UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Implementación de una pasarela para la interconexión de dispositivos BLE mediante tecnología LoRa

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: D. Carlos Santiago Viera Betancor

Tutores: D. Valentín De Armas Sosa

D. Félix B. Tobajas Guerrero

Fecha: Enero de 2019

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Implementación de una pasarela para la interconexión de dispositivos BLE mediante tecnología LoRa

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: _____

Vocal **Secretario/a**

Fdo.: _____ Fdo.: _____

Fecha: Enero de 2019

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Acrónimos

Acrónimo	Significado
ADC	*Analog to Digital Converter*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
ATT	*Attribute Protocol*
BLE	*Bluetooth Low Energy*
BOE	*Boletín Oficial del Estado*
CPU	*Central Processing Unit*
CRC	Verificación de redundancia cíclica
CSS	*Chirp Spread Spectrum*
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
ETSI	*European Telecommunications Standards Institute*
FEC	*Forward Error-correction Code*
FSK	*Frequency Shifting Keying*
GAP	*Generic Access Profile*
GATT	*Generic Attribute Profile*
GFSK	Modulación por Desplazamiento de Frecuencia Gausiana
GPIO	*General Purpose Input/Output*
HCI	*Host Controller Interface*
HW/SW	*Hardware/Software*
I2C	*Inter-Integrated Circuit*
IEC	*International Electrotechnical Commission*
IoT	*Internet of Things*
ISM	*Industrial Scientific and Medical*
ISO	*International Organization for Standardization*
ITU	*International Telecommunication Union*
L2CAP	*Logical Link Control and Adaptation Protocol*
LED	*Light Emitting Diode*
LL	Capa de Enlace
LoRa	*Long Range*
LPWAN	*Low Power Wide Area Network*
MAC	*Medium Access Control*
NVM	*Non-Volatile Memory*
P2P	*Peer-To-Peer*
PC	*Personal Computer*
PWM	*Pulse-Width Modulation*
RAM	*Random Access Memory*
REPL	*Read-Eval-Print-Loop*
RGB	*Red Green Blue*
RSSI	*Indicador de Fuerza de la Señal Recibida*
SDIO	*Secure Digital Input Output*
SIG	*Special Interest Group*
SMP	*Security Manager Protocol*

SoC	*System on Chip*
SPI	*Serial Peripheral Interface*
SRAM	*Static Random Access Memory*
SSL	*Secure Sockets Layer*
TFG	Trabajo Fin de Grado
TLS	*Transport Layer Security*
UART	*Universal Asynchronous Receiver-Transmitter*
ULPGC	Universidad de Las Palmas de Gran Canaria
USB	*Universal Serial Bus*
UUID	*Identificador Único Universal*

MEMORIA

CAPÍTULO 1: INTRODUCCIÓN

En este capítulo se presentan los motivos que han dado lugar al planteamiento de este Trabajo de Fin de Grado (TFG), así como los objetivos que se pretenden satisfacer durante su desarrollo. Asimismo, se incluye el peticionario y la estructura del presente documento con el fin de proporcionar una idea global del trabajo realizado

1.1. Antecedentes

El concepto de *Internet of Things* (IoT) surge como una solución para la interconexión entre dispositivos, los cuales integran componentes electrónicos, software, sensores y protocolos de conectividad inalámbrica, que recopilan e intercambian información a través de redes no cableadas conectadas a Internet [1]. IoT permite que dispositivos conectados puedan comunicarse y controlarse de forma remota a través de aplicaciones, combinando las infraestructuras de Internet ya existentes, con diferentes sistemas de comunicación inalámbrica [2].

En la actualidad, IoT tiene una presencia notable y exponencialmente creciente en un gran número de ámbitos de aplicación, principalmente en el sector industrial y en el sector servicios, destacando su impacto en la electrónica de consumo, el sector de la automoción, los servicios públicos, las ciudades y edificios inteligentes, o las cadenas de suministros [3]. Según *Machina Research* [4], el principal proveedor de análisis de mercado y visión estratégica sobre IoT, el número de dispositivos conectados superará los 25 billones para el año 2025 [5].

En base a la evolución actual de IoT, se prevé que la mayoría de las conexiones provendrá de tecnologías fijas y de corto alcance, como *WiFi* o *Bluetooth*, adecuadas por lo general para aplicaciones en las que el consumo de energía y la duración de la batería no representan un problema. Por otra parte, se encuentran las conexiones relacionadas con tecnologías LPWA (*Low Power Wide Area*) [6], las cuales ofrecen soluciones viables para aquellas aplicaciones IoT que tengan que cumplir con requisitos estrictos de consumo de potencia, orientadas para la transmisión y recepción de volúmenes de datos relativamente bajos.

Así, las redes LPWA están orientadas a servicios y aplicaciones que pueden requerir de una comunicación de largo alcance (hasta decenas de kilómetros) para intercambiar datos entre dispositivos, con un bajo consumo de potencia, de forma que puedan operar durante periodos prolongados de tiempo de manera remota sin tener que reemplazar las baterías con frecuencia. Así, surge el estándar *LoRa* (*Long Range*) [7], especificación que ofrece una combinación de largo alcance, bajo consumo de potencia y transmisión segura de datos. Las redes, tanto públicas como privadas, que incluyan esta tecnología, podrán proporcionar cobertura de mayor alcance en comparación con las redes celulares existentes. Entre las aplicaciones que se están desarrollando en la actualidad relacionadas con esta tecnología se encuentra la instalación realizada por parte de la empresa *Samsung* en Corea del Sur, de una red IoT *LoRa* para la gestión de tareas como la

autorregulación del alumbrado público en función de las condiciones ambientales y de tráfico, además de enviar información sobre la contaminación del aire, entre otras [8].

Lo interesante de esta tecnología, es que permite establecer redes en ubicaciones geográficas de difícil acceso, ya sea técnica o económicamente. Dicha dificultad se debe a que un gran número de las tecnologías empleadas en IoT dependen de las infraestructuras de Telecomunicación, tanto cableadas (Ethernet o Fibra) como torres de retransmisión inalámbrica (redes celulares), para su correcto funcionamiento. Estas limitaciones desaparecen cuando se combina una nueva tecnología como *LoRa*, con una que ya está claramente establecida y que representa un elemento fundamental en una gran cantidad de dispositivos y de implementaciones IoT, como es BLE (*Bluetooth Low Energy*) [9].

Actualmente, el protocolo BLE puede implementarse prácticamente en cualquier plataforma empotrada, dado su pequeño tamaño y arquitectura de bajo consumo, lo que permite implementar sensores que puedan trabajar durante años de manera autónoma, aunque con un alcance significativamente reducido [10]. En este tipo de soluciones, la integración de la tecnología *LoRa* con BLE proporcionaría la posibilidad de realizar transferencias seguras de datos, bidireccionales, entre dispositivos BLE a largas distancias.

1.2. Peticionario

Actúa como peticionario del presente Trabajo Fin de Grado (TFG) la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC) como requisito indispensable para la obtención del título de Graduado en Ingeniería en Tecnologías de la Telecomunicación, tras haber superado con éxito las asignaturas especificadas en el Plan de Estudios.

1.3. Objetivos del trabajo

Atendiendo a lo expuesto anteriormente, el objetivo principal del presente TFG consiste en el diseño e implementación de una pasarela para la conexión de dispositivos BLE mediante una comunicación bidireccional basada en la tecnología *LoRa*, orientada a aplicaciones de IoT. Para ello, inicialmente se hará uso de la placa de desarrollo *LoPy*, de la empresa *Pycom*, diseñada específicamente para aplicaciones de IoT y compatible con las tecnologías *WiFi*, BLE y *LoRa* para

implementar la funcionalidad de la pasarela. La programación del dispositivo *LoPy* se realiza en lenguaje *MicroPython*.

Posteriormente, y con el fin de proporcionar una solución alternativa más genérica, que permita su integración con otros dispositivos IoT que no soporten ambas tecnologías de comunicación, la pasarela bidireccional BLE-*LoRa*-BLE se implementará a partir de un dispositivo *Central/Peripheral* BLE implementado en la plataforma hardware *RedBear Duo*, de la empresa *RedBear*, y de varios dispositivos *LoPy*, que en este caso proporcionarán únicamente conectividad *LoRa* en la plataforma final. La programación de los dispositivos *RedBear Duo* se realizará en *Wiring*.

1.4. Estructura de la memoria

El presente documento se ha dividido en cuatro partes diferenciadas: *Memoria*, *Pliego de condiciones*, *Presupuesto* y *Anexo*. Así, la *Memoria* se ha estructurado en 7 capítulos, con el contenido descrito a continuación:

- **Capítulo 1. Introducción.** En este capítulo se presentan los motivos que han dado lugar al planteamiento de este Trabajo Fin de Grado, así como los objetivos que se pretenden satisfacer durante su desarrollo. Asimismo, se incluye el peticionario y la estructura del documento con el fin de proporcionar una idea global del trabajo realizado.

- **Capítulo 2. Introducción a *Bluetooth Low Energy*.** En este capítulo se presentará una introducción a la tecnología de comunicación *Bluetooth Low Energy*, explicando en mayor profundidad los aspectos más relevantes en relación con el presente Trabajo Fin de Grado.

- **Capítulo 3. Introducción a *LoRa*.** En este capítulo se presentará una introducción a la tecnología de comunicación *LoRa*, explicando en mayor profundidad los aspectos más relevantes en relación con el presente Trabajo Fin de Grado.

- **Capítulo 4. Componentes utilizados:** En este capítulo se presentarán los componentes *hardware* y *software* empleados en el presente TFG, analizando sus principales características y funcionalidad, en casa caso.

- **Capítulo 5. Desarrollo de la pasarela basada en el dispositivo LoPy:** En este capítulo se realizará una primera integración de la pasarela BLE-*LoRa*-BLE basada en el dispositivo

LoPy, exponiendo el *firmware* desarrollado para cada dispositivo empleado en esta solución inicial, así como los pasos seguidos para su verificación.

- **Capítulo 6. Desarrollo de la pasarela basada en los dispositivos LoPy y RedBear Duo:** En este capítulo se realizará la integración final de la pasarela BLE-*LoRa*-BLE basada en los dispositivos *RedBear Duo* y *LoPy*, exponiendo el *firmware* desarrollado para cada dispositivo empleado en esta solución final, así como los pasos seguidos para su verificación.

- **Capítulo 7. Conclusiones:** En este capítulo se recogen las conclusiones obtenidas tras haber completado los objetivos propuestos para este Trabajo Fin de Grado.

Finalmente, se presentará la Bibliografía utilizada, así como el Pliego de Condiciones con las condiciones bajo las que se ha desarrollado el presente TFG, el Presupuesto, donde se recogen los gastos generados en la realización del trabajo, y un Anexo con la estructura de los ficheros adjuntos.

CAPÍTULO 2: INTRODUCCIÓN A BLUETOOTH LOW ENERGY

En este capítulo se presenta una introducción a la tecnología de comunicación *Bluetooth Low Energy*, explicando en mayor profundidad los conceptos más relevantes relacionados con el presente Trabajo Fin de Grado.

2.1. Introducción al estándar de comunicación BLE

La tecnología de red de área personal *Bluetooth Low Energy* (BLE) surge a partir de la especificación de *Bluetooth* 4.0, presentándose como una versión reducida y altamente optimizada con respecto a *Bluetooth Classic*, aunque orientada a propósitos completamente diferentes, lo que la convierte en una propuesta independiente. El objetivo principal de BLE es proporcionar una tecnología inalámbrica con el mínimo consumo de potencia posible, y específicamente orientada a soluciones de bajo coste, limitado ancho de banda y reducida complejidad [11].

Si se analiza la evolución histórica de *Bluetooth* con respecto a las velocidades de transmisión de datos de sus diferentes versiones, representada en la Tabla 1, se puede observar cómo, con la aparición de BLE, se consigue un enfoque opuesto, pasando de incrementar la tasa de transferencia de datos a priorizar el consumo de potencia. Como consecuencia, tanto la duración de la conexión como el volumen de datos a transmitir se reducen para satisfacer el bajo consumo.

Versión	Tasa de datos	Nombre característico
1.2	721 kb/s	
2.0 + EDR	3 Mb/s	Enhanced Data Rate (EDR)
3.0 + HS	24Mb/s	High-Speed
4.0	1 Mb/s (BLE)	Bluetooth Low Energy (BLE)

TABLA 1: VERSIONES DE BLE

Otro requisito indispensable de BLE es representar el mínimo coste posible, dado que está orientado a desplegarse en grandes volúmenes, principalmente en dispositivos que no cuentan con ninguna tecnología inalámbrica. Así, para cumplir con estos requisitos de diseño, se han considerado las tres siguientes premisas:

1) Utilizar la banda libre *Industrial Scientific and Medical* (ISM) de 2.4 GHz, con lo que se evita tener que pagar una licencia, lo que se traduce en un ahorro de costes.

2) Dado que en sus inicios *Wibree*, que fue como se denominó en un principio a esta tecnología, eligió el *Bluetooth Special Interest Group* (SIG) para su estandarización, el coste de licencia de los dispositivos Bluetooth se reduce en gran medida.

3) Emplear pilas de botón que, con su tamaño, precio y disponibilidad, se ajustan perfectamente a los requisitos de BLE: bajo coste, baja tasa de datos y reducido consumo.

Por tanto, los diseños orientados al uso de BLE buscarán trabajar con baterías de menor tamaño, precio y con disponibilidad asegurada. Como ya se ha mencionado previamente, este requisito limita la velocidad de transmisión de datos o imposibilita trabajar con un consumo de energía reducido para grandes volúmenes de datos, siendo este aspecto el más determinante a la hora de diferenciar BLE de la especificación *Bluetooth Classic*.

Haciendo hincapié en las premisas descritas con anterioridad, se aclara que la decisión de operar en la banda de 2.4 GHz radica principalmente en la capacidad de operar a nivel global de forma *low-cost* y para altos volúmenes de manufacturación. Por otra parte, al tratarse de una banda libre, está suele encontrarse congestionada. Bandas como 60 GHz ISM no interesaban desde el punto de vista económico, y otras como las bandas 800/900 MHz tienen diferentes frecuencias y reglas dependiendo de la ubicación geográfica, lo que complicaba la globalización del estándar.

En definitiva, resultaba imprescindible diseñar una radio capaz de trabajar en todo momento en un entorno hostil, en cuanto a interferencias se refiere. Esto se consiguió gracias a la utilización de la técnica *Adaptive Frequency Hopping*, que ayuda no solo a detectar rápidamente fuentes de interferencias, sino a evitarlas de forma adaptativa en el futuro. También incluye entre sus características la capacidad de recuperarse rápidamente de la pérdida inevitable de paquetes debida a interferencias. Esta robustez es la clave para el éxito de cualquier tecnología inalámbrica.

2.2. Aspectos generales

Tratándose de una tecnología de comunicación inalámbrica con unos requisitos muy concretos, es normal que BLE presente una serie de limitaciones, inevitables para poder cumplir con los objetivos establecidos en su especificación. Así, BLE no pretende ser una solución para cualquier necesidad de transferencia de datos inalámbricos, sino únicamente cumplir con los requisitos de diseño. Para ayudar a comprender esta tecnología, a continuación, se analizarán sus principales limitaciones, y cómo afectan éstas a los productos BLE [12].

La tasa de modulación de la radio de BLE está establecida a una velocidad constante de 1 MegaBit por segundo (Mbps). Esto determina el límite superior de velocidad a la que un dispositivo BLE puede transmitir, pero generalmente este límite se reduce significativamente debido a una serie de factores, tales como el tráfico bidireccional (en caso de que lo haya), sobrecarga del protocolo, limitaciones de la *Central Processing Unit* (CPU) y radio, etc.

El rango de cualquier dispositivo inalámbrico depende de multitud de factores, como el entorno, el diseño de la antena, o la orientación del dispositivo, aunque en este caso, BLE está especialmente orientado a comunicaciones a distancias muy reducidas.

Normalmente, la potencia de transmisión se configura en un rango comprendido entre -30 decibelio-milivatio (dBm) y 0 dBm, pero mientras mayor sea la potencia, además de ofrecer un mayor rango de operación, se incrementa la demanda en la batería, reduciendo su tiempo de vida. En consecuencia, es posible configurar un dispositivo BLE que pueda transmitir de forma efectiva datos a 30 metros, pero típicamente se configura para un rango de operación entre 2 y 5 metros, con el objetivo de aumentar la vida útil de la batería.

La especificación *Bluetooth* contempla tanto la definición de *Bluetooth Classic* como de *Bluetooth Low Energy*, si bien ambos estándares no son directamente compatibles; un dispositivo con una versión anterior a *Bluetooth* 4.0 no podrá comunicarse con un dispositivo BLE. Por ello, existen dos tipos de dispositivos que implementan BLE:

- *Dual-mode*: dispositivo que soporta tanto BLE como *Bluetooth Classic*.
- *Single-mode*: dispositivo que solo soporta BLE.

También existen los dispositivos *Classic*, los cuales no soportan BLE. En la Tabla 2 se muestra cómo se comunicarían los diferentes dispositivos.

	Single-Mode	***Dual-Mode***	***Classic***
Single-Mode	LE	LE	no
Dual-Mode	LE	*Classic*	*Classic*
Classic	no	*Classic*	*Classic*

TABLA 2: COMUNICACIÓN DISPOSITIVOS BLUETOOTH

2.3. Arquitectura

La arquitectura BLE se divide en tres unidades básicas, a saber: *Application*, *Host*, y *Controller*, las cuales describen y recogen las partes indispensables que conforman un dispositivo BLE, desde la antena hasta la interfaz de usuario. Generalmente, el usuario final interactuará únicamente con las capas superiores de la pila de protocolo BLE, mientras que, en el caso particular de este TFG, se trabajará en su mayoría con el *Host*, teniendo especial importancia la definición de los perfiles GAP y GATT, descritos más adelante. En la Figura 1 se recoge cada uno de los bloques básicos en los que se divide un dispositivo BLE *single-mode* completo.

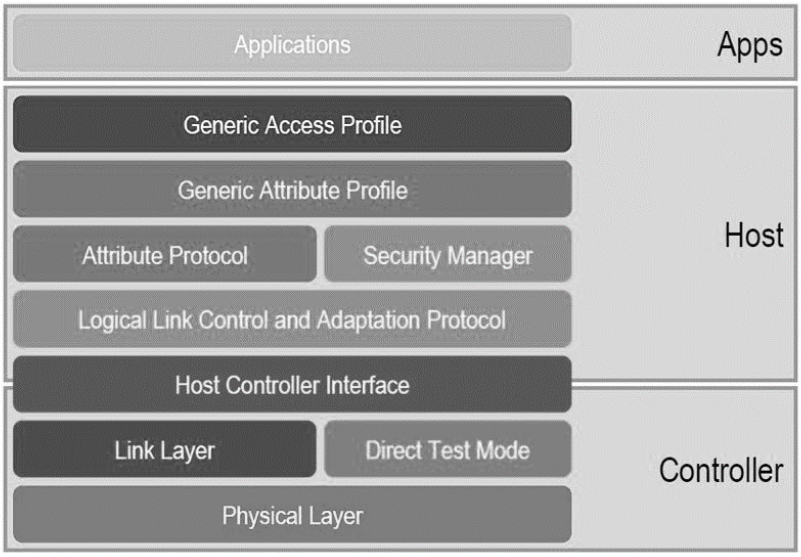

FIGURA 1: ARQUITECTURA BLE

En términos generales, se puede describir el **controlador** como el conjunto de partes analógicas y digitales de los componentes de radiofrecuencia, así como el hardware encargado de la transmisión y recepción de paquetes. Se interconecta con el mundo exterior a través de una antena, y al *Host* a través de la *Host Controller Interface* (HCI).

El ***Host*** contiene las capas de multiplexación, los diferentes protocolos que definen BLE, y las bases para la comunicación entre dispositivos. Está construido encima de la interfaz *Host Controller Interface*. En el nivel más cercano a esta interfaz se encuentra el protocolo *Logical Link Control and Adaptation Protocol* (L2CAP). A continuación, se encuentran dos bloques fundamentales para el sistema: el protocolo de seguridad *Security Manager Protocol* (SMP), encargado de la autenticación y la configuración de conexiones seguras, y el protocolo de atributos *Attribute Protocol* (ATT), que define cómo se transfieren los datos. Basado en el protocolo ATT se encuentra el perfil *Generic Attribute Profile* (GATT), que define cómo se utiliza el protocolo ATT para habilitar los servicios que exponen las características de un dispositivo. Finalmente, el perfil *Generic Access Profile* (GAP) define cómo los dispositivos se encuentran y se interconectan. Por encima del *Host* no hay una interfaz superior definida, cada sistema operativo o entorno de desarrollo tendrá una forma diferente de presentar las *Application Programming Interface* (API) del *Host*, ya sea a través de una interfaz funcional u orientada a objetos.

Por encima del controlador y el *Host* está la capa de **aplicación**, en la cual se describen tres tipos de especificaciones: característica, servicio y perfil. Cada una de estas especificaciones se construye sobre el perfil GATT, donde se definen grupos de atributos que contienen las

características y servicios. Por otro lado, las aplicaciones definen las especificaciones que utilizan estos grupos de atributos.

2.3.1. DIFERENCIA ENTRE PROTOCOLO Y PERFIL

La especificación de *Bluetooth* introdujo desde sus inicios una diferenciación clara entre los conceptos de Protocolo y Perfil:

- **Protocolos**: bloques utilizados por todos los dispositivos que siguen la especificación *Bluetooth*. Son las capas que implementan los diferentes formatos de paquete, encaminamiento, multiplexación, codificación y decodificación, que permiten que los datos se envíen de manera efectiva entre dispositivos.

- **Perfiles**: "divisiones verticales" de funcionalidad que cubren los modos básicos de operación requeridos por todos los dispositivos (GAP, GATT) o usos en casos específicos (Perfil de Glucosa, Perfil de Proximidad). Esencialmente, definen cómo deben usarse los protocolos para lograr un objetivo particular, ya sea genérico o específico. Un Perfil no existe realmente en el propio dispositivo *Peripheral* BLE, es simplemente una colección de Servicios predefinidos, que ha sido compilada por el *Bluetooth* SIG o por los mismos diseñadores de periféricos.

2.4. Topología de red

Los dispositivos BLE tienen dos formas principales de comunicarse con el medio, a través de *Broadcasting* o mediante conexiones. Cada mecanismo tiene sus ventajas y limitaciones, y ambos están sujetos a las pautas establecidas por el perfil *Generic Access Profile* (GAP), descrito más adelante.

2.4.1. BROADCAST

Mediante *Broadcast*, sin establecimiento de conexión, se consigue enviar datos a cualquier dispositivo receptor o en proceso de *scanning* que se encuentre dentro del rango de escucha. En la Figura 2 se representa cómo este mecanismo lo que habilita es, básicamente, una comunicación unidireccional en la que un dispositivo hace de *Broadcaster*, y todos los demás adoptan el rol de observador, limitándose a recibir los datos enviados.

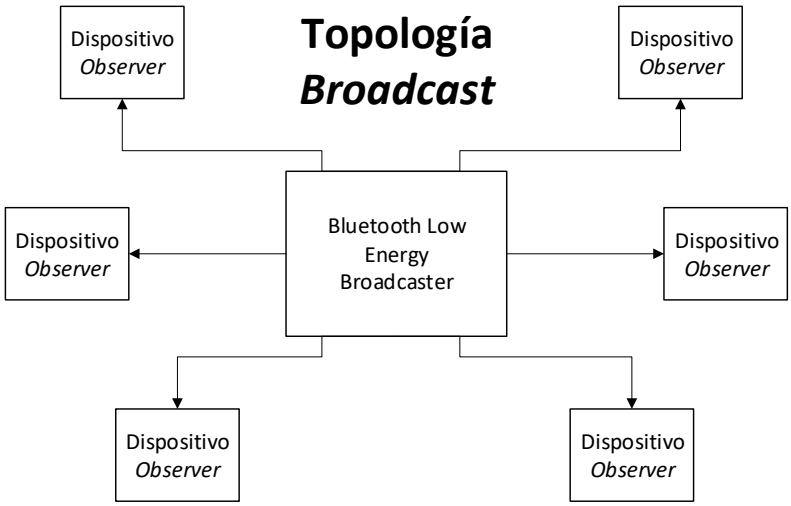

FIGURA 2: TOPOLOGÍA BROADCAST

El dispositivo con el rol de *Broadcaster* envía paquetes de *advertising*, no conectables y de forma periódica, a cualquier dispositivo dispuesto a recibirlos. Por una parte, el dispositivo que actúa como Observador escanea repetidamente las frecuencias preestablecidas para recibir cualquier paquete de *advertising* no conectable que se esté transmitiendo en ese momento.

El paquete de *advertising* estándar tiene un *payload* de 31 bytes. En caso de no ser suficiente, BLE cuenta con un paquete de *advertising* secundario y opcional de otros 31 bytes.

Broadcasting es la única forma que se tiene de transmitir datos a más de un dispositivo BLE a la vez. Es un método rápido y sencillo de usar, siendo una elección adecuada cuando se desea transmitir una pequeña cantidad de datos en un horario fijo determinado, o para múltiples dispositivos. Desafortunadamente, cuenta con una gran limitación, no ofrece garantías en seguridad o privacidad, por lo que no es ideal para información sensible.

2.4.2. CONEXIONES

Si se pretende establecer una transmisión bidireccional, o si se precisa enviar más datos de los que caben en los dos *advertising payloads*, se necesitará de una conexión. Una conexión se puede definir como un intercambio de paquetes, de forma permanente y periódica, entre dos dispositivos BLE. Es, por tanto, inherentemente privada, ya que solo hay dos dispositivos involucrados en la comunicación. Las conexiones incluyen dos roles diferenciados:

- *Central* (*master*): dispositivo que escanea repetidamente las frecuencias preestablecidas en busca de paquetes de *advertising* y, cuando sean adecuados, inicia una conexión. Una

vez establecida la conexión, el dispositivo *Central* controla la temporización e inicia los intercambios periódicos de datos.

- *Peripheral* (*slave*): dispositivo que envía paquetes de *advertising* conectables periódicamente y acepta conexiones entrantes. Cuando exista una conexión activa, el dispositivo *Peripheral* sigue la temporización establecida por el dispositivo *Central*, intercambiando datos regularmente con él.

Para iniciar una conexión, el dispositivo *Central* recoge los paquetes de *advertising* de un dispositivo *Peripheral* y le envía una solicitud para establecer una conexión exclusiva entre ambos. Cuando la conexión se haga efectiva, el dispositivo *Peripheral* abandona el proceso de *advertising* y comienza el intercambio de datos. En la Figura 3 se muestra un ejemplo de este tipo de topología.

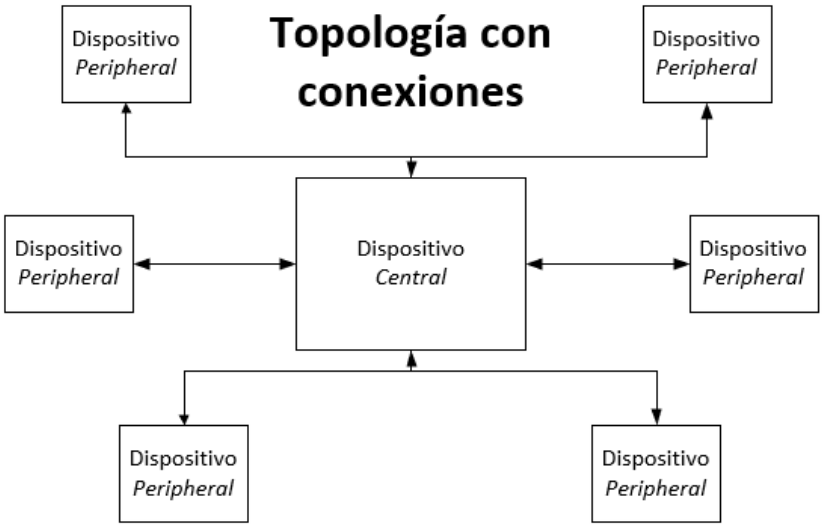

FIGURA 3: TOPOLOGÍA CON CONEXIONES

En definitiva, una conexión no es más que un intercambio periódico de datos entre dos dispositivos. Hay que aclarar que a pesar de ser el dispositivo maestro el que gestiona el establecimiento de conexión, ambos dispositivos pueden enviar datos, y los roles no imponen restricciones en cuanto al rendimiento de datos o la prioridad.

A partir de la versión 4.1 de la especificación de *Bluetooth Low Energy*, se eliminan las restricciones en cuanto a las combinaciones de roles: un dispositivo puede actuar tanto como *Central* como *Peripheral*, además de que un dispositivo *Central* puede establecer conexión con múltiples dispositivos *Peripheral* y viceversa, pudiendo un dispositivo *Peripheral* estar vinculado a

varios dispositivos *Central*. La principal ventaja de las conexiones es la capacidad que ofrecen para organizar los datos con un control mucho más preciso de cada campo o propiedad, gracias al uso de capas de protocolo adicionales y, más concretamente, al perfil *Generic Attribute Profile* (GATT), en el que los datos se organizan en unidades denominadas Servicios y Características.

Las conexiones permiten establecer un modelo de datos en capas mucho más completo. También proporcionan el potencial de usar mucha menos energía que el modo *Broadcast*, dado que pueden extender el retraso entre eventos de conexión, o enviar datos solo cuando hay nuevos valores disponibles, en lugar de tener que publicitar continuamente el *data payload* sin saber quién está escuchando y con qué frecuencia. Y no solo eso, sino que al saber cuándo se producirán los eventos de conexión, puede desactivar la radio durante mayor tiempo, lo que representa un ahorro de la batería. Finalmente, indicar que estas topologías se pueden combinar en una red BLE, como se muestra en la Figura 4.

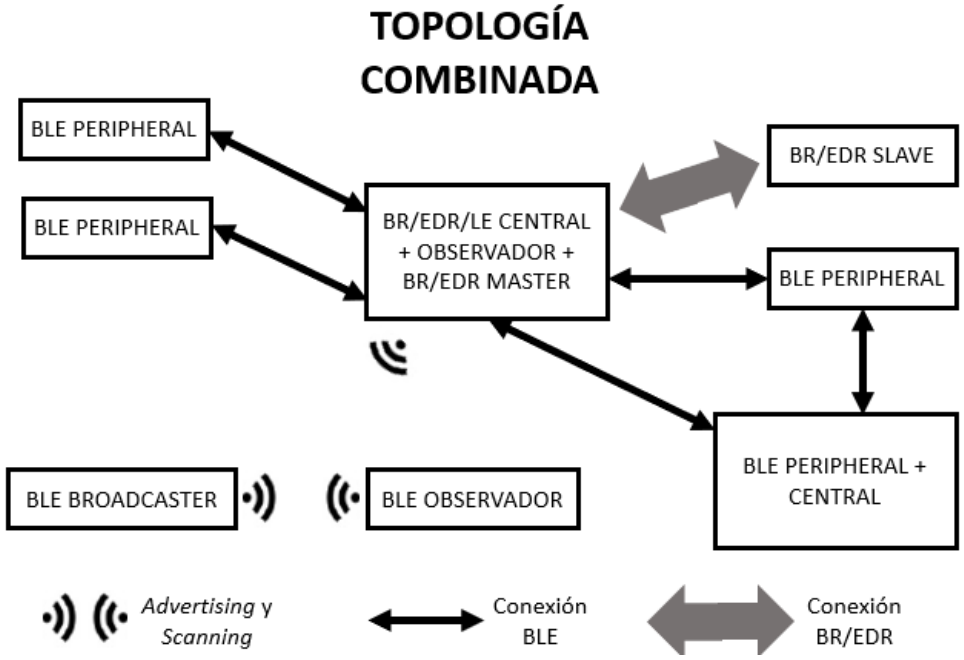

FIGURA 4: EJEMPLO DE TOPOLOGÍA MIXTA

2.5. Controller

El controlador se puede definir como el chip *Bluetooth* o la radio. Sin embargo, esta descripción es bastante simplista. Una descripción más acertada sería definir el controlador como el conjunto de partes analógicas y digitales de los componentes de radiofrecuencia, así como el *hardware* para la transmisión y recepción de paquetes.

2.5.1. CAPA FÍSICA (PHY)

La capa física desempeña la tarea de transmitir y recibir bits usando la radio de 2.4 Gigahercios (GHz) y 40 canales, de los cuales 37 se usan para datos de conexión, y tan solo 3 para el proceso de *advertising* [13], como se puede observar en la Figura 5. Normalmente, las ondas de radio transmiten información variando la amplitud, la fase o la frecuencia de la onda, dentro de una banda de frecuencia determinada. En BLE, se utiliza la variación en frecuencia de las ondas de radio para enviar un 0 o un 1, utilizando el esquema de modulación por desplazamiento de frecuencia gausiana (GFSK).

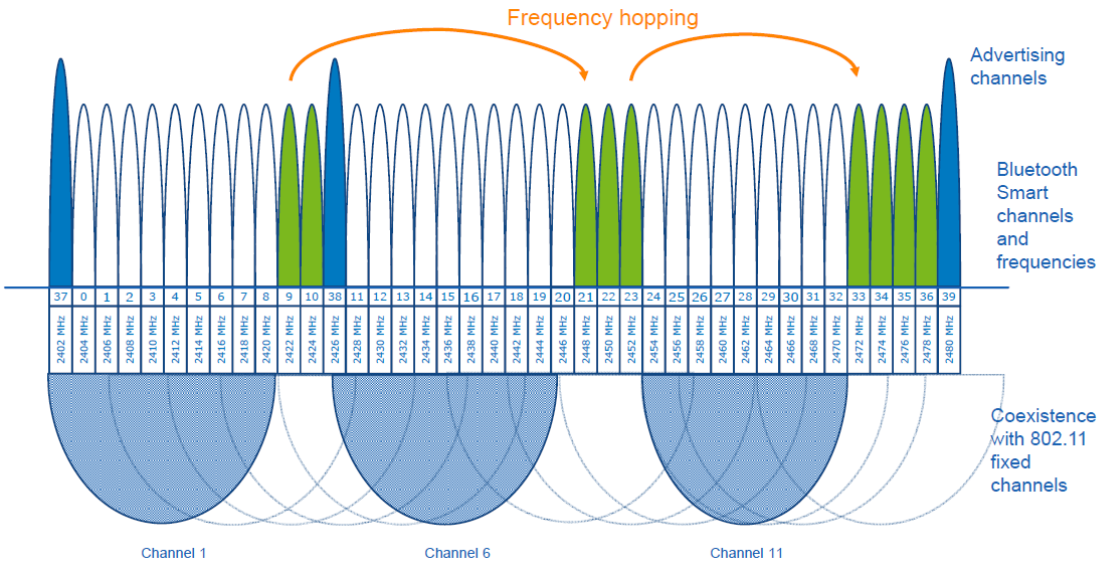

FIGURA 5: CANALES BLE Y SUS FRECUENCIAS

La codificación por desplazamiento de frecuencia significa que los unos y los ceros se codifican desplazando ligeramente la frecuencia hacia arriba y hacia abajo. Si la frecuencia se desplaza bruscamente hacia cualquiera de los lados, en el momento que se produce este cambio se origina un impulso de energía que se extiende sobre un rango más amplio de frecuencias. Por tanto, se utiliza un filtro para detener la propagación de la energía en frecuencias más altas o bajas. En el caso de GFSK, el filtro utilizado tiene la forma de una curva gaussiana. El filtro usado en BLE no es tan ajustado como el de *Bluetooth Classic*, lo que significa que la señal de radio de baja energía se extiende un poco más que la señal de radio clásica.

Esta ligera ampliación de la señal, definida por el índice de modulación, significa que la radio está sujeta a las regulaciones de radio de amplio espectro, que permiten que una radio transmita en menos frecuencias que la radio de *Bluetooth Classic*, que se rige por las regulaciones de radio de

salto en frecuencia. Sin este filtro con forma más suavizada, la radio BLE no podría realizar el proceso de *advertising* en solo tres canales, sino que tendría que usar muchos más, lo que aumentaría la potencia del sistema.

El índice de modulación describe cómo será el ancho de las frecuencias superiores e inferiores con respecto a una frecuencia central. Cuando se transmite la señal de radio, una desviación de frecuencia positiva de más de 185 Kilohercios (KHz) de la frecuencia central representa un bit de valor 1; una desviación de frecuencia negativa de más de 185 KHz representa un bit de valor 0.

Como ya se ha mencionado, la banda de 2.4 GHz se divide en 40 canales de Radiofrecuencia (RF) separados 2 Megahercios (MHz) entre sí. Además, hay que mencionar que la capa física transmite información a una velocidad de 1 bit de datos de aplicación cada microsegundo.

2.5.2. DIRECT TEST MODE

El modo *Direct Test Mode* se encarga de testear la capa física. En la mayoría de los estándares inalámbricos no existe una manera estandarizada de hacer que un dispositivo ejecute pruebas a la capa física. Como resultado, los fabricantes patentan sus propios métodos de testeo, lo que se traduce en un aumento del coste económico en toda la industria. El modo *Direct Test Mode* permite que un *tester* transmita o reciba una secuencia de paquetes de prueba, para posteriormente analizarlos, ya sea los mismos paquetes o el número de paquetes recibidos por el dispositivo bajo prueba, para determinar si el funcionamiento de la capa física cumple las especificaciones.

El *tester* también puede medir varios parámetros de RF de los paquetes recibidos para determinar si la capa física cumple con las especificaciones de RF. El modo *Direct Test Mode* no solo es aplicable a pruebas de calidad, también se puede utilizar para realizar pruebas en líneas de producción y calibración de radios.

2.5.3. CAPA DE ENLACE (LL)

La capa de enlace es la parte más compleja de la arquitectura BLE. Es responsable de los procesos de *advertising*, *scanning*, así como de la creación y mantenimiento de las conexiones. También es responsable de garantizar que los paquetes sigan la estructura correcta, con los valores de verificación y las secuencias de cifrado correctamente calculados. Para ello, se definen tres conceptos básicos: canales, paquetes y procedimientos.

Existen dos tipos de canales en la capa de enlace: canales de *advertising* y canales de datos. Los canales de *advertising* son utilizados por dispositivos que no se encuentran en una conexión. Como ya se ha comentado, de los 40 canales disponibles, solo 3 canales de *advertising*, los cuales son usados por los dispositivos para transmitir datos en *broadcast*, para anunciar que son conectables y descubribles mediante el proceso de *advertising*, y para el proceso de *scanning* y establecimiento de conexión. Los canales de datos solo se utilizan una vez establecida una conexión. Hay 37 canales de datos usados mediante la técnica de *Adaptative Frequency Hopping*. Los canales de datos permiten que entre dispositivos se envíen datos, se confirme su recepción y, si es necesario, se reenvíen. Además, estos canales se pueden encriptar e incluir autenticación de paquetes.

Para enviar datos en cualquiera de estos canales (datos o *advertising*), se definen los paquetes. Un paquete encapsula una pequeña cantidad de datos que se envían desde un transmisor a un receptor en un período de tiempo muy corto. Los paquetes incluyen información para poder identificar al receptor deseado, así como un *checksum* que garantiza la validez del paquete. La estructura básica de los paquetes es idéntica para ambos tipos de canal, con un mínimo de 80 bits de direccionamiento, encabezado e información de verificación incluidos en cada paquete. En la Figura 6 se presenta una descripción general de la estructura general de los paquetes de la capa de enlace [14].

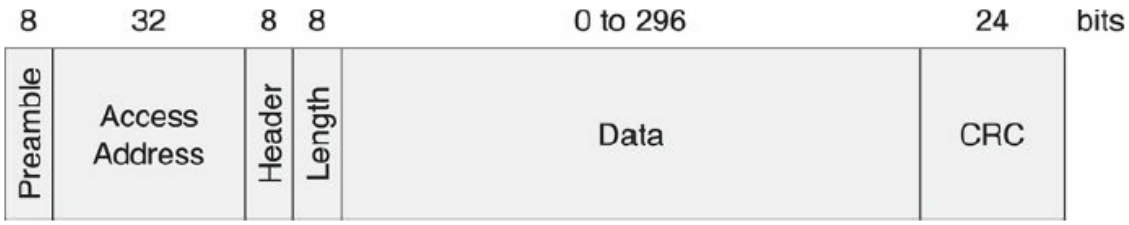

FIGURA 6: ESTRUCTURA DE LOS PAQUETES EN LA CAPA DE ENLACE

Los paquetes están optimizados para aumentar su robustez mediante el uso de un *preamble* de 8 bits, suficientemente grande para sincronizar el timing de los bits y configurar el control automático de ganancia de la radio; una dirección de acceso de 32 bits, fija para paquetes de *advertising*, pero completamente aleatoria y privada para paquetes de datos; una cabecera de 8 bits para describir el contenido del paquetes; un *payload* de 8 bits para indicar la longitud de la carga útil; una carga útil de longitud variable que contiene datos útiles de la aplicación o la pila de dispositivos del *host*; y finalmente, un valor de verificación de redundancia cíclica (CRC) de 24 bits para garantizar que no hayan bits erróneos en el paquete recibido.

El paquete más pequeño que se puede enviar tiene una longitud de 80 microsegundos (µs), es decir, un paquete de datos vacío. En cambio, el paquete más grande es un paquete de *advertising* completamente cargado, teniendo una latencia de 376 µs. A pesar de este margen en cuanto a longitud, la mayoría de los paquetes de *advertising* tienen un tamaño de solo 128 µs y la mayoría de los paquetes de datos un tamaño de 144 µs.

Para comprender la capa de enlace es necesario conocer la máquina de estados de la capa de enlace y sus implicaciones en el diseño de BLE. En la Figura 7 se puede ver la relación entre los 5 estados que componen la máquina de estados de la capa de enlace:

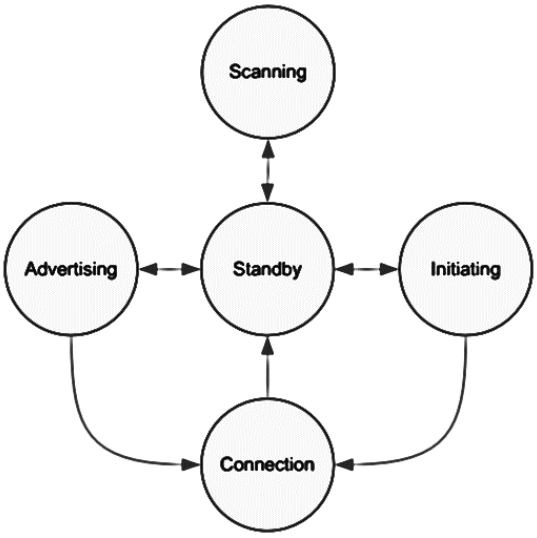

FIGURA 7: MÁQUINA DE ESTADOS CAPA DE ENLACE BLE

- ***Standby***: una vez que las capas de enlace reciben alimentación y se encienden, inician en el estado de *standby*, permaneciendo en él hasta que las capas de *host* le indiquen lo contrario. A partir de este estado es posible pasar a los estados de *advertising*, *scanning* o *initiating*. Además, cualquier otro estado puede pasar al de *standby*, siendo este el más importante, aunque inactivo.

- ***Advertising***: el estado de *advertising* permite que la capa de enlace transmita paquetes de *advertising*. En él también se puede responder a las solicitudes de *scanning* de dispositivos que se encuentren de forma activa en el proceso de *scanning*, enviando una respuesta en forma de *scan response*. El estado de *advertising* es obligatorio si un dispositivo quiere ser detectable o conectable. También se emplea este estado cuando un dispositivo quiere transmitir datos en modo *broadcast*. Es posible pasar del estado de *advertising* al estado de *standby* deteniendo el proceso de *advertising*.

- ***Scanning*:** en el estado de *scanning*, un dispositivo recibirá paquetes de los canales de *advertising*. Esto podría usarse para ver qué dispositivos se encuentran en proceso de *advertising* en el área. El estado correspondiente al *scanning* se compone de dos subestados: *scanning* pasivo y *scanning* activo. El *scanning* pasivo solo recibe paquetes de *advertising*. El activo también envía solicitudes de *scanning* a dispositivos en proceso de *advertising*.

- ***Initiating***: para iniciar una conexión con otro dispositivo, la capa de enlace primero tiene que pasar al estado de inicio. En este estado, el receptor se encuentra en proceso de escucha, esperando por los paquetes de *advertising* del dispositivo al que quiere conectarse. En caso de recibir un paquete de *advertising* de este dispositivo, la capa de enlace enviará una solicitud de conexión al dispositivo en proceso de *advertising*, pasando al estado *connection*. En este estado también se puede pasar al estado *standby*, deteniendo por tanto la conexión.

- ***Connection:*** el estado final de la máquina de estados de la capa de enlace es el estado de conexión, denominado *connection*. Se puede entrar en este estado a partir del estado de *advertising* o del estado *initiating*. Ambas transiciones son causadas por un dispositivo en estado de inicio que envía un paquete de solicitud de conexión a un dispositivo en estado de *advertising*. De nuevo, este tiene dos subestados: maestro o esclavo. En el estado de conexión, los paquetes del canal de datos se envían y reciben entre los dos dispositivos. Este es el único estado en el que se utilizan los canales de datos; todos los demás utilizan los canales de *advertising*. Solo es posible dejar el estado de conexión moviéndose al estado de espera, es decir, terminando la conexión.

2.5.4. HOST/CONTROLLER INTERFACE (HCI)

Para una gran variedad de dispositivos se proporcionará una interfaz entre el *host* y el controlador, HCI, que permite que un *host* se comunique con el controlador a través de una interfaz estándar. Esta división a nivel de arquitectura ha demostrado ser extremadamente útil en el *Bluetooth Classic*, donde más del 60% de los controladores *Bluetooth* se utilizan a través de la interfaz HCI. Se compone de dos partes diferenciadas: la interfaz lógica y la interfaz física.

- **Interfaz lógica:** define los comandos y eventos, así como su comportamiento asociado. La interfaz lógica puede proporcionarse a través de cualquiera de los transportes físicos, o a

través de una interfaz de programación de aplicaciones (API) local en el controlador, lo que permite incluir una pila de *host* integrada dentro del controlador.

- **Interfaz física:** define cómo se transportan los comandos, eventos y datos a través de diferentes tecnologías de conexión. Las interfaces físicas que se definen incluyen *Universal Serial Bus* (USB), *Secure Digital Input Output* (SDIO) y dos variantes de la *Universal Asynchronous Receiver-Transmitter* (UART), aunque la mayoría de los controladores admitirán solo una o dos interfaces.

2.6. Host

La capa *Host* se sitúa justo debajo de la capa de aplicación, y consta de múltiples protocolos de red y transporte que permiten a las aplicaciones comunicarse con dispositivos pares de una forma estandarizada.

2.6.1. Logic Link Control and Adaptation Protocol (L2CAP)

El protocolo L2CAP es la capa de multiplexación para BLE. Esta capa define dos conceptos básicos: el canal L2CAP y los comandos de señalización L2CAP. Un canal L2CAP es un canal de datos bidireccional único que termina en un determinado protocolo o perfil en su dispositivo par. Cada canal es independiente y puede tener su propio control de flujo e información de configuración adicional. Mientras que *Bluetooth Classic* utiliza la mayoría de las funciones de L2CAP, BLE solo utiliza el mínimo indispensable del protocolo L2CAP. Solo se utilizan canales fijos, uno para el canal de señalización, uno para el protocolo *Security Manager* y otro para el protocolo de atributos.

2.6.2. Security Manager Protocol (SMP)

El administrador de seguridad define un protocolo simple para el *pairing* y distribución de claves. El *pairing* es el proceso de intentar "confiar" en otro dispositivo, generalmente mediante la autenticación del otro dispositivo. Este proceso suele ir seguido del cifrado del enlace y la distribución de la clave. Gracias a la distribución de claves, la información sensible se puede distribuir entre maestro y esclavo, de modo que, en futuras conexiones, el proceso de autenticación se realizará más rápidamente, gracias a estos datos previamente compartidos.

2.6.3. ATTRIBUTE PROTOCOL (ATT)

ATT es simplemente un protocolo Cliente/Servidor basado en los atributos presentados por un dispositivo. En BLE, cada dispositivo es un Cliente, un Servidor, o ambos, independientemente de si es Maestro o Esclavo. Un Cliente solicita datos de un Servidor y un Servidor envía datos a los Clientes. El protocolo es estricto cuando se trata de su secuenciación: si una solicitud aún está pendiente (aún no recibió respuesta), no se pueden enviar más solicitudes hasta que la respuesta se reciba y procese. Esto se aplica de forma independiente en el caso en que dos pares actúen como Cliente y Servidor.

Cada Servidor contiene datos organizados en forma de atributos, a cada uno de los cuales se le asigna un identificador de atributo de 16 bits, un identificador único universal (UUID), un conjunto de permisos y, por supuesto, un valor. El identificador de atributo es utilizado simplemente para acceder a un valor de atributo. El UUID especifica el tipo y la naturaleza de los datos contenidos en el valor.

Cuando un Cliente desea leer o escribir valores de atributos desde, o hacia un Servidor, emite una solicitud de lectura o escritura al Servidor con el identificador. El Servidor responderá con el valor del atributo o un acuse de recibo. En el caso de una operación de lectura, le corresponde al Cliente analizar el valor y comprender el tipo de datos en función del UUID del atributo. Por otro lado, durante una operación de escritura, se espera que el Cliente proporcione datos que sean consistentes con el tipo de atributo y el Servidor puede rechazar la operación si ese no es el caso.

2.6.4. GENERIC ATTRIBUTE PROFILE (GATT)

El perfil *Generic Attribute Profile* establece en detalle cómo intercambiar todos los datos de perfil y de usuario a través de una conexión BLE. A diferencia del perfil GAP, que define las interacciones de bajo nivel con los dispositivos, el perfil GATT solo se ocupa de los procedimientos y formatos de transferencia de datos.

El perfil GATT también proporciona un marco de referencia para todos los perfiles basados en el perfil GATT, que cubre usos específicos y garantiza la interoperabilidad entre dispositivos de diferentes proveedores. Por lo tanto, todos los perfiles BLE estándar se basan en GATT, y deben cumplirlo para funcionar correctamente. Esto convierte al perfil GATT en una sección clave de la especificación BLE, ya que cada elemento de datos relevante para las aplicaciones y usuarios debe formatearse, empaquetarse y enviarse según sus reglas.

Importante mencionar que el perfil GATT utiliza el protocolo ATT, así como su protocolo de transporte, para intercambiar datos entre dispositivos. Estos dispositivos están organizados jerárquicamente en secciones llamadas Servicios, que agrupan conceptualmente piezas de datos de usuario llamadas Características.

2.6.4.1. Roles del perfil GATT

Al igual que con cualquier otro protocolo o perfil en la especificación *Bluetooth*, el perfil GATT tiene una serie de roles definidos que los dispositivos pueden adoptar:

- **Cliente**: el Cliente GATT corresponde al Cliente del protocolo ATT. Envía solicitudes a un Servidor y recibe respuestas. El Cliente GATT no conoce previamente nada acerca de los atributos del Servidor, por lo que primero debe consultar acerca de la presencia y la naturaleza de esos atributos, realizando el descubrimiento del Servicio. Después de completar el descubrimiento del Servicio, puede comenzar a leer y escribir atributos que se encuentren en el Servidor, así como recibir actualizaciones iniciadas por el Servidor.

- **Servidor**: el Servidor GATT corresponde al Servidor del protocolo ATT. Recibe solicitudes de un Cliente y devuelve respuestas. También envía actualizaciones iniciadas por el Servidor cuando esté configurado para hacerlo, y es el rol responsable de almacenar y poner a disposición del Cliente los datos del usuario, organizados en atributos. Cada dispositivo BLE debe incluir al menos un Servidor GATT básico que pueda responder a las solicitudes de los Clientes, aunque solo sea para devolver una respuesta de error.

Hay que mencionar nuevamente que las funciones del perfil GATT son completamente independientes de los roles GAP, y que también son compatibles entre sí. Esto significa que tanto un dispositivo Central GAP como un dispositivo *Peripheral* GAP pueden actuar como un Cliente o Servidor GATT, o incluso actuar como ambos al mismo tiempo.

2.6.4.2. Universally Unique Identifier (UUID)

El UUID es un número de 128 bits (16 bytes) que se garantiza que es único a nivel global. Los UUID se utilizan en muchos protocolos y aplicaciones distintas de *Bluetooth*, y su formato, uso y generación se especifican en la *International Organization for Standardization* (ISO)/*International Electrotechnical Commission* (IEC) 9834-8: 2005.

El SIG proporciona UUIDs para todos los tipos, servicios y perfiles que define y especifica. Pero si una aplicación necesita el suyo propio, ya sea porque las que ofrece el SIG no cubren los requisitos, o porque quiere implementar un nuevo uso no considerado previamente en las especificaciones de perfil, se pueden generar en la página de la *International Telecommunication Union* (ITU).

2.6.4.3. Atributos

Los atributos son la entidad de datos más pequeña definida por el perfil GATT. Son piezas direccionables de información que pueden contener datos de usuario (o metadatos) relevantes sobre la estructura y la agrupación de los diferentes atributos contenidos en el servidor. Tanto el perfil GATT como el protocolo ATT solo pueden funcionar con atributos, por lo que para que los Clientes y Servidores interactúen, toda la información debe organizarse de esta forma.

Conceptualmente, los atributos siempre se ubican en el Servidor, a los que el Cliente puede acceder (y potencialmente modificar). La especificación define atributos solo conceptualmente, y no obliga a las implementaciones ATT y GATT a usar un formato o mecanismo de almacenamiento interno particular, aunque generalmente los atributos se almacenan en una mezcla entre memoria no volátil y *Random Access Memory* (RAM). Cada atributo, además de los datos, contiene información sobre el atributo en sí, la cual se divide en los campos descritos a continuación.

- **Handle**: es un identificador único de 16 bits para cada atributo en un Servidor GATT particular. Es la parte que convierte a un atributo en direccionable.
- **Tipo**: el tipo del atributo no es más que un UUID. Determina el tipo de dato presente en el valor del atributo.
- **Permisos**: metadato que especifica qué operaciones ATT pueden ejecutarse en cada atributo y bajo qué requerimientos específicos de seguridad.
- **Valor**: parte del atributo que contiene los datos. No hay restricciones sobre el tipo de datos que puede contener, aunque su tamaño está limitado a 512 bytes.

2.6.4.4 Jerarquía del perfil GATT

El perfil GATT establece una estricta jerarquía para organizar los atributos de forma reutilizable y práctica, permitiendo el acceso y la recuperación de información entre el Cliente y el Servidor, siguiendo un conciso conjunto de reglas que constituyen las bases seguidas por todos los perfiles basados en GATT. La Figura 8 ilustra esta jerarquía.

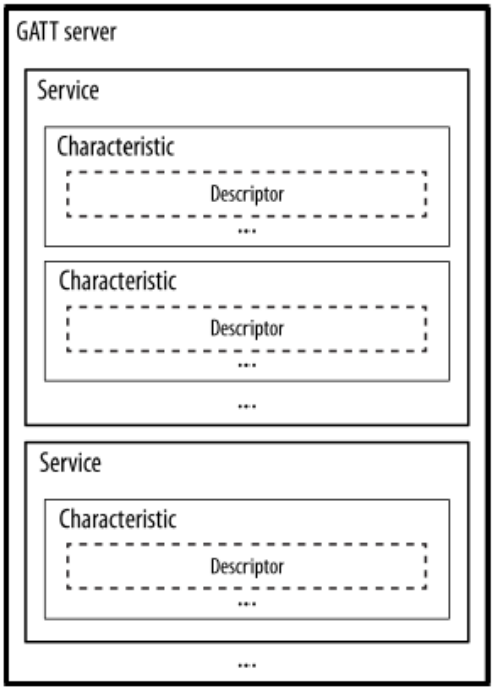

FIGURA 8: JERARQUÍA GATT

Los atributos en un Servidor GATT se agrupan en Servicios, los cuales a su vez pueden contener Características. Las Características, a su vez, pueden incluir Descriptores. Esta jerarquía se aplica estrictamente en cualquier dispositivo con compatibilidad GATT, lo que significa que todos los atributos en un Servidor GATT están incluidos en una de estas tres categorías, sin excepciones.

Servicios

Los Servicios son colecciones de Características y relaciones con otros Servicios que encapsulan el comportamiento de parte de un dispositivo. Es importante comprender la diferencia entre Servicios Primarios y Secundarios. Un Servicio Primario es el tipo de Servicio estándar del GATT, que incluye la funcionalidad estándar relevante expuesta por el Servidor del GATT. Por otro lado, un Servicio Secundario está pensado para ser incluido solamente en otros Servicios Primarios, como modificador. En la práctica, rara vez se utilizan los Servicios Secundarios.

Características

Así, pueden considerarse las Características como contenedores para los datos del usuario. Siempre incluyen al menos dos atributos: la declaración de la Característica (metadatos sobre los datos de usuario) y el valor de la Característica. Además, el valor de la Característica puede ir seguido de Descriptores, que amplían aún más los metadatos contenidos en la declaración de la Característica.

I. Lectura de Características y Descriptores

Para obtener el valor actual de una Característica concreta o de un Descriptor, el Cliente tiene las siguientes opciones:

- **Leer el valor de una Característica o Descriptor**: se puede utilizar para leer el contenido de una Característica o de un Descriptor utilizando su *Handle*.

- **Leer el valor de una Característica o Descriptor, cuando la longitud del valor es mayor:** se puede utilizar si el valor es demasiado largo para leerse con el método anterior. Para poder leer el valor de la Característica o el Descriptor, se incluye un *Offset* además del *Handle* en la solicitud, dividiendo el valor en fragmentos sucesivos. Dependiendo del tamaño del valor del atributo que se está leyendo, se pueden requerir múltiples pares de solicitud/respuesta.

Adicionalmente, se pueden emplear los siguientes métodos, solo aplicables en Características:

- **Leer el valor de una Característica por medio de su UUID:** cuando un Cliente no conoce el *Handle* específico de una o varias Características, puede leer el valor de todas las Características de un tipo específico. El Cliente proporciona un rango de *Handles* y un UUID, recibiendo como respuesta un *array* con los valores de las Características dentro del rango especificado.

- **Leer el valor de múltiples Características:** por otro lado, si un Cliente ya conoce los identificadores de las características de las que desea conocer el valor, puede enviar una solicitud con el conjunto de *Handles*, y posteriormente recibir sus valores.

II. Escritura de Características y Descriptores

Para escribir el valor de una Característica o Descriptor, el Cliente cuenta con las siguientes opciones:

- **Escribir el valor de una Característica o Descriptor:** se puede utilizar para escribir el valor de una Característica o Descriptor. El cliente proporciona un *Handle* y el contenido del valor a escribir. El Servidor confirmará la operación de escritura con una respuesta.

- **Escribir el valor de una Característica o Descriptor, cuando la longitud del valor es mayor:** permite a un Cliente escribir un valor mayor que en el método anterior. Consiste en encolar varias operaciones *prepare write*, las cuales incluyen un *Offset* y los datos a escribir. Finalmente, se escriben atómicamente con una operación *execute write*.

Adicionalmente, se pueden emplear los siguientes métodos, solo aplicables a las Características:

- **Escribir sin respuesta:** esta opción supone lo opuesto al empleo de *Notifications* (explicadas en el siguiente apartado) y hace uso de paquetes *Write Command*. Los paquetes *Write Command* son paquetes sin confirmación, que incluyen un *Handle* y el valor a escribir. Se pueden enviar en cualquier momento y cantidad, sin ningún mecanismo de control de flujo. El Servidor es libre de descartar los paquetes, y el Cliente nunca lo sabrá.

- ***Reliable writes**:* se emplean cuando un Cliente desea encolar las operaciones de escritura de un conjunto de Características. Se envía un paquete final al Servidor para informar de las operaciones de escritura pendientes y ejecutarlas.

III. Server-Initiated Updates

Server-Initiated Updates hace referencia a las actualizaciones iniciadas por el Servidor. Son los únicos paquetes asíncronos, es decir, no como respuesta a una solicitud de un Cliente, que se pueden transmitir del Servidor al Cliente. Estas actualizaciones se envían periódicamente para informar sobre cambios en el valor de una Característica, sin tener el Cliente que comprobarlo continuamente por técnicas de *polling*, con lo que se ahorra en energía y ancho de banda. Existen dos tipos de *Server-Initiated Updates*:

- ***Notifications:*** son paquetes que incluyen el *Handle* de una Característica y su valor actual. El Cliente recibe estos paquetes y puede decidir si actuar o no, pero no confirma su recepción al Servidor. Junto con el método de escritura sin respuesta, este es el único paquete que no cumple con el estándar de control de flujo de solicitud/respuesta.

- ***Indications:*** por otro lado, los paquetes de *Indications*, siguen el mismo formato de *Handle/valor*, pero requieren una confirmación explícita por parte del Cliente. Hay que

tener en cuenta que, a pesar de que el Servidor no puede enviar *Indications* adicionales hasta recibir confirmación del Cliente, esto no afecta a las solicitudes pendientes por parte del Cliente, las cuales tiene que atender el Servidor.

2.6.5. GENERIC ACCESS PROFILE (GAP)

En esencia, GAP es la capa de control de mayor nivel de abstracción de BLE. Este perfil debe estar presente en todos los dispositivos BLE. Es la piedra angular que permite a los dispositivos BLE interactuar entre sí.

Proporciona un marco que cualquier implementación de BLE debe seguir para permitir que los dispositivos se descubran entre sí, transmitan datos, establezcan conexiones seguras y realicen otras muchas operaciones fundamentales dentro de un estándar, universalmente consensuado. Para entender el perfil GAP, inicialmente es necesario definir los siguientes aspectos sobre la interacción entre dispositivos:

- **Roles**: cada dispositivo puede operar en uno o más roles al mismo tiempo. Cada rol impone restricciones e incluye ciertos requisitos de actuación. Determinadas combinaciones de roles permiten que los dispositivos se comuniquen entre sí, y el perfil GAP se encarga de establecer las interacciones entre estos roles.

- **Modos**: un modo es un estado al que el dispositivo puede cambiar durante un cierto período de tiempo para alcanzar un objetivo concreto, o más específicamente, para permitir a otro dispositivo ejecutar un procedimiento concreto. La conmutación entre modos puede activarse a través de la interfaz de usuario, o automáticamente cuando sea necesario.

- **Procedimientos**: un procedimiento es una secuencia de acciones que permite a un dispositivo lograr un determinado objetivo. Los procedimientos suelen estar asociados a un modo concreto.

2.6.5.1 Roles del perfil GAP

El perfil GAP especifica cuatro roles posibles, que los dispositivos pueden adoptar para formar parte de una red BLE. Cada dispositivo puede implementar diferentes roles al mismo tiempo, los cuales son:

1) **Broadcaster**: optimizado para aplicaciones que transmiten datos de forma regular. El rol *Broadcaster* consiste en enviar periódicamente datos a través de los paquetes de *advertising*. Teóricamente, el rol *Broadcaster* puede ser usado con radios que solo permitan la transmisión de datos, no la recepción, pero en la práctica, este rol suele asignarse a un dispositivo que permita ambas.

2) **Observer**: optimizado para aplicaciones de solo recepción que deseen recopilar datos de los dispositivos de radiodifusión. El observador escucha en busca de datos en los paquetes de *advertising* enviados por los dispositivos *Broadcaster*. Un ejemplo de este tipo de dispositivos sería una aplicación en un móvil que muestra la temperatura recogida por un sensor de temperatura en *broadcasting*.

3) **Central**: el rol de dispositivo *Central* corresponde a la capa de enlace del dispositivo Maestro. En un dispositivo capaz de establecer múltiples conexiones, es el encargado de iniciar las conexiones. En esencia, su función es permitir a los dispositivos entrar en la red. Este rol suele desempeñarlo un *smartphone, tablet*, o cualquier otro dispositivo con cierta potencia, ya que se necesita cumplir ciertos requisitos de memoria y CPU. El dispositivo *Central* comienza por escuchar los paquetes de *advertising* de otros dispositivos, para luego iniciar la conexión con un dispositivo seleccionado.

4) **Peripheral**: el rol de dispositivo *Peripheral* corresponde a la capa de enlace del dispositivo Esclavo. Este rol emplea paquetes de *advertising* para permitir que los dispositivos de tipo *Central* lo encuentren y, posteriormente, establezcan una conexión con él. El protocolo BLE está optimizado para que la implementación de dispositivos *Peripheral* no requiera de muchos recursos de procesamiento y memoria.

2.6.5.2. Modos y procedimientos del perfil GAP

En BLE, los paquetes de *advertising* se envían a ciegas, de manera unidireccional y a intervalos fijos, constituyendo esto la base, tanto del proceso de *broadcasting* y *observing*, como del proceso de *discovery*. En ellos, un dispositivo escaneando en busca de paquetes de *advertising* puede recibirlo o no, dependiendo de si justo en ese momento se envía un paquete, y una vez recibido, no tiene por qué establecer una conexión, sino únicamente limitarse a recibir los paquetes.

Las conexiones, por otro lado, requieren un par que realice intercambios de datos de forma síncrona a intervalos regulares y proporcione garantías en cuanto a transmisión y rendimiento de datos. En la Tabla 3 se recogen los diferentes modos y los procedimientos aplicables a estos, dentro del perfil GAP.

Modo	Rol(es) aplicable(s)	Procedimiento(s) aplicable(s) al par
Broadcast	Broadcaster	Observation
Non-discoverable	Peripheral	N/A
Limited discoverable	Peripheral	Limited and General discovery
General discoverable	Peripheral	General Discovery
Non-connectable	Peripheral, broadcaster, observer	N/A
Any connectable	Peripheral	Cualquier establecimiento de conexión

TABLA 3: MODOS Y SUS RESPECTIVOS PROCEDIMIENTOS APLICABLES

Así, en la Tabla 4 se encuentran los modos en los que el otro par tiene que estar para realizar cada uno de los procedimientos GAP.

Procedimiento	Rol(es)	Modo(s) aplicable(s) al par
Observation	Observer	Broadcast
Limited Discovery	Central	Limited discoverable
General Discovery	Central	Limited and General discoverable
Name Discovery	Peripheral Central	N/A
Cualquier establecimiento de conexión	Central	Cualquiera conectable
Actualización parámetro de conexión	Peripheral, Central	N/A
Terminar conexión	Peripheral, Central	N/A

TABLA 4: PROCEDIMIENTOS Y SUS RESPECTIVOS MODOS APLICABLES

2.6.5.3. Broadcast y Observación

El modo *Broadcast* y el procedimiento de observación definidos en GAP establecen las bases del envío de datos de forma unidireccional entre dispositivos, como pasa con un *Broadcaster* transmitiendo a los *Observer*. Dado que se va a trabajar con dispositivos *Central* y dispositivos *Peripheral*, no se va a hacer uso de estos modos.

2.6.5.4. Descubrimiento ente dispositivos Central y dispositivos Peripheral

La capacidad de descubrimiento de un dispositivo hace referencia a cómo un dispositivo *Peripheral* anuncia su presencia a otros dispositivos, y qué pueden o deben hacer estos con la información. Las diferencias entre los diferentes modos y procedimientos de descubrimiento conciernen a si realmente se están realizando los procesos de *advertising* y *scanning*, pero también tienen en cuenta la naturaleza de los datos incluidos en los paquetes de *advertising*. Más concretamente, un campo opcional definido por el SIG dentro de los datos de *advertising*, denominado *Flag AD*, determina el modo de descubrimiento del dispositivo.

Usado solo por dispositivos *Peripheral*, estos modos permiten a los dispositivos *Central* descubrir dispositivos *Peripheral* dentro de su rango de escucha. El descubrimiento comúnmente se refiere a detectar la presencia y la información básica de otro dispositivo cercano. Eso no implica necesariamente la intención de establecer una conexión o intercambiar datos, aunque sea el caso natural. En algunos casos, y especialmente con dispositivos *Central* equipados con *displays* para el usuario, el descubrimiento se usa simplemente para completar una lista con los dispositivos disponibles.

Modos de descubrimiento

A continuación se muestran los diferentes modos de descubrimiento que pueden emplear los diseñadores de periféricos, según las especificaciones del diseño.

I. Non-discoverable

No ser detectable significa que otros dispositivos no pueden conocer la presencia del dispositivo *Peripheral* ni realizar ninguna consulta sobre su naturaleza. Este modo se usa normalmente cuando un dispositivo no quiere ser detectado por los dispositivos *Central*.

II. Limited discoverable mode

Este modo permite que un dispositivo sea detectable durante un período de tiempo limitado y con una prioridad reducida. En este modo, un dispositivo envía paquetes de *advertising* con el *flag* de *Limited Discoverable* en el campo *Flags AD.* La popularidad de este modo ha ido disminuyendo con el tiempo, siendo la tendencia actual utilizar el modo de descubrimiendo general, aplicando ciertos filtros en todo caso.

III. General discoverable mode

Este modo hace que un dispositivo sea detectable durante el tiempo que sea necesario, o se considere oportuno. Un dispositivo que pasa a este modo expresa su deseo de ser descubierto por pares *Central*, generalmente con la intención de establecer una conexión. Solo los dispositivos *Central* que realizan el procedimiento de descubrimiento general encontrarán dispositivos *Peripheral* en este modo.

Procedimientos de descubrimiento

La especificación define dos procedimientos de descubrimiento:

I. Limited Discovery Procedure

Un dispositivo *Central* realizando este procedimiento inicia el escaneo activo y analiza cada paquete de *advertising* que recibe. Si en estos paquetes está activo el *flag* de descubrimiento limitado, el dispositivo par se reporta a la aplicación.

II. General Discovery Procedure

Un dispositivo *Central* realizando este procedimiento inicia el escaneo activo y analiza cada paquete de *advertising* que recibe. Si se detectan los *flags* de descubrimiento limitado o descubrimiento general, el dispositivo par se reporta a la aplicación.

2.6.5.5. Establecimiento de conexión entre dispositivos Central y dispositivos Peripheral

Para que un dispositivo *Central* pueda establecer una conexión con un dispositivo *Peripheral*, este último debe estar en modo *Connectable.* Al igual que pasa con el proceso de descubrimiento, la selección de dispositivos con los que interactuar están controlados por una serie de modos y procedimientos.

Modos de establecimiento de conexión

La diferencia entre los siguientes modos de establecimiento de conexión depende del tipo de paquete de *advertising* que use el dispositivo *Peripheral*.

I. Non-connectable mode

Un dispositivo en este modo no envía paquetes de *advertising*, o envía paquetes de *advertising* del tipo ADV_NONCONN_IND o ADV_SCAN_IND. En ambos casos, el dispositivo es, como indica el nombre del modo, no conectable, lo que significa que ningún dispositivo *Central* puede establecer una conexión con él.

II. Directed conectable mode

Un dispositivo en este modo envía paquetes de *advertising* ADV_DIRECT_IND. Al realizar *advertising* dirigido, un dispositivo envía paquetes de *advertising* a alta frecuencia y por un corto tiempo, sin carga de datos de usuario y con una dirección de *Bluetooth Central* de destino. Se trata de un modo de reconexión rápida, que normalmente se utiliza cuando el dispositivo *Peripheral* sospecha que el dispositivo *Central* está intentando establecer la conexión.

III. Undirected conectable mode

Un dispositivo en este modo envía paquetes de *advertising* ADV_IND. Este es el modo de conexión estándar, a través del cual un dispositivo *Peripheral* se puede mantener en un estado conectable por un período de tiempo más largo y puede intentar conectarse con un dispositivo *Central* nuevo o con uno que ya conoce.

Procedimientos de establecimiento de conexión

Dado que un dispositivo *Central* no tiene medios para seleccionar los paquetes de *advertising* que va a recibir durante el escaneo, con intención de conectarse, las diferencias entre los procedimientos de establecimiento de conexión no dependen de los tipos de paquetes de *advertising*. En cambio, el tipo de procedimiento vendrá determinado por el tipo de filtrado que aplica el dispositivo *Central* a los paquetes entrantes.

I. Auto connection establishment procedure

Con este procedimiento de paso único, el *host* rellena una lista blanca con una serie de dispositivos *Peripheral* conocidos y luego le indica al controlador que se conecte al primero que detectó. En términos generales, este procedimiento es útil cuando el dispositivo *Central* ya conoce un conjunto limitado de dispositivos y no tiene preferencias a la hora de conectarse a ellos.

II. General connection establishment procedure

Este procedimiento de dos pasos se usa comúnmente para conectarse a un dispositivo *Peripheral* desconocido. El dispositivo *Central* comienza el escaneo sin una lista blanca, aceptando todos los paquetes de *advertising* entrantes. Para cada dispositivo *Peripheral* detectado, la aplicación debe decidir si conectarse o continuar con el siguiente. Una vez que se elige un dispositivo *Peripheral*, el dispositivo *Central* se conecta a él mediante el procedimiento de establecimiento de conexión directo.

III. Selective connection establishment procedure

Este procedimiento es idéntico al anterior, con la excepción de que el *host* utiliza una lista blanca de dispositivos conocidos previamente para filtrar los paquetes de *advertising* entrantes.

IV. Direct connection establishment procedure

Este procedimiento estándar de establecimiento de conexión de un solo paso conecta un dispositivo *Central* a un dispositivo *Peripheral* concreto. El *host* utiliza la capa de enlace para iniciar una conexión a un solo dispositivo, identificado por su dirección de *Bluetooth*, sin conocimiento previo de su presencia. El procedimiento puede fallar si el dispositivo *Peripheral* de destino no está disponible o no está en un modo conectable.

CAPÍTULO 3: INTRODUCCIÓN A LORA

En este capítulo se presenta una introducción a la tecnología de comunicación *LoRa*, explicando en mayor profundidad los conceptos más relevantes relacionados con el presente Trabajo Fin de Grado.

3.1. Introducción al estándar de comunicación LoRa

Inicialmente desarrollada por *Cycleo* en 2010, empresa adquirida por *Semtech* en 2012, *LoRa* es una tecnología de comunicación inalámbrica de largo alcance, baja potencia y reducida tasa de bits, diseñada como una solución para infraestructuras de IoT: dispositivos finales empleando *LoRa* para interconectar de forma inalámbrica diferentes redes, haciendo de pasarela entre estos dispositivos finales y los correspondientes servidores. En este capítulo se presenta una descripción general de *LoRa* y un análisis en profundidad de sus componentes funcionales [15].

En la actualidad, no existe una tecnología capaz de cumplir con los requisitos de todas las aplicaciones desarrolladas para IoT. WiFi y BLE son estándares ampliamente adoptados e implementados en soluciones relacionadas con la comunicación de dispositivos personales. La tecnología celular es ideal para aplicaciones que requieren un alto rendimiento de datos y que gozan de su propia fuente de alimentación. LPWAN ofrece una duración de batería de varios años y está diseñado para sensores y aplicaciones que necesiten enviar pequeñas cantidades de datos a largas distancias en cualquier tipo de entorno. Es así como surge *LoRa*, una técnica de modulación de espectro ensanchado derivada de la modulación *Chirp Spread Spectrum* (CSS). En la Figura 9 se recoge una comparativa entre las tecnologías descritas.

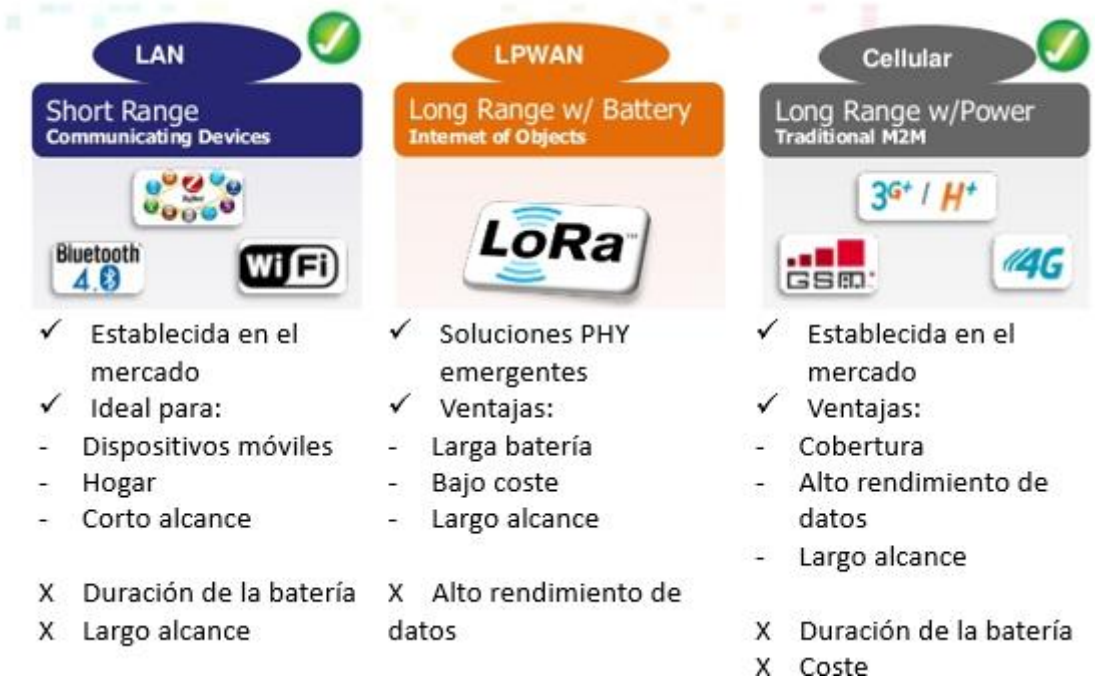

FIGURA 9: COMPARATIVA TECNOLOGÍAS DE COMUNICACIÓN

Se puede definir el estándar *LoRa* como la capa física o modulación inalámbrica utilizada para actuar de enlace en comunicaciones de largo alcance. Mientras que muchos sistemas inalámbricos emplean la modulación *Frequency Shifting Keying* (FSK) como capa física dada su eficiencia a la hora de lograr un bajo consumo de potencia, *LoRa*, como ya se ha mencionado, se basa en la modulación CSS, que mantiene las mismas características de baja potencia de la modulación FSK, pero aumenta significativamente el rango de comunicación [16]. La modulación CSS se ha empleado durante décadas en comunicaciones militares y espaciales, gracias a las largas distancias de comunicación que ofrece y a su resistencia frente a interferencias, sin embargo, *LoRa* es la primera implementación de bajo coste orientada a usos comerciales.

La principal ventaja de *LoRa* radica en su capacidad de ofrecer comunicaciones de largo alcance. Una única pasarela o estación base puede cubrir ciudades enteras o cientos de kilómetros cuadrados. El rango depende en gran medida del entorno o de los obstáculos presentes en una ubicación determinada, pero *LoRa* ofrece una cobertura mucho mayor que cualquier otro estándar de comunicaciones.

3.2. Características de LoRa

Como ya se ha mencionado, *LoRa* es una modulación CSS, que usa "chirridos" de frecuencia con una variación lineal de la frecuencia a lo largo del tiempo para codificar la información [17]. Este "chirrido" con el que CSS codifica los datos es esencialmente una señal sinusoidal de frecuencia modulada en banda ancha que aumenta o disminuye con el tiempo, como se observa en la Figura 10. Debido a la linealidad de los pulsos de "chirrido", las desviaciones de frecuencia entre el receptor y el transmisor son equivalentes a *offsets* en el *timing*, los cuales se pueden eliminar fácilmente en el decodificador. Esto también hace a la modulación inmune al efecto *Doppler*, equivalente a un *offset* en frecuencia.

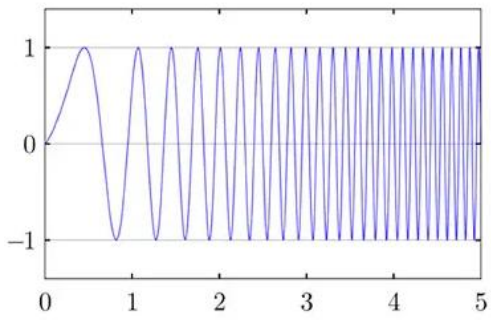

FIGURA 10: REPRESENTACIÓN GRÁFICA DE UN CHIRRIDO

El *offset* de frecuencia entre el transmisor y el receptor puede alcanzar hasta un 20% del ancho de banda sin afectar al rendimiento de decodificación. Esto ayuda a reducir el precio de los transmisores *LoRa*, ya que los cristales de los transmisores no necesitan una precisión extrema. Los receptores *LoRa* pueden bloquear los "chirridos" de frecuencia recibidos, ofreciendo una sensibilidad del orden de -130 dBm. En otras palabras, *LoRa* mejora significativamente la sensibilidad del receptor.

Además, presenta correcciones *Forward Error-correction Codes* (FECs), y su velocidad de transmisión oscila entre los 0.3 Kbps y los 50 Kbps. En cuanto a su rango de alcance, LoRa tiene un rango de más de 15 kilómetros. Todas estas características convierten a la tecnología de comunicación *LoRa* en una solución ideal para comunicaciones *Peer-To-Peer* (P2P) entre nodos.

La tecnología de comunicación *LoRa* puede trabajar en varios rangos de frecuencia dependiendo de la región en la que se encuentre. En Europa, la banda utilizada para *LoRa* es la ISM 863-870 MHz, regulada por la *European Telecommunications Standards Institute* (ETSI). Para ello, utiliza diez canales elegidos arbitrariamente según la UN-111, recogida en el Boletín Oficial del Estado (BOE)-A-2013-4845 español, donde se especifica un ancho de banda de 0.3 MHz por canal. En la Figura 11 se recogen los canales para LoRa asignados en Europa [18].

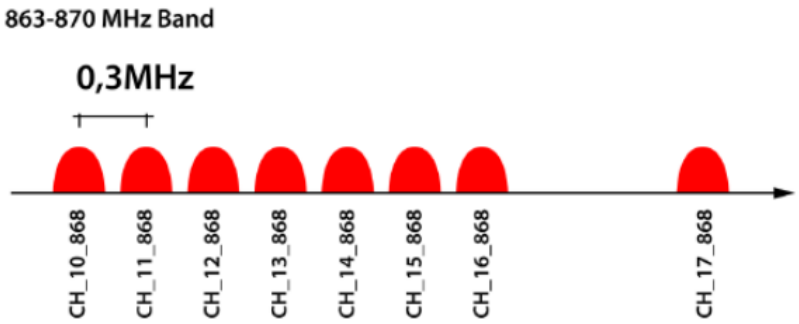

FIGURA 11: CANALES LORA EN EUROPA

La Tabla 5 recoge todas las características previamente descritas, así como los parámetros correspondientes a América del Norte.

	Europa	América del Norte
Banda de frecuencia	867-869 MHz	902-928 MHz
Canales	10	64 + 8 + 8
Canal banda ancha ascendente	125/250 kHz	125/500 kHz
Canal banda ancha descendente	125 kHz	500 kHz
TX encendido	+14 dBm	+20 dBm típ (+30 dBm permitidos)
TX desconectar	+14 dBm	+27 dBm
SF Up	7-12	7-10
Velocidad de datos	250 bps - 50 kbps	980 bps - 21.9 kbps
Link Budget Up	155 dB	154 dB
Link Budget Dn	155 dB	157 dB

TABLA 5: CARACTERÍSTICAS MODULACIÓN LORA

3.3. LoRaWAN

LoRaWAN hace referencia a la especificación que define la capa *Medium Access Control* (MAC) para redes LPWAN [19]. Se implementa en la parte superior de la capa física *LoRa* y define el protocolo de comunicaciones y la arquitectura de una red basada en el estándar *LoRa*. La topología de este tipo de redes suele ser en estrella, como se ve en la Figura 12, donde cada nodo final se comunica con varias puertas de enlace que a su vez se comunican con un servidor principal. En el presente TFG no se trabaja con redes *LoRaWAN*, únicamente con la capa física *LoRa*, de ahí que no se profundice en los conocimientos relativos a este tipo de redes.

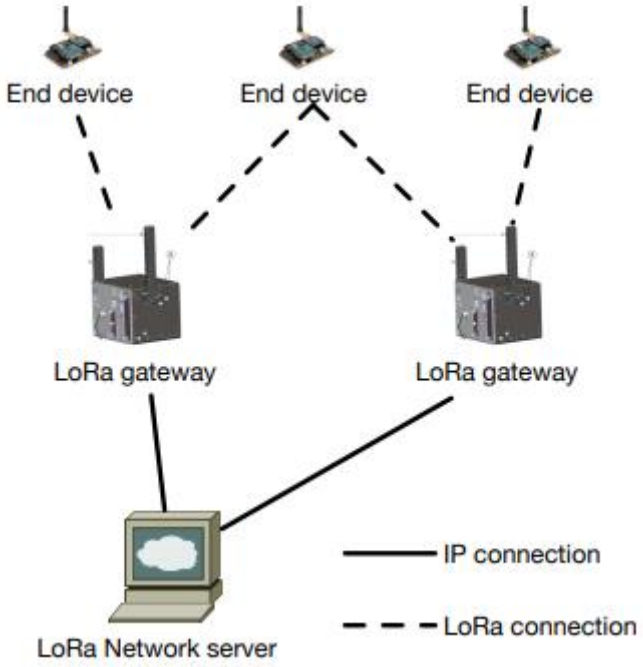

FIGURA 12: TOPOLOGÍA DE UNA RED LORAWAN

CAPÍTULO 4: COMPONENTES UTILIZADOS

En este capítulo se recogen las herramientas *hardware* y *software* empleadas para el desarrollo del presente TFG, haciendo un breve análisis y presentación de cada componente, así como la función asignada para cada solución desarrollada. Para una mejor organización se ha dividido este capítulo en dos apartados bien diferenciados: componentes *hardware* y componentes *software.*

4.1. Componentes hardware

En esta sección se describen todos los componentes *hardware* con los que se ha trabajado en el presente TFG.

4.1.1. ORDENADOR PORTÁTIL

El dispositivo utilizado para el desarrollo del código correspondiente al *firmware* de los dispositivos IoT, y la posterior evaluación de los eventos producidos durante la comunicación entre dispositivos en ambas soluciones desarrolladas, es el portátil *Personal Computer* (PC) *Notebook* Compaq 15-h051ns [20], mostrado en la Figura 13, junto con sus características principales.

Número de Procesadores	4
Tipo de procesador	AMD A6-5200
Velocidad del Procesador	2 mHz
Disco duro	1 TB
Memoria RAM	4 GB
Tarjeta Gráfica	AMD Radeon HD 8400
Conectividad	2 USB 2.0 1 USB 3.0 Bluetooth 4.0 WiFi HDMI

FIGURA 13: PC NOTEBOOK COMPAQ 15-H051NS

Mediante su conectividad USB y WiFi se ha procedido a cargar el *firmware* desarrollado en los diferentes dispositivos IoT empleados, y una vez implementado el código, los puertos USB han servido para alimentar los dispositivos.

4.1.2. SMARTPHONE

Tanto en la plataforma inicial como en la final, un *smartphone* ejecutando la aplicación *nRF Connect* actuará como dispositivo BLE *Central* final, es decir, como dispositivo Cliente en ambas pasarelas BLE-*LoRa*-BLE desarrolladas. Para ello, se ha hecho uso del *smartphone* Redmi Note 4 de la empresa china Xiaomi, siendo el único requisito exigido que el dispositivo tuviera conectividad BLE [21]. En la Figura 14 se muestra una imagen del dispositivo móvil junto con sus características técnicas principales.

FIGURA 14: XIAOMI REDMI NOTE 4

4.1.3. DISPOSITIVO LOPY

El dispositivo *LoPy*, mostrado en la Figura 15, de la empresa *Pycom*, destaca por ser uno de los pocos microcontroladores que trabajan con el lenguaje de programación *Micropython* y se encuentren disponibles en el mercado, capaces de ofrecer conectividad para tres tecnologías de comunicación: WiFi, BLE y *LoRa*, convirtiéndolo en uno de los dispositivos más potentes y versátiles para el desarrollo de plataformas de IoT [22].

FIGURA 15: DISPOSITIVO LOPY

Con un tamaño de 55 milímetros (mm) x 20mm x 3.5mm, el dispositivo *LoPy* está basado en el SoC ESP32 de Espressif, una serie de microcontroladores de bajo coste y consumo de potencia que integran WiFi y *Bluetooth*. Este *System on Chip* (SoC) incluye un microprocesador Xtensa LX6

dual-core, 512 KBytes de memoria RAM y 4 MBytes de memoria *Flash*. Dado que el microprocesador no incluye *LoRa*, el dispositivo *LoPy* integra un módulo adicional conectado al microprocesador por el bus *Serial Peripheral Interface* (SPI), un transceptor SX1272 de *Semtech*, dotando al dispositivo *LoPy* de un módem *LoRa*. La elección de este kit de desarrollo *hardware* para las soluciones definidas en este TFG se basa en la capacidad de integrar las tecnologías BLE y *LoRa* en el mismo dispositivo, haciendo posible su uso en ambas soluciones planteadas. En la Figura 16 se muestra el diagrama de bloques del dispositivo *LoPy* de forma esquemática.

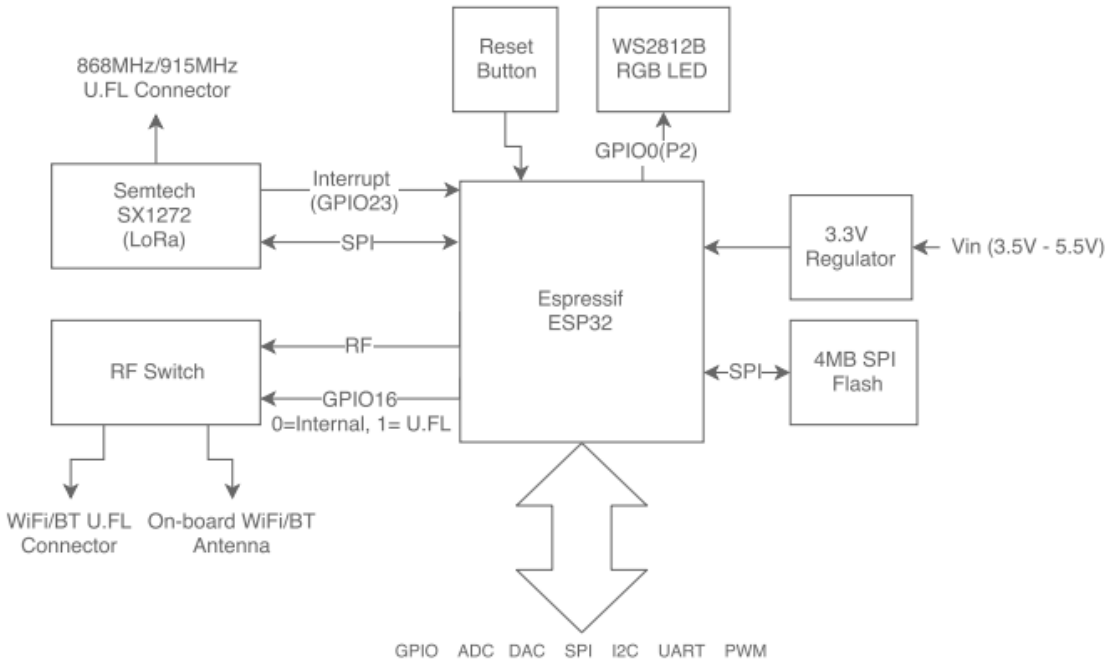

FIGURA 16: DIAGRAMA DE BLOQUES DEL DISPOSITIVO LOPY

El dispositivo acepta el uso de antenas externa, siendo en el caso de *LoRa* necesaria. El dispositivo cuenta con conectores para una antena WiFi y una antena *LoRa*. El uso de las antenas externas proporciona un aumento en las distancias efectivas de cobertura de ambas tecnologías de comunicación, llegan a ser el alcance de la antena WiFi de hasta 1 kilómetro (km) y para *LoRa*, en condiciones perfectas y actuando como nodo dentro de una red, el rango de alcance puede llegar hasta los 40 km. El dispositivo *LoPy* puede configurarse como *nano-gateway* en una red *LoRaWAN*, con un rango de 22 km y capacidad de conexión de hasta 100 nodos. Además, proporciona soporte a una variedad de métodos de seguridad y encriptación como *Transport Layer Security* (TLS) y *Secure Sockets Layer* (SSL). Para la alimentación, puede tomar valores entre 3.3 y 5.5 voltios de entrada, aunque en el caso particular de este TFG, al hacer uso de la tarjeta de expansión, se puede conectar fácilmente el dispositivo *LoPy* a la alimentación mediante conexión USB. La tarjeta de

expansión también permite conectar una batería externa en caso necesario. El modelo usado es el *Expansion Board* 2.0, compatible con los dispositivos *WiPy* 2.0, *SiPy* y *FiPy*, además de con el dispositivo *LoPy*. Su diagrama de bloques se muestra en la Figura 17.

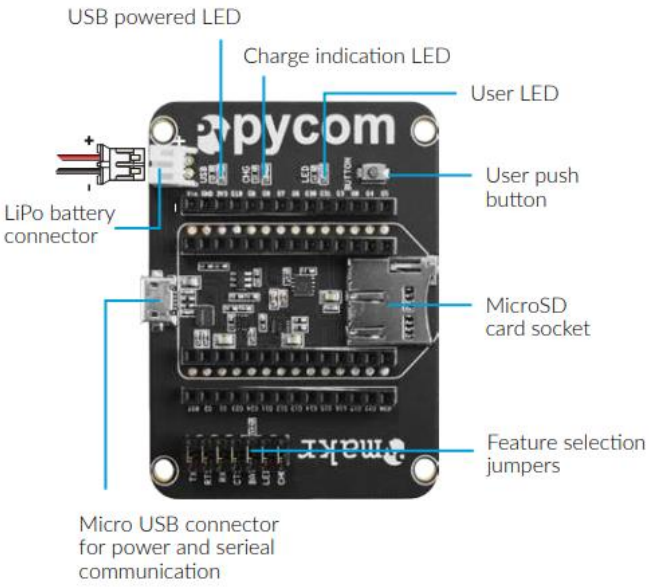

FIGURA 17: DIAGRAMA DE BLOQUES EXPANSION BOARD 2.0 PARA DISPOSITIVO LOPY

El dispositivo *LoPy* cuenta con una *General Purpose Input/Output* (GPIO) amplia, ideal para trabajar con redes de sensores, incluyendo 8 entradas analógicas, lo que elimina la necesidad de emplear *Analog to Digital Converter* (ADC) externos. Dispone de bus *Inter-Integrated Circuit* (I2C) y SPI para interconectar ADC con periféricos externos, como sensores digitales. 8 de los GPIO también se pueden configurar como salidas ADC y 18 pines GPIO son compatibles con *Pulse-Width Modulation* (PWM). En la Figura 18 se muestra el *pinout* del dispositivo *LoPy*.

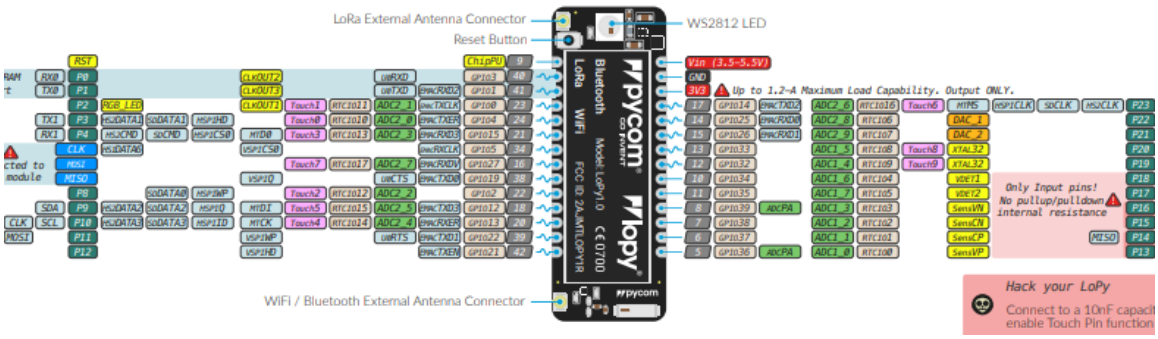

FIGURA 18: PINOUT DISPOSITIVO LOPY

Para la programación del dispositivo *LoPy* se ha hecho uso del IDE *Atom* que, junto con el *plugin* propio de la empresa *Pycom*, *Pymakr*, permite gestionar los dispositivos propios de esta compañía. El uso del lenguaje de programación *Micropython*, facilita la creación rápida de código y permite al usuario beneficiarse de multitud de bibliotecas relacionadas con sensores y otros periféricos, convirtiéndolo en el lenguaje ideal para desarrollar soluciones de IoT.

4.1.4. DISPOSITIVO REDBEAR DUO

El dispositivo DUO de la empresa *RedBear* es una placa de desarrollo de tamaño reducido diseñada para simplificar el proceso de creación de productos de IoT [23]. Está construido con un potente microcontrolador ARM Cortex-M3, un módulo inalámbrico que equipa conectividad WiFi y BLE, y abundantes periféricos. La elección del kit de desarrollo hardware *RedBear DUO* para el desarrollo de este TFG se basa principalmente en la versatilidad que ofrece gracias a su fácil programación, la utilización de código abierto, y la integración de conectividad WiFi y BLE, siendo esta última necesaria para satisfacer los objetivos propuestos en este TFG. Además, su reducido tamaño y bajo coste lo convierten en una solución ideal para proyectos de IoT. En la Figura 19 se muestra el dispositivo *RedBear DUO*.

FIGURA 19: DISPOSITIVO REDBEAR DUO

El microcontrolador STM32F205 ARM Cortex-M3, fabricado por *STMicroelectronics*, que incorpora el dispositivo DUO, usa la tecnología de proceso *Non-Volatile Memory* (NVM) de 90 nanómetros con un acelerador de memoria adaptable en tiempo real (*ART Accelerator*) y una matriz de bus multicapa, ofreciendo una gran relación entre rendimiento y precio. Se caracteriza por presentar un alto grado de integración, incluyendo 1 Mbyte de memoria *Flash* y 128 Kbytes de *Static Random Access Memory* (SRAM*)*.

El dispositivo DUO soporta las tecnologías de comunicación WiFi y BLE, permitiendo a otros dispositivos conectarse localmente al dispositivo DUO mediante *Bluetooth* y este a su vez con la red WiFi para interactuar con la web. Para albergar ambas tecnologías de comunicación, el dispositivo DUO cuenta con un chip Broadcom BCM43438 que combina WiFi 802.11b/g/n y Bluetooth 4.1 en modo dual, compartiendo la misma antena de 2.4 GHz, pudiendo funcionar simultáneamente ambas tecnologías. En el caso particular de este TFG, los dispositivos DUO utilizados establecerán conexión BLE con otros dispositivos finales, para posteriormente comunicarse entre ellos a distancias que no cubren las antenas BLE a través de la tecnología de comunicación *LoRa*.

En la Figura 20 se recogen los componentes principales que conforman el dispositivo DUO. Así, además de los chips previamente mencionados, el dispositivo DUO cuenta con varios LED integrados, 18 pines de entrada y salida de propósito general que constituyen la GPIO, un conector para el uso de una antena externa, y diversas interfaces de comunicación: SPI, UART, I2S, I2C, CAN y USB, entre otras. El sistema operativo *FreeRTOS* permite al dispositivo DUO gestionar operaciones en tiempo real. Para completar la interfaz de usuario se cuenta con los botones de *Setup* y *Reset*, necesarios para la configuración de las credenciales WiFi así como para reiniciar el dispositivo o reestablecer los valores de fábrica.

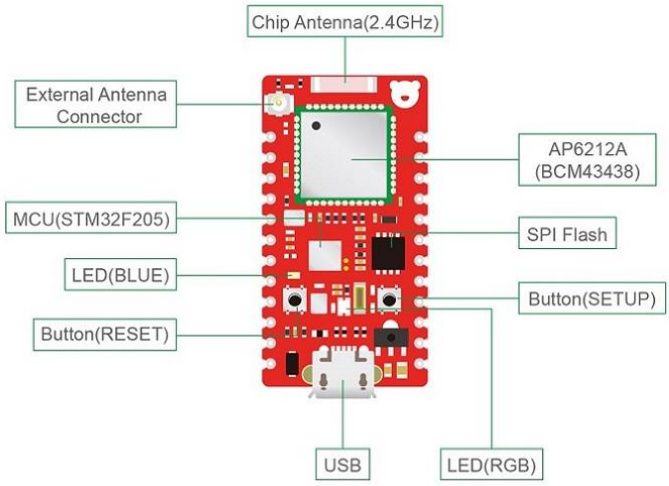

FIGURA 20: DIAGRAMA DE BLOQUES DEL DISPOSITIVO REDBEAR DUO

Para alimentar el dispositivo DUO generalmente se hace uso del puerto micro-USB, empleado tanto para alimentación como para la programación del dispositivo, y para comunicaciones seriales con el ordenador. La tensión de entrada del puerto USB estará limitada a 3.3 V, dado que todos los circuitos del dispositivo DUO trabajan con este valor. El pin *VIN* también suministra un voltaje parecido al ofrecido por el puerto USB, y el pin *3V3*, como su nombre indica,

ofrece un suministro de 3.3 V. Finalmente, el pin *VBAT* permite conectar al dispositivo DUO una batería de reserva para alimentarlo mientras se encuentre en modo *deep sleep*, conservando el estado de la memoria para reanudarlo cuando se necesite. El *pinout* del dispositivo DUO se muestra en la Figura 21. En este TFG únicamente se hará uso de los pines *PA9* y *PA10*, es decir UART1_RX y UART1_TX, para las comunicaciones seriales necesarias en la solución final.

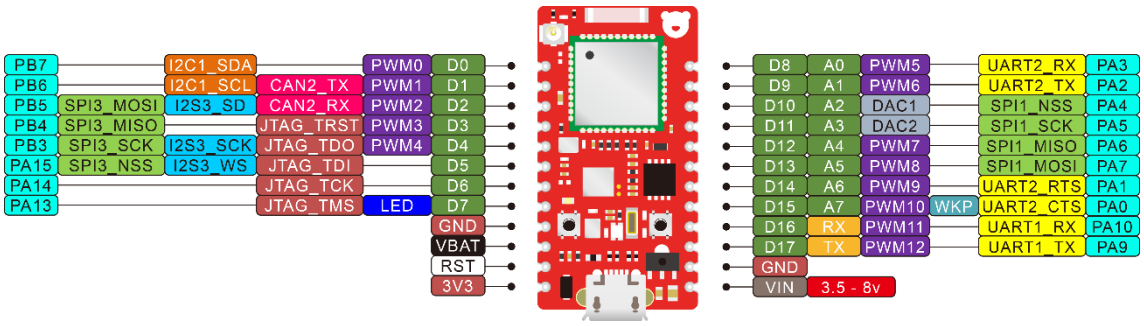

FIGURA 21: PINOUT DEL DISPOSITIVO REDBEAR DUO

El dispositivo DUO es compatible con varios lenguajes de programación, como son Arduino, C/C++, *JavaScript* y *Python*. Para el desarrollo de aplicaciones para DUO se pueden emplear los entornos de desarrollo *Arduino IDE*, *Particle Web IDE*, *Espruino Web IDE*, *Broadcom WICED SDK* y *GCC*. En cuanto a aplicaciones, la potencia del dispositivo DUO le permite trabajar en campos tan diversos como la automatización industrial o de viviendas, así como en aplicaciones más centradas en el usuario final como pueden ser aplicaciones para casas inteligentes, redes de sensores IoT o pasarelas WiFi/BLE.

4.1.5 Dispositivo Bluz DK

El dispositivo *Bluz DK* [24], mostrado en la Figura 22, será empleado en el presente TFG como dispositivo *Peripheral* final en ambas soluciones desarrolladas, primero estableciendo conexión con un dispositivo *LoPy* actuando como dispositivo BLE *Central*, y posteriormente con un dispositivo *RedBear DUO*. La elección del kit de desarrollo *hardware* de IoT *Bluz DK*, de la empresa *Bluz*, para el desarrollo de este TFG se basa principalmente en su bajo coste, fácil programación, utilización de código abierto y la integración de un módulo BLE, tecnología de comunicación necesaria para satisfacer los objetivos propuestos en este TFG. Se trata de un microcontrolador pensado para aplicaciones de IoT, permitiendo conectarse a la nube, desde donde se puede controlar el dispositivo y gestionar los proyectos desarrollados. Al igual que el dispositivo *RedBear DUO*, el dispositivo *Bluz* se puede utilizar con la plataforma *Particle IDE* para dispositivo IoT.

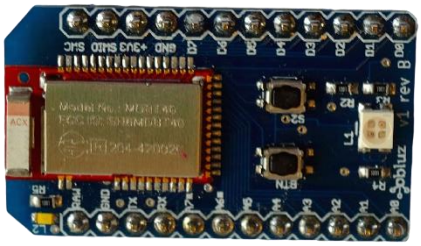

FIGURA 22: DISPOSITIVO BLUZ DK

El dispositivo *Bluz DK* está basado en el SoC nRF51822 de la empresa *Nordic Semiconductor*, dotándolo de conectividad BLE y permitiendo el desarrollo de aplicaciones para comunicaciones inalámbricas de ultra bajo consumo a 2.4 GHz. Así, el dispositivo *Bluz* contiene un procesador ARM Cortex-M0, el más pequeño de los procesadores ARM disponibles, que trabaja a 16 MHz, con 32 KBytes de memoria RAM y 256 KBytes de memoria *Flash*. Su tamaño excepcionalmente reducido, su bajo consumo de energía y el mínimo *footprint* que presenta, permite lograr un rendimiento de 32 bits por el precio de uno de 8 bits. El dispositivo *Bluz* también cuenta con varios LED, 18 pines de entrada/salida de propósito general y multitud de periféricos *hardware* (UART, SPI, I2C, etc.). Los botones de *Reset* y *Setup* permiten reiniciar y configurar el dispositivo. La ubicación de estos elementos se muestra en la Figura 23.

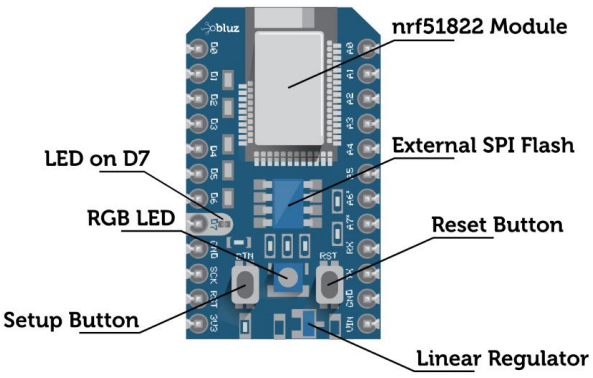

FIGURA 23: DIAGRAMA DE BLOQUES DEL DISPOSITIVO BLUZ DK

En la Figura 24 se recoge el *pinout* del dispositivo, indicando la localización de los pines correspondientes a las entradas y salidas tanto analógicas como digitales, así como los pines de alimentación.

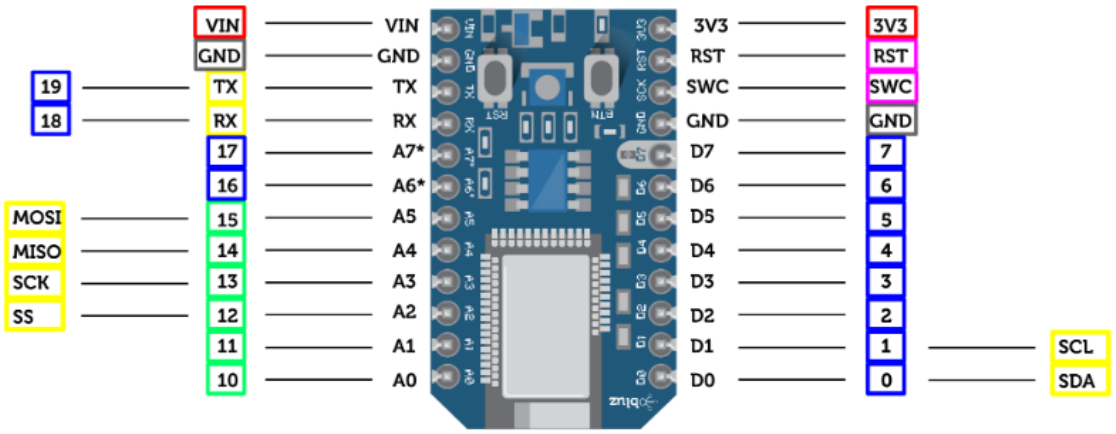

FIGURA 24: PINOUT DEL DISPOSITIVO BLUZ DK

En definitiva, el dispositivo *Bluz DK* es un kit de desarrollo que actúa como un dispositivo Arduino, pero que goza de comunicación BLE integrada. Se trata de una solución pensada para aplicaciones inalámbricas que requieren una batería de larga duración. Con el uso de BLE, el dispositivo BLE puede durar meses o años con una batería de célula, manteniéndose conectado a la nube, convirtiéndolo en un dispositivo completamente accesible. En el presente TFG, al dispositivo *Bluz DK* se le ha añadido el módulo *Battery Shield*, como se ve en la Figura 25, que permite alimentar al dispositivo *Bluz* con una batería CR2032 a través del pin *3V3*. Su uso se basa en un interruptor que enciende o apaga la placa.

FIGURA 25: BATTERY SHIELD V1.0 PARA DISPOSITIVO BLUZ DK

4.1.6. DISPOSITIVO HELTEC WIFI LORA 32

La placa de desarrollo *Heltec WiFi LoRa 32*, fabricada por la empresa china *Heltec Automation*, es un dispositivo que incorpora las tecnologías de comunicación BLE, WiFi y *LoRa*, convirtiéndolo en un dispositivo apto para el desarrollo de una gran variedad de soluciones [25]. El principal inconveniente que presenta es la falta de soporte e información disponible, razón principal por la que, pese a incorporar los requisitos necesarios en cuanto a características técnicas se refiere para el desarrollo de las diferentes soluciones incluidas en este TFG, su uso se ha limitado al de verificación del correcto funcionamiento de la primera solución, quedando excluido de los esquemas finales desarrollados en el presente TFG. En la Figura 26 se muestra una imagen del dispositivo *Heltec*.

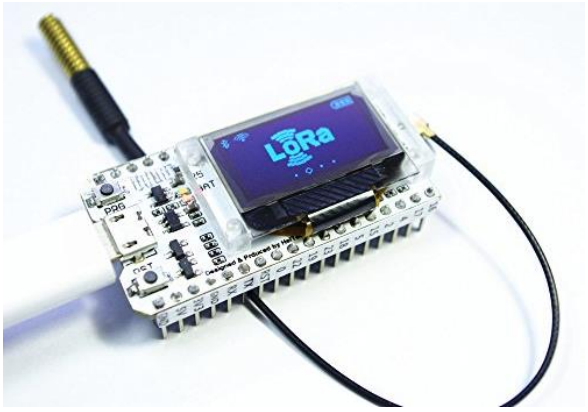

FIGURA 26: DISPOSITIVO HELTEC WIFI LORA 32

Este kit de desarrollo de bajo coste incluye como chip principal un Lexin ESP32, que cuenta con un procesador de doble núcleo Tensilica LX6 con una frecuencia de operación de 240 MHz. Además, este chip incluye una memoria SRAM de 520 KBytes, un transceptor WiFi 802.11b/g/N HT40 y *Bluetooth* integrado en modo dual. Para incluir conectividad *LoRa*, el dispositivo *Heltec* cuenta con un chip *LoRa* SX1278, con una sensibilidad de -148 dBm, +20 dBm de potencia de salida y una distancia de transmisión medida en un área abierta de hasta 2.6 kilómetros. Adicionalmente, cuenta con una memoria *Flash* de 32 Mbyte, una antena *LoRa* (indispensable para hacer uso de la conectividad *LoRa*), una pantalla OLED de 0.96 pulgadas y conexión *microusb* para la alimentación y carga del *firmware*. En la Figura 27 se muestra el *pinout* del dispositivo *Heltec*.

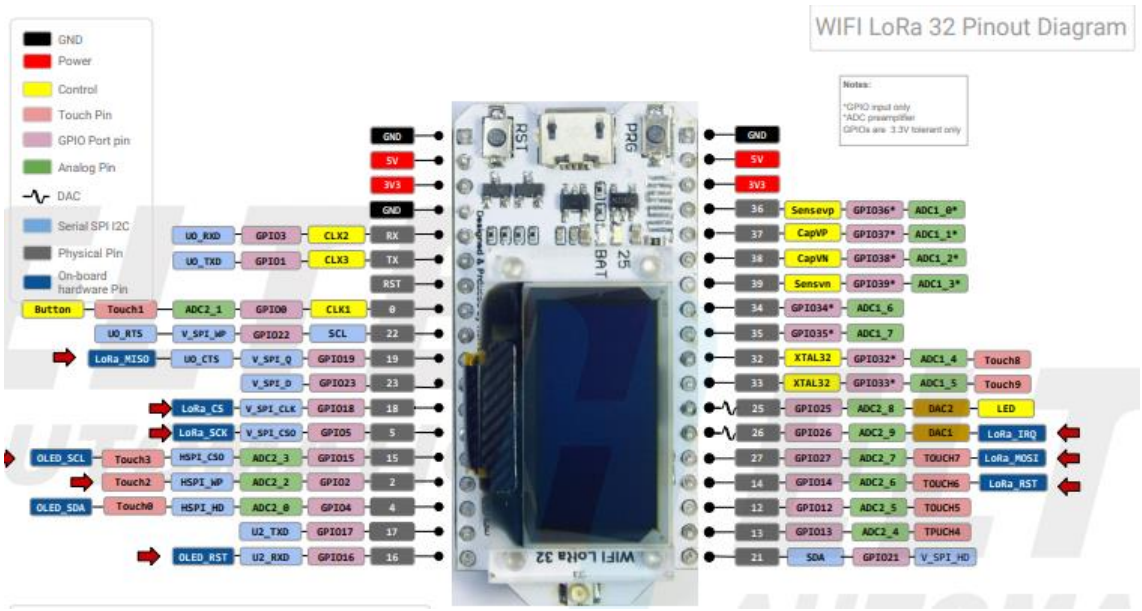

FIGURA 27: PINOUT DISPOSITIVO HELTEC WIFI LORA 32

4.2. Componentes software

En este apartado se describen todos los componentes *software* con los que se ha trabajado en el presente TFG.

4.2.1. Atom

Atom es un editor de código abierto, disponible para *macOS*, *Linux* y *Microsoft Windows*, con soporte para complementos escritos en Node.js y con control Git integrado, lo que permite incluir librerías presentes en *Github* [26]. *Atom* es una aplicación de escritorio creada utilizando tecnologías web, y en este TFG se ha empleado para el desarrollo del código de los dispositivos *LoPy*, así como para establecer comunicación serial con los mismos y analizar el flujo de datos durante las comunicaciones en la pasarela. La mayoría de los paquetes incluidos tienen licencias de *software* gratuitas y están mantenidas por la comunidad. Si bien en la Figura 28 se muestra la interfaz de usuario de este programa, la descripción de su uso se encuentra recogida en el Capítulo 5. Para facilitar el uso de los dispositivos de *Pycom*, la propia compañía desarrolló el *plugin Pymakr*, compatible con el editor de texto *Atom* y *Visual Studio Code.*

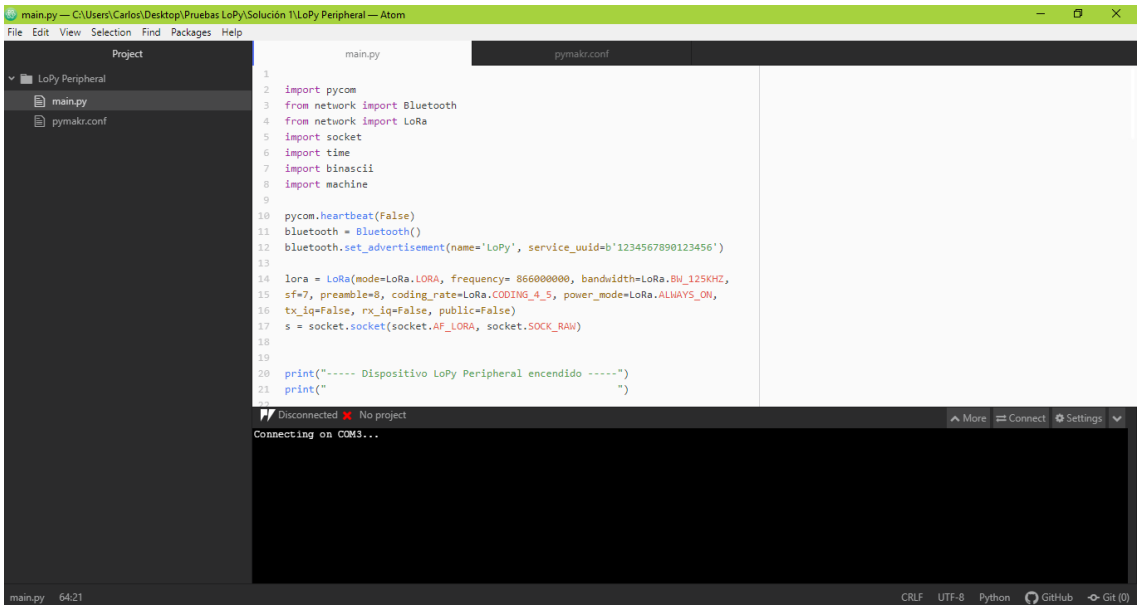

FIGURA 28: INTERFAZ DE USUARIO ATOM IDE

4.2.2. PARTICLE BUILD

Particle Build es un entorno de desarrollo integrado, o IDE, que permite el desarrollo de *software* en una aplicación fácil de usar y ejecutable en un navegador web, lo que lo convierte en un Web IDE [27]. En el presente TFG se ha hecho uso del IDE de *Particle* para el desarrollo del *firmware* de los dispositivos *RedBear DUO*. En la Figura 29 se muestra la interfaz de usuario, mientras que la descripción del manejo de este IDE se recoge en el Capítulo 6, junto con el *firmware* desarrollado.

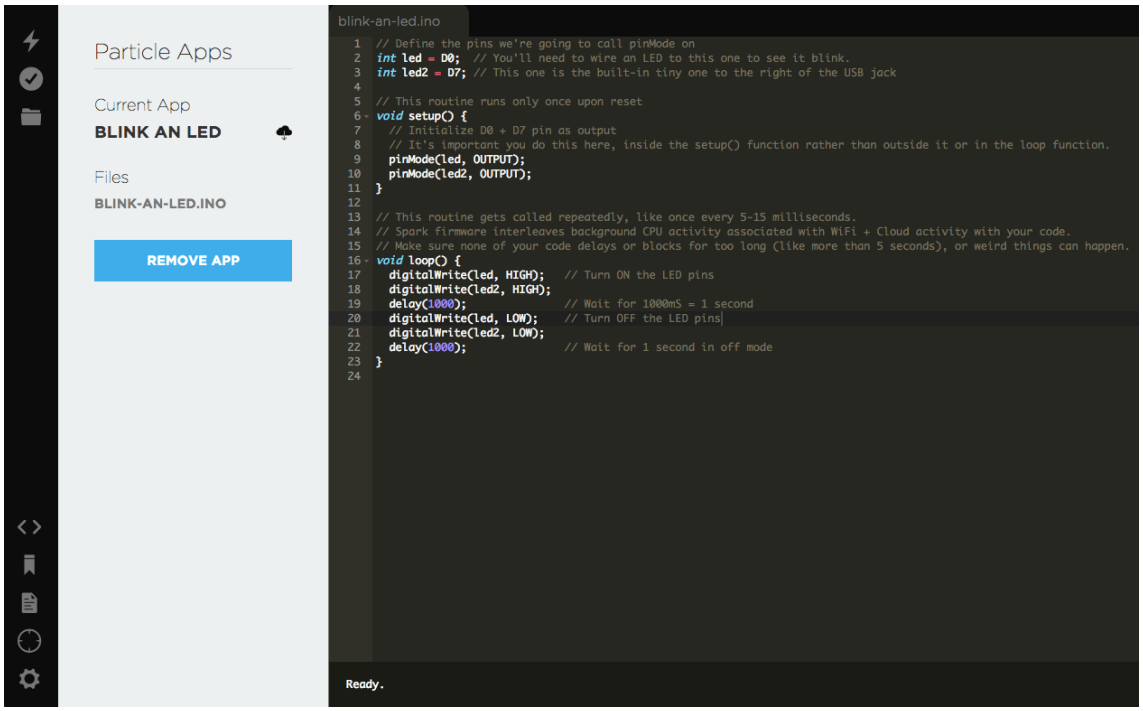

FIGURA 29: INTERFAZ DE USUARIO PARTICLE BUILD IDE

4.2.3. PuTTY

El *software PuTTY* es un cliente SSH y telnet, desarrollado originalmente por Simon Tatham para el sistema operativo Windows. Se trata de un *software* de código abierto, cuyo código fuente está disponible y ha sido desarrollado y respaldado por un grupo de voluntarios [28]. Su función en este TFG ha sido la de establecer comunicación serial con los dispositivos *RedBear DUO* para analizar el flujo de datos a través de dichos dispositivos, con el objetivo de comprobar el correcto funcionamiento de la pasarela. En la Figura 30 se muestra el icono del *software PuTTy*.

FIGURA 30: ICONO DEL SOFTWARE PUTTY

4.2.4. nRF Connect para Android

La aplicación *nRF Connect* para dispositivos móviles es una potente herramienta que permite implementar en un *Smartphone*, un dispositivo BLE *Peripheral* o *Central*, permitiendo realizar los procesos de s*canning* y *advertising* según corresponda, así como establecer conexión con otros dispositivos y gestionar las comunicaciones BLE [29]. En el presente TFG, se ha hecho uso de esta aplicación para actuar en ambas soluciones desarrolladas como dispositivo *Central* final. En la Figura 31 se muestra el icono de la aplicación nRF Connect.

FIGURA 31: ICONO NRF CONNECT

CAPÍTULO 5: DESARROLLO DE LA PASARELA BASADA EN EL DISPOSITIVO LOPY

En este capítulo se describe el desarrollo de la pasarela BLE-*LoRa*-BLE basada en el dispositivo *LoPy*. Inicialmente, para la implementación de la pasarela BLE-*LoRa*-BLE, se propone una solución basada en la plataforma de desarrollo *LoPy*, proporcionando una conexión bidireccional entre dispositivos con conectividad BLE a través de la tecnología de comunicación *LoRa*. La elección del dispositivo *LoPy* se basa principalmente en la versatilidad que ofrece, soportando tanto conectividad BLE como *LoRa*.

5.1. Objetivos de la implementación

El objetivo principal de la primera solución propuesta en el presente TFG es el desarrollo de una pasarela basada en una plataforma que integre ambas tecnologías, BLE y *LoRa*, ejerciendo de intermediario entre los dispositivos finales. En definitiva, se trata de asegurar la funcionalidad básica del proyecto, estableciendo las bases para la solución final, en la cual las especificaciones se adaptan atendiendo al estado actual del mercado y las posibles aplicaciones prácticas de este trabajo.

En la Figura 32 se recoge el esquema de bloques de la plataforma *Hardware/Software* (HW/SW) desarrollada, basada en el dispositivo *LoPy*. Como dispositivo *Peripheral* final y dispositivo *Central* final se ha empleado el dispositivo *Bluz DK* y un *smartphone* ejecutando la aplicación *nRF Connect*, respectivamente, como se describirá en los siguientes apartados. En esta representación gráfica se puede observar cómo el objetivo es dotar a la plataforma de una comunicación completa en ambas direcciones entre dispositivos con conectividad BLE, a través de la tecnología de comunicación *LoRa*.

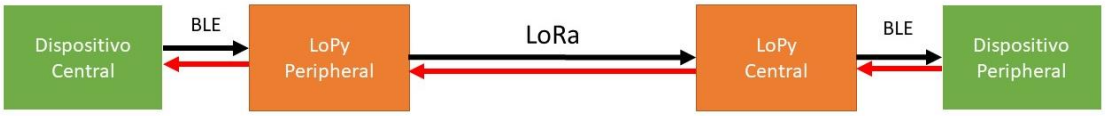

FIGURA 32:PASARELA BASADA EN EL DISPOSITIVO LOPY

Por tanto, se ha desarrollado el *firmware* de forma independiente para cada uno de los extremos de la pasarela. Así, en primer lugar, se ha configurado un dispositivo *LoPy* como dispositivo BLE *Peripheral*, y se le ha dotado de conectividad *LoRa*. En la parte de BLE, cuenta con un servicio primario, en el que a su vez se han definido dos características:

- *chr1*: incluye las propiedades de lectura y escritura. Esta característica permitirá al dispositivo final *Central* escribir los mensajes que desea enviar a través de la pasarela.

- *chr2*: incluye únicamente la propiedad de notificación. El dispositivo final *Central* estará suscrito a esta característica, de manera que a través de ella recibirá los mensajes enviados por el dispositivo final *Peripheral*.

En cuanto a la parte correspondiente a la tecnología de comunicación *LoRa*, ésta se limitará a transmitir por la antena *LoRa* del dispositivo *LoPy* las escrituras realizadas por parte del dispositivo final *Central* en la característica *chr1*. Así, cualquier mensaje recibido por la antena *LoRa* se escribirá en la característica *chr2*, de forma que el dispositivo final *Central* recibirá dicho mensaje en forma de notificación. Dado que la funcionalidad *LoRa* no contempla un establecimiento previo de conexión, en este punto se considera el otro lado de la pasarela como una caja negra. En la Figura 33 queda reflejada esta sección de la solución.

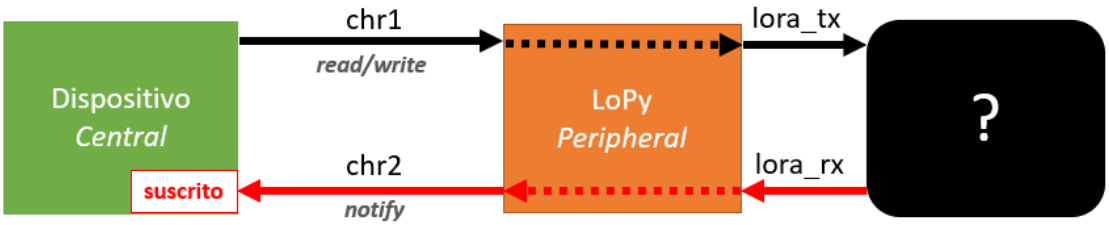

FIGURA 33: LOPY ACTUANDO COMO BLE PERIPHERAL Y CON FUNCIONALIDAD LORA

En segundo lugar, se ha pasado a configurar un segundo dispositivo *LoPy*, esta vez como dispositivo BLE *Central*, dotándolo igualmente de conectividad *LoRa*. En este caso, se hará el proceso inverso al descrito previamente, siendo el dispositivo *LoPy* el encargado de suscribirse a una característica específica del dispositivo final *Peripheral* a la que, para esta explicación, se la denominará *p_char2*. Una segunda característica, *p_char1*, será usada por el dispositivo *LoPy* para escribir las solicitudes enviadas por el dispositivo final *Central*.

- *p_char1*: incluye las propiedades de escritura y lectura. Esta característica permitirá al dispositivo *LoPy Central* escribir en el dispositivo final *Peripheral* los mensajes recibidos por la antena *LoRa*, provenientes a su vez del dispositivo final *Central*, al otro lado de la pasarela.
- *p_char2*: incluye únicamente la propiedad de notificación. El dispositivo *LoPy Central* estará suscrito a esta característica de manera que, al recibir una notificación, esta será reenviada a través de la antena *LoRa*.

En la Figura 34 se representa la otra parte de la pasarela planteada en esta primera solución desarrollada en el presente TFG.

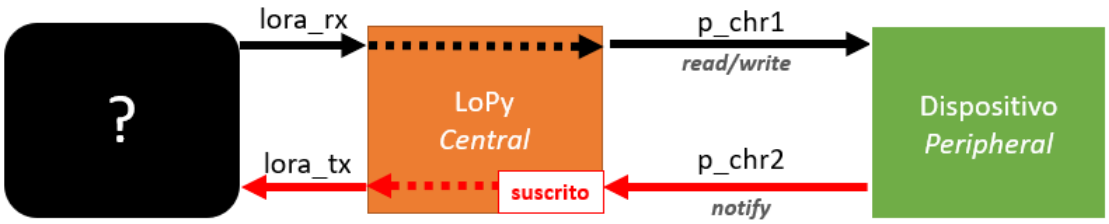

FIGURA 34: LOPY ACTUANDO COMO BLE CENTRAL Y CON FUNCIONALIDAD LORA

Una vez definidas cada una de las partes que conforman la pasarela basada en dispositivos *LoPy*, se pasa a su integración, resultando en el diagrama de bloques final, recogido en la Figura 35. Se sigue considerando una caja negra al espacio entre los dos dispositivos *LoPy* dado que, aun recibiendo los mensajes correspondientes, cada *LoPy* desconoce el remitente y destinatario, limitándose a enviar y recibir información al medio.

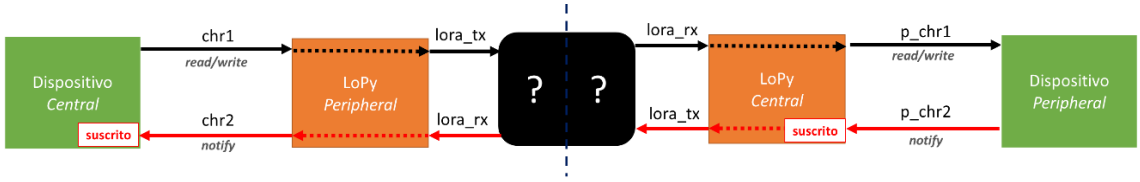

FIGURA 35: PASARELA FINAL BASADA EN EL DISPOSITIVO LOPY

5.2. Diagrama de flujo

El diagrama de flujo asociado con el funcionamiento del *firmware* de la plataforma HW/SW inicial que se ha desarrollado en el presente TFG se ha dividido en dos secciones para su mejor comprensión, representando cada una un extremo de la pasarela.

En la Figura 36 se representa el extremo de la pasarela correspondiente al dispositivo *Central* final, es decir, a la parte del Cliente. Inicialmente, el dispositivo *Central* final iniciará el proceso de *scanning* en busca de paquetes de *advertising* del dispositivo *LoPy Peripheral*. Una vez detectado el dispositivo *LoPy*, el dispositivo *Central* final inicia el proceso de conexión. En caso de realizarse satisfactoriamente, el sistema estará preparado para iniciar la comunicación. En este punto, pueden tener lugar dos acciones: el proceso de escritura en la característica *chr1* del dispositivo *LoPy Peripheral* o el envío de una notificación al dispositivo *Central* final, proveniente de la característica *chr2* del dispositivo *LoPy Peripheral*.

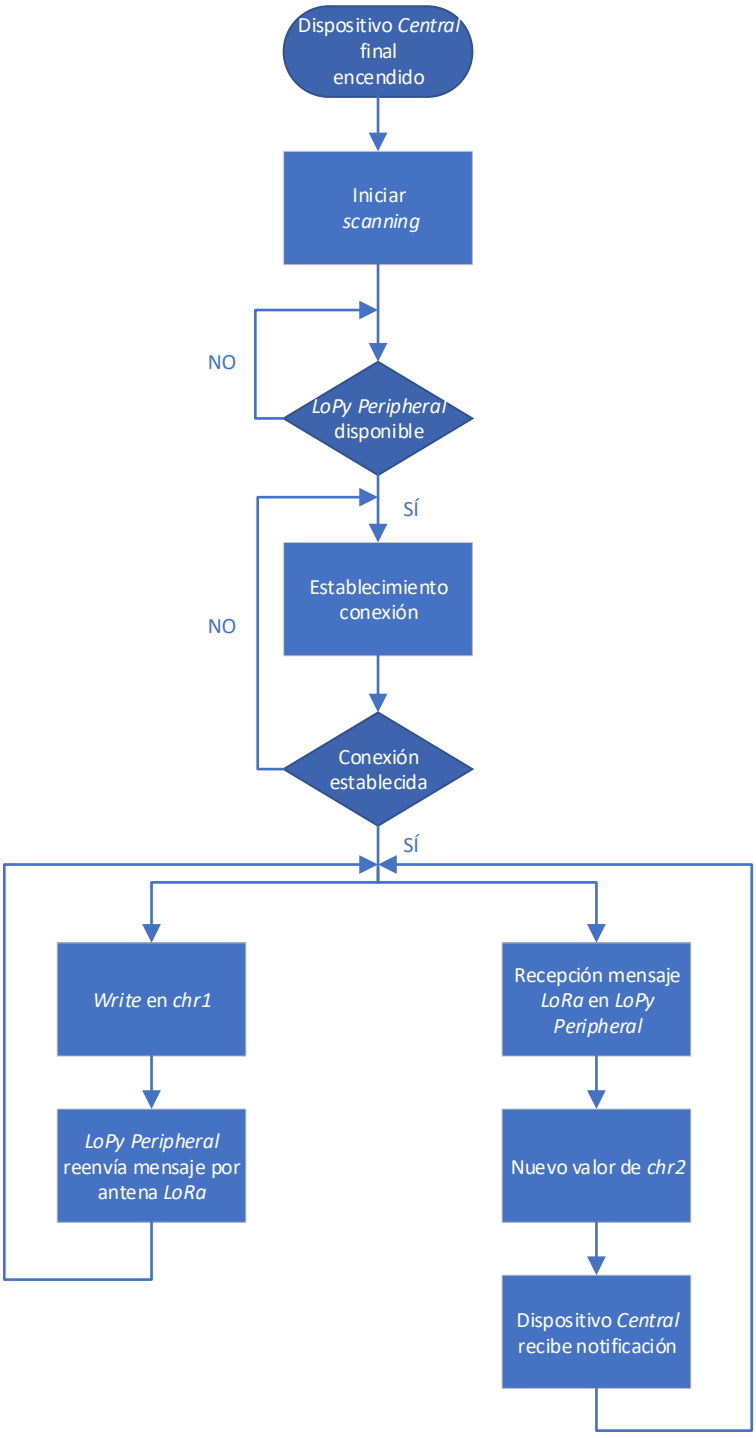

FIGURA 36: DIAGRAMA DE FLUJO DE LA PLATAFORMA HW/SW INICIAL (I)

Así, en la Figura 37 se muestra el extremo de la pasarela correspondiente al Servidor, es decir, al dispositivo *Peripheral* final, conectado mediante comunicación BLE al dispositivo *LoPy*. El dispositivo *Peripheral* final iniciará el proceso de *advertising*, manteniéndose en éste hasta que el

dispositivo *LoPy Central* establezca una conexión. Entonces, pueden tener lugar dos acciones diferentes: escritura en la característica *p_chr1* o notificación en la característica *p_chr2*.

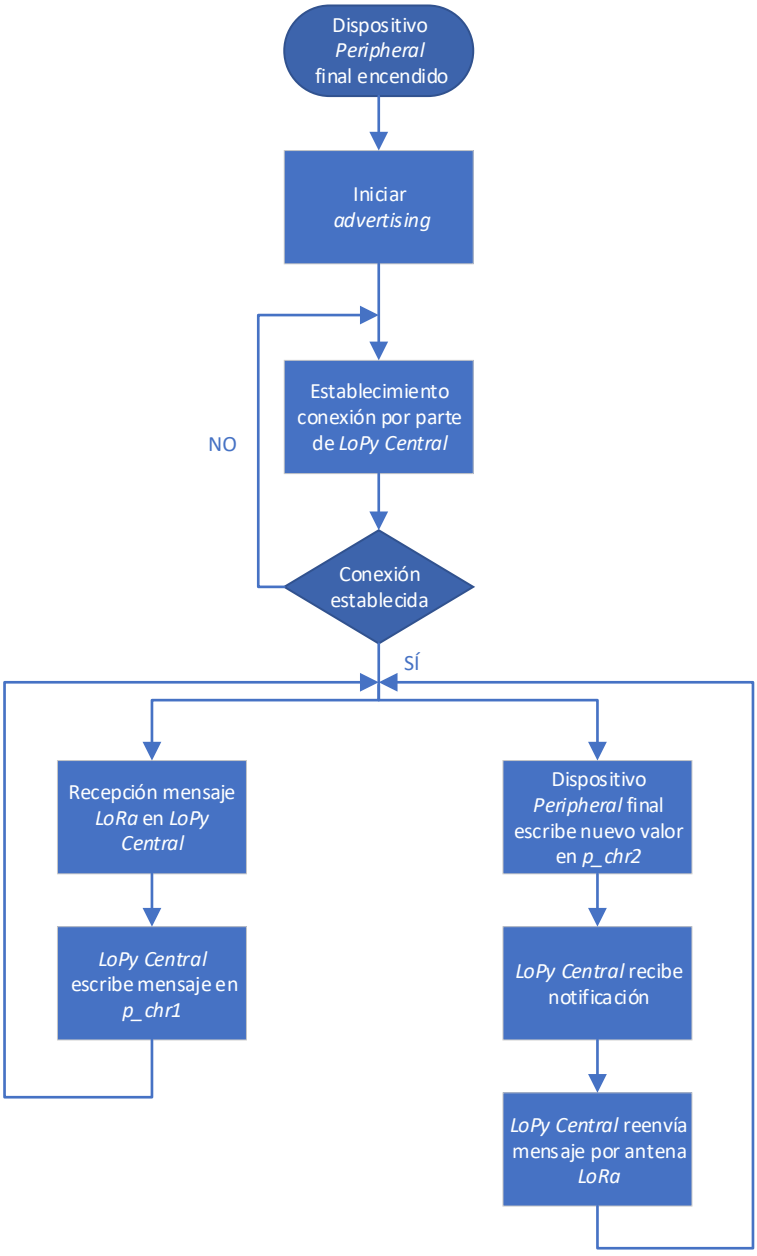

FIGURA 37: DIAGRAMA DE FLUJO DE LA PLATAFORMA HW/SW INICIAL (II)

5.3. Desarrollo del firmware

Para el desarrollo del *firmware*, tanto para BLE como para *LoRa*, se ha partido de las librerías recogidas en la documentación oficial de *Pycom*, así como de los ejemplos propuestos en éstas. Si bien a continuación se describen las estructuras y funciones empleadas en el desarrollo particular de este TFG, las librerías completas empleadas se han añadido en forma de referencias para facilitar la lectura de la memoria, dado que se trabaja con diferentes tecnologías y dispositivos.

Asimismo, se puede dividir el desarrollo del *firmware* en dos secciones bien diferenciadas, de la misma forma que se planteó en el apartado 5.1., presentándose el código correspondiente al dispositivo *LoPy Peripheral*, y al dispositivo *LoPy Central*.

5.3.1. LoPy Peripheral

El *firmware* de este dispositivo se encarga de implementar en el dispositivo *LoPy* el comportamiento de un dispositivo BLE *Peripheral*, así como de incluir la funcionalidad del protocolo de comunicación *LoRa*. Para ello, inicialmente se han importado las librerías de *LoPy* correspondientes al desarrollo de redes BLE y *LoRa*, como se observa en la Figura 38.

```
2  from network import Bluetooth
3  from network import LoRa
```

FIGURA 38: IMPORTACIÓN LIBRERÍAS BLE Y LORA

De la misma manera, ha sido necesario importar una serie de librerías para el correcto funcionamiento del *firmware*. En la Figura 39 se recogen los módulos importados de la librería del dispositivo *LoPy*.

```
3  import pycom
4  import socket
5  import time
6  import binascii
7  import machine
```

FIGURA 39: RESTO DE MÓDULOS IMPORTADOS

El módulo *pycom* contiene las funciones para controlar ciertas características concretas de los dispositivos de *Pycom*, como puede ser el control de los LED, los cuales se usarán para indicar el

estado de la conexión BLE. Para poder trabajar con el *socket* de *LoRa*, ha sido necesario importar el módulo *socket*. La gestión de temporizadores ha sido posible gracias al uso del módulo *time*. El módulo *binascii* se ha empleado para la conversión de los UUID de binario a formato *American Standard Code for Information Interchange* (ASCII). Por último, para la gestión de las funciones relacionadas con el *hardware*, se ha importado el módulo *machine*.

En el fragmento de código mostrado en la Figura 40 se recoge la configuración de la conectividad *LoRa* y BLE del dispositivo. En este apartado queda determinado que el dispositivo *LoPy* actuará como dispositivo BLE *Peripheral*, así como las características del modo *LoRa* y del *LoRa socket*.

```
9  pycom.heartbeat(False)
10 bluetooth = Bluetooth()
11 bluetooth.set_advertisement(name='LoPy', service_uuid=b'1234567890123456')
12
13 lora = LoRa(mode=LoRa.LORA, frequency= 866000000, bandwidth=LoRa.BW_125KHZ,
14 sf=7, preamble=8, coding_rate=LoRa.CODING_4_5, power_mode=LoRa.ALWAYS_ON,
15 tx_iq=False, rx_iq=False, public=False)
16 s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
```

FIGURA 40: CONFIGURACIÓN BLE Y LORA

Por defecto, el LED *heartbeat* del dispositivo *LoPy* parpadea en color azul cada 4 segundos para indicar que el sistema está activo, por lo que se ha optado por mantenerlo apagado, a través del módulo *pycom* e indicándolo con la siguiente línea de código `pycom.heartbeat(False)`.

Para establecer las características BLE del dispositivo, inicialmente se ha hecho uso del constructor para crear un objeto *Bluetooth*, llamado en este caso `bluetooth`, como se puede observar en la línea 10 de la sección de código mostrada en la Figura 40. En su configuración por defecto, los parámetros del objeto *Bluetooth* toman los siguientes valores:

- ***id***: solo hay un periférico *Bluetooth* disponible, por lo que siempre deberá estar a 0.
- ***mode***: actualmente el único modo que soporta es BLE.
- ***antenna***: con este parámetro se determina el uso de la antena externa o interna. Por defecto, la placa de desarrollo utiliza la antena interna.

Con la función `bluetooth.set_advertisement ()` se configuran los datos que se enviarán durante el proceso de *advertising*. Llamando a esta función se está indicando que el

dispositivo *LoPy* actuará como dispositivo BLE *Peripheral*. En caso de tratarse de un dispositivo BLE *Central*, en lugar del proceso de *advertising*, se llamará a la función que configure el proceso de *scanning*, como se indicará en el siguiente apartado. Los parámetros propios de la función que se han modificado son los siguientes:

- ***name***: nombre, en formato *String*, que se mostrará en los paquetes de *advertising*. En el caso actual se ha decidido llamarlo "*LoPy*", para facilitar el reconocimiento del dispositivo.

- ***service_uuid**:* UUID del servicio de *advertising*.

Además de los parámetros con los que se ha trabajado, esta función incluye dos parámetros adicionales, de los que no se ha hecho uso:

- ***service_data***: datos del servicio de *advertising*.

- ***manufacturer_data***: datos del fabricante incluidos en el paquete de *advertising*.

El constructor *LoRa* permite definir los parámetros de configuración, así como inicializar su funcionalidad. El objeto *LoRa* creado en este caso tiene las siguientes características:

- ***mode**:* puede ser tanto `LoRa.LORA` como `LoRA.LORAWAN`. Dado que no se está trabajando con redes *LoRaWAN*, sino con *LoRa*, se establece la primera opción.

- ***frequency**:* al encontrarse en la región europea, la frecuencia a la que trabajará la antena *LoRa* tendrá que estar en la banda de 868 MHz, en un valor comprendido entre los 863 y los 870 MHz.

- ***bandwith***: indica el ancho de banda del canal en KHz. Los valores aceptados en la banda de 868 MHz son `LoRa.BW_125KHZ` y `LoRa.BW_250KHZ`.

- ***sf***: determina el factor de propagación, con un valor comprendido entre 7 y 12.

- ***preamble***: configura el número de símbolos de *preamble.* El valor predeterminado es 8.

- ***coding_rate***: puede tomar los siguientes valores `LoRa.CODING_4_5`, `LoRa.CODING_4_6`, `LoRa.CODING_4_7` o `LoRa.CODING_4_8`, siendo elegido en este caso el primero de ellos.

- ***power_mode***: en el modo `ALWAYS_ON`, la radio está siempre escuchando por paquetes entrantes, aun cuando no se está realizando una transmisión.

- Los parámetros ***tx_iq*** y ***rx_iq*** habilitan la inversión TX IQ y RX IQ, respectivamente. Dado que no se hace uso de estos parámetros, se les asigna el valor *False*. De la misma forma, el argumento ***public*** se establece al valor *False*.

Finalmente, se crea el *socket LoRa*, el cual siempre debe crearse después de haber inicializado la tarjeta de red *LoRa*. Una vez definido el comportamiento del dispositivo para BLE y *LoRa*, se pasa a describir las funciones que componen el *firmware*. En el lenguaje *Python*, los bloques de funciones comienzan con la palabra clave *def* seguida del nombre de la función y los paréntesis. En estos paréntesis irán los parámetros o argumentos de entrada. El bloque de código dentro de cada función comienza con dos puntos (:) y está indentado. Así, en primer lugar, se define la función `conn_cb`, que servirá para detectar el estado de conexión con un dispositivo *Central*. El código correspondiente a esta función se encuentra en la Figura 41.

```
21  def conn_cb (bt_o):
22      events = bt_o.events()
23      if  events & Bluetooth.CLIENT_CONNECTED:
24          print("--------------------------------------------------------------")
25          print("- Dispositivo Cliente BLE conectado")
26          print("                                   ")
27          pycom.rgbled(0x007f00) # green
28      elif events & Bluetooth.CLIENT_DISCONNECTED:
29          print("--------------------------------------------------------------")
30          print("- Dispositivo Cliente BLE desconectado")
31          print("                                      ")
32          pycom.rgbled(0x7f0000) # red
33          time.sleep(5.0)
34          pycom.rgbled(0x000000)
35
36  bluetooth.callback(trigger=Bluetooth.CLIENT_CONNECTED
37  | Bluetooth.CLIENT_DISCONNECTED,handler=conn_cb)
38  bluetooth.advertise(True)
```

FIGURA 41: FUNCIÓN CORRESPONDIENTE A LA CONEXIÓN CON DISPOSITIVO CENTRAL

El método `bt_o.events()` devuelve el valor de los bits de *flag* que identifican los eventos que se han producido desde la última llamada al método. Al llamar a esta función se borran los eventos, motivo por el que se almacena su valor en la variable local `events`. Para comprobar el estado de conexión, se realiza una operación lógica AND entre la variable local `events` y las constantes `CLIENT_CONNECTED` y `CLIENT_DISCONNECTED`. En caso de que exista una conexión activa con un dispositivo *Central*, el bit correspondiente a la constante `CLIENT_CONNECTED` se encontrará activo, pasando el led RGB a verde para indicar el estado de conexión. Así, el bit correspondiente a la constante `CLIENT_DISCONNECTED` estará activo cuando el cliente se haya desconectado, pasando el led *Red Green Blue* (RGB) a rojo durante 5 segundos para posteriormente apagarse.

La función `conn_cb` solo puede ser llamada mediante *callback*. El método `bluetooth.callback()` crea un *callback* que se ejecutará cuando se produzca cualquiera de los *triggers*. Los argumentos de este método son:

- ***trigger***: en este caso, los *triggers* son `CLIENT_CONNECTED` y `CLIENT_DISCONNECTED`.

- ***handler***: se indica la función a ejecutar cuando se active el *trigger* de la devolución de llamada, siendo esta la función `conn_cb`.

- ***arg***: es el argumento que se le pasa a la devolución de llamada. Si no se incluye nada en este parámetro, como es el caso, se usa el objeto *bluetooth*.

Una vez definidos los constructores BLE y *LoRa*, así como la respuesta del sistema ante un estado de conexión o desconexión, se pasa a activar el proceso de *advertising* del dispositivo *LoPy* mediante el método `bluetooth.advertise()`. Fijando su valor a *True*, comienza el envío de paquetes de *advertising*. A continuación, se describe la creación de servicios y características BLE. Para ello, se declara en primer lugar la función `uuid2bytes()`, que se encargará de convertir el UUID del servicio y las características creadas a formato *byte*, ya que los métodos empleados para la creación de servicios y características no trabajan directamente con el formato por defecto de los UUID. En la Figura 42, se encuentra el código correspondiente a esta funcionalidad.

```
33  def uuid2bytes(uuid):
34      uuid = uuid.encode().replace(b'-',b'')
35      tmp = binascii.unhexlify(uuid)
36      return bytes(reversed(tmp))
```

FIGURA 42: FUNCIÓN DE CONVERSIÓN DE UUID A BYTE

Como se indicó en la descripción de esta solución, se cuenta con un servicio primario, denominado *srv1*, que a su vez contiene las características *chr1* y *chr2*. Las líneas de código correspondientes a la creación del servicio se encuentran en la Figura 43.

```
38  srv1=bluetooth.service(uuid=uuid2bytes('0000fff0-0000-1000-8000-00805f9b34fb')
39  , isprimary=True, nbr_chars=2, start=True)
```

FIGURA 43: CREACIÓN DE SERVICIO BLE

El método `bluetooth.service()` crea un nuevo servicio en el servidor interno GATT, devolviendo un objeto de tipo *BluetoothServerService*, el cual se almacena en la variable `srv1`. Los argumentos de este método son:

- ***uuid:*** corresponde al UUID del servicio. Dado que el tipo de dato introducido solo puede ser *int* o *byte*, se hace uso de la función `uuid2bytes()` para convertirlo a dato de tipo *byte*.

- ***isPrimary:*** para seleccionar si el servicio es primario o no. El valor es de tipo booleano, por lo que se fija a *True*, indiciando que se trata de un servicio primario.

- ***nbr_char:*** especifica el número de características que tendrá el servicio, siendo en este caso dos (*chr1* y *chr2*).

- ***start:*** cuando se fija su valor a *True*, se inicia el servicio inmediatamente.

Por otra parte, las líneas de código correspondientes a la creación de las características `chr1` y `chr2` se recogen en la Figura 44.

```
41  chr1=srv1.characteristic(uuid=uuid2bytes('0000fff1-0000-1000-8000-00805f9b34fb')
42  , properties=(Bluetooth.PROP_READ | Bluetooth.PROP_WRITE), value=5)
43
44  chr2=srv1.characteristic(uuid=uuid2bytes('0000fff1-0000-1000-8000-00805f9b3fff')
45  , properties=(Bluetooth.PROP_NOTIFY), value=0)
```

FIGURA 44: CREACIÓN CARACTERÍSTICAS BLE

El método `service.characteristic()` crea una nueva característica en el servicio en cuestión. Devuelve un objeto de la clase *GATTSCharacteristic*, el cual se almacenará en las variables *chr1* y *chr2*. Los argumentos de este método son:

- ***uuid:*** corresponde al UUID de la característica. Dado que el tipo de dato introducido solo puede ser *int* o *byte*, se hace uso de la función `uuid2bytes()` para convertirlo a dato de tipo *byte*.

- ***properties***: configura las propiedades de la característica. Su valor corresponde a un entero con una combinación de *flags*. En el caso de *chr1*, tiene las propiedades de lectura y escritura, por lo que las constantes activas son `Bluetooth.PROP_READ` y `Bluetooth.PROP_WRITE`. Así, *chr2* únicamente tiene la propiedad de notificación, por lo que la constante incluida en el argumento es `Bluetooth.PROP_NOTIFY`.

- ***permissions:*** configura los permisos de la característica. Su valor corresponde a un entero con una combinación de *flags*.

- ***value***: establece el valor inicial de la característica. Puede ser un objeto de tipo entero, *string* o *byte*.

Para la gestión de los eventos de lectura y escritura en la característica *chr1* se ha creado la función `char1_cb_handler()`, recogida en la Figura 45. Se trata de una función llamada mediante *callback*, siendo los *triggers* las constantes correspondientes a la lectura y escritura de características. Básicamente, esta función se encarga de reenviar el dato escrito a través de la antena *LoRa* en caso de escritura en la característica.

```
54  def char1_cb_handler(chr):
55      events = chr.events()
56      if  events & Bluetooth.CHAR_WRITE_EVENT:
57          print("-----------------------------------------------------------------")
58          print("- BLE: ESCRITURA en chr1 con valor = {}".format(chr.value()))
59          s.send(chr.value())
60          print("- LORA: Datos ENVIADOS por antena LoRa con valor = {}".
61          format(binascii.hexlify(chr.value())))
62          return 'ESCRITURA CORRECTA'
63      elif events & Bluetooth.CHAR_READ_EVENT:
64          print("-----------------------------------------------------------------")
65          print("- BLE: LECTURA EN chr1")
66          print("    ")
67      else:
68          print('Ningún evento de lectura o escritura')
69
70  char1_cb = chr1.callback(trigger=Bluetooth.CHAR_WRITE_EVENT |
71  Bluetooth.CHAR_READ_EVENT , handler=char1_cb_handler)
```

FIGURA 45: DEFINICIÓN DE LA FUNCIÓN CHAR1_CB_HANDLER()

El método `characteristic.events()` devuelve el valor de los bits de *flag* que identifican los eventos que se han producido desde la última llamada al método. Al llamar a esta función se borran los eventos, motivo por el que se guarda su valor en la variable local `events`. Para comprobar el estado de la característica *chr1*, se realiza una operación lógica AND entre la variable local `events` y las constantes `CHAR_WRITE_EVENT` y `CHAR_READ_EVENT`.

En caso de que exista una solicitud de escritura en la característica *chr1*, por parte del dispositivo final BLE *Central*, el bit correspondiente a la constante `CHAR_WRITE_EVENT` tendrá valor 1, por lo que se ejecutará la función. En este caso, se reenviará el mensaje a través de la antena *LoRa* haciendo uso de la función `s.send()`. Este método se encarga de enviar los datos pasados por parámetros, en formato *byte*, mediante el *socket LoRa* creado previamente. Además, se muestran por pantalla los datos enviados.

Así, el bit correspondiente a la constante `CHAR_READ_EVENT`, estará activo cuando se produzca una solicitud de lectura en la característica *chr1*. En este caso, únicamente se mostrará por pantalla que se ha realizado una solicitud de lectura.

En las líneas 70 y 71 se recoge la devolución de llamada, definida por el método `characteristic.callback()`, que hará que se ejecute la función `char1_cb_handler()`. Los *trigger* de este *callback* serán las constantes `CHAR_WRITE_EVENT` y `CHAR_READ_EVENT`, siendo el *handler* la función en cuestión. El parámetro *arg*, que indica el argumento que se le pasa a la devolución de llamada, al no estar especificado, hace referencia por defecto al objeto característica con el que se está trabajando, es decir, *chr1*.

Por tanto, únicamente quedaría por describir la función `lora_rx()`, encargada de la recepción de paquetes por la antena *LoRa*, para su posterior escritura en la característica *chr2*, cuyo valor será recibido por el dispositivo *Central* final como notificación. En la Figura 46 se muestra el código de esta función.

Inicialmente, se han importado las variables globales `s` y `chr2`, siendo la primera necesaria para el almacenamiento de los mensajes recibidos por la antena *LoRa*, y la segunda para la escritura en la característica que lleva su nombre.

```
73  def lora_rx(lora):
74      global s
75      global chr2
76      events=lora.events()
77      if events & LoRa.RX_PACKET_EVENT:
78          data = s.recv(64)
79          if (len(data) > 0):
80              print("-----------------------------------------------------------
81              print("- LORA: Datos RECIBIDOS por antena LoRa con valor = {}".
82              print("    ")
83              format(data))
84              print("- BLE: valor actualizado en chr2")
85              print("    ")
86              chr2.value(data)
87
88  lora.callback(trigger = LoRa.RX_PACKET_EVENT , handler = lora_rx)
```

FIGURA 46: DEFINICIÓN DE LA FUNCIÓN LORA_RX()

El método `lora.events()` devuelve un valor con un conjunto de bits que identifican el evento o eventos que han activado la devolución de llamada. Al llamar a este método, el registro interno de eventos se borra automáticamente, por lo que se almacena su valor en la variable local `events`, de la misma forma que en las funciones anteriores. Una vez que se ha activado el *trigger* y se ejecuta la función `lora_rx()`, se realiza una operación lógica AND entre la variable local `events` y la constante `RX_PACKET_EVENT`, propia de *LoRa*, que en caso de encontrarse con valor 1, indicaría la recepción de un paquete a través de la antena *LoRa*.

En caso de recibir un paquete, la función entrará dentro del *if* de la línea 77, pasando a almacenar el paquete recibido en la variable local `data`. La función `s.recv()` recupera los datos recibidos en el *socket LoRa*, indicándole por parámetro el tamaño del búfer.

Una vez almacenados estos datos, si su longitud es mayor que 0, es decir, no está vacío, se pasa a escribir el mensaje en la característica *chr2* mediante la función `chr2.value()`, indicando por parámetro el nuevo valor que tomará ésta. Como ya se ha indicado previamente, al tratarse de una característica que solo contiene la propiedad de notificación, al modificar el valor de *chr2*, el dispositivo suscrito a dicha característica será informado del cambio en su valor mediante notificación.

Finalmente, en la línea 88 se recoge la devolución de llamada, definida por el método `lora.callback()`, que hará que se ejecute la función `lora_rx()`. El *trigger* de este *callback* será la constante `RX_PACKET_EVENT`, siendo el *handler* la función en cuestión. El parámetro *arg*,

que indica el argumento que se le pasa a la devolución de llamada, al no estar definido, hace referencia por defecto al objeto *LoRa* con el que se está trabajando, es decir, *lora*.

De esta forma quedaría definido el código correspondiente a la implementación de un dispositivo BLE *Peripheral* completamente operativo en el dispositivo *LoPy*, incluyendo además conectividad *LoRa*, habilitando el envío y recepción de mensajes a través de la antena *LoRa*.

5.3.2. LoPy Central

El *firmware* de este dispositivo se encarga de implementar en el dispositivo *LoPy* el comportamiento de un dispositivo BLE *Central*, así como de incluir la funcionalidad del protocolo de comunicación *LoRa*. Para ello, al igual que en el caso anterior, se han importado las librerías del dispositivo *LoPy* necesarias para el desarrollo de redes BLE y *LoRa*, así como una serie de librerías adicionales para el correcto funcionamiento del *software*. En la Figura 47 se muestran las librerías añadidas.

```
1  from network import Bluetooth
2  from network import LoRa
3
4  import socket
5  import machine
6  import binascii
7  import time
```

FIGURA 47: LIBRERÍAS IMPORTADAS BLE CENTRAL

En el fragmento de código mostrado en la Figura 48 se recoge la configuración *LoRa* del dispositivo, definiendo las propiedades del modo *LoRa* y el *LoRa socket*. El motivo por el que no se añade su descripción es que se trata de la misma configuración que para el dispositivo *LoPy Peripheral*, la cual se encuentra en el apartado anterior. Únicamente, cabe recordar que el constructor *LoRa* permite definir los parámetros de configuración, así como inicializar su funcionalidad.

```
9  lora = LoRa(mode=LoRa.LORA, frequency= 866000000, bandwidth=LoRa.BW_125KHZ,
10 sf=7, preamble=8, coding_rate=LoRa.CODING_4_5, power_mode=LoRa.ALWAYS_ON,
11 tx_iq=False, rx_iq=False, public=False)
12 s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
```

FIGURA 48: CONFIGURACIÓN LORA EN DISPOSITIVO LOPY CENTRAL

Para establecer las características BLE del dispositivo, inicialmente se ha hecho uso del constructor para crear un objeto *Bluetooth*, llamado en este caso `bt`, como se puede observar en el segmento de código mostrado en la Figura 49. En su configuración por defecto, los parámetros del objeto *Bluetooth* toman los siguientes valores:

- ***id***: solo hay un periférico *Bluetooth* disponible, por lo que siempre deberá estar al valor 0.

- ***mode***: actualmente el único modo que soporta es BLE.

- ***antenna***: con este parámetro se determina el uso de la antena externa o interna. Por defecto, la placa de desarrollo utiliza la antena interna.

```
14  bt = Bluetooth()
15  bt.start_scan(-1)
```

FIGURA 49: CONFIGURACIÓN BLE EN DISPOSITIVO LOPY CENTRAL

El método `bt.start_scan()` permite realizar un proceso de escaneo, escuchando los paquetes de *advertising* que envían dispositivos BLE *Peripheral*. Esta función siempre retorna inmediatamente, realizándose el proceso de *scanning* en segundo plano. El uso de esta función indica que el dispositivo *LoPy* se va a comportar como un dispositivo BLE *Central*. El único parámetro de este método es:

- ***timeout***: especifica la cantidad de tiempo en segundos que el dispositivo se encontrará realizando el proceso de escaneo, no pudiendo ser 0. Si el tiempo de espera es mayor que 0, entonces la radio BLE escuchará los paquetes de *advertising* hasta que transcurra el valor especificado en segundos. Si el tiempo de espera es menor que 0, como en este caso, que es -1, entonces no hay tiempo de espera, debiendo llamar a la función `bt.stop_scan()` para cancelar el proceso de escaneo.

El dispositivo *LoPy Central* estará suscrito a una característica concreta del dispositivo final *Peripheral*. Asimismo, se encargará de escribir en otra característica los paquetes recibidos por la antena *LoRa*. Ambos procesos requieren que exista una conexión previa con el dispositivo *Peripheral*. Para ello, deberá escanear los paquetes de *advertising* hasta encontrar el correspondiente al dispositivo con el que se desea establecer conexión. Posteriormente, tendrá que identificar las características deseadas y almacenarlas en variables globales para su gestión en las diferentes funciones. En la Figura 50 se recoge este proceso de identificación.

```
while True:
  adv = bt.get_adv()
  if adv and bt.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL) == 'Bluz DK':
      try:
          bt.stop_scan()
          conn = bt.connect(adv.mac)
          print('Conexión BLE establecida con dispositivo Bluz DK')
          services = conn.services()
          for service in services:
              time.sleep(0.050)
              serv1 = service.uuid()
              if (serv1 == b'\xb2-\x14\xaa\xb3\x9fA\xed\xb1w\xff8#\x02\x1e\x87'):
                  servicio=service
                  print('Servicio deseado encontrado')
                  chars = service.characteristics()
                  for char in chars:
                      chr_t = char.uuid()
                      if (chr_t == b'\xb2-\x14\xaa\xb3\x9fA\xed\xb1w\xff8$\x02\x1e\x87'):
                          char2 = char
                          print('Característica 2 encontrada')
                      if (chr_t == b'\xb2-\x14\xaa\xb3\x9fA\xed\xb1w\xff8%\x02\x1e\x87'):
                          char1 = char
                          print('Característica 1 encontrada')
          break
      except:
          pass
  else:
      time.sleep(0.050)
```

FIGURA 50: IDENTIFICACIÓN DISPOSITIVO FINAL BLE PERIPHERAL

Para el proceso de identificación se hace uso de un bucle *while*, del que se saldrá una vez encontrado el dispositivo deseado, así como el servicio y características especificadas. Por definición, un bucle *while* estará iterando mientras la condición de salida sea evaluada como verdadera. Al tratarse de un *While True*, la expresión siempre se considerará verdadera por definición, por lo que se trata de un bucle infinito del que únicamente se saldrá usando *break*, *return*, o terminándolo de forma forzada.

El método `bt.get_adv()` obtiene una tupla con los datos de *advertising* recibidos durante el proceso de *scanning*, siendo la información obtenida la siguiente:

- ***mac***: dirección MAC de 6 *bytes* del dispositivo que envío el paquete de *advertising*.
- ***addr_type***: indica el tipo de dirección.
- ***adv_type***: tipo de paquete de *advertising* recibido.

- ***rssi***: dato de tipo entero con signo que indica la intensidad de la señal del paquete de *advertising*.

- ***data***: contiene los 31 *bytes* del mensaje de *advertising*. Para analizar los datos se puede utilizar el método `bt.resolve_adv_data()`.

Con el método `bt.resolve_adv_data()` se analizan los datos de *advertising* y devuelve el tipo de datos solicitado, en caso de estar presente. Si el tipo de datos no está presente, la función devuelve *None*. Los argumentos de este método son:

- ***data***: es el objeto *byte* con los datos completos del paquete de *advertising*. En este caso será la variable `adv`, donde se almacenó el resultado de la llamada a la función `bt.resolve_adv_data()`.

- ***data_type***: es el tipo de dato a obtener a partir de los datos de *advertising*.

En la Figura 51 se observa la línea de código en la que se compara el nombre del dispositivo que envía el paquete de *advertising* con el nombre del dispositivo al que se desea conectar. En caso de que se cumpla, se pasará a identificar sus servicios y características.

```
19    if adv and bt.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL) == 'Bluz DK':
```

FIGURA 51: IDENTIFICACIÓN DEL DISPOSITIVO FINAL BLE PERIPHERAL

Una vez identificado el dispositivo deseado, se interrumpe el proceso de *scanning*, haciendo uso del método `bt.stop_scan()`. A continuación es establece la conexión con el dispositivo final mediante el método `bt.connect()`. Este método se encarga de establecer una conexión BLE con el dispositivo especificado por el argumento *mac_addr*. La función se bloquea hasta que la conexión se realiza correctamente o falla. Si la conexión tiene éxito, devuelve un objeto de tipo *GATTCConnection*. El argumento *mac_addr* empleado se ha obtenido del mismo paquete de *advertising*, indicándolo con `adv.mac`. La conexión solo puede iniciarla el dispositivo *Central*, permitiendo un total de cuatro conexiones simultáneas.

El método `conn.services()` realiza una búsqueda de los servicios del dispositivo *Peripheral* BLE conectado y devuelve una lista que contiene objetos de la clase *GATT Service*, en caso de realizar favorablemente la búsqueda. Esta lista se ha almacenado en la variable `services`. Las líneas de código descritas en estos dos últimos párrafos se recogen en la Figura 52.

```
21            bt.stop_scan()
22            conn = bt.connect(adv.mac)
23            print('Conexión BLE establecida con dispositivo Bluz DK')
24            services = conn.services()
```

FIGURA 52: ESTABLECIMIENTO CONEXIÓN DISPOSITIVO FINAL BLE PERIPHERAL

El siguiente paso consiste en identificar un servicio específico para la posterior obtención de sus características. Con el bucle *for* que aparece en la Figura 53, se recorren los diferentes servicios dentro de la lista de servicios obtenida en el punto anterior, comparando el UUID del servicio con el UUID del servicio que se quiere localizar. El método `service.uuid()` devuelve el UUID del servicio. En el caso de UUID de 16 o 32 bits de longitud, el valor devuelto es un entero, pero para los UUID de 128 bits de longitud el valor devuelto es un objeto de tipo *byte*. Por tanto, inicialmente se tendrá que conocer el valor del UUID del servicio a identificar, para su posterior comparación con los servicios obtenidos.

```
25            for service in services:
26                time.sleep(0.050)
27                serv1 = service.uuid()
28                if (serv1 == b'\xb2-\x14\xaa\xb3\x9fA\xed\xb1w\xff8#\x02\x1e\x87'):
29                    servicio=service
30                    print('Servicio deseado encontrado')
```

FIGURA 53: IDENTIFICACIÓN DE SERVICIOS DEL DISPOSITIVO FINAL BLE PERIPHERAL

Una vez encontrado el servicio en cuestión, se obtienen sus características gracias al método `service.characteristics()`, como se observa en la Figura 54. Este método realiza una solicitud de obtención de características en el dispositivo *Peripheral* BLE conectado y devuelve una lista que contiene objetos de la clase *GATTCCharacteristic* si la solicitud se realiza correctamente. El objeto devuelto se almacena en la variable `chars`. Con el método `char.uuid()` se obtiene el UUID de la característica en formato *byte*, para después compararlo con las características buscadas, las cuales se almacenan en las variables `char1` y `char2`. Una vez localizadas, se obliga al sistema a salir del bucle *while* incluyendo un *break*.

```
31                  chars = service.characteristics()
32                  for char in chars:
33                      chr_t = char.uuid()
34                      if (chr_t == b'\xb2-\x14\xaa\xb3\x9fA\xed\xb1w\xff8$\x02\x1e\x87'):
35                          char2 = char
36                          print('Característica 2 encontrada')
37                      if (chr_t == b'\xb2-\x14\xaa\xb3\x9fA\xed\xb1w\xff8%\x02\x1e\x87'):
38                          char1 = char
39                          print('Característica 1 encontrada')
40          break
```

FIGURA 54: IDENTIFICACIÓN DE CARACTERÍSTICAS DEL DISPOSITIVO FINAL BLE PERIPHERAL

En este punto, se pasa a definir las funciones que componen el código para la implementación de un dispositivo BLE *Central* con conectividad *LoRa*. Para ello, se construye la función `lora_rx()` con el fin de indicar el comportamiento del dispositivo en caso de recibir un paquete a través de la antena *LoRa*, y la función `lora_tx()` para especificar los mensajes a transmitir por *LoRa*. Ambas funciones se ejecutarán mediante *callback*. El código de la función `lora_rx()` se encuentra en la Figura 55.

```
58  def lora_rx(lora):
59      global s
60      global char1
61      events=lora.events()
62      if events & LoRa.RX_PACKET_EVENT:
63          data = s.recv(64)
64          if (len(data) > 0):
65              char1.write(data)
66              print("------------------------------------------------------------")
67              print("- LORA: Datos RECIBIDOS por antena LoRa con valor = {}".
68              format(binascii.hexlify(char1.read())))
69              print("    ")
70              print("- BLE: Datos ESCRITOS en chr1")
71              print(" ")
72  lora.callback(trigger = LoRa.RX_PACKET_EVENT , handler = lora_rx)
```

FIGURA 55: CÓDIGO CORRESPONDIENTE A LA FUNCIÓN LORA_RX()

Inicialmente, se importa la variable global `s`, correspondiente al *socket LoRa*, y la variable global `char1`, que hace referencia a la característica en la que se realizará la escritura de los mensajes recibidos por la antena *LoRa*. En la variable local `events`, se guarda el objeto creado por el método `lora.events()`. Se trata de un conjunto de bits que identifican el evento o eventos que han activado la devolución de llamada. Al llamar a este método, el registro interno de eventos se borra automáticamente.

Cuando se active el *trigger* y se ejecute la función, se realizará una operación lógica AND entre la variable local `events` y la constante `RX_PACKET_EVENT`. En caso de encontrarse el bit correspondiente a la constante con valor 1, indicaría la recepción de un paquete a través de la antena *LoRa*, por lo que entraría en el *if*. A continuación, se almacena el paquete recibido en la variable local `data`. La función `s.recv()` recupera los datos recibidos en el *socket LoRa*, indicándole por parámetro el tamaño del *buffer*. Una vez almacenados los datos, si su longitud es mayor que 0, es decir, no está vacío, se pasa a escribir el mensaje en la característica *char1* mediante la función `char1.write()`, indicando por parámetro la variable local `data`. El método `char1.write()` escribe el valor dado en la característica, en formato *byte*.

Finalmente, en la línea 72 se recoge la devolución de llamada, definida por el método `lora.callback()`, que hará que se ejecute la función `lora_rx()`. El *trigger* de este *callback* será la constante `RX_PACKET_EVENT`, siendo el *handler* la función en cuestión. El parámetro *arg*, que indica el argumento que se le pasa a la devolución de llamada, al no estar definido, por defecto hace referencia al objeto *LoRa* con el que se está trabajando, es decir, `lora`.

La segunda y última función definida es `lora_tx()`, que se encarga de reenviar la notificación procedente del dispositivo BLE *Peripheral* final a través de la antena *LoRa*, de forma que el mensaje llegue al otro extremo de la pasarela. En la Figura 56 se recoge el código que implementa esta funcionalidad.

```
75  def lora_tx(chr):
76      global s
77      s.send(chr.value())
78      print("-----------------------------------------------------------------")
79      print("- BLE: Notificación RECIBIDA de chr2 con valor = {}".format(binascii.
80      hexlify(chr.value())))
81      print("    ")
82      print("- LORA: Datos ENVIADOS por antena LORA")
83      print("    ")
84
85  char2.callback(trigger=Bluetooth.CHAR_NOTIFY_EVENT, handler=lora_tx)
```

FIGURA 56: CÓDIGO CORRESPONDIENTE A LA FUNCIÓN LORA_TX()

Nuevamente se importa la variable global correspondiente al *socket LoRa* `s`, necesaria para el envío de paquetes *LoRa*. Una vez dentro de la función, se reenviará el mensaje recibido como notificación BLE hacia el extremo opuesto de la pasarela a través de la antena *LoRa*, mediante el *socket LoRa*, haciendo uso de la función `s.send()`. En la línea 85 se recoge la devolución de llamada, definida por el método `char2.callback()`. El *trigger* será la constante

`CHAR_NOTIFY_EVENT`, que hace referencia a las notificaciones de la característica `char2`. El *handler* es la función `lora_tx()`.

Así, una vez descrito el código implementado en este dispositivo *LoPy*, quedaría completamente definido el dispositivo BLE *Central* con funcionalidad *LoRa*, así como la pasarela correspondiente a esta primera solución. En el siguiente apartado se desarrolla el conjunto de pruebas llevadas a cabo para la comprobación y validación de la funcionalidad sistema.

5.4. Comprobación y validación

El proceso de comprobación del correcto funcionamiento de la plataforma inicial HW/SW desarrollada en este TFG, se basa en la verificación del intercambio de datos de forma bidireccional entre ambos extremos de la pasarela, es decir, entre los dispositivos BLE finales, comprobando la transferencia de los paquetes a través de todos los elementos de la pasarela.

Para ello, como dispositivo BLE *Central* final se ha empleado un *smartphone* ejecutando la aplicación *nRF Connect*, mientras que como dispositivo BLE *Peripheral* final se ha empleado el dispositivo *Bluz DK*. La planificación para la verificación del sistema ha seguido la misma estructura que el desarrollo del *firmware*, comprobando de forma independiente cada extremo de la pasarela, para finalmente validar el comportamiento del conjunto.

5.4.1. COMPROBACIÓN DEL EXTREMO DE LA PASARELA CORRESPONDIENTE AL DISPOSITIVO PERIPHERAL FINAL

Una vez desarrollado el *firmware* del dispositivo *LoPy Peripheral*, para comprobar que no existen errores de sintaxis habrá que cargar el *firmware* en el dispositivo *LoPy,* al no incluir el entorno de desarrollo *Atom* una opción para la verificación del código. En caso de que haya errores en el código, saldrá una notificación indicando que no se ha realizado satisfactoriamente la carga del *firmware* en el dispositivo, destacando el error y su ubicación en el código.

Para la gestión de los dispositivos de *Pycom*, la empresa ha desarrollado un *plugin* denominado *Pymakr*, para el editor de texto *Atom*, el cual se puede instalar directamente en el entorno de desarrollo, dentro de la opción *Settings*, en la pestaña de *Install*, como se muestra en la Figura 57. Dentro de esta pestaña, se introduce en el panel de búsqueda *Pymakr* y se acepta su instalación. Este *plugin* añade una consola *Read-Eval-Print-Loop* (REPL) a *Atom* para la conexión con

el dispositivo *LoPy*. Desde este *plugin* se puede ejecutar código o sincronizar los archivos del proyecto.

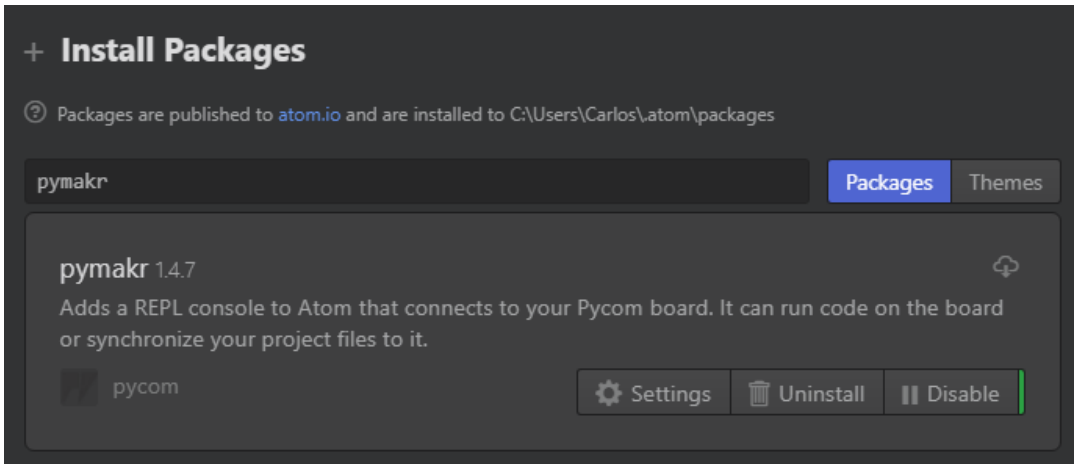

FIGURA 57: INSTALACIÓN DEL PLUGIN PYMAKR EN EL ENTORNO DE DESAROLLO ATOM

Una vez se haya completado la instalación y reiniciado el programa, aparecerá una nueva pestaña en *Atom*, correspondiente a la consola que añade el *plugin Pymakr*, al igual que se muestra en la Figura 58.

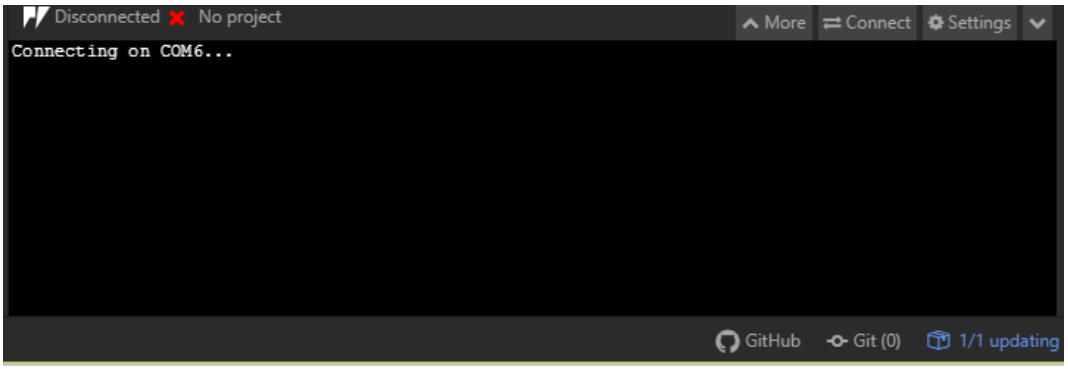

FIGURA 58: INTERFAZ DEL PLUGIN PYMAKR EN EL ENTORNO DE DESARROLLO ATOM

Como ya se ha comentado previamente, los dispositivos de la empresa *Pycom* trabajan con *MicroPython*, implementación 3.5 de *Python* optimizada para su ejecución en microcontroladores, permitiendo un desarrollo de procesos mucho más rápido y sencillo que empleando lenguaje *C*. Al iniciar el dispositivo, se ejecutan automáticamente dos archivos: primero *boot.py* y luego *main.py*. Estos se almacenan en la carpeta */flash*. Cualquier otro archivo o librería también irá ubicada en esta carpeta, aunque se pueden incluir directamente en los archivos *boot.py* o *main.py*.

El *plugin Pymakr* soporta la carga de proyectos de usuario, lo que permite configuraciones predefinidas, como son el puerto COM al que debe conectarse, los archivos que se deben cargar, y las carpetas de sincronización. El contenido del archivo de configuración del dispositivo *LoPy Peripheral*, llamado *pymakr.conf*, se recoge en la Figura 59.

```
{
    "address": "COM6",
    "username": "micro",
    "password": "python",
    "sync_folder": "",
    "sync_file_types": [
        "py",
        "txt",
        "log",
        "json",
        "xml",
        "html",
        "js",
        "css",
        "mpy",
        "pem",
        "cet",
        "crt",
        "key"
    ],
    "sync_all_file_types": false,
    "open_on_start": true,
    "safe_boot_on_upload": true,
    "py_ignore": []
}
```

FIGURA 59: ARCHIVO DE CONFIGURACIÓN PYMAKR.CONF DEL DISPOSITIVO LOPY PERIPHERAL

Para poder cargar en memoria *Flash* el *firmware* desarrollado dentro del proyecto, el archivo que contenga el código debe llamarse *main.py*. Así, el proyecto constaría de dos archivos, el principal con el *firmware* desarrollado, y el de configuración del *plugin Pymakr*, como se muestra en la Figura 60.

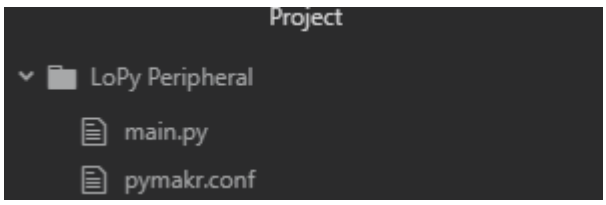

FIGURA 60: ARCHIVOS DEL PROYECTO LOPY PERIPHERAL

En el proceso de carga del proyecto en el dispositivo *LoPy*, inicialmente se conecta éste a un puerto USB del ordenador en el que se esté ejecutando el entorno de desarrollo *Atom*, y se selecciona en la consola *Read-Eval-Print-Loop* (REPL) la opción *Connect*, pasando a mostrar el estado *Connected* si la conexión se ha efectuado, como se muestra en la Figura 61, dando en este caso la opción de cargar el código, descargarlo, o ejecutarlo, en caso de que la ejecución se encuentre detenida.

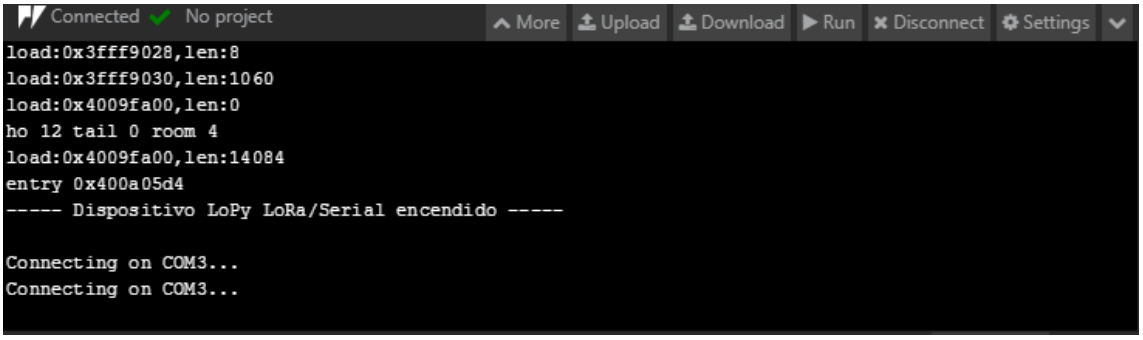

FIGURA 61: ESTABLECIMIENTO DE CONEXIÓN DE ATOM CON DISPOSITIVO LOPY PERIPHERAL

En este punto, se pasa a cargar el archivo *main.py* en el dispositivo *LoPy*, como se ve en la Figura 62, dejando completamente operativo el dispositivo *LoPy Peripheral*, y manteniendo abierta la consola para evaluar los eventos que ocurran durante la comunicación con la pasarela.

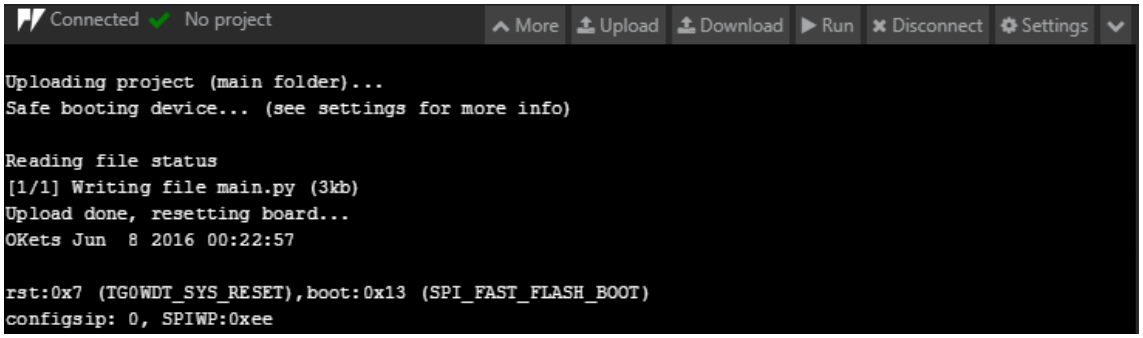

FIGURA 62: CARGA DEL PROYECTO EN EL DISPOSITIVO LOPY PERIPHERAL

Para la comprobación de este extremo de la pasarela se ha empleado como dispositivo *Central* final un *smartphone* ejecutando la aplicación *nRF Connect*. Así, en la Figura 63 se muestra la interfaz de la aplicación móvil, configurada para actuar como dispositivo BLE *Central*, donde se puede observar que al iniciar el proceso de *scanning*, este detecta los paquetes de *advertising* enviados por el dispositivo *LoPy Peripheral*, apareciendo como nombre del dispositivo "*LoPy*". Además, en esta pantalla principal se muestra una breve descripción del dispositivo *DUO*

Peripheral, mostrando su dirección MAC, la potencia de la señal recibida y los 128 bits del UUID del Servicio GAP.

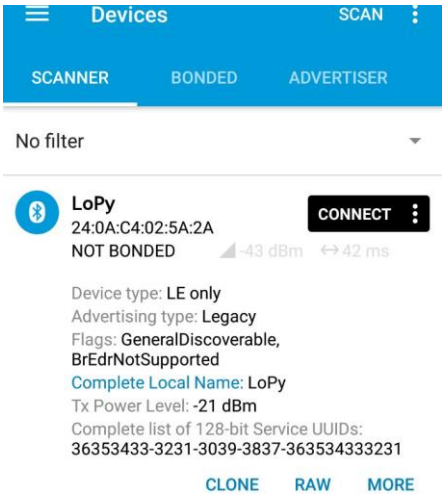

FIGURA 63: ESCANEO DISPOSITIVO LOPY PERIPHERAL EN APLICACIÓN NRF CONNECT

Una vez establecida la conexión BLE entre el terminal móvil y el dispositivo *LoPy*, se tiene acceso a sus diferentes servicios y características. Para la comunicación bidireccional con la pasarela se hace uso del servicio mostrado en la Figura 64. La característica mostrada en el rectángulo azul, con las propiedades de lectura y escritura, es la correspondiente a la descrita en la solución inicial como *chr1*, y es donde el dispositivo *Central* final escribirá los datos que desee enviar al dispositivo *Peripheral* final. Así, la característica mostrada en el rectángulo rojo corresponde a *chr2*, a la cual estará suscrito el dispositivo *Central* final, y por donde recibirá las respuestas en forma de notificación del dispositivo *Peripheral* final.

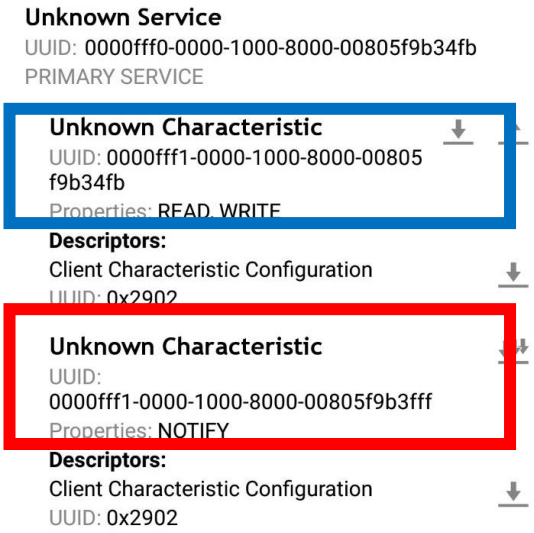

FIGURA 64: SERVICIO Y CARACTERÍSTICAS DEFINIDAS POR EL USUARIO VISTAS EN NRF CONNECT

En la comprobación del funcionamiento del dispositivo *LoPy Peripheral* se ha incluido en el otro extremo de la pasarela el dispositivo *Heltec*, actuando de forma que cuando reciba datos por la antena *LoRa*, provenientes del dispositivo *LoPy Peripheral*, aumentará en uno el valor de un contador, reenviando su nuevo valor por la antena *LoRa*. Este mensaje lo recibirá el dispositivo *LoPy* y lo escribirá en la característica *chr2* para que le dispositivo *Central* final lo reciba como notificación. El diagrama de bloques correspondiente a esta prueba se encuentra recogido en la Figura 65.

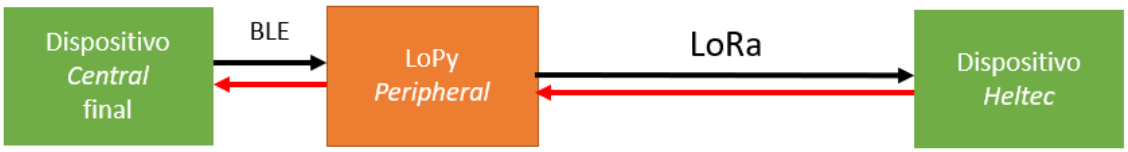

FIGURA 65: DIAGRAMA DE BLOQUES PARA LA COMPROBACIÓN DEL DISPOSITIVO LOPY PERIPHERAL

En primer lugar, se escribe en la característica *chr1* del dispositivo *LoPy Peripheral* cualquier dato, dado que el dispositivo *Heltec* responderá independientemente de los datos recibidos, por lo que se decide enviar el texto "Prueba". En la Figura 66 aparecen los mensajes de inicio mostrados por el dispositivo *LoPy Peripheral*, así como el mensaje que confirma la solicitud de escritura en la característica *chr1*, así como su reenvío por la antena *LoRa*.

```
----- Dispositivo LoPy Peripheral encendido -----

MicroPython v1.8.6-849-baa8c33 on 2018-01-29; LoPy with ESP32
Type "help()" for more information.
>>> ------------------------------------------------------------
- Dispositivo Cliente BLE conectado

------------------------------------------------------------
- BLE: Datos ESCRITOS en chr1 con valor = b'Prueba'

- LORA: Datos ENVIADOS por antena LoRa
```

FIGURA 66: ENVÍO DE DATOS AL EXTREMO OPUESTO DE LA PASARELA

En el dispositivo *Heltec*, una vez recibidos los datos, aumentará el valor de su contador en uno y reenviará su valor actualizado por la antena LoRa, el cual recibirá el dispositivo *LoPy*, actualizando el valor de la característica *chr2* con el correspondiente al del contador, como se ve en la Figura 67. Este mensaje lo recibirá como notificación el dispositivo *Central* final al estar suscrito a la característica *chr2*.

```
-------------------------------------------------------------
- BLE: Datos ESCRITOS en chr1 con valor = b'Prueba'

- LORA: Datos ENVIADOS por antena LoRa

-------------------------------------------------------------
- LORA: Datos RECIBIDOS por antena LoRa con valor = b' \ x01'

- BLE: Valor actualizado en chr2
```

FIGURA 67: MENSAJE RECIBIDO POR ANTENA LORA Y NOTIFICACIÓN EN CHR2

Así, quedaría verificado el funcionamiento del extremo de la pasarela correspondiente al Cliente BLE. En la Figura 68 se muestran varias comunicaciones sucesivas en las que se observa cómo se va actualizando el valor del contador del dispositivo *Heltec*.

```
-------------------------------------------------------------
- BLE: Datos ESCRITOS en chr1 con valor = b'Prueba'

- LORA: Datos ENVIADOS por antena LoRa

-------------------------------------------------------------
- LORA: Datos RECIBIDOS por antena LoRa con valor = b' \ x01'

- BLE: Valor actualizado en chr2

-------------------------------------------------------------
- BLE: Datos ESCRITOS en chr1 con valor = b'Prueba2'

- LORA: Datos ENVIADOS por antena LoRa

-------------------------------------------------------------
- LORA: Datos RECIBIDOS por antena LoRa con valor = b' \ x02'

- BLE: Valor actualizado en chr2
```

FIGURA 68: COMUNICACIONES PARA COMPROBAR EL FUNCIONEMIENTO DEL DISPOSITIVO LOPY PERIPHERAL

5.4.2. COMPROBACIÓN DEL EXTREMO DE LA PASARELA CORRESPONDIENTE AL DISPOSITIVO CENTRAL FINAL

Una vez desarrollado el *firmware* del dispositivo *LoPy Central*, se verifica y se carga en el dispositivo la aplicación siguiendo los pasos descritos en el apartado 5.4.1. Para el análisis de los eventos producidos en el dispositivo *LoPy Central* se vuelve a hacer uso de la consola REPL que ofrece el *plugin Pymakr* en el editor de código *Atom*. Quedando este extremo de la pasarela configurado, se pasa a verificar su funcionamiento. Para su comprobación, las pruebas se realizarán sobre la pasarela final directamente, dado que la sección del dispositivo *LoPy Peripheral* se encuentra completamente operativa y verificada. Por tanto, el dispositivo *Peripheral* final elegido

para comprobar el funcionamiento de la solución inicial desarrollada en el presente TFG será el dispositivo *Bluz DK*, como ya se ha mencionado. Para evitar caer en redundancias, el flujo de datos en el extremo de la pasarela ya comprobado se obviará.

Inicialmente, el dispositivo *LoPy Central* se encontrará en proceso de *scanning*, y una vez descubierto el dispositivo *Bluz DK*, establecerá conexión con el mismo, pasando a descubrir los servicios y características definidos por el usuario, como se ve en la Figura 69.

FIGURA 69: ESTABLECIMIENTO DE CONEXIÓN DE DISPOSITIVO LOPY CENTRAL CON DISPOSITIVO BLUZ DK

Una vez establecida la conexión, la pasarela se encontrará operativa, quedando a la espera de las solicitudes por parte del dispositivo *Central* final. Cuando se produzca una solicitud de encendido del LED, asociado a las cadenas de *bytes*, `/x04/x6E` y `/x03/x04`, los datos recorrerán la pasarela hasta ser recibidos, en el extremo de la pasarela correspondiente al dispositivo *Peripheral* final, por la antena *LoRa* del dispositivo *LoPy Central*, que automáticamente lo escribirá en la característica denominada *p_chr1* del dispositivo *Bluz DK*, resultando en el encendido del LED. Este intercambio de mensajes se recoge en la Figura 70.

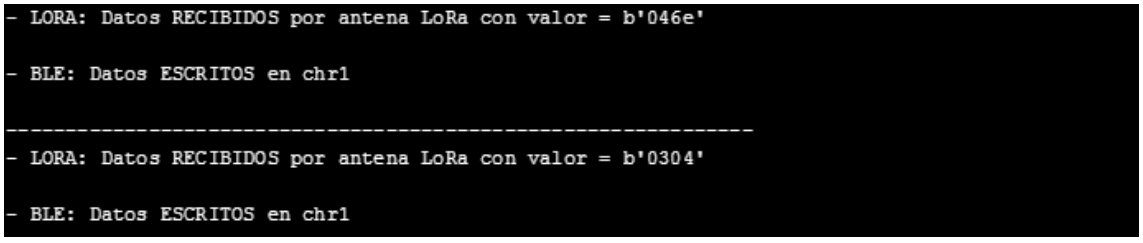

FIGURA 70: ESCRITURA DE LA SECUENCIA DE ENCENDIDO EN EL DISPOSITIVO BLUZ DK

Una vez que se produzca el evento de encendido del LED, el dispositivo *Bluz DK* emite una notificación en su otra característica, denominada *p_chr2*, a la que se encuentra suscrito el

dispositivo *LoPy Central*, que reenvía los datos por la antena *LoRa*, hacia el extremo correspondiente al Cliente BLE. La recepción de las notificaciones queda recogida en la Figura 71.

```
------------------------------------------------------------
- BLE: Notificación RECIBIDA de chr2 con valor = b'0304'

- LORA: Datos ENVIADOS por antena LORA
```

FIGURA 71: RECEPCIÓN DE NOTIFICACIÓN DE ENCENDIDO POR PARTE DEL DISPOSITIVO BLUZ DK EN EL DISPOSITIVO LOPY CENTRAL

Para el apagado del LED en el dispositivo *Peripheral* final, es necesario repetir el flujo de datos, con la diferencia de que las cadenas de *bytes* enviadas en este caso son `/x04/x6E` y `/x03/x04`, por lo que sería redundante repetir el proceso. Así, quedaría verificado el funcionamiento del extremo de la pasarela correspondiente al Servidor BLE y con ello quedaría implementada la pasarela completa de la solución inicial desarrollada en el presente TFG.

5.4.3. FIRMWARE DISPOSITIVO HELTEC

En este apartado se incluye como referencia, el *firmware* desarrollado para el dispositivo *Heltec* en la solución inicial desarrollada en el presente TFG. El código se muestra en la Figura 72, Figura 73 y Figura 74.

```
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include "SSD1306.h"
#include "images.h"
#define SCK     5    // GPIO5  -- SX1278's SCK
#define MISO    19   // GPIO19 -- SX1278's MISO
#define MOSI    27   // GPIO27 -- SX1278's MOSI
#define SS      18   // GPIO18 -- SX1278's CS
#define RST     14   // GPIO14 -- SX1278's RESET
#define DI0     26   // GPIO26 -- SX1278's IRQ(Interrupt Request)
#define BAND    866E6

// the OLED used
SSD1306 display(0x3c, 4, 15);
String rssi = "RSSI --";
String packSize = "--";
String packet ;

void logo(){
  display.clear();
  display.drawXbm(0,5,logo_width,logo_height,logo_bits);
  display.display();
}
```

FIGURA 72: FIRMWARE DISPOSITIVO HELTEC (I)

```
void loraData(){
  display.clear();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);
  display.drawString(0 , 15 , "Received "+ packSize + " bytes");
  display.drawStringMaxWidth(0 , 26 , 128, packet);
  display.drawString(0, 0, rssi);
  display.display();
}

void cbk(int packetSize) {
  packet ="";
  packSize = String(packetSize,DEC);
  int availableBytes = LoRa.available();
  Serial.println(availableBytes);
//  for (int i = 0; i < packetSize; i++) { packet += LoRa.read(); }
//  read packet
  while (LoRa.available()) {
    int dataread=LoRa.read();
    if (dataread <0x10) {packet += String(0, HEX);}
    packet += String(dataread, HEX);
    packet += String(" ");
  }
  rssi = "RSSI " + String(LoRa.packetRssi(), DEC) ;
  loraData();
  Serial.println(packet);
  int stringsize = display.getStringWidth(packet);

    Serial.println(stringsize);
}

void setup() {
  pinMode(16,OUTPUT);
  digitalWrite(16, LOW);    // set GPIO16 low to reset OLED
  delay(50);
  digitalWrite(16, HIGH); // while OLED is running, must set GPIO16 in high、

  Serial.begin(9600);
  while (!Serial);
  Serial.println();
  Serial.println("LoRa Receiver Callback");
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DIO);
//  LoRa.setSignalBandwidth(7.8E3); // By default 125E3!
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    display.drawString(0, 1, "Starting LoRa failed!");
    while (1);
  }
//  LoRa.onReceive(cbk);
//  LoRa.receive();
  Serial.println("init ok");
  display.init();
```

FIGURA 73: FIRMWARE DISPOSITIVO HELTEC (II)

```
//  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  logo();

//  LoRa.dumpRegisters(Serial);

  delay(1500);
}

void loop() {
  int packetSize = LoRa.parsePacket();
  if (packetSize) { cbk(packetSize);  }

//  delay(10);
}
```

FIGURA 74: FIRMWARE DISPOSITIVO HELTEC (III)

5.4.4. FIRMWARE DISPOSITIVO BLUZ DK

En este apartado se incluye como referencia, el *firmware* realizado para el dispositivo *Bluz DK* en la solución inicial desarrollada en el presente TFG. El código se muestra en la Figura 75 y Figura 76.

```
#include "application.h"
SYSTEM_MODE(MANUAL);

bool newData = false;
uint16_t sav_length;
uint8_t sav_data[20];
uint8_t rsp_data[20];

void dataCallbackHandler(uint8_t *data, uint16_t length) {
    sav_length = length;
    for (int i=0; i<length; i++) sav_data[i] = *data++;
    newData = true;
}

void setup() {
    pinMode(D7, OUTPUT);
    digitalWrite(D7, LOW);

    BLE.registerDataCallback(dataCallbackHandler);

    pinMode(D6, INPUT_PULLDOWN);
    if (digitalRead(D6) == HIGH) {
        SYSTEM_MODE(AUTOMATIC);
    }
}
```

FIGURA 75: FIRMWARE DISPOSITIVO BLUZ DK (I)

```
void loop() {
    System.sleep(SLEEP_MODE_CPU);
    if (newData)
    {
        switch (sav_data[0]) {
            case 0x6e: // led D7 ON
                digitalWrite(D7, HIGH);
                break;
            case 0x6f: // led D7 OFF
                digitalWrite(D7, LOW);
                break;
        }
        rsp_data[0] = 'O';
        rsp_data[1] = 'K';
        for (int i=0; i<sav_length; i++) rsp_data[i+2] = sav_data[i];

        BLE.sendData(rsp_data, (sav_length+2));
        newData = false;
    }
}
```

FIGURA 76: FIRMWARE DISPOSITIVO BLUZ DK (II)

CAPÍTULO 6: DESARROLLO DE LA PASARELA BASADA EN LOS DISPOSITIVOS LOPY Y REDBEAR DUO

En este capítulo se describe el desarrollo de la pasarela BLE-*LoRa*-BLE basada en los dispositivos *LoPy* y *RedBear DUO*. Una vez verificada la primera solución, se pasa a desarrollar una nueva pasarela, esta vez basada en las plataformas de desarrollo hardware *LoPy* y *RedBear DUO*, permitiendo nuevamente la conexión bidireccional entre dispositivos BLE a través de tecnología *LoRa*. La funcionalidad BLE se implementará en los dispositivos *RedBear DUO*, mientras que los dispositivos *LoPy* proporcionarán a la pasarela conectividad *LoRa*. La conexión entre los dispositivos *DUO* y los dispositivos *LoPy* se realizará de forma serial.

6.1. Objetivos de la implementación

En la solución inicial desarrollada, se buscaba probar el funcionamiento de la pasarela BLE-*LoRa*-BLE planteada partiendo de dispositivos que integraban, tanto conectividad *LoRa* como conectividad BLE (dispositivo *LoPy*), sin embargo, analizando el estado actual de la tecnología de comunicación *LoRa*, en términos de presencia en el mercado de las comunicaciones inalámbricas, se trata de una propuesta poco flexible. En cambio, BLE es una tecnología completamente integrada en el sector tecnológico, estando presente en prácticamente cualquier plataforma de desarrollo y, a nivel de consumidor, en la mayoría de los *smartphones* y dispositivos del mercado, convirtiéndola en la tecnología ideal para el desarrollo de una pasarela cuya finalidad es la interconexión de dispositivos mediante tecnología *LoRa*.

Así, se pasa a desarrollar una pasarela BLE-*LoRa*-BLE partiendo de una plataforma de desarrollo con conectividad BLE, pero que carece de conectividad *LoRa,* de manera que sea necesario incluir un módulo adicional o bien conectarlo a un microcontrolador diferente, para poder establecer en la pasarela la comunicación vía *LoRa*. Por tanto, se elige el dispositivo *RedBear DUO* para el establecimiento de conexión mediante tecnología BLE con los dispositivos finales, mientras que para proporcionar conectividad *LoRa* se eligen nuevamente los dispositivos *LoPy*, los cuales estarán conectados a los dispositivos *RedBear DUO* mediante conexión serie.

En la Figura 77 se recoge el esquema de bloques de la plataforma HW/SW desarrollada, basada en los dispositivos *RedBear DUO* y *LoPy*. Como dispositivo *Peripheral* final y dispositivo *Central* final se ha empleado el dispositivo *Bluz DK* y un *smartphone* ejecutando la aplicación *nRF Connect*, respectivamente. En esta representación gráfica se puede observar cómo queda establecida la comunicación bidireccional entre dispositivos con conectividad BLE a través de la tecnología de comunicación *LoRa*, haciendo uso de un dispositivo adicional para proporcionar a la pasarela conectividad *LoRa*.

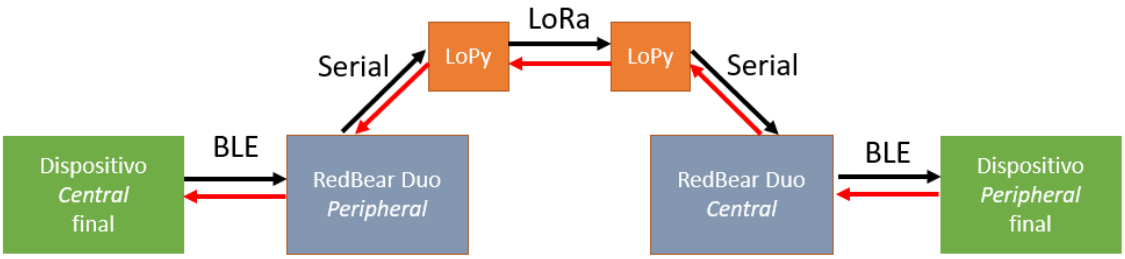

FIGURA 77: PASARELA BASADA EN LOS DISPOSITIVOS LOPY Y REDBEAR DUO

El *firmware* se ha desarrollado en el presente TFG de forma independiente para cada extremo de la pasarela. Así, se ha configurado un dispositivo *RedBear DUO* como dispositivo BLE *Peripheral*, para la interconexión con el dispositivo *Central* final. Este dispositivo BLE *Peripheral* cuenta con un servicio primario, en el que a su vez se han definido dos características:

- *chr1*: incluye las propiedades de lectura y escritura. Esta característica permitirá al dispositivo *Central* final escribir los datos que desea transmitir a través de la pasarela.

- *chr2*: incluye las propiedades de lectura y notificación. El dispositivo *Central* final estará suscrito a esta característica, a través de la cual recibirá los datos transmitidos por el dispositivo final *Peripheral*.

El dispositivo *RedBear DUO*, actuando como BLE *Peripheral*, estará conectado al dispositivo *LoPy* a través del canal *serial*, mediante los pines TX y RX de los dispositivos. Cuando se produzca una escritura en la característica *chr1*, se transmitirán los datos mediante comunicación *serial* al dispositivo *LoPy*, el cual reenviará los datos por la antena LoRa hacia el extremo opuesto de la pasarela. Así, cuando el dispositivo *LoPy* reciba datos por la antena *LoRa*, reenviará los mismos datos por el puerto serie al dispositivo *RedBear DUO*, que se encargará de escribirlos en la característica *chr2*. Cuando se produzca un cambio en el valor de *chr2*, el dispositivo *Central* final lo recibirá como notificación.

El dispositivo *LoPy*, a diferencia de la solución inicial, deja de implementar la funcionalidad BLE, para limitarse a reenviar los mensajes recibidos por puerto serie y antena *LoRa*. Dado que el protocolo de comunicación *LoRa* no contempla un establecimiento previo de conexión, el otro extremo de la pasarela se considera una caja negra. En la Figura 78 queda reflejada esta sección de la solución.

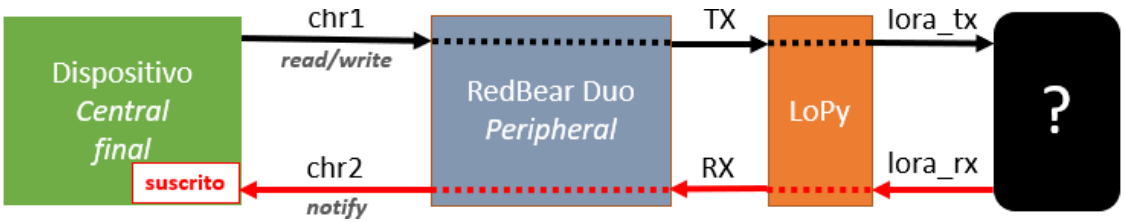

FIGURA 78: EXTREMO DE LA PASARELA CORRESPONDIENTE AL DISPOSITIVO REDBEAR DUO PERIPHERAL

En segundo lugar, se ha configurado un dispositivo *RedBear DUO* como dispositivo BLE *Central*, conectado igualmente al dispositivo *LoPy* mediante comunicación serial. En este caso, se realizará el proceso inverso al ya descrito, siendo el dispositivo *RedBear DUO* el encargado de

suscribirse a una característica concreta del dispositivo *Peripheral* final, denominada *p_chr2*. Una segunda característica, *p_chr1*, será usada por el dispositivo *RedBear DUO* para escribir las solicitudes enviada por el dispositivo final *Central*, en el otro extremo de la pasarela:

- *p_chr1*: incluye las propiedades de lectura y escritura. Esta característica permitirá al dispositivo *RedBear DUO Central* escribir en el dispositivo final *Peripheral* los datos recibidos por puerto serie del dispositivo *LoPy*, recibidos a su vez por la antena *LoRa*.

- *p_chr2:* incluye las propiedades de lectura y notificación. El dispositivo *RedBear DUO Central* estará suscrito a esta característica de manera que, al recibir una notificación, esta será reenviada por puerto serie al dispositivo *LoPy*, que a su vez lo reenviará a través de la antena *LoRa*.

En ambos extremos de la pasarela, el dispositivo *LoPy* actuará del mismo modo, reenviando los datos recibidos por puerto serie y la antena *LoRa*, por lo que se implementará el mismo *firmware*. En la Figura 79 se representa esta sección de la pasarela.

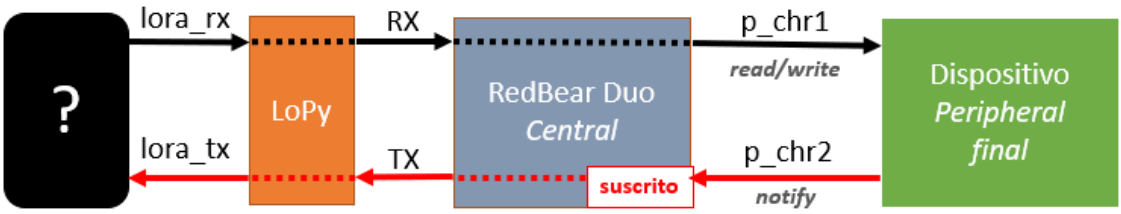

FIGURA 79: LADO DE LA PASARELA CON DISPOSITIVO REDBEAR DUO CENTRAL

Habiendo definido ambos extremos de la pasarela, se pasa a integrar las secciones que conforman la solución final desarrollada en el presente TFG, resultando en el diagrama de bloques representado en la Figura 80. Se sigue considerando una caja negra el espacio entre los dispositivos *LoPy*, debido a la ausencia de conexión e identificación entre ambos dispositivos, limitándose estos a enviar y recibir la información del medio.

FIGURA 80: PASARELA FINAL BASADA EN LOS DISPOSITIVOS LOPY Y REDBEAR DUO

6.2. Diagrama de flujo

El diagrama de flujo asociado con el funcionamiento del *firmware* de la plataforma HW/SW final que se ha desarrollado en el presente TFG se ha dividido en dos secciones para su mejor comprensión, representando cada una un extremo de la pasarela.

En la Figura 81 se representa el extremo de la pasarela correspondiente al dispositivo *Central* final, es decir, a la parte del cliente. Inicialmente, el dispositivo *Central* final iniciará el proceso de *scanning* en busca de paquetes de *advertising* del dispositivo *DUO Peripheral*. Una vez detectado el dispositivo *DUO*, el dispositivo *Central* final inicia el proceso de conexión. En caso de realizarse satisfactoriamente, el sistema estará preparado para iniciar la comunicación. En este punto, pueden tener lugar dos acciones: el proceso de escritura en la característica *chr1* del dispositivo *RedBear DUO Peripheral*, por parte del dispositivo final, o el envío de una notificación al dispositivo *Central* final, proveniente de la característica *chr2* del dispositivo *RedBear DUO Peripheral*.

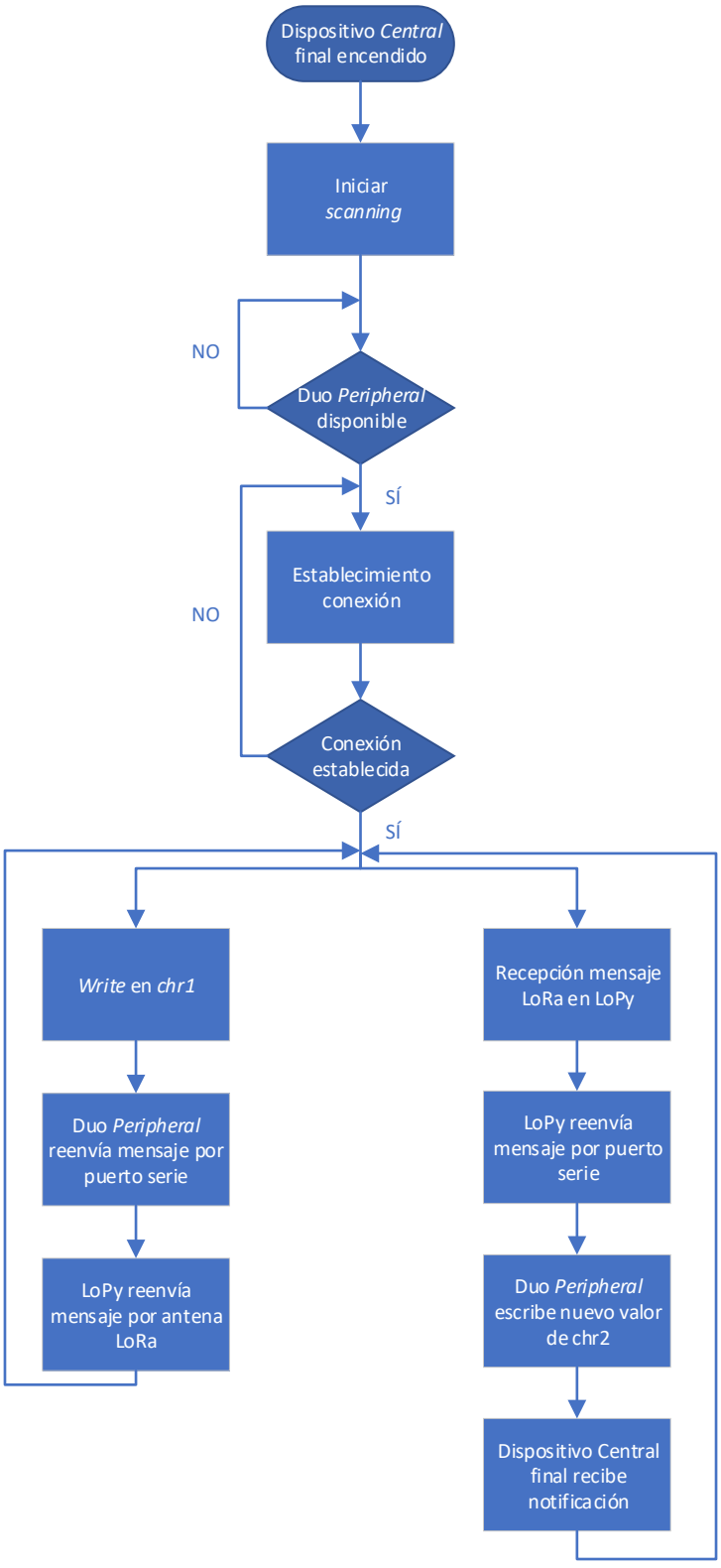

FIGURA 81: DIAGRAMA DE FLUJO DE LA PLATAFORMA HW/SW FINAL (I)

Así, en la Figura 82 se muestra el extremo de la pasarela correspondiente al servidor, es decir, el dispositivo *Peripheral* final, conectado mediante comunicación BLE al dispositivo *RedBear DUO*. El dispositivo *Peripheral final* iniciará el proceso de *advertising*, manteniéndose en éste hasta que el dispositivo *DUO Central* establezca una conexión. Entonces, pueden tener lugar dos acciones diferentes: escritura en la característica *p_chr1* o notificación en la característica *p_chr2*.

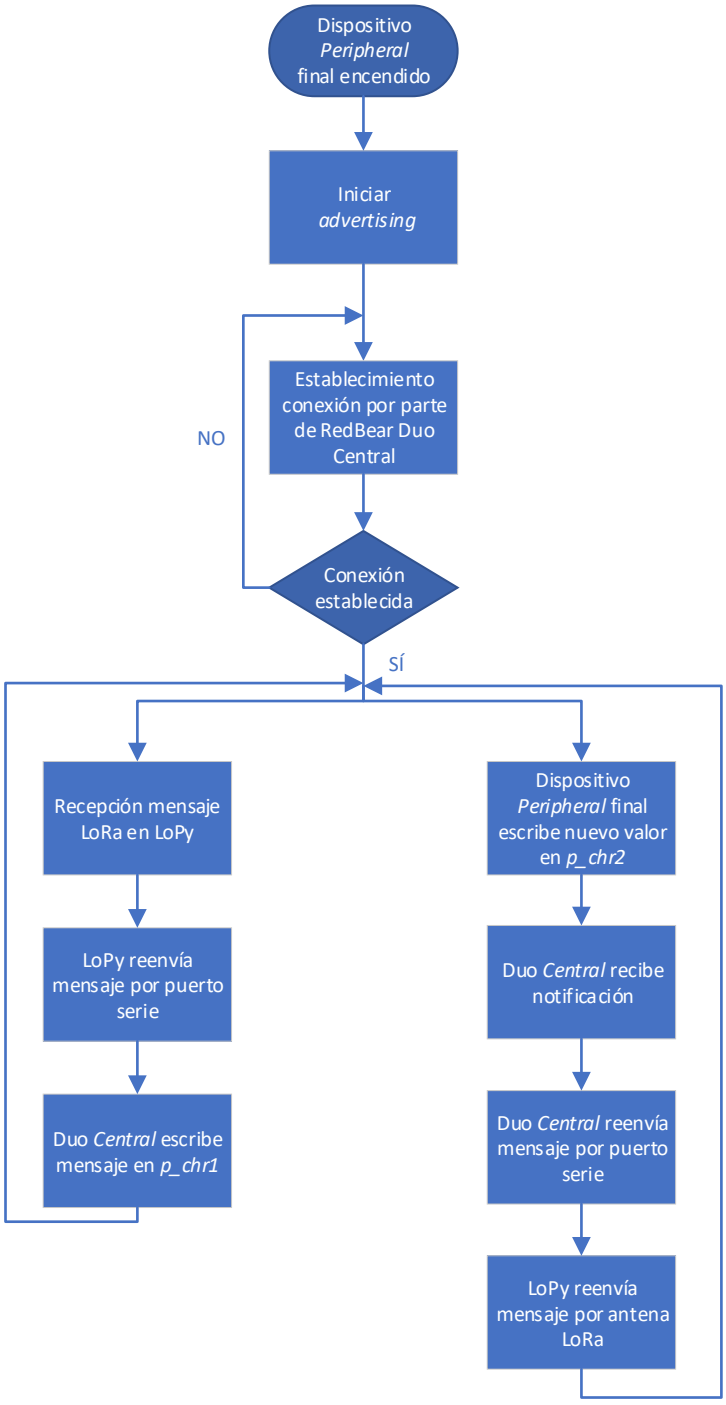

FIGURA 82: DIAGRAMA DE FLUJO DE LA PLATAFORMA HW/SW FINAL (II)

6.3. Desarrollo del firmware

Para el desarrollo del *firmware*, tanto para la implementación de la conectividad BLE como para la conectividad *LoRa*, se ha partido de las librerías propias de los dispositivos, recogidas en la documentación oficial, tanto de *Pycom*, en el caso de los dispositivos *LoPy*, como de *RedBear*, para los dispositivos *DUO*. Para el desarrollo del *firmware* de los dispositivos *RedBear DUO*, se ha partido de los modelos de referencia proporcionados en el repositorio *Github* de la empresa *RedBear*, así como de ejemplos que implementan el comportamiento de dispositivos BLE *Peripheral* y BLE *Central*, adaptándolos en cada caso para esta solución particular.

En los modelos seguidos para la implementación de un dispositivo BLE *Peripheral* y un dispositivo BLE *Central*, denominados `SimpleBLEPeripheral.ino` y `SimpleBLECentral.ino`, respectivamente, se establecen las bases que permiten a un dispositivo IoT establecer una comunicación BLE estable con otro dispositivo que utilice el rol de *Central* o *Peripheral*, según el caso. La metodología seguida parte inicialmente del análisis de las declaraciones de variables y constantes. Una vez descrito este punto, se procede al análisis de las funciones implementadas en el código. Los dispositivos *LoPy* implementarán ambos el mismo código, dado que se limitan a retransmitir por *LoRa* los datos recibidos por puerto serie y viceversa.

Así, el desarrollo del *firmware* se divide en tres secciones bien diferenciadas, presentándose en los siguientes apartados el código correspondiente a los dispositivos *LoPy*, al dispositivo *RedBear DUO Peripheral*, y al dispositivo *RedBear DUO Central*.

6.3.1. LoPy

El *firmware* de los dispositivos *LoPy* se encarga de implementar la conectividad del protocolo de comunicación *LoRa* y activar la comunicación serial a través de los pines TX/RX. Para ello, inicialmente se han importado las librerías del dispositivo *LoPy* correspondientes al desarrollo de redes *LoRa* y comunicaciones serie, como se observa en la Figura 83.

```
1  from network import LoRa
2  from machine import UART
```

FIGURA 83: IMPORTACIÓN LIBRERÍAS LORA Y UART EN DISPOSITIVO LOPY

Del mismo modo, ha sido necesario importar una serie de librerías para el correcto funcionamiento del *firmware*. En la Figura 84 se recogen los módulos importados de la librería del dispositivo *LoPy*.

```
4  import socket
5  import machine
6  import time
```

FIGURA 84: RESTO DE MÓDULOS IMPORTADOS EN DISPOSITIVO LOPY

El módulo `socket` contiene las funciones necesarias para trabajar con el *socket* de *LoRa*. Las funciones relacionadas con la gestión del *hardware* se importan con el módulo `machine`. Por último, para la gestión de temporizadores se ha importado el módulo `time`.

En el fragmento de código mostrado en la Figura 85 se recoge la configuración de la conectividad *LoRa* del dispositivo *LoPy*, quedando definidas las propiedades del modo *LoRa* y el *LoRa socket*. Se trata de la misma configuración que la descrita en la solución 1, por lo que se omite la descripción de los parámetros del constructor.

```
10 lora = LoRa(mode=LoRa.LORA, frequency= 866000000, bandwidth=LoRa.BW_125KHZ,
11 sf=7, preamble=8, coding_rate=LoRa.CODING_4_5, power_mode=LoRa.ALWAYS_ON,
12 tx_iq=False, rx_iq=False, public=False)
13
14 s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
```

FIGURA 85: CONFIGURACIÓN LORA EN DISPOSITIVO LOPY

El módulo UART importado implementa en el dispositivo el protocolo estándar de comunicaciones en serie dúplex UART. A nivel físico, consta de 2 líneas: RXD y TXD. La unidad de comunicación es un carácter, que puede tener 5, 6, 7 u 8 bits de ancho. Para implementarlo se ha hecho uso del constructor, creando un objeto UART, llamado `uart1`, como se puede observar en el segmento de código mostrado en la Figura 86.

```
16 uart1 = UART(1, 9600)
17 uart1.init(9600, bits=8, parity=None, stop=1)
```

FIGURA 86: CONFIGURACIÓN UART EN DISPOSITIVO LOPY

Con los parámetros introducidos en el objeto `uart1` se define el puerto UART usado y el *baudrate*. El método `uart1.init()` inicializa el bus UART con los siguientes parámetros:

- ***baudrate***: indica el número de unidades de señal por segundo.
- ***bits***: es el número de bits por carácter. Puede ser 5, 6, 7 u 8.
- ***parity***: indica la paridad. Puede tomar el valor `None` (sin paridad), `UART.EVEN` o `UART.ODD`.
- ***stop***: número de bits de *stop*, pudiendo tomar el valor 0 o 1.

Además de los parámetros definidos, se pueden configurar dos parámetros adicionales:

- ***timeout_char***: tiempo de espera para la recepción de datos, definido en número de caracteres. El valor dado se multiplicará por el tiempo que tardan en transmitirse los caracteres a la velocidad de transmisión configurada.
- ***pins***: es una lista de 2 o 4 elementos que indica los pines TXD, RXD, RTS y CTS (en ese orden).

La única función definida en el código implementado en los dispositivos *LoPy* es `lora_rx()`, recogida en la Figura 87. Esta función indica el comportamiento del dispositivo al recibir un paquete a través de la antena *LoRa*, y gestionada mediante interrupción. Únicamente cuando el *flag* correspondiente a la recepción de un paquete *LoRa* se active, se ejecutará la función.

```
23  def lora_rx(lora):
24      global s
25      events=lora.events()
26      if events & LoRa.RX_PACKET_EVENT:
27          data = s.recv(64)
28          print("-----------------------------------------------------------------")
29          print("- LORA: Datos RECIBIDOS con valor = {}".format(data))
30          print("   ")
31          if (len(data) > 0):
32              uart1.write(data)
33              print("- UART: Datos ENVIADOS por puerto serie")
34
35  lora.callback(trigger = LoRa.RX_PACKET_EVENT , handler = lora_rx)
```

FIGURA 87: FUNCIÓN LORA_RX EN DISPOSITIVO LOPY

Inicialmente, se importa la variable global `s`, correspondiente al *socket LoRa*, y se almacena en la variable `events` el objeto creado por el método `lora.events()`. Este método devuelve un conjunto de bits que identifican el evento, o eventos, que han activado la devolución de llamada. La llamada a este método borra automáticamente el registro interno de eventos.

Cuando se active el *trigger* de la devolución de llamada, se realizará una operación AND lógica entre la variable local `events` y la constante `RX_PACKET_EVENT`. En caso de cumplirse la condición del *if*, se almacenan en la variable local `data` los datos recibidos en el *socket LoRa*, recuperados mediante la función `s.recv()`. Una vez almacenados los datos, si la variable `data` no está vacía, se reenvía el mensaje por la UART, haciendo uso del método `uart1.write()`. Este método escribe el *buffer* de bytes, indicado por parámetro, en el bus. En todo momento se presenta por pantalla los datos recibidos y enviados.

Así, en la línea 35 de la Figura 87, se recoge la devolución de llamada, definida por el método `lora.callback()`. El *trigger* de este *callback* será la constante `RX_PACKET_EVENT`, siendo el *handler* la función `lora_rx()`. El parámetro *arq*, al no estar definido, hace referencia por defecto al objeto *LoRa* con el que se está trabajando, es decir, la variable global `lora`.

Finalmente, en la Figura 88 se muestra el código correspondiente a un bucle *while* encargado de evaluar mediante *polling* la recepción de paquetes en el pin *RX* de la UART. En un principio, se pretendía implementar mediante interrupción la recepción de mensajes por el puerto serie, pero en el momento de desarrollar el código aún no se había incluido en la librería una función de devolución de llamada para la UART, por lo que se tuvo que recurrir al bucle *while*.

```
37  while True:
38      if uart1.any():
39          data = uart1.readall()
40          print("--------------------------------------------------------------------")
41          print("- UART: Datos RECIBIDOS  con valor = {}".format(data))
42          print("    ")
43          s.send(data)
44          print("- LORA: Datos ENVIADOS por antena LORA:")
45          print("    ")
46      time.sleep(0.25)
```

FIGURA 88: BUCLE PARA LA RECEPCIÓN DE PAQUETES POR LA UART EN DISPOSITIVO LOPY

Se trata de un bucle infinito, dado que siempre se encuentra a la espera de recibir alguna transmisión por la UART. El método `uart1.any()` devuelve el número de caracteres disponibles para lectura. Por tanto, cuando exista algún mensaje en el *buffer* de entrada de la UART, se cumplirá

la condición del *if*. Entonces, se hace uso del método `uart.read()` para recuperar los caracteres recibidos, almacenándolos en la variable `data`. Una vez almacenados los datos, se retransmiten por la antena *LoRa* a través del *LoRa socket*, haciendo uso de la función `s.send()`. Para asegurar que el *buffer* de entrada se limpia completamente, se añade un temporizador de 0,25 segundos.

En definitiva, el comportamiento de los dispositivos *LoPy* quedaría completamente definido, estableciendo la conectividad *LoRa* y serial de estos. En los siguientes apartados se describe la implementación de la conectividad BLE en los dispositivos *RedBear DUO*, actuando como dispositivo *Peripheral* en un extremo de la pasarela, y como dispositivo *Central* en el otro.

6.3.2. REDBEAR DUO PERIPHERAL

El *firmware* desarrollado se encarga de implementar en el dispositivo *RedBear DUO* el comportamiento de un dispositivo BLE *Peripheral*, así como de habilitar la comunicación *serial* a través de la UART con un dispositivo *LoPy*.

6.3.2.1. Constantes

En la Figura 89 se muestra el código correspondiente a la definición de las constantes utilizadas para caracterizar los parámetros de conexión de la comunicación BLE.

```
1.  #define MIN_CONN_INTERVAL          0x0028 // 50ms.
2.  #define MAX_CONN_INTERVAL          0x0190 // 500ms.
3.  #define SLAVE_LATENCY              0x0000 // No slave latency.
4.  #define CONN_SUPERVISION_TIMEOUT   0x03E8 // 10s.
5.
6.  // Learn about appearance:
http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicV
iewer.aspx?u=org.bluetooth.characteristic.gap.appearance.xml
7.  #define BLE_PERIPHERAL_APPEARANCE  BLE_APPEARANCE_UNKNOWN
8.
9.  #define BLE_DEVICE_NAME            "BLE_Peripheral"
10.
11. // Length of characteristic value.
12. #define CHARACTERISTIC1_MAX_LEN    20
13. #define CHARACTERISTIC2_MAX_LEN    2
```

FIGURA 89: DEFINICIÓN CONSTANTES DISPOSITIVO DUO PERIPHERAL

En esta sección de código se han definido el intervalo máximo y mínimo del tiempo de conexión, la latencia admisible en la respuesta por parte del dispositivo *Slave*, y el *timeout* establecido para la supervisión de la conexión. Los valores que pueden tomar en cada caso estos parámetros son:

- **Mínimo intervalo de conexión**: es el resultado de la multiplicación `MIN_CONN_INTERVAL` * 1.25 ms, donde `MIN_CONN_INTERVAL` puede tomar valores comprendidos entre 0x0006 y 0x0C80, siendo el valor 0x0028 el seleccionado, resultando en un intervalo mínimo de conexión de 50 ms.

- **Máximo intervalo de conexión**: es el resultado de la multiplicación `MAX_CONN_INTERVAL` * 1.25 ms, donde `MAX_CONN_INTERVAL` puede tomar valores comprendidos entre 0x0006 y 0x0C80, siendo el valor 0x0028 el seleccionado, resultando en un intervalo máximo de conexión de 500 ms.

- **Latencia del dispositivo *Slave***: puede variar entre los valores 0x0000 y 0x03E8, siendo el valor establecido 0x0000.

- ***Timeout* de la supervisión de conexión**: se calcula multiplicando el valor de la constante `CONN_SUPERVISION_TIMEOUT` por 10 ms, donde `CONN_SUPERVISION_TIMEOUT` puede variar entre 0x000A y 0x0C80.

También se definen las constantes asociadas a la apariencia del dispositivo BLE *Peripheral*, así como el nombre del dispositivo. Por último, se asigna un valor a las constantes `CHARACTERISTIC1_MAX_LEN` y `CHARACTERISTIC2_MAX_LEN`, correspondientes a la longitud máxima de las características BLE, declaradas y comentadas más adelante.

6.3.2.2. Variables

En este apartado se recoge la declaración de las variables globales empleadas en el código. En la Figura 90 se muestra la definición del UUID del servicio y las características asociadas a la configuración del dispositivo BLE *Peripheral* en el *DUO*. También se definen el valor de las características propias de los perfiles GAP y GATT, así como la inicialización de los parámetros de conexión, definidos previamente en el apartado 6.3.2.1.

```
1.  // Primary service 128-bits UUID
2.  static uint8_t service1_uuid[16] = { 0x71,0x3d,0x00,0x00,0x50,0x3
e,0x4c,0x75,0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
3.  // Characteristics 128-bits UUID
4.  static uint8_t char1_uuid[16]    = { 0x71,0x3d,0x00,0x02,0x50,0x3
e,0x4c,0x75,0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
5.  static uint8_t char2_uuid[16]    = { 0x71,0x3d,0x00,0x03,0x50,0x3
e,0x4c,0x75,0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
6.
7.  // GAP and GATT characteristics value
8.  static uint8_t  appearance[2] = {
9.    LOW_BYTE(BLE_PERIPHERAL_APPEARANCE),
10.   HIGH_BYTE(BLE_PERIPHERAL_APPEARANCE)
11. };
12.
13. static uint8_t  change[4] = {
14.   0x00, 0x00, 0xFF, 0xFF
15. };
16.
17. static uint8_t  conn_param[8] = {
18.   LOW_BYTE(MIN_CONN_INTERVAL), HIGH_BYTE(MIN_CONN_INTERVAL),
19.   LOW_BYTE(MAX_CONN_INTERVAL), HIGH_BYTE(MAX_CONN_INTERVAL),
20.   LOW_BYTE(SLAVE_LATENCY), HIGH_BYTE(SLAVE_LATENCY),
21.   LOW_BYTE(CONN_SUPERVISION_TIMEOUT), HIGH_BYTE(CONN_SUPERVISION_TIME
OUT)
22. };
```

FIGURA 90: DEFINICIÓN VARIABLES DISPOSITIVO DUO PERIPHERAL (I)

En el código de la Figura 91 se recoge la definición de los parámetros asociados al proceso de *advertising*, en los que se indican los intervalos mínimo y máximo de envío de paquetes de *advertising*, el tipo de paquete de *advertising*, el tipo de dirección del paquete, los canales de envío de paquetes, y la política de filtrado. Únicamente se destaca que el tipo de paquete de *advertising* seleccionado es indirecto, determinado por `BLE_GAP_ADV_TYPE_ADV_IND`, por lo que los paquetes no están dirigidos a ningún dispositivo concreto en el proceso de *scanning*, es decir, se presenta como un dispositivo *Peripheral* escaneable y conectable para cualquier dispositivo *Central*. Los demás tipos de paquetes de *advertising* elegibles son los siguientes:

- `BLE_GAP_ADV_TYPE_ADV_DIRECT_IND`: conectable, no escaneable y directo.
- `BLE_GAP_ADV_TYPE_ADV_SCAN_IND`: no conectable, escaneable e indirecto.
- `BLE_GAP_ADV_TYPE_ADV_NONCONN_IND`: no conectable, no escaneable e indirecto.

Asimismo, se ha establecido la información relativa al nombre del dispositivo que recibirán los dispositivos en proceso de *scanning*, siendo el nombre elegido: “DUO PERIPHERAL”.

```
1. static advParams_t adv_params = {
2.    .adv_int_min   = 0x0030,
3.    .adv_int_max   = 0x0030,
4.    .adv_type      = BLE_GAP_ADV_TYPE_ADV_IND,
5.    .dir_addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
6.    .dir_addr      = {0,0,0,0,0,0},
7.    .channel_map   = BLE_GAP_ADV_CHANNEL_MAP_ALL,
8.    .filter_policy = BLE_GAP_ADV_FP_ANY
9. };
10.
11.// BLE peripheral advertising data
12.static uint8_t adv_data[] = {
13.  0x02,
14.  BLE_GAP_AD_TYPE_FLAGS,
15.  BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE,
16.
17.  0x11,
18.  BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE,
19.  0x1e, 0x94, 0x8d, 0xf1, 0x48, 0x31, 0x94, 0xba, 0x75, 0x4c, 0x3e, 0x
50, 0x00, 0x00, 0x3d, 0x71
20.};
21.
22.// BLE peripheral scan respond data
23.static uint8_t scan_response[] = {
24.  0x08,
25.  BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME,
26.  'D', 'U', 'O', '', 'P', 'E', 'R', 'I', 'P', 'H', 'E', 'R', 'A', 'L'
27.};
```

FIGURA 91: DEFINICIÓN VARIABLES DISPOSITIVO DUO PERIPHERAL (II)

Finalmente, en la Figura 92 se define el valor del *handle* de las características BLE implementadas, y se crean los vectores en los que se almacenará el valor de éstas, teniendo como tamaño máximo el indicado por las constantes `CHARACTERISTIC1_MAX_LEN` y `CHARACTERISTIC2_MAX_LEN`, en cada caso. También se define la variable `connected_id` para verificar el estado de la conexión. Así, quedarían declaradas las variables globales utilizadas en el código correspondiente a la implementación de un dispositivo *Peripheral* en el dispositivo *DUO*.

```
1. // Characteristic value handle
2. static uint16_t character1_handle = 0x0000;
3. static uint16_t character2_handle = 0x0000;
4.
5. // Buffer of characterisitc value.
6. uint8_t characteristic1_data[CHARACTERISTIC1_MAX_LEN] = {0x00};
7. uint8_t characteristic2_data[CHARACTERISTIC2_MAX_LEN] = {0x00};
8.
9. uint16_t characteristic1_data_len = CHARACTERISTIC1_MAX_LEN;
10.
11. // Estado de la conexión
12. static uint16_t connected_id = 0xFFFF;
```

FIGURA 92: DEFINICIÓN VARIABLES DISPOSITIVO DUO PERIPHERAL (III)

6.3.2.3. Funciones

En este apartado se procede a analizar las funciones incluidas en el código para la implementación de la conectividad BLE y la configuración del comportamiento de la UART ante la recepción y el envío de paquetes, con el objetivo de establecer la comunicación deseada entre los dispositivos utilizados. En la Figura 93 se indica el fragmento de código correspondiente a la función `deviceConnectedCallback()`.

```
1. void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
2.   switch (status) {
3.     case BLE_STATUS_OK:
4.       Serial.println("Device connected!");
5.       connected_id = handle;
6.       Serial.print("  - Device connected handle: ");
7.       Serial.println(connected_id);
8.       break;
9.     default: break;
10.      }
11.   }
```

FIGURA 93: CÓDIGO FUNCIÓN DEVICECONNECTEDCALLBACK() DISPOSITIVO DUO PERIPHERAL

Se trata de una función de devolución de llamada que se ejecutará cuando se produzca la conexión con un dispositivo BLE *Central*. La función toma dos parámetros y no devuelve nada. Uno de los parámetros, `BLEStatus_t`, refleja el estado de la conexión, y puede tomar cualquiera de los siguientes valores: `BLE_STATUS_CONNECTION_ERROR` o `BLE_STATUS_OK`. El siguiente parámetro es un *handle* de tipo *uint16_t* asignado en caso de que el establecimiento de conexión sea correcto. El *handle* tomará el valor 0xFFFF en caso de que la conexión no sea válida. En caso de que el parámetro `status` tenga el valor `BLE_STATUS_OK`, se actualiza el valor de la variable global `connected_id` con el del parámetro *handle*, mostrando a su vez por pantalla mensajes que indican que la conexión se ha realizado de forma satisfactoria y presentando el nuevo valor del *handle*.

Análogamente, en la Figura 94 se incluye el código correspondiente a la devolución de llamada ejecutada cuando se produce la desconexión de un dispositivo. Esta función toma un parámetro de tipo *uint16_t*, que indica el estado de la conexión, y no devuelve nada. La función se limita a cambiar la variable `connected_id` al valor de desconexión.

```
1. void deviceDisconnectedCallback(uint16_t handle) {
2.   Serial.println("Device disconnected!");
3.   connected_id = 0xFFFF;
4. }
```

FIGURA 94: CÓDIGO FUNCIÓN DEVICEDISCONNECTEDCALLBACK DISPOSITIVO DUO PERIPHERAL

El código de la función `gattReadCallback()`, *callback* correspondiente a las solicitudes de lectura en las características BLE del dispositivo *DUO Peripheral*, se muestra en la Figura 95. Esta función contiene tres parámetros, descritos a continuación, y devuelve un objeto de tipo *uint16_t*, indicando el tamaño de la característica a la que se le solicitó la lectura:

- ***value_handle***: el *handle* de la característica, de tipo *uint16_t*.
- ***buffer***: puntero de tipo *uint8_t* al *buffer* que leerá el dispositivo *Central*.
- ***buffer_size***: valor de tipo *uint16_t* que indica la longitud del *buffer*.

```
1.  uint16_t gattReadCallback(uint16_t value_handle, uint8_t * buffer, ui
nt16_t buffer_size) {
2.
3.    uint8_t characteristic_len = 0;
4.
5.    Serial.print("Read value handler: ");
6.    Serial.println(value_handle, HEX);
7.
8.    if (character1_handle == value_handle) {
9.      Serial.print("  - characteristic1 read: ");
10.     memcpy(buffer, characteristic1_data, characteristic1_data_len);
11.     characteristic_len = CHARACTERISTIC1_MAX_LEN;
12.     for (uint8_t index = 0; index < CHARACTERISTIC1_MAX_LEN;index++){
13.       Serial.print(buffer[index], HEX);
14.     }
15.     Serial.println();
16.   }
17.   else if (character2_handle == value_handle) {
18.     Serial.print("  - characteristic2 read: ");
19.     memcpy(buffer, characteristic2_data, CHARACTERISTIC2_MAX_LEN);
20.     characteristic_len = CHARACTERISTIC2_MAX_LEN;
21.     for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN;index++){
22.       Serial.print(buffer[index], HEX);
23.     }
24.     Serial.println();
25.   }
26.   return characteristic_len;
27. }
```

FIGURA 95: CÓDIGO FUNCIÓN GATTREADCALLBACK() DISPOSITIVO DUO PERIPHERAL

Cuando el valor del parámetro `value_handle` coincida con el correspondiente al de la característica *chr1*, indicará una solicitud de lectura en la característica mencionada. Así, cuando `value_handle` coincida con el valor del *handle* de *chr2*, se estaría efectuando una solicitud de lectura en esta otra característica. En ambos casos, cuando tenga lugar esta situación, haciendo uso del método `memcpy()`, se copia en el parámetro `buffer` el valor almacenado en la variable

characteristic1_data o characteristic2_data, según el caso, imprimiendo por pantalla los datos copiados. Por otro lado, en la Figura 96 se muestra el código correspondiente a la devolución de llamada en caso de escritura en una característica BLE.

```
1.  int gattWriteCallback(uint16_t value_handle, uint8_t *buffer, uint16_
t size) {
2.    Serial.print("Write value handler: ");
3.    Serial.println(value_handle, HEX);
4.
5.    if (character1_handle == value_handle) {
6.      characteristic1_data_len = size;
7.      memcpy(characteristic1_data, buffer, size);
8.
9.      Serial.print("  - characteristic1 write value: ");
10.     for (uint8_t index = 0; index < size; index++) {
11.       Serial.print((char) characteristic1_data[index]);
12.
13.     }
14.     Serial.println(" ");
15.
16.     Serial1.write(characteristic1_data, size );
17.     Serial.print("Enviado por puerto serie: ");
18.     for (uint8_t index2 = 0; index2 < size; index2++) {
19.       Serial.print((char) characteristic1_data[index2]);
20.
21.     }
22.     Serial.println(" ");
23.   }
24.   else if (character2_handle+1 == value_handle) {    // Client
Characteristic Configuration Descriptor Handle.
25.     Serial.print("  - characteristic2 CCCD write value: ");
26.     for (uint8_t index = 0; index < size; index++) {
27.       Serial.print(buffer[index], HEX);
28.     }
29.     Serial.println(" ");
30.   }
31.   return 0;
32. }
```

FIGURA 96: CÓDIGO FUNCIÓN GATTWRITECALLBACK() DISPOSITIVO DUO PERIPHERAL

La función gattWriteCallback() tiene los mismos parámetros que la función gattReadCallback(), devolviendo en este caso un valor de tipo *int*. Cuando el valor del parámetro value_handle coincida con el *handle* de alguna de las características, se procederá a copiar el valor del parámetro *buffer* en las variables characteristic1_data o characteristic2_data, según corresponda, haciendo uso del método memcpy(), e imprimiendo por pantalla el valor escrito en la característica. Así, quedaría actualizado el valor de la característica en la que se ha producido la solicitud de escritura. La función correspondiente a la recepción de datos por el pin RX se recoge en la Figura 97.

```
1.  void serialEvent1(){
2.      uint8_t size  = 0;
3.      if(Serial1.available()){
4.          size = min (Serial1.available(),15);
5.          uint8_t buf[size];
6.          for(uint8_t i=0; i<size;i++){
7.              buf[i] = Serial1.read();
8.          }
9.          ble.sendNotify(character2_handle, buf, CHARACTERISTIC2_MAX_
LEN);
10.     }
11. }
```

FIGURA 97:CÓDIGO FUNCIÓN SERIALEVENT1() DISPOSITIVO DUO PERIPHERAL

La función `serialEvent1()` se ejecutará cuando haya datos disponibles en el puerto `Serial1`, entre las llamadas al bucle de aplicación `loop()`. Esto significa que si la función `loop()` se ejecuta durante un tiempo prolongado debido a retardos u otras llamadas de bloqueo, el *buffer* del puerto serie podría llenarse entre las diferentes llamadas a `serialEvent1()`, existiendo la posibilidad de perder caracteres. Por tanto, es necesario evitar las llamadas con *delays* prolongados.

Dentro del código de la función se declara la variable local `size`, de tipo *uint8_t*, inicializándola al valor 0. Para comprobar que haya datos disponibles se hace uso del método `Serial1.available()`. Este método obtiene el número de bytes disponibles para leer desde el puerto serie. Estos son datos ya recibidos que se han almacenado en el *buffer* de recepción serial. El tamaño del *buffer* de recepción para el canal serie `Serial1` es de 64 bytes.

En caso de que el valor devuelto por el método `Serial1.available()` sea mayor que 0, se almacena en la variable local `size` el número de bytes disponibles, obtenidos gracias a la función `min()`. La función `min()` calcula el valor mínimo entre dos números, incluidos como parámetros. En este punto, se crea un *buffer* del tamaño indicado por la variable `size` y, haciendo uso de un bucle *for*, se van almacenando en él los caracteres recibidos en el *buffer* propio del puerto serie, mediante la función `Serial1.read()`. Este método devuelve el primer *byte* de datos disponible.

Finalmente, se envían como notificación BLE de la característica *chr2* los datos recibidos por puerto serie, gracias a la función `ble.sendNotify()`, que notifica al dispositivo *Central* final que el valor de la característica ha cambiado. Solo se podrá enviar la notificación si el cliente GATT está suscrito a ella. Toma tres parámetros: el *handle* de la característica, los datos que se enviarán, y la longitud de estos. La longitud máxima de los datos está limitada a 20 bytes.

La función `setup()`, ejecutada al inicio del programa, se recoge en la Figura 98, y es utilizada para definir las propiedades iniciales del entorno.

```
1.  void setup() {
2.
3.    Serial1.begin(9600);
4.    Serial.begin(115200);
5.    delay(5000);
6.    Serial.println("RedBear DUO BLE Peripheral");
7.
8.    // Initialize ble_stack.
9.    ble.init();
10.
11.   // Register BLE callback functions.
12.   ble.onConnectedCallback(deviceConnectedCallback);
13.   ble.onDisconnectedCallback(deviceDisconnectedCallback);
14.   ble.onDataReadCallback(gattReadCallback);
15.   ble.onDataWriteCallback(gattWriteCallback);
16.
17.   // Add GAP service and characteristics
18.   ble.addService(BLE_UUID_GAP);
19.   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_DEVICE_NAME, ATT_
PROPERTY_READ|ATT_PROPERTY_WRITE, (uint8_t*)BLE_DEVICE_NAME, sizeof(BLE_D
EVICE_NAME));
20.   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_APPEARANCE, ATT_P
ROPERTY_READ, appearance, sizeof(appearance));
21.   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_PPCP, ATT_PROPERT
Y_READ, conn_param, sizeof(conn_param));
22.
23.   // Add GATT service and characteristics
24.   ble.addService(BLE_UUID_GATT);
25.   ble.addCharacteristic(BLE_UUID_GATT_CHARACTERISTIC_SERVICE_CHANGED,
 ATT_PROPERTY_INDICATE, change, sizeof(change));
26.
27.   // Add primary service1.
28.   ble.addService(service1_uuid);
29.   // Add characteristic to service1, return value handle of
characteristic.
30.   character1_handle = ble.addCharacteristicDynamic(char1_uuid, ATT_PR
OPERTY_READ|ATT_PROPERTY_WRITE, characteristic1_data, CHARACTERISTIC1_MAX
_LEN);
31.   character2_handle = ble.addCharacteristicDynamic(char2_uuid, ATT_PR
OPERTY_READ|ATT_PROPERTY_NOTIFY, characteristic2_data, CHARACTERISTIC2_MA
X_LEN);
32.
33.   // Set BLE advertising parameters
34.   ble.setAdvertisementParams(&adv_params);
35.
36.   // Set BLE advertising and scan respond data
37.   ble.setAdvertisementData(sizeof(adv_data), adv_data);
38.   ble.setScanResponseData(sizeof(scan_response), scan_response);
39.
40.   // Start advertising.
41.   ble.startAdvertising();
42.   Serial.println("BLE peripheral start advertising");
43. }
```

FIGURA 98: CÓDIGO FUNCIÓN SETUPL() DISPOSITIVO DUO PERIPHERAL

Inicialmente se inicializan las comunicaciones seriales, tanto la correspondiente a los pines TX/RX como al canal de comunicación a través del puerto USB, para la conexión con el ordenador, haciendo uso de los métodos `Serial1.begin()` y `Serial.begin()`. El *baudrate* elegido para el puerto `Serial1` es de 9600 y, el modo de operación por defecto es de 8 bits de datos, sin paridad y 1 bit de stop. Así, la tasa de baudios seleccionada para la comunicación USB es de 115200. En la Figura 99 se muestran las líneas de código referentes a la inicialización de las comunicaciones seriales.

```
1.  Serial1.begin(9600);
2.  Serial.begin(115200);
3.  delay(5000);
4.  Serial.println("RedBear DUO BLE Peripheral");
```

FIGURA 99: INICIALIZACIÓN COMUNICACIONES SERIALES DISPOSITIVO DUO PERIPHERAL

Las líneas de código correspondientes a la inicialización de la conectividad BLE se encuentra en la Figura 100. El método `ble.init()` habilita la interfaz HCI entre el *Host* y el controlador, además de inicializar el estado predeterminado del controlador. Crea un *thread* para gestionar los comandos y eventos HCI. Debe ser llamado antes que a cualquier otro método de los perfiles GAP y GATT. Así, los métodos recogidos en las líneas 5-8 de la Figura 100, registran las devoluciones de llamada que deben ser ejecutadas en los procesos de conexión y desconexión, así como en las solicitudes de lectura y escritura de las características.

```
1.  // Initialize ble_stack.
2.  ble.init();
3.
4.  // Register BLE callback functions.
5.  ble.onConnectedCallback(deviceConnectedCallback);
6.  ble.onDisconnectedCallback(deviceDisconnectedCallback);
7.  ble.onDataReadCallback(gattReadCallback);
8.  ble.onDataWriteCallback(gattWriteCallback);
```

FIGURA 100: INICIALIZACIÓN COMUNICACIÓN BLE DISPOSITIVO DUO PERIPHERAL

La creación de servicios y características correspondientes a la configuración de los servidores GAP y GATT se muestra en la Figura 101. Para la creación de servicios se usa el método `ble.addService()`, introduciendo por parámetro el UUID del servicio. Las características creadas a continuación pertenecen por defecto a este servicio, hasta que se cree uno nuevo. Así, el método `ble.addCharacteristic()` se encarga de crear las características de los perfiles GAP y GATT, introduciéndole cuatro parámetros:

- ***UUID***: puede ser un UUID de 16 bits o de 128 bits.

- ***flags***: indica las propiedades de la característica. Para las características GAP y GATT creadas, solo se ha hecho uso de los *flags* correspondientes a las propiedades de lectura y escritura, siendo estos `ATT_PROPERTY_REAAD` y `ATT_PROPERTY_WRITE`.

- ***data***: puntero de tipo *uint8_t* al valor de la característica.

- ***data_len***: longitud máxima de la característica.

```
1.   // Add GAP service and characteristics
2.   ble.addService(BLE_UUID_GAP);
3.   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_DEVICE_NAME, ATT_P
ROPERTY_READ|ATT_PROPERTY_WRITE, (uint8_t*)BLE_DEVICE_NAME, sizeof(BLE_DE
VICE_NAME));
4.   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_APPEARANCE, ATT_PR
OPERTY_READ, appearance, sizeof(appearance));
5.   ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_PPCP, ATT_PROPERTY
_READ, conn_param, sizeof(conn_param));
6.
7.   // Add GATT service and characteristics
8.   ble.addService(BLE_UUID_GATT);
9.   ble.addCharacteristic(BLE_UUID_GATT_CHARACTERISTIC_SERVICE_CHANGED,
ATT_PROPERTY_INDICATE, change, sizeof(change));
```

FIGURA 101: INICIALIZACIÓN PERFILES GAP Y GATT DISPOSITIVO DUO PERIPHERAL

En la Figura 102 se muestra el fragmento de código en el que se configuran el servicio y las características con las que se va a trabajar en la solución. Para su creación, se emplean los mismos métodos que en los casos anteriores. La característica *chr1*, utilizada por el dispositivo *Central* final para escribir los datos que desea enviar al otro extremo de la pasarela, contará con las propiedades de lectura y escritura. Así, la característica *chr2*, necesaria para notificar al dispositivo *Central* final las respuestas enviadas por el dispositivo servidor BLE en el lado opuesto de la pasarela, tendrá las propiedades de notificación y lectura.

```
1.   // Add primary service1.
2.   ble.addService(service1_uuid);
3.   // Add characteristic to service1, return value handle of charact.
4.   character1_handle = ble.addCharacteristicDynamic(char1_uuid, ATT_PRO
PERTY_READ|ATT_PROPERTY_WRITE, characteristic1_data, CHARACTERISTIC1_MAX_
LEN);
5.   character2_handle = ble.addCharacteristicDynamic(char2_uuid, ATT_PRO
PERTY_READ|ATT_PROPERTY_NOTIFY, characteristic2_data, CHARACTERISTIC2_MAX
_LEN);
```

FIGURA 102: CREACIÓN CHAR1 Y CHAR2 EN DISPOSITIVO DUO PERIPHERAL

Una vez definidos los perfiles GAP y GATT, así como la creación de servicios y características, se pasa a inicializar los parámetros del proceso de *advertising*. En la Figura 103 aparecen todas las funciones correspondientes al proceso de *advertising*.

```
1.    // Set BLE advertising parameters
2.    ble.setAdvertisementParams(&adv_params);
3.
4.    // Set BLE advertising and scan respond data
5.    ble.setAdvertisementData(sizeof(adv_data), adv_data);
6.    ble.setScanResponseData(sizeof(scan_response), scan_response);
7.
8.    // Start advertising.
9.    ble.startAdvertising();
10.      Serial.println("BLE peripheral start advertising");
```

FIGURA 103: INICIALIZACIÓN PROCESO DE ADVERTISING DISPOSITIVO DUO PERIPHERAL

El método `ble.setAdvertisementParams()` establece el comportamiento durante el proceso de *advertising* del dispositivo *DUO Peripheral*, indicándole por parámetro un objeto de tipo `advParams_t`, definido previamente, en el que se incluye la configuración de todos sus parámetros. Análogamente, el método `ble.setAdvertisementData()` establece los datos incluidos en los paquetes de *advertising*. Estos datos son esenciales en el proceso de *advertising*, y están limitados a 31 bytes, según el formato definido en la especificación de *Bluetooth*. Así, el método `ble.setScanResponseData` define los datos de respuesta a un proceso de *scanning*. Los datos de respuesta son opcionales y están limitados a 31 bytes. Por último, mediante el método `ble.startAdvertising()`, se inicia el proceso de *advertising* para que el dispositivo *DUO Peripheral* sea detectable por los dispositivos *Central* en proceso de *scanning*.

La última función que compone el código para la implementación de un dispositivo BLE *Peripheral* en el dispositivo *DUO* es el bucle principal `loop()`, el cual en este caso particular se encuentra vacío, como se puede apreciar en la Figura 104.

```
1. void loop() {
2. }
```

FIGURA 104: FUNCIÓN LOOP() DISPOSITIVO DUO PERIPHERAL

6.3.3. REDBEAR DUO CENTRAL

El *firmware* de este dispositivo se encarga de implementar en el dispositivo *RedBear DUO* el comportamiento de un dispositivo BLE *Central*, así como de habilitar la comunicación serial a través de la UART con un dispositivo *LoPy*.

6.3.3.1. Constantes

En la Figura 105 se muestra el código correspondiente a la definición de las constantes utilizadas para caracterizar los parámetros relacionados con el proceso de *scanning*. En esta sección de código se han definido el tipo de escaneo, el intervalo de escaneo y la ventana de escaneo. Los valores que pueden tomar los parámetros en cada caso son:

- ***BLE_SCAN_TYPE***: cuando se trate de escaneo pasivo, es decir, que no se envían paquetes de solicitud de *scanning*, se asigna el valor 0x00, siendo esta la opción por defecto. Para el escaneo activo, en el cual se envían paquetes de solicitud de *scanning*, el valor que debe tomar este parámetro es 0x01.

- ***BLE_SCAN_INTERVAL***: se define como el intervalo de tiempo desde que el controlador inició el último proceso de *scanning* hasta que inicia el siguiente. Se calcula multiplicando el valor asignado por 0,625 ms. El rango de valores que puede tomar va desde 0x0004 hasta 0x4000.

- ***BLE_SCAN_WINDOW***: indica la duración del proceso de *scanning*. La ventana de escaneo debe ser menor o igual que el intervalo de *scanning*.

```
1. #define BLE_SCAN_TYPE         0x00   // Passive scanning
2. #define BLE_SCAN_INTERVAL     0x0060 // 60 ms
3. #define BLE_SCAN_WINDOW       0x0030 // 30 ms
```

FIGURA 105: DEFINICIÓN CONSTANTES SCANNING DISPOSITIVO DUO CENTRAL

El lenguaje de programación C provee una palabra clave llamada *typedef*, utilizada para asignar nombres alternativos a tipos de datos existentes. Se usa principalmente con tipos de datos definidos por el usuario, cuando los nombres de los tipos de datos se vuelven un poco complicados de usar en el programa. Así, en la Figura 106 se encuentra la definición de la estructura que conforma los objetos de tipo `Device_t`, con su servicio, características y descriptores.

```
1.  typedef struct {
2.    uint16_t  connected_handle;
3.    uint8_t   addr_type;
4.    bd_addr_t addr;
5.    struct {
6.      gatt_client_service_t service;
7.      struct {
8.        gatt_client_characteristic_t chars;
9.        gatt_client_characteristic_descriptor_t descriptor[2]; //
User_descriptor and client charactersitc configuration descriptor.
10.     } chars[2];
11.   } service; // Service contains two characteristics
12. } Device_t;
```

FIGURA 106: DEFINICIÓN ESTRUCTURA DISPOSITIVO DUO CENTRAL

Por último, en la Figura 107 se define el intervalo máximo y mínimo de tiempo de conexión, la latencia admisible en la respuesta por parte del dispositivo *Slave* y el *timeout* establecido para la supervisión de la conexión.

```
1.  #define MIN_CONN_INTERVAL          0x0008 // 10ms.
2.  #define MAX_CONN_INTERVAL          0x0009 // .
3.  #define SLAVE_LATENCY              0x0000 // No slave latency.
4.  #define CONN_SUPERVISION_TIMEOUT   0x03E8 // 10s.
```

FIGURA 107: DEFINICIÓN CONSTANTES DE CONEXIÓN DISPOSITIVO DUO CENTRAL

6.3.3.2 Variables

En este apartado se recoge la declaración de las variables globales empleadas en el código. En la Figura 108 se muestra la definición de la variable `device`, de tipo `Device_t`, así como las variables utilizadas para la gestión de las funciones, posteriormente descritas. También se declara el *handle* de conexión, llamado `connected_id`, inicializado al valor de desconexión, y el UUID del servicio a descubrir en el dispositivo *Bluz DK*.

```
1.  Device_t device;
2.  uint8_t  chars_index = 0;
3.  uint8_t  desc_index = 0;
4.  uint8_t  write_index = 0;
5.  static uint8_t gatt_notify_flag = 0;
6.  unsigned long lastmills = 0;
7.  // Connect handle.
8.  static uint16_t connected_id = 0xFFFF;
9.  // The service uuid to be discovered.
10. static uint8_t service1_uuid[16] = { 0x87,0x1e,0x02,0x23,0x38,0xff,0x
77,0xb1,0xed,0x41,0x9f,0xb3,0xaa,0x14,0x2d,0xb2 };
```

FIGURA 108: DEFINICIÓN VARIABLES DISPOSITIVO DUO CENTRAL

6.3.3.3. Funciones

En este apartado se procede a analizar las funciones incluidas en el código para la implementación de la conectividad BLE y la configuración del comportamiento de la UART ante la recepción y envío de paquetes, con el objetivo de establecer la comunicación deseada entre los dispositivos utilizados. En la Figura 109 se indica el fragmento de código correspondiente a la función `ble_advdata_decode()`, encargada de encontrar el tipo de datos indicado dentro de los paquetes de *advertising*, es decir, decodificar los paquetes de *advetising* recibidos. El parámetro `type` indica el tipo de datos que incluye, mientras que la longitud de los datos de *advertising* viene determinada en el parámetro `advdata_len`. Así, el parámetro `*p_advdata` es un puntero a los datos de *advertisng*, cuyo tamaño viene determinado por el valor al que apunta el puntero `*len`. Por último, el parámetro `*p_field_data` apunta al *buffer* en el que se almacenarán los datos de campo.

```
1.  uint32_t ble_advdata_decode(uint8_t type, uint8_t advdata_len, uint8_
t *p_advdata, uint8_t *len, uint8_t *p_field_data) {
2.    uint8_t index = 0;
3.    uint8_t field_length, field_type;
4.
5.    while (index < advdata_len) {
6.      field_length = p_advdata[index];
7.      field_type = p_advdata[index + 1];
8.      Serial.print("       - AVD/SR data decoding -> ad_type: ");
9.      Serial.print(field_type, HEX);
10.     Serial.print(", length: ");
11.     Serial.println(field_length, HEX);
12.     if (field_type == type) {
13.       memcpy(p_field_data, &p_advdata[index + 2], (field_length - 1))
;
14.       *len = field_length - 1;
15.       return 0;
16.     }
17.     index += field_length + 1;
18.   }
19.   return 1;
20. }
```

FIGURA 109: CÓDIGO FUNCIÓN BLE_ADVDATA_DECODE() DISPOSITIVO DUO CENTRAL

En la Figura 110 se recoge la primera sección de código correspondiente a la función `reportCallback()`, *callback* ejecutada cuando se recibe un nuevo paquete de *advertising* o un paquete de respuesta al escaneo por parte del dispositivo par. La devolución de llamada toma por parámetro un objeto de tipo `advertisementReport_t`, y no devuelve nada. El objeto `advertisementReport_t` contiene información sobre el paquete recibido. En este primer

fragmento de código de la función, se imprime por pantalla la información contenida en el objeto `advertisementReport_t`.

```
1.  void reportCallback(advertisementReport_t *report) {
2.    uint8_t index;
3.    Serial.println("");
4.    Serial.println("* BLE scan callback: ");
5.    Serial.print("   - Advertising event type: ");
6.    Serial.println(report->advEventType, HEX);
7.    Serial.print("   - Peer device address type: ");
8.    Serial.println(report->peerAddrType, HEX);
9.    Serial.print("   - Peer device address: ");
10.   for (index = 0; index < 6; index++) {
11.     Serial.print(report->peerAddr[index], HEX);
12.     Serial.print(" ");
13.   }
14.   Serial.println(" ");
15.
16.   Serial.print("   - RSSI: ");
17.   Serial.println(report->rssi, DEC);
18.
19.   Serial.print("   - Advertising/Scan response data packet: ");
20.   for (index = 0; index < report->advDataLen; index++) {
21.     Serial.print(report->advData[index], HEX);
22.     Serial.print(" ");
23.   }
24.   Serial.println(" ");
25.
26.   uint8_t len;
27.   uint8_t adv_name[31];
```

FIGURA 110: CÓDIGO FUNCIÓN REPORTCALLBACK() DISPOSITIVO DUO CENTRAL (I)

Así, en el código de la Figura 111 se obtiene el nombre del dispositivo que envía el paquete de *advertising* o la respuesta al proceso de *scanning* y, en caso de coincidir éste con el del dispositivo *Bluz DK*, se detiene el proceso de *scanning* haciendo uso del método `ble.stopScanning()`, y se inicia la conexión con dicho dispositivo, mediante el uso del método `ble.connect()`. También se copian los parámetros `addr_typ` y `addr`, indicados en el paquete recibido, a la variable global `device`.

El método `ble.stopScanning()` detiene el escaneo de dispositivos BLE alrededor. Por otro lado, el método `ble.connect()` inicia el establecimiento de conexión con un dispositivo par, siendo en este caso el dispositivo *Bluz DK*. Este método toma dos parámetros: la dirección del dispositivo par, `report->peerAddr`, y el tipo de dirección de éste, `{BD_ADDR_TYPE_LE_RANDOM}`, asignada de forma aleatoria. El valor del primer parámetro se ha obtenido analizando el paquete de *advertising* del dispositivo par.

```
1.    if (0x00 == ble_advdata_decode(0x09, report->advDataLen, report-
>advData, &len, adv_name)) {
2.      Serial.println("");
3.      Serial.print("   - Complete Local Name: ");
4.      Serial.println((const char *)adv_name);
5.
6.      if (0x00 == memcmp(adv_name, "Bluz DK", min(7, len))) {
7.        Serial.println("____ Found BluzDK");
8.        Serial.println("");
9.        ble.stopScanning();
10.       device.addr_type = report->peerAddrType;
11.       memcpy(device.addr, report->peerAddr, 6);
12.
13.       ble.connect(report->peerAddr, {BD_ADDR_TYPE_LE_RANDOM});
14.     }
15.   }
16. }
```

FIGURA 111: CÓDIGO FUNCIÓN REPORTCALLBACK() DISPOSITIVO DUO CENTRAL (II)

Para la devolución de llamada correspondiente al establecimiento de conexión con un dispositivo BLE *Peripheral*, se ha definido la función `deviceConnectedCallback()`, recogida en la Figura 112. Se ejecutará cuando se produzca una conexión con un dispositivo BLE *Peripheral*.

La función toma dos parámetros y no devuelve nada. Uno de los parámetros, `BLEStatus_t`, refleja el estado de la conexión, y puede tomar cualquier de los siguientes valores: `BLE_STATUS_CONNECTION_ERROR` o `BLE_STATUS_OK`. El siguiente parámetro es un *handle* de tipo *uint16_t* asignado si el establecimiento de conexión es correcto. El *handle* tomará el valor 0xFFFF en caso de que no sea válida la conexión. En caso de que el parámetro `status` tenga valor `BLE_STATUS_OK`, se actualiza el valor de la variable global `connected_id` con el del parámetro *handle*, mostrando a su vez por pantalla mensajes que indican que la conexión se ha realizado de forma satisfactoria y presentando el nuevo valor del *handle*. Además, se procede a descubrir los servicios primarios del dispositivo *Peripheral* final mediante el método `ble.discoverPrimaryServices()`.

El método `ble.discoverPrimaryServices()` descubre los servicios primarios en el servidor GATT del dispositivo par. Solo se pueden descubrir sus servicios primarios una vez que quede establecida la conexión. Se debe pasar como parámetro el *handle* de conexión. Este método devuelve un valor *uint8_t* que indica el resultado de la operación de *Discovery*: 0 en caso de que se realice satisfactoriamente.

```
1.  void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
2.    switch (status) {
3.      case BLE_STATUS_OK:
4.        Serial.println("");
5.        Serial.println("_____ Device connected");
6.        // Connect to remote device, start to discover service.
7.        connected_id = handle;
8.        device.connected_handle = handle;
9.        Serial.print("         - Device connected handle: ");
10.       Serial.println(connected_id);
11.       // Start to discover service, will report result on
discoveredServiceCallback.
12.       ble.discoverPrimaryServices(handle);
13.       break;
14.     default: break;
15.   }
16. }
```

FIGURA 112: CÓDIGO FUNCIÓN DEVICECONNECTEDCALLBACK() DISPOSITIVO DUO CENTRAL

Análogamente, en la Figura 113 se incluye el código correspondiente a la devolución de llamada ejecutada cuando se produce la desconexión de un dispositivo. Esta función toma un parámetro de tipo *uint16_t*, que indica el estado de la conexión, y no devuelve nada. La función se limita a cambiar la variable `connected_id` al valor de desconexión. Al producirse la desconexión con el dispositivo *Peripheral* final, se reanuda el proceso de *scanning* haciendo uso del método `ble.startScanning()`.

```
1.  void deviceDisconnectedCallback(uint16_t handle){
2.    Serial.println("");
3.    Serial.println("_____ Device disconnected");
4.    Serial.print("         - Device disconnected handle: ");
5.    Serial.println(handle,HEX);
6.    if (connected_id == handle) {
7.      Serial.println("");
8.      Serial.println("_____ BLE Central restart scanning!");
9.      // Disconnect from remote device, restart to scanning.
10.     connected_id = 0xFFFF;
11.     ble.startScanning();
12.   }
13. }
```

FIGURA 113: CÓDIGO FUNCIÓN DEVICEDISCONNECTEDCALLBACK() DISPOSITIVO DUO CENTRAL

Cuando se descubre un nuevo servicio, se ejecuta la función `discoveredServiceCallback()`, recogida en la Figura 114. Esta función *callback* toma tres parámetros y no devuelve nada. El parámetro `status`, de tipo `BLEStatus_t`, indica el estado de la operación, `conn_handle` el *handle* de conexión y el parámetro `service` indica el servicio descubierto, de tipo `gatt_client_service_t`. Cuando la variable `status` tome el valor

`BLE_STATUS_OK`, significará que se ha descubierto un servicio, pasando a mostrar por pantalla sus propiedades. En caso de que el UUID del servicio descubierto coincida con el valor de la variable global `service1_uuid`, indicará que se ha descubierto el servicio definido por el usuario, asignándolo a la variable `device`. Así, cuando la variable `status` tome el valor `BLE_STATUS_DONE`, se habrá completado el descubrimiento de los servicios. A continuación, se pasa a descubrir las características propias de los servicios ya descubiertos, a través del método `ble.discoverCharacteristics()`.

```
1.  static void discoveredServiceCallback(BLEStatus_t
status, uint16_t con_handle, gatt_client_service_t *service) {
2.    uint8_t index;
3.    if (status == BLE_STATUS_OK) {    // Found a service.
4.      Serial.println(" ");
5.      Serial.println("* Service found successfully");
6.      Serial.print("    - Service start handle: ");
7.      Serial.println(service->start_group_handle, HEX);
8.      Serial.print("    - Service end handle: ");
9.      Serial.println(service->end_group_handle, HEX);
10.     Serial.print("    - Service uuid16: ");
11.     Serial.println(service->uuid16, HEX);
12.     Serial.print("    - Service uuid128 : ");
13.     for (index = 0; index < 16; index++) {
14.       Serial.print(service->uuid128[index], HEX);
15.       Serial.print(" ");
16.     }
17.     Serial.println(" ");
18.
19.     if (0x00 == memcmp(service->uuid128, service1_uuid, 16)) {
20.       Serial.println("    - User defined Service uuid128 found
successfully");
21.       device.service.service = *service;
22.     }
23.   }
24.   else if (status == BLE_STATUS_DONE) {
25.    Serial.println(" ");
26.    Serial.println("* Discover all services completed");
27.     // All sevice have been found, start to discover characteristics.
28.     // Result will be reported on discoveredCharsCallback.
29.     ble.discoverCharacteristics(device.connected_handle, &device.serv
ice.service);
30.   }
31. }
```

FIGURA 114: CÓDIGO FUNCIÓN DISCOVEREDSERVICECALLBACK() DISPOSITIVO DUO CENTRAL

Para gestionar el descubrimiento de las diferentes características del dispositivo *Peripheral* final, se hace uso del método `discoveredCharsCallback()`, que se ejecutará mediante devolución de llamada cada vez que se descubra una característica. Este método toma tres parámetros y no devuelve nada:

- ***status***: estado de la operación, de tipo `BLEStatus_t`.
- ***conn_handle***: *handle* de conexión, de tipo *uint16_t*.
- ***characteristic***: característica descubierta, de tipo `gatt_client_characteristic_t`.

Del mismo modo que con la función `discoveredServiceCallback()`, cuando el valor de la variable `status` sea `BLE_STATUS_OK`, indicará que se ha descubierto una característica, imprimiendo por pantalla sus propiedades, y asignándolas a la variable global `device`. La primera sección del código correspondiente a esta función se encuentra en la Figura 115.

```
1.  static void discoveredCharsCallback(BLEStatus_t
status, uint16_t con_handle, gatt_client_characteristic_t *characteristic
) {
2.    uint8_t index;
3.    if (status == BLE_STATUS_OK) {   // Found a characteristic.
4.      Serial.print("* Characteristic found successfully ");
5.      Serial.print(chars_index, HEX);
6.      Serial.println(" :");
7.      Serial.print("   - Characteristic start handle: ");
8.      Serial.println(characteristic->start_handle, HEX);
9.      Serial.print("   - Characteristic end handle: ");
10.     Serial.println(characteristic->end_handle, HEX);
11.     Serial.print("   - Characteristic value handle: ");
12.     Serial.println(characteristic->value_handle, HEX);
13.     Serial.print("   - Characteristic properties: ");
14.     Serial.println(characteristic->properties, HEX);
15.     Serial.print("   - Characteristic uuid16: ");
16.     Serial.println(characteristic->uuid16, HEX);
17.     Serial.print("   - Characteristic uuid128 : ");
18.     for (index = 0; index < 16; index++) {
19.       Serial.print(characteristic->uuid128[index], HEX);
20.     }
21.     Serial.println(" ");
22.     if (chars_index < 2) {
23.       device.service.chars[chars_index].chars= *characteristic;
24.       chars_index++;
25.     }
26.   }
27.   else if (status == BLE_STATUS_DONE) {
28.     Serial.println(" ");
29.     Serial.println("* Discover all characteristics completed");
30.     chars_index = 0;
31.     ble.discoverCharacteristicDescriptors(device.connected_handle, &d
evice.service.chars[chars_index].chars);
32.   }
33. }
```

FIGURA 115: CÓDIGO FUNCIÓN DISCOVEREDCHARSCALLBACK() DISPOSITIVO DUO CENTRAL

Cuando la variable `status` tome el valor `BLE_STATUS_DONE`, estarán descubiertas todas las características, procediendo entonces a descubrir los descriptores de las mismos, mediante el

método `ble.discoverCharacteristicDescriptors`. Así, la función `discoveredCharsDescriptorsCallback()`, mostrada en la Figura 116, representa la devolución de llamada correspondiente al descubrimiento de los descriptores, siguiendo el mismo formato que con el descubrimiento de los servicios y las características del dispositivo *Peripheral final.* En esta función también se realiza la suscripción a la característica denominada *p_chr2* en la descripción de la solución, quedando habilitadas las notificaciones.

```
1.  static void discoveredCharsDescriptorsCallback(BLEStatus_t
status, uint16_t con_handle, gatt_client_characteristic_descriptor_t *des
criptor) {
2.    uint8_t index;
3.    if (status == BLE_STATUS_OK) {    // Found a descriptor.
4.      Serial.print("* Descriptor found successfully ");
5.      Serial.print(desc_index, HEX);
6.      Serial.print(" - Characteristic ");
7.      Serial.print(chars_index, HEX);
8.      Serial.println(" :");
9.      Serial.print("    - Descriptor handle: ");
10.     Serial.println(descriptor->handle, HEX);
11.     Serial.print("    - Descriptor uuid16: ");
12.     Serial.println(descriptor->uuid16, HEX);
13.     Serial.print("    - Descriptor uuid128 : ");
14.     for (index = 0; index < 16; index++) {
15.       Serial.print(descriptor->uuid128[index], HEX);
16.     }
17.     Serial.println(" ");
18.     if (desc_index < 2) {
19.       device.service.chars[chars_index].descriptor[desc_index++] = *d
escriptor;
20.     }
21.   }
22.   else if (status == BLE_STATUS_DONE) {
23.     Serial.println("* Discover all descriptors completed");
24.     chars_index++;
25.     if (chars_index < 2) {
26.       desc_index=0; // !!!
27.       ble.discoverCharacteristicDescriptors(device.connected_handle,
&device.service.chars[chars_index].chars);
28.     }
29.     else {
30.       chars_index = 0;
31.       desc_index = 0;
32.       Serial.println("* Write CCCD:");
33.       Serial.print("    - Connection handle: ");
34.       Serial.println(device.connected_handle, HEX);
35.       Serial.print("RETURN CODE: ");
36.     // Enable notify.
37.       Serial.println(ble.writeClientCharsConfigDescriptor(device.conn
ected_handle, &device.service.chars[0].chars, GATT_CLIENT_CHARACTERISTICS
_CONFIGURATION_NOTIFICATION), HEX);
38.     }
39.   }
40. }
```

FIGURA 116: CÓDIGO FUNCIÓN CÓDIGO FUNCIÓN DISCOVEREDCHARSDESCRIPTORSCALLBACK() DISPOSITIVO DUO CENTRAL

La función `gattReadCallback()`, cuyo código se muestra en la Figura 117, es una devolución de llamada correspondiente a las solicitudes de lectura en las características BLE del dispositivo *Peripheral* final. Esta función contiene cinco parámetros, descritos a continuación, y no devuelve nada:

- ***status***: estado de la operación, de tipo `BLEStatus_t`.
- ***conn_handle***: *handle* de conexión, de tipo *uint16_t*.
- ***value_handle***: *handle* del valor del atributo de la característica, de tipo *uint16_t*.
- ***value***: puntero de tipo *uint8_t* al *buffer* que contiene los datos de lectura.
- ***length***: valor de tipo *uint16_t* que indica la longitud de los datos de lectura.

```
1.  void gattReadCallback(BLEStatus_t
status, uint16_t con_handle, uint16_t value_handle, uint8_t *value, uint1
6_t length) {
2.    uint8_t index;
3.    if (status == BLE_STATUS_OK) {
4.      Serial.println(" ");
5.      Serial.println("* Read characteristic value successfully:");
6.      Serial.print("   - Connection handle: ");
7.      Serial.println(con_handle, HEX);
8.      Serial.print("   - Characteristic value attribute handle: ");
9.      Serial.println(value_handle, HEX);
10.     Serial.print("   - Characteristic value : ");
11.     for (index = 0; index < length; index++) {
12.       Serial.print(value[index], HEX);
13.       Serial.print(" ");
14.     }
15.     Serial.println("");
16.   }
17.   else if (status != BLE_STATUS_DONE) {
18.     Serial.println(" ");
19.     Serial.println("! Read characteristic value FAILED");
20.     Serial.println(" ");
21.     }
22. }
```

FIGURA 117: CÓDIGO FUNCIÓN GATTREADCALLBACK()

Por otro lado, la función `gattWritenCallback()`, recogida en la Figura 118, se ejecuta cuando se produzca un evento de escritura en alguna característica del dispositivo *Peripheral* final. Esta función toma dos parámetros: `status` y `conn_handle`, y no devuelve nada.

```
1.  void gattWrittenCallback(BLEStatus_t status, uint16_t con_handle) {
2.    if (status == BLE_STATUS_DONE) {
3.      Serial.println(" ");
4.      Serial.println("* Write characteristic value done:");
5.      Serial.print("   - Connection handle: ");
6.      Serial.println(con_handle, HEX);
7.    }
8.    else {
9.      Serial.println(" ");
10.     Serial.println("! Write characteristic value FAILED");
11.     Serial.println(" ");
12.   }
13. }
```

FIGURA 118: CÓDIGO FUNCIÓN GATTWRITTENCALLBACK() DISPOSITIVO DUO CENTRAL

La gestión de las devoluciones de llamada asociadas a las solicitudes de lectura en los descriptores propios del dispositivo *Peripheral* final se lleva a cabo en la función `gattReadDescriptorCallback()`, mostrada en la Figura 119, que toma los mismos cinco parámetros que la función `gattReadCallback()`, haciendo en este caso referencia a los descriptores.

```
1.  void gattReadDescriptorCallback(BLEStatus_t
status, uint16_t con_handle, uint16_t value_handle, uint8_t *value, uint1
6_t length) {
2.    uint8_t index;
3.    if(status == BLE_STATUS_OK) {
4.      Serial.println(" ");
5.      Serial.println("* Read descriptor value successfully:");
6.      Serial.print("   - Connection handle: ");
7.      Serial.println(con_handle, HEX);
8.      Serial.print("   - Descriptor value attribute handle: ");
9.      Serial.println(value_handle, HEX);
10.     Serial.print("   - Descriptor value : ");
11.     for (index = 0; index < length; index++) {
12.       Serial.print(value[index], HEX);
13.       Serial.print(" ");
14.     }
15.     Serial.println(" ");
16.   }
17.   else if (status == BLE_STATUS_DONE) {
18.
19.   }
20. }
```

FIGURA 119: CÓDIGO FUNCIÓN READDESCRIPTORCALLBACK() DISPOSITIVO DUO CENTRAL

La devolución de llamada correspondiente a las operaciones de escritura en el *Client Characteristic Configuration Descriptor* (CCCD) viene definida en la función

`gattWriteCCCDCallback()`, cuyo código se encuentra en la Figura 120. El descriptor CCCD define cómo la característica puede ser configurada por un cliente específico.

```
1.  void gattWriteCCCDCallback(BLEStatus_t status, uint16_t con_handle) {
2.    Serial.print("D --- gattWriteCCCDCallback (");
3.    Serial.print(status, HEX);
4.    if (status == BLE_STATUS_DONE) {
5.      Serial.println("* Write CCCD value successfully");
6.      Serial.print("   - Connection handle: ");
7.      Serial.println(con_handle, HEX);
8.      Serial.print("   - CCCD value: ");
9.      Serial.println(GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICA
TION, HEX);
10.     ble.readDescriptorValue(device.connected_handle, device.service.c
hars[0].descriptor[0].handle);
11.   }
12.   else {
13.     Serial.println("! Write CCCD value FAILED");
14.   }
15. }
```

FIGURA 120: CÓDIGO FUNCIÓN GATTWRITECCCDCALLBACK() DISPOSITIVO DUO CENTRAL

La función `gattNotifyUpdateCallback()`, cuyo código se muestra en la Figura 121, se ejecutará cuando el dispositivo *DUO Central* reciba una notificación por parte del dispositivo *Peripheral* final. Esta función toma los mismos cinco parámetros que la función `gattReadCallback()` y no devuelve nada. En el momento de recibir una notificación de la característica *p_chr2*, a la que se encuentra suscrito el dispositivo *DUO Central*, se imprime por pantalla el valor de los parámetros `con_handle`, `value_handle` y, además, se reenvía por puerto serie los datos de la notificación recibida hacia el otro extremo de la pasarela. Para la escritura en puerto serie se hace uso del método `Serial1.write()`, incluyendo por parámetro los datos a transmitir y su longitud.

```
1. void gattNotifyUpdateCallback(BLEStatus_t
status, uint16_t con_handle, uint16_t value_handle, uint8_t *value, uint1
6_t length) {
2.   uint8_t index;
3.   Serial.print("* Received new notification (");
4.   Serial.print("   - Connection handle: ");
5.   Serial.println(con_handle, HEX);
6.   Serial.print("   - Characteristic value attribute handle: ");
7.   Serial.println(value_handle, HEX);
8.   Serial.print("   - Notified value: ");
9.   for (index = 0; index < length; index++) {
10.     Serial.print(value[index], HEX);
11.   }
12.   Serial1.write(value,length);  // enviar notificación por Serial1
13. }
```

FIGURA 121: CÓDIGO FUNCIÓN GATTNOTIFYUPDATECALLBACK() DISPOSITIVO DUO CENTRAL

La función `serialEvent1()`, recogida en la Figura 122, se ejecutará cuando haya datos disponibles en el puerto *Serial1*, entre las llamadas al bucle de aplicación `loop()`. Dentro del código de la función se evalúa conjuntamente si existen datos en el *buffer* de recepción de la UART, mediante el método `Serial1.available()`, y si existe una conexión activa, cuando la variable `connected_id` tenga un valor diferente a 0xFFFF. En caso de que se cumplan ambas condiciones, se almacenan en la variable local `buf` los datos recibidos en el *buffer* de recepción. Finalmente, estos datos se escriben en la característica *p_chr1* del dispositivo *Peripheral* final, gracias al método `ble.writeValue()`.

```
1.  void serialEvent1(){
2.      if((Serial1.available())&& (connected_id != 0xFFFF)){
3.          delay(100);
4.          uint8_t ava = Serial1.available();
5.          uint8_t buf[15];
6.          for(uint8_t i=0; i<ava;i++){
7.              buf[i] = Serial1.read();
8.              Serial.print(buf[i], HEX);
9.              Serial.print(" ");
10.         }
11.         ble.writeValue(device.connected_handle, device.service.chars[
1].chars.value_handle, ava, buf);
12.         Serial.println(" ");
13.     }
14. }
```

FIGURA 122: CÓDIGO FUNCIÓN SERIALEVENT1() DISPOSITIVO DUO CENTRAL

La función `setup()`, ejecutada al inicio del programa, se recoge en la Figura 123, y es utilizada para definir las propiedades iniciales del entorno. Inicialmente se inicializan las comunicaciones seriales, tanto la correspondiente a los pines TX/RX como al canal de comunicación a través del puerto USB, para la conexión con el ordenador, haciendo uso de los métodos `Serial1.begin()` y `Serial.begin()`. El *baudrate* elegido para el puerto `Serial1` es de 9600 y, el modo de operación por defecto es de 8 bits de datos, sin paridad y 1 bit de stop. Así, la tasa de baudios seleccionada para la comunicación USB es de 115200.

El método `ble.init()` habilita la interfaz HCI entre el *Host* y el controlador, además de inicializar el estado predeterminado del controlador. Crea un *thread* para gestionar los comandos y eventos HCI. Debe ser llamado antes que a cualquier otro método de los perfiles GAP y GATT. Así, los métodos recogidos en las líneas 11-24, registran las devoluciones de llamada. Por último, se configuran los parámetros del proceso de *scanning* con el método `ble.setScanParams()`, y se inicia el proceso de *scanning* con el método `ble.startScanning()`.

```
1.  void setup() {
2.
3.    Serial1.begin(9600);
4.    Serial.begin(115200);
5.    delay(5000);
6.
7.    Serial.println("BLE bulb!");
8.    // Initialize ble_stack.
9.    ble.init();
10.
11.   // Register callback functions.
12.   ble.onConnectedCallback(deviceConnectedCallback);
13.   ble.onDisconnectedCallback(deviceDisconnectedCallback);
14.   ble.onScanReportCallback(reportCallback);
15.
16.   ble.onServiceDiscoveredCallback(discoveredServiceCallback);
17.   ble.onCharacteristicDiscoveredCallback(discoveredCharsCallback);
18.   ble.onDescriptorDiscoveredCallback(discoveredCharsDescriptorsCallba
ck);
19.   ble.onGattCharacteristicReadCallback(gattReadCallback);
20.   ble.onGattCharacteristicWrittenCallback(gattWrittenCallback);
21.   ble.onGattDescriptorReadCallback(gattReadDescriptorCallback);
22.
23.   ble.onGattWriteClientCharacteristicConfigCallback(gattWriteCCCDCall
back);
24.   ble.onGattNotifyUpdateCallback(gattNotifyUpdateCallback);
25.
26.   // Set scan parameters.
27.   ble.setScanParams(BLE_SCAN_TYPE, BLE_SCAN_INTERVAL, BLE_SCAN_WINDOW
);
28.
29.   // Start scanning.
30.   ble.startScanning();
31.   Serial.println("Start scanning! ");
32. }
```

FIGURA 123: CÓDIGO FUNCIÓN SETUP() DISPOSITIVO DUO CENTRAL

La última función que compone el código para la implementación de un dispositivo BLE *Central* en el dispositivo *DUO* es el bucle principal `loop()`, el cual en este caso particular se encuentra vacío, como se puede apreciar en la Figura 124.

```
1. void loop() {
2.
3. }
```

FIGURA 124: CÓDIGO FUNCIÓN LOOP() DISPOSITIVO DUO CENTRAL

6.4. Comprobación y validación

El proceso de comprobación del correcto funcionamiento de la plataforma HW/SW final desarrollada en este TFG, se basa en la verificación del intercambio de datos de forma bidireccional entre ambos extremos de la pasarela, es decir, entre los dispositivos BLE finales, comprobando la transferencia de los paquetes a través de todos los elementos de la pasarela.

Para ello, como dispositivo BLE *Central* final se ha empleado un *smartphone* ejecutando la aplicación *nRF Connect*, mientras que como dispositivo BLE *Peripheral* final se ha empleado el dispositivo *Bluz DK*. La planificación para la verificación funcional del sistema ha seguido la misma estructura que el desarrollo del *firmware*, comprobando de forma independiente cada extremo de la pasarela para finalmente validar el comportamiento del conjunto.

6.4.1. Comprobación del extremo de la pasarela correspondiente al dispositivo Central final

Una vez desarrollado el *firmware* del dispositivo *DUO Peripheral*, se verifica la aplicación en el propio entorno de desarrollo *Particle Build*, seleccionando la opción resaltada dentro del cuadrado rojo en la Figura 125, teniendo la aplicación abierta.

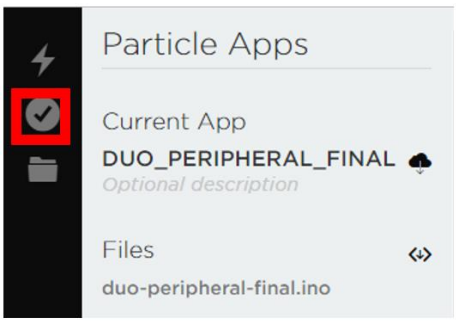

FIGURA 125: VERIFICAR APLICACIÓN EN ENTORNO DE DESARROLLO DE PARTICLE

En caso de realizarse satisfactoriamente la compilación del código, se mostrará un mensaje indicando que el *firmware* está listo (*Ready)* para ser cargado en memoria, además de mostrar el nivel de ocupación que tendrá en memoria *Flash* y memoria *RAM* el código desarrollado en el dispositivo *DUO*, como se puede observar en la Figura 126.

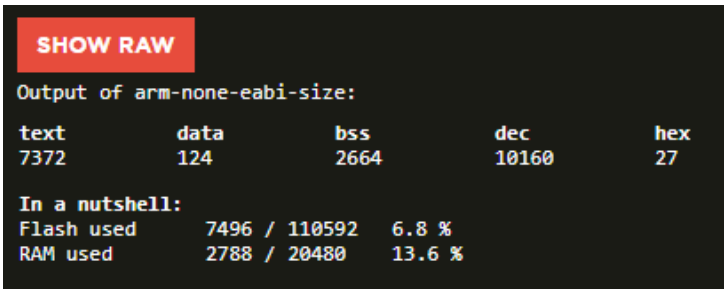

FIGURA 126: RESULTADOS VERIFICACIÓN CÓDIGO ENTORNO DE DESARROLLO DE PARTICLE EN DISPOSITIVO DUO PERIPHERAL

Una vez verificada la aplicación correctamente, se pasa a cargar el código en el dispositivo *DUO*, para implementar un dispositivo BLE *Peripheral*, clicando en el icono destacado en la Figura 127.

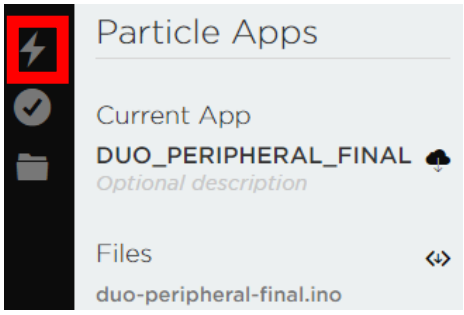

FIGURA 127: FLASHEAR APLICACIÓN EN ENTORNO DE DESARROLLO DE PARTICLE

Durante la programación OTA de la aplicación, el LED del dispositivo *DUO* parpadeará en color magenta, y cuando haya terminado de cargarse el código, la consola del IDE web mostrará el mensaje "`Flash successful! Please wait a moment while your device is updated...`". A continuación, el *DUO* se reiniciará para ejecutar la nueva aplicación, pasando el LED a parpadear en verde. En este punto, el dispositivo *DUO Peripheral* se encontrará completamente operativo. Así, en la Figura 128 se muestra la interfaz de la aplicación móvil *nRFConnect*, configurada para actuar como dispositivo BLE *Central*, donde se puede observar que al iniciar el proceso de *scanning*, éste detecta los paquetes de *advertising* enviados por el dispositivo *DUO Peripheral*, apareciendo como nombre del dispositivo "*DUO PER*". Además, en esta pantalla principal se muestra una breve descripción del dispositivo *DUO Peripheral*, mostrando su dirección MAC, la potencia de la señal recibida y los 128 bits del UUID del Servicio GAP.

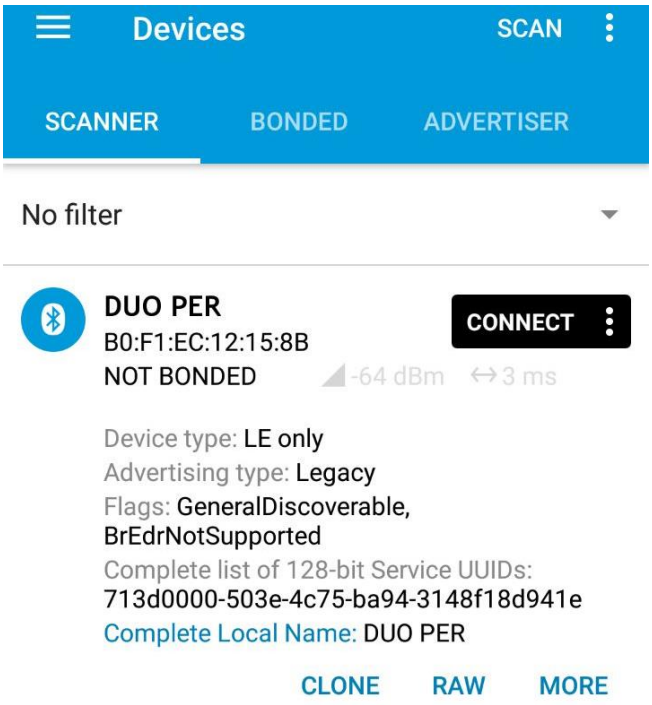

FIGURA 128: ESCANEO DISPOSITIVO DUO PERIPHERAL EN APLICACIÓN NRF CONNECT

Previa conexión, la aplicación permite ver las propiedades de los paquetes de *advertising*, mostrando en una gráfica la evolución en el tiempo del indicador de fuerza de la señal recibida (RSSI) y el intervalo de tiempo entre los paquetes de *advertising*, el cual se había fijado a 30 ms en el código y efectivamente se cumple, como se puede observar en la Figura 129.

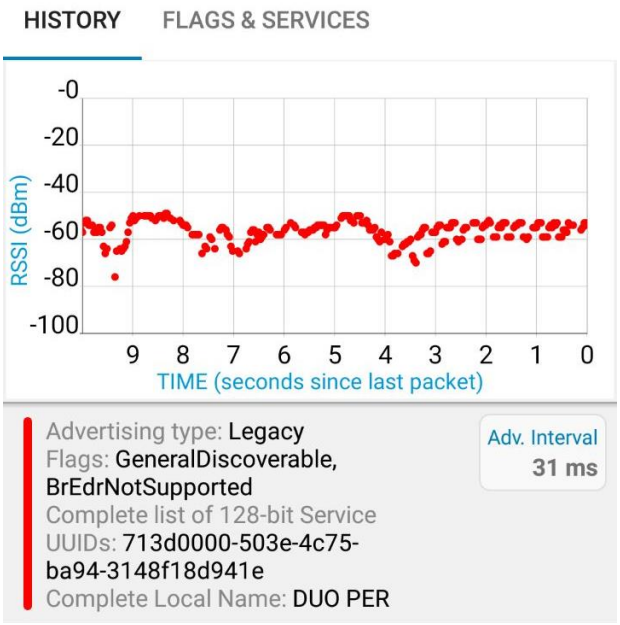

FIGURA 129: ANÁLISIS PAQUETES DE ADVERTISING DEL DISPOSITIVO DUO PERIPHERAL EN APLICACIÓN NRF CONNECT

Una vez establecida la conexión BLE entre el terminal móvil y el dispositivo *RedBear DUO*, se tiene acceso a sus diferentes servicios y características. Para la comunicación bidireccional con la pasarela se hace uso del servicio representado en la Figura 130. La característica mostrada en el rectángulo azul, con las propiedades de lectura y escritura, es la correspondiente a la descrita en la solución final como *chr1*, y es donde el dispositivo *Central* final escribirá los datos que desee enviar al dispositivo *Peripheral* final. Así, la característica mostrada en el rectángulo rojo corresponde a *chr2*, a la cual estará suscrito el dispositivo *Central* final, y por donde recibirá las respuestas en forma de notificación del dispositivo *Peripheral* final.

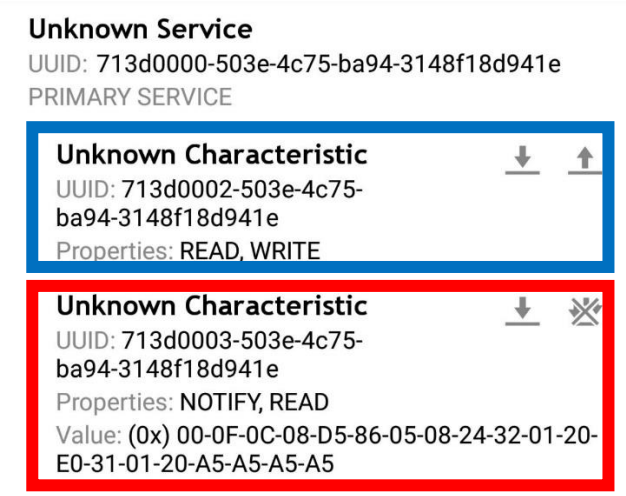

FIGURA 130: SERVICIO Y CARACTERÍSTICAS DEFINIDAS POR EL USUARIO VISTAS EN NRF CONNECT

Para analizar los eventos producidos en el dispositivo *DUO Peripheral* durante la comunicación, se dispone en el ordenador portátil del *software PuTTY*, que ofrece una interfaz de usuario para que, a través de un puerto serie virtual, se muestre en un terminal los mensajes escritos en el código para el seguimiento de los eventos. En concreto, para la conexión con el dispositivo *DUO Peripheral*, se ha establecido la configuración mostrada en la Figura 131.

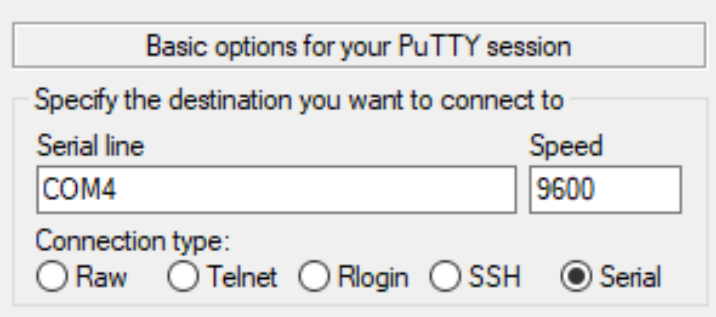

FIGURA 131: CONFIGURACIÓN PUTTY PARA DISPOSITIVO DUO PERIPHERAL

Finalmente, se implementa en el dispositivo *LoPy* el código correspondiente a esta solución del modo descrito en el primer caso, haciendo uso del IDE *Atom*, que ofrece su propia consola para la evaluación de los eventos en el dispositivo. Quedando este extremo de la pasarela configurado, se pasa a comprobar el envío de mensajes por parte del dispositivo *Central* final hacia el extremo opuesto de la pasarela. Dado que ambas soluciones desarrolladas en este TFG son equivalentes entre sí, pudiendo un lado de la pasarela implementar una solución, y el otro extremo la otra solución, para verificar esta solución final se implementa en el extremo de la pasarela correspondiente al dispositivo *Peripheral* final, la solución basada en el dispositivo *LoPy*, quedando el diagrama de bloques como en la Figura 132.

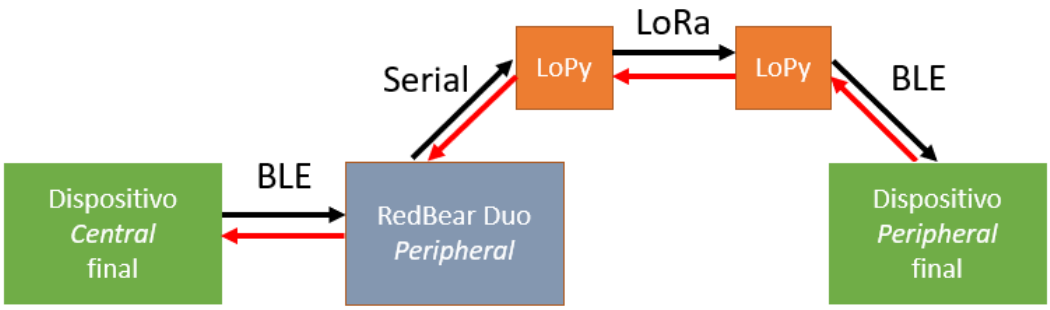

FIGURA 132: DIAGRAMA DE BLOQUES PARA LA COMPROBACIÓN DISPOSITIVO DUO PERIPHERAL

En primer lugar, se escribe en la característica *chr1* del dispositivo *DUO Peripheral* las cadenas de *bytes* `/x04/x6E` y `/x03/x04`, necesarias para el encendido del LED en el dispositivo *Bluz DK*, y así poder comprobar que el proceso de escritura se realiza correctamente, que posteriormente estos datos se retransmiten a través del puerto serial al dispositivo *LoPy*, y que finalmente se envían por la antena *LoRa*. En la Figura 133 aparecen los mensajes de inicio mostrados por el dispositivo *DUO Peripheral*, así como el mensaje que confirma la escritura correcta de los datos en la característica *chr1* y su reenvío por el pin TX.

```
RedBear DUO BLE Peripheral
BLE peripheral start advertising
-----------------------------------
Device connected!
  - Device connected handle: 64
-----------------------------------
Write value handler: 11
  - characteristic2 CCCD write value: 10
-----------------------------------
Write value handler: E
  - characteristic1 write value: 46e
Mensaje reenviado por puerto serie
-----------------------------------
Write value handler: E
  - characteristic1 write value: 34
Mensaje reenviado por puerto serie
```

FIGURA 133: ENVÍO DE SECUENCIA DE ENCENDIDO DISPOSITIVO DUO PERIPHERAL

En el dispositivo *LoPy* se recibirán las dos cadenas de *bytes*, para automáticamente reenviarlas por la antena *LoRa*. En este punto, lo datos recorren el otro extremo de la pasarela y se enciende el LED en el dispositivo *Peripheral* final, que responde con una notificación, que realiza el proceso inverso y lo recibe la antena *LoRa* del dispositivo *LoPy*. La comunicación y flujo de datos en el extremo de la pasarela correspondiente al dispositivo *Peripheral* final no se ha añadido dado que está incluida en el apartado de verificación de la primera solución. En la Figura 134 se recogen los mensajes asociados a la secuencia de encendido del LED y la recepción de la notificación del dispositivo *Bluz DK* en el dispositivo *LoPy*.

```
----- Dispositivo LoPy LoRa/Serial encendido -----

------------------------------------------------------------
- UART: Datos RECIBIDOS por puerto serie con valor = b'\x04n'

- LORA: Datos ENVIADOS por antena LORA:

------------------------------------------------------------
- UART: Datos RECIBIDOS por puerto serie con valor = b'\x03\x04'

- LORA: Datos ENVIADOS por antena LORA:

------------------------------------------------------------
- LORA: Datos RECIBIDOS por antena LoRa con valor = b'\x04OKn\x03\x04'

- UART: Datos ENVIADOS por puerto serie
------------------------------------------------------------
```

FIGURA 134: ENVÍO DE SECUENCIA DE ENCENDIDO DISPOSITIVO LOPY

A su vez, cuando el dispositivo *LoPy* reenvía los datos por la UART, debido al gran tamaño de la notificación, la lectura en el dispositivo *DUO Peripheral* se realiza de forma escalonada, como se ve en la Figura 135. En este punto, se escribe en la característica *chr2* del dispositivo *DUO Peripheral* estos datos, los cuales recibe el dispositivo *Central* final como notificación.

```
-----------------------------------
Mensaje recibido por puerto serie
     - Notificación en chr2
-----------------------------------
Mensaje recibido por puerto serie
     - Notificación en chr2
-----------------------------------
Mensaje recibido por puerto serie
     - Notificación en chr2
-----------------------------------
Mensaje recibido por puerto serie
     - Notificación en chr2
-----------------------------------
Mensaje recibido por puerto serie
     - Notificación en chr2
-----------------------------------
```

FIGURA 135: RECEPCIÓN NOTIFICACIÓN DISPOSITIVO PERIPHERAL FINAL EN DISPOSITIVO DUO PERIPHERAL

Para el apagado del LED en el dispositivo *Peripheral* final, es necesario escribir en la característica concreta las cadenas de bytes `/x04/x6E` y `/x03/x04`. El flujo de los datos será el mismo que el descrito previamente, por lo que sería redundante añadir nuevamente las capturas asociadas a la recepción y envío de datos en los diferentes dispositivos que conforman esta pasarela. Así, quedaría verificado el funcionamiento del extremo de la pasarela correspondiente al Cliente BLE.

6.4.2. COMPROBACIÓN DEL EXTREMO DE LA PASARELA CORRESPONDIENTE AL DISPOSITIVO PERIPHERAL FINAL

Una vez desarrollado el *firmware* del dispositivo *DUO Central*, se verifica y se carga en el dispositivo la aplicación siguiendo los pasos descritos en el apartado 6.4.3. En la Figura 136 se recoge la ocupación en memoria *RAM* y *Flash* del código implementado.

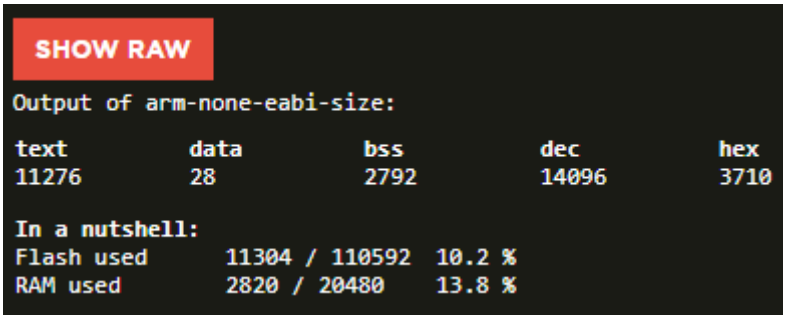

FIGURA 136: RESULTADOS VERIFICACIÓN CÓDIGO ENTORNO DE DESARROLLO DE PARTICLE EN DISPOSITIVO DUO CENTRAL

Para el análisis de los eventos producidos en el dispositivo *DUO Central*, se vuelve a hacer uso de la interfaz de usuario ofrecida por el *software PuTTY*. En el dispositivo *LoPy* se implementa el mismo código que en su dispositivo par en el extremo opuesto de la pasarela, dado que cumplen la misma función de reenviar por *LoRa* los mensajes recibidos por la UART, y viceversa. Quedando este extremo de la pasarela configurado, se pasa a verificar su funcionamiento. Para su comprobación, como se indicó en el apartado anterior, se puede implementar en el extremo de la pasarela correspondiente al Cliente BLE la primera solución propuesta, pero dado que la sección del dispositivo *DUO Peripheral* se encuentra completamente operativa y verificada, las pruebas se realizarán sobre la pasarela final directamente. Para evitar caer en redundancias, el flujo de datos en el extremo de la pasarela ya comprobado se obviará.

Inicialmente, el dispositivo *DUO Central* se encontrará en proceso de *scanning*, y una vez descubierto el dispositivo *Bluz DK*, establecerá conexión con el mismo, pasando a descubrir sus servicios, características y descriptores, como se ve en la Figura 137, Figura 138 y Figura 139.

```
Start scanning!

* BLE scan callback:
   - Advertising event type: 0
   - Peer device address type: 1
   - Peer device address: CB 9A 1D 8C 9E 6F
   - RSSI: -62
   - Advertising/Scan response data packet: 8 9 42 6C 75 7A 20 44 4B 3 19 0 0 2
1 6
      - AVD/SR data decoding -> ad_type: 9, length: 8

   - Complete Local Name: Bluz DK
____ Found BluzDK


_____ Device connected
        - Device connected handle: 64

* Service found successfully
   - Service start handle: 1
   - Service end handle: 7
   - Service uuid16: 1800
   - Service uuid128 : 0 0 18 0 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Service found successfully
   - Service start handle: 8
   - Service end handle: 8
   - Service uuid16: 1801
   - Service uuid128 : 0 0 18 1 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Service found successfully
   - Service start handle: 9
   - Service end handle: FFFF
   - Service uuid16: 0
   - Service uuid128 : 87 1E 2 23 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2
   - User defined Service uuid128 found successfully

* Discover all services completed
```

FIGURA 137: ESTABLECIMIENTO DE CONEXIÓN DISPOSITIVO DUO CENTRAL CON DISPOSITIVO BLUZ DK (I)

```
* Characteristic found successfully 0 :
   - Characteristic start handle: A
   - Characteristic end handle: C
   - Characteristic value handle: B
   - Characteristic properties: 12
   - Characteristic uuid16: 0
   - Characteristic uuid128 : 87 1E 2 24 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2

* Characteristic found successfully 1 :
   - Characteristic start handle: D
   - Characteristic end handle: FFFF
   - Characteristic value handle: E
   - Characteristic properties: E
   - Characteristic uuid16: 0
   - Characteristic uuid128 : 87 1E 2 25 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2

* Discover all characteristics completed
```

FIGURA 138: ESTABLECIMIENTO CONEXIÓN DISPOSITIVO DUO CENTRAL CON DISPOSITIVO BLUZ DK (II)

```
* Descriptor found successfully 0 - Characteristic 0 :
   - Descriptor handle: C
   - Descriptor uuid16: 2902
   - Descriptor uuid128 : 0 0 29 2 0 0 10 0 80 0 0 80 5F 9B 34 FB

* Discover all descriptors completed

* Discover all descriptors completed

* Write CCCD:
   - Connection handle: 40

RETURN CODE: 0

D --- gattWriteCCCDCallback (1)

* Write CCCD value successfully
   - Connection handle: 40
   - CCCD value: 1

* Read descriptor value successfully:
   - Connection handle: 40
   - Descriptor value attribute handle: C
   - Descriptor value : 1 0
```

FIGURA 139: ESTABLECIMIENTO CONEXIÓN DISPOSITIVO DUO CENTRAL CON DISPOSITIVO BLUZ DK (III)

Una vez establecida la conexión, la pasarela se encontrará operativa, quedando a la espera de las solicitudes por parte del dispositivo *Central* final. Cuando se produzca una solicitud de encendido del LED, asociado a las cadenas de *bytes* `/x04/x6E` y `/x03/x04`, los datos recorrerán la pasarela hasta ser recibidos, en el extremo de la pasarela correspondiente al dispositivo *Peripheral*

final, por la antena *LoRa* del dispositivo *LoPy*, que automáticamente lo reenviará por puerto serie al dispositivo *DUO Central*, como se ve en la Figura 140.

```
----- Dispositivo LoPy LoRa/Serial encendido -----

------------------------------------------------------------
- LORA: Datos RECIBIDOS por antena LoRa con valor = b'\x04n'

- UART: Datos ENVIADOS por puerto serie
------------------------------------------------------------
- LORA: Datos RECIBIDOS por antena LoRa con valor = b'\x03\x04'

- UART: Datos ENVIADOS por puerto serie
------------------------------------------------------------
```

FIGURA 140: RECEPCIÓN DE SECUENCIA DE ENCENDIDO DISPOSITIVO LOPY

Los datos recibidos en el dispositivo *DUO Central* se escriben a continuación en la característica correspondiente en el dispositivo *Bluz DK*, como se ve en la Figura 141, resultando en el encendido del LED.

```
4 6E

* Write characteristic value done:
   - Connection handle: 40
3 4

* Write characteristic value done:
   - Connection handle: 40
```

FIGURA 141: ESCRITURA SECUENCIA DE ENCENDIDO EN DISPOSITIVO BLUZ DK

Una vez que se produzca el evento de encendido del LED, el dispositivo *Bluz DK* emite una notificación en su otra característica definida por el usuario, a la que se encuentra suscrito el dispositivo *DUO Central*, que almacena los datos en un *buffer* y los reenvía por puerto serie. La recepción de las notificaciones queda representada en la Figura 142.

```
* Received new notification (   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 4F 4B 6E

* Received new notification (   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4
```

FIGURA 142: RECEPCIÓN DE NOTIFICACIÓN DE ENCENDIDO POR PARTE DEL DISPOSITIVO BLUZ DK EN DISPOSITIVO DUO CENTRAL

Finalmente, el dispositivo *LoPy* recibe los datos de la notificación del dispositivo *Bluz DK* a través del puerto serie y los reenvía por la antena *LoRa*, hacia el extremo correspondiente al Cliente BLE, como se ve en la Figura 143.

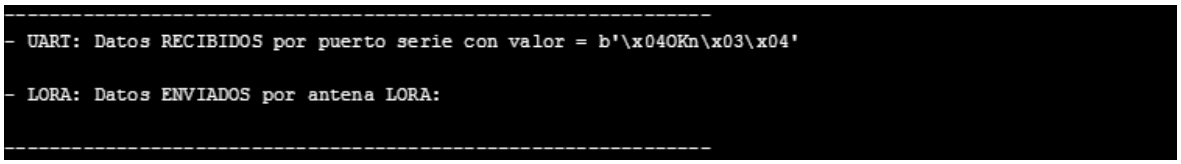

FIGURA 143: ENVÍO NOTIFICACIÓN DE ENCENDIDO EN DISPOSITIVO LOPY

Para el apagado del LED en el dispositivo *Peripheral* final, es necesario repetir el flujo de datos, con la diferencia de que las cadenas de *bytes* enviadas en este caso son `/x04/x6E` y `/x03/x04`, por lo que sería redundante repetir el proceso. Así, quedaría verificado el funcionamiento del extremo de la pasarela correspondiente al Servidor BLE y quedaría implementada la pasarela completa de la solución final desarrollada en el presente TFG.

CAPÍTULO 7: CONCLUSIONES

En este capítulo se recogen las conclusiones obtenidas tras haber completado los objetivos propuestos para este Trabajo Fin de Grado.

7.1. Conclusiones

Después de realizar el análisis de los resultados obtenidos a partir del proceso de validación funcional, se puede afirmar que se han cumplido los objetivos principales de este Trabajo Fin de Grado, habiéndose utilizado una estrategia claramente enfocada a la realización de la pasarela HW/SW mediante la separación de las tareas reconocidas. Así, ha sido posible diseñar e implementar una pasarela BLE-*LoRa*-BLE que constituya una solución de bajo coste y consumo para la interconexión de dispositivos BLE mediante la tecnología de largo alcance *LoRa*, de manera que sea posible la comunicación bidireccional entre dispositivos finales.

Para ello, en primer lugar se ha llevado a cabo un estudio teórico de los estándares de comunicación BLE y *LoRa*, así como la adquisición de conocimientos sobre los fundamentos de IoT y las redes *LPWAN*, haciendo especial énfasis en la interacción entre dispositivos y aplicaciones desarrolladas. A continuación, se ha verificado la correcta implementación de una solución inicial que implementa la pasarela BLE-*LoRa*-BLE partiendo de un dispositivo que integra ambas tecnologías de comunicación empleadas en el presente TFG, el dispositivo *LoPy*. Además del desarrollo del *firmware* de los dispositivos *LoPy* empleados, se ha hecho uso del dispositivo *Bluz DK* como dispositivo BLE *Peripheral* final, y del dispositivo *Heltec* para la realización de pruebas sobre la pasarela inicial, lo que supone un valor añadido y expande el abanico de posibilidades de cara a futuros proyectos.

Una vez verificada la implementación de la comunicación bidireccional entre dispositivos finales mediante el uso del dispositivo *LoPy* como pasarela, se ha llevado a cabo el desarrollo de una plataforma HW/SW final de manera que se separa la integración de BLE y *LoRa* en un mismo dispositivo, para pasar a implementarla de forma independiente. Así, se ha procedido a desarrollar el *firmware* de los dispositivos *RedBear DUO* para que implementen conectividad BLE y se comuniquen mediante puerto serie con los dispositivos *LoPy*. Por otro lado, se ha desarrollado un nuevo *firmware* para los dispositivos *LoPy*, de manera que en esta solución final únicamente cuenten con conectividad *LoRa* y comunicación serial con los dispositivos *DUO*. Finalmente, una vez verificada la pasarela final, se puede afirmar que los objetivos del presente TFG se han cumplido de forma satisfactoria, quedando probada la correcta implementación de una pasarela BLE-*LoRa*-BLE partiendo de dispositivos que inicialmente no cuentan con conectividad *LoRa*, es decir, atendiendo a las condiciones actuales del mercado tecnológico mundial.

Asimismo, el *firmware* desarrollado en el presente TFG permite la interacción entre ambas soluciones, haciendo posible el intercambio de extremos de las pasarelas, atendiendo a las condiciones del proyecto a desarrollar, lo que lo convierte en un proyecto fácilmente adaptable a diferentes aplicaciones y versátil en cuanto a posibilidades se refiere. Finalmente, este TFG sirve de introducción para las posibilidades que ofrece y ofrecerá la tecnología de comunicación *LoRa*, la cual, aún a falta de introducirse completamente en el campo tecnológico, presenta un potencial y unas características que auguran la importancia que tendrá en el desarrollo de futuros proyectos de IoT.

7.2. Líneas futuras

En cuanto a líneas futuras se refiere, la evolución natural del presente TFG consistiría en adaptar lo desarrollado a una solución concreta, abandonando el prototipo genérico que se presenta en este trabajo, en el que queda probada la posibilidad de interconectar una red o redes basadas en BLE con un servidor central que pueda gestionar dichas redes de manera remota, mediante el uso de tecnología *LoRa*. Así, sería posible implementar los conceptos propuestos en este TFG en campos tan variados como las ciudades inteligentes, control industrial, medicina inteligente, edificios inteligentes, etc.

Por otro lado, otra línea futura posible sería aprovechar los conceptos adquiridos sobre *LoRa* para introducirse en el protocolo de redes de área extensa *LoRaWAN*, permitiendo la implementación de redes mucho más complejas, explorando las posibilidades que ofrece este nuevo estándar de comunicación.

BIBLIOGRAFÍA

[1] IEEE IoT Technical Community. "Towards a definition of the Internet of Things (IoT)". IEEE Internet of Things, Rev. 1, 27 May. 2015

[2] Hughes B., "*The Internet of things: an overview",* Computer Weekly, 2017. [Online] Disponible en: http://www.computerweekly.com/opinion/The-internet-of-things-an-overview. [Último acceso: 15-Septiembre-2018]

[3] Rahul, "*IoT applications spanning across industries*", IBM, 2017. [Online] Disponible en: https://www.ibm.com/blogs/internet-of-things/iot-applications-industries/?lnk=hm. [Último acceso: 15-Septiembre-2018]

[4] "About Machina Research", Machina Research, 2016. [Online] Disponible en: https://machinaresearch.com/news/press-release-global-internet-of-things-market-to-grow-to-27-billion-devices-generating-usd3-trillion-revenue-in-2025/.[Último acceso: 15-Septiembre-2018]

[5] "*Global Internet of Things market to grow to 27 billion devices, generating usd3 trillion revenue in 2025"*, Machina Research, 2016. [Online] Disponible en: https://machinaresearch.com/news/press-release-global-internet-of-things-market-to-grow-to-27-billion-devices-generating-usd3-trillion-revenue-in-2025/.[Último acceso: 15-Septiembre-2018]

[6] "*What is LPWA?*, Ingenu, 2018. [Online] Disponible en: http://www.ingenu.com/portfolio/what-is-lpwa-white-paper/. [Último acceso: 15-Septiembre-2018]

[7] "*LoRa Alliance Technology"*, LoRa Alliance, 2018. [Online] Disponible en: https://www.lora-alliance.org/technology. [Último acceso: 15-Septiembre-2018]

[8] Drinkwater D., "*Samsung Rolls out LoRa IoT network in South Korea"*, Internet of Business, 2016. [Online] Disponible en: https://internetofbusiness.com/samsung-rolls-lora-network-south-korea/. [Último acceso: 15-Septiembre-2018]

[9] Orange Connected Objects & Partnerships, "*LoRa Device Developer Guide"*, Abril 2016.

[10] Gomez C., Oller J., &Paradells J., "*Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology",* Sensors: 2012; 12(9): 11734–11753, 2012

[11] Townsend K., Cufí C., Davidson A., Davidson R., "*Getting Started with Bluetooth Low Energy*", O'reilly, 2014

[12] Heydon R., "*Bluetooth Low Energy: The developer's handbook*", Prentice Hall, 2012.

[13] "Bluetooth Low Energy Channels", Microchip Developer, 2017 [Online] Disponible en http://microchipdeveloper.com/wireless:ble-link-layer-channels. [Último acceso: 10-Abril-2018]

[14] "*Bluetooth Low Energy Packet Types*", Microchip Developer, 2017 [Online]. Disponible en http://microchipdeveloper.com/wireless:ble-link-layer-packet-types. [Último acceso: 15-Septiembre-2018]

[15] Augustin A.; Yi J., Clausen T. y Townsley M, "*A Study of LoRa: Long Range & Low Power Networks for the Internet of Things*", Francia, 2016.

[16] "*LoRaWAN Specification V1.1*", LoRa Alliance, 2015. Disponible en: https://lora-alliance.org/sites/default/files/2018-04/lorawantm_specification_-v1.1.pdf. [Último acceso: 3-Mayo-2018]

[17] Springer A.; Gugler, W.; Huemer, M.; Reind, L.; Ruppel, C.; Weigel, R.; "*Spread spectrum communications using chirp signals*". In Proceedings of the IEEE/AFCEA Information Systems for Enhanced Public Safety and Security (EUROCOMM 2000), Munich, Germany, 19 May 2000; pp. 166–170

[18] "*LoRa Channels List |Table of LoRa Frequency Channels"*, RF Wireless World [Online]. Disponible en: http://www.rfwireless-world.com/Tutorials/LoRa-channels-list.html. [Último acceso: 3-Mayo-2018]

[19] "*What is the LoRaWAN specification*", LoRa Alliance, 2018 [Online]. Disponible en: https://lora-alliance.org/about-lorawan. [Último acceso: 3-Mayo-2018]

[20] *"PC Notebook Compaq 15-h051ns (ENERGY STAR) - Guías de usuario"*, Support HP, 2018 [Online]. Disponible en: https://support.hp.com/es-es/product/compaq-15-h000-notebook-pc-series/6545572/model/7500811/manuals. [Último acceso: 7-Octubre-2018]

[21] "*Redmi Note 4*", Xiaomi, 2018 [Online]. Disponible en: https://www.mi.com/es/note4/. [Último acceso: 7-Octubre-2018].

[22] "*LoPy*", docs.pycom.io, 2018 [Online]. Disponible en: https://docs.pycom.io/datasheets/development/lopy.html. [Último acceso: 7-Octubre-2018]

[23] "*RedBear Duo",* Github, 2018 [Online]. Disponible en: https://github.com/redbear/Duo. [Último acceso: 7-Octubre-2018]

[24] "*Bluz* DK", docs.bluz.io, 2018 [Online]. Disponible en: http://docs.bluz.io/hardware/dk/. [Último acceso: 7-Octubre-2018]

[25] "*New version WiF LoRa 32(V2)*", Heltec Automation, 2018 [Online]. Disponible en: http://www.heltec.cn/project/wifi-lora-32/?lang=en. [Último acceso: 7-Octubre-2018]

[26] "*Atom*", atom.io, 2018 [Online]. Disponible en: https://atom.io/. [Último acceso: 7-Octubre-2018]

[27] "*Particle IDE*", build.particle.io, 2018 [Online]. Disponible en: https://build.particle.io/. [Último acceso: 7-Octubre-2018]

[28] "*Download PuTTY - a free SSH and telnet client for Windows*", Putty.org, 2018. [Online]. Disponible en: https://www.putty.org/. [Último acceso: 7-Octubre-2018].

[29] "*nRF Connect for Mobile / Nordic mobile Apps / Products / Home - Ultra Low Power Wireless Solutions from NORDIC SEMICONDUCTOR*", Nordicsemi.com, 2018. [Online]. Disponible en: https://www.nordicsemi.com/eng/Products/Nordic-mobile-Apps/nRF-Connect-for-Mobile. [Último acceso: 7-Octubre-2018].

PLIEGO DE CONDICIONES

Se procede a presentar las condiciones bajo las que se ha desarrollado el presente Trabajo Fin de Grado. Se realizará una diferenciación entre el conjunto de herramientas *hardware*, *software* y *firmware* empleadas para llevar a cabo su realización.

1. Condiciones *hardware*

En la Tabla 6 se indica el conjunto de dispositivos *hardware* utilizados, con su modelo correspondiente.

Dispositivo/herramienta	Modelo	Fabricante/comerciante
Smartphone	Redmi Note 4	Xiaomi
Ordenador portátil	PC Notebook Compaq 15-h051ns	Compaq
LoPy	LoPy con Expansion Board 2.0	Pycom
RedBear DUO	RedBear DUO con pines soldados	RedBear
Bluz DK	Bluz DK con pines soldados	Bluz
Heltec WiFi LoRa 32	Heltec WiFi LoRa 32 con pines soldados	Heltec Automation

TABLA 6: CONDICIONES HARDWARE

2. Condiciones *software*

En la Tabla 7 se recogen las aplicaciones *software* utilizadas, con su versión correspondiente.

Software	Versión	Desarrollador
Sistema operativo portátil	Microsoft Windows 10 Home	Microsoft
Sistema operativo smartphone	Android 8 Oreo	Xiaomi-Google
Microsoft Office	Microsoft Office 365 ProPlus	Microsoft
Microsoft Visio	2016	Microsoft
Microsoft Project	2016	Microsoft
Particle IDE	-	Particle
PuTTY	V0.70	PuTTY project
Google Chrome	V 71.0.3578.98/ 64 bits	Google
Adobe Reader	V11.0.21.18	Adobe Systems Software Ireland Ltd.
nRF Connect	V4.19.0	Nordic Semiconductor
Atom IDE	V 1.34.0	Github

TABLA 7: CONDICIONES SOFTWARE

3. Condiciones *firmware*

En la Tabla 8 se indica el *firmware* utilizado para cada dispositivo, y su versión correspondiente.

Firmware	Versión
LoPy	v1.18.0
RedBear DUO	v0.3.1
Bluz DK	v1.1.47
Heltec	-

TABLA 8: CONDICIONES DE FIRMWARE

PRESUPUESTO

En este capítulo se abordará el presupuesto que recoge los gastos generados en la realización del presente Trabajo Fin de Grado. El presupuesto, asimismo, está compuesto por las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, dividida a su vez en:
  - Amortización del material hardware.
  - Amortización del material software.
- Redacción de la documentación.
- Derechos de visado del COITT (Colegio Oficial de Ingenieros Técnicos de Telecomunicación).
- Gastos de tramitación y envío.
- Material fungible.

Una vez analizados los puntos que componen el presupuesto se procederá a aplicar los impuestos vigentes y se procederá a la obtención del coste total del Trabajo Fin de Grado.

1. Trabajo tarifado por tiempo empleado

En esta sección se contabilizarán los gastos que corresponden a la mano de obra según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación. Con este objetivo se ha utilizado la fórmula de la Ecuación 1.

$$\boldsymbol{H = Ct \cdot 74{,}88 \cdot Hn + Ct \cdot 96{,}72 \cdot He} \qquad (1)$$

Donde:

- H: Honorarios totales por el tiempo dedicado.
- Hn: Número de horas normales trabajadas dentro de la jornada laboral.
- Ct: Factor de corrección que depende del número de horas trabajadas.
- He: Número de horas especiales trabajadas.

Se han invertido un total de 300 horas en la realización del presente Trabajo Fin de Grado. Todas ellas se han realizado dentro del horario normal, por lo que el número de horas especiales es cero. De acuerdo con lo establecido por el COITT, el factor de corrección Ct que se aplica para 300 horas trabajadas es de 0,60, como se aprecia en la Tabla 9.

Horas	Factor de correción
Hasta 36	1
Exceso de 36 hasta 72	0,9
Exceso de 72 hasta 108	0,8
Exceso de 108 hasta 144	0,7
Exceso de 144 hasta 180	0,65
Exceso de 180 hasta 360	0,6

TABLA 9: COEFICIENTES REDUCTORES PARA TRABAJO TARIFADO SEGÚN EL COITT

Por tanto, haciendo uso de la ecuación 1:

$$H = 0{,}6 \cdot 74{,}88 \cdot 300 + 0{,}6 \cdot 96{,}72 \cdot 0 = 13.478{,}40€$$

El trabajo tarifado por tiempo empleado asciende a la cantidad de ***trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos.***

2. Amortización del inmovilizado material

El inmovilizado material se compone de los recursos *hardware* y *software* empleados para la realización de este Trabajo Fin de Grado.

Se ha utilizado un sistema de amortización lineal, en el que se supone que el inmovilizado material se deprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula como se indica en la Ecuación 2.

$$Cuota\ anual = Valor\ de\ adquisición / Valor\ residual Número\ de\ años\ de\ vida\ útil \quad (2)$$

El Valor residual corresponde con el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil. A continuación, se analizará la amortización de los recursos *hardware* y *software* utilizados.

- **Amortización del material *hardware***: En la Tabla 10 se especifican los elementos *hardware* amortizables utilizados para la realización del presente trabajo, indicando su valor de adquisición y su amortización.

Dispositivo/herramienta	Valor de adquisición	Valor de amortización
Smartphone	120 €	120 €
Ordenador portátil	500 €	30 €
LoPy x 2	69,90 €	69,90 €
RedBear DUO x 2	38,54 €	38,54 €
Bluz DK	10,39 €	10,39 €
Heltec WiFi LoRa 32	13,77 €	13,77 €
Total	753 €	283 €

TABLA 10: AMORTIZACIÓN DEL MATERIAL HARDWARE

Debido al bajo precio de los dispositivos, el conjunto formado por los dispositivos IoT y el *smartphone*, su valor de amortización coincide con su valor de adquisición. Para el resto de los elementos *hardware* se ha utilizado la formula indicada por la Ecuación 2. El coste total del material hardware asciende a ***doscientos ochenta y tres euros.***

- **Amortización del material *software***: En la Tabla 11 se especifican los elementos *software* amortizables utilizados para la realización del presente trabajo, indicando su valor de adquisición y su amortización.

Software	Valor de adquisición	Valor de amortización
Sistema operativo portátil	Licencia ULPGC	0 €
Microsoft Office	Licencia ULPGC	0 €
Microsoft Visio	Licencia ULPGC	0 €
Microsoft Project	Licencia ULPGC	0 €
Particle IDE	0.00 € / Software acceso libre	0 €
PuTTY	0.00 € / Software acceso libre	0 €
Google Chrome	0.00 € / Software acceso libre	0 €
Adobe Reader	0.00 € / Software acceso libre	0 €
nRF Connect	0.00 € / Software acceso libre	0 €
Atom IDE	0.00 € / Software acceso libre	0 €

TABLA 11: AMORTIZACIÓN DEL MATERIAL SOFTWARE

El coste total asociado al material *software* es de ***cero euros***. Sumando los costes del inmovilizado del material *hardware* y *software* se obtiene el coste total de inmovilizado material, indicado en la Tabla 12.

Concepto	Coste
Material Hardware	283 €
Material Software	0 €
Total	283 €

TABLA 12: COSTE TOTAL INMOVILIZADO MATERIAL

Por tanto, el coste total del inmovilizado material asciende a ***doscientos ochenta y tres euros.***

3. Redacción del trabajo

Para el cálculo del coste asociado a la redacción de la memoria del presente trabajo se ha usado la fórmula mostrada en la Ecuación 3.

$$\boldsymbol{R = 0{,}07 \cdot P \cdot Cn} \qquad (3)$$

Donde:

- **R**: Honorarios por la redacción del trabajo.
- **P**: Presupuesto.
- $\boldsymbol{Cn}$: Coeficiente de ponderación en función del presupuesto.

El cálculo del presupuesto se realiza mediante la suman de los costes del trabajo tarifado por tiempo empleado y la amortización del inmovilizado material. En la Tabla 13 se indica este cálculo.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	283 €
Total	13.761,40 €

TABLA 13: PRESUPUESTO

A continuación se analiza el coeficiente de ponderación Cn para este presupuesto. Según el COITT, para sueldos inferiores a 30.050,00 € les corresponde un valor de 1,00. En consecuencia, el coste asociado a la redacción del Trabajo Fin de Grado se indica en la Ecuación 4:

$$\boldsymbol{R = 0.07 \cdot 13.761{,}40 \cdot 1 = 963{,}29\ €} \qquad (4)$$

El coste de la redacción del trabajo asciende a ***novecientos sesenta y tres euros con veintinueve céntimos.***

4. Derechos de visado del COITT

Los derechos de visado para proyectos técnicos de carácter general en el año 2018 quedan establecidos por COITT, según la Ecuación 5.

$$\boldsymbol{V = 0{,}006 \cdot P1 \cdot C1 + 0{,}003 \cdot P2 \cdot C2} \qquad (5)$$

Donde:

- $\boldsymbol{V}$: Coste de visado del trabajo.
- $\boldsymbol{P1}$: Presupuesto del proyecto.
- $\boldsymbol{C1}$: Coeficiente reductor en función del presupuesto.
- $\boldsymbol{P2}$: Presupuesto de ejecución material correspondiente a la obra civil.
- $\boldsymbol{C2}$**:** Coeficiente reductor en función del presupuesto de ejecución material correspondiente a la obra civil.

El coeficiente $C1$ es de 1,00 € para proyectos de presupuesto inferior a 30.050,00 €. Por otro lado, como El valor $P2$ es de 0,00 € debido a que no se realiza ninguna obra civil, el coeficiente $C2$ no se aplica. En la Tabla 14 se muestra el presupuesto del proyecto que se obtiene al sumar las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del trabajo.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	283 €
Redacción del trabajo	963,29 €
Total	14.724,69 €

TABLA 14: PRESUPUESTO INCLUYENDO TRABAJO TARIFADO, AMORTIZACIÓN Y COSTE DE REDACCIÓN

En consecuencia, el cálculo del coste por derechos de visados del presupuesto está indicado en la Ecuación 6.

$$\boldsymbol{V = 0,006 \cdot 14.724,69 \cdot 1 + 0,03 \cdot 0 \cdot C2 = 88,35 \text{ €} \quad (6)}$$

Por tanto, el coste por derechos de visado del presupuesto asciende a ***ochenta y ocho euros con treinta y cinco céntimos.***

5. Gastos de tramitación y envío

Los gastos derivados de la tramitación y envío ascienden a *seis* euros (6,00 €) por cada documento visado de forma telemática.

6. Material Fungible

Durante el desarrollo de este Trabajo Fin de Grado se han empleado otros materiales aparte de los recursos *hardware* y *software* ya analizados. Estos materiales se documentan como material fungible. En la Tabla 15 se indican los costes derivados de estos recursos:

Concepto	Coste
Folios	10 €
Tóner de la impresora	30 €
Encuadernado	5 €
Total	45 €

TABLA 15: COSTE MATERIAL FUNGIBLE

El coste del material fungible asciende a ***cuarenta y cinco euros.***

7. Aplicación de impuestos y coste total

La realización del presente Trabajo Fin de Grado está gravada con el Impuesto General Indirecto Canario (IGIC) en un siete por ciento (7%). En la Tabla 16 se indica el cálculo del presupuesto total del Trabajo Fin de Grado aplicando el impuesto.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13.478,40 €
Amortización del inmovilizado material	283 €
Redacción del trabajo	963,29 €
Derechos de visado del COITT	88,35 €
Gastos de tramitación y envío	6 €
Costes de material fungible	45 €
Total (sin IGIC)	14.864,04 €
IGIC (7%)	1.040,48 €
Total	15.905 €

TABLA 16: PRESUPUESTO TOTAL TRABAJO FIN DE GRADO

Por tanto, el presupuesto total del Trabajo Fin de Grado "Implementación de una pasarela para la interconexión de dispositivos BLE mediante tecnología LoRa" asciende a ***quince mil novecientos cinco euros***.

En Las Palmas de Gran Canaria, a 16 de enero de 2019.

Fdo. Carlos S. Viera Betancor

ANEXO

Adjunto a la memoria del presente Trabajo Fin de Grado se encuentra disponible un archivo ZIP denominado ***TFG_CarlosVieraBetancor.zip***. En este anexo se especifica el contenido incluido en este archivo ZIP:

- Memoria del TFG "*Implementación de una pasarela para la interconexión de dispositivos BLE mediante tecnología LoRa"* en lengua española y formato PDF.

- Carpeta ***Firmware*** con los archivos correspondientes al código desarrollado en el presente TFG, que a su vez incluye:

  - Carpeta ***Solución Inicial***: carpeta con el código desarrollado en la solución inicial.

    1. Archivo ***lopy_central.py***: *firmware* desarrollado para el dispositivo *LoPy Central*.
    2. Archivo **lopy_peripheral.py**: *firmware* desarrollado para el dispositivo *LoPy Peripheral*.
    3. Archivo **bluz-bulb.ino**: *firmware* implementado en el dispositivo *Bluz DK* para actuar como dispositivo BLE *Central* final.
    4. Archivo ***heltec.ino***: *firmware* implementado en el dispositivo *Heltec* para realizar pruebas de verificación en la solución inicial.

  - Carpeta ***Solución final****:* carpeta con el código desarrollado en la solución final.

    1. Archivo ***lopy_final.py***: *firmware* desarrollado para los dispositivos *LoPy*.
    2. Archivo ***duo-central.ino:*** *firmware* desarrollado para el dispositivo *DUO Central.*
    3. Archivo ***duo-peripheral.ino****: firmware* desarrollado para el dispositivo *DUO Peripheral*.