Universidad Zaragoza
1542

Trabajo Fin de Máster

Sensor inteligente para monitorización de animales con técnicas de bajo consumo

Smart sensor for animal monitoring with low-power techniques

Autor

Fernando Herranz Rodríguez

Director

Roberto Casas Nebra

Escuela de Ingeniería y Arquitectura
2020

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster)

(Título del Trabajo)

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza,

Fdo:

Agradecimientos

En primer lugar, me dirijo a mi familia para agradecerles su apoyo incondicional y buenos consejos, no sólo durante la realización de este Trabajo Fin de Máster, sino durante todo el camino recorrido en estos 6 años de estudios universitarios.

Por supuesto, dar las gracias a todos mis compañeros del grupo de investigación HOWLab de la Universidad de Zaragoza, por la ayuda prestada siempre que la necesitaba. En especial, agradecer a mi tutor en este proyecto, Roberto Casas, por las lecciones aprendidas y por la confianza depositada en mí.

Por último, me gustaría acordarme de todos los compañeros y amigos que me han acompañado durante mis estudios en el Grado y Máster de Ingeniería de Telecomunicaciones, por los momentos vividos y lo mucho que he aprendido de todos ellos.

Gracias a todos.

RESUMEN

El estudio del comportamiento animal permite conocer los mecanismos de supervivencia de las especies, analizar su bienestar e incluso detectar enfermedades. Tradicionalmente, este estudio ha sido realizado por parte del personal Veterinario, a través de la observación del animal. Sin embargo, esta tarea supone una gran inversión de tiempo.

En este Trabajo Fin de Máster se ha desarrollado un sensor inteligente adherido al cuerpo del animal que permite obtener una descripción precisa y constante de sus movimientos. Este tipo de dispositivos se conocen habitualmente como *wearables*, y suelen presentar un gran inconveniente en términos de duración de la batería, que limita muchas veces su implementación. Como respuesta a este problema, en este proyecto se utilizan diversas técnicas y tecnologías que permiten minimizar el consumo energético, como por ejemplo realizar la lectura del movimiento del animal mientras los procesadores del sensor están dormidos.

La información inercial recabada por el sensor se envía a un servidor externo situado en Internet, lo que se conoce habitualmente como la "nube". Con la finalidad de monitorizar a medio plazo un número elevado de animales en un espacio reducido, se ha seleccionado la tecnología WiFi para el envío.

Como protocolo de nivel de aplicación y formato de los datos enviados se han utilizado MQTT y SenML, respectivamente. Ambos son estándares de IoT y permiten, entre otras cosas, reducir el consumo y asegurar la interoperabilidad con otras aplicaciones. El almacenamiento de la información enviada se produce en una base de datos de series temporales. A través de Internet, es posible acceder a ella para su representación y futuro procesado.

Para la validación del sistema propuesto al completo se han diseñado dos prototipos del sensor inteligente: uno inicial para la experimentación en un escenario controlado con el movimiento del cuerpo humano, y otro más compacto en forma de placa de circuito impreso para la monitorización de las ovejas de la Facultad de Veterinaria de la Universidad de Zaragoza.

Acrónimos

HTTP – Hypertext Transfer Protocol

FTP – File Transfer Protocol

IoT – Internet of Things

IMU – Inertial Measurement Unit

MQTT – Message Queue Telemetry Protocol

QoS – Quality of Service

SenML – Sensor Measurement Lists

JSON – JavaScript Object Notation

XML – Extensible Markup Language

OSI – Open Systems Interconnection

HOWLab – Human Openware Research Lab

URN – Uniform Resource Name

SSID – Service Set Identifier

NVS – Non-Volatile Storage

UART – Universal Asynchronous Receiver-Transmitter

TSDB – Time Series Database

MCU – Microcontroller Unit

BLE – Bluetooth Low Energy

SPI – Serial Peripheral Interface

SoC – System On a Chip

GPIO – General Purpose Input/Output

I2C – Inter-Integrated Circuit

ADC – Analog-to-Digital Converter

DOF – Degrees of Freedom

MEMS – Micro-Electromechanical System

DHCP – Dynamic Host Configuration Protocol

Tabla de contenidos

Lista de figuras

Lista de tablas

1. Introducción

1.1. Motivación

El estudio científico del comportamiento de los animales se conoce como etología. El comportamiento de los animales es de especial interés para detectar enfermedades, analizar su bienestar e incluso entender sus mecanismos de supervivencia. Este estudio se realiza habitualmente mediante la grabación con cámaras y su posterior observación, lo que supone una gran inversión de tiempo por parte del personal veterinario.

Las ovejas, en particular, se ven afectadas en muchas ocasiones por virus que afectan a su comportamiento. Además, algunas de las vacunas aplicadas a estos animales pueden provocar también una alteración de su conducta. Otra de las enfermedades habituales en ovejas es la cojera, que afecta a un gran número de éstas y que no es posible observar fácilmente, entre otras cosas, porque estos animales tienden a quedarse casi estáticos ante la presencia de humanos.

El objetivo último de esta línea de trabajo consiste en llegar a distinguir entre una oveja sana y otra con una conducta anómala, de forma automatizada. Con este fin, se propone la monitorización constante del movimiento de las ovejas mediante tecnologías propias de IoT.

El término IoT fue utilizado por primera vez por Kevin Ashton (Procter & Gamble) en 2009. Este concepto hace referencia a la interacción autónoma entre sistemas, dotándoles de cierta inteligencia que les permita tomar decisiones y recabar información del mundo que nos rodea [1]. El elemento clave del IoT son los sensores, los cuales permiten obtener esa información.

Los sensores miniaturizados que se pueden adherir al cuerpo humano o al de un animal se conocen con el nombre de *wearables*. A través de *wearables*, es posible tener información del movimiento de las ovejas durante largos periodos de tiempo, sin que la conducta de la oveja esté afectada por factores como incomodidad o intimidación.

Si, adicionalmente, esa información es accesible a través de Internet, mediante un procesado y análisis de esos datos de movimiento sería posible identificar qué ovejas están sanas y cuáles no.

1.2. Objetivos

Los principales objetivos abordados en la realización de este Trabajo Fin de Máster son:

- Desarrollo de *firmware* para monitorización constante de la información inercial de animales, minimizando la pérdida de datos y el consumo energético.
- Diseño, fabricación y validación de prototipos *hardware* del sensor inteligente *wearable*.
- Subida de datos a la nube y almacenamiento en bases de datos.
- Utilización de protocolos y estructuras típicas de *Internet Of Things*.
- Realización de pruebas en ovejas de la Facultad de Veterinaria para validación del sistema.
- Interpretación de los datos de monitorización y reconocimiento automático de patrones.

1.3. Descripción del sistema

El sistema desarrollado en este Trabajo Fin de Máster consiste en un sensor inteligente adherido a modo de *wearable* a una oveja de la Facultad de Veterinaria. El sensor monitoriza de forma constante la información inercial del animal y la sube a la nube vía WiFi, a través del protocolo de nivel de aplicación MQTT, con el formato típico de IoT SenML. Esta información se almacena en una base de datos, a la cual se puede acceder desde Internet para su análisis.

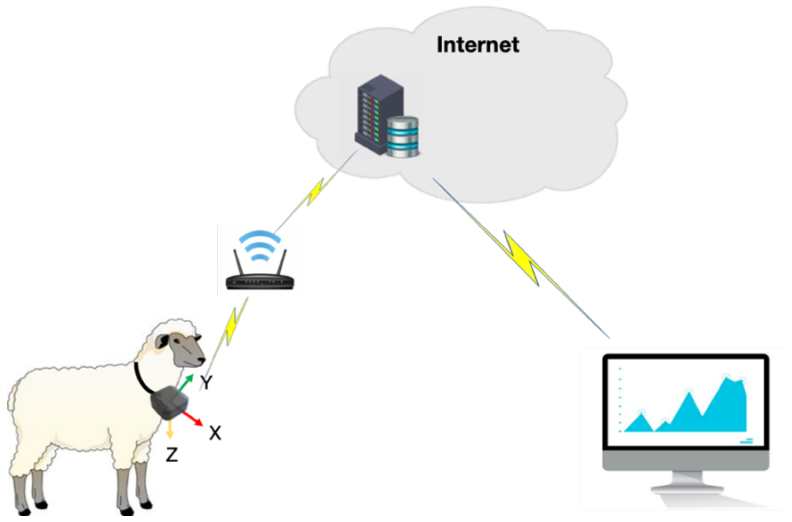

Figura 1. Diagrama simplificado del sistema propuesto en este TFM

El sensor inteligente consta fundamentalmente de dos bloques: (1) un dispositivo IoT con comunicaciones embebidas que hará las veces de microcontrolador, y (2) un sensor IMU que realizará la medición de las variables inerciales -explicado en detalle en el *Anexo I*-.

El dispositivo IoT con comunicaciones seleccionado es la LoPy4, de Pycom. El principal motivo de esta elección es que puede convertirse en un *hardware* comodín para aplicaciones de IoT, ya que permite comunicaciones por WiFi, BLE, Sigfox y LoRa. De este modo, podría usarse para múltiples escenarios diferentes en términos de alcance, cantidad de datos a enviar, tasa de envío, etc. Todo ello, mediante una programación con Micropython, más sencillo e intuitivo que otros lenguajes de programación de microcontroladores.

Sin embargo, como se verá en el desarrollo de este TFM, para tener un mayor control sobre el *hardware* y alcanzar consumos de energía más bajos -objetivo fundamental en IoT- será necesario eliminar las librerías de Micropython del dispositivo y realizar la programación -en lenguaje C y ensamblador- directamente sobre el microcontrolador interno de la LoPy4: el ESP32.

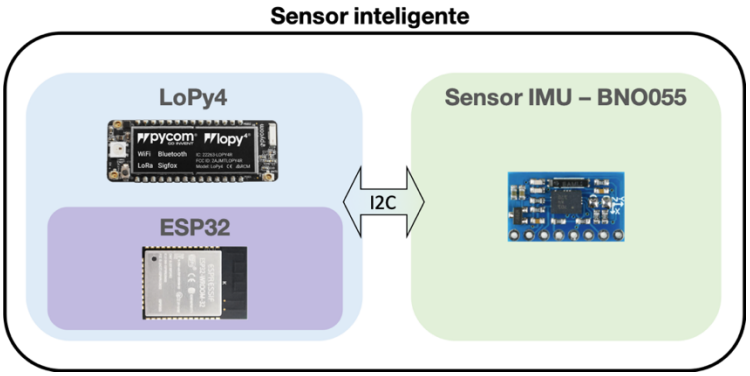

Figura 2. Diagrama de bloques simplificado del sensor inteligente

1.4. Estado del arte

La observación continua del animal para su estudio es una tarea tediosa y larga, bien sea en tiempo real o incluso a posteriori mediante la grabación con cámaras. Sin embargo, con la aparición en los últimos años de sensores inerciales microelectromecánicos MEMS, se ha abierto la posibilidad de automatizar estos procesos, permitiendo obtener datos de comportamiento animal durante largos periodos de tiempo [2].

Las unidades inerciales de medida IMU contienen múltiples sensores -típicamente acelerómetros, giróscopos y magnetómetros- y proporcionan hoy en día la descripción de

movimiento más exacta posible. Por ello, son utilizados en aplicaciones que requieren una precisión máxima, como por ejemplo misiles y aeronaves [3].

El desarrollo en la actualidad de unidades IMU basadas en sensores MEMS permite alcanzar tamaños, costes y consumos de energía muy reducidos. Esto las convierte en una solución adoptada en múltiples aplicaciones de IoT de descripción de movimiento.

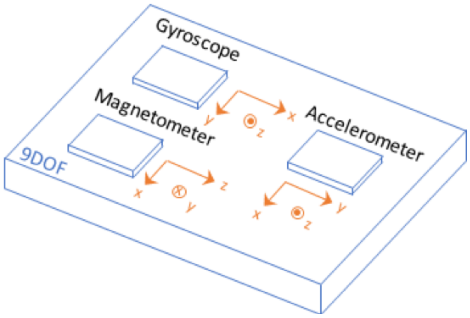

Figura 3. Sensor IMU 9-DOF: acelerómetro, giróscopo y magnetómetro

En este nuevo escenario de monitorización, la tarea de observación se limita a un periodo reducido de tiempo, que sea suficiente para etiquetar con precisión la información inercial en base a la cual, mediante técnicas de *machine learning*, se podrán diferenciar automáticamente patrones de comportamiento.

Los primeros experimentos de estudio animal mediante *wearables* adheridos a su cuerpo estaban basados en una descarga posterior -no en tiempo real- de las medidas efectuadas por el sensor, para entonces realizar el procesado de los datos y eventualmente clasificar el comportamiento.

En esta línea de post-procesado de los datos, Painter et al., a partir de medidas tomadas con acelerómetros y magnetómetros, realizaron una clasificación del comportamiento de los zorros en tres clases: saltar, correr y búsqueda de alimento [4]. Otro ejemplo es el de Guo et al., que a través de sensores IMU hicieron una comparación del comportamiento de las ovejas durante el pasto en función de la altura de la hierba [2].

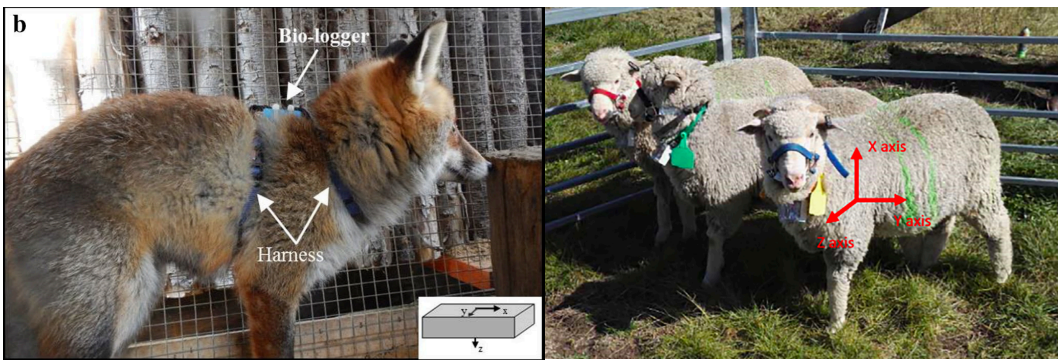

Figura 4. Ejemplos de monitorización animal con wearables en [4] (izq) y [2] (dcha)

Dadas las altas tasas de muestreo de los sensores inerciales utilizados para el estudio del comportamiento animal, se manejan grandes volúmenes de datos que deben ser enviados de forma inalámbrica por el *wearable*, por ejemplo, a través de WiFi. Esto supone un desafío casi insuperable para el sensor inteligente, dadas sus limitaciones en batería y almacenamiento. Por ello, en aplicaciones más actuales resulta inevitable realizar el procesado -o al menos parte de él- en el propio *wearable* [5].

Un ejemplo de procesado y clasificación de comportamiento en tiempo real es el propuesto en [6] para la identificación de actividades en ovejas y rinocerontes, a partir de las medidas de los tres ejes de un acelerómetro. Dadas las limitaciones del sensor inteligente, en este experimento se propuso la utilización de un clasificador lineal LDA, debido a que demanda una menor cantidad de recursos y tiempo de computación.

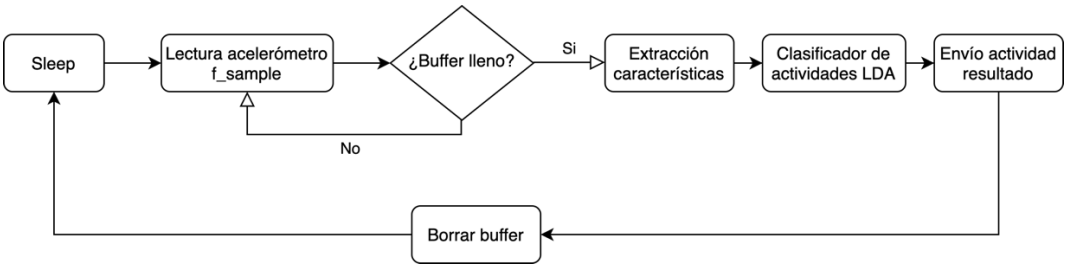

Figura 5. Diagrama de flujo simplificado propuesto en [6] para monitorización de animales

El experimento de Kaler et al. [7] en noviembre de 2019 para la detección de cojera en ovejas puede servir como referencia para analizar la precisión con la que se clasifica el comportamiento de estos animales en la actualidad. Estos autores proponen un clasificador en el propio *wearable* en tres actividades para la oveja: en pie sin movimiento, andando y tumbada. La información inercial es proporcionada por acelerómetros y giróscopos.

La precisión con la que finalmente estos autores determinan si una oveja tiene cojera o no es aproximadamente del 80%. Estos resultados pueden obtenerse a partir de cualquiera de las tres actividades en que se divide el comportamiento de la oveja. Sin embargo, la técnica de aprendizaje con la que se obtiene una mayor precisión es la de combinación por *bagging* de árboles de decisión mediante *Random Forests*.

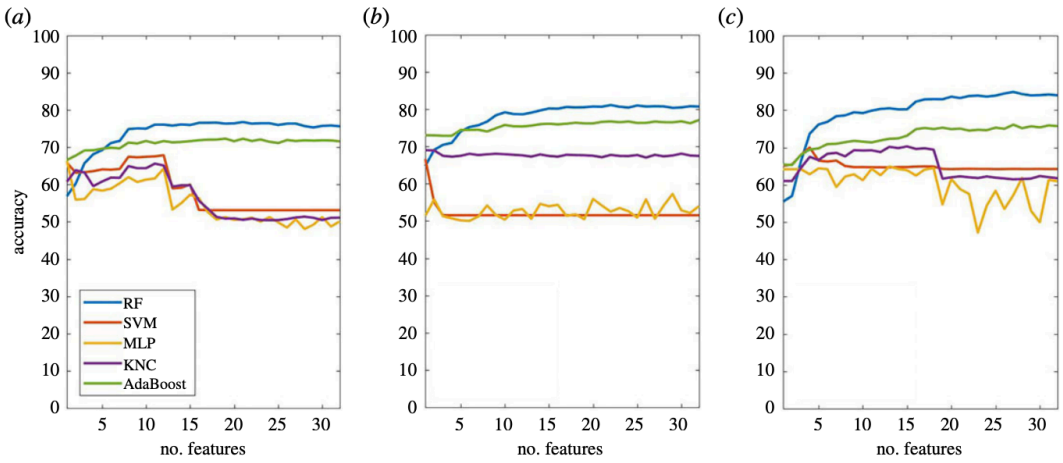

Figura 6. Evaluación de técnicas para detección de cojera en ovejas: (a) andando, (b) de pie y (c) tumbada [7]

1.5. Estructura de la memoria

La memoria del Trabajo Fin de Máster que se presenta en este documento queda organizada de la siguiente manera.

El apartado 2 *Hardware* comienza con el criterio seguido para la selección del sensor IMU y muestra el estudio de los modos de operación del sensor elegido. Posteriormente se detallan los dos prototipos diseñados: uno utilizado para la monitorización de la actividad del cuerpo humano, y otro para la realización de pruebas en animales.

En el apartado 3 *Desarrollo software del sensor inteligente* se muestra el problema inicial del consumo energético y la solución adoptada en este proyecto para afrontarlo. Posteriormente se da una visión general de las limitaciones *hardware* del dispositivo que se deben tener en cuenta en la programación y el algoritmo de monitorización propuesto. Finalmente se introduce el procedimiento de configuración planteado para el sensor inteligente.

El apartado 4 *Envío y análisis de los datos* detalla las comunicaciones inalámbricas del sensor para la subida de información a la nube, incluyendo la instalación WiFi realizada. Además, se introduce el formato de los datos enviados y el almacenamiento en bases de datos de series temporales. En el segundo bloque de este apartado se expone el análisis de los datos de monitorización en un escenario controlado, conformado por un pre-procesado y una posterior agrupación en categorías mediante algoritmos de *machine learning* de aprendizaje no supervisado.

El apartado 5 *Resultados experimentales* se divide en dos partes. En primer lugar, se muestra un estudio del consumo energético del sistema final. Seguidamente, se presentan los datos de monitorización obtenidos mediante las pruebas en ovejas de la Facultad de Veterinaria.

En el apartado 6 *Conclusiones y líneas de futuro* se describen los resultados alcanzados con este Trabajo Fin de Máster y el desarrollo futuro en esta línea de trabajo.

Finalmente, se presentan los *Anexos* y las *Referencias* utilizadas para la realización de este documento.

2. Hardware

El sensor inteligente desarrollado en este Trabajo Fin de Máster consta de dos elementos básicos: un sensor para medir variables inerciales y la LoPy4, un microcontrolador con comunicaciones embebidas para implementar el algoritmo de monitorización y gestionar las comunicaciones.

En este apartado se muestran en primer lugar los criterios de selección del sensor IMU BNO055 como dispositivo de medida, así como un análisis de sus modos de operación. Posteriormente, se presentan los dos prototipos desarrollados en este proyecto, uno de ellos más compacto, en forma de PCB, diseñado con el *software* CircuitMaker y soldado a mano.

2.1. Selección del sensor y estudio de funcionamiento

En este proyecto se busca realizar una medida precisa del movimiento y posición de un animal, en este caso de una oveja. El consumo energético debería ser lo más bajo posible, puesto que es necesaria una monitorización constante por parte del sensor, es decir, que sea *"always-on"*.

Los sensores IMU se han convertido en una solución muy extendida para este tipo de aplicaciones debido a que permiten obtener la aceleración de un cuerpo, su velocidad angular y en algunos casos su orientación relativa o absoluta en el espacio, mediante una combinación de los datos procedentes de (1) acelerómetros, (2) giróscopos y (3) magnetómetros. Estos dispositivos permiten medir, respectivamente: (1) aceleración, (2) orientación y velocidad angular, y (3) campos magnéticos.

Además, con el desarrollo en la actualidad de sensores IMU basados en acelerómetros, giróscopos y magnetómetros MEMS se alcanzan tamaños, costes y consumos de energía muy reducidos, convirtiéndose así en una solución muy atractiva para IoT [8].

Típicamente, los sensores IMU tienen dos modos de operación: no-fusión y fusión [9].

- Modo no-fusión. Los datos que proporciona el sensor IMU son directamente las medidas realizadas por los sensores almacenados en él, i.e. acelerómetros, giróscopos y magnetómetros.
- Modo fusión. El sensor IMU realiza una combinación de los datos procedentes de los diferentes sensores que posee. Como resultado se obtienen precisiones mucho mayores en las medidas, además de la posibilidad de calcular orientaciones absolutas en el espacio.

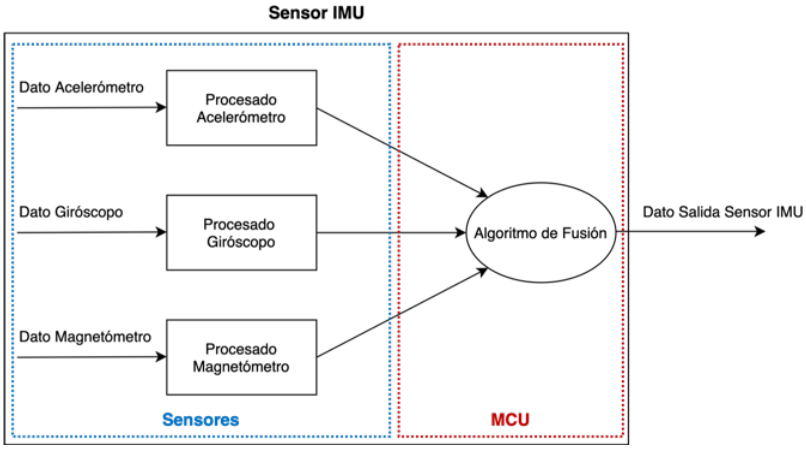

Figura 7. Componentes de un sensor IMU

Por lo tanto, el objetivo es buscar un sensor IMU que permita trabajar en modo fusión y alcanzar una buena calidad de dato, sin comprometer otras características importantes para la aplicación como son tamaño, precio y consumo de energía. Esto se deberá fundamentalmente a lo bueno que sea el algoritmo de fusión embebido en el microcontrolador presente en el IMU.

El circuito integrado BNO055 [10] es una opción muy extendida en aplicaciones de *wearables* y robótica, y que además ha sido utilizado previamente en el grupo de investigación HOWLab. En este sensor IMU, tanto el *hardware* como el *software* -es decir, el algoritmo de fusión de datos- han sido diseñados por el mismo fabricante, Bosch Sensortec.

Sin embargo, hay una tendencia creciente en la que el fabricante del circuito integrado colabora con empresas especializadas en *software*, de tal forma que son éstas las que se encargan de desarrollar el algoritmo de fusión embebido. Así, la empresa Hillcrest Labs ha lanzado al mercado el sensor IMU BNO085 [11], que comparte el mismo *hardware* que el BNO055, pero con su algoritmo de fusión de datos propio.

Se ha realizado un estudio comparativo de estas dos aproximaciones a la hora de elegir un sensor IMU en el *Anexo IV*. Por las características de la aplicación de este Trabajo Fin de Máster se ha decidido utilizar el BNO055 como sensor IMU para la monitorización de los animales.

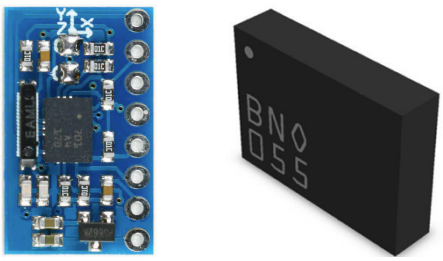

Figura 8. Módulo BNO055 utilizado (izq) y chip BNO055 de Bosch Sensortec (dcha)

El módulo BNO055 empleado en este proyecto ya dispone del circuito recomendado por Bosch Sensortec para el sensor IMU. Además, es posible soldar fácilmente en los pines deseados. A pesar de que el módulo BNO055 presenta 8 pines, tan sólo se van a utilizar 4 de ellos: 2 para comunicación I2C con la LoPy4 (SDA y SCL), 1 para alimentación a 3V3 y 1 para unir las masas.

Estudio de los modos de operación del sensor BNO055

El BNO055 de Bosch Sensortec permite múltiples modos de operación, que implican básicamente qué sensores intervienen en la realización de las medidas, y cuáles no. Lógicamente, un mayor número de sensores activos y la ejecución del algoritmo de fusión implican un incremento en el consumo energético del sensor.

Los modos de operación del sensor que pueden resultar más interesantes en una aplicación como la presentada en este TFM son: (1) acelerómetro, (2) giróscopo, (3) acelerómetro y giróscopo, (4) acelerómetro, giróscopo y magnetómetro, y (5) 9-DOF. El único de estos modos que supone la utilización del algoritmo de fusión es el 9-DOF, donde se calcula la orientación absoluta en forma de cuaternios a partir de los datos del resto de sensores.

Modos de operación	**Variables inerciales medidas**			
	Aceleración lineal	**Velocidad angular**	**Campo magnético**	**Orientación absoluta**
ACC	✓			
GYR		✓		
ACC+GYR	✓	✓		
ACC+GYR+MAG	✓	✓	✓	
9-DOF	✓	✓	✓	✓

Tabla 1. Variables inerciales medidas en los diferentes modos de operación del BNO055

Se ha medido experimentalmente el consumo de corriente del sensor en los diferentes modos de operación. En el caso de este Trabajo Fin de Máster se utiliza el BNO055 en modo 9-DOF, puesto que se desea obtener una descripción completa del movimiento del animal. Sin embargo, en etapas más avanzadas de diseño del sensor inteligente será posible reducir notablemente el consumo, una vez que se haya analizado qué información es realmente relevante y se haya seleccionado un modo de operación más específico.

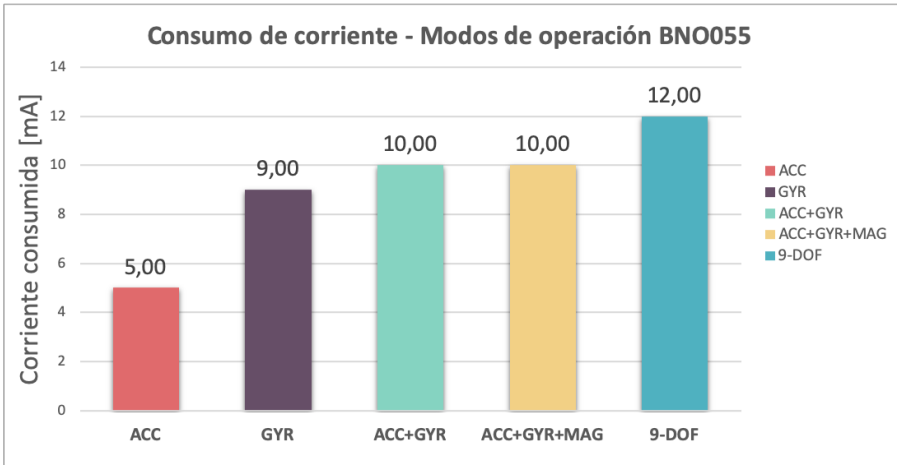

Figura 9. Consumos de corriente de los diferentes modos de operación del BNO055

2.2. Prototipo inicial

Se ha construido un primer prototipo del sensor inteligente -previo a la fabricación del prototipo definitivo-, para su utilización en el primer experimento realizado en este TFM, correspondiente a la monitorización de la actividad del cuerpo humano en un escenario controlado. Este *hardware* consta de los elementos básicos que permiten realizar la tarea de monitorización: la LoPy4 y el sensor IMU.

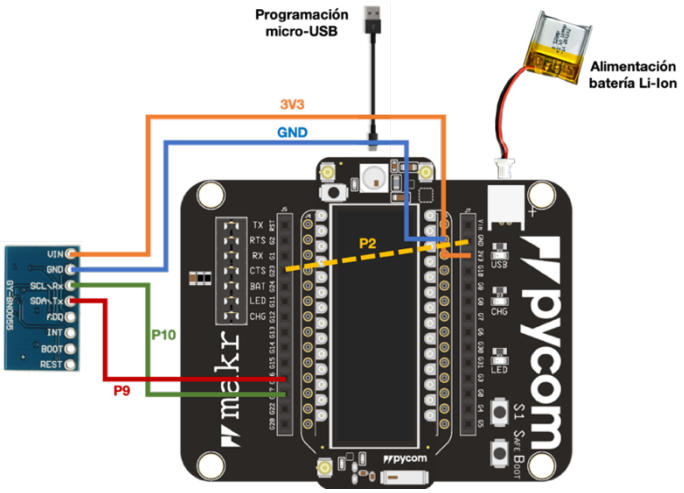

Figura 10. Conexionado del prototipo inicial del sensor inteligente

En este caso la placa de expansión 3.0 de Pycom forma parte del *hardware*, dado que (1) facilita el conexionado mediante pines -evitando así tener que soldar-, (2) permite cargar programas en la LoPy4 fácilmente con micro-USB, y (3) dispone de un conector para batería,

permitiendo además su recarga. La LoPy4 irá colocada a modo de *shield* sobre esta placa de expansión.

El sensor IMU BNO055 seleccionado se controla a través de I2C. La línea de datos SDA del sensor va conectada al pin P9 de la LoPy4, y la línea de reloj SCL al pin P10. También es necesario unir las masas de ambos dispositivos mediante el pin GND, y proporcionar alimentación al BNO055 a través del pin 3V3.

La batería seleccionada es de tecnología Li-Ion y tiene un amperaje de 4000mAh. Esto significa sencillamente que, si el sistema consumiera 4A, entonces la duración de la batería sería de 1 hora.

Como se indica en el *Anexo III*, es necesario colocar un *jumper* entre el pin P2 y GND para poder cargar programas escritos en lenguaje C y ensamblador en la LoPy4. En la sección 3.1.1. *Solución propuesta: programación a más bajo nivel*, se explica en detalle esta circunstancia.

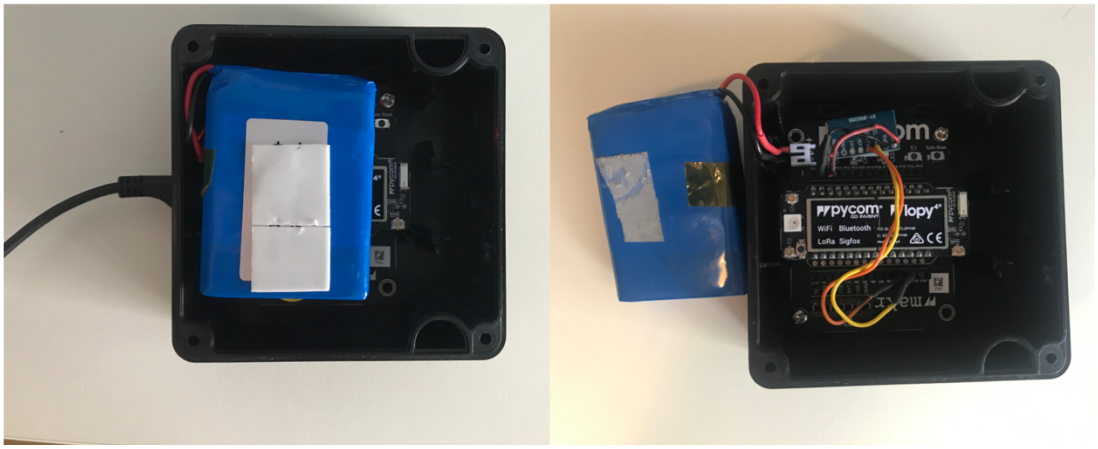

Figura 11. Imagen real del prototipo inicial del sensor inteligente

Todo el montaje se coloca en el interior de una caja de dimensiones 80x80x40mm, conformando un peso total aproximado de 190 gramos. Dicha caja tiene una abertura para el puerto micro-USB, de tal forma que no haya que abrirla para cargar la batería. Sin embargo, el hecho de que no sea estanca implicaría tener que forrarla a posteriori para evitar la entrada de agua, si se deseara utilizar este prototipo para la monitorización de los animales.

2.3. Prototipo definitivo

Para la realización de pruebas en animales se ha diseñado una PCB como *hardware* definitivo del sensor inteligente con el programa gratuito CircuitMaker. El tamaño es más reducido que en el caso del prototipo inicial: 25x51mm. Estas dimensiones son ligeramente superiores a las de la propia LoPy4: 20x55mm.

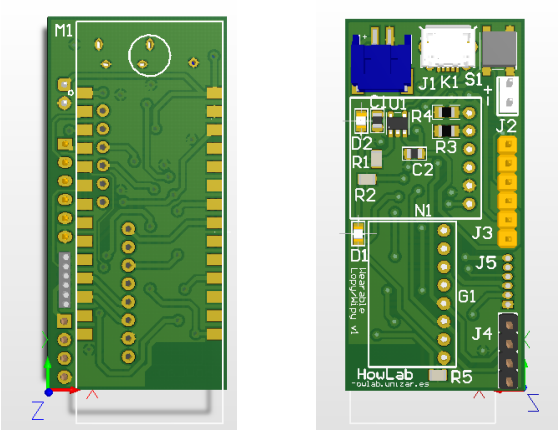

Figura 12. Imagen de la cara TOP (izq) y BOTTOM (dcha) de la PCB

En este Trabajo Fin de Máster las comunicaciones se hacen a través de WiFi, por lo que la tecnología LoRa de la LoPy4 no se utiliza. Por esta razón, la PCB diseñada permite colocar indiferentemente un dispositivo LoPy4 o uno WiPy [12] -del mismo fabricante-, que tan sólo permite comunicaciones WiFi y BLE, las mismas que el propio ESP32 interno. Debido a su menor coste, en este TFM se han comprado dispositivos WiPy para el montaje de los prototipos destinados a las pruebas iniciales.

La antena de ambos dispositivos está localizada en la parte inferior derecha. Por esta razón la PCB tiene una longitud inferior y un recorte en el plano de masa, para evitar que cualquiera que sea el dispositivo utilizado la radiación de la antena no se vea afectada.

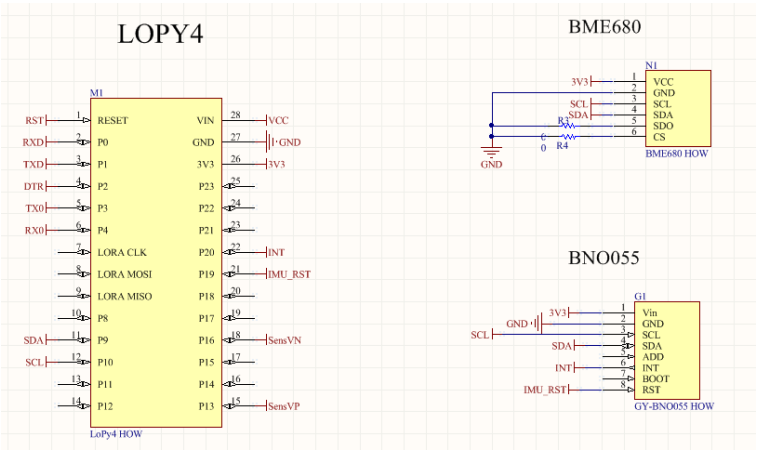

Figura 13. Esquemático I del prototipo definitivo

Además del módulo sensor IMU BNO055 seleccionado, y de la propia LoPy4/WiPy con el *software* embebido, se ha habilitado el ruteo para la posible conexión en un futuro del sensor BME680 [13], que permite medir temperatura, humedad, presión y gas.

Una vez obtenidos los datos de movimiento del animal, el sensor BME680 permitiría sacar conclusiones interesantes acerca de cómo afectan factores como la temperatura y la presión -por tanto también la altitud- al comportamiento del animal.

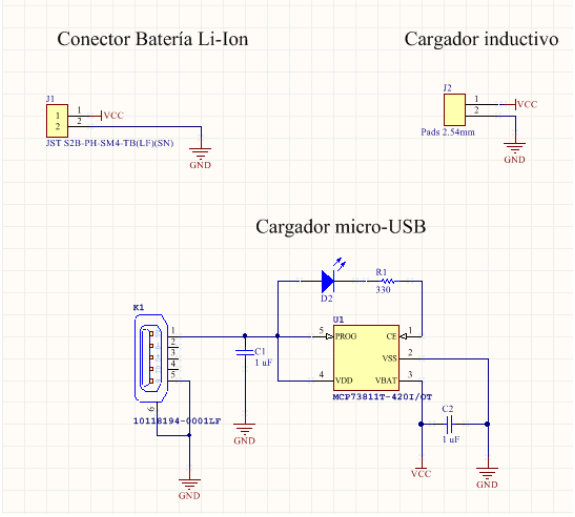

Figura 14. Esquemático II del prototipo definitivo

La alimentación del *hardware* es de nuevo a partir de una batería de Li-Ion, por lo que se ha habilitado un conector en la propia PCB. En un *wearable* alimentado con batería, es fundamental permitir su recarga y, a poder ser, de forma sencilla para el usuario. Existen dos formas distintas de cargar la batería en este caso:

1. A través de un cable micro-USB.
2. Con un cargador inductivo. Dicho cargador acaba en dos cables, por lo que se han dejado dos vías en la placa para soldarlos si fuera necesario. Para este TFM se han dejado sin conectar, puesto que el método de carga utilizado será mediante micro-USB.

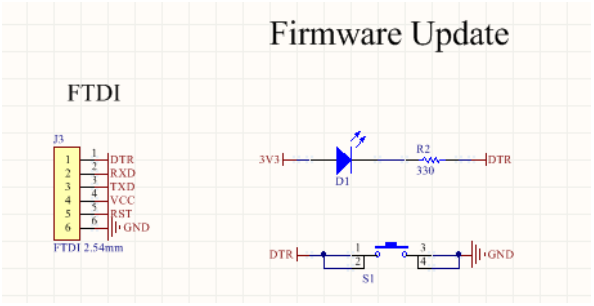

Figura 15. Esquemático III del prototipo definitivo

Dado que en este prototipo no se dispone de la placa de expansión, no es posible cargar programas en la LoPy4 mediante micro-USB. Por ello este procedimiento debe realizarse a través de FTDI.

Como se explica en el *Anexo III*, para programar la LoPy4 en lenguaje C y ensamblador, es necesario ponerla en modo *Firmware Update* llevando el pin P2 a masa durante un reset. El conversor USB-serie utilizado presenta las salidas RTS y DTR, que son configurables en alto o bajo vía *software*. De este modo, si se conectan DTR con P2, y RTS con el pin de reset de la LoPy4, en el proceso de grabación de *firmware* se podrá prescindir de interruptores.

De todas formas, dado que quedaba espacio en la PCB, se ha colocado un pulsador a modo de interruptor para poder llevar la LoPy4 a modo *Firmware Update* de forma manual.

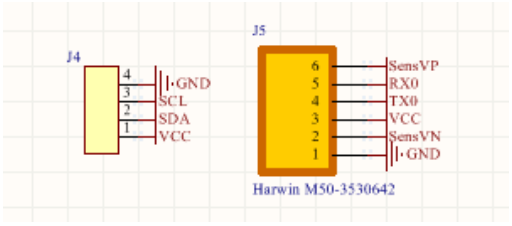

Figura 16. Esquemático IV del prototipo definitivo

Por último, la PCB tiene varios pines para añadir ciertas funcionalidades que, aunque no se han considerado en este TFM, pueden ser utilizadas en un futuro. Por ejemplo, los pines SensVP y SensVN corresponden a una tensión diferencial procedente del sensor Hall interno del ESP32, que permite medir campos magnéticos. Por otra parte, también es posible conectar sensores adicionales por I2C (SDA y SCL) o UART (RX0 y TX0).

A continuación, se muestran imágenes del prototipo una vez soldado y listo para efectuar las pruebas en animales. La caja en la que irá colocado es de dimensiones 80x75x20 mm, y el peso de todo el sistema es de aproximadamente 140 gramos.

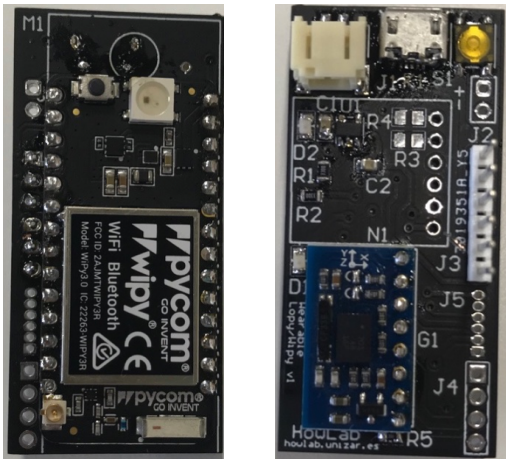

Figura 17. Imagen real de la PCB del prototipo definitivo del sensor inteligente

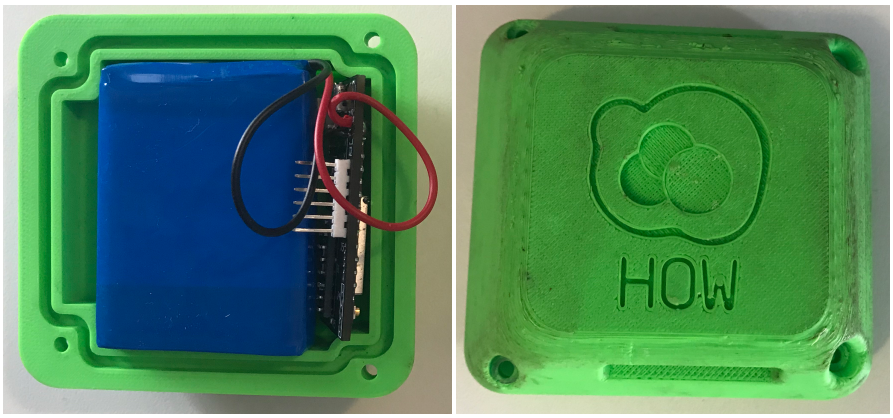

Figura 18. Imagen real del montaje del prototipo definitivo

3. Desarrollo software del sensor inteligente

En esta sección se introduce el problema del elevado consumo energético resultante de la programación con Micropython de la LoPy4. La solución propuesta consiste en programar directamente -en lenguaje C y ensamblador- el microcontrolador interno de la LoPy4: el ESP32. De este modo, se consigue una reducción del consumo de corriente del microcontrolador durante la lectura del sensor IMU de un 91% aproximadamente.

Posteriormente, se muestra por partes el diagrama de flujo del algoritmo de monitorización desarrollado y el proceso de configuración implementado en el sensor inteligente para el momento de su despliegue.

3.1. Aproximación inicial: el problema del consumo energético

El fabricante del dispositivo IoT con comunicaciones embebidas LoPy4 propone la programación en Micropython con el entorno de desarrollo Visual Studio Code, previa instalación del *plugin* Pymakr. Micropython [14] es una implementación de Python optimizada para ser utilizada en microcontroladores y en redes de comunicaciones con recursos limitados.

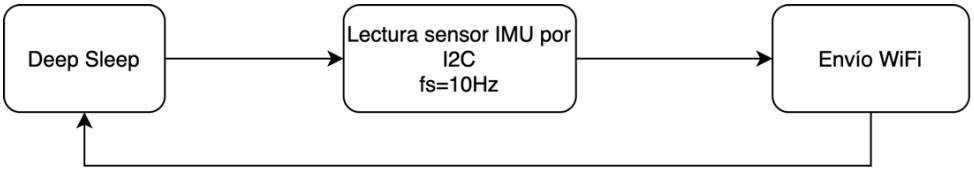

Figura 19. Algoritmo propuesto para evaluación inicial del consumo energético

Inicialmente, el objetivo ha sido evaluar el consumo de energía de la LoPy4 en condiciones similares a las que se trabajará en la aplicación final. Se propone una división de la actividad del microcontrolador en tres estados, para caracterizar el consumo en cada una de ellos: (1) dormir, (2) lectura del sensor por I2C y (3) envío por WiFi de las medidas realizadas. Para dar una idea del peso del consumo del propio sensor IMU BNO055 en la actividad del *wearable* se muestran los dos modos de operación para los cuales la diferencia es más notable: sólo acelerómetro y 9-DOF.

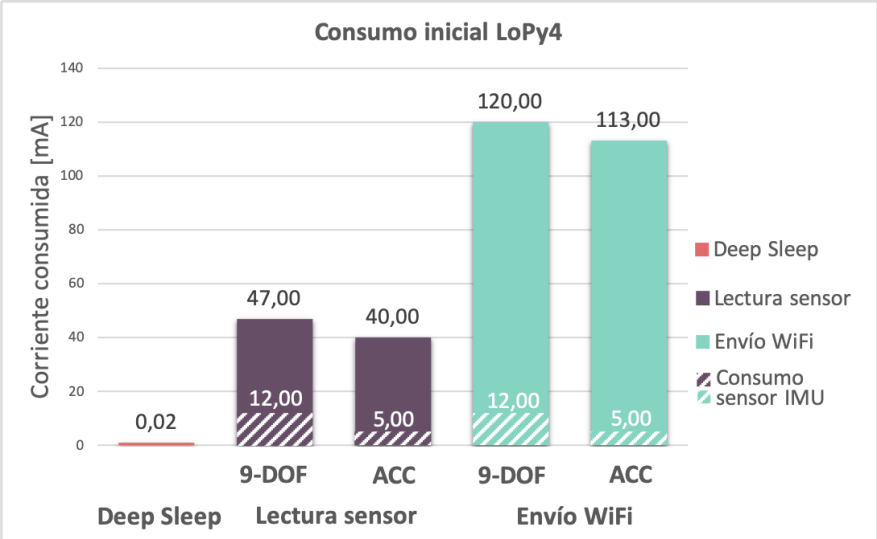

Figura 20. Consumo de corriente medido inicialmente en la LoPy4. 2 modos de operación del sensor IMU

A pesar de que el consumo durante el envío WiFi es elevado, puede llegar a ser tolerable. En una aplicación típica de IoT se tratará de que el sensor inteligente esté el menor tiempo posible en este estado, jugando aquí un papel importante la capacidad de almacenamiento del microcontrolador seleccionado. De este modo, en el cómputo global del consumo de energía no tendrá tanto peso.

Sin embargo, los consumos obtenidos implican fundamentalmente dos problemas a la hora de utilizar este microcontrolador en una aplicación como la que se propone en este Trabajo Fin de Máster.

1. En primer lugar, la monitorización debe ser constante, de tal forma que se minimice la pérdida de información de movimiento del animal. En este caso, por mucho que el consumo en *deep sleep* sea muy reducido (20µA), durante todo el tiempo que el microcontrolador esté en ese estado no se estará leyendo el sensor IMU y toda esa información se habrá perdido.

   Por otra parte, no es posible que el microcontrolador se duerma y se despierte con la frecuencia de muestreo, de 10 veces por segundo. Por lo tanto, en una aplicación de este tipo no se podría dar este estado de *sleep*.

   Este es uno de los desafíos que se debe hacer frente al hacer una aplicación de IoT de monitorización de movimiento. En otros casos, como en medición de temperatura o humedad, el sensor inteligente sí que podría permanecer dormido durante largos periodos de tiempo, para periódicamente despertarse y enviar la información correspondiente.

2. El segundo problema consiste en el alto consumo en el proceso de lectura del sensor. Obviando la corriente consumida por el sensor IMU BNO055 -de 12mA para modo 9-

DOF y de 5mA para modo ACC-, queda un consumo de corriente de 35mA pertenecientes únicamente al microcontrolador. Este valor resulta muy elevado, sobre todo teniendo en cuenta que el microcontrolador debería leer "en todo momento" la información inercial del sensor.

En resumen, a través del desarrollo propuesto por el fabricante del *hardware* que se ha elegido para este proyecto, el consumo de corriente del sensor inteligente supondría un tiempo de vida muy reducido del *wearable*. El sistema cumpliría con su función, sí, pero en una aplicación real como la que se persigue en este Trabajo Fin de Máster, implicaría tener que recargar con mucha frecuencia la batería del sistema. Esto, con total seguridad, no terminaría de satisfacer al cliente.

Para dar una idea aproximada de por qué estos consumos son intolerables en esta aplicación en concreto. Si se utilizara una batería de Litio-Ion con una capacidad de 4000mAh -que ha sido empleada previamente en aplicaciones similares del grupo de investigación HOWLab-, y suponiendo que ni siquiera se produce el envío WiFi de los datos, el tiempo de vida sería:

$$Tiempo\ de\ vida\ [9DOF]\ (h) = \frac{Capacidad\ de\ la\ batería\ (mAh)}{Corriente\ consumida\ 9DOF\ (mA)} = \frac{4000mAh}{47mA} \approx 85\ horas\ (3'5\ días)$$

$$Tiempo\ de\ vida\ [ACC]\ (h) = \frac{Capacidad\ de\ la\ batería\ (mAh)}{Corriente\ consumida\ ACC\ (mA)} = \frac{4000mAh}{40mA} \approx 100\ horas\ (4\ días)$$

3.1.1 Solución propuesta: programación a más bajo nivel

El origen del problema consiste en que el desarrollo en Micropython no permite tener un gran control sobre el *hardware* del microcontrolador. Se trata de un lenguaje de programación interpretado, a muy alto nivel, que encapsula varias capas de programación por debajo, más cercanas al código binario que entienden las CPUs.

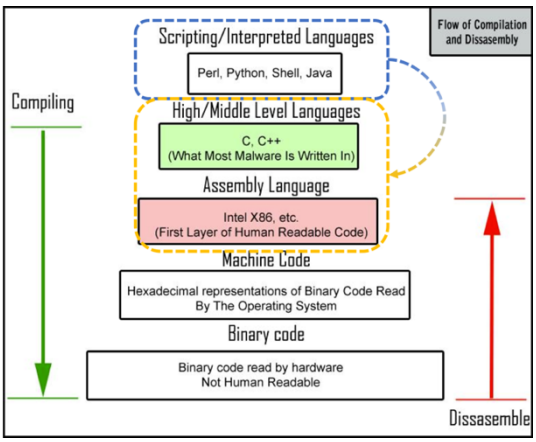

Figura 21. Niveles de los lenguajes de programación

La pregunta es, si se pudiera alcanzar un mayor control sobre el *hardware* mediante el desarrollo con un lenguaje de programación de más bajo nivel, ¿sería posible leer el sensor IMU a la vez que el microcontrolador está "dormido" con el objetivo de reducir el consumo? La respuesta es sí, pero no exactamente con la programación de la LoPy4 tal y como la entiende el fabricante Pycom.

Tras un estudio del *datasheet* de la LoPy4 [15] se ha visto que el microcontrolador contenido en este dispositivo es el ESP32 de Espressif [16]. El ESP32 es un MCU con tecnologías WiFi a 2.4GHz y BLE integradas. Está diseñado para trabajar con muy bajo consumo energético en aplicaciones de *wearables* e IoT. Además del ESP32, la LoPy4 alberga dos chips que permiten las comunicaciones LoRa y Sigfox, comunicados con el MCU principal a través de SPI.

La programación del ESP32 se realiza con el *toolchain* ESP-IDF y el IDE de Eclipse para programación en C/C++. La instalación del *toolchain* y la configuración del proyecto están explicadas en detalle en el *Anexo II*.

La CPU del ESP32 es un microprocesador de doble núcleo Xtensa LX6, de 32 bits. Sin embargo, en el SoC existe un coprocesador *Ultra Low Power* ULP capaz de realizar medidas utilizando el ADC, el sensor de temperatura y sensores I2C externos, mientras los procesadores principales están dormidos. Esto sería idóneo para este proyecto: el ULP lee el sensor IMU a través de I2C y almacena las medidas, al mismo tiempo que el resto del *hardware* está durmiendo. Llegados a cierto punto, la CPU principal se despierta y sube los datos leídos por el ULP a la nube.

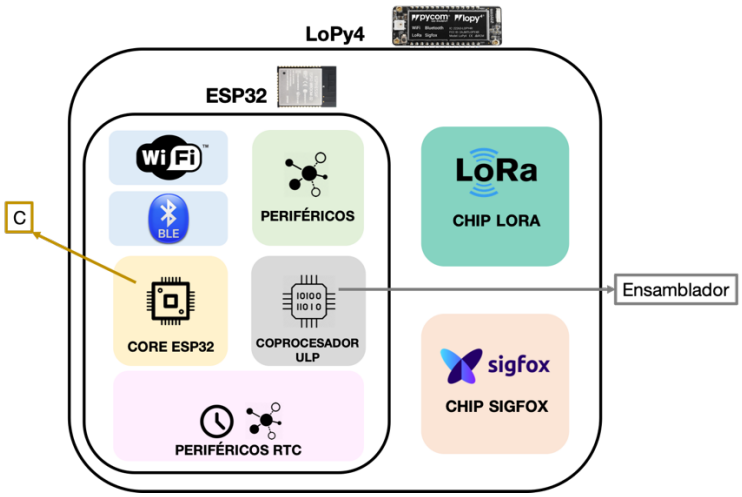

Figura 22. Diagrama de bloques del dispositivo IoT LoPy4

Sin embargo, la programación del coprocesador ULP no es tarea sencilla, puesto que se debe hacer en lenguaje ensamblador propio de Espressif [17]. Además, el ULP presenta

ciertas limitaciones, entre otras: (1) sólo puede acceder a la región de memoria *RTC_SLOW_MEM* de 4KB y (2) los GPIOs utilizados deben ser del tipo RTC -atendiendo al *datasheet* del ESP32-.

De ahora en adelante en este Trabajo Fin de Máster, con el fin de lograr una reducción del consumo energético, se programa directamente el ESP32 contenido en la LoPy4: en lenguaje C para la CPU principal, y en lenguaje ensamblador para el coprocesador ULP.

El objetivo es desarrollar un *firmware* que sea igualmente válido para el caso de tener un dispositivo LoPy4, como si se tuviera únicamente un ESP32 y se prescindiera de la funcionalidad LoRa y Sigfox. Para ello, es necesario utilizar pines que sean compartidos por ambos dispositivos.

Por último, para acceder al ESP32 contenido en la LoPy4 es necesario eliminar todas las librerías de Micropython contenidas en el dispositivo. Todo este proceso no viene en la documentación de Pycom, dado que el fabricante no recomienda hacerlo. En el *Anexo III* se explica el procedimiento para (1) borrar Micropython, (2) cargar programas escritos en C y ensamblador en el ESP32 de la LoPy4 y (3) restaurar la LoPy4 a su estado original.

3.2. Lectura del sensor mediante el coprocesador ULP

En una aplicación IoT de monitorización de movimiento, la tarea que el sensor inteligente realiza la mayor parte del tiempo es la de lectura del propio sensor. Esto es siempre así, independientemente de la tecnología para envío de datos, el algoritmo, etc. Idealmente, se debería leer el sensor en todo momento. Por ello, minimizar el consumo de esta tarea al máximo es fundamental para el éxito del *wearable*, en términos de ciclo de vida.

En este Trabajo Fin de Máster se consigue realizar el procedimiento de lectura del sensor IMU mientras el procesador principal está dormido, y la mayor parte de la memoria RAM y los periféricos digitales apagados. Esto es posible gracias al coprocesador ULP, que será el encargado de leer el sensor. Al estar la RAM apagada, ya se puede preveer que los datos leídos por el ULP deberán guardarse en otra región de memoria, como se explicará más adelante en este capítulo.

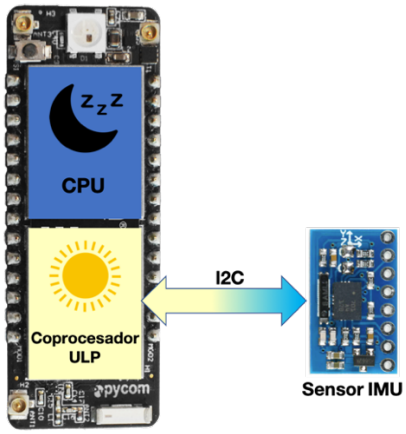

Figura 23. Lectura del sensor IMU por parte del ULP mientras la CPU duerme

3.2.1. Comunicación ULP – Sensor IMU: I2C bit-bang

El coprocesador ULP tiene accesibles el ADC y el bus de comunicaciones I2C, que no se dormirán junto con el resto del sensor inteligente. El bus maestro-esclavo I2C es un protocolo de comunicaciones bidireccional *half-duplex*, síncrono y serie, cuya interfaz consta únicamente de dos líneas: (1) SCL para el reloj y (2) SDA para los datos. Además, permite la existencia de múltiples esclavos -sensores para lectura- compartiendo bus con el maestro -coprocesador ULP- [18].

El ESP32 -y por tanto la LoPy4- tiene dos puertos o controladores exclusivamente para manejar las comunicaciones I2C con los sensores. La ventaja de tener *hardware* dedicado en microcontroladores radica en que las especificaciones de tiempos en las comunicaciones I2C se cumplirán sin realizar ningún paso adicional.

Sin embargo, se ha comprobado -como también indicaban varios desarrolladores con ESP32- que estos puertos I2C no permiten realizar algunas tareas correctamente cuando el procesador principal está dormido, es decir, cuando el que hace la lectura es el ULP. Un ejemplo es el *clock stretching*, mediante el cual el esclavo toma control sobre la línea de reloj transmitida por el maestro y la mantiene en bajo, cuando debe esperar para transmitir datos al maestro [19].

Por ello, en este Trabajo Fin de Máster se ha realizado una versión *software* en lenguaje ensamblador -propio de Espressif- de las comunicaciones I2C, mediante el control y lectura de los estados de los GPIOs directamente por parte del ULP. Esta técnica se conoce como *bit-banging* [18].

Como consecuencia, debe asegurarse el cumplimiento de las especificaciones de tiempos del sensor IMU seleccionado. Aunque como ventaja, se van a poder realizar lecturas de 16 bits del sensor, mientras que el *hardware* dedicado en el ESP32 tan sólo podía leer cada 8 bits. Esto es muy interesante al trabajar con el BNO055, puesto que las medidas realizadas por el sensor son de 16 bits, y por tanto una lectura por medida sería suficiente.

La técnica utilizada para la comunicación I2C es la de "*repeated start condition*", indicada por el fabricante Bosch Sensortec en la hoja de datos del sensor IMU [20]. Se ha tomado como ejemplo el algoritmo desarrollado en pseudo-C en [21] para la implementación en este proyecto.

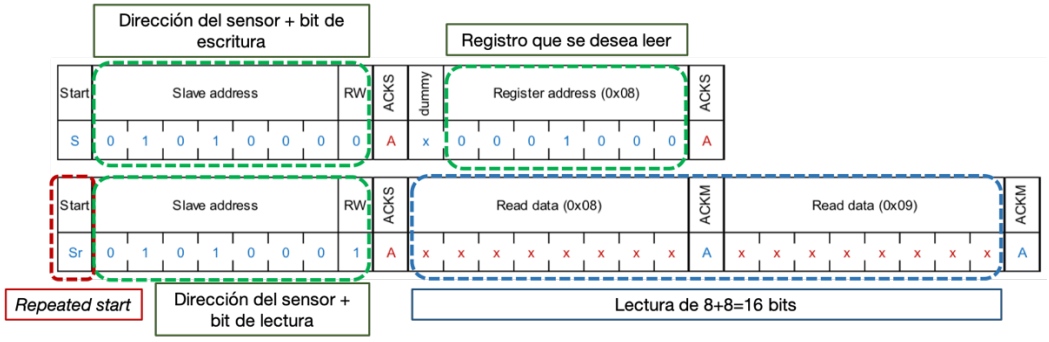

Figura 24. Método "repeated start condition" para comunicación I2C

En este método, para que el maestro pueda realizar una lectura del sensor, debe: (1) enviar señal de *start*, (2) escribir en el bus 1 *Byte* indicando la dirección del sensor, (3) escribir en el bus 1 *Byte* indicando el registro del sensor que desea leer, (4) enviar otra señal de *start*, (5) enviar de nuevo la dirección del sensor con un bit de lectura. Una vez realizado este proceso el maestro libera el bus y lee los 16 bits del sensor.

Este procedimiento de lectura se realiza para todos los registros del sensor que se deseen leer (acelerómetro eje X, giróscopo eje Y…), estando cada uno de ellos en una dirección diferente de memoria, indicada en la hoja de datos del sensor BNO055.

3.2.2. Mejora de consumo energético en la lectura del sensor IMU

El consumo de corriente del *wearable* durante la lectura del sensor IMU con esta solución se ve ampliamente reducido con respecto a la aproximación de aplicación inicial que se desarrolló en Micropython. De nuevo, se muestran los consumos asociados al sensor IMU para aquellos modos de operación (9-DOF y ACC) donde esta diferencia es más acusada.

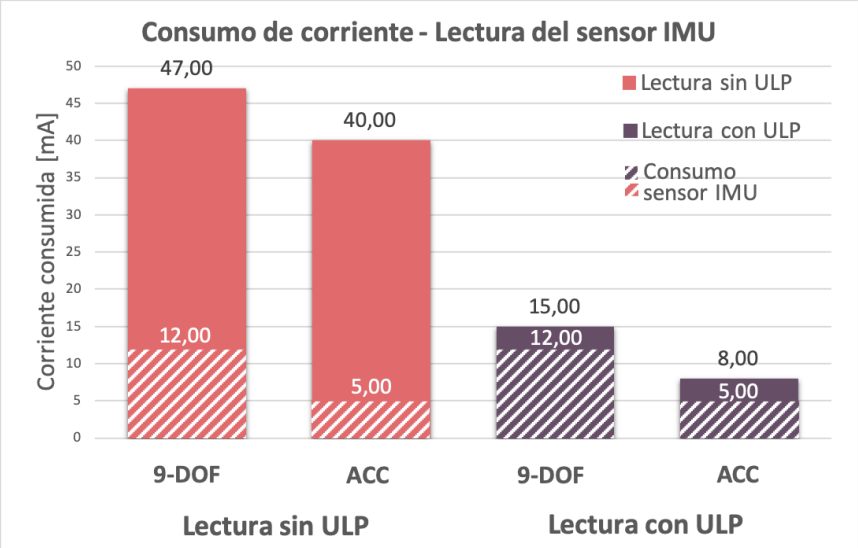

Figura 25. Mejora en el consumo de corriente del wearable en la lectura del sensor IMU al leer con ULP

En este caso el consumo de corriente del microcontrolador pasa de 35mA (caso inicial sin ULP) a únicamente 3mA (solución propuesta con ULP). Es decir, el consumo se ve reducido aproximadamente un 91% en el proceso de lectura del sensor IMU con la solución propuesta. Este resultado permitirá alargar considerablemente el tiempo de vida del *wearable*.

En cuanto al consumo del propio sensor IMU, puede intuirse la importancia que tendrá en fases posteriores de diseño del sensor inteligente una elección precisa de la información de movimiento necesaria, puesto que con la solución propuesta sería posible reducir el consumo prácticamente un 50% adicional. Si aún así este consumo de corriente asociado al sensor IMU fuera demasiado alto para algún tipo de aplicación, debería optarse por escoger otro sensor IMU que, posiblemente, diera una menor calidad de dato que el BNO055 elegido en este caso.

En el apartado 3.1. *Aproximación inicial: el problema del consumo energético*, se ha dado una idea muy aproximada del tiempo de vida del sensor inteligente mediante el desarrollo con Micropython, suponiendo que el único estado de operación es aquel en el que el microcontrolador pasa la mayor parte del tiempo: la lectura del sensor IMU. En esa primera aproximación el tiempo de vida estimado era de unos 3'5 ó 4 días, en función del modo de operación del sensor IMU seleccionado. Sin embargo, realizando la lectura con el ULP este tiempo aumenta:

$$Tiempo\ de\ vida\ [9DOF]\ (h) = \frac{Capacidad\ de\ la\ batería\ (mAh)}{Corriente\ consumida\ 9DOF\ (mA)} = \frac{4000mAh}{15mA} \approx 266\ horas\ (11\ días)$$

$$Tiempo\ de\ vida\ [ACC]\ (h) = \frac{Capacidad\ de\ la\ batería\ (mAh)}{Corriente\ consumida\ ACC\ (mA)} = \frac{4000mAh}{8mA} \approx 500\ horas\ (21\ días)$$

3.3. Algoritmo de monitorización

3.3.1. Condicionantes del hardware en la programación

Antes de mostrar el algoritmo que se ejecutará en el sensor inteligente para la monitorización de los animales es conveniente conocer algunas limitaciones presentes en el *hardware* del dispositivo LoPy4/ESP32, que afectan directamente a la programación y al desarrollo del algoritmo.

Memoria y almacenamiento

El almacenamiento es, junto con la batería, el principal obstáculo al que se debe hacer frente en una aplicación IoT con *wearables*. En un escenario de monitorización constante, la cantidad de datos medidos aumenta a gran velocidad, por ello la subida de datos a la nube no puede darse cada mucho tiempo.

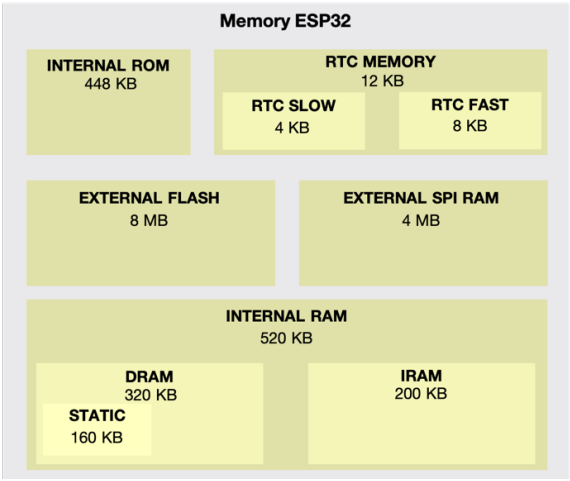

Figura 26. Regiones de memoria del ESP32

Atendiendo a la información detallada de la hoja de datos del microcontrolador se observan diferencias notables en la memoria disponible, respecto a lo que podría parecer a priori. Por ejemplo, si se desea almacenar las medidas del sensor IMU en la memoria RAM, sólo hay 160 KB disponibles debido a "una limitación técnica" [16], a diferencia de los 520 KB que se mencionan en el marketing del dispositivo.

Por otro lado, la memoria *external SPI RAM* no está habilitada por defecto y por tanto es necesario configurarla. Además, presenta ciertas restricciones en su uso, que implican alguna diferencia respecto al almacenamiento en la *internal RAM*. La más notable consiste en que tan sólo se puede utilizar mediante asignaciones dinámicas durante tiempo de ejecución, y no estáticas.

Adicionalmente, no todas las regiones de memoria son accesibles por todos los núcleos del ESP32, y no todas preservan el estado del mismo modo en función del modo de *sleep*, como se irá viendo más adelante en esta sección.

Procesadores

La CPU del ESP32 es un microprocesador de doble núcleo Xtensa LX6, de 32 bits. Ambos núcleos presentan algunas diferencias, por ello en la documentación son referidos con diferentes nombres: Core 0 (*Protocol CPU*) y Core 1 (*Application CPU*). A estos dos núcleos se añade el ya mencionado coprocesador ULP.

Procesador	**Lenguaje de programación**	**Frecuencia de reloj**	**Regiones de memoria accesibles**			
			RTC SLOW	**RTC FAST**	***External* SPI RAM**	***Internal* RAM**
			4 KB	*8 KB*	*4 MB*	*520 KB*
Core 0	C	160MHz	✓	✓	✓	✓
Core 1	C	160MHz			✓	✓
ULP	Ensamblador	8MHz	✓			

Tabla 2. Comparación de los tres procesadores del ESP32: Core 0, Core 1 y coprocesador ULP

Existe un elevado número de diferencias entre los dos procesadores "reales" y el coprocesador. Las más significativas son posiblemente el lenguaje de programación y la frecuencia de reloj, de 8MHz frente a 160MHz.

El ULP tan sólo puede acceder a la región de memoria *RTC SLOW*, por lo que ahí es donde guarda todas las medidas. Sin embargo, desde el código principal se debe fijar que el Core 0 sea el que lea las variables guardadas por el ULP en *RTC SLOW*, puesto que el Core 1 no tiene acceso a esa región. Del mismo modo, si se desean guardar variables en la región *RTC FAST* -que no se borra en *deep sleep*-, se deberá utilizar el Core 0.

Modos de sleep

Este microcontrolador permite dos posibilidades a la hora de dormir: *light sleep* y *deep sleep*.

- Dormir en *deep sleep* permite alcanzar consumos más bajos -de aproximadamente la mitad respecto a *light sleep*-, como se mostrará en el apartado *Resultados experimentales*.
- En *deep sleep* tan sólo permanecen despiertos los periféricos RTC, el ULP y la región de memoria RTC -*slow* y *fast*-. Por otra parte, en *light sleep* los periféricos digitales,

y parte de la RAM y la CPU también estarán despiertos *clock-gated*, una técnica utilizada para reducir el consumo energético.

- Al despertarse de *light sleep* el ESP32 preserva su estado, hay retención de RAM. Sin embargo, despertarse de *deep sleep* es a efectos prácticos similar a un reset, donde todo lo que no estuviera guardado en la región RTC de memoria, se borrará.

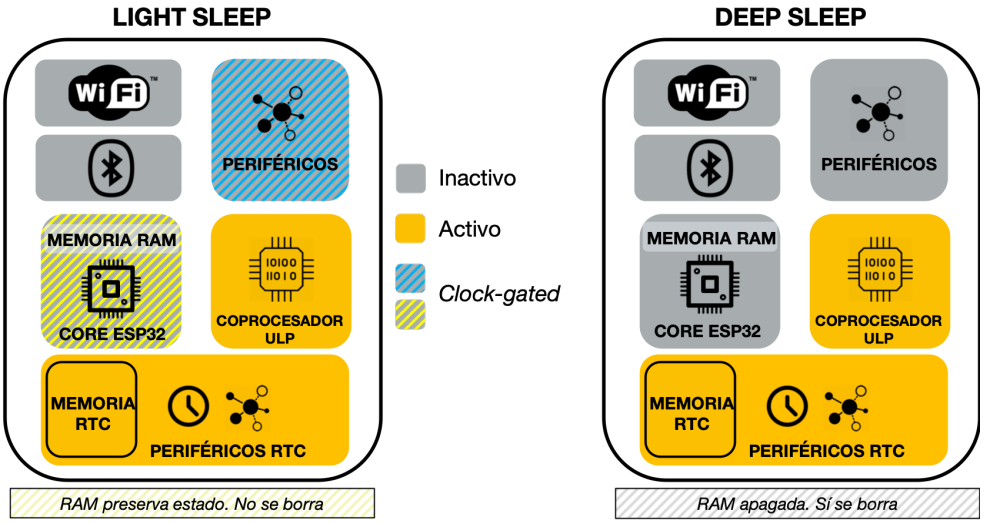

Figura 27. Comparación light sleep (izq) vs deep sleep (dcha)

3.3.2. Algoritmo propuesto

El proceso de diseño de un sensor inteligente tiene varias fases. En una primera aproximación -donde se sitúa este TFM-, se debe enviar la mayor cantidad de información posible, aunque haya redundancia. En esta aplicación, esto implica: (1) la lectura de todas las variables inerciales del sensor IMU, y (2) una frecuencia de muestreo elevada, de 10 Hz -la lectura cada 100 ms del movimiento de ovejas es más que suficiente-.

La idea en esta primera fase es tener información lo más exacta posible del movimiento del animal, minimizando al máximo la pérdida de datos potencialmente relevantes. Con un procesado adecuado de esos datos "completos", ya sería posible determinar qué variables inerciales son imprescindibles para identificar diferentes actividades del animal, con qué frecuencia de muestreo, etc.

En un segundo paso, el sensor inteligente mandaría tan sólo la información suficiente y necesaria para poder identificar la actividad del animal. Finalmente, mediante aprendizaje automático en el propio *wearable*, él mismo sería capaz de enviar únicamente si el comportamiento de un animal es anómalo en un cierto intervalo de tiempo.

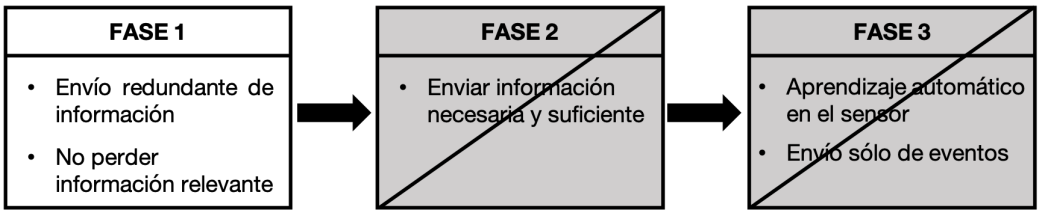

Figura 28. Fases del proceso de diseño de un sensor inteligente

En este Trabajo Fin de Máster se ha desarrollado un algoritmo para monitorización constante de acelerómetro, giróscopo, magnetómetro y cuaternios a una frecuencia de muestreo del sensor de 10 Hz. Los datos se guardarán en el sensor inteligente y, cuando el almacenamiento esté lleno, se subirán a la nube mediante tecnología WiFi, utilizando el protocolo de aplicación de IoT MQTT.

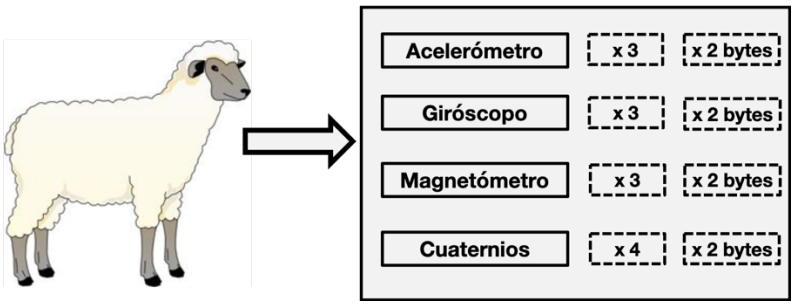

Figura 29. Variables inerciales del animal medidas por el sensor inteligente

A continuación, se muestra el diagrama de flujo del algoritmo propuesto, junto con las regiones de memoria que intervienen en el almacenamiento de los datos. La explicación del algoritmo se divide en tres partes, correspondientes a: (1) lectura del sensor IMU por parte del ULP, (2) almacenamiento en memoria de las medidas del ULP, y (3) envío de los datos.

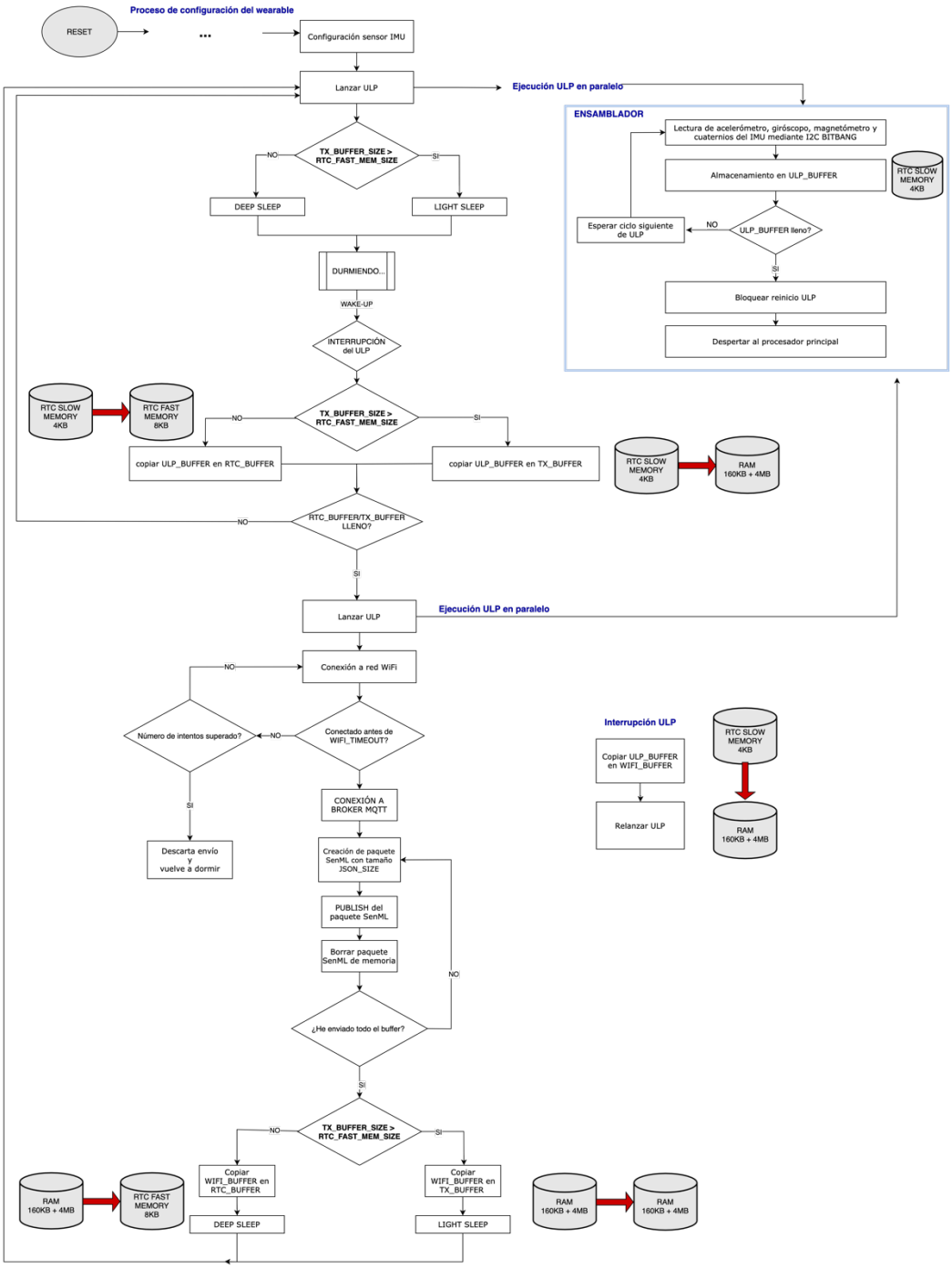

Figura 30. Diagrama de flujo completo del algoritmo de monitorización propuesto

Lectura del sensor IMU por parte del ULP

El coprocesador ULP es el encargado de leer el sensor cada 100 ms. Las tareas del ULP pueden verse como un hilo paralelo que se lanza desde el código principal. La rutina de este hilo del ULP -en lenguaje ensamblador- consiste en leer una vez los registros del

sensor IMU. Cuando termina la rutina, espera un tiempo -que en este caso coincide con el periodo de muestreo- y vuelve a empezar.

Todas las medidas se guardan en la región de memoria *RTC SLOW*, de 4KB. Cuando esta región está llena, el ULP despierta al procesador principal y éste copia los datos leídos en un *buffer* de la memoria RAM o de la memoria RTC, como se verá a continuación.

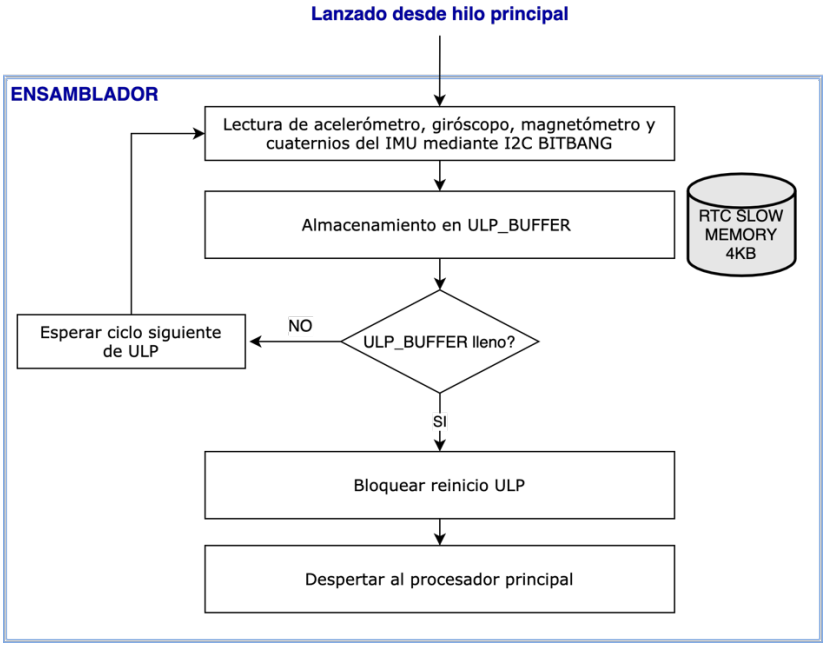

Figura 31. Diagrama de flujo I: lectura del sensor IMU por parte del ULP

Almacenamiento en memoria de las medidas del ULP

Cuando se resetea el sensor inteligente comienza un proceso de configuración -explicado en más detalle en la siguiente sección-, donde el usuario puede decidir entre otras cosas la cantidad de *bytes* de datos que el sensor inteligente envía cada ciclo de operación.

A partir de los condicionantes *hardware* desarrollados en la sección anterior, se abren dos posibilidades respecto al modo de trabajo del *wearable* en función del número de *bytes* a enviar:

1. Si la cantidad de *bytes* a enviar cabe en la región de memoria *RTC FAST* -de 8KB-, entonces el microcontrolador trabajará en *deep sleep*, puesto que esa zona de almacenamiento permite mantener los datos en este modo de *sleep*.
2. Si la cantidad de *bytes* a enviar excede estos 8KB de *RTC FAST*, entonces el microcontrolador dormirá en modo *light sleep*, y guardará los datos almacenados en la memoria RAM -tanto la interna de 160KB como la externa por SPI de 4MB-.

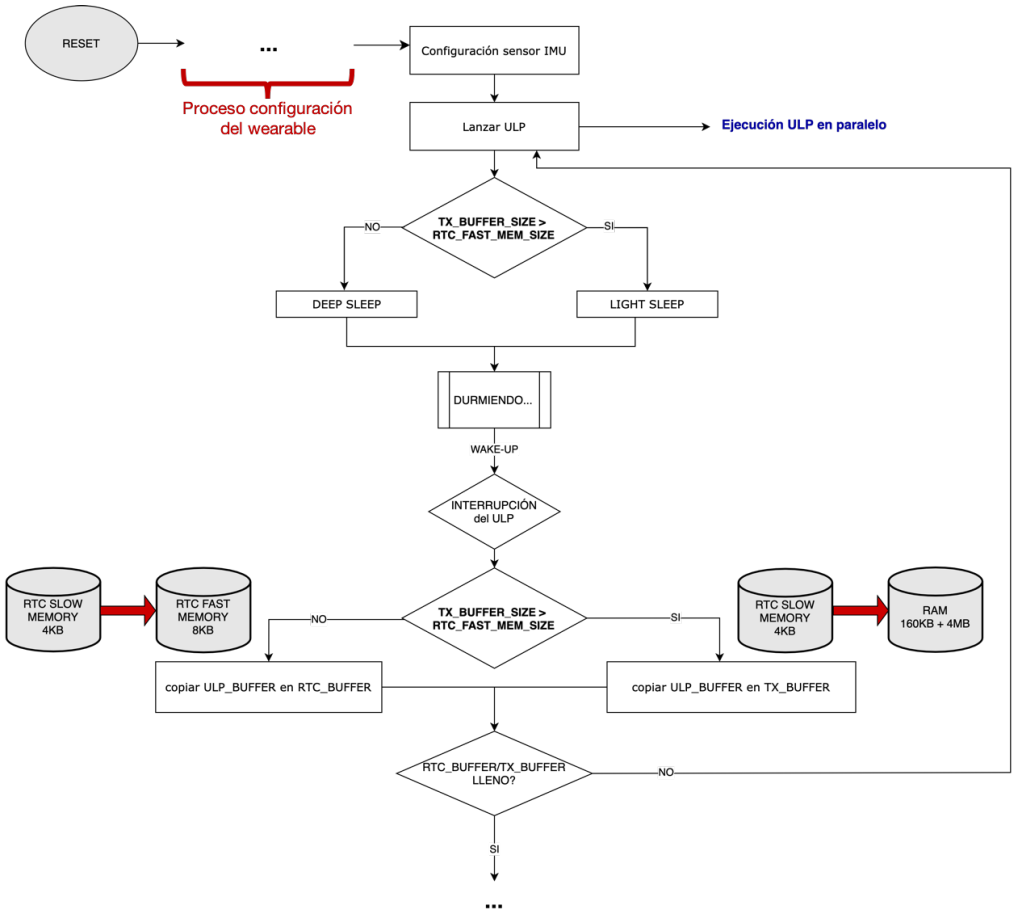

Figura 32. Diagrama de flujo II: Hilo principal copia medidas del ULP en la memoria correspondiente

Cuando el ULP despierta al procesador principal, se copia el contenido de las medidas del ULP -memoria *RTC SLOW*, por tanto con Core 0- en la región *RTC FAST* (*deep sleep*) o en la RAM (*light sleep*). Si no se han alcanzado los *bytes* de datos establecidos en la configuración, el *wearable* volverá a dormirse y a lanzar el ULP. Este proceso durará hasta que se llene el *buffer* de envío.

Envío de los datos

Una vez lleno el *buffer* de envío, el sensor inteligente se conecta a la red WiFi -cuyas credenciales se han guardado en memoria no volátil durante la configuración- y prepara los datos en el formato adecuado para su envío al *broker* MQTT. Estos datos se guardarán en una base de datos de series temporales en la nube. Cuando ya se han subido todos los datos, vuelve a lanzarse el ULP y a empezar el proceso de lectura.

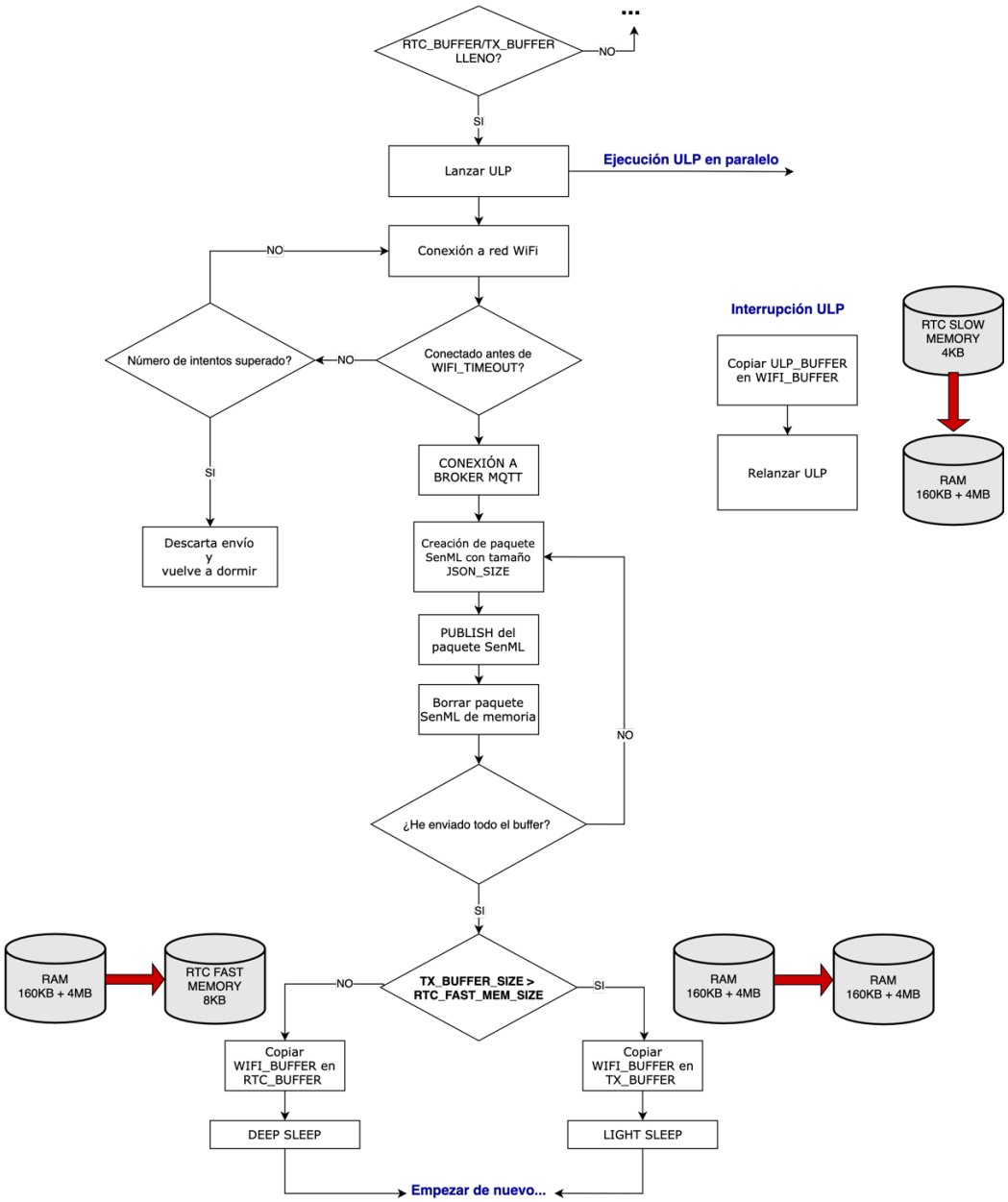

Figura 33. Diagrama de flujo III: Conexión a red WiFi y subida de los datos a la nube por MQTT

En un primer desarrollo del algoritmo de monitorización, una vez lleno el *buffer* de envío no se seguía leyendo el sensor IMU hasta que el microcontrolador no volviera a dormirse. Es decir, durante todo el proceso de conexión a la red WiFi y envío de los datos había un agujero en la información de movimiento del animal.

Para evitar perder información de movimiento relevante durante ese tiempo, se vuelve a lanzar el ULP para que siga leyendo el sensor IMU en paralelo a todo este proceso. Estos datos se guardarán para su envío en la siguiente ronda.

3.4. Configuración del sensor inteligente

En este Trabajo Fin de Máster se propone la utilización del sensor inteligente para la monitorización de las ovejas localizadas en la Facultad de Veterinaria. Sin embargo, es posible que el mismo sensor pueda ser implementado a corto plazo en otras aplicaciones de monitorización de variables inerciales.

En escenarios diferentes, el cliente podría estar interesado en unos parámetros de funcionamiento distintos, como por ejemplo una frecuencia de muestreo menor, si los animales monitorizados se movieran con mayor lentitud. Además, la red WiFi a la que se conecta el sensor inteligente sería distinta para cada aplicación, con un SSID y contraseña que habría que indicar en cada caso.

El objetivo consiste en que para cada aplicación de monitorización no haya que pasar por el proceso de instalación del entorno de desarrollo, configuración del proyecto y modificación del código fuente. El *firmware* embebido en el sensor inteligente debe ser siempre el mismo.

Por ello, se propone un sistema de configuración del sensor inteligente en el mismo momento de despliegue del *wearable*. Idealmente, este proceso debería ser sencillo de llevar a cabo e incluso realizable por el mismo cliente que desea utilizar el sensor inteligente en su sistema de monitorización. Se propone una división de este procedimiento en dos fases:

- Una primera fase, imprescindible en cualquier caso, correspondiente al aprovisionamiento de las credenciales WiFi del escenario en cuestión.
- Una segunda, relativa a los parámetros de funcionamiento de la aplicación, como son la frecuencia de muestreo del sensor o el número de variables inerciales medidas. Esta fase sería opcional, puesto que en el caso de no darse se tomaría la configuración por defecto.

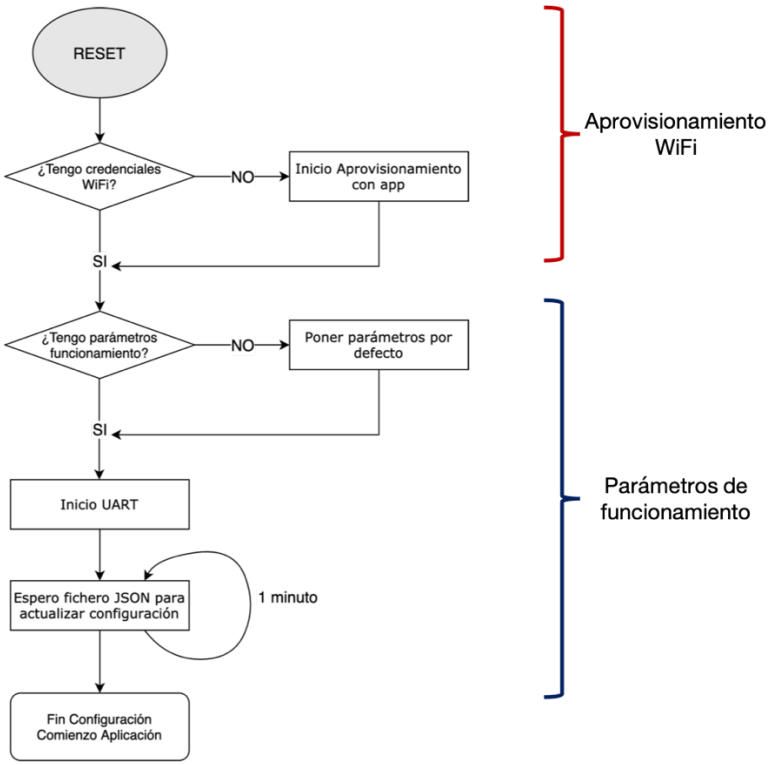

Figura 34. Diagrama de flujo IV: Proceso de configuración del sensor inteligente

Por supuesto, en el caso de efectuarse un reinicio del dispositivo esta configuración no debería borrarse. Por esta razón se ha hecho una partición de la memoria *flash* del sensor, destinada a almacenamiento no volátil. Para borrar la configuración se debe realizar un reinicio del sensor a nivel de fábrica. En el prototipo diseñado se ha habilitado un botón para que efectúe esta tarea cuando permanezca pulsado más de tres segundos.

Aprovisionamiento de WiFi

El envío de las credenciales WiFi, i.e. SSID y contraseña de la red, se realiza a través de la *app* móvil *ESP SoftAP Provisioning*, del fabricante Espressif. La *app* es gratuita y está disponible para Android [22] e iOS [23].

El aprovisionamiento se activará sólo en el caso de que el sensor inteligente no tenga ya unas credenciales almacenadas en la región de memoria no volátil. En el caso de que sí las tenga y se deseen cambiar por algún motivo, será necesario efectuar un reinicio a nivel de fábrica para poder volver a realizar el aprovisionamiento.

El procedimiento de envío de credenciales a través de la *app* es muy sencillo:

- El sensor inteligente arranca como punto de acceso AP.

- El usuario se conecta con la *app* móvil a la red del sensor, de nombre *PROV_XXXXXX*. Este nombre es único para cada dispositivo, puesto que se obtiene a partir de la dirección MAC del mismo.
- Una vez conectado, el usuario accede al apartado *Provision* de la *app*, donde hace un escaneo de las redes WiFi a su alcance.
- El usuario selecciona la red deseada e introduce la contraseña.
- El sensor inteligente al recibir las credenciales pasa a modo estación STA y trata de conectarse a la red WiFi.
- Si las credenciales son correctas y el sensor ha conseguido conectarse se verá un mensaje de éxito en la *app*. En caso contrario, el mensaje en la *app* será de error.

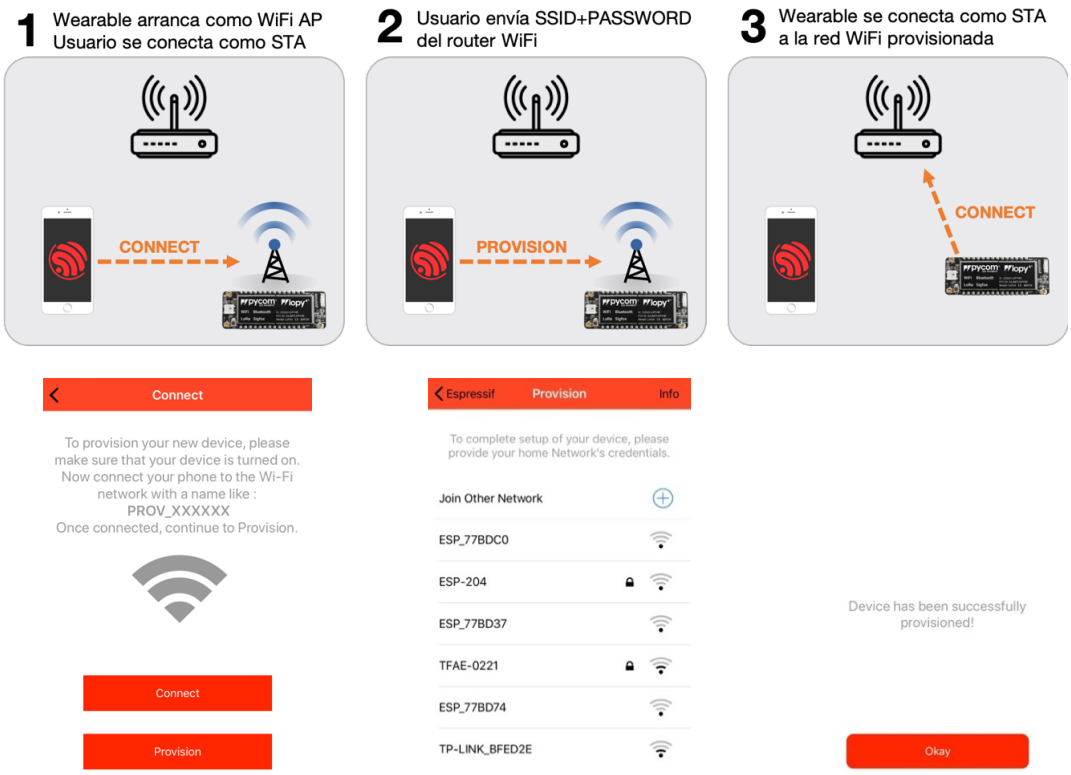

Figura 35. Aprovisionamiento WiFi del sensor mediante app móvil

Configuración de parámetros de funcionamiento

Una vez el sensor inteligente tiene unas credenciales WiFi correctas, pasa a la configuración de los parámetros de funcionamiento. Este proceso es más complejo que el anterior, puesto que es necesario transferir estas variables a través del periférico UART. Para ello es necesario disponer en el momento de la configuración de un ordenador y un adaptador de USB a serie.

- Inicialmente, el sensor comprueba si ya hay parámetros guardados en la región de memoria no volátil. Si no los hay, utilizará unos por defecto.
- A continuación, inicializa el periférico UART y espera durante 1 minuto la recepción de comandos. El protocolo de comunicación en este caso consta de dos tipos de mensajes: (1) SET para establecer una configuración y (2) GET para que el sensor envíe la configuración actual.
- Ante la recepción de parámetros de configuración, el sensor los almacena en la región de memoria no volátil para que no se pierdan con un reinicio del dispositivo.
- Una vez pasado este minuto, el proceso de configuración ha terminado y comienza la ejecución de la aplicación de monitorización.

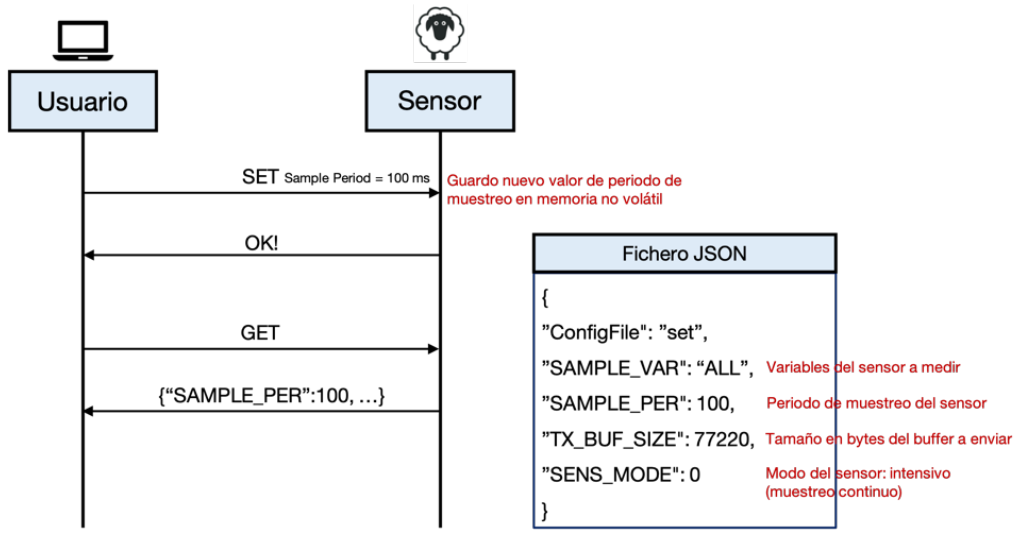

Figura 36. Configuración de los parámetros de funcionamiento (izq) y fichero de configuración tipo (dcha)

4. Envío y análisis de los datos

A lo largo de este apartado se muestra el procedimiento de subida de datos a la nube a través de WiFi y el protocolo de nivel de aplicación MQTT seleccionado. Se introduce también el formato de los datos SenML utilizado y cómo se produce el almacenamiento en una base de datos de series temporales como *InfluxDB*.

Como cierre de esta sección se presenta un análisis de los datos procedentes de un experimento de monitorización en un escenario controlado, evaluando las prestaciones de un ejemplo de pre-procesado y agrupación automática de actividades mediante *machine learning*.

4.1. Subida de datos a la nube – Comunicaciones

Cualquier proceso de comunicación entre dos dispositivos sigue un conjunto de reglas -un protocolo- que regula su comportamiento. Los modelos TCP/IP y OSI definen el conjunto de protocolos que determina la conexión de los equipos a Internet y el envío de datos [24].

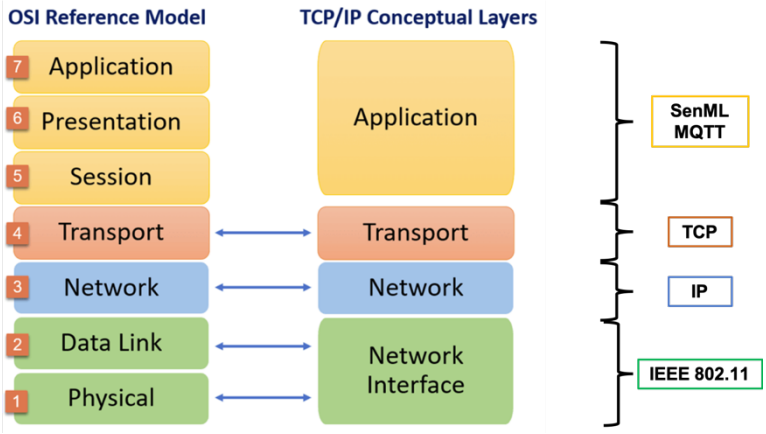

Figura 37. Modelos OSI y TCP/IP y protocolos seleccionados

En esta sección se muestran los protocolos seleccionados en este Trabajo Fin de Máster para las comunicaciones del sensor inteligente: IEEE 802.11 para los niveles físico y de enlace, y MQTT y SenML para nivel de aplicación. Los niveles de red y transporte serán IP y TCP, respectivamente.

4.1.1 IEEE 802.11 – WiFi

En el diseño de una red de sensores como la del presente TFM es fundamental elegir una tecnología de comunicaciones inalámbricas -de nivel físico y de enlace- apropiada para el escenario en cuestión. La elección de esta tecnología afectará directamente a la cobertura, la tasa de envío de datos, el consumo energético…

En la aplicación en particular de este proyecto se desea monitorizar un número elevado de ovejas en un espacio reducido. Esto, unido a los deseos de la Facultad de Veterinaria de no instalar una infraestructura inalámbrica adicional hacen que la tecnología seleccionada sea WiFi. WiFi es el nombre con el que se conoce comúnmente a la familia de estándares IEEE 802.11, que especifican el nivel físico y de enlace de las redes inalámbricas. Dichos estándares permiten operar en diferentes bandas de frecuencia, aunque en este caso se trabajará únicamente en la banda de 2.4GHz, donde opera la antena de la LoPy4.

En este proyecto se ha implantado la infraestructura WiFi para la utilización del sensor inteligente en la Facultad de Veterinaria de la Universidad de Zaragoza. En primer lugar, se ha instalado el router inalámbrico de exteriores *Nanostation Loco M2*, de la marca Ubiquiti. Dicho router permite crear una red inalámbrica a partir de la LAN cableada de la Facultad. Todos los dispositivos que se conecten a esta red recibirán una IP asignada dinámicamente mediante DHCP.

Figura 38. Router (izq) y repetidor (dcha) inalámbricos seleccionados

Sin embargo, en las pruebas de funcionamiento de la red se ha comprobado que la señal que llega a la zona de los animales es muy tenue, dado que hay una distancia de más de 100 metros y varios obstáculos por el camino. Por este motivo, la señal no consigue atravesar las paredes de la caseta en la que se encuentran las ovejas en particular.

Como solución, se ha propuesto la instalación de un repetidor WiFi en la propia caseta de las ovejas, de tal forma que se extienda la cobertura proporcionada por el router principal. Se ha buscado una línea de visión directa LOS entre el router y el repetidor para

maximizar la señal que llega a la zona de los animales. El repetidor seleccionado es el *CPE-210* de la marca TP-Link.

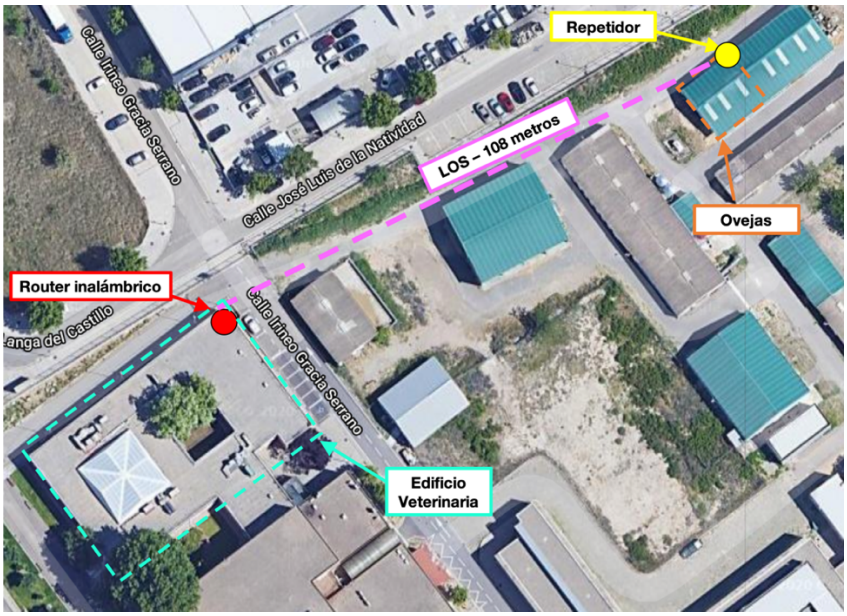

Figura 39. Imagen de satélite de la instalación WiFi realizada en la Facultad de Veterinaria

En la siguiente figura se muestra un diagrama simplificado de las redes que intervienen en la subida de datos de los sensores a un servidor externo -lo que se conoce habitualmente como la nube-.

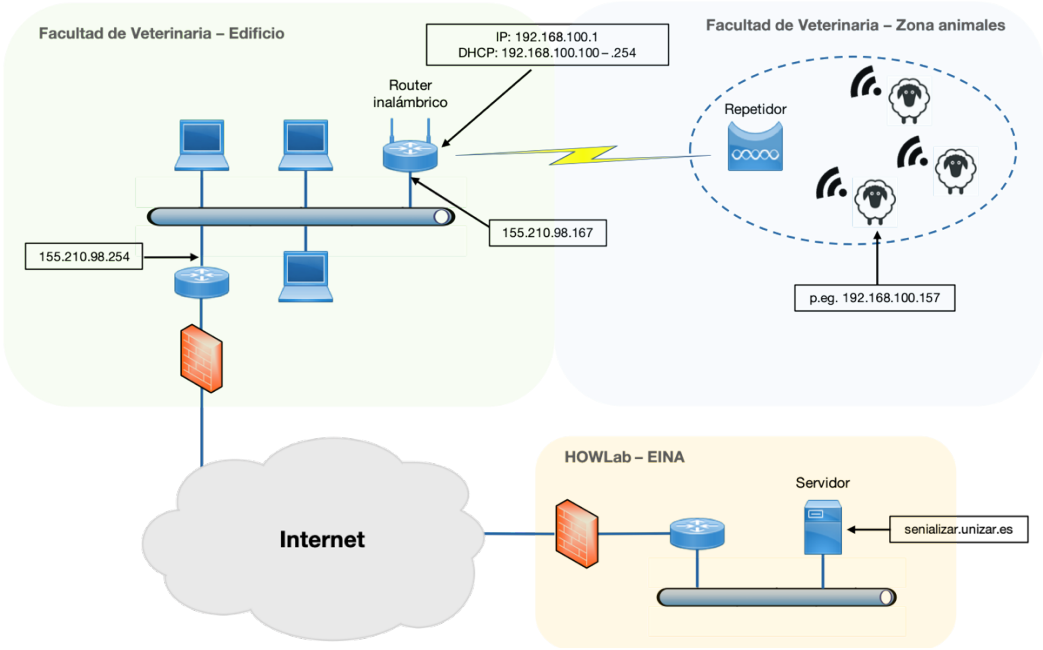

Figura 40. Diagrama de red simplificado del sistema

4.1.2 MQTT

El acceso a Internet requiere el uso de protocolos de nivel de aplicación sobre la pila TCP/IP. Los protocolos utilizados típicamente en redes de comunicaciones, como FTP o HTTP, no son eficientes en redes IoT y suponen una degradación de las prestaciones del sistema.

El principal motivo reside en que debido a las características de los sistemas IoT, donde la conexión con los dispositivos suele ser intermitente dadas sus limitaciones en batería, y se envían una gran cantidad de paquetes de datos de tamaño muy reducido en un ancho de banda limitado, los protocolos del tipo *request/response* suponen un *overhead* -o datos enviados que no corresponden con información al uso- muy elevado [25].

El protocolo de nivel de aplicación adoptado como estándar para IoT es MQTT, el cual se ha implementado en este Trabajo Fin de Máster. A diferencia de HTTP, está basado en una arquitectura *publish/subscribe*, de tal forma que los mensajes son empujados desde el servidor hasta los clientes, y no solicitados por éstos. Además, es un protocolo con bajo *overhead*, eficiente en el uso de ancho de banda y sencillo de implementar [26].

El elemento central de la comunicación con MQTT es el *broker*, que se encarga de recibir y enviar los mensajes a los clientes que se conecten a él. En este caso el *broker* consiste en un servidor situado en el laboratorio de investigación HOWLab de la Universidad de Zaragoza, al que es posible conectarse mediante un nombre de usuario y una contraseña. El protocolo de transporte subyacente a MQTT es TCP, que por defecto no encripta los mensajes. Por ello, en este proyecto se utiliza TLS para la comunicación cliente-servidor, mediante autenticación del *broker* con certificado.

Los clientes corresponden a los sensores *wearable* que publicarán en el *broker* la información de movimiento de las ovejas. Puede haber miles de clientes conectados al *broker* MQTT, y no tienen por qué conocerse entre sí. La información publicada por los sensores quedará almacenada en el *broker* en una cadena de caracteres UTF-8 jerárquica, organizada mediante barras. Esto corresponde al *topic*. Si un cliente se suscribe a un determinado *topic*, recibirá la información contenida en él [26].

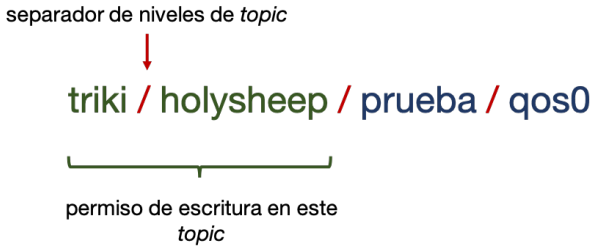

Figura 41. Topic MQTT para los datos del sensor inteligente

Lógicamente, el mismo *broker* puede conectarse a clientes que pertenecen a diferentes redes IoT. Los *wearables* utilizados en animales en este Trabajo Fin de Máster tienen permiso para escribir en el *topic* "*triki/holysheep*", pudiendo crear a su vez diferentes *topics* situados por debajo jerárquicamente.

Otra ventaja de MQTT es que proporciona 3 niveles distintos de calidad de servicio QoS. La QoS define la fiabilidad de la comunicación entre cliente y servidor, es decir la correcta recepción de los mensajes enviados [27]. En este proyecto se utiliza el nivel más bajo de QoS, conocido como *best effort*, donde el emisor envía el mensaje una sola vez, sin preocuparse de si ha llegado correctamente al destino. La principal razón es que se reduce el número de mensajes enviados y, por tanto, el consumo energético, siendo que no es crucial para el funcionamiento del sistema perder de vez en cuando un paquete de información de movimiento del animal.

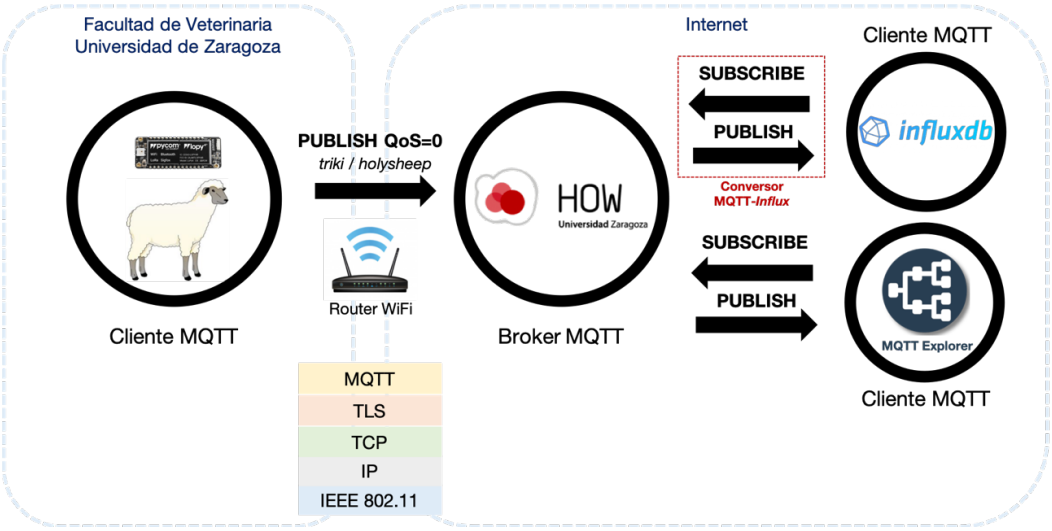

Figura 42. Esquema de clientes y broker MQTT

Almacenamiento en base de datos de series temporales: InfluxDB

La información inercial del animal publicada por el cliente MQTT corresponde a una serie temporal, es decir, una secuencia ordenada de valores en intervalos de tiempo igualmente espaciados [28].

Toda esta información se sube a la nube a través de mensajes MQTT entre cliente y *broker*. Sin embargo, es necesario que quede almacenada en una base de datos. Las bases de datos de series temporales TSDB están optimizadas para una gestión más eficiente de este tipo de medidas, y presentan ciertas ventajas en este tipo de aplicaciones respecto a bases de datos más convencionales [29]:

- Localización de los datos de un mismo intervalo temporal en la misma región de la memoria, para agilizar las consultas sobre un determinado periodo.
- Mayor escalabilidad, debido a que las series temporales de datos implican normalmente un crecimiento más rápido de la base de datos.
- Compresión de datos flexible, perdiendo granularidad o precisión en aquellos datos lo suficientemente antiguos.

La TSDB más popular entre empresas y desarrolladores es *InfluxDB* [30]. Es de código abierto, y está escrita en lenguaje de programación *Go*. *InfluxDB* actúa como cliente MQTT, que se suscribe a un determinado *topic* del *broker*, y a través de un conversor traduce los datos MQTT a "lenguaje *InfluxDB"*. En este caso, el cliente *InfluxDB* y el *broker* MQTT corresponden al mismo equipo -un servidor localizado en la Universidad de Zaragoza-, pero no tendría por qué ser así necesariamente.

El acceso a la base de datos es a través de Internet, tanto por HTTP (puerto 4883), como por HTTPS (puerto 6883). Para la autenticación se utiliza la cuenta de correo de la Universidad de Zaragoza. Además de almacenamiento, a través del complemento *Chronograf* se proporciona una interfaz para la visualización de los datos mediante la realización de consultas a la base de datos. El procedimiento para la representación es sencillo: se hacen consultas a la base de datos y, (1) si son correctas sintácticamente y (2) realmente existen los datos por los que se está preguntando, entonces aparecerán en el panel de visualización las medidas correspondientes al intervalo temporal seleccionado.

Escribir las consultas puede ser tedioso y conducir a errores, sin embargo, mediante la interfaz gráfica se pueden seleccionar fácilmente los datos que se deseen visualizar, el rango temporal e incluso otros parámetros como el identificador único del dispositivo, en este caso un EUI-64. Después, *InfluxDB* se encarga automáticamente de escribir una consulta correcta sintácticamente.

Figura 43. Interfaz gráfica del complemento Chronograf de InfluxDB

También es posible crear tableros de visualización, de tal forma que esas consultas sean fijas y se queden guardadas. Dentro de un mismo tablero puede haber múltiples paneles de visualización. Por ejemplo, uno para representar los datos de acelerómetro y otro para representar los de giróscopo.

Figura 44. Creación de tableros de visualización con Chronograf

La interfaz gráfica de *InfluxDB* también permite descargar rápidamente los datos visualizados en formato CSV, en la parte superior derecha del panel. De este modo se obtienen los datos en local para su procesado con una herramienta más apropiada para esta tarea.

Sin embargo, en el propio *Chronograf* también es posible aplicar ciertas funciones matemáticas a los datos representados. Por ejemplo, puede ser útil utilizar en algunos casos la mediana de un grupo reducido de medidas para filtrar aquellos valores extremos

que han sido captados de forma errónea por el sensor y que estén falseando la representación.

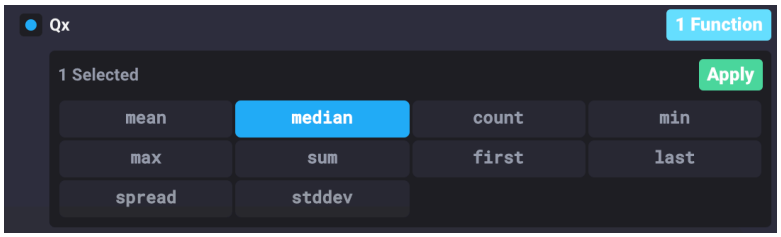

Figura 45. Creación de funciones matemáticas en Chronograf

MQTT Explorer

Previamente a la implementación de *InfluxDB* para la creación de bases de datos, durante la realización de este Trabajo Fin de Máster se ha utilizado la herramienta *MQTT Explorer* para el análisis y comprobación de los mensajes MQTT enviados por el sensor *wearable* al *broker*.

MQTT Explorer actúa como un cliente MQTT que se conecta al *broker* de HOWLab, y se suscribe al *topic* en el que los sensores están publicando la información, mostrándola por pantalla. Todo ello con una interfaz gráfica que permite navegar fácilmente por los diferentes *topics* del *broker* [31].

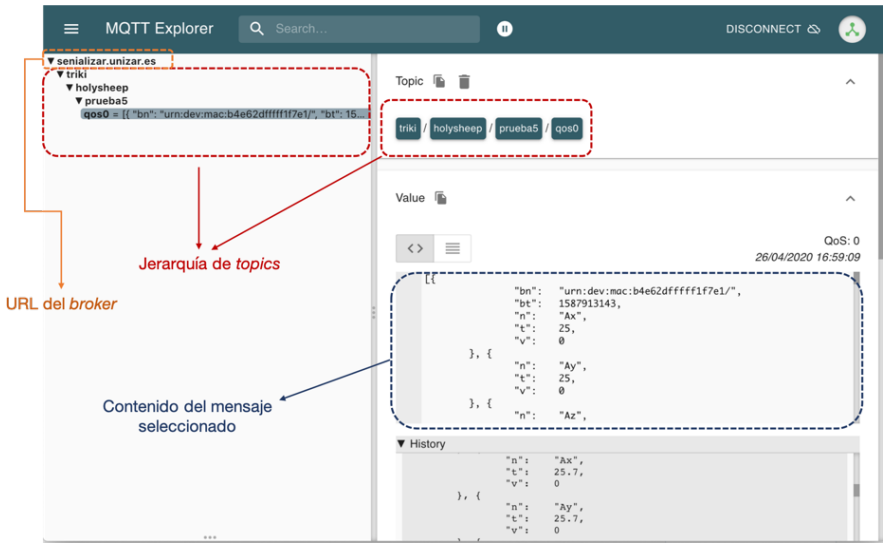

Figura 46. Interfaz gráfica de MQTT Explorer para visualización de datos

4.1.3 Formato de los datos: SenML y JSON

Dentro del nivel de aplicación de la pila de protocolos se encuentra el formato o presentación de los datos. Si se tomara como referencia el modelo OSI, estas tareas estarían localizadas en la capa de presentación. El objetivo de la capa de presentación consiste en representar los datos en un formato que pueda ser comprendido en recepción [32].

Tradicionalmente, XML ha sido el formato de representación de datos utilizado para el intercambio de información en Internet. Sin embargo, se ha extendido en los últimos años el uso de JSON [33] como sustituto, debido a las ventajas que presenta, especialmente interesantes para sistemas IoT. En primer lugar, es más ligero, puesto que utiliza un menor número de caracteres para la representación. Y, en segundo lugar, el *parseo* -o lectura de los datos en recepción- es más rápido [34]. Estas dos características hacen que sea el formato elegido para la representación de los datos en este proyecto, debido a los objetivos marcados de bajo consumo.

Ya se ha expresado anteriormente que el *broker* MQTT al que se envía la información inercial de los animales puede manejar paralelamente conexiones con otros clientes MQTT, desconocidos y pertenecientes a otra aplicación IoT. Sin embargo, es importante garantizar la interoperabilidad del *broker* con ambas aplicaciones IoT. Es decir, debe ser capaz de entender la información recibida de cualquier cliente conectado a él, por muy diferentes que sean las variables físicas medidas por los sensores de cada aplicación.

El estándar SenML (RFC 8428) define un formato para representación de medidas de sensores, con el objetivo de garantizar esta interoperabilidad entre aplicaciones IoT [35]. Por ello, es el formato utilizado en este Trabajo Fin de Máster. SenML forma una capa por encima de los formatos de representación de datos previamente enunciados, es decir, puede construirse a partir de notación JSON, XML… Como se ha adelantado, en este caso el formato subyacente es JSON.

Figura 47. Estándares/Protocolos del nivel de aplicación

La principal ventaja de este formato consiste en el número reducido de caracteres que utiliza para la representación de la información. En sistemas IoT esto cobra gran importancia, ya que a mayor número de *bytes* enviados, mayores son el consumo y el almacenamiento en memoria. A continuación, se muestra un ejemplo de paquete enviado en este proyecto, conformado por tres medidas del eje X del acelerómetro de un único sensor, realizadas con una frecuencia de muestreo de 10 Hz.

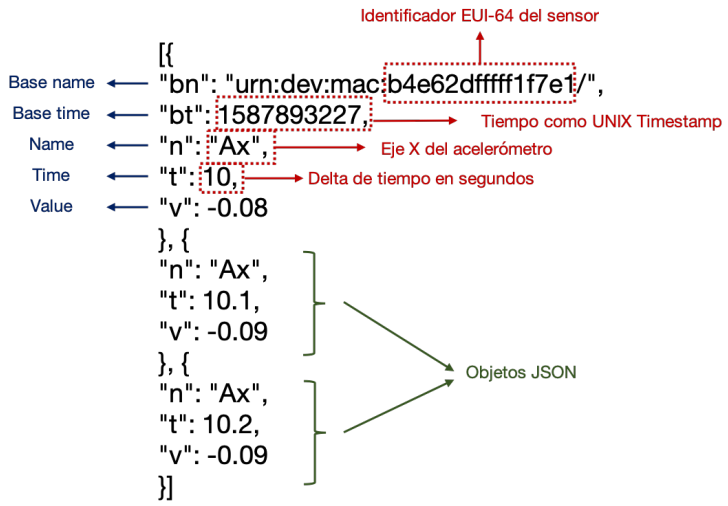

Figura 48. Ejemplo de paquete SenML: 3 medidas del eje X del acelerómetro

Los campos presentes en un paquete tipo SenML enviado por el sensor inteligente de este proyecto son los siguientes:

- ***Base name***: cadena de caracteres que se añade como prefijo a cada campo *name* de los objetos JSON sucesivos en el mismo paquete SenML. Permite identificar al emisor del mensaje. Siguiendo la recomendación de la RFC, en este Trabajo Fin de Máster se utiliza como identificador para cada sensor una URN basada en el identificador EUI-64, obtenido a partir de la dirección MAC del sensor [36].
- ***Base time***: instante temporal origen del paquete, al que se le suman cada uno de los deltas de tiempo del resto de medidas. El formato elegido es *UNIX Timestamp*, muy útil para indicar fechas en aplicaciones distribuidas como ésta, tanto en el lado del cliente como del servidor, dado que es igual independientemente del lugar del mundo en el que nos encontremos [37]. En este caso, el valor de 1587893227 equivale a las 9:27:07am (UTC) del 26 de abril de 2020.
- ***Name***: cadena de caracteres que permite identificar a cada una de las medidas realizadas. En este caso *Ax* corresponde al eje X del acelerómetro.

- ***Time***: delta de tiempo (en segundos) que, sumado al valor de *base time*, identifica el instante temporal al que pertenece cada medida. En este ejemplo las tres medidas se habrían realizado en los siguientes instantes temporales, respectivamente: 9:27:17.0am - 9:27:17.1am - 9:27:17.2am.
- ***Value***: valor medido por el sensor para el registro indicado por *name*. Se debe seleccionar un número de decimales adecuado en función de la variable física medida por el sensor y de la precisión de la aplicación.

  Para poder interpretar estos datos correctamente desde cualquier otro cliente MQTT que se suscriba a este *topic* -por ejemplo, el cliente donde se encuentra la base de datos de series temporales *InfluxDB*- es necesario que todos utilicen este mismo formato de representación SenML. Así, se consigue la interoperabilidad de este sistema IoT entre diferentes dispositivos.

4.2. Análisis de los datos en escenario controlado

El objetivo último de la línea de trabajo en la que se ubica este TFM consiste en la detección automática de las ovejas enfermas en base a una conducta anómala. La idea es que mediante técnicas de aprendizaje automático sea posible detectar tareas con un nivel de actividad muy diferente, como por ejemplo ausencia de movimiento, movimiento lento y movimiento rápido.

Previamente al análisis de los datos reales de monitorización de las ovejas, se ha realizado un experimento para la monitorización del cuerpo humano, en un escenario controlado. De este modo, es posible estudiar la descripción de movimiento proporcionada por el sensor inteligente desarrollado en este proyecto, mediante actividades muy marcadas por el sujeto a intervalos bien definidos.

El objetivo es tener una idea de cómo varían los datos inerciales en diferentes actividades, además de qué algoritmos de *machine learning* y pre-procesado de la señal -si es necesario- hacen falta para una correcta agrupación automática de muestras en actividades. Una vez realizado este estudio en un escenario controlado, el siguiente paso consiste en analizar de forma análoga los datos reales de monitorización de animales (escenario no controlado), lo cual se muestra en el apartado 5.2 *Datos de monitorización animal*.

4.2.1 Comparación de señales temporales por actividad

Para la generación de datos de monitorización en un escenario controlado, el sujeto que lleva el *wearable* ha realizado cíclicamente 4 actividades distintas: sentado, de pie, andando y corriendo. Cada actividad se ha realizado durante 1 minuto, un total de 6 veces. Esto implica la adquisición de 25 minutos de datos de movimiento humano, a una frecuencia de muestreo de 10 Hz.

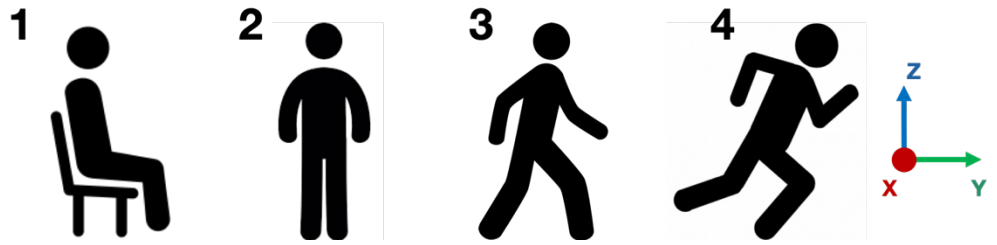

Figura 49. Actividades para el experimento de monitorización del cuerpo humano (escenario controlado)

En primer lugar, se ha representado con el *software* MATLAB cómo varían las series temporales de datos inerciales recogidos, correspondientes a cada una de las 4 actividades -notar los cambios en las escalas de representación para una mejor visualización-. Se ha realizado una distinción de la información recogida por cada sensor -acelerómetro, giróscopo y magnetómetro-, además de los cuaternios obtenidos por el algoritmo de fusión del módulo sensor.

A partir de la representación de los datos ya se pueden extraer conclusiones interesantes acerca de qué información inercial puede ser más prescindible en el desarrollo futuro del sensor inteligente, dada su escasa sensibilidad ante el cambio de actividad.

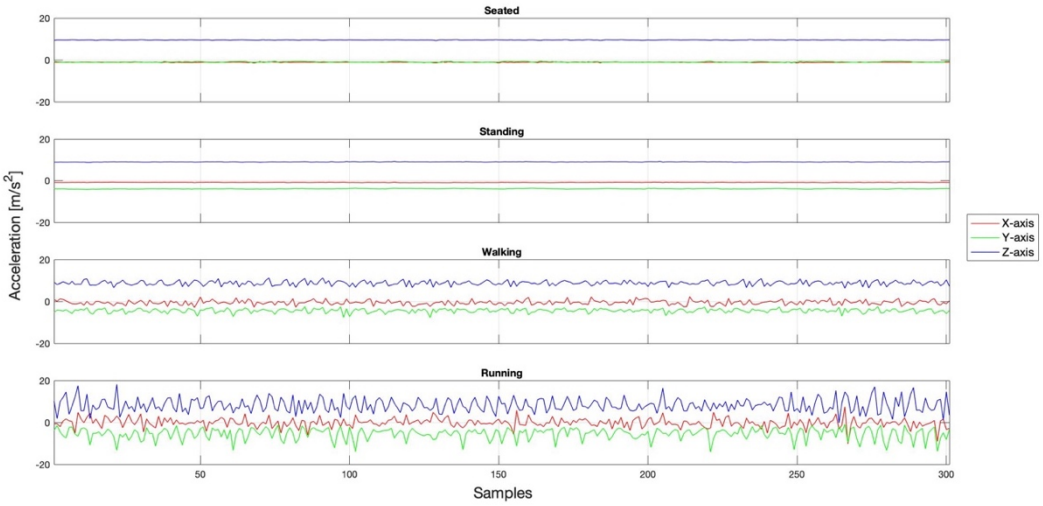

Figura 50. Serie de datos temporal del acelerómetro. 4 actividades del cuerpo humano

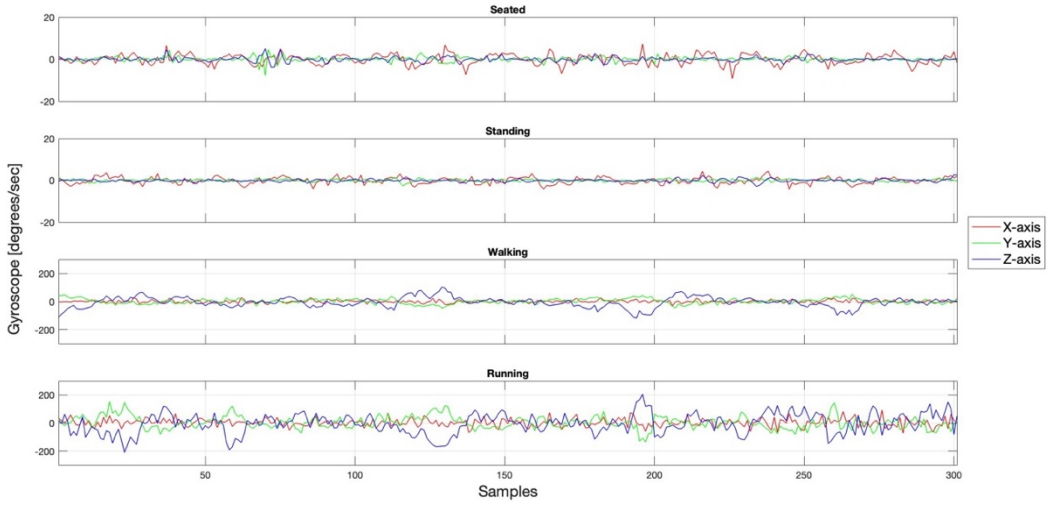

Figura 51. Serie de datos temporal del giróscopo. 4 actividades del cuerpo humano

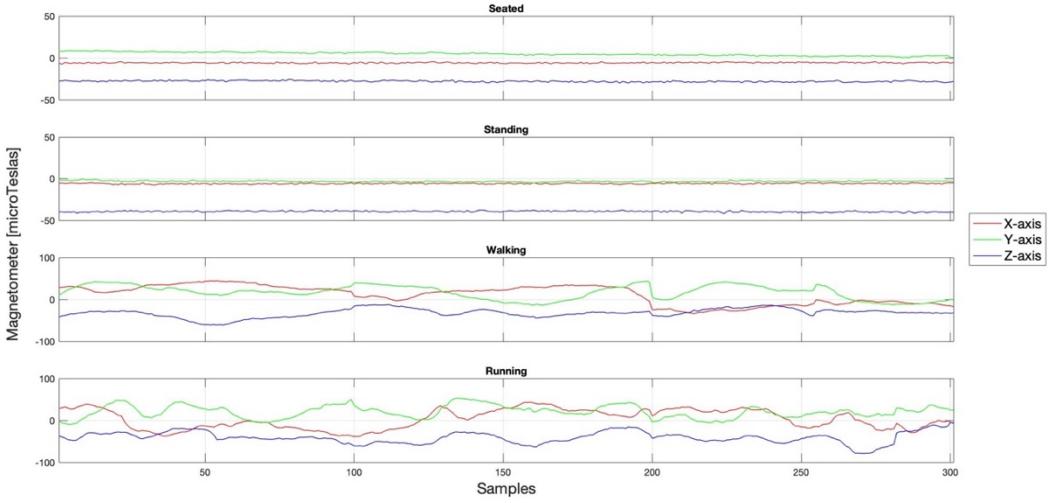

Figura 52. Serie de datos temporal del magnetómetro. 4 actividades del cuerpo humano

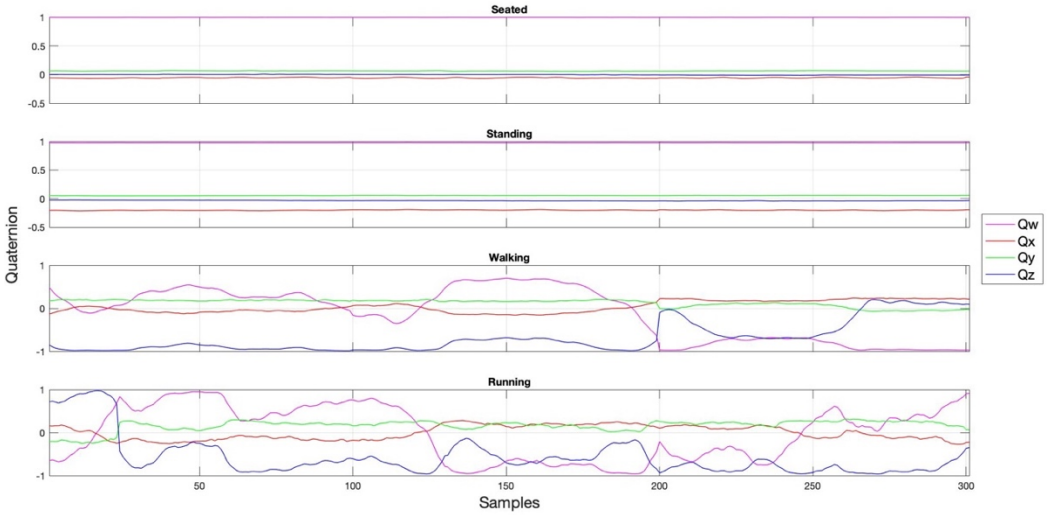

Figura 53. Serie de datos temporal de los cuaternios. 4 actividades del cuerpo humano

4.2.2 Clustering con machine learning no supervisado

El objetivo del análisis de los datos consiste en agrupar de forma automática los datos recibidos en tres categorías: sin movimiento (*Seated* + *Standing*), movimiento lento (*Walking*) y movimiento rápido (*Running*). Si esta agrupación se hace de forma efectiva, debería coincidir con las etiquetas que se han puesto a priori para cada intervalo de tiempo, dado que se conoce con exactitud cuándo tiene lugar cada actividad.

Esto corresponde a un problema típico de *machine learning* de reconocimiento de patrones con algoritmos de aprendizaje no supervisado: el *clustering*. En el *clustering* la única entrada es un conjunto de datos que deben ser agrupados atendiendo al criterio del algoritmo en cuestión. El algoritmo sólo tiene acceso a las características de los datos -en este caso 13, correspondientes a la medida de cada eje de los sensores-, y no recibe ninguna información adicional, como por ejemplo etiquetas.

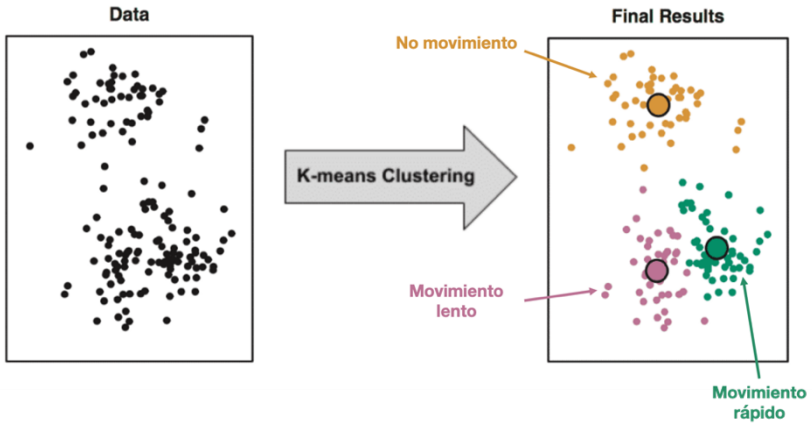

Figura 54. Diagrama ejemplo (datos ficticios) de una tarea de clustering con K-means

En este Trabajo Fin de Máster el desarrollo del mecanismo de *clustering* se ha realizado en lenguaje Python, a través de la herramienta Jupyter Notebooks. Para evaluar las prestaciones de las diferentes soluciones propuestas se muestra: (1) el porcentaje de error en el *clustering*, (2) una comparación entre la actividad predicha y la real para cada muestra y (3) la matriz de confusión, que permite ver fácilmente cómo se está equivocando el algoritmo en las diferentes categorías.

Primera aproximación: K-means

Como primer intento de *clustering* de los datos inerciales del sensor, se ha optado por aplicar directamente el algoritmo *K-means* [38] sobre los datos de entrada destinados al entrenamiento -en este caso el 80% del total-.

El *K-means* es uno de los algoritmos más utilizados en tareas de *clustering*. Inicialmente, se seleccionan en el espacio K centros de *cluster* aleatoriamente. Cada dato de entrada será asignado al *cluster* con cuyo centro presente una menor distancia. El centro del *cluster* se actualiza entonces para estar a la misma distancia de todos los datos asignados a él. El algoritmo converge cuando los centros de los *clusters* dejan de actualizarse, momento en el que la etapa de entrenamiento del algoritmo se da por concluida.

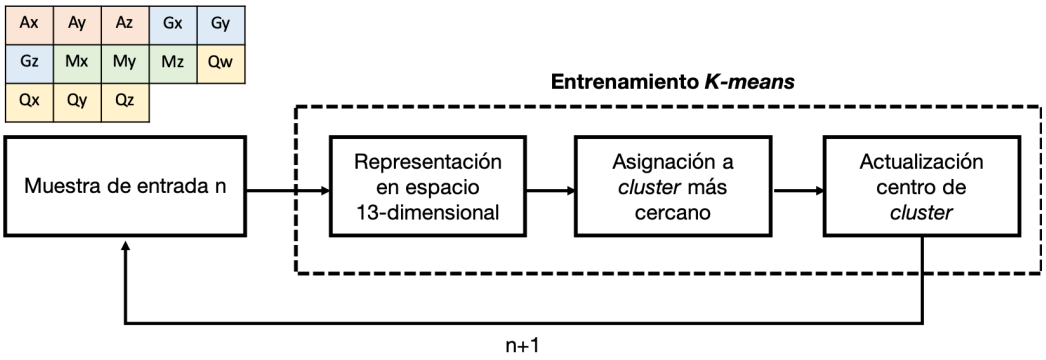

Figura 55. Mecanismo de entrenamiento del algoritmo de clustering K-means

En este caso, cada dato de entrada al algoritmo corresponde a las 13 medidas del sensor inteligente en un determinado instante temporal. Típicamente, es común referirse a estas 13 dimensiones con el nombre de características.

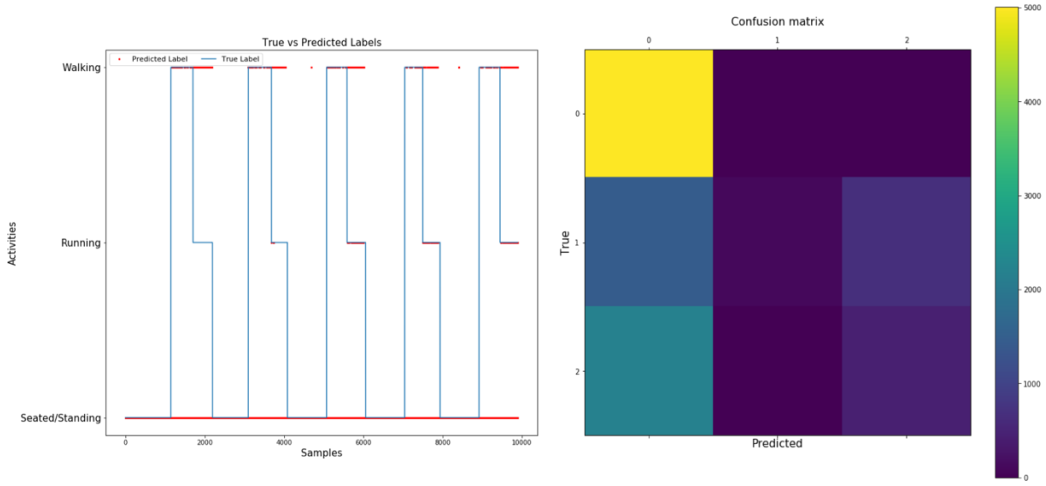

Figura 56. Aproximación inicial: resultado del clustering (izq) y matriz de confusión (dcha) [train]

El resultado del *clustering* es muy pobre en este primer caso. El porcentaje de error del algoritmo en la fase de entrenamiento es del 44% a la hora de agrupar en el mismo *cluster* intervalos de muestras pertenecientes realmente a la misma actividad. Por ello, ni siquiera se evalúa el modelo de *clustering* resultante del entrenamiento con nuevos datos de entrada, en lo que se conoce como fase de test.

Mejora: descarte de datos de monitorización superfluos

A simple vista de las series temporales de datos es apreciable cómo hay medidas del sensor que no cambian entre actividades, son datos superfluos que incluso dificultan la tarea de *clustering* del algoritmo, dando una idea de similitud entre grupos de muestras pertenecientes en realidad a distintas categorías.

En este caso, por la posición del sensor, las medidas del eje X son menos variantes que las medidas del resto de ejes. Si, por ejemplo, se descartan las medidas del eje X del giróscopo en el algoritmo *K-means*, se consigue reducir el porcentaje de error en la fase de entrenamiento del *clustering* a un 39%.

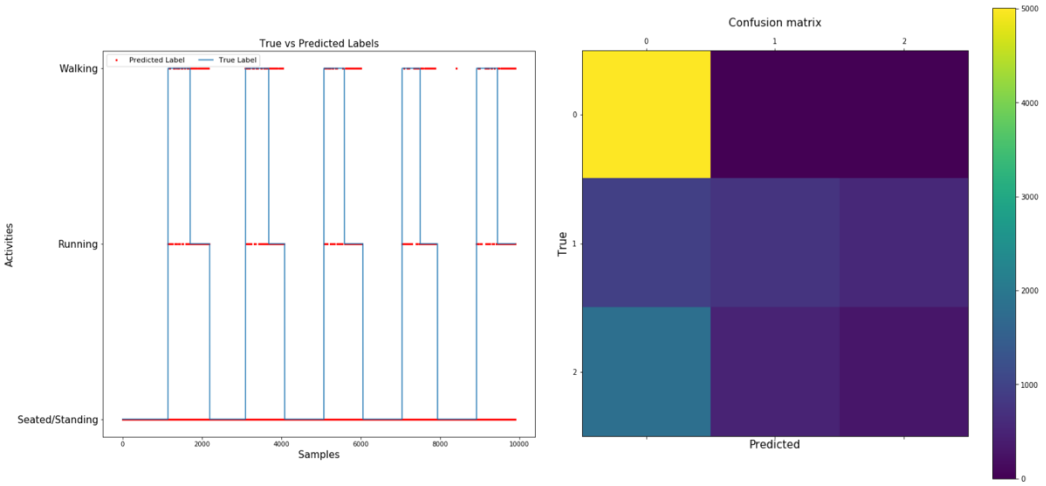

Figura 57. Descarte de datos superfluos: resultado del clustering (izq) y matriz de confusión (dcha) [train]

Las prestaciones del algoritmo de aprendizaje no supervisado siguen sin ser satisfactorias. Para obtener mejores resultados una posibilidad sería utilizar algoritmos más complejos del estado del arte, que permiten realizar un *clustering* más preciso que el *K-means* mediante mecanismos de extracción de las características más relevantes de los datos. Sin embargo, también es posible realizar un pre-procesado de estos datos previo a la implementación del algoritmo de *machine learning*, lo cual se ha realizado en este proyecto, mejorando ostensiblemente los resultados.

Solución propuesta: pre-procesado de los datos

El fin que se busca con el pre-procesado de los datos es facilitar al algoritmo la tarea de *clustering* en diferentes categorías, haciendo más visibles las diferencias entre las muestras de los tres niveles de actividad que se desean agrupar.

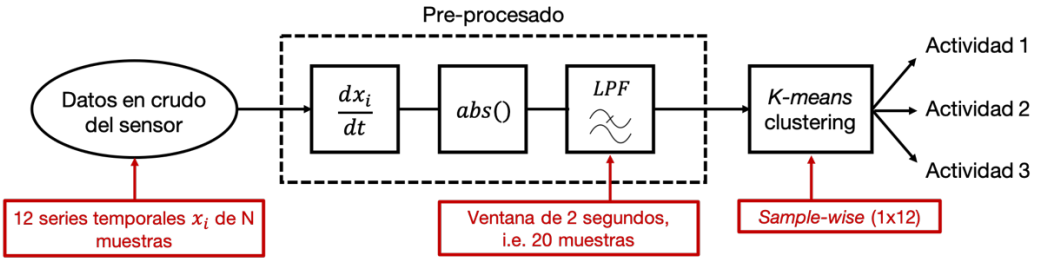

Figura 58. Diagrama de bloques de la solución propuesta para análisis de los datos (escenario controlado)

Si se observan las series de datos temporales -12 en total, puesto que se ha descartado el eje X de giróscopo-, se aprecia como un mayor nivel de actividad implica una mayor variación de las señales, alcanzándose máximos y mínimos de más amplitud, y pendientes más acusadas en la forma de la señal. Por lo tanto, si se hace una aproximación por diferencias finitas de la derivada de cada una de las señales temporales, se conseguirá: (1) que estos cambios en la tendencia de la señal sean más notables y (2) que los grupos de muestras con poca variabilidad tiendan a una amplitud nula.

$$f'(x) = \frac{f'(x+1) - f(x-1)}{\Delta x}$$

Posteriormente, se calcula el valor absoluto del resultado previo, puesto que no importa si estos cambios de tendencia son en un sentido o en otro, el objetivo es detectar variaciones.

El último paso consiste en suavizar la señal con un filtro paso bajo del tipo gaussiano, que atenúa las altas frecuencias. El filtrado de cada una de las doce señales temporales se realiza con un tiempo de ventana de dos segundos, que con la frecuencia de muestreo utilizada de 10 Hz corresponde a 20 muestras. Cuanto mayor sea este tiempo de ventana, mayor será el filtrado realizado.

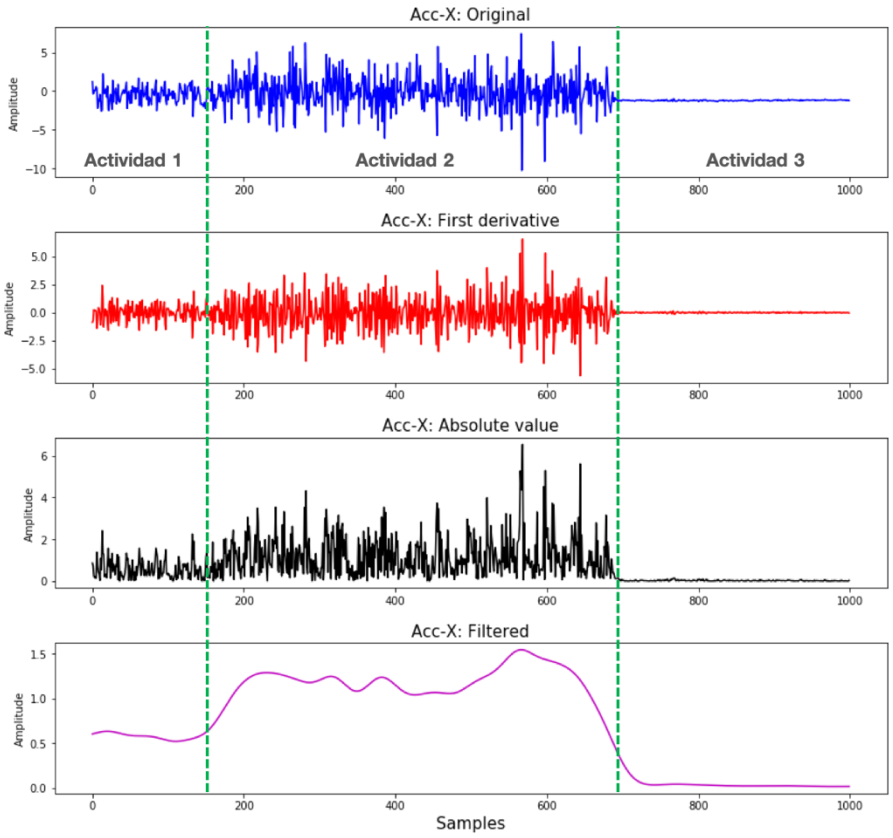

Figura 59. Pre-procesado de un grupo de muestras del eje X del acelerómetro

Tras el pre-procesado de los datos, es posible distinguir fácilmente tres niveles distintos de amplitud en cada una de las señales temporales. Cada uno de estos niveles estará asociado a las tres actividades que se desean detectar. Al pasarle como entrada al algoritmo *K-means* estas muestras procesadas, se alcanza un porcentaje de error en la fase de entrenamiento del *clustering* de hasta el 2.8%, muy inferior a los obtenidos previamente.

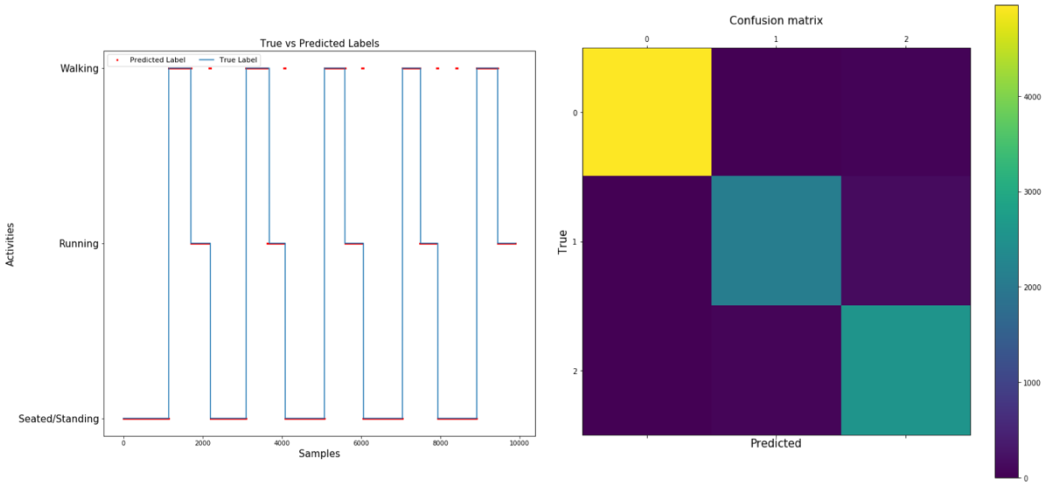

Figura 60. Solución final: resultado del clustering (izq) y matriz de confusión (dcha) [train]

Una vez se ha obtenido un modelo potencialmente válido de *clustering*, es hora de evaluar sus predicciones con el 20% de los datos restantes, correspondientes a la parte de test. El porcentaje de error obtenido en esta fase es del 2.4%. Para asegurar un funcionamiento adecuado del mecanismo de agrupación automática, sería conveniente evaluar sus prestaciones con cantidades mucho mayores de datos. Sin embargo, el objetivo no es generar un modelo de *clustering* definitivo, sino mostrar la validez de los datos obtenidos en el proceso de monitorización, y un ejemplo de procesado que pueda implementarse a corto plazo en el propio sensor inteligente.

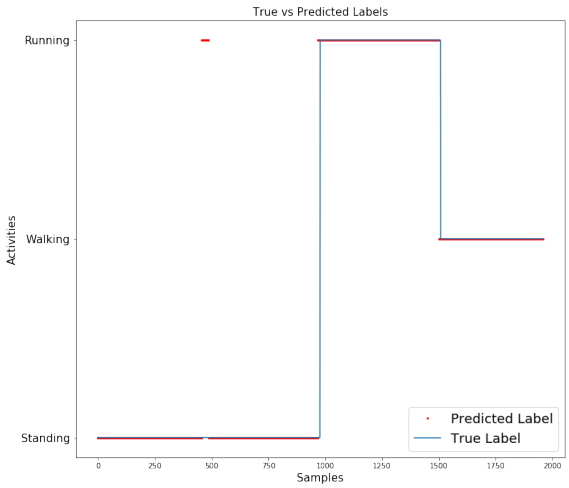

Figura 61. Solución final: resultado del clustering [test]

Una parte importante de este proyecto ha estado centrada en la reducción del consumo del microcontrolador en la acción más importante en una aplicación de monitorización: la lectura o muestreo del sensor IMU. Sin embargo, también se ha puesto de relieve la importancia del consumo energético del propio sensor IMU, que como se ha mostrado en el apartado 3.2.2 *Mejora de consumo energético en la lectura del sensor IMU*, puede oscilar entre el 62% y el 80% del consumo total del sensor inteligente de este TFM en el procedimiento de lectura, dependiendo del modo de operación en el que trabaje.

En este sentido, surge una cuestión importante: cuál es el compromiso existente entre reducir el consumo del sensor IMU -seleccionando un modo que proporcione menos información- y la precisión alcanzada en la agrupación automática en actividades. Se ha realizado un estudio del porcentaje de error en el *clustering* para los diferentes modos de operación del sensor IMU, mostrando además el consumo energético asociado a cada uno de ellos. Los datos del eje X del giróscopo se han descartado en todos ellos.

Modos de operación del sensor IMU	Error (%) en el entrenamiento del clustering	Consumo de corriente (mA) del sensor IMU
ACC	35	5
GYR	2.7	9
ACC+GYR	2.7	10
ACC+GYR+MAG	2.7	10
9-DOF	2.7	12

Tabla 3. Prestaciones de la detección automática de actividades según modo de operación del sensor IMU

A la vista de los resultados, se puede concluir que simplemente con los datos procedentes de dos ejes del giróscopo es posible realizar una agrupación automática de las actividades en tres categorías, sin perder precisión. Este es un paso muy importante a la hora de mejorar el funcionamiento del sensor inteligente en etapas de diseño futuras, puesto que (1) pueden reducirse 3mA de consumo del sensor IMU sin empeorar las prestaciones y (2) el envío inalámbrico de información se daría con una menor frecuencia. Como justificación de esto último: en este proyecto -modo 9-DOF- se leen 13 registros de 2 *bytes* cada uno, lo que supone 260 *bytes*/*s* a 10Hz; mientras que se podría pasar sin penalización a 2 registros, teniendo así 40 *bytes*/*s*, menos de una sexta parte.

5. Resultados experimentales

La importancia de un consumo energético reducido en aplicaciones IoT con *wearables* es vital para un funcionamiento adecuado del sistema. En la primera parte de esta sección se muestra el estudio realizado del consumo de corriente del sistema final desarrollado en este TFM.

Posteriormente, se presentan los datos de monitorización de las ovejas de la Facultad de Veterinaria obtenidos con el prototipo definitivo, así como el resultado del análisis de estos datos de forma similar a como se ha realizado en el apartado 4.2 *Análisis de los datos en escenario controlado*, mostrando la diferencia de prestaciones entre ambos experimentos.

5.1. Consumo energético

La reducción en la medida de lo posible del consumo de corriente es fundamental a la hora de diseñar un sensor inteligente. El desarrollo del *software* embebido en el microcontrolador ha estado enfocado con este objetivo, logrando reducciones muy notables, especialmente en la tarea clave del *wearable*: la lectura del sensor. Por ello, en esta sección se muestran varias gráficas comparativas de los consumos medidos con el prototipo definitivo del sensor inteligente. Los resultados mostrados en esta sección corresponden a un modo de operación 9-DOF del sensor BNO055 -lo que supone un consumo de corriente del sensor IMU de 12mA-.

Todas las medidas han sido realizadas experimentalmente con el dispositivo ADC-20 de Pico Technology. En el *Anexo V* puede verse el montaje y el *software* utilizado con la configuración correspondiente.

Gráfica	Descripción	Conclusiones
Figura 62	Lectura de sensor en *deep sleep* vs *light sleep*	2mA vs 4mA de consumo
Figura 63	2 ciclos de envío idénticos *deep sleep* vs *light sleep*	Mayor rapidez al despertar con *light sleep*
Figura 64	2 ciclos de envío diferentes *deep sleep* vs *light sleep*	Envío cada más tiempo con *light sleep* dado el mayor almacenamiento
Figura 65	Lectura del sensor en paralelo al envío WiFi	Se evita la pérdida de datos. Se adelantan los envíos posteriores

Tabla 4. Resumen de las gráficas de consumo de la sección 5.1

Lectura del sensor IMU: deep sleep vs light sleep

La lectura del sensor IMU es llevada a cabo por el coprocesador ULP mientras los procesadores principales están dormidos. Dependiendo de si el modo de *sleep* es *light* o *deep*, el consumo de corriente medio del sensor inteligente variará. Se ha comprobado como, efectivamente, el consumo de corriente pasa de 2 mA a más de 4 mA -obviando el consumo asociado al sensor IMU-.

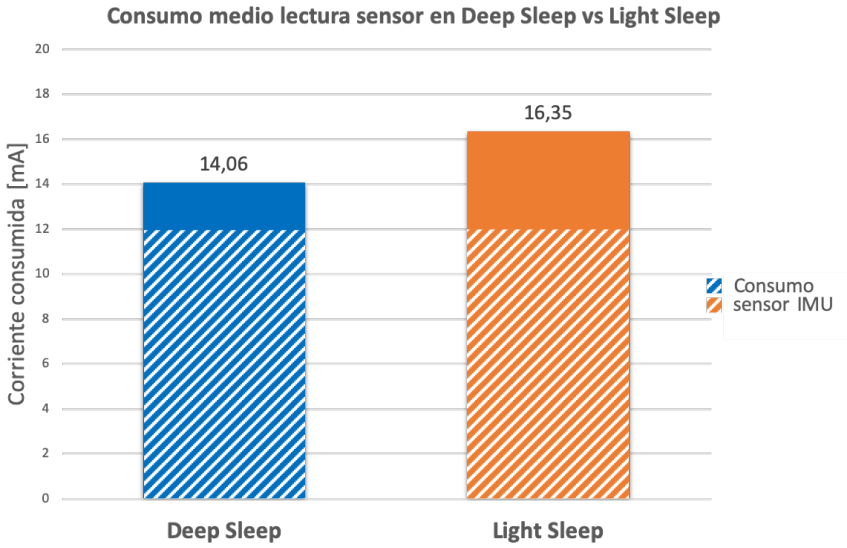

Figura 62. Consumo de corriente para la lectura del sensor IMU en deep sleep vs light sleep

Algoritmo de monitorización completo: deep sleep vs light sleep

En la siguiente gráfica se muestra una comparación de las dos opciones de funcionamiento del sensor inteligente: *light sleep* y almacenamiento en RAM, o *deep sleep* y almacenamiento en memoria *RTC FAST*. Se suponen dos ciclos de envío a la nube idénticos, cada 30 segundos.

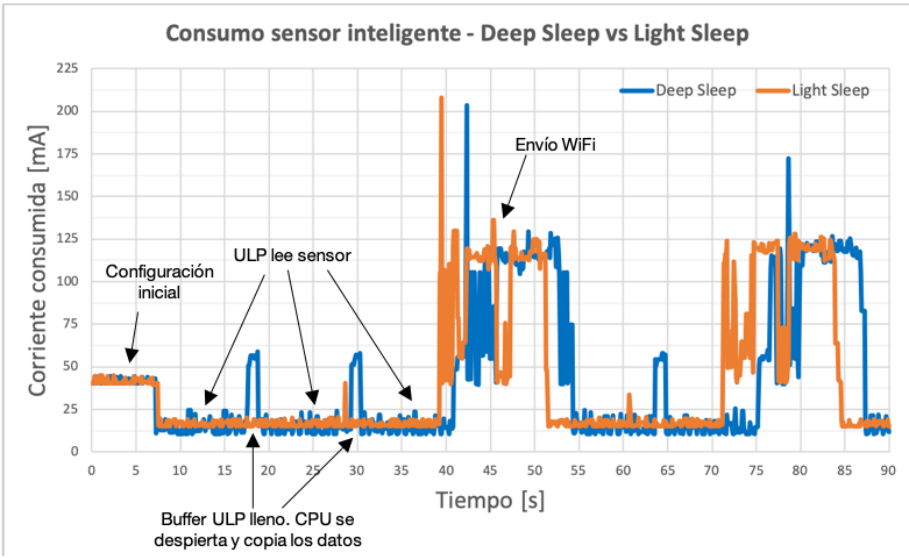

Figura 63. Comparación de consumo de los modos de funcionamiento del wearable. 2 envíos WiFi idénticos

Durante la etapa de *sleep*, mientras el ULP lee el sensor, se observa el aumento en el consumo mostrado anteriormente para el modo *light sleep*. El ULP es capaz de almacenar aproximadamente 10 segundos de datos en la memoria *RTC SLOW*. Una vez lleno este *buffer*, la CPU principal se despierta -apreciándose en el consumo de corriente-, copia el contenido del *buffer* y se vuelve a dormir. Cuando se han llenado los 30 segundos de datos del *buffer* de envío, el sensor inteligente se conecta a la red WiFi y manda los datos.

En el modo *light sleep* no es posible ver el aumento en el consumo de corriente cuando la CPU se despierta, copia los datos del buffer ULP, y se vuelve a dormir. Esto se debe sencillamente a que el proceso de dormir/despertar con *light sleep* es más veloz que con *deep sleep*, y una precisión de 100ms -la máxima disponible en el ADC-20- en la representación no es suficiente para apreciarlo. Por este mismo motivo, el envío WiFi se produce ligeramente antes en *light sleep*.

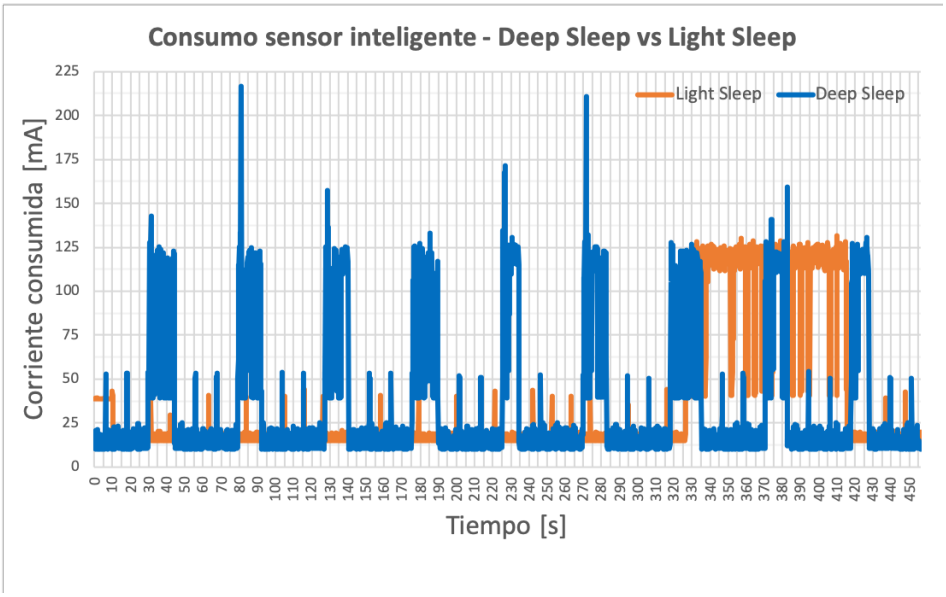

Figura 64. Comparación de consumo de los modos de funcionamiento del wearable. Diferente tiempo de envío

La principal ventaja de trabajar con *light sleep*, sin embargo, consiste en periodos de envío más elevados, dada la mayor capacidad de almacenamiento. Esto se muestra en la Figura 64, donde en *deep sleep* el ciclo de envío es de nuevo 30 segundos, pero en *light sleep* es de 5 minutos. De este modo, el tiempo total de envío WiFi se ve reducido considerablemente, puesto que se minimiza el número de intentos de conexión a la red.

$$I_{consumida_{media}}\,[Deep\ Sleep] = 44mA \;\rightarrow\; E_{consumida\ en\ 1\ hora} = 602\,J$$

$$I_{consumida_{media}}\,[Light\ Sleep] = 34mA \;\rightarrow\; E_{consumida\ en\ 1\ hora} = 465\,J$$

A partir de la corriente media consumida en estos 5 minutos en cada uno de los dos modos de trabajo, y teniendo en cuenta que la batería utilizada en el momento de la toma de medidas

tiene una tensión de salida de 3.8V, es posible obtener la diferencia de energía media consumida en una hora entre ambas situaciones.

Lectura del sensor IMU durante el envío WiFi

A continuación, se muestra la diferencia en el consumo de corriente si se lee el sensor IMU en paralelo al envío de datos WiFi, con respecto a si la lectura no se realiza y se produce ese "agujero" en los datos de movimiento.

Efectivamente, si se lee el sensor IMU en paralelo, se adelantará el envío de datos en el siguiente ciclo de operación, dado que el *buffer* de envío se llena antes.

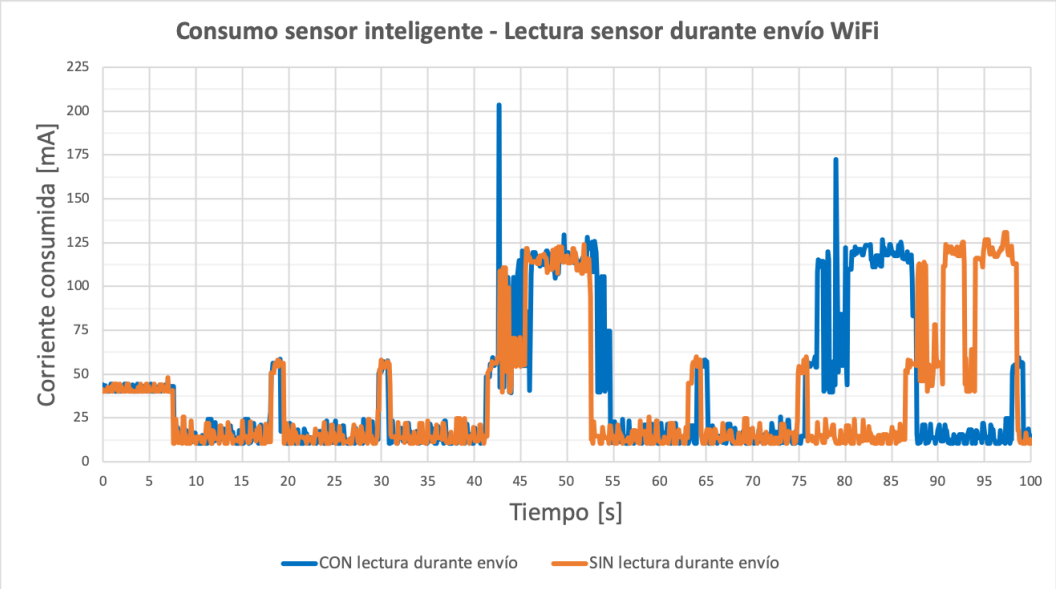

Figura 65. Diferencia de consumo energético al leer el sensor IMU en paralelo al envío WiFi

5.2. Datos de monitorización animal

En este proyecto se ha realizado un segundo experimento de monitorización, en este caso con ovejas, en lo que sería el escenario definitivo donde se desea implementar el sensor inteligente desarrollado en este TFM.

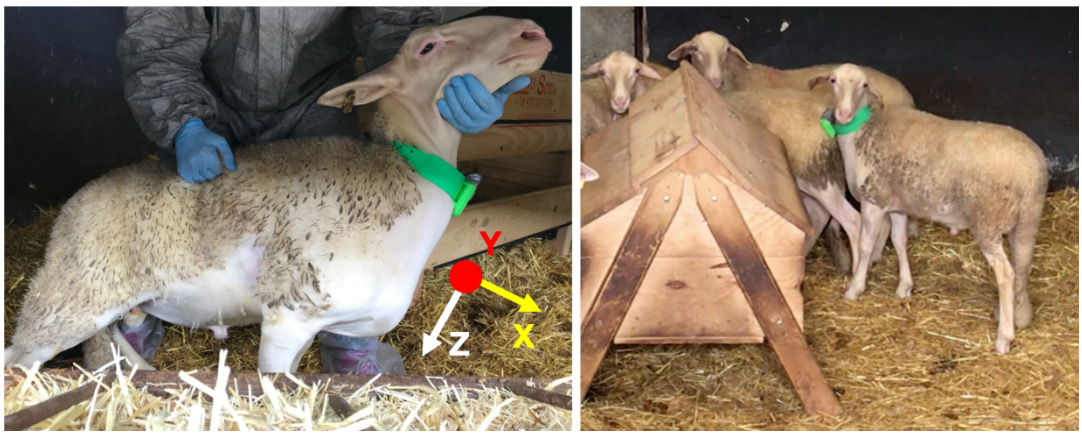

Figura 66. Localización del sensor inteligente en la oveja para la realización de pruebas

Para este segundo experimento, se ha utilizado el prototipo definitivo presentado en el apartado 2.3 de este documento. El sensor se ha colocado en una única oveja de la Facultad de Veterinaria, con el objetivo de analizar y procesar estos datos de monitorización, del mismo modo que se hizo en el primer experimento. Se pretende además evaluar las prestaciones de este tratamiento de los datos, teniendo en cuenta que el objetivo último de esta línea de trabajo consiste en la detección automática de ovejas enfermas en base a una conducta anómala.

De nuevo, se busca realizar una agrupación automática de los datos recabados en tres categorías, correspondientes a: (1) de pie sin movimiento (*Standing*), (2) movimiento lento (*Walking*), y (3) movimiento rápido (*Running*). Para verificar el funcionamiento del sistema de *clustering* en este experimento, se ha colocado un móvil grabando el movimiento de las ovejas, de tal forma que sea posible etiquetar manualmente grupos de muestras en las tres categorías.

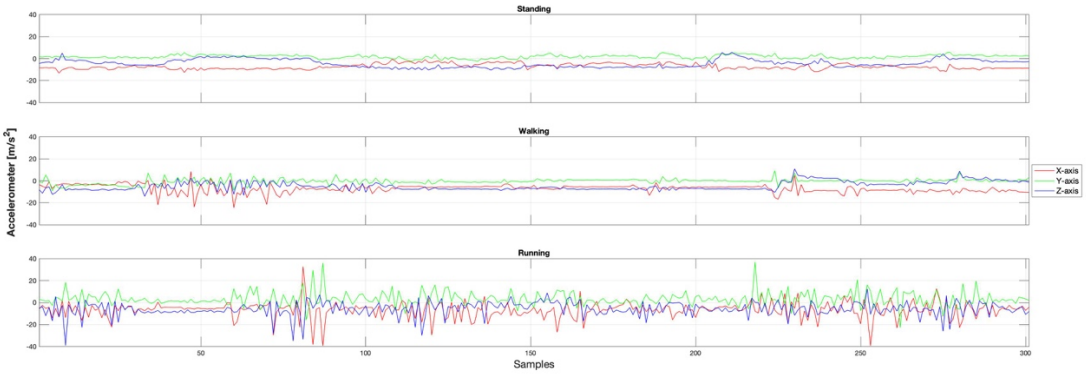

Figura 67. Serie de datos temporal del acelerómetro por actividades. Experimento con ovejas

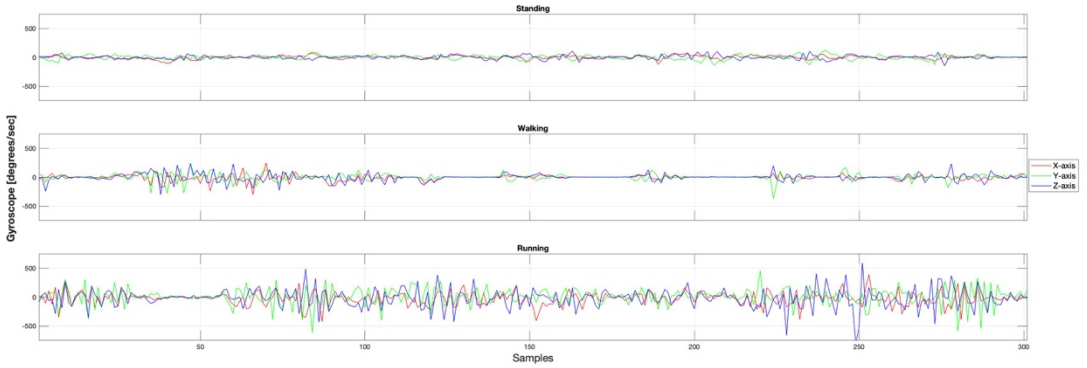

Figura 68. Serie de datos temporal del giróscopo por actividades. Experimento con ovejas

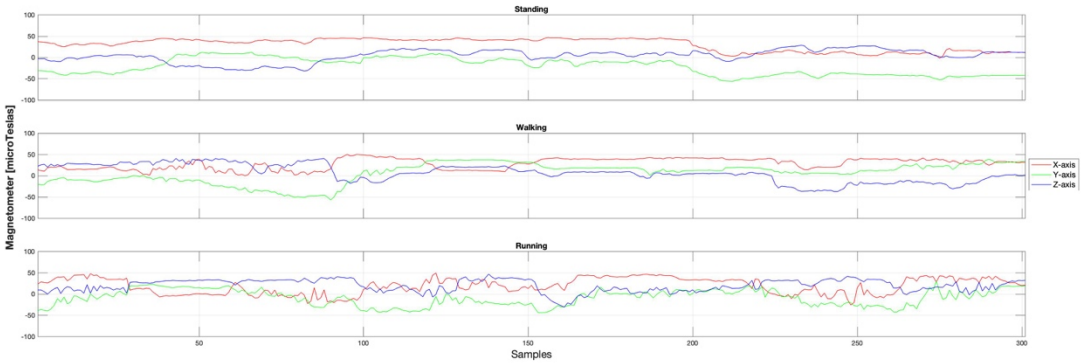

Figura 69. Serie de datos temporal del magnetómetro por actividades. Experimento con ovejas

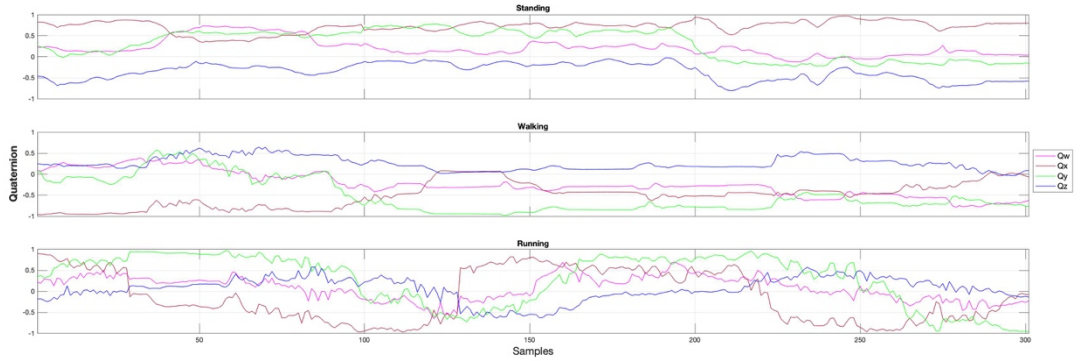

Figura 70. Serie de datos temporal de los cuaternios por actividades. Experimento con ovejas

En base a la observación de las grabaciones de las ovejas, y de los datos obtenidos agrupados por actividades, es posible identificar dos cambios importantes en la información inercial obtenida, respecto del primer experimento en un escenario controlado.

En primer lugar, debido al instinto gregario de las ovejas -como se ha confirmado con el personal veterinario-, tienden a agruparse entre ellas, teniendo mucho contacto especialmente en la parte del cuello, donde va colocado el collar del *wearable*. Esto supone cambios eventuales en la posición del sensor, que no sucedían en el primer experimento. Como consecuencia, las medidas de un determinado eje no son casi estáticas, y no es posible descartar ningún dato, al menos a priori.

Por otra parte, las secuencias de movimiento de las ovejas son breves y esporádicas, derivando en un número de muestras de movimiento muy reducido respecto a las de situación estática.

Agrupación automática en categorías (evaluación con dataset externo)

El pre-procesado de los datos es similar al realizado con anterioridad. En primer lugar, se deriva por diferencias finitas cada una de las series temporales de datos asociadas a las medidas del sensor. A continuación, se obtiene el valor absoluto de las señales temporales diferenciadas, y finalmente se filtran paso bajo. En este caso el filtrado es ligeramente mayor, con una ventana temporal de 4 segundos, o lo que es lo mismo, 40 muestras.

Sobre estos datos se aplica el algoritmo de *clustering K-means* para agrupación de cada muestra temporal -de dimensión igual al número de características consideradas- en tres categorías. En este caso, los datos de entrada para los que se obtienen mejores resultados son los tres ejes del giróscopo, i.e. tres dimensiones.

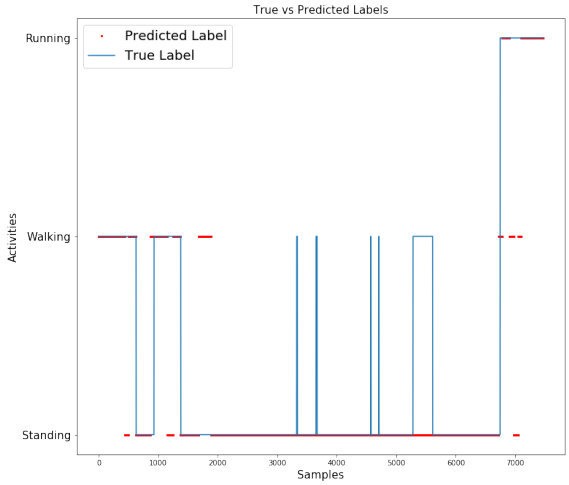

Figura 71. Resultado del clustering de los datos de monitorización de ovejas de Veterinaria [train]

Las prestaciones del algoritmo no son tan positivas como en el primer experimento (escenario controlado), obteniéndose un porcentaje de error en el entrenamiento del 14% en la agrupación automática de muestras, respecto a las etiquetas puestas en función de las grabaciones observadas.

Sin embargo, el mecanismo de *clustering* está funcionando mejor de lo que puede parecer a priori. La razón es que el etiquetado de los datos no es del todo preciso, puesto que el movimiento del animal es muy intermitente durante los intervalos de actividad. Del mismo modo, en periodos estáticos también aparecen movimientos de cuello o del propio collar que el sensor interpreta como actividad del animal.

Mediante la observación en un histograma del número de muestras asignadas a cada categoría, para el caso de las etiquetas a partir de las grabaciones -supuesto caso real-, respecto del caso de la agrupación automática con *K-means*; puede observarse como no existen grandes diferencias.

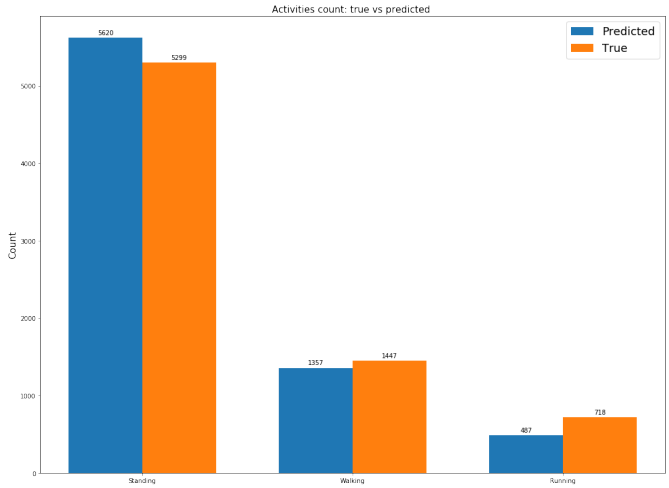

Figura 72. Número de muestras asignadas a cada actividad: caso real vs clustering automático

Teniendo en cuenta que el objetivo final de esta línea de trabajo consiste en detectar si una oveja está enferma a partir de la comparación de sus datos de movimiento con los de una oveja sana, estos resultados podrían considerarse como potencialmente válidos, a la espera de verificar realmente cómo son los datos inerciales de una oveja enferma, lo cual no se ha realizado en este Trabajo Fin de Máster.

Como cierre de la sección de *Resultados experimentales*, resulta interesante evaluar cómo se comporta en fase de test el modelo de *clustering* desarrollado, con datos externos de monitorización de ovejas. En [39] se muestra el dataset utilizado para esta tarea, elaborado por la Universidad de Twente (Países Bajos). Dicho dataset está conformado por los datos de

(1) acelerómetro, (2) giróscopo y (3) magnetómetro de una oveja durante un día completo, etiquetados en 9 actividades distintas.

En este proyecto, los datos de entrada que pasarán el pre-procesado y que utilizará el algoritmo *K-means* serán únicamente los tres ejes de giróscopo, mientras que las tres actividades de interés serán de nuevo *Standing*, *Walking* y *Running*.

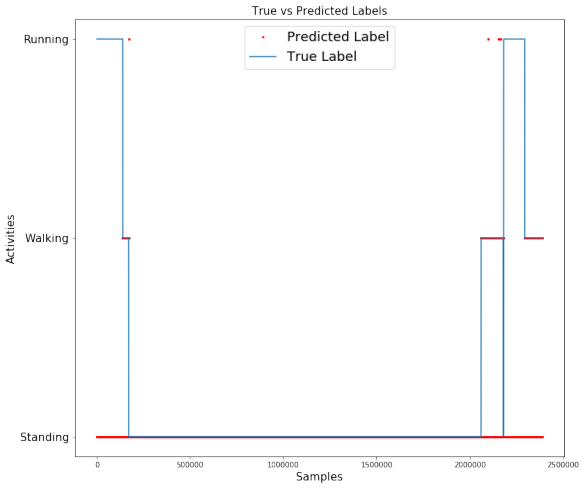

Figura 73. Resultado del clustering de los datos de monitorización de ovejas de Twente [test]

El porcentaje de error en la fase de test es del 15%, tomando como entrada las medidas de giróscopo realizadas por la Universidad de Twente en [39]. Como se ha comentado con anterioridad, mediante algoritmos más complejos de *machine learning*, que permitieran realizar una extracción precisa de las características más relevantes de las medidas del sensor, sería posible obtener mejores prestaciones, tomando como referencia los resultados mostrados en el apartado 1.4 *Estado del arte*.

Sin embargo, en este proyecto se ha mostrado un ejemplo de procesado y técnicas de aprendizaje automático sencillas, que pueden ser perfectamente implementados a corto plazo en el propio sensor inteligente, pudiendo diferenciar con relativamente buena precisión, al menos, si el animal se está moviendo.

Así, en etapas futuras de diseño del sistema -atendiendo a la Figura 28 de este documento-, el sensor inteligente realizaría el procesado, enviando tan sólo eventos como por ejemplo: “la oveja se está moviendo”. Llegados a ese punto, el cómputo global del consumo energético de la fase de envío WiFi sería mínimo, estando el consumo del sensor condicionado en prácticamente su totalidad por la lectura de sensor IMU, que debería seguir dándose constantemente.

Minimizar el consumo de corriente de esa etapa de muestreo del sensor IMU ha sido uno de los objetivos centrales de este Trabajo Fin de Máster, logrando una reducción del 91%

respecto del caso inicial, como se ha visto en el apartado 3.2 *Lectura del sensor mediante el coprocesador ULP*.

6. Conclusiones y líneas de futuro

6.1. Conclusiones

En este Trabajo Fin de Máster se ha desarrollado un sensor inteligente para monitorización de ovejas en la Facultad de Veterinaria de la Universidad de Zaragoza. El proyecto ha tenido lugar dentro del grupo de investigación HOWLab.

Como conclusiones de este trabajo, se destaca la realización de las siguientes tareas:

- Análisis y selección de componentes del sensor inteligente.
- Estudio del microcontrolador con comunicaciones embebidas LoPy4 con una aplicación de monitorización tipo, en lenguaje Micropython.
- Desarrollo en lenguaje C y ensamblador de una solución más adecuada en términos de consumo energético, programando directamente el microcontrolador interno del dispositivo LoPy4: el ESP32.
- Diseño de un algoritmo de monitorización constante, minimizando la pérdida de datos de movimiento, haciendo frente a las limitaciones y restricciones del *hardware* utilizado, especialmente en almacenamiento.
- Elaboración de un procedimiento de configuración del sensor inteligente en el mismo momento de despliegue del *wearable*, permitiendo así su utilización en diferentes escenarios sin necesidad de acceder al código fuente.
- Construcción de dos prototipos *hardware*, uno de ellos más compacto en forma de PCB, diseñado con el *software* CircuitMaker.
- Implantación de la infraestructura WiFi en la Facultad de Veterinaria.
- Estudio de protocolos y tecnologías típicas de IoT para la subida de datos a la nube y su almacenamiento en bases de datos de series temporales.
- Realización de pruebas de funcionamiento en (1) humanos y (2) ovejas, para la obtención de datos reales de monitorización.
- Análisis e interpretación de los datos de monitorización con técnicas básicas de procesado y de *machine learning*.

6.2. Líneas de futuro

En base al trabajo realizado en este TFM, se abren varias opciones de mejora o de futura investigación en el sistema de monitorización de animales propuesto:

- Selección de variables inerciales y frecuencia de muestreo óptimos para la monitorización de ovejas de la Facultad de Veterinaria en particular.
- Programación del sensor IMU BNO055 para dotarle de cierta inteligencia por separado al microcontrolador, permitiéndole bajar el consumo cuando no haya movimiento del animal, y enviando datos tan sólo cuando se detecte actividad.
- Desarrollo de un algoritmo de extracción de características y aprendizaje no supervisado de *machine learning* más complejo, que permita agrupar con eficacia los datos en un mayor número de categorías. Implementación del algoritmo en el propio sensor inteligente.
- Utilización de la tecnología LoRa del dispositivo LoPy4 para envío de los datos en otros escenarios.
- Análisis de la temperatura y presión medidas por el BME680, para encontrar el efecto de estos factores en el comportamiento animal.
- Implementación del sensor inteligente en diferentes escenarios, como por ejemplo la monitorización de vacas en el Pirineo Aragonés.

Anexo I. Variables inerciales de un sensor IMU

En este anexo se pretende dar una visión general de las variables físicas medidas por los sensores IMU, que están formados normalmente por (1) giróscopos, (2) acelerómetros y (3) magnetómetros [40].

1. Un giróscopo mide la velocidad angular en radianes por segundo. Típicamente, suelen ser de 3-DOF, puesto que miden 3 ejes ortogonales: X, Y, Z.

   La medida de un giróscopo se ve afectada por un *bias* dependiente de la temperatura, y por el ruido introducido por el mero proceso de medición.

   $$\widetilde{\omega} = \omega + b + \eta$$

   Donde $\widetilde{\omega}$ es la velocidad angular medida, ω la velocidad angular real, b el *bias*, y η el ruido. Mediante el proceso de calibración del giróscopo se puede estimar el *bias*, aunque puede ir variando lentamente en lo que se conoce como *drifting*. Sin embargo, puede contrarrestarse mediante un algoritmo de fusión adecuado, como los que se dan en los sensores IMU.

2. Un acelerómetro mide la aceleración lineal en metros por segundo al cuadrado. La aceleración lineal es una combinación de la fuerza de la gravedad y otras fuerzas externas, como por ejemplo el movimiento, precisamente lo que se desea medir.

   A diferencia del giróscopo, el acelerómetro no se ve afectado por el *drifting*, aunque sí que sigue estando presente el ruido aditivo del proceso de medida.

   $$\tilde{a} = a^g + a^l + \eta$$

   Donde $\tilde{a}$ es la aceleración lineal medida, a^g la fuerza de la gravedad, a^l las fuerzas externas y η el ruido. De nuevo, suelen ser de 3-DOF. En ausencia de movimiento, el valor medido en un eje por un acelerómetro apuntando en esa misma dirección debería ser 9.81 m/s^2.

3. Un magnetómetro mide la fuerza del campo magnético en una dirección. Las unidades son micro-Teslas. El principal inconveniente es que son muy sensibles a las distorsiones de campo magnético a su alrededor, producidas por cualquier tipo de dispositivo. Por ello, es fundamental una calibración efectiva. También suelen medir tres ejes ortogonales 3-DOF.

Mediante el algoritmo de fusión de los sensores IMU, es posible convertir en cuaternios las medidas realizadas por los sensores acelerómetro, giróscopo y magnetómetro. De esta forma se puede obtener la orientación absoluta del objeto en cuestión.

Los cuaternios permiten definir la rotación de un objeto en el espacio. Están definidos por cuatro coeficientes: un escalar y tres vectores -que corresponden a la parte imaginaria del cuaternio como número complejo, teniendo diferentes unidades cada uno de ellos-.

$$q = q_w + iq_x + jq_y + kq_z$$

Es posible transformar los cuaternios en una matriz de rotación que define la orientación de un objeto de la siguiente forma:

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} q_w^2 + q_x^2 - q_y^2 - q_z^2 & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y \\ 2q_xq_y + 2q_wq_z & q_w^2 - q_x^2 + q_y^2 - q_z^2 & 2q_yq_z - 2q_wq_x \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{pmatrix}$$

Anexo II. Instalación de ESP-IDF y configuración de proyecto en Eclipse

Para facilitar el desarrollo de aplicaciones de IoT usando *hardware* basado en el microcontrolador ESP32, Espressif proporciona el *toolchain* ESP-IDF (*Espressif IoT Development Framework*), el cual se ha utilizado en este Trabajo Fin de Máster.

En este anexo se muestra el proceso de instalación de este *software* paso por paso, en un ordenador MacBook Pro 2017 con la versión 10.14.6 del sistema operativo macOS Mojave. Los comandos mencionados deben ser introducidos en la aplicación Terminal.

Instalación del toolchain

El *toolchain* de ESP32 contiene los programas necesarios para compilar y construir las aplicaciones.

1. Descargar el *toolchain* del microcontrolador ESP32 para macOS en el siguiente enlace https://dl.espressif.com/dl/xtensa-esp32-elf-osx-1.22.0-80-g6c4433a-5.2.0.tar.gz
   y extraerlo en el directorio /esp.

   ```
   mkdir -p ~/esp
   cd ~/esp
   tar -xzf ~/Downloads/xtensa-esp32-elf-osx-1.22.0-80-g6c4433a-5.2.0.tar.gz
   ```

2. Actualizar la variable PATH en el fichero de perfil *profile* de arranque en UNIX, de tal forma que el *toolchain* esté disponible para cada sesión. Para ello se abre el fichero desde el terminal con *nano* y se añade la siguiente línea.

   ```
   nano .bash_profile
   export PATH=/Users/Fer/esp/xtensa-esp32-elf/bin:$PATH
   ```

   El intérprete de Python que utiliza ESP-IDF será el que viene instalado de fábrica en macOS, correspondiente en este caso a la versión 2.7. Para evitar problemas con la instalación de paquetes posterior se recomienda eliminar en el fichero *profile* todo lo relativo a plataformas como Anaconda, que hayan podido instalar otras versiones y/o paquetes de Python.

Obtención del API y librerías de ESP-IDF

1. En primer lugar, se debe instalar en el ordenador el *software* de gestión de paquetes *pip*

   ```
   sudo easy_install pip
   ```

2. Obtener una copia local del *toolchain* ESP-IDF copiando el repositorio de Github en el directorio deseado.

```
cd ~/esp
git clone -b v3.3.1 --recursive https://github.com/espressif/esp-idf.git
```

3. A continuación, abrir el fichero *profile* y añadir la variable IDF_PATH de tal forma que el *toolchain* pueda acceder a ESP-IDF, y por tanto compilar las aplicaciones.

```
nano .bash_profile
export IDF_PATH=~/esp/esp-idf
```

4. Los paquetes de Python requeridos por ESP-IDF se encuentran en el fichero *requirements.txt* del IDF_PATH. Instalarlos mediante el siguiente comando:

```
python -m pip install --user -r $IDF_PATH/requirements.txt
```

Una vez instalado ESP-IDF mediante el procedimiento previamente explicado se pueden empezar a desarrollar aplicaciones mediante un entorno de desarrollo que permita escribir programas en C, como por ejemplo Eclipse, que se ha utilizado en este Trabajo Fin de Máster.

Configuración de un proyecto de ESP-IDF en Eclipse

Nota: este procedimiento pretende servir como ejemplo ilustrativo, debiendo utilizar cada usuario los paths de los directorios adecuados.

1. Copiar y pegar la carpeta de un proyecto tipo de ESP-IDF en el directorio en el que se vaya a trabajar.
2. En Eclipse: `Import – C/C++ – Existing Code as Makefile Project → Next`
3. En la sección *Existing Code Location* poner la carpeta del proyecto tipo que se haya seleccionado. Marcar la opción Cross GCC y hacer click en *Finish*. Ahora ya se puede ver la carpeta del proyecto en Eclipse.
4. Ir a las propiedades del proyecto haciendo click derecho e ir a *C/C++ Build – Environment*. Añadir lo siguiente:
    a. `BATCH_BUILD → 1`
    b. `IDF_PATH → /Users/Fer/esp/esp-idf`
    c. `PYTHONPATH → /opt/local/lib/python2.7/site-packages`

    Editar el campo *PATH* para que quede de la siguiente forma:

    d. `PATH → /bin:/usr/bin:/usr/sbin:/sbin:/Users/Fer/esp/xtensa-esp32-elf/bin`

5. Ir ahora a *C/C++ General – Preprocessor Include Paths – Providers*. En *CDT Cross GCC Built-in compiler settings* cambiar la sección *Command to get compiler specs* a:

```
xtensa-esp32-elf-gcc ${FLAGS} -std=c++11 -E -P -v -dD "${INPUTS}"
```

En *CDT GCC Build Output Parser* cambiar *Compiler command pattern* a

```
xtensa-esp32-eld-(gcc|g\+\+|c\+\+|cc|cpp|clang)
```

6. En *C/C++ General – Indexer* marcar la casilla de '*Enable project specific settings*', y

desmarcar la de *'Allow heuristic resolution of includes'*

7. Por ultimo, en *C/C++ Build – Behavior* marcar *'Enable parallel build'*.
8. Aplicar los cambios y guardar la configuración.

Anexo III. Programación del ESP32 contenido en LoPy4

Para poder programar directamente el ESP32 contenido en la LoPy4 en lenguaje C es necesario poner la LoPy4 en modo *Firmware Update* siempre que se cargue un programa. El procedimiento es el siguiente:

1. Colocar un *jumper* entre el pin P2 de la placa de expansión y el pin de GND.
2. Pulsar botón de reset de la LoPy4 para entrar en modo *Firmware Update*.
3. Cargar el programa.
4. Desconectar *jumper*.
5. Volver a pulsar RESET para salir del modo *Firmware Update*.

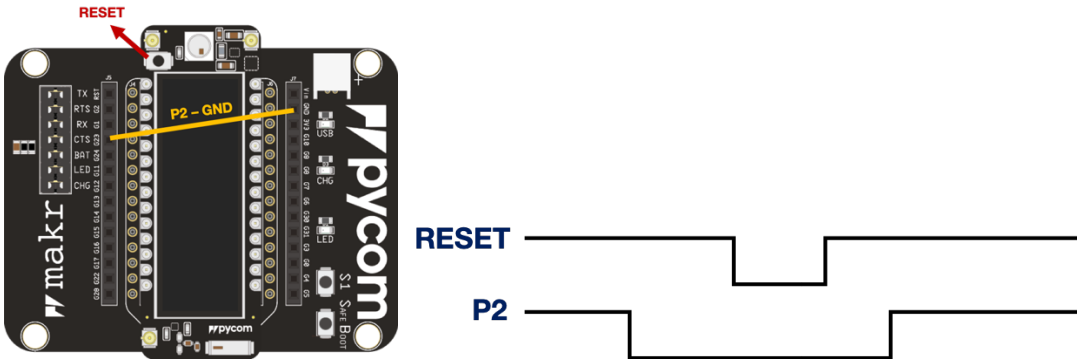

Figura 74. Mecanismo para llevar la LoPy4 a modo Firmware Update

Al realizar este proceso, será imposible cargar programas en la LoPy4 mediante el IDE Visual Studio Code, puesto que se han borrado las librerías de Micropython. Sin embargo, esta acción es reversible tantas veces como se desee.

Para ello se debe instalar el programa *Pycom Upgrade*, disponible en el siguiente enlace https://docs.pycom.io/gettingstarted/installation/firmwaretool/. Con esta herramienta simplemente se selecciona el puerto en el que se encuentra la LoPy4 y se indica la versión del *firmware* que se quiera instalar. Una vez hecho esto ya se puede volver a trabajar con Micropython.

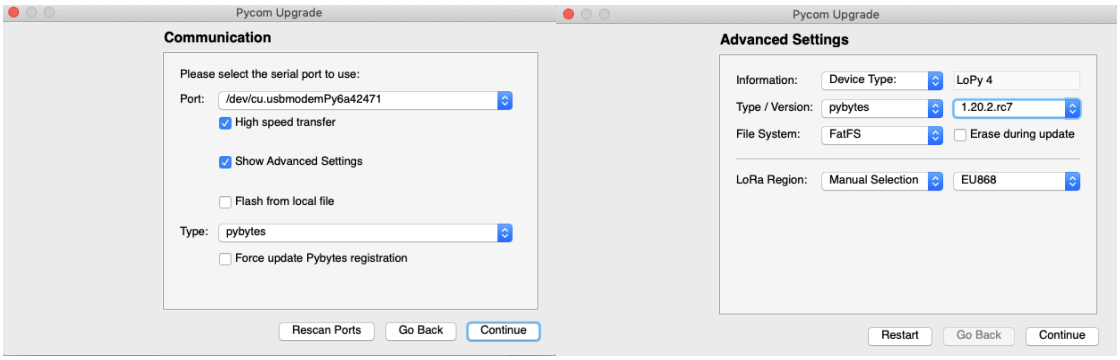

Figura 75. Procedimiento para recuperar Micropython en LoPy4 con Pycom Upgrade

Anexo IV. Comparativa sensores IMU BNO055-BNO085

	BNO055	**BNO085**
Precio/unidad	10.13€	10.76€
Especificaciones eléctricas	Idénticas (mismo *hardware*)	
Interfaces digitales	I2C, UART	I2C, UART, SPI
Rangos de medida sensores	Acelerómetro (m/s^2) : ±2g, ±4g, ±8g, ±16g Giróscopo (dps): ±125, ±250, ±500, ±1000, ±2000	Acelerómetro (m/s^2) : ±8g Giróscopo (dps): ±2000
Resolución sensores	Acelerómetro: 14 bits Giróscopo: 16 bits Magnetómetro (x/y/z): 13/13/15 bits	Acelerómetro: 12 bits Giróscopo: 16 bits Magnetómetro: sin información
Mecanismo comunicación IMU-MCU	*Repeated Start Condition*	SHTP (*Sensor Hub Transport Protocol*)
Consumo#	36.9 mW	35.25 mW
ODR (*Output Data Rate*)	100 Hz	400 Hz
Modos de operación en modo fusión	4	7
Precisión y sensibilidad del sensado	Similar (mismo *hardware*)	
Modos de potencia	Baja potencia, suspensión, normal	Normal
Calibración	Automática. Se ejecuta constantemente y no se puede desactivar	Estática y Dinámica
Errores en medidas	Mayor drift en la medida de la orientación debido a la calibración automática	-
Funcionalidad adicional	Permite trabajar en modo no fusión	Predicción de la orientación en 20-30 *ms* futuros Información al MCU de la precisión actual del IMU Compatible con Android Sistema de clasificación: estabilidad, pulsado, pasos, intensidad de actividad, agitación

Consumo en modo fusión con 9DOF y 100 Hz de ODR

Tabla 5. Comparación BNO055 vs BNO085

El BNO055 presenta una serie de ventajas respecto al BNO085. En primer lugar, el precio. Si bien es cierto que los precios de los circuitos integrados son parecidos, si se tiene en cuenta el circuito recomendado por el fabricante esta diferencia se incrementa. Por ejemplo -aunque este precio no sea significativo puede servir para comparar- un mismo proveedor vende el módulo del BNO055 con el circuito recomendado por 17.5€, y el módulo del BNO085 por 28.8€.

Otro punto a favor del BNO055 es la posibilidad de trabajar en modo no fusión, lo cual es de especial interés cuando se requieren consumos muy bajos, ya que de esta forma pueden apagarse los sensores que no se estén utilizando.

La comunicación entre el microcontrolador y el BNO055 es la ventaja principal de este circuito integrado. El método de "*repeated start condition*" para la comunicación I2C es estándar y utilizado en la mayor parte de las aplicaciones. Por otro lado, el BNO085 utiliza el protocolo propio SHTP que, aunque supone un menor *overhead*, dificulta la programación del microcontrolador y una posible sustitución futura del IMU por otro de un fabricante distinto.

Por otro lado, el BNO085 consigue suplir el problema de *drifting* en el cálculo de la orientación que tiene el BNO055. Este *drifting* se debe a que en la calibración automática que se realiza en segundo plano no se soluciona el *skew* -no ortogonalidad de los tres ejes- del IMU, que sí que se ve solventado con la calibración estática del BNO085.

Además, el BNO085 dispone de un mayor número de modos de fusión en los que puede trabajar, permitiendo una mayor flexibilidad y adaptación a cada aplicación. Entre estos modos destaca el "*Gyro rotation vector prediction*", que introduce de forma novedosa la predicción en los sensores IMU, pudiendo anticipar la rotación en los próximos 20-30 ms y, por tanto, mejorando la latencia.

Teniendo todo lo expuesto en cuenta se ha decidido optar para este TFM por el BNO055, desarrollado íntegramente por Bosch Sensortec. A pesar de que el BNO085 permite eliminar el pequeño error en el cálculo de la orientación del BNO055, la calidad y la precisión del dato son suficientes para una aplicación de seguimiento y caracterización del movimiento de animales.

Adicionalmente, los sistemas de clasificación del contexto del BNO085 diseñados para la compatibilidad con dispositivos Android no son de interés para este TFM, puesto que el microcontrolador utilizado será el ESP32 presente en la LoPy4.

Anexo V. Medidas de consumo con ADC-20

El ADC-20 es un registrador de datos del fabricante Pico Technology. Tiene una resolución de 20 bits y 12 canales: 4 diferenciales y 8 de terminación única. La frecuencia máxima de conversión que permite es de 10 Hz, o lo que es lo mismo, cada 100 ms.

Con la *Terminal Board* se pueden construir circuitos para realizar las medidas. En este caso se han colocado resistencias para la medida de la corriente consumida por la LoPy4 en el canal 2, y para la medida de la tensión de la batería en el canal 5.

Puesto que la tensión máxima que acepta el ADC-20 es de 2.5V, y la batería de Li-Ion utilizada en este TFM tiene un voltaje de salida de aproximadamente 4.2V, se ha colocado un divisor resistivo para reducirla a la mitad.

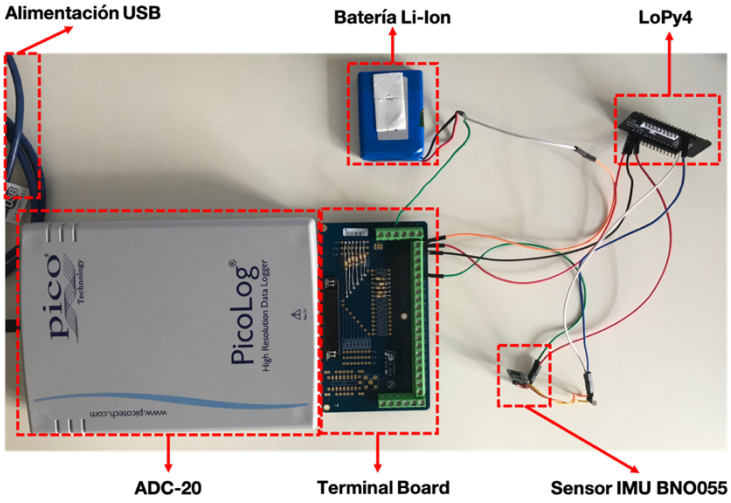

Figura 76. Montaje para realizar medidas de consumo con ADC-20

Las conversiones realizadas por el ADC-20 quedan registradas a través del *software* PicoLog 6, de descarga gratuita. Para ello, es necesario configurar los dos canales con el intervalo de muestreo adecuado -en este caso el mínimo posible, de 100ms-, y seleccionar si se desea que se realice alguna operación matemática en la representación. En el caso del canal de voltaje, habrá que multiplicar el resultado por dos para compensar el divisor resistivo.

Figura 77. Configuración del software PicoLog 6 para realización de medidas con ADC-20

Anexo VI. Código del ULP (ensamblador)

main.S

```
/*
 * Main of the ULP program
 * Author: Fernando Herranz 699869@unizar.es
 */

#include "soc/rtc_cntl_reg.h"
#include "soc/rtc_io_reg.h"
#include "soc/soc_ulp.h"
#include "definitions.h"
#include "stack.S"

.set BNO055_ADDR,0x29 // BNO055 address

    /* Define variables, which go into .bss section (zero-initialized data) */
    .bss

    .global counter
counter: .long 0
    .global cntReadings
cntReadings: .long 0
    .global valueReady
valueReady: .long 0


    .global stack
stack:
    .skip 100
    .global stackEnd
stackEnd:
    .long 0


    /* Code goes into .text section */
    .text
    .global entry
entry:
    move r3,stackEnd

    psr // processor-state-register to get data of CPU state

    jump read_imu

    // when reached it means we are ready to wakeup the CPU and use our last read value
    move r1,cntReadings // R3 points to cntReadings
    ld r0,r1,0 // load value of cntShift in R0
    add r0,r0,1 // add 1
    st r0,r1,0 // store new value in R3
    jumpr wakeUp,ULP_BUFFER_SIZE,ge // check if R0 (register for intermediate calculations)
meets condition

    jump waitNext

wakeUp:
    // Restart cntReadings every time CPU wakes up
    move r1,cntReadings
    ld r0,r1,0
    sub r0,r0,r0 // R0=R0-R0, i.e. restart
    st r0,r1,0 // store new value

    // Value ready
    move r1,valueReady
    ld r0,r1,0
    add r0,r0,1
    st r0,r1,0

    // Wake up the SoC, end program
    wake
    // Stop the wakeup timer so it does not restart ULP
    WRITE_RTC_FIELD(RTC_CNTL_STATE0_REG, RTC_CNTL_ULP_CP_SLP_TIMER_EN, 0)

waitNext:
    halt

noNegate:
    ret
```

i2c-bno055.S (I)

```
/*
 * I2C read of BNO055 selected registers (16-bit reads)
 * Author: Fernando Herranz 699869@unizar.es
 */

#include "soc/rtc_cntl_reg.h"
#include "soc/rtc_io_reg.h"
#include "soc/soc_ulp.h"
#include "definitions.h"
#include "stack.S"

.set BNO055_ADDR,0x29 // BNO055 address

    /* Define variables, which go into .bss section (zero-initialized data) */
    .bss

    .global acc_x_result // array (each index corresponds to 2 memory addresses p.eg. 0x00 and
0x01)
acc_x_result:    .fill ULP_BUFFER_SIZE,4,0
    .global acc_y_result
acc_y_result:    .fill ULP_BUFFER_SIZE,4,0
    .global acc_z_result
acc_z_result:    .fill ULP_BUFFER_SIZE,4,0
    .global mag_x_result
mag_x_result:    .fill ULP_BUFFER_SIZE,4,0
    .global mag_y_result
mag_y_result:    .fill ULP_BUFFER_SIZE,4,0
    .global mag_z_result
mag_z_result:    .fill ULP_BUFFER_SIZE,4,0
    .global gyr_x_result
gyr_x_result:    .fill ULP_BUFFER_SIZE,4,0
    .global gyr_y_result
gyr_y_result:    .fill ULP_BUFFER_SIZE,4,0
    .global gyr_z_result
gyr_z_result:    .fill ULP_BUFFER_SIZE,4,0
    .global qua_w_result
qua_w_result:    .fill ULP_BUFFER_SIZE,4,0
    .global qua_x_result
qua_x_result:    .fill ULP_BUFFER_SIZE,4,0
    .global qua_y_result
qua_y_result:    .fill ULP_BUFFER_SIZE,4,0
    .global qua_z_result
qua_z_result:    .fill ULP_BUFFER_SIZE,4,0

    .global myOffset
myOffset: .long 0
    .global myCounter
myCounter: .long 0

    /* Code goes into .text section */
    .text

.global read_imu  // Dont touch R3 pointer, linked to stack
read_imu:
    // ACC_DATA_X: start-WR slave add-WR reg add-start-WR slave add-READ16-stop
    move r1,BNO055_ADDR
    push r1
    move r1,0x08
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,acc_x_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // ACC_DATA_Y
    move r1,BNO055_ADDR
    push r1
    move r1,0x0A
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0
```

i2c-bno055.S (II)

```
    move r2,acc_y_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // ACC_DATA_Z
    move r1,BNO055_ADDR
    push r1
    move r1,0x0C
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,acc_z_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // MAG_DATA_X
    move r1,BNO055_ADDR
    push r1
    move r1,0x0E
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,mag_x_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // MAG_DATA_Y
    move r1,BNO055_ADDR
    push r1
    move r1,0x10
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,mag_y_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // MAG_DATA_Z
    move r1,BNO055_ADDR
    push r1
    move r1,0x12
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,mag_z_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // GYR_DATA_X
    move r1,BNO055_ADDR
    push r1
    move r1,0x14
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0
```

i2c-bno055.S (III)

```
    move r2,gyr_x_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // GYR_DATA_Y
    move r1,BNO055_ADDR
    push r1
    move r1,0x16
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,gyr_y_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // GYR_DATA_Z
    move r1,BNO055_ADDR
    push r1
    move r1,0x18
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,gyr_z_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // QUATERNION W
    move r1,BNO055_ADDR
    push r1
    move r1,0x20
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,qua_w_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // QUATERNION X
    move r1,BNO055_ADDR
    push r1
    move r1,0x22
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,qua_x_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // QUATERNION Y
    move r1,BNO055_ADDR
    push r1
    move r1,0x24
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0
```

i2c-bno055.S (IV)

```
    move r2,qua_y_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0

    // QUATERNION Z
    move r1,BNO055_ADDR
    push r1
    move r1,0x26
    push r1
    psr
    jump read16
    add r3,r3,2 // remove call parameters from stack
    move r1,r0 // save result
    move r0,r2 // test for error
    move r2,cntReadings
    ld r0,r2,0

    move r2,qua_z_result
    add r2,r2,r0  //+1 for every new measure stored (iterative with cntReadings)
    st r1,r2,0
    ret

// Wait for r2 milliseconds
waitMs:
    wait 8000
    sub r2,r2,1
    jump doneWaitMs,eq
    jump waitMs
doneWaitMs:
    ret
```

stack.S

```
/*
 * ULP stack and subroutine macros
 */

.macro push rx
    st \rx,r3,0
    sub r3,r3,1
.endm

.macro pop rx
    add r3,r3,1
    ld \rx,r3,0
.endm

// Prepare subroutine jump, uses scratch register sr
.macro psr sr=r1 pos=.
    .set _next2,(\pos+16)
    move \sr,_next2
    push \sr
.endm

// Return from subroutine
.macro ret sr=r1
    pop \sr
    jump \sr
.endm
```

i2c.S (I)

```
/*
 * Demo of I2C ULP routines
 * Bit-banged I2C implementation, because the hardware ULP I2C support cannot read 16 bit values
by itself
 * Source: https://github.com/tomtor/ulp-i2
 * Author: Fernando Herranz 699869@unizar.es
 */

#include "soc/rtc_cntl_reg.h"
#include "soc/rtc_io_reg.h"
#include "soc/soc_ulp.h"

#include "stack.S"

.bss
i2c_started:
    .long 0

i2c_didInit:
    .long 0

.text

.global i2c_start_cond
.global i2c_stop_cond
.global i2c_write_bit
.global i2c_read_bit
.global i2c_write_byte
.global i2c_read_byte

.macro I2C_delay
    wait 10 //minimal 4.7us
.endm

.macro read_SCL // Return current level of SCL line, 0 or 1
    //READ_RTC_REG(RTC_GPIO_IN_REG, RTC_GPIO_IN_NEXT_S + 9, 1) // RTC_GPIO_9 == GPIO_32
    READ_RTC_REG(RTC_GPIO_IN_REG, RTC_GPIO_IN_NEXT_S + 14, 1) // RTC_GPIO_14 == GPIO_13
.endm

.macro read_SDA // Return current level of SDA line, 0 or 1
    //READ_RTC_REG(RTC_GPIO_IN_REG, RTC_GPIO_IN_NEXT_S + 8, 1) // RTC_GPIO_8 == GPIO_33
    READ_RTC_REG(RTC_GPIO_IN_REG, RTC_GPIO_IN_NEXT_S + 15, 1) // RTC_GPIO_15 == GPIO_12
.endm

.macro set_SCL // Do not drive SCL (set pin high-impedance)
    WRITE_RTC_REG(RTC_GPIO_ENABLE_W1TC_REG, RTC_GPIO_ENABLE_W1TC_S + 14, 1, 1)
.endm

.macro clear_SCL // Actively drive SCL signal low
    // Output mode
    WRITE_RTC_REG(RTC_GPIO_ENABLE_W1TS_REG, RTC_GPIO_ENABLE_W1TS_S + 14, 1, 1)
.endm

.macro set_SDA // Do not drive SDA (set pin high-impedance)
    WRITE_RTC_REG(RTC_GPIO_ENABLE_W1TC_REG, RTC_GPIO_ENABLE_W1TC_S + 15, 1, 1)
.endm

.macro clear_SDA // Actively drive SDA signal low
    // Output mode
    WRITE_RTC_REG(RTC_GPIO_ENABLE_W1TS_REG, RTC_GPIO_ENABLE_W1TS_S + 15, 1, 1)
.endm

i2c_start_cond:
    move r1,i2c_didInit
    ld r0,r1,0
    jumpr didInit,1,ge
    move r0,1
    st r0,r1,0
// set GPIO to pull low when activated
    WRITE_RTC_REG(RTC_GPIO_OUT_REG, RTC_GPIO_OUT_DATA_S + 14, 1, 0)
    WRITE_RTC_REG(RTC_GPIO_OUT_REG, RTC_GPIO_OUT_DATA_S + 15, 1, 0)
didInit:
    move r2,i2c_started
    ld r0,r2,0
    jumpr not_started,1,lt
// if started, do a restart condition
// set SDA to 1
    set_SDA
    I2C_delay
    set_SCL
clock_stretch:
    read_SCL
    jumpr clock_stretch,1,lt
    I2C_delay
```

i2c.S (II)

```
not_started:
// SCL is high, set SDA from 1 to 0.
    clear_SDA
    I2C_delay
    clear_SCL
    move r0,1
    st r0,r2,0
    ret

i2c_stop_cond:
// set SDA to 0
    clear_SDA
    I2C_delay
    set_SCL

clock_stretch_stop:
    read_SCL
    jumpr clock_stretch_stop,1,lt
// Stop bit setup time, minimum 4us
    I2C_delay
// SCL is high, set SDA from 0 to 1
    set_SDA
    I2C_delay
    move r2,i2c_started
    move r0,0
    st r0,r2,0
    ret

// Write a bit to I2C bus
i2c_write_bit:
    jumpr bit0,1,lt
    set_SDA
    jump bit1
bit0:
    clear_SDA
bit1:
// SDA change propagation delay
    I2C_delay
// Set SCL high to indicate a new valid SDA value is available
    set_SCL
// Wait for SDA value to be read by slave, minimum of 4us for standard mode
    I2C_delay

clock_stretch_write:
    read_SCL
    jumpr clock_stretch_write,1,lt
    // Clear the SCL to low in preparation for next change
    clear_SCL
    ret

// Read a bit from I2C bus
i2c_read_bit:
// Let the slave drive data
    set_SDA
// Wait for SDA value to be written by slave, minimum of 4us for standard mode
    I2C_delay
// Set SCL high to indicate a new valid SDA value is available
    set_SCL

clock_stretch_read:
    read_SCL
    jumpr clock_stretch_read,1,lt
// Wait for SDA value to be written by slave, minimum of 4us for standard mode
    I2C_delay
// SCL is high, read out bit
    read_SDA
// Set SCL low in preparation for next operation
    clear_SCL
    ret // bit in r0
// Write a byte to I2C bus. Return 0 if ack by the slave.
i2c_write_byte:
    stage_rst
next_bit:
    and r0,r2,0x80
    psr
    jump i2c_write_bit
    lsh r2,r2,1
    stage_inc 1
    jumps next_bit,8,lt

    psr
    jump i2c_read_bit
    ret // nack
```

i2c.S (III)

```
// Read a byte from I2C bus
i2c_read_byte:
    push r2
    move r2,0
    stage_rst
next_bit_read:
    psr
    jump i2c_read_bit
    lsh r2,r2,1
    or r2,r2,r0
    stage_inc 1
    jumps next_bit_read,8,lt
    pop r0
    psr
    jump i2c_write_bit
    move r0,r2
    ret
```

i2c-util.S (I)

```
/*
 * I2C BNO055 routines
 * START - WR slave address - WR register address - RESTART - WR slave address - READ16 - STOP
 * Source: https://github.com/tomtor/ulp-i2c
 * Author: Fernando Herranz 699869@unizar.es
 */
#include "soc/rtc_cntl_reg.h"
#include "soc/rtc_io_reg.h"
#include "soc/soc_ulp.h"
#include "stack.S"

.text
write_intro:
    psr
    jump i2c_start_cond // send START
    ld r2,r3,16 // write Slave Address
    lsh r2,r2,1
    psr
    jump i2c_write_byte
    jumpr popfail,1,ge
    ld r2,r3,12 // write Register Address
    psr
    jump i2c_write_byte
    jumpr popfail,1,ge
    ret

.global write8
write8:
    psr
    jump write_intro
write_b:
    ld r2,r3,8 // data byte
    psr
    jump i2c_write_byte
    jumpr fail,1,ge
    psr
    jump i2c_stop_cond
    move r2,0 // Ok
    ret

.global write16
write16:
    psr
    jump write_intro
    ld r2,r3,8 // data byte 1
    rsh r2,r2,8
    psr
    jump i2c_write_byte
    jumpr fail,1,ge
    jump write_b
read_intro:
    psr
    jump i2c_start_cond // send START
    ld r2,r3,16 // write Slave Address
    lsh r2,r2,1
    psr
    jump i2c_write_byte
```

i2c-util.S (II)

```
    jumpr popfail,1,ge

    ld r2,r3,12 // write Register Address
    psr
    jump i2c_write_byte
    jumpr popfail,1,ge

    psr
    jump i2c_start_cond // send RESTART

    ld r2,r3,16 // write Slave Address
    lsh r2,r2,1
    or r2,r2,1 // Address Read
    psr
    jump i2c_write_byte
    jumpr popfail,1,ge

    ret
popfail:
    pop r1 // pop caller return address
    move r2,1
    ret


.global read8
read8:
    psr
    jump read_intro

    move r2,1 // last byte
    psr
    jump i2c_read_byte
    push r0

    psr
    jump i2c_stop_cond

    pop r0

    move r2,0 // OK
    ret
fail:
    move r2,1
    ret

.global read16
read16:
    psr
    jump read_intro //start-WR slave add-WR reg add-start-WR slave add

    move r2,0
    psr
    jump i2c_read_byte
    push r0 // save result of i2c_read_byte (R0) in the stack

    move r2,1 // last byte
    psr
    jump i2c_read_byte
    push r0 // save result of i2c_read_byte (R0) in the stack
    // 2 Bytes read from i2c so far
    psr
    jump i2c_stop_cond // send STOP

    pop r0 // save first byte (right part)
    pop r2
    lsh r2,r2,8 // R2 = R2<<8bits
    or r2,r2,r0 // R2 = R2 OR R0, where R0 is the value of the 8 bits, and R2 shows if it is the
1 or 2 byte
    move r0,r2

    move r2,0 // R2 to zeros to prove that everything went OK
    ret
```

Anexo VII. Código de la CPU principal (C)

ulp_example_main.c (I)

```c
/*
* Author: Fernando Herranz 699869@unizar.es
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include "esp_sleep.h"
#include "nvs.h"
#include "nvs_flash.h"
#include "soc/rtc_cntl_reg.h"
#include "soc/rtc_io_reg.h"
#include "soc/sens_reg.h"
#include "soc/soc.h"
#include "driver/gpio.h"
#include "driver/rtc_io.h"
#include "esp32/ulp.h"
#include "sdkconfig.h"
#include "definitions.h"
#include "ulp_main.h"
#include "bno055/bno055.h"
#include "bno_i2c_config.h"
#include "wifi_app.h"
#include "cJSON.h"
#include "iot_button.h"
#include <wifi_provisioning/manager.h>
#include <wifi_provisioning/scheme_softap.h>
#include "uart_config.h"
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event_loop.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "freertos/queue.h"
#include "freertos/event_groups.h"
#include "lwip/sockets.h"
#include "lwip/dns.h"
#include "lwip/netdb.h"
#include "esp_log.h"
#include "mqtt_client.h"
#include "esp_sntp.h"

/*************** Global variables ***************/
struct stBuffer {
     int16_t reg0;
     int16_t reg1;
     int16_t reg2;
     int16_t reg3;
     int16_t reg4;
     int16_t reg5;
     int16_t reg6;
     int16_t reg7;
     int16_t reg8;
     int16_t reg9;
     int16_t reg10;
     int16_t reg11;
     int16_t reg12;
};
extern const uint8_t ulp_main_bin_start[] asm("_binary_ulp_main_bin_start");
extern const uint8_t ulp_main_bin_end[]   asm("_binary_ulp_main_bin_end");
const gpio_num_t gpio_scl = GPIO_NUM_13;  //Lopy:P10
const gpio_num_t gpio_sda = GPIO_NUM_12;  //Lopy:P9
struct stBuffer* tx_buffer;
struct stBuffer* wifi_buffer;
struct stBuffer* temp_buffer;
int bufferPointerCore1 = 0;
uint32_t currentTime = 0;
TaskHandle_t taskHandle;
int mqttIsFinished = 0;
int wifiIsConnected = 0;
int publishIsFinished = 0;
int alreadyConnected = 0;
EventGroupHandle_t wifi_event_group;
/*************** Global variables ***************/

/*************** RTC Fast Memory ***************/
/* 8 KB available - Dynamic allocation not possible */
RTC_FAST_ATTR struct stBuffer rtc_buffer[RTC_FAST_MEM_SIZE / (13 * 2)]; //maximum possible
RTC_FAST_ATTR int bufferPointer = 0;
RTC_FAST_ATTR int bootCounter = 0;
RTC_FAST_ATTR char URN_DEV_64[30] = "urn:dev:mac:";
/*************** RTC Fast Memory ***************/

/*************** Functions ***************/
cJSON * senML_builder(int jsonPointer, uint32_t currentTime, struct stBuffer *buffer);
void publish_json(esp_mqtt_client_handle_t client, struct stBuffer *buffer);
int write_buffer_from_ulp(struct stBuffer *buffer, int bufferSize, int *bufferPointer);
void copy_buffer(struct stBuffer *srcBuffer, struct stBuffer *dstBuffer, int ceiling);
```

ulp_example_main.c (II)

```
static void init_ulp_program();
void get_MAC();
static void mqtt_app_start(void);
static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event);
static void wifi_init(void);
static esp_err_t wifi_event_handler(void *ctx, system_event_t *event);
static void button_press_3sec_cb(void *arg);
void bno_awake_task(void *pvParameter);
uint32_t computeTimeNow(void);
/*************** Functions ***************/

/*************** Configuration parameters ***************/
extern int SAMPLE_PERIOD;
extern int TX_BUFFER_SIZE;
extern char SAMPLE_VARIABLES[3];
extern int NUM_SAMPLE_VAR;
extern int SENSOR_MODE; //0:INTENSIVE, 1:IF MOVEMENT, 2:SMART
/*************** Configuration parameters ***************/

void app_main()
{
     /* Just once after reset */
     if (bootCounter == 0){
          // Configure factory reset with P2 to GND (3 sec)
          button_handle_t btn_handle = iot_button_create(0,0);
          iot_button_add_custom_cb(btn_handle, 3, button_press_3sec_cb, NULL);

          // Configure sensor: WiFi + parameters
          get_credentials();
          nvs_open("storage", NVS_READWRITE, &my_nvs_handle);
          uart_init();
          config_with_uart();
          read_config();

          // Get @MAC and configure BNO
          get_MAC();
          miBNO_config(SENSOR_MODE, SAMPLE_VARIABLES);
          bootCounter++;

          // Run ULP only if SENSOR_MODE = INTENSIVE
          if (SENSOR_MODE == 0){
               init_ulp_program();
               ESP_ERROR_CHECK( esp_sleep_enable_ulp_wakeup() );
               esp_err_t err = ulp_run((&ulp_entry - RTC_SLOW_MEM) / sizeof(uint32_t));
               ESP_ERROR_CHECK(err);
          }
          else if (SENSOR_MODE == 1){
               ESP_ERROR_CHECK (esp_sleep_enable_ext1_wakeup(0x04000000, ESP_EXT1_WAKEUP_ANY_HIGH));
          }

        // Choose sleep mode
        if (TX_BUFFER_SIZE > RTC_FAST_MEM_SIZE) {
          tx_buffer = malloc(TX_BUFFER_SIZE);
               if (tx_buffer == NULL) {
                    free(tx_buffer);
               }
          esp_light_sleep_start();
        }
        else esp_deep_sleep_start();
     }
     /* Just once after reset */


     // Loop needed as light sleep resumes at the same point
     while(1){
          read_config();
          esp_sleep_wakeup_cause_t cause = esp_sleep_get_wakeup_cause();

          // If waken up by BNO (SENSOR_MODE = IF_MOVEMENT)
          if (cause == ESP_SLEEP_WAKEUP_EXT1) {
               init_ulp_program();
               ESP_ERROR_CHECK( esp_sleep_enable_ulp_wakeup() );
               // Start the ULP program
               esp_err_t err = ulp_run((&ulp_entry - RTC_SLOW_MEM) / sizeof(uint32_t));
               ESP_ERROR_CHECK(err);
          }

          // If waken up by ULP
          else if (cause == ESP_SLEEP_WAKEUP_ULP) {
               int isBufferFull;
               if (bufferPointer == TX_BUFFER_SIZE/(NUM_SAMPLE_VAR*2)) isBufferFull = 1; //buffer filled by
wifi_buffer while we were sending data
               else {
                    if (TX_BUFFER_SIZE > RTC_FAST_MEM_SIZE) isBufferFull = write_buffer_from_ulp(tx_buffer,
TX_BUFFER_SIZE/(NUM_SAMPLE_VAR*2), &bufferPointer);
                    else isBufferFull = write_buffer_from_ulp(rtc_buffer, TX_BUFFER_SIZE/(NUM_SAMPLE_VAR*2),
&bufferPointer);
               }

               // If buffer full, then send. If not, go to sleep again and restart ULP
               if (isBufferFull == 1){
                    wifi_buffer = malloc(WIFI_BUFFER_SIZE);
                    if (wifi_buffer == NULL) esp_restart();

                    //copy_buffer(temp_buffer, wifi_buffer, bufferPointerCore1);
                    bufferPointer = 0;
                    mqttIsFinished = 0;
                    // Run ULP and save measurements while sending WiFi through bno_awake_task() in Core 1
```

ulp_example_main.c (III)

```
                    ulp_valueReady = 0;
                    ESP_ERROR_CHECK( esp_sleep_enable_ulp_wakeup() );
                    esp_err_t err = ulp_run((&ulp_entry - RTC_SLOW_MEM) / sizeof(uint32_t));
                    ESP_ERROR_CHECK(err);
                    xTaskCreatePinnedToCore(&bno_awake_task, "bno_awake_task", 4096, NULL, 10, &taskHandle, 1);
                    //printf("wifiIsConnected: %d \n", wifiIsConnected);
                    // Connect to WiFi and publish buffer via MQTT
                    nvs_flash_init();
                    wifi_init();
                    while (!wifiIsConnected) vTaskDelay(500 / portTICK_PERIOD_MS);
                    //printf("Empieza start \n");
                    mqtt_app_start();
                    // Wait here until everything has been published
                    while (!mqttIsFinished) vTaskDelay(500 / portTICK_PERIOD_MS);
                    free(wifi_buffer);
                    //refill a partir de CntReadings-1 !!!!!!!
                    //printf("Bucle out \n\n");

                    esp_wifi_stop();
               }

               // Go to sleep again
               if (SENSOR_MODE == 0){
                    ESP_ERROR_CHECK( esp_sleep_enable_ulp_wakeup() );
                    esp_err_t err = ulp_run((&ulp_entry - RTC_SLOW_MEM) / sizeof(uint32_t));
                    ESP_ERROR_CHECK(err);
               }
               else if (SENSOR_MODE == 1){
                    ESP_ERROR_CHECK (esp_sleep_enable_ext1_wakeup(0x04000000, ESP_EXT1_WAKEUP_ANY_HIGH));
               }
          }

          // Choose sleep mode
          if (TX_BUFFER_SIZE > RTC_FAST_MEM_SIZE) esp_light_sleep_start();
          else esp_deep_sleep_start();

          vTaskDelay(100 / portTICK_PERIOD_MS);
     }
}

/*
 * Initialize ULP task according to selected config
*/
static void init_ulp_program()
{
    rtc_gpio_init(gpio_scl);
    rtc_gpio_set_direction(gpio_scl, RTC_GPIO_MODE_INPUT_ONLY);
    rtc_gpio_init(gpio_sda);
    rtc_gpio_set_direction(gpio_sda, RTC_GPIO_MODE_INPUT_ONLY);

    esp_err_t err = ulp_load_binary(0, ulp_main_bin_start,
            (ulp_main_bin_end - ulp_main_bin_start) / sizeof(uint32_t));
    ESP_ERROR_CHECK(err);

    // ULP wakeup period (default 100 msec.)
    REG_SET_FIELD(SENS_ULP_CP_SLEEP_CYC0_REG, SENS_SLEEP_CYCLES_S0, ULP_COUNTER);
}

/*
 * Performs factory reset if P2 to GND more than 3 sec
 * This should be done during the config period, before CPU goes to sleep
 * When LoPy is slept it doesnt check buttons
*/
static void button_press_3sec_cb(void *arg)
{
nvs_flash_erase();
esp_restart();
}

/*
 * When ULP_BUFFER is full (ulp_valueReady = 1) then copy measurements in wifi_buffer
 * Task performed in Core 1 in order to avoid collisions
*/
void bno_awake_task(void *pvParameter){
     int taskCounter = 0;
     while(taskCounter < (TX_BUFFER_SIZE/(NUM_SAMPLE_VAR*2) / ULP_BUFFER_SIZE)){
          if ((ulp_valueReady & 0xFFFF) == 0x0001){
               write_buffer_from_ulp(wifi_buffer, WIFI_BUFFER_SIZE/(NUM_SAMPLE_VAR*2), &bufferPointerCore1);
               ESP_ERROR_CHECK( esp_sleep_enable_ulp_wakeup() );
               esp_err_t err = ulp_run((&ulp_entry - RTC_SLOW_MEM) / sizeof(uint32_t));
               ESP_ERROR_CHECK(err);
               ulp_valueReady = 0;
               taskCounter++;
          }
          vTaskDelay(500 / portTICK_PERIOD_MS);
     }
     vTaskDelete( taskHandle );
}

/*
 * Copy ULP_BUFFER in TX_BUFFER (if light sleep), RTC_BUFFER (if deep sleep) or WIFI_BUFFER (if awake)
 * Return 1 if buffer full, 0 if not
*/
int write_buffer_from_ulp(struct stBuffer *buffer, int bufferSize, int *bufferPointer){
     int ret = 0;
     int j = 0;
     for(int i=0; i<ULP_BUFFER_SIZE; i++){
```

ulp_example_main.c (IV)

```
          /*
           * Fill TX_BUFFER (if light sleep)or RTC_BUFFER (if deep sleep)
           */
          if(*bufferPointer+i < bufferSize){ // i.e. if buffer not full
               (&ulp_acc_x_result)[i] = ((((int16_t)(&ulp_acc_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_acc_x_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg0 = ((int16_t)(&ulp_acc_x_result)[i]);
               (&ulp_acc_y_result)[i] = ((((int16_t)(&ulp_acc_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_acc_y_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg1 = ((int16_t)(&ulp_acc_y_result)[i]);
               (&ulp_acc_z_result)[i] = ((((int16_t)(&ulp_acc_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_acc_z_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg2 = ((int16_t)(&ulp_acc_z_result)[i]);
               (&ulp_mag_x_result)[i] = ((((int16_t)(&ulp_mag_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_mag_x_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg3 = ((int16_t)(&ulp_mag_x_result)[i]);
               (&ulp_mag_y_result)[i] = ((((int16_t)(&ulp_mag_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_mag_y_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg4 = ((int16_t)(&ulp_mag_y_result)[i]);
               (&ulp_mag_z_result)[i] = ((((int16_t)(&ulp_mag_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_mag_z_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg5 = ((int16_t)(&ulp_mag_z_result)[i]);
               (&ulp_gyr_x_result)[i] = ((((int16_t)(&ulp_gyr_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_gyr_x_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg6 = ((int16_t)(&ulp_gyr_x_result)[i]);
               (&ulp_gyr_y_result)[i] = ((((int16_t)(&ulp_gyr_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_gyr_y_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg7 = ((int16_t)(&ulp_gyr_y_result)[i]);
               (&ulp_gyr_z_result)[i] = ((((int16_t)(&ulp_gyr_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_gyr_z_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg8 = ((int16_t)(&ulp_gyr_z_result)[i]);
               (&ulp_qua_w_result)[i] = ((((int16_t)(&ulp_qua_w_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_w_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg9 = ((int16_t)(&ulp_qua_w_result)[i]);
               (&ulp_qua_x_result)[i] = ((((int16_t)(&ulp_qua_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_x_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg10 = ((int16_t)(&ulp_qua_x_result)[i]);
               (&ulp_qua_y_result)[i] = ((((int16_t)(&ulp_qua_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_y_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg11 = ((int16_t)(&ulp_qua_y_result)[i]);
               (&ulp_qua_z_result)[i] = ((((int16_t)(&ulp_qua_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_z_result)[i] & 0xFF00) >> 8));
               buffer[*bufferPointer+i].reg12 = ((int16_t)(&ulp_qua_z_result)[i]);
          }
          /*
           * Fill wifi_buffer while sending wifi data
           */
          else {
               ret = 1;
               (&ulp_acc_x_result)[i] = ((((int16_t)(&ulp_acc_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_acc_x_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg0 = ((int16_t)(&ulp_acc_x_result)[i]);
               (&ulp_acc_y_result)[i] = ((((int16_t)(&ulp_acc_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_acc_y_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg1 = ((int16_t)(&ulp_acc_y_result)[i]);
               (&ulp_acc_z_result)[i] = ((((int16_t)(&ulp_acc_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_acc_z_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg2 = ((int16_t)(&ulp_acc_z_result)[i]);
               (&ulp_mag_x_result)[i] = ((((int16_t)(&ulp_mag_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_mag_x_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg3 = ((int16_t)(&ulp_mag_x_result)[i]);
               (&ulp_mag_y_result)[i] = ((((int16_t)(&ulp_mag_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_mag_y_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg4 = ((int16_t)(&ulp_mag_y_result)[i]);
               (&ulp_mag_z_result)[i] = ((((int16_t)(&ulp_mag_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_mag_z_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg5 = ((int16_t)(&ulp_mag_z_result)[i]);
               (&ulp_gyr_x_result)[i] = ((((int16_t)(&ulp_gyr_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_gyr_x_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg6 = ((int16_t)(&ulp_gyr_x_result)[i]);
               (&ulp_gyr_y_result)[i] = ((((int16_t)(&ulp_gyr_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_gyr_y_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg7 = ((int16_t)(&ulp_gyr_y_result)[i]);
               (&ulp_gyr_z_result)[i] = ((((int16_t)(&ulp_gyr_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_gyr_z_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg8 = ((int16_t)(&ulp_gyr_z_result)[i]);
               (&ulp_qua_w_result)[i] = ((((int16_t)(&ulp_qua_w_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_w_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg9 = ((int16_t)(&ulp_qua_w_result)[i]);
               (&ulp_qua_x_result)[i] = ((((int16_t)(&ulp_qua_x_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_x_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg10 = ((int16_t)(&ulp_qua_x_result)[i]);
               (&ulp_qua_y_result)[i] = ((((int16_t)(&ulp_qua_y_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_y_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg11 = ((int16_t)(&ulp_qua_y_result)[i]);
               (&ulp_qua_z_result)[i] = ((((int16_t)(&ulp_qua_z_result)[i]) &
0x00FF)<<8)|((((int16_t)(&ulp_qua_z_result)[i] & 0xFF00) >> 8));
               wifi_buffer[j].reg12 = ((int16_t)(&ulp_qua_z_result)[i]);

               j++;
               bufferPointerCore1 = j ;
          }
     }
     *bufferPointer = *bufferPointer + ULP_BUFFER_SIZE;
     if (*bufferPointer == (TX_BUFFER_SIZE / (NUM_SAMPLE_VAR*2))) ret = 1; //no leftovers
     return ret;
}
```

ulp_example_main.c (V)

```
/*
 * Copy content of one buffer in another. Optimize this function
*/
void copy_buffer(struct stBuffer *srcBuffer, struct stBuffer *dstBuffer, int ceiling){

     for(int i=0; i<ceiling; i++){
          dstBuffer[i].reg0 = srcBuffer[i].reg0;
          dstBuffer[i].reg1 = srcBuffer[i].reg1;
          dstBuffer[i].reg2 = srcBuffer[i].reg2;
          dstBuffer[i].reg3 = srcBuffer[i].reg3;
          dstBuffer[i].reg4 = srcBuffer[i].reg4;
          dstBuffer[i].reg5 = srcBuffer[i].reg5;
          dstBuffer[i].reg6 = srcBuffer[i].reg6;
          dstBuffer[i].reg7 = srcBuffer[i].reg7;
          dstBuffer[i].reg8 = srcBuffer[i].reg8;
          dstBuffer[i].reg9 = srcBuffer[i].reg9;
          dstBuffer[i].reg10 = srcBuffer[i].reg10;
          dstBuffer[i].reg11 = srcBuffer[i].reg11;
          dstBuffer[i].reg12 = srcBuffer[i].reg12;
     }
}

/*
 * Get MAC of the device and save it as URN_DEV_64 in RTC_FAST_MEMORY
*/
void get_MAC(){
     uint8_t base_mac_addr[6] = {0};
     esp_efuse_mac_get_default(base_mac_addr);
     char eui_64[16];
     sprintf(eui_64, "%02x%02x%02x%s%02x%02x%02x/", base_mac_addr[0], base_mac_addr[1], base_mac_addr[2],\
               "ffff", base_mac_addr[3], base_mac_addr[4], base_mac_addr[5]);
     strcat(URN_DEV_64,eui_64);
}

/*
 * Connect to SNTP server and get current time in Spain
 * Return time as UNIXTimestamp value
*/
uint32_t computeTimeNow(void){
     // initialize the SNTP service
     sntp_setoperatingmode(SNTP_OPMODE_POLL);
     sntp_setservername(0, "pool.ntp.org");
     sntp_init();
     // wait for the service to set the time
     time_t now;
     struct tm timeinfo;
     time(&now);
     localtime_r(&now, &timeinfo);
     while(timeinfo.tm_year < (2016 - 1900)) {
          time(&now);
          localtime_r(&now, &timeinfo);
     }
     // change the timezone to Spain
     setenv("TZ", "CET-2", 1);
     tzset();
     localtime_r(&now, &timeinfo);
     sntp_stop();
     return now;
}

/*
 * Build SenML packet. Size of each packet determined by JSON_SIZE
*/
cJSON * senML_builder(int jsonPointer, uint32_t currentTime, struct stBuffer *buffer){
    cJSON *root = NULL;
    cJSON *reg0[JSON_SIZE];
    cJSON *reg1[JSON_SIZE];
    cJSON *reg2[JSON_SIZE];
    cJSON *reg3[JSON_SIZE];
    cJSON *reg4[JSON_SIZE];
    cJSON *reg5[JSON_SIZE];
    cJSON *reg6[JSON_SIZE];
    cJSON *reg7[JSON_SIZE];
    cJSON *reg8[JSON_SIZE];
    cJSON *reg9[JSON_SIZE];
    cJSON *reg10[JSON_SIZE];
    cJSON *reg11[JSON_SIZE];
    cJSON *reg12[JSON_SIZE];
    float RTC_TIME = 0.0;
    int j = jsonPointer;

     root = cJSON_CreateArray();
     for (int i=0; i<JSON_SIZE; i++)
     {
          if (j < (TX_BUFFER_SIZE/(NUM_SAMPLE_VAR*2))){
               // Get field "t" [0 T_sample 2*T_sample......N*T_sample]
               RTC_TIME = j*(SAMPLE_PERIOD / 1000.0);

               // Array of num_registers objects. Each register is an object
               cJSON_AddItemToArray(root, reg0[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg1[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg2[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg3[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg4[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg5[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg6[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg7[i] = cJSON_CreateObject());
               cJSON_AddItemToArray(root, reg8[i] = cJSON_CreateObject());
```

ulp_example_main.c (VI)

```
                cJSON_AddItemToArray(root, reg9[i] = cJSON_CreateObject());
                cJSON_AddItemToArray(root, reg10[i] = cJSON_CreateObject());
                cJSON_AddItemToArray(root, reg11[i] = cJSON_CreateObject());
                cJSON_AddItemToArray(root, reg12[i] = cJSON_CreateObject());

                // Fill the objects (registers)
                // First one with "bn" and "bt"
                if (i == 0){
                     cJSON_AddStringToObject(reg0[i], "bn", URN_DEV_64);
                     cJSON_AddNumberToObject(reg0[i], "bt", currentTime);
                }

                cJSON_AddStringToObject(reg0[i], "n", "Ax");
                cJSON_AddNumberToObjectOpts(reg0[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg0[i], "v", (float)(buffer[j].reg0/100.0), 2);

                cJSON_AddStringToObject(reg1[i], "n", "Ay");
                cJSON_AddNumberToObjectOpts(reg1[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg1[i], "v", (float)(buffer[j].reg1/100.0), 2);

                cJSON_AddStringToObject(reg2[i], "n", "Az");
                cJSON_AddNumberToObjectOpts(reg2[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg2[i], "v", (float)(buffer[j].reg2/100.0), 2);

                cJSON_AddStringToObject(reg3[i], "n", "Mx");
                cJSON_AddNumberToObjectOpts(reg3[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg3[i], "v", (float)(buffer[j].reg3/16.0), 3);

                cJSON_AddStringToObject(reg4[i], "n", "My");
                cJSON_AddNumberToObjectOpts(reg4[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg4[i], "v", (float)(buffer[j].reg4/16.0), 3);

                cJSON_AddStringToObject(reg5[i], "n", "Mz");
                cJSON_AddNumberToObjectOpts(reg5[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg5[i], "v", (float)(buffer[j].reg5/16.0), 3);

                cJSON_AddStringToObject(reg6[i], "n", "Gx");
                cJSON_AddNumberToObjectOpts(reg6[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg6[i], "v", (float)(buffer[j].reg6/16.0), 3);

                cJSON_AddStringToObject(reg7[i], "n", "Gy");
                cJSON_AddNumberToObjectOpts(reg7[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg7[i], "v", (float)(buffer[j].reg7/16.0), 3);

                cJSON_AddStringToObject(reg8[i], "n", "Gz");
                cJSON_AddNumberToObjectOpts(reg8[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg8[i], "v", (float)(buffer[j].reg8/16.0), 3);

                cJSON_AddStringToObject(reg9[i], "n", "Qw");
                cJSON_AddNumberToObjectOpts(reg9[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg9[i], "v", (float)(buffer[j].reg9/16384.0), 7);

                cJSON_AddStringToObject(reg10[i], "n", "Qx");
                cJSON_AddNumberToObjectOpts(reg10[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg10[i], "v", (float)(buffer[j].reg10/16384.0), 7);

                cJSON_AddStringToObject(reg11[i], "n", "Qy");
                cJSON_AddNumberToObjectOpts(reg11[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg11[i], "v", (float)(buffer[j].reg11/16384.0), 7);

                cJSON_AddStringToObject(reg12[i], "n", "Qz");
                cJSON_AddNumberToObjectOpts(reg12[i], "t", RTC_TIME, 1);
                cJSON_AddNumberToObjectOpts(reg12[i], "v", (float)(buffer[j].reg12/16384.0), 7);

                j++;
            }
            else break;
        }
        return root;
};

/*
 * Handler for WiFi events
*/
static esp_err_t wifi_event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect();
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            xEventGroupSetBits(wifi_event_group, CONNECTED_BIT);
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect();
            xEventGroupClearBits(wifi_event_group, CONNECTED_BIT);
            break;
        default:
            break;
    }
    return ESP_OK;
}

/*
 * Initialize WiFi and connect to SSID saved in non-volatile NVS region
 * If WIFI_TIMEOUT completed for NUM_RETRIES times, then discard buffer and go to sleep
*/
static void wifi_init(void)
```

ulp_example_main.c (VII)

```
static void wifi_init(void)
{
     EventBits_t uxBits;
    tcpip_adapter_init();
    wifi_event_group = xEventGroupCreate();
    // Deep Sleep vs Light Sleep does make a difference here
    if (TX_BUFFER_SIZE > RTC_FAST_MEM_SIZE) esp_event_loop_set_cb(wifi_event_handler, NULL);
    else ESP_ERROR_CHECK(esp_event_loop_init(wifi_app_event_handler, NULL));
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_start());
    for (int numRetries = 1 ; numRetries <= NUM_RETRIES ; numRetries++ ){
     // Wait for WiFi connection for WIFI_TIMEOUT msec.
          uxBits = xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT, false, true,
WIFI_TIMEOUT/portTICK_PERIOD_MS);
          // if timeout expired
          if ((uxBits & CONNECTED_BIT) != CONNECTED_BIT){
               if (numRetries == NUM_RETRIES){
                    bufferPointer = 0;
                    // refill wifi_buffer cntreadings
                    ESP_ERROR_CHECK( esp_sleep_enable_ulp_wakeup() );
                    esp_err_t err = ulp_run((&ulp_entry - RTC_SLOW_MEM) / sizeof(uint32_t));
                    ESP_ERROR_CHECK(err);
                    esp_wifi_stop();
                    free(wifi_buffer);
                    vTaskDelete( taskHandle );
                    bufferPointerCore1 = 0;
                   // Choose sleep mode
                   if (TX_BUFFER_SIZE > RTC_FAST_MEM_SIZE) esp_light_sleep_start();
                   else esp_deep_sleep_start();
               }
          }
          // if succesfully connected
          else break;
    }
    wifiIsConnected = 1;
}

/*
 * When Lopy status is CONNECTED to broker, then publish buffer
 * As publish is finished, copy BNO's measurements stored in wifi_buffer in corresponding buffer
 * Tell main() that publish is finished by setting mqttIsFinished to 1
*/
static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event)
{
    esp_mqtt_client_handle_t client = event->client;
    switch (event->event_id) {
        case MQTT_EVENT_CONNECTED:
              if (TX_BUFFER_SIZE > RTC_FAST_MEM_SIZE){
                    publish_json(client, tx_buffer);
                    while((ulp_valueReady & 0xFFFF) == 0x0001) vTaskDelay(500 / portTICK_PERIOD_MS);
                    vTaskDelete( taskHandle );
                    copy_buffer(wifi_buffer, tx_buffer, bufferPointerCore1);
              }
              else {
                    publish_json(client, rtc_buffer);
                    while((ulp_valueReady & 0xFFFF) == 0x0001) vTaskDelay(500 / portTICK_PERIOD_MS);
                    vTaskDelete( taskHandle );
                    copy_buffer(wifi_buffer, rtc_buffer, bufferPointerCore1);
              }

              vTaskDelay(250 / portTICK_PERIOD_MS); //wait 250ms so there is time to publish last message
              bufferPointer = bufferPointerCore1;
              bufferPointerCore1 = 0;
              alreadyConnected = 0;
              publishIsFinished = 0;
              mqttIsFinished = 1;
              esp_mqtt_client_destroy(client);

            break;

        default:
            break;
    }
    return ESP_OK;
}

/*
 * Connect to MQTT BROKER and start mqtt handler
 * Two possibilities: with SSL (default) or without SSL
*/
static void mqtt_app_start(void)
{
     if (SSL_CONNECTION == 0) {
          const esp_mqtt_client_config_t mqtt_cfg = {
               .uri = BROKER_URI,
               .event_handle = mqtt_event_handler_cb,
               .disable_auto_reconnect = true,
          };
          esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
          //esp_mqtt_client_register_event(client, ESP_EVENT_ANY_ID, mqtt_event_handler, client);
          esp_mqtt_client_start(client);
     }
     else{
          const esp_mqtt_client_config_t mqtt_cfg = {
```

ulp_example_main.c (VIII)

```
                .uri = BROKER_URI_SSL,
                .event_handle = mqtt_event_handler_cb,
                .disable_auto_reconnect = true,
            };
            esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
            //esp_mqtt_client_register_event(client, ESP_EVENT_ANY_ID, mqtt_event_handler, client);
            esp_mqtt_client_start(client);
        }

}

/*
 * Go through all the buffer and call SenML_builder for packet creation with size JSON_SIZE
 * Sequential: Create SenML - Send - Delete from memory - Repeat until buffer empty
*/
void publish_json(esp_mqtt_client_handle_t client, struct stBuffer *buffer){
    int jsonPointer = 0;
    currentTime = computeTimeNow();
    while (jsonPointer < (TX_BUFFER_SIZE/(NUM_SAMPLE_VAR*2))){
        cJSON *request_json = senML_builder(jsonPointer, currentTime, buffer);
        char *buf = NULL;
        size_t len = 0;
        char *out = cJSON_Print(request_json);

        if (out == NULL){
            free(out);
            continue; // go to next loop iteration
        }
        len = strlen(out) + 5;
        free(out);//////
        buf = (char*)malloc(len);
        if (buf == NULL)
        {
            free(buf);
            continue; // go to next loop iteration
        }
        cJSON_PrintPreallocated(request_json, buf, (int)len, 1);
        cJSON_Delete(request_json);
        esp_mqtt_client_publish(client, TOPIC_MQTT, buf, 0, 0, 0);
        //Free memory to avoid memory leak!
        free(buf);
        jsonPointer = jsonPointer+JSON_SIZE;
    }
}
```

definitions.h

```
/*
* File for constants definition
* Author: Fernando Herranz 699869@unizar.es
*/
#ifndef __definitions_H
#define __definitions_H

/*************** Default Configuration Parameters ***************/
#define SENSOR_MODE_DEFAULT 0
#define SAMPLE_VARIABLES_DEFAULT "ALL"
#define SAMPLE_PERIOD_DEFAULT 100//100 // 100 msec. = 10 Hz
#define TX_BUFFER_SIZE_DEFAULT 7800 // 30 seconds = 7800 / (13*2) / 10
#define NUM_SAMPLE_VARIABLES_DEFAULT 13
/*************** Default Configuration Parameters ***************/

/*************** MQTT AND WIFI ***************/
#define SSL_CONNECTION 1 // if 1 SSL, if 0 not SSL
#define BROKER_URI "mqtt://holysheep:H01y5h3eP!@senializar.unizar.es:4883"
#define BROKER_URI_SSL "mqtts://holysheep:H01y5h3eP!@senializar.unizar.es:6883"
#define TOPIC_MQTT "triki/holysheep/prueba5/qos0"
#define JSON_SIZE 150
#define WIFI_TIMEOUT 10000 //msec
#define NUM_RETRIES 2
/*************** MQTT AND WIFI ***************/

/*************** ULP ***************/
#define ULP_COUNTER (150000 * SAMPLE_PERIOD / 1000) // 150kHz RTC_SLOW_CLK * 50 msec. = 7500
#define ULP_BUFFER_SIZE 100//110 //50
/*************** ULP ***************/

/*************** BUFFER & STORAGE ***************/
#define WIFI_BUFFER_SIZE 31200 // aprox. 2 min //15600
#define RTC_FAST_MEM_SIZE 7800//7800
/*************** BUFFER & STORAGE ***************/

/*************** OTHER ***************/
// GPIO mask for BNO interruption
#define BUTTON_PIN_BITMASK 0x200000000 // 2^33 in hex (because of GPIO33)
// Time in seconds for UART config
#define CONFIG_TIME 15
// Select Debug mode
#define DEBUG_MODE 1
/*************** OTHER ***************/

#endif
```

uart_config.h (I)

```
/*
 * Configuration of LoPy4 parameters before the beginning of the monitoring application
 * Carried out with UART
 * Author: Fernando Herranz 699869@unizar.es
 */

// FreeRTOS includes
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

// UART driver
#include "driver/uart.h"

#include "esp_system.h"
#include "soc/uart_struct.h"
#include "string.h"

// Error library
#include "esp_err.h"

/*************** Configuration Parameters **************/
int SAMPLE_PERIOD;
int TX_BUFFER_SIZE;
char SAMPLE_VARIABLES[3];
int NUM_SAMPLE_VAR;
int SENSOR_MODE; //0:INTENSIVE, 1:IF MOVEMENT, 2:SMART
/*************** Configuration Parameters **************/

/*************** Global ***************/
const int UART_RX_BUF_SIZE = 1024;
nvs_handle my_nvs_handle;
#define TX1_PIN (GPIO_NUM_4)
#define RX1_PIN (GPIO_NUM_15)
/*************** Global ***************/


/*
 * Initialize UART peripheral
*/
void uart_init() {
    const uart_config_t uart_config = {
        .baud_rate = 9600,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
    };
    uart_param_config(UART_NUM_1, &uart_config);
    uart_set_pin(UART_NUM_1, TX1_PIN, RX1_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    // We won't use a buffer for sending data.
    uart_driver_install(UART_NUM_1, UART_RX_BUF_SIZE * 2, 0, 0, NULL, 0);

     uart_write_bytes(UART_NUM_1, "Waiting for config...\r\n", 23);

}


/*
 * Read current configuration stored in NVS memory region
 * Store value in corresponding variable
*/
void read_config() {
     // open the partition in R/W mode
     nvs_open("storage", NVS_READWRITE, &my_nvs_handle);

     // get the configuration values
     size_t string_size;
     nvs_get_str(my_nvs_handle, "SAMPLE_VAR", NULL, &string_size);
     if(nvs_get_str(my_nvs_handle, "SAMPLE_VAR", SAMPLE_VARIABLES, &string_size) != ESP_OK)
strcpy(SAMPLE_VARIABLES, SAMPLE_VARIABLES_DEFAULT);
     if(nvs_get_i32(my_nvs_handle, "NUM_SAMPLE_VAR", &NUM_SAMPLE_VAR) != ESP_OK) NUM_SAMPLE_VAR =
NUM_SAMPLE_VARIABLES_DEFAULT;
     if(nvs_get_i32(my_nvs_handle, "SAMPLE_PER", &SAMPLE_PERIOD) != ESP_OK) SAMPLE_PERIOD =
SAMPLE_PERIOD_DEFAULT; //default 100 msec
     if(nvs_get_i32(my_nvs_handle, "TX_BUF_SIZE", &TX_BUFFER_SIZE) != ESP_OK) TX_BUFFER_SIZE =
TX_BUFFER_SIZE_DEFAULT;
     if(nvs_get_i32(my_nvs_handle, "SENS_MODE", &SENSOR_MODE) != ESP_OK) SENSOR_MODE = SENSOR_MODE_DEFAULT;

}


/*
 * Wait for CONFIG_TIME to receive configuration
 * Upon receive, update the default value of the parameters
*/
void config_with_uart(){
     uint8_t *uart_data = (uint8_t *) malloc(1024);
     char data2parse [1024] = "";
     int temp = 0;

     while((esp_timer_get_time()/1000000) < CONFIG_TIME){
          // Read data from the UART
          int rxBytesUart = uart_read_bytes(UART_NUM_1, uart_data, 1, 1000 / portTICK_RATE_MS); //1 sec.
          // Write data back to the UART
          if(rxBytesUart > 0) {
               uart_data[rxBytesUart] = '\0';
               printf("%s", uart_data);
               fflush(stdout);
```

uart_config.h (II)

```
                strncat(data2parse, (char *)uart_data, 1);
          }
          else{
               cJSON *uart_json = cJSON_Parse(data2parse);
               if(uart_json != NULL && strstr(data2parse, "ConfigFile")){
                    cJSON *uart_file = cJSON_GetObjectItemCaseSensitive(uart_json, "ConfigFile");
                    /* SET CONFIGURATION */
                    if (strstr(uart_file->valuestring, "set")){
                         /* SAMPLE VARIABLES */
                         if( strstr(data2parse, "SAMPLE_VAR") ){
                              cJSON *sample_var = cJSON_GetObjectItemCaseSensitive(uart_json, "SAMPLE_VAR");
                              if (nvs_set_str(my_nvs_handle, "SAMPLE_VAR", sample_var->valuestring) != ESP_OK)
uart_write_bytes(UART_NUM_1, "SampleVariable Error\r\n", 22);
                              if (strcmp(sample_var->valuestring , "ALL") == 0) nvs_set_i32(my_nvs_handle,
"NUM_SAMPLE_VAR", 13), temp = 13;
                              else if (strcmp(sample_var->valuestring , "IMU") == 0) nvs_set_i32(my_nvs_handle,
"NUM_SAMPLE_VAR", 9), temp = 9;
                              else if (strcmp(sample_var->valuestring , "MOV") == 0) nvs_set_i32(my_nvs_handle,
"NUM_SAMPLE_VAR", 6), temp = 6;
                              else if (strcmp(sample_var->valuestring , "ACC") == 0) nvs_set_i32(my_nvs_handle,
"NUM_SAMPLE_VAR", 3), temp = 3;
                              else if (strcmp(sample_var->valuestring , "GYR") == 0) nvs_set_i32(my_nvs_handle,
"NUM_SAMPLE_VAR", 3), temp = 3;
                              else uart_write_bytes(UART_NUM_1, "ERROR SAMPLE_VAR UNKNOWN\r\n", 26);
                         }
                         /* SAMPLE PERIOD */
                         if( strstr(data2parse, "SAMPLE_PER") ){
                              cJSON *sample_per = cJSON_GetObjectItemCaseSensitive(uart_json, "SAMPLE_PER");
                              printf("Valor period: %f \n", sample_per->valuedouble);
                              if (sample_per->valuedouble >= 1 && sample_per->valuedouble <= 1000){
                                   if (nvs_set_i32(my_nvs_handle, "SAMPLE_PER", sample_per->valuedouble) != ESP_OK)
uart_write_bytes(UART_NUM_1, "SamplePeriod Error\r\n", 20);
                              }
                              else uart_write_bytes(UART_NUM_1, "SAMPLE_PER no valido\r\n", 22);
                         }
                         /* TX BUFFER SIZE */
                         if( strstr(data2parse, "TX_BUF_SIZE") ){
                              cJSON *tx_buf_size = cJSON_GetObjectItemCaseSensitive(uart_json, "TX_BUF_SIZE");
                              printf("Valor tx_buffer_size: %f \n", tx_buf_size->valuedouble);
                              if (tx_buf_size->valuedouble < RTC_FAST_MEM_SIZE){
                                   int nearestInteger = temp * 2 * (floor (RTC_FAST_MEM_SIZE / (temp * 2))) ;
                                   if (nvs_set_i32(my_nvs_handle, "TX_BUF_SIZE", nearestInteger) != ESP_OK)
uart_write_bytes(UART_NUM_1, "TxBufferSize Error\r\n", 20);
                              }
                              else{
                                   if ((int)(tx_buf_size->valuedouble) % (temp*2) == 0){
                                        if (nvs_set_i32(my_nvs_handle, "TX_BUF_SIZE", tx_buf_size->valuedouble) !=
ESP_OK) uart_write_bytes(UART_NUM_1, "TxBufferSize Error\r\n", 20);
                                   }
                                   else uart_write_bytes(UART_NUM_1, "TX_BUF_SIZE debe ser multiplo de
NUM_VAR\r\n", 42);
                              }
                         }
                         /* SENSOR MODE */
                         if( strstr(data2parse, "SENS_MODE") ){
                              cJSON *sens_mode = cJSON_GetObjectItemCaseSensitive(uart_json, "SENS_MODE");
                              printf("Valor SENS_MODE: %f \n", sens_mode->valuedouble);
                              if((int)sens_mode->valuedouble == 0) nvs_set_i32(my_nvs_handle, "SENS_MODE", 0);
                              else if((int)sens_mode->valuedouble == 1) nvs_set_i32(my_nvs_handle, "SENS_MODE",
1);
                              else if((int)sens_mode->valuedouble == 2) nvs_set_i32(my_nvs_handle, "SENS_MODE",
2);
                              else uart_write_bytes(UART_NUM_1, "SensorMode Unknown\r\n", 20);
                         }
                         if (nvs_commit(my_nvs_handle) != ESP_OK) uart_write_bytes(UART_NUM_1, "NVSCommit
Error\r\n", 17);
                         else {
                              uart_write_bytes(UART_NUM_1, "OK!\r\n", 5);
                         }
                    }

                    /* GET CONFIGURATION */
                    else if (strstr(uart_file->valuestring, "get")){
                         printf("Sending current config...\n");
                         cJSON *root_config = cJSON_CreateObject();
                         cJSON_AddStringToObject(root_config, "SAMPLE_VAR", SAMPLE_VARIABLES);
                         cJSON_AddNumberToObject(root_config, "SAMPLE_PER", SAMPLE_PERIOD);
                         cJSON_AddNumberToObject(root_config, "TX_BUF_SIZE", TX_BUFFER_SIZE);
                         cJSON_AddNumberToObject(root_config, "SENS_MODE", SENSOR_MODE);
                         char *text_config = cJSON_Print(root_config);
                         uart_write_bytes(UART_NUM_1, text_config, strlen(text_config));
                         free(root_config);
                    }
                    /* UNKNOWN REQUEST */
                    else{
                         uart_write_bytes(UART_NUM_1, "UnknownRequest Error\r\n",22);
                    }
                    printf("data2parse: %s \n", data2parse);
                    free(uart_file);
               }
               memset(data2parse, 0 , sizeof(data2parse));
                         free(uart_json);
          }
          }

     free(uart_data);
     nvs_close(my_nvs_handle);
}
```

wifi_app.h (I)

```
/*
 * Library for WiFi Provisioning with Espressif app (Android & iOS)
 * Author: Fernando Herranz 699869@unizar.es
 */

#include <stdio.h>
#include <esp_log.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "freertos/queue.h"
#include "freertos/event_groups.h"
#include <math.h>
#include "nvs.h"
#include "nvs_flash.h"
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event_loop.h"
#include <wifi_provisioning/manager.h>
#include <wifi_provisioning/scheme_softap.h>

/*************** Global variables ***************/
extern EventGroupHandle_t wifi_event_group;
const static int CONNECTED_BIT = BIT0;
TaskHandle_t taskHandle2;
void sleep_no_config(void *pvParameter);
/*************** Global variables ***************/

/*************** Functions ***************/
static esp_err_t wifi_app_event_handler(void *ctx, system_event_t *event);
void get_credentials(void);
static void get_device_service_name(char *service_name, size_t max);
static void prov_event_handler(void *user_data, wifi_prov_cb_event_t event, void *event_data);
/*************** Functions ***************/

/*
 * Wait a max of 2min for the wifi credentials
 * If there are no credentials after this time, go to sleep
*/
void sleep_no_config(void *pvParameter){

     while((esp_timer_get_time()/1000000) < 120){
          vTaskDelay(1000 / portTICK_PERIOD_MS);
     }
     wifi_prov_mgr_deinit();
     esp_wifi_stop();
     esp_deep_sleep_start();
     vTaskDelete( taskHandle2 );
}

/*
 * Start as AP, wait for the app to connect and provision.
 * Then check if connection can be established
*/
void get_credentials(void){
     xTaskCreatePinnedToCore(&sleep_no_config, "sleep_no_config", 4096, NULL, 10, &taskHandle2, 1);
    // Initialize NVS partition
    esp_err_t ret = nvs_flash_init();
        if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
            ESP_ERROR_CHECK(nvs_flash_erase());
            // Retry nvs_flash_init
            ESP_ERROR_CHECK(nvs_flash_init());
        }
    // Initialize TCP/IP and the event loop
    tcpip_adapter_init();
    ESP_ERROR_CHECK(esp_event_loop_init(wifi_app_event_handler, NULL));
    wifi_event_group = xEventGroupCreate();

    // Initialize Wi-Fi
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    // Initialize provisioning manager
    wifi_prov_mgr_config_t config = {
        .scheme = wifi_prov_scheme_softap,
        .scheme_event_handler = WIFI_PROV_EVENT_HANDLER_NONE,
        .app_event_handler = {
            .event_cb = prov_event_handler,
            .user_data = NULL
        }
    };
    ESP_ERROR_CHECK(wifi_prov_mgr_init(config));
    bool provisioned = false;
    ESP_ERROR_CHECK(wifi_prov_mgr_is_provisioned(&provisioned));

    // If device is not provisioned start provisioning service
    if (!provisioned) {
        //printf("Starting provisioning\n");
        char service_name[12];
        get_device_service_name(service_name, sizeof(service_name));
        wifi_prov_security_t security = WIFI_PROV_SECURITY_1;
        const char *pop = "abcd1234";
        const char *service_key = NULL;
        ESP_ERROR_CHECK(wifi_prov_mgr_start_provisioning(security, pop, service_name, service_key));

        wifi_prov_mgr_wait();
        wifi_prov_mgr_deinit();
```

wifi_app.h (II)

```
        EventBits_t uxBits = xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT, false, true,
60000/portTICK_PERIOD_MS);
        if ((uxBits & CONNECTED_BIT) != CONNECTED_BIT){
          esp_restart();
        }
    }
    //printf("Finish");
    vTaskDelete( taskHandle2 );
    // Stop WiFi when provisioned
    esp_wifi_stop();
}


/*
 * Get name of the device with which we will see it in the app
*/
static void get_device_service_name(char *service_name, size_t max)
{
    uint8_t eth_mac[6];
    const char *ssid_prefix = "PROV_";
    esp_wifi_get_mac(WIFI_IF_STA, eth_mac);
    snprintf(service_name, max, "%s%02X%02X%02X",
             ssid_prefix, eth_mac[3], eth_mac[4], eth_mac[5]);
}

/*
* Event handler for catching provisioning manager events
*/
static void prov_event_handler(void *user_data, wifi_prov_cb_event_t event, void *event_data)
{
    switch (event) {
        case WIFI_PROV_START:
            //printf("Provisioning started\n");
            break;
        case WIFI_PROV_CRED_RECV: {
            wifi_sta_config_t *wifi_sta_cfg = (wifi_sta_config_t *)event_data;
            // If SSID length is exactly 32 bytes, null termination will not be present, so explicitly obtain the
length
            size_t ssid_len = strnlen((const char *)wifi_sta_cfg->ssid, sizeof(wifi_sta_cfg->ssid));
            break;
        }
        case WIFI_PROV_CRED_FAIL: {
            wifi_prov_sta_fail_reason_t *reason = (wifi_prov_sta_fail_reason_t *)event_data;
            nvs_flash_erase();
            esp_restart();
            break;
        }
        case WIFI_PROV_CRED_SUCCESS:
            break;
        case WIFI_PROV_END:
            // De-initialize manager once provisioning is finished
            wifi_prov_mgr_deinit();
            break;
        default:
            break;
    }
}


/*
* Event handler for WiFi connection
*/
static esp_err_t wifi_app_event_handler(void *ctx, system_event_t *event)
{
     wifi_prov_mgr_event_handler(ctx, event);

    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect();
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            xEventGroupSetBits(wifi_event_group, CONNECTED_BIT);
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect();
            xEventGroupClearBits(wifi_event_group, CONNECTED_BIT);
            break;
        default:
            break;
    }
    return ESP_OK;
}
```

bno_i2c_config.h (I)

```
/*
 * Library for BNO055 configuration
 * Author: Fernando Herranz 699869@unizar.es
 */
#include <stdio.h>
#include <esp_log.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <math.h>
#include "driver/i2c.h"
#include "bno055/bno055.h"
#include "bno055/bno055.c"
#include "definitions.h"
#define SDA_PIN GPIO_NUM_12
#define SCL_PIN GPIO_NUM_13
#define I2C_MASTER_ACK 0
#define I2C_MASTER_NACK 1

/*
 * Configure I2C
*/
void i2c_master_init() {
     i2c_config_t conf;
     conf.mode = I2C_MODE_MASTER;
     conf.sda_io_num = SDA_PIN;
     conf.scl_io_num = SCL_PIN;
     conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
     conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
     // 10kHz. More than 20kHz could lead to problems in communication
     conf.master.clk_speed = 10000;
     i2c_param_config(I2C_NUM_0, &conf);
     i2c_driver_install(I2C_NUM_0, I2C_MODE_MASTER, 0, 0, 0);
}

/*
 * I2C write function.
 * Not used in this version, as ULP performs the writings in assembly code (.S)
*/
s8 BNO055_I2C_bus_write(u8 dev_addr, u8 reg_addr, u8 *reg_data, u8 cnt) {
     u8 iError = BNO055_INIT_VALUE;
     esp_err_t espRc;
     i2c_cmd_handle_t cmd = i2c_cmd_link_create();
     i2c_master_start(cmd);
     i2c_master_write_byte(cmd, (dev_addr << 1) | I2C_MASTER_WRITE, 1);
     i2c_master_write_byte(cmd, reg_addr, 1);
     i2c_master_write(cmd, reg_data, cnt, 1);
     i2c_master_stop(cmd);

     espRc = i2c_master_cmd_begin(I2C_NUM_0, cmd, 10 / portTICK_PERIOD_MS);
     if (espRc == ESP_OK) {
          iError = BNO055_SUCCESS;
     } else {
          iError = BNO055_ERROR;
     }
     i2c_cmd_link_delete(cmd);
     return (s8) iError;
}

/*
 * I2C read function.
 * Not used in this version, as ULP performs the readings in assembly code (.S)
*/
s8 BNO055_I2C_bus_read(u8 dev_addr, u8 reg_addr, u8 *reg_data, u8 cnt) {
     u8 iError = BNO055_INIT_VALUE;
     esp_err_t espRc;

     i2c_cmd_handle_t cmd = i2c_cmd_link_create();

     i2c_master_start(cmd);
     i2c_master_write_byte(cmd, (dev_addr << 1) | I2C_MASTER_WRITE, 1);
     i2c_master_write_byte(cmd, reg_addr, 1);

     i2c_master_start(cmd);
     i2c_master_write_byte(cmd, (dev_addr << 1) | I2C_MASTER_READ, 1);

     if (cnt > 1) {
          i2c_master_read(cmd, reg_data, cnt - 1, I2C_MASTER_ACK);
     }
     i2c_master_read_byte(cmd, reg_data + cnt - 1, I2C_MASTER_NACK);
     i2c_master_stop(cmd);

     espRc = i2c_master_cmd_begin(I2C_NUM_0, cmd, 10 / portTICK_PERIOD_MS);
     if (espRc == ESP_OK) {
          iError = BNO055_SUCCESS;
     } else {
          iError = BNO055_ERROR;
     }

     i2c_cmd_link_delete(cmd);

     return (s8) iError;
}

/*
 * I2C delay function.
 * Not used in this version, as ULP manages to meet timing constraints in assembly code (.S)
*/
void BNO055_delay_msek(u32 msek) {
     vTaskDelay(msek / portTICK_PERIOD_MS);
}
```

bno_i2c_config.h (II)

```
/*
 * Config BNO before going to sleep.
*/
void miBNO_config(int SENSOR_MODE, char *SAMPLE_VARIABLES) {
     i2c_master_init();

     struct bno055_t myBNO = { .bus_write = BNO055_I2C_bus_write, .bus_read =
               BNO055_I2C_bus_read, .dev_addr = BNO055_I2C_ADDR2, .delay_msec =
               BNO055_delay_msek };

     bno055_init(&myBNO);

     // if SENSOR_MODE = IF_MOVEMENT
     if(SENSOR_MODE == 1){
          bno055_set_power_mode(BNO055_POWER_MODE_LOWPOWER);
          bno055_set_intr_rst(1);

          /* Configure sensor interrupts */
          bno055_set_accel_any_motion_no_motion_axis_enable(0,1);
          bno055_set_accel_any_motion_no_motion_axis_enable(1,1);
          bno055_set_accel_any_motion_no_motion_axis_enable(2,1);

          bno055_set_accel_any_motion_thres(254);
          bno055_set_accel_any_motion_durn(1);
          bno055_set_intr_accel_any_motion(1);
          bno055_set_intr_mask_accel_any_motion(1);
     }

     /* Mode of operation according to SAMPLE_VARIABLES */
     if (!strcmp (SAMPLE_VARIABLES , "ALL")) bno055_set_operation_mode(BNO055_OPERATION_MODE_NDOF);
     else if (!strcmp (SAMPLE_VARIABLES , "IMU")) bno055_set_operation_mode(BNO055_OPERATION_MODE_AMG);
     else if (!strcmp (SAMPLE_VARIABLES , "MOV")) bno055_set_operation_mode(BNO055_OPERATION_MODE_ACCGYRO);
     else if (!strcmp (SAMPLE_VARIABLES , "ACC")) bno055_set_operation_mode(BNO055_OPERATION_MODE_ACCONLY);
     else if (!strcmp (SAMPLE_VARIABLES , "GYR")) bno055_set_operation_mode(BNO055_OPERATION_MODE_GYRONLY);
     else bno055_set_operation_mode(BNO055_OPERATION_MODE_NDOF);

     // BNO Calibration
     unsigned char accel_calib_status = 0;
     unsigned char gyro_calib_status = 0;
     unsigned char mag_calib_status = 0;
     unsigned char sys_calib_status = 0;
     bno055_get_accel_calib_stat(&accel_calib_status);
     bno055_get_mag_calib_stat(&mag_calib_status);
     bno055_get_gyro_calib_stat(&gyro_calib_status);
     bno055_get_sys_calib_stat(&sys_calib_status);
}
```

Anexo VIII. Código para análisis de los datos (Python)

```
import csv
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.cluster import MiniBatchKMeans
from sklearn.mixture import GaussianMixture
from scipy import signal
from scipy.signal import savgol_filter
from sklearn.manifold import TSNE
from sklearn.metrics import confusion_matrix
from scipy.fft import fft, ifft
from sklearn.cluster import AgglomerativeClustering

# Median filtering of window_size samples of input data
def median_filter(data, window_size):
    samples, features=data.shape
    filtered_data=np.zeros([samples, features])
    for i in range(features):
        filtered_data[:,i]=signal.medfilt(data[:,i], window_size)
    return filtered_data

# Def for confusion matrix
def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues, labels=None):
    fig = plt.figure(figsize=(12,12))
    ax = fig.add_subplot(111)
    cax = ax.matshow(cm)
    plt.title(title,fontsize=15)
    fig.colorbar(cax)
    if labels.all():
        ax.set_xticklabels(['']  labels)
        ax.set_yticklabels(['']  labels)
    plt.xlabel('Predicted',fontsize=15)
    plt.ylabel('True',fontsize=15)
    plt.show()

# Split data into segments. Not used

segment_length = 10
sample_rate = 10

def split_data(data, segment_length, sample_rate):
    segments = []
    index = 0
    while index < len(data):
        segment = data[index:index + segment_length*sample_rate]
        segments.append(segment)
        index += segment_length*sample_rate
    return segments

segments = split_data(IMU_data, segment_length, sample_rate)
segments = np.asarray(segments)
print(IMU_data.shape)
print(segments.shape)
print(segments[1])
plt.plot(segments[1])

#data = pd.read_csv('DatosCasaConEtiquetas.csv', ';')
data = pd.read_csv('DatosOvejasConEtiquetas.csv', ';')
data=np.asarray(data)

Train

#train_data = np.concatenate((data[:,1:4] , data[:,5:-1]), axis = 1) # exclude gyro X
#train_data = np.concatenate((data[:,4:5] , data[:,6:7]), axis = 1)
train_data =data[:,4:7] # just gyro data
train_true_labels=data[:,-1]
train_time = data[:,1]
print(train_data.shape)
```

```
# Pre-processing of data. Change number_of_data according to the number of registers to be consi
dered
from scipy.ndimage.filters import gaussian_filter1d
filtered_train_data = np.zeros(train_data.shape)
number_of_data = 3

fig, axs = plt.subplots(number_of_data, figsize=(12,20))
fig.tight_layout(pad=3.0)

for i in range (number_of_data):
        filtered_segment = np.gradient(train_data[:,i])
        #filtered_segment = np.gradient(filtered_segment)
        filtered_segment = np.abs(filtered_segment)
        filtered_segment = gaussian_filter1d(filtered_segment, 40)
        filtered_train_data[:,i] = filtered_segment
        axs[i].plot(filtered_train_data)

print(filtered_segment.shape)
print(filtered_train_data.shape)


# Apply clustering algorithms
kmeans = KMeans(n_clusters=3, random_state=0).fit(filtered_train_data[:,:])
# mb_kmeans = MiniBatchKMeans(n_clusters=3, random_state=0).fit(filtered_IMU_data[:,:])
# gmm_labels = GaussianMixture(n_components=2, covariance_type='diag', max_iter=1000).fit_predic
t(filtered_IMU_data)

# In this case we keep k-means result
train_labels = kmeans.labels_
#labels = gmm_labels

# Create integer labels according to labels set manually
tr_integer_labels = []
for label in train_true_labels:
    if(label == 'Seated'):
        tr_integer_labels.append(0)
    elif(label == 'Standing'):
        tr_integer_labels.append(0)
    elif(label == 'Walking'):
        tr_integer_labels.append(1)
    elif(label == 'Running'):
        tr_integer_labels.append(2)

tr_integer_labels = np.asarray(tr_integer_labels)

# Compare true vs predicted labels
fig,ax = plt.subplots(figsize=(12,11))
fig.canvas.draw()
ax.plot(train_labels, 'ro', markersize=2)
ax.plot(tr_integer_labels)
ax.set_xlabel('Samples',fontsize=15)
ax.set_ylabel('Activities',fontsize=15)
ax.set_title('True vs Predicted Labels',fontsize=15)
ax.set_yticks([0,1,2])
ax.set_yticklabels(['Standing','Walking','Running'],fontsize=15)
ax.legend(('Predicted Label', 'True Label'),loc='upper left',ncol=1,fontsize=18)

plt.show()

# Plot confusion matrix
cm = confusion_matrix(tr_integer_labels, train_labels)
plot_confusion_matrix(cm, labels=train_labels)

# Compute error with K-means algorithm
tr_error_kmeans = (np.sum(kmeans.labels_ != tr_integer_labels)/len(tr_integer_labels))*100
print('Error: '+ str(tr_error_kmeans) + '%')
```

Test

```
#test_data = np.concatenate((data[7942:,1:4] , data[7942:,5:-1]), axis = 1) # exclude gyro X
#test_true_labels=data[7942:,-1]
#test_time = data[7942:,1]
extern_data = pd.read_csv('S1.csv', ',')
extern_data=np.asarray(extern_data)
test_data =extern_data[:,13:16] # just gyro data
test_true_labels=extern_data[:,0]
print(test_data.shape)
```

```
# Pre-processing of data. Change number_of_data according to the number of registers to be consi
dered
from scipy.ndimage.filters import gaussian_filter1d
filtered_test_data = np.zeros(test_data.shape)
number_of_data = 3

fig, axs = plt.subplots(number_of_data, figsize=(12,20))
fig.tight_layout(pad=3.0)

for i in range (number_of_data):
        ts_filtered_segment = np.gradient(test_data[:,i])
        #filtered_segment = np.gradient(filtered_segment)
        ts_filtered_segment = np.abs(ts_filtered_segment)
        ts_filtered_segment = gaussian_filter1d(ts_filtered_segment, 20)
        filtered_test_data[:,i] = ts_filtered_segment
        axs[i].plot(filtered_test_data)

print(ts_filtered_segment.shape)
print(filtered_test_data.shape)

test_labels = kmeans.predict(filtered_test_data[:,:])

# Create integer labels according to labels set manually
ts_integer_labels = []
for label in test_true_labels:
    if(label == 'standing' or label == 'lying' or label == 'grazing'):
        ts_integer_labels.append(0)
    elif(label == 'walking'):
        ts_integer_labels.append(2)
    elif(label == 'running' or label == 'trotting'):
        ts_integer_labels.append(1)
    else: ts_integer_labels.append(0)

ts_integer_labels = np.asarray(ts_integer_labels)
print(ts_integer_labels.shape)

# Compare true vs predicted labels
fig,ax = plt.subplots(figsize=(12,11))
fig.canvas.draw()
ax.plot(test_labels, 'ro', markersize=2)
ax.plot(ts_integer_labels)
ax.set_xlabel('Samples',fontsize=15)
ax.set_ylabel('Activities',fontsize=15)
ax.set_title('True vs Predicted Labels',fontsize=15)
ax.set_yticks([0,2,1])
ax.set_yticklabels(['Standing','Running','Walking'],fontsize=15)
ax.legend(('Predicted Label', 'True Label'),loc='best',ncol=1,fontsize=18)

plt.show()

# Plot confusion matrix
cm = confusion_matrix(ts_integer_labels, test_labels)
plot_confusion_matrix(cm, labels=test_labels)

# Compute error with K-means algorithm
ts_error_kmeans = (np.sum(test_labels != ts_integer_labels)/len(ts_integer_labels))*100
print('Error: '+ str(ts_error_kmeans) + '%')
```

Plotting

```
extern_data = pd.read_csv('S1.csv', ',')
extern_data=np.asarray(extern_data)
ext_true_labels=extern_data[:,0]
print(ext_true_labels)

ext_integer_labels = []
for label in ext_true_labels:
    if(label == 'standing'):
        ext_integer_labels.append(0)
    elif(label == 'walking'):
        ext_integer_labels.append(1)
    elif(label == 'running'):
        ext_integer_labels.append(2)

ext_integer_labels = np.asarray(ext_integer_labels)

# Count occurrences of each label for true and predicted

pred_standing = np.sum(train_labels == 0)
print(pred_standing)
real_standing = np.sum(tr_integer_labels == 0)
```

```
print(real_standing)
pred_walking = np.sum(train_labels == 1)
print(pred_walking)
real_walking = np.sum(tr_integer_labels == 1)
print(real_walking)
pred_running = np.sum(train_labels == 2)
print(pred_running)
real_running = np.sum(tr_integer_labels == 2)
print(real_running)

vector_pred = [pred_standing, pred_walking, pred_running]
print(vector_pred)
vector_true = [real_standing, real_walking, real_running]

summ = pred_standing+pred_walking+pred_running
vector_pred_percent = [pred_standing*100/summ ,pred_walking*100/summ ,pred_running*100/summ ]
vector_pred_percent = np.round(vector_pred_percent, decimals=2)
print(vector_pred_percent)

# Plot histogram with occurrences of each label for true and predicted (computed in previous cel
l)
labels = ['Standing', 'Walking', 'Running']
x = np.arange(len(labels))  # the label locations
width = 0.35  # the width of the bars
fig, ax = plt.subplots(figsize=(15,11))
rects1 = ax.bar(x - width/2, vector_pred, width, label='Predicted')
rects2 = ax.bar(x + width/2, vector_true, width, label='True')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Count',fontsize=15)
ax.set_title('Activities count: true vs predicted ',fontsize=15)
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend(('Predicted', 'True'),loc='upper right',ncol=1,fontsize=18)


def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)
fig.tight_layout()
plt.show()

# Count occurrences of each label for real sheep (external dataset Twente University)
ext_integer_labels = np.asarray(ext_integer_labels)
ext_standing = np.sum(ext_integer_labels == 0)
print(ext_standing)
ext_walking = np.sum(ext_integer_labels == 1)
print(ext_walking)
ext_running = np.sum(ext_integer_labels == 2)
print(ext_running)
vector_extern = [ext_standing, ext_walking, ext_running]
summ = ext_standing + ext_walking + ext_running
vector_extern_percent = [ext_standing*100/summ ,ext_walking*100/summ ,ext_running*100/summ ]
vector_extern_percent = np.around(vector_extern_percent, decimals=2)
print(vector_extern_percent)

# Plot histogram with percentage of occurrences of each label for our predictions (our sheep) an
d external dataset

labels = ['Standing', 'Walking', 'Running']

x = np.arange(len(labels))  # the label locations
width = 0.35  # the width of the bars

fig, ax = plt.subplots(figsize=(15,11))
rects1 = ax.bar(x - width/2, vector_pred_percent, width, label='Our sheep (Predicted by LoPy4)')
rects2 = ax.bar(x + width/2, vector_extern_percent, width, label='External dataset')
```

```
# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Percentage (%)',fontsize=15)
ax.set_title('Distribution of activities: Our sheep vs External sheep ',fontsize=15)
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend(('Our sheep (Predicted by LoPy4)', 'External dataset'),loc='upper right',ncol=1,fontsi
ze=18)


def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')


autolabel(rects1)
autolabel(rects2)

fig.tight_layout()

plt.show()
```

Referencias

[1] J. Lee, H.-A. Kao, and S. Yang, "Service Innovation and Smart Analytics for Industry 4.0 and Big Data Environment," Procedia CIRP, vol. 16, pp. 3–8, 2014.

[2] Guo, Leifeng & Welch, Mitchell & Dobos, Robin & Kwan, Paul & Wang, Wensheng. (2018). Comparison of grazing behaviour of sheep on pasture with different sward surface heights using an inertial measurement unit sensor. Computers and Electronics in Agriculture. 150. 394-401. 10.1016/j.compag.2018.05.004.

[3] Fountain S, Windolf M, Henkel J, et al. Monitoring Healing Progression and Characterizing the Mechanical Environment in Preclinical Models for Bone Tissue Engineering. Tissue Eng Part B Rev. 2016;22(1):47-57.

[4] Painter, M.S., Blanco, J.A., Malkemper, E.P. et al. Use of bio-loggers to characterize red fox behavior with implications for studies of magnetic alignment responses in free-roaming animals. Anim Biotelemetry **4,** 20 (2016).

[5] Wall, Jake & Wittemyer, George & Klinkenberg, Brian & Douglas-Hamilton, Iain. (2014). Novel opportunities for wildlife conservation and research with real-time monitoring. Ecological Applications. 24. 593-601. 10.1890/13-1971.1.

[6] Le Roux, Solomon & Wolhuter, Riaan & Niesler, Thomas & Marias, Jacques. (2017). Animal-borne behaviour classification for sheep (Dohne Merino) and Rhinoceros (Ceratotherium simum and Diceros bicornis). Animal Biotelemetry. 5. 10.1186/s40317-017-0140-0.

[7] Kaler J, Mitsch J, Vázquez-Diosdado JA, Bollard N, Dottorini T, Ellis KA. Automated detection of lameness in sheep using machine learning approaches: novel insights into behavioural differences among lame and non-lame sheep. R Soc Open Sci. 2020;7(1):190824. Published 2020 Jan 15.

[8] Ahmad, Norhafizan & Raja Ghazilla, Raja Ariffin & Khairi, Nazirah & Kasi, Vijayabaskar. (2013). Reviews on Various Inertial Measurement Unit (IMU) Sensor Applications. International Journal of Signal Processing Systems. 1. 256-262. 10.12720/ijsps.1.2.256-262.

[9] Galar, Diego & Kumar, Uday. (2017). eMaintenance: Essential Electronic Tools for Efficiency.

[10] B. Sensortec, "BNO055", Bosch Sensortec, 2020. [Online]. Available: https://www.bosch-sensortec.com/products/smart-sensors/bno055.html.

[11] "BNO080/BNO085 Datasheet - CEVA", CEVA, 2020. [Online]. Available: https://www.ceva-dsp.com/resource/bno080_085-datasheet/.

[12] "WiPy 3.0 - Pycom", Pycom, 2020. [Online]. Available: https://pycom.io/product/wipy-3-0/.

[13] B. Sensortec, "BME680", Bosch Sensortec, 2020. [Online]. Available: https://www.bosch-sensortec.com/products/environmental-sensors/gas-sensors-bme680/.

[14] "MicroPython - Python for microcontrollers", Micropython.org, 2020. [Online]. Available: http://micropython.org/.

[15] Docs.pycom.io, 2020. [Online]. Available: https://docs.pycom.io/index.html

[16] Espressif.com, 2020. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[17] "ESP32 ULP coprocessor instruction set - ESP32 - — ESP-IDF Programming Guide latest documentation", Docs.espressif.com, 2020. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/ulp_instruction_set.html.

[18] S. Henrik Olav, "Instruction Set for Bit-Banging Operations - Increasing flexibility for low power communication", Master's Thesis, NTNU: Norwegian University of Science and Technology, 2016.

[19] Kappaganthu, Lakshmi & Prakash M, Durga & Yadlapati, Avinash. (2017). I 2 C protocol and its clock stretching verification using system verilog and UVM. 478-480. 10.1109/ICICCT.2017.7975245.

[20] Cdn-shop.adafruit.com, 2020. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf.

[21] "I²C", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/I%C2%B2C#Example_of_bit-banging_the_I.C2.B2C_master_protocol.

[22] Play.google.com, 2020. [Online]. Available: https://play.google.com/store/apps/details?id=com.espressif.provsoftap.

[23] "ESP SoftAP Provisioning", App Store, 2020. [Online]. Available: https://apps.apple.com/in/app/esp-softap-provisioning/id1474040630.

[24] J. Ruiz Más and I. Martínez Ruiz, "Fundamentos de Redes Unidad I. Introducción a las redes de comunicaciones", Universidad de Zaragoza, 2015.

[25] T. Yokotani and Y. Sasaki, "Comparison with HTTP and MQTT on required network resources for IoT," 2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC), Bandung, 2016, pp. 1-6.

[26] MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[27] Lee, Shinho & Kim, Hyeonwoo & Hong, Dong-Kweon & Ju, Hongtaek. (2013). Correlation analysis of MQTT loss and delay according to QoS level. 714-717. 10.1109/ICOIN.2013.6496715.

[28] S. Y. Syeda Noor Zehra Naqvi, “Time series databases and influxdb,” Studienarbeit, Universite Libre de Bruxelles, 2017, URL: ´ http://cs.ulb.ac.be/public/ media/teaching/influxdb 2017.pdf

[29] "Time Series Database (TSDB) Explained | InfluxDB | InfluxData", InfluxData, 2020. [Online]. Available: https://www.influxdata.com/time-series-database/.

[30] "DB-Engines Ranking", DB-Engines, 2020. [Online]. Available: https://db-engines.com/en/ranking/time+series+dbms.

[31] T. Nordquist, "MQTT Explorer", MQTT Explorer, 2020. [Online]. Available: http://mqtt-explorer.com/.

[32] Zucker, Gerhard & Frank, Heinz. (2011). The Industrial Electronics Handbook.

[33] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[34] Nurseitov, N., Paulson, M., Reynolds, R., & Izurieta, C. (2009). Comparison of JSON and XML Data Interchange Formats: A Case Study. CAINE.

[35] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Sensor Measurement Lists (SenML)", RFC 8428, DOI 10.17487/RFC8428, August 2018, <https://www.rfc-editor.org/info/rfc8428>.

[36] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <https://www.rfc-editor.org/info/rfc8141>.

[37] "Unix Time Stamp - Epoch Converter", Unixtimestamp.com, 2020. [Online]. Available: https://www.unixtimestamp.com/.

[38] "sklearn.cluster.KMeans — scikit-learn 0.23.1 documentation", *Scikit-learn.org*, 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[39] Jacob W. Kamminga, Helena C. Bisby, Duv V. Le, Nirvana Meratnia, and Paul J.M. Havinga. Generic online animal activity recognition on collar tags. In Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp/ISWC'17). ACM, September 2017

[40] G. Wetzstein, "3-DOF Orientation Tracking with IMUs", Stanford University, 2020.