timer
            GenerarMapa.Period = 1; % Se establece el periodo en segundos
            GenerarMapa.StartDelay = 0; % Se establece el tiempo para iniciar en segundos
            GenerarMapa.ExecutionMode='fixedRate'; % Se establece el modo fixed rate para el timer (los
eventos ocurren con un intervalo fijo)
            GenerarMapa.TimerFcn=@app.Funcion_GenerarMapa; % Cuando se dispare el evento del timer se
ejecutará la función Generar Mapa

            gx = geoaxes(app.MapaPanel);
            geoplot(gx,0,0);
            gx.Basemap = 'satellite';
            gx.ZoomLevel = 2;

            app.ControlButton.Text='Start'; % Se cambia el texto del botón a "Start"
            Start = 0;

            DatosGuardados = [];
            TimeStamp = [];

            app.Lamp.Color = [1,1,1]; lamp = 0;

            ControlTab = 1;
            InstrumentTab = 0;
```

```
            MapaTab = 0;

            cont=0;
        end

        % Value changed function: ControlButton
        function ControlButtonValueChanged(app, event)
            global x cont latarray lonarray
            global vel alt vel2 alt2 pres temp hdng pitch roll lat lon

            Start = app.ControlButton.Value;

            if (Start==1) % Si al hacer clic el botón está en la posición de inicio:
                app.ControlButton.Text='Reset'; % Se cambia el texto del botón a "Reset"
                x=0; % Se inicializa el vector del eje de las abcisas (nº de muestras)

                %Variables
                vel     = 0;
                alt     = 0;
                vel2    = 0;
                alt2    = 0;
                pres    = 0;
                temp    = 0;
                hdng    = 0;
                pitch   = 0;
                roll    = 0;
                lat     = 0;
                lon     = 0;

                latarray = [];
                lonarray = [];

                cont=0; % Se inicializa el contador de muestras a 1

                app.ZoomSpinner.Enable = "on";
                app.SaveFlight.Enable = "on";

            else % Si al hacer clic el botón está en la posición de reset:
                app.ControlButton.Text = 'Start'; % Se cambia el texto del botón a "Start"
                app.Lamp.Color = [1,1,1];
                app.ZoomSpinner.Enable = "off";
                app.SaveFlight.Enable = "off";
            end
        end

        % Value changed function: SaveFlight
        function SaveFlightValueChanged(app, event)
            global Puerto DatosGuardados TimeStamp fecha % Se declaran las variables globales

            value = app.SaveFlight.Value;

            if (value==1) % Si al hacer clic el botón está en la posición de guardar vuelo
                app.SaveFlight.Text = 'Detener guardado';

                fwrite(Puerto,'7');

                fecha = strrep(strrep(datestr(datetime('now')),' ','_'),':','');

            else % Si al hacer clic el botón está en la posición de detener guardado
                app.SaveFlight.Text = 'Comenzar guardado';

                fwrite(Puerto,'8');

                % Generar nombres de variables
                nombres_variables = {'Tiempo','Velocidad GPS', 'Velocidad IMU','Altitud GPS','Alt.
barom.','Presion', 'Temperatura', 'Rumbo', 'Cabeceo', 'Alabeo', 'Latitud','Longitud'};

                % Unir datos en formato cell
                datos_cell = num2cell(DatosGuardados);
                timestamps = cellstr(datestr(TimeStamp));

                AirData = [nombres_variables; timestamps datos_cell];

                % Obtener el nombre del archivo CSV de salida
                nombre_archivo = join(['AirData_APP_',fecha,'.csv']);

                % Abrir el archivo CSV en modo de escritura
                archivo_csv = fopen(nombre_archivo, 'w');

                % Obtener el tamaño de la matriz
                [n_rows, n_col] = size(AirData);
```

```
            % Guardar los datos de la matriz de celdas en el archivo CSV
            for row = 1:n_rows
                for col = 1:n_col
                    % Convertir el contenido de cada celda a texto
                    contenido_celda = mat2str(AirData{row, col});

                    % Escribir el contenido de la celda en el archivo CSV
                    fprintf(archivo_csv, '%s,', contenido_celda);
                end
                fprintf(archivo_csv, '\n'); % Siguiente fila
            end

            % Cerrar el archivo CSV
            fclose(archivo_csv);

        end
    end

    % Close request function: Data
    function closeRequest(app, event)
        global Puerto GenerarMapa

        Puerto = [];
        clear Puerto;

        delete(GenerarMapa)
        clear GenerarMapa;

        delete(app)

    end

    % Value changed function: SwitchTS
    function SwitchTSValueChanged(app, event)
        global Puerto
        swtch = app.SwitchTS.Value;

        if swtch(end) == 'f'        % Valor = Off
            fwrite(Puerto,'0');

        elseif swtch(end) == 'n'    % Valor = On
            fwrite(Puerto,'1');

        end

    end

    % Button down function: PaneldecontrolTab
    function PaneldecontrolTabButtonDown(app, event)
        global ControlTab InstrumentTab MapaTab GenerarMapa

        ControlTab = 1;
        InstrumentTab = 0;
        MapaTab = 0;

        stop(GenerarMapa)
    end

    % Button down function: PaneldeinstrumentosTab
    function PaneldeinstrumentosTabButtonDown(app, event)
        global ControlTab InstrumentTab MapaTab GenerarMapa

        ControlTab = 0;
        InstrumentTab = 1;
        MapaTab = 0;

        stop(GenerarMapa)
    end

    % Button down function: MapaTab
    function MapaTabButtonDown(app, event)
        global ControlTab InstrumentTab MapaTab GenerarMapa

        ControlTab = 0;
        InstrumentTab = 0;
        MapaTab = 1;

        start(GenerarMapa)
    end
end
```

A.4 Visualizador de ThingSpeak: Mapa

```
% Visualizador de ubicación por Carlos Cantarino, 2023

% Channel Read API Key
readAPIKey = 'OILOCQQBTUG5ZLW1';
% Channel ID to read data from
readChannelID = 2149521;


% Latitude Field ID
LatFieldID = 7;
% Longitude Field ID
LonFieldID = 8;


% Variables
tiempo  = 2; % Cuánto tiempo de vuelo mostrar en minutos
tsfreq  = 15; % Frecuencia en segundos de subida de datos a ThingSpeak
res     = 0.001; % Resolución (más pequeño, más zoom)


% Obtener latitud y longitud
n = tiempo*60/tsfreq;
lat = ThingSpeakRead(readChannelID,'Fields',LatFieldID,...
    'NumPoints',n,'ReadKey',readAPIKey);
lon = ThingSpeakRead(readChannelID,'Fields',LonFieldID,...
    'NumPoints',n,'ReadKey',readAPIKey);


% Colores para los marcadores: azul punto inicial, verde intermedios, rojo
final.
colormap([0 0 1; 0 1 0; 1 0 0]);
if n == 1
    colors = 1;
elseif n == 2
    colors = [1 3];
else
    colors = 2 * ones(1,n-2);
    colors = [1 colors 3];
end

% Crear geoscatter plot
%          Coordenadas  Tamaño   Colores   Relleno
geoscatter(lat, lon,    50,      colors,   'filled','MarkerEdgeColor','k');


geobasemap satellite % Tipo de mapa (satelital en este caso)
geolimits([lat(n)-res lat(n)+res],[lon(n)-res lon(n)+res]) % Límites del
mapa
```

A.5 Visualizador de ThingSpeak: Rumbo

```
% Visualizador de rumbo por Carlos Cantarino, 2023

% Channel Read API Key
readAPIKey = 'OILOCQQBTUG5ZLW1';
% Channel ID to read data from
readChannelID = 2149521;


% Heading Field ID
HdnFieldID = 4;


% Obtener último valor de rumbo
hdn = ThingSpeakRead(readChannelID,'Fields',HdnFieldID,'NumPoints',1,
'ReadKey',readAPIKey);


% Convertir a radianes
rad = hdn*2*pi/360;


% Coordenadas polares (puntos extra para crear flecha)
theta = [0, rad, rad+0.1, rad, rad-0.1, rad];
r     = [0, 0.8,     0.8,   1,     0.8, 0.8];

% Generar brújula
polarplot(theta, r,'LineWidth',3,'Color','r');


% Propiedades
ax = gca;
ax.ThetaZeroLocation = 'top'; % Etiqueta de "0º" en la parte superior
ax.ThetaDir = 'clockwise';    % Números en sentido horario
ax.ThetaTick = [0:15:345];    % Resolución
ax.RTickLabel = {};           % Sin etiquetas en la dirección r
```

A.6 Guardado local con MATLAB de datos de ThingSpeak

```
% Guardado local de datos de ThingSpeak por Carlos Cantarino, 2023

% Channel ID y API key
readChannelID = 2149521;
readAPIKey = 'OILOCQQBTUG5ZLW1';

% Muestras a guardar
tiempo  = 10; % Cuánto tiempo de vuelo mostrar en mins (max. 133 min a 1Hz)
tsfreq  = 15; % Frecuencia en segundos de subida de datos a ThingSpeak

% Leer de ThingSpeak
n = tiempo*60/tsfreq;
[data, time] = ThingSpeakRead(readChannelID,'ReadKey',
readAPIKey,'Fields',1:8,'NumPoints',n);

% Encontrar y eliminar filas con valores NaN
filas_nan = any(isnan(data), 2);
data = data(~filas_nan, :);
time = time(~filas_nan, :);

% Unir datos en formato cell
datos_cell = num2cell(data);
timestamps = cellstr(datestr(time));
nombres_variables = {'Tiempo','Velocidad', 'Altitud', 'Temperatura',
'Rumbo', 'Cabeceo', 'Alabeo', 'Latitud','Longitud'};

AirData = [nombres_variables; timestamps datos_cell];

% Obtener el nombre del archivo CSV de salida
fecha = strrep(strrep(datestr(time(1)),' ','_'),':','');

nombre_archivo = join(['AirData_TS_',fecha,'.csv']);

% Abrir el archivo CSV en modo de escritura
archivo_csv = fopen(nombre_archivo, 'w');

% Obtener el tamaño de la matriz
[n_rows, n_col] = size(AirData);

% Guardar los datos de la matriz de celdas en el archivo CSV
for row = 1:n_rows
    for col = 1:n_col
        % Convertir el contenido de cada celda a texto
        contenido_celda = mat2str(AirData{row, col});

        % Escribir el contenido de la celda en el archivo CSV
        fprintf(archivo_csv, '%s,', contenido_celda);
    end
    fprintf(archivo_csv, '\n'); % Siguiente fila
end

% Cerrar el archivo CSV
fclose(archivo_csv);
```

A.7 Guardado local con MATLAB de vuelos de ThingSpeak

```
% Guardado local de vuelos de ThingSpeak por Carlos Cantarino, 2023

% Channel ID y API key
readChannelID = 2149521;
readAPIKey = 'OILOCQQBTUG5ZLW1';

% Muestras a guardar
[data, time] =
ThingSpeakRead(readChannelID,'ReadKey',readAPIKey,'Fields',1:8,'NumPoints',
8000);

% Generar nombres de variables
nombres_variables = {'Tiempo','Velocidad', 'Altitud', 'Temperatura',
'Rumbo', 'Cabeceo', 'Alabeo', 'Latitud','Longitud'};

% Encontrar y eliminar filas con valores NaN
filas_nan = any(isnan(data), 2);
data = data(~filas_nan, :);
time = time(~filas_nan, :);

% Buscar marcadores de vuelo
[N, ~] = find(data(:,1) == -100);

for i=1:length(N) % Crear un archivo CSV para cada vuelo

    n = N(i) + 1; % Punto de inicio

    dataFlight = []; % Nueva matriz de datos con solo el vuelo guardado
    timeFlight = []; % Nuevas etiquetas de tiempo con solo el vuelo
guardado


    while (data(n,1) ~= -100 && data(n,1) ~= -200)

        dataFlight = [dataFlight; data(n,:)];
        timeFlight = [timeFlight; time(n)];

        n = n + 1;
        if n > size(data,1) % ¿Se ha llegado al fin de la matriz?
            break;
        end
    end

    % Unir datos en formato cell
    datos_cell = num2cell(dataFlight);
    timestamps = cellstr(datestr(timeFlight));

    AirData = [nombres_variables; timestamps datos_cell];

    % Obtener el nombre del archivo CSV de salida
    fecha = datestr(timeFlight(1));
    fecha = fecha(1:11); % Recortamos la fecha para quitar la hora

    nombre_archivo = join(['AirData_TS_',fecha,'_Vuelo-',
num2str(i),'.csv']);
```

```
    % Abrir el archivo CSV en modo de escritura
    archivo_csv = fopen(nombre_archivo, 'w');

    % Obtener el tamaño de la matriz
    [n_rows, n_col] = size(AirData);

    % Guardar los datos de la matriz de celdas en archivo CSV
    for row = 1:n_rows
        for col = 1:n_col
            % Convertir el contenido de cada celda a texto
            contenido_celda = mat2str(AirData{row, col});

            % Escribir el contenido de la celda en el archivo CSV
            fprintf(archivo_csv, '%s,', contenido_celda);
        end
        fprintf(archivo_csv, '\n'); % Siguiente fila
    end

    % Cerrar el archivo CSV
    fclose(archivo_csv);

end
```

A.8 Mapeado 2D y 3D con MATLAB de vuelos guardados

```
% Mapeado 3D y 2D de vuelos guardados (tanto ThingSpeak como app de PC).
% Por Carlos Cantarino, 2023

% Nombre del archivo CSV
filename = '';

% Referencia para altitud
ref = 'terrain'; % Puede ser altitud respecto a 'geoid' (MSL), 'terrain'
(suelo) o 'ellipsoid' (WGS84).

% Leer archivo CSV
data = readmatrix(filename, 'Delimiter', ',');

data(:,1) = [];

% Encontrar y eliminar filas con valores NaN
filas_nan = any(isnan(data), 2);
data = data(~filas_nan, :);

filas_nan = any(isnan(data), 2);
data = data(~filas_nan, :);

% Extraer datos necesarios

if filename(9)=='A' % El archivo proviene de guardado local de app de
MATLAB
    alt = data(:,4);
    lat = data(:,10);
    lon = data(:,11);

elseif filename(9)=='T'
    alt = data(:,2);
    lat = data(:,7);
    lon = data(:,8);
else
    error('Nombre de archivo incorrecto')
end

close all
% Graficar mapa en 2D
figure(1)
geoplot(lat,lon,'c-o',"Linewidth",2,
"MarkerSize",4,"MarkerFaceColor","b","MarkerEdgeColor","b");
hold on
geoplot(lat(1),lon(1),'o',"Linewidth",3,
"MarkerSize",8,"MarkerFaceColor","y","MarkerEdgeColor","b");
geoplot(lat(end),lon(end),'o',
"MarkerSize",8,"MarkerFaceColor","r","MarkerEdgeColor","r");
gx = gca;
gx.Basemap = 'satellite';
gx.MapCenter = [lat(end),lon(end)];
e = 0.0005; % Espaciado
latLim = [min(lat)-e, max(lat)+e];
lonLim = [min(lon)-e, max(lon)+e];
geolimits(latLim, lonLim);
```

```
% Mapa 3D
uif = uifigure;
g = geoglobe(uif);
geoplot3(g,lat,lon,alt,"r-o","HeightReference",ref,"LineWidth",3,
"MarkerSize",6,"MarkerIndices",length(lat))
```

B. FICHAS TÉCNICAS

Los componentes que conforman el prototipo del sistema de procesado, monitorizado y almacenamiento de datos y los enlaces a sus respectivas fichas técnicas se listan a continuación:

- Microcontrolador ESP32. Ficha técnica accedida el 16 de julio de 2023 desde: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

- Módulo LoRa SX1278. Ficha técnica accedida el 16 de julio de 2023 desde: https://cdn-shop.adafruit.com/product-files/3179/sx1276_77_78_79.pdf

- Sensor BMP280. Ficha técnica accedida el 16 de julio de 2023 desde: https://cdn-shop.adafruit.com/datasheets/BST-BMP280-DS001-11.pdf

- IMU BNO055. Ficha técnica accedida el 16 de julio de 2023 desde: https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf

- GPS BN-880. Ficha técnica accedida el 16 de julio de 2023 desde: https://store.beitian.com/products/beitian%2dcompass%2dqmc5883l%2damp2%2d6%2dpix4%2dpixhawk%2dgnss%2dgps%2dglonass%2ddual%2dflight%2dcontrol%2dgps%2dmodule-bn-880q?_pos=1&_sid=390d0f576&_ss=r&variant=44696120295711

- Convertidor PowerBoost 500C. Ficha técnica accedida el 16 de julio de 2023 desde: https://learn.adafruit.com/adafruit-powerboost/downloads

- Batería LiPo EEMB LP653042. Ficha técnica accedida el 16 de julio de 2023 desde: https://oss.eemb.com//uploads/20230315/28f8e7468414d2b421c8660279f01e45.pdf