ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN

Y ELECTRÓNICA

TRABAJO FIN DE GRADO

"Desarrollo de un dispositivo Central/Peripheral BLE 5 para aplicaciones de IoT basado en la placa Sparkfun Artemis Thing Plus"

TITULACIÓN:	**Grado en Ingeniería en Tecnologías de la Telecomunicación**
MENCIÓN:	**Sistemas Electrónicos**
AUTOR:	**Jorge Fernández Heredero**
TUTORES:	**D. Valentín de Armas Sosa**
	D. Félix Tobajas Guerrero
FECHA:	**Julio 2021**

Agradecimientos

"*A mi familia por el apoyo incondicional en esta y todas las etapas de mi vida. A todos los compañeros que conocí en este camino y han sido fuente de motivación continua. A mis tutores Valentín y Félix por ser ejemplos como ingenieros y como personas que algún día querré ser. Especialmente a Tomás Real, amigo y compañero de vida. Por último, a Ylenia y Emma compañeras de sufrimiento y amigas de corazón.*"

Resumen

En los últimos años, la tecnología *Bluetooth Low Energy* (BLE) ha adquirido especial importancia en el día a día, no solo por la versatilidad que proporciona disponer de una conexión de corto alcance con velocidades de transmisión elevadas, sino por el incremento del uso de la tecnología *IoT*.

El presente TFG se asienta sobre la base teórica de la especificación BLE 5. Esta última versión del protocolo *Bluetooth Low Energy* ofrece diversas mejoras con respecto a sus antecesores.

En su desarrollo, se lleva a cabo la implementación funcional de un dispositivo *Peripheral y* un dispositivo *Central* en la plataforma *Artemis Thing Plus* de *Sparkfun* usando el entorno de desarrollo *Ambiq Apollo 3* basado en la pila *BLE Cordio*. Esta implementación, tiene como objetivo final realizar un análisis del *Thougthput* de la plataforma *Artemis Thing Plus*.

Abstract

In recent years, Bluetooth *Low Energy* (BLE) technology has gained special importance in everyone's daily life, not only because of the versatility provided by having a short-range connection with high transmission speeds, but also because of the increasing use of IoT technology.

This TFG is based on the basis of the BLE 5 specification. This latest version of the Bluetooth *Low Energy* protocol offers several improvements over its predecessors.

The functional implementation of a *Peripheral* device and a *Central* device on *Sparkfun*'s *Artemis Thing Plus* platform is carried out using the *Ambiq Apollo 3* development environment based on the *BLE Cordio* stack. The objective of this implementation is to analize the *Thougthput* of the *Artemis Thing Plus* platform.

Índices

Memoria

Pliego de condiciones

Presupuesto

Índice de ilustraciones

Índice de tablas

Acrónimos

ACL Access Control List

ADC Analog Digital Converter

API Application Programming Interfaces

ARM Advanced RISC Machine

ATT Attribute protocol

BC BroadCast

BLE Bluetooth *Low Energy*

CAN Controller Area Network

CCCD Client Characteristic Configuration Descriptor

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DCT Device Configuration Table

DFU Device Firmware Update

EITE Escuela de Ingeniería de Telecomunicación y Electrónica

FHSS Frequency Hopping Spread Spectrum

FT File transfer

GAP Generic Access Profile

GATT Generic Attribute profile

GCC GNU Compile Collection

GFSK Gaussian Frequency Shift Keying

GPIO General Purpose Input-Output

HCI *Host* Controller Interface

HW Hardware

I2C Inter Integrated Circuits

I2S Inter-IC Sound.

IDE Integrated Development Kit

IEEE Institute of Electrical and Electronics Engineers

IFS Inter Frame Space

IGIC Impuesto General Indirecto Canario

IoT Internet of Things

IRK Clave de resolución de identidad

ISM Industrial, Scientific and Medical

JTAG Joint Test Action Group.

KB Kilobyte

L2CAP Logical Link Control and Adaptation Protocol

LED Light-Emitting Diode

LL *Link Layer*

MB Megabyte

Mbps Mega bit por segundo

MD More Data

MIT Massachusetts Institute of Technology

MTU Maximum Transmission Unit

OTA Over-The-Air

PAL Phase Alternate Line

PB Packet Boundary

PC Personal Computer

PDU Protocol Data Unit

PHY PHYsical layer

PWM Pulse-Width Modulation

RGB Red, Green, Blue

RTC Real Time Clock

RTOS Real-Time Operating System

SDK Software Development Kit

SIG Special Interest Group

SM Security Manager

SMP Security Manager Protocol

SO Sistema operativo

SOC System on chip

SPI Serie Peripheral Interface

SPP Serie Port Profile

SRAM Static Random-Access Memory

TFG Trabajo Fin de Grado

TLV Type-Length-Value

UART Universal Asynchronous Receiver-Transmitter

UIT Unión Internacional de Telecomunicaciones

ULPGC Universidad de Las Palmas de Gran Canaria

USB Universal Serie Bus

UUID Universally Unique Identifier

WDXS Wireless Data Exchange Server

WICED Wireless Internet Connectivity for Embedded Devices

WPAN Personal Area Network

Memoria

Capítulo 1: Introducción

1.1 Antecedentes

En los últimos años, la evolución de las tecnologías ha desembocado de forma indómita en Internet. Es por ello, que al hablar de tecnología en la actualidad, se hace en referencia en muchos casos a IoT (*Internet of Things*). Aunque parezca un concepto relativamente novedoso, fue en 1999 cuando se acuña la primera definición de la mano de Kevin Asthon, que afirmó "IoT es el mundo en el que cada objeto tiene una identidad virtual propia y capacidad potencial para integrarse e interactuar de manera independiente en la Red con cualquier otro individuo, ya sea una máquina (M2M) o un humano" [1].

Junto a la evolución de Internet, las posibilidades tecnológicas aumentan sin cesar de manera que, el número de "cosas" conectadas a Internet sobrepasó en 2008 el número de habitantes del planeta. Siguiendo la previsión que se observa en el portal de estadísticas *Statista,* se estima que en 2025 habrá más de 75.000 millones de dispositivos conectados en todo el mundo, y casi 31.000 millones en el actual año 2020, como se observa en la Ilustración 1 [2] .

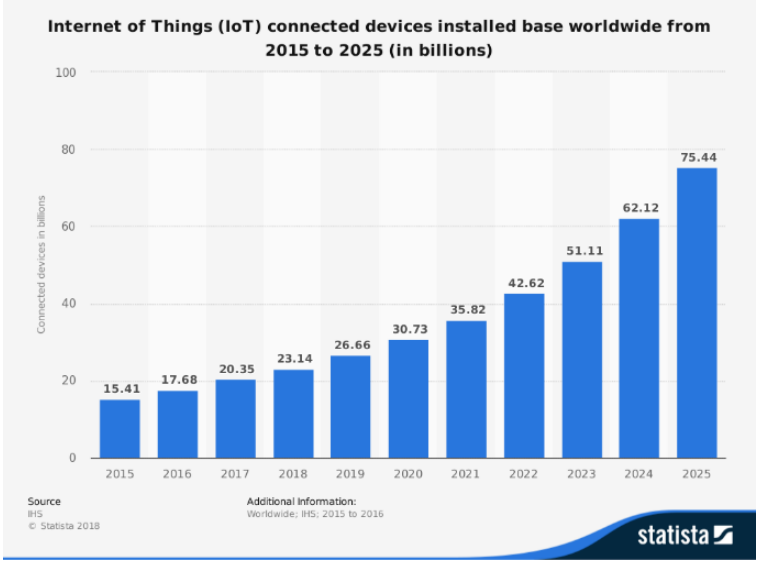


Ilustración 1 : Dispositivos conectados a la red 2015 – 2025 (*IoT*). [2]

IoT tiene como propósito hacer más sencilla la vida de las personas, haciendo interactivos los objetos que se utilizan en la vida cotidiana. Estos se valen de sistemas hardware especializados con conectividad, ya sea a Internet o utilizando estándares de menor alcance como *Bluetooth*.

El estándar *Bluetooth* será objeto de estudio a lo largo de este Trabajo Fin de Grado (TFG), siendo una especificación industrial para WPAN (*Wireless Personal Area Network*) que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia. *Bluetooth* nace de la necesidad de empresas informáticas y de telecomunicaciones de desarrollar una interfaz abierta y de bajo coste para facilitar la comunicación entre dispositivos inalámbricos [3].

A lo largo de los años, a medida que aumenta la complejidad de los dispositivos y sus funcionalidades, la cantidad de datos que se necesita trasmitir ha aumentado consigo, por lo que *Bluetooth* se ha ido actualizando hasta llegar a sus versiones de *Low Energy*, caracterizadas por un menor consumo de potencia y un aumento de densidad en sus tramas de datos. En la Ilustración 2, se pueden observar las diferentes versiones de BLE (*Bluetooth Low Energy*) [4].

En una conexión BLE es posible diferenciar dos tipos de dispositivos, *Peripheral* y *Central.* Los dispositivos *Peripheral* son dispositivos pequeños, de baja potencia y reducidos recursos, que pueden conectarse a dispositivos de tipo *Central*, por lo general mucho más potentes. Un ejemplo de dispositivo *Peripheral* puede ser un glucómetro, un medidor de pulsaciones, etc. Por otro lado, el dispositivo *Central* se corresponde normalmente con un teléfono móvil o una tableta, que presentan unas prestaciones mucho mayores [5].

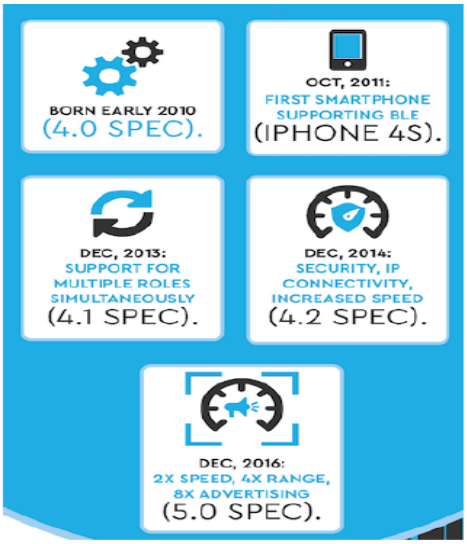


Ilustración 2 : Historia del BLE [5].

En este TFG se analizará detalladamente la especificación de *Bluetooth Low Energy* (BLE), contemplando su versión 5.0. Desde su lanzamiento en 2016, sigue siendo uno de los protocolos con más potencial, debido a su compromiso entre energía, velocidad y alcance. Se estudiará a partir de la implementación de un dispositivo *Central*/*Peripheral* por medio de la plataforma *Artemis Thing Plus,* mostrada en la Ilustración 3 diseñada por la empresa *SparkFun* [6] con el objetivo de implantar aplicaciones

IoT con *Bluetooth Low Energy*. La elección de esta placa en concreto se ha realizado para satisfacer la necesidad de su estudio dentro del grupo de investigación, ya que destaca por su bajo coste y su compatibilidad con *Bluetooth Low Energy*, características muy relevantes en el marco tecnológico.

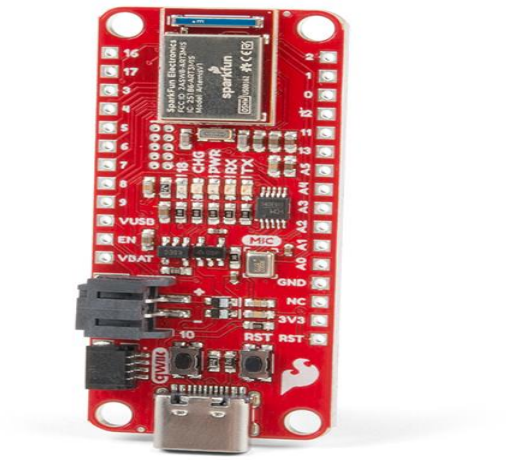


Ilustración 3 : Artemis Thing Plus

En anteriores Trabajos Fin de Título realizados en el seno del grupo de investigación, se han implementado las funcionalidades de dispositivos BLE 4.x mediante lenguaje *Wiring o Wiced* SDK, como en el caso [7]. Sin embargo, a diferencia de trabajos previos, en el presente TFG se utilizará el entorno *Ambiq SDK* basado en la librería BLE *Cordio* de ARM, proporcionada por el fabricante del microprocesador Apollo3, que representa el MCU (*Micro Controller Unit)* de la plataforma *Artemis Thing Plus*. En este TFG se apuesta por una programación a más bajo nivel que en casos anteriores, siendo uno de los aspectos diferenciadores junto con el uso de BLE 5.

1.2 Objetivos

Como se puede deducir del anterior apartado, el objetivo principal del presente TFG consiste en implementar la funcionalidad, tanto de un dispositivo *Central*, como de un dispositivo *Peripheral,* interconectados mediante el protocolo BLE, utilizando la plataforma IoT *Artemis Thing Plus* de la empresa *Sparkfun*.

La implantación de la funcionalidad de los dispositivos *Central / Peripheral* se realizará a partir del entorno de desarrollo *Ambiq SDK* basado en la librería BLE *Cordio*, además de utilizarse la pila oficial del estándar BLE, con el fin de poder analizar la influencia de los parámetros asociados al *firmware* de usuario, con especial énfasis en el *Throughput*, para de esta forma, analizar su funcionamiento e intentar optimizar este parámetro lo máximo posible. Este objetivo principal se puede desglosar en los siguientes objetivos parciales:

O1. Análisis del funcionamiento de BLE 5.

O2. Análisis de las características y el flujo de desarrollo de aplicaciones en la placa *Artemis Thing Plus*.

O3. Estudio del entorno Ambiq SDK para el microprocesador Apollo3.

O4. Desarrollo, implementación y verificación funcional de un dispositivo *Peripheral* BLE 5.

O5. Desarrollo, implementación y verificación funcional de un dispositivo *Central* BLE 5.

O6. Caracterización de las prestaciones de un enlace BL5 a partir de los dispositivos *Peripheral/Central* implementados.

1.3 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.

1.4 Estructura del documento.

El presente documento está dividido en tres partes diferenciadas: Memoria, Pliego de condiciones y Presupuesto. A su vez, la Memoria se ha estructurado en seis capítulos, además de las referencias bibliográficas empleadas, tal como se describe a continuación:

En el ***Capítulo 1*** se realiza una introducción que proporciona al lector una visión básica acerca de los conceptos que se tratarán en el presente TFG. Además, se presentan los objetivos a satisfacer, el peticionario, y la estructura que seguirá el documento.

En el ***Capítulo 2*** se describe el protocolo de comunicaciones BLE 5, con el objetivo de comprender su funcionamiento y sus principales características.

En el ***Capítulo 3*** se describe el funcionamiento de *Cordio Stack BLE*, pila de protocolos usada para implementar la funcionalidad de los dispositivos *Central* y *Peripheral*, además de describir las características de la plataforma *Artemis Thing Plus*.

En el ***Capítulo 4*** se documenta todo el proceso realizado para la implementación del dispositivo *Peripheral* y del dispositivo *Central,* además de verificar experimentalmente su correcta funcionalidad.

En el ***Capítulo 5*** se implementa la funcionalidad del *Stream* de datos, establecida para poder alcanzar la máxima tasa de envío de datos entre los dispositivos *Central* y *Peripheral* implementados.

En el ***Capítulo 6*** se lleva a cabo la caracterización del *Throughput* alcanzable a partir del *Stream* de datos, además de analizar la forma de optimizarlo.

La segunda parte del documento consiste en el Pliego de Condiciones, mientras que la tercera parte se corresponde con el Presupuesto:

En el **Pliego de condiciones** se exponen las condiciones bajo las que se ha desarrollado el presente TFG.

En el **Presupuesto** se recogen los gastos generados en la realización del presente TFG.

Capítulo 2: El protocolo BLE

Con el objetivo de comprender el funcionamiento y las características que ofrece el protocolo BLE, en este capítulo se exponen sus funcionalidades más importantes, así como los aspectos básicos relacionados con su implementación.

2.1 Aspectos básicos y características BLE

Bluetooth comenzó como remplazo al cable, siendo un protocolo de comunicaciones inalámbricas. La primera versión fue datada en 1994 de la mano de *Ericsson*. Hoy en día se pueden encontrar dos versiones: *Bluetooth Classic* y *Bluetooth Low Energy (BLE)* [4].

BLE fue introducido en su cuarta versión (4.0), diferenciada de las anteriores por su bajo consumo de recursos, lo que conlleva muchas ventajas a la hora de implementarlo en cualquier dispositivo hardware (temperatura, velocidad, ahorro de energía, etc.).

BLE está diseñado con el objetivo de realizar transmisiones de pequeñas cantidades de datos y con tiempos de transmisión cortos, reduciendo el consumo de potencia y siendo bastante más fiable a poca distancia que cualquier otro. El rango de uso está limitado por ciertos factores:

- Opera en la banda de radio industriales, científicas y médicas (ISM) 2.4 GHz, viéndose afectado por obstáculos como objetos metálicos, paredes y agua.
- El encapsulado de la antena sería una limitación, especialmente si se trata de una antena interna.
- La orientación del dispositivo determinado directamente la orientación de la antena.

La topología de red básica usada en BLE es de tipo estrella, comunicación *Master*/*Slave* en la que los dispositivos *Master* podrán establecer múltiples conexiones con dispositivos *Slave.* Sin embargo, los dispositivos *Slave solo* podrán conectarse a un dispositivo *Master* simultáneamente. Todos los dispositivos BLE pueden comunicarse de acuerdo con dos modos de operación [8]:

1) *Broadcasting*, mecanismo de difusión que permite enviar datos a cualquier dispositivo receptor en el rango de escucha, habilitando la comunicación en un solo sentido con cualquier dispositivo capaz de recibir los datos transmitidos. Se pueden diferenciar dos roles: *Broadcaster* y *Observer*. El dispositivo *Broadcaster* envía paquetes de *Advertising* periódicamente a cualquier dispositivo BLE (*Master*); el dispositivo *Observer*, por su parte, escanea repetidamente las frecuencias preconfiguradas para recibir cualquier paquete de *Advertising* no conectable que se esté

transmitiendo (*Slave*). La Ilustración 4 muestra la topología de este mecanismo de comunicación BLE.

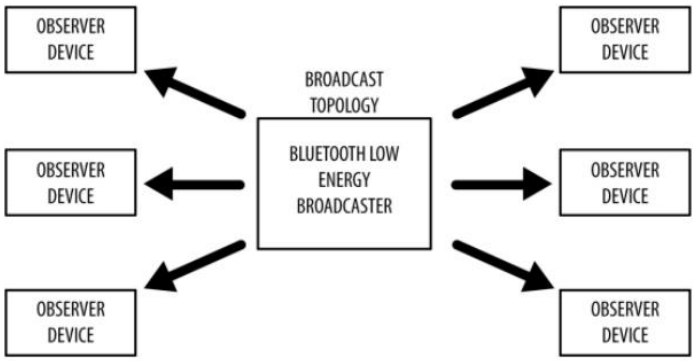


Ilustración 4: Topología Broadcasting BLE[8].

2) *Connections,* en el que se establece una conexión entre dos dispositivos, a través de la cual se transmiten datos en ambas direcciones, siendo este intercambio permanente y periódico. Dicha conexión es privada (los datos solo se envían y reciben entre los dos dispositivos involucrados). Al igual que en el anterior modo de operación, se pueden diferenciar dos roles:

- *Central:* Explora continuamente frecuencias prestablecidas para recibir paquetes de *Advertising* y, cuando es necesario, establece una conexión. Una vez establecida la conexión, será el dispositivo *Central* el que controle la temporización e iniciaría el intercambio de datos.
- *Peripheral:* Envía continuamente paquetes de *Advertising* conectables de forma periódica, y acepta conexiones entrantes (como máximo una). Una vez esté conectado, el dispositivo *Peripheral* sigue la sincronización iniciada por el dispositivo *Central,* realizando el intercambio de datos cuando se solicite.

Para iniciar una conexión, un dispositivo *Central* recibe en primer lugar los paquetes de *Advertising* conectables, y a continuación, envía una solicitud al dispositivo *Peripheral* para establecer una conexión exclusiva entre ambos. Una vez que se establece la conexión, el dispositivo *Peripheral* deja de enviar *paquetes de Advertising* y los dos dispositivos pueden comenzar a intercambiar paquetes de datos en ambas direcciones, como se muestra esquemáticamente en la Ilustración 5.

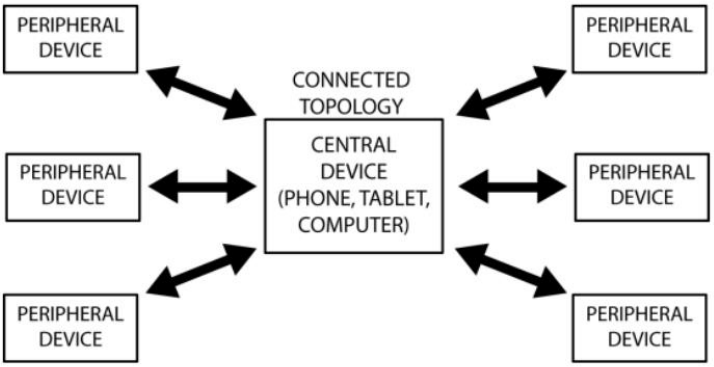


Ilustración 5: Topología conexión dispositivos Central/Peripheral [8].

2.2 Pila del protocolo BLE

En la Ilustración 6 se representa la arquitectura de BLE, que se divide en tres bloques principales: Aplicación, *Host* y Controlador [4].

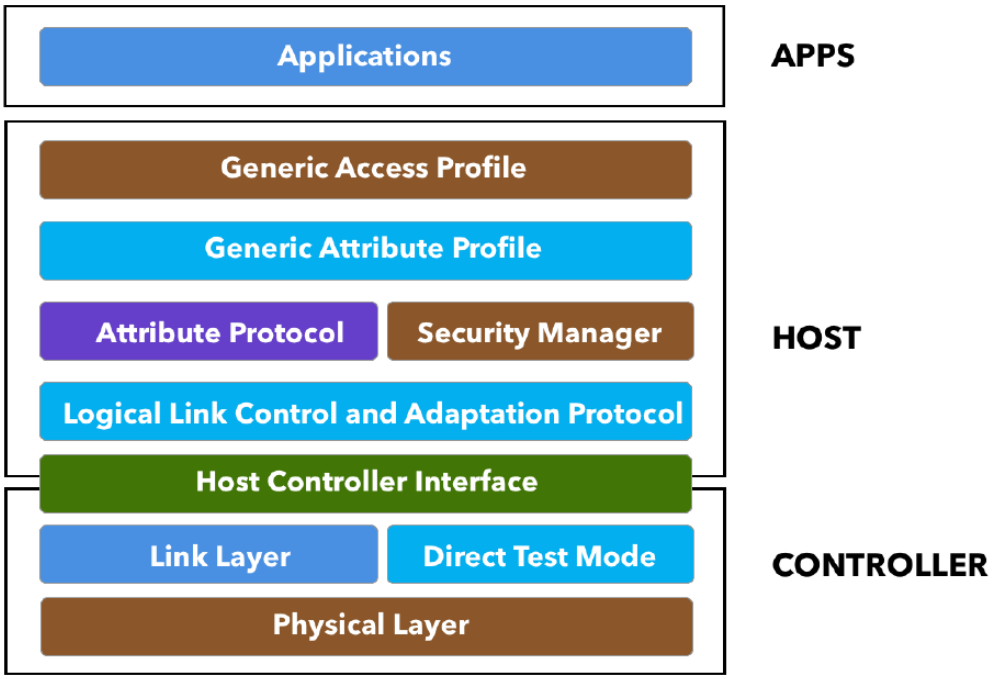


Ilustración 6: Pila del protocolo BLE 5[4].

Inicialmente se exponen las diferentes capas de forma general, ya que a lo largo del capítulo se estudiarán en mayor profundidad:

- La capa Aplicación (APP), resuelve cómo se gestionan los datos recibidos y enviados a otros dispositivos, así como la funcionalidad de este proceso.

- La capa *Host* contiene las siguientes capas, que serán descritas con mayor detalle a lo largo del presente capítulo:

    - *Generic Access Profile (GAP).*
    - *Generic Attribute Profile (GATT).*
    - *Attribute Protocol (ATT).*
    - *Security Manager (SM).*
    - *Logical Link Control and Adaptation Protocol (L2CAP).*
    - *Host Controller Interface* (*HCI*).

- La capa *Controller* representa el dispositivo físico que permite transmitir y recibir señales de radio e interpretarlas como paquetes de información. Esta parte de la pila contiene los siguientes elementos:

    - *Physical Layer (PHY).*
    - *Link Layer.*
    - *Direct Test Mode.*
    - *Host Controller Interface (HCI).*

2.2.1 Physical layer (PHY)

Cuando se habla de la capa física en este ámbito, se hace referencia al hardware radio usado para la comunicación/modulación/de-modulacion de datos. BLE opera en la banda *ISM (*2.4 GHz en el espectro) diferenciándose en 40 canales, separados cada 2 MHz desde 2.4002 MHz hasta 2.480 MHz, tal y como se muestra en la Ilustración 7 [4].

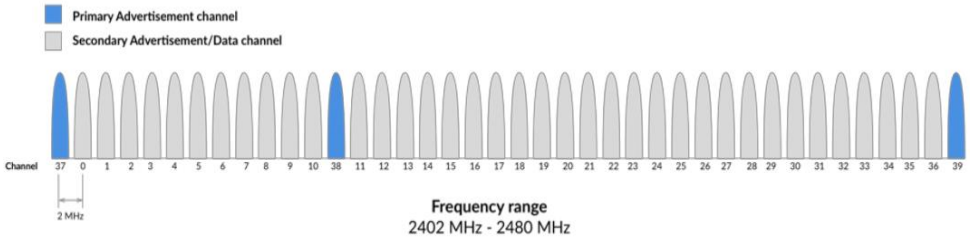


Ilustración 7: Rango de frecuencia usado en BLE 5 [4].

Entre todos los canales, tres de ellos se denominan "*Primary Advertising Channel"* y los restantes se identifican como *"Secondary Advertisements".* Como se ha mencionado con anterioridad, *Advertising* es el modo que tiene este protocolo para iniciar el establecimiento de una conexión.

El proceso de *Advertising* dará comienzo con el envío de paquetes *Advertising* en los tres canales principales (*Primary Advertising Channel)*. Esto permite la detección de un dispositivo para leer sus datos de *Advertising*. A continuación, el dispositivo *Scanner* puede iniciar una conexión si el dispositivo *Advertiser* lo permite. También puede solicitar un *scan request* (solicitud de escaneo) y si el dispositivo *Advertiser* admite esta función, responderá con un *scan response* (respuesta de escaneo*)*. Las solicitudes de escaneo y las respuestas de escaneo permiten al dispositivo *Advertiser* enviar datos de *Advertising* a dispositivos que estén interesados en recibirlos.

Por otro lado, es importante destacar otros aspectos técnicos relacionados con la física de la radio:

- Utiliza *Frequency Hopping Spread Spectrum (FHSS),* que habilita en los dos dispositivos conectados cambiar a frecuencias seleccionas al azar *agreed-on (*acordadas) para intercambiar datos. Permite así, evitar canales de frecuencia que pueda estar congestionados, o en uso por otros dispositivos. Al tratarse de la banda ISM, esto podría llegar a ser un problema.

- Los niveles de potencia de transmisión son:
  - Máximo: 100 mW (+20 dBm) para la versión> = 5, 10 mW (+10 dBm) para la versión <= 4.2.
  - Mínimo: 0,01 mW (-20 dBm).

- *Bluetooth* 5 agrega dos nuevas variantes a la especificación PHY utilizada en *Bluetooth* 4. Cada variante PHY tiene sus propias características, siendo su denominación LE 1M, LE 2M y *LE Coded* [9].
  - *LE 1M* es la PHY utilizada en *Bluetooth* 4. Utiliza la codificación por desplazamiento de frecuencia gaussiana y tiene una velocidad de símbolo de 1 M/S. Esto se correlaciona con una tasa de bits de 1 Mb / s, ya que un símbolo corresponde a un bit de datos. *LE 1M* sigue estando disponible para su uso en *Bluetooth* 5 y, de hecho, su soporte es obligatorio.

- *LE 2M PHY* permite que la capa física opere a 2 Ms / s y por lo tanto permite velocidades de datos más altas que *LE 1M* y *Bluetooth* 4. Su soporte en una implementación de la pila es opcional.

- *LE Coded PHY* permite obtener un alcance cuatro veces mayor, aproximadamente, en comparación con *Bluetooth* 4, y sin aumentar la potencia de transmisión requerida.

2.2.2 *Link Layer* (LL)

Link Layer (Capa de Enlace), es la capa que interactúa con la Capa Física (PHY), así como una forma de comunicarse con la radio a través de *HCI* (*Host Controller Interface*, una capa de nivel superior). Es la responsable de gestionar el estado de la radio, así como los requisitos de tiempos necesarios para satisfacer la especificación BLE. También es responsable de administrar las operaciones aceleradas por hardware como el CRC (*Cyclic Redundancy Code*) [4].

Los estados principales en los cuales operan los dispositivos BLE son:

- *Advertising state.*
- *Scanning state.*
- *Connected state.*

Esta capa gestiona diferentes estados de la radio, clasificados tal y como se muestra en la Ilustración 8.

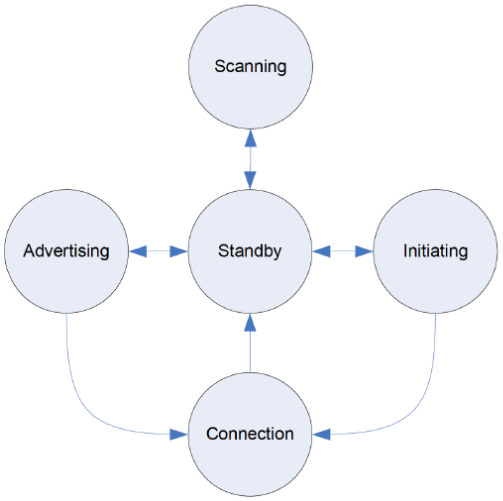


Ilustración 8: Estados capa LL [8].

- *Advertising*: la Capa de Enlace de los dispositivos de tipo *Peripheral* transmiten paquetes de *Advertising* en los canales de *Advertising*. En este estado también se puede escuchar a los dispositivos de tipo *Central* que responden a los paquetes de *Advertising,* y responder a esos dispositivos. A este estado se puede ingresar desde el estado *Standby*, cuando la Capa de Enlace decide iniciar el proceso de *Advertising*.

- *Scanning*: la Capa de Enlace del dispositivo *Central* escucha los paquetes de *Advertising* enviados por el dispositivo *Advertiser* a través de los canales asignados, pudiendo solicitarle que proporcione información adicional con el fin de explorar los dispositivos BLE existentes. A este estado se puede entrar desde el estado *Standby,* cuando la Capa de Enlace decide comenzar el proceso de *Scanning*. Se denomina *Advertiser* al dispositivo BLE que utiliza los canales de *Advertising* para anunciar que es conectable y detectable, o que puede ser descubierto. A un dispositivo en estado de *Scanning* se le denomina *Scanner.*

- *Initiating*: La Capa de Enlace escucha los paquetes recibidos desde el dispositivo *Advertiser* y responde a ellos iniciando una conexión. En general, este es el estado en el que entra el dispositivo *Central* antes de pasar al estado de conexión. A este estado se puede pasar desde el estado *Standby,* cuando el dispositivo *Scanner* decide iniciar una conexión con el dispositivo *Advertiser*, actuando como dispositivo *Initiator.*

- *Connection*: El dispositivo *Central* está conectado a otro dispositivo *Peripheral*, definiéndose los roles de *Master* y *Slave*. A este estado se puede pasar desde el estado *Initiating*, o desde el estado *Advertising.* Cuando se ingresa desde el estado *Initiating*, el dispositivo actúa como *Master*, mientras que cuando se entra desde el estado *Advertising*, el dispositivo actúa como *Slave* en el intercambio periódico de datos. Los canales de datos se utilizan una vez establecida la conexión entre ambos dispositivos.
- *Standby*: Este es el estado predeterminado de la Capa de Enlace. En este estado no se reciben ni se transmiten paquetes. Un dispositivo puede entrar en este estado desde cualquiera de los otros estados [7].

2.2.3 *Host* Control Interface (HCI)

HCI es un protocolo estándar definido por la especificación *Bluetooth* que permite a la capa *Host* comunicarse con la capa *Controller.* Estas capas pueden existir *on chip.* En el caso de estar en chips separados, la interfaz *HCI* se implementarará sobre la interfaz de comunicación física. Oficialmente, las tres interfaces hardware compatibles con *Bluetooth* son: UART, USB y SDIO (*Secure Digital Input Ouput*). En el

caso donde las dos capas (*Host* y *Controller)* conviven en el mismo *chipset*, la interfaz *HCI* actuará en lugar de interfaz lógica y no mediante hardware [4].

La función de la capa *HCI* es transmitir comandos desde el *Host* al Controlador, y enviar copia de seguridad de los eventos desde el Controlador al *Host*. En la Ilustración 9 se muestra el ejemplo de una captura de comandos *HCI*, eventos *HCI* y comandos *ATT* que se intercambian entre el *Host* y capas del Controlador.

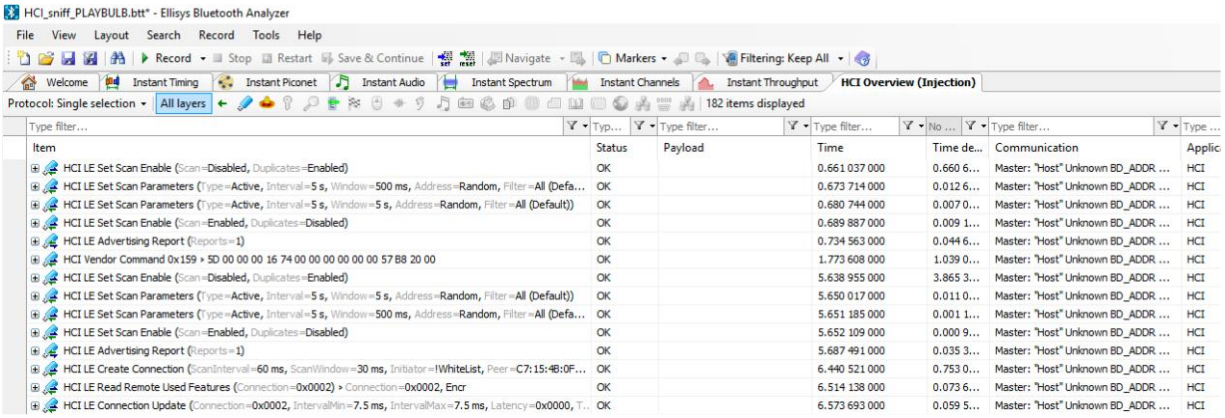


Ilustración 9: Parámetros de conexión HCI [7].

Como se puede observar en la Ilustración 9, algunos comandos *HCI* están relacionados con la configuración y la actualización de los parámetros de conexión.

2.2.4 Logical Link Control and Adaptation Protocol (*L2CAP*)

L2CAP actúa como capa de multiplexación entre capas. Es una prestación de *Bluetooth Classic Standard, que realiza* las siguientes funciones BLE [4]:

- Toma múltiples protocolos de capas superiores, y los coloca en paquetes *BLE* estándar, que se transmiten a las capas inferiores.
- Gestiona la fragmentación y el reensamblado de paquetes. Toma los paquetes más grandes de capas superiores y los divide en trozos que se ajustan al tamaño máximo de carga útil BLE admitido para su transmisión. En el lado del receptor, toma varios paquetes y los combina en un paquete que puede ser gestionado por las capas superiores.

Para BLE, la capa *L2CAP* gestiona dos protocolos principales: *Attribute Protocol* (*ATT*) y *Security Manager Protocol* (*SMP*). Desde el punto de vista del desarrollador de la aplicación, es importante tener

en cuenta que la cabecera del paquete *L2C*AP ocupa cuatro bytes, lo que significa que la longitud efectiva por defecto de los datos de usuario es 27 - 4 = 23 bytes (siendo 27 bytes el tamaño de carga útil de la Capa de Enlace) [8].

2.2.5 Security Manager Protocol (*SMP*)

SMP es tanto un protocolo como una serie de algoritmos de seguridad diseñados para proporcionar fiabilidad a la pila de protocolos *Bluetooth*. Permite adquirir la capacidad de generar e intercambiar claves de seguridad para realizar emparejamientos de forma segura a través de un enlace cifrado. Este protocolo define dos roles [4]:

- Rol *initiator*, en este protocolo le corresponde al dispositivo *Central.*
- Rol *responser,* en BLE le corresponde al dispositivo *Peripheral.*

2.3 Servicios, Características, Perfiles y Atributos

Servicios, Características, Perfiles y Atributos son los cuatro conceptos principales para comprender la forma en que se estructuran los datos BLE. Estos conceptos se usan específicamente para establecer una estructura de datos jerárquica, como se muestra de forma simbólica en la Ilustración 10 [4].

2.3.1 Servicios

Un Servicio es una agrupación de uno o más Atributos, los cuales tienen alguna Característica, o no, si estos son declaraciones. Su estructura se representa en Ilustración 10. Está destinado a agrupar Atributos relacionados que satisfacen una funcionalidad específica en el *Servidor*. Existen dos tipos de Servicios:

a) Servicios primarios, representan las funcionalidades primarias del dispositivo
b) Servicios secundarios, aporta funcionalidades auxiliares del dispositivo que suelen ser referencias (*includes*) para al menos otro Servicio primario del dispositivo.

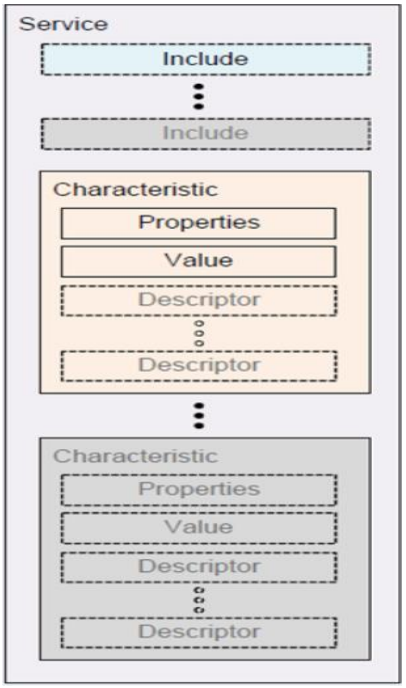


Ilustración 10: Jerarquía ATT [4].

2.3.2 Características

Una Característica es siempre parte de un Servicio, representando un dato/información que un *Servidor* quiere exponer a un *Cliente*. Las Características pueden contener otros Atributos que ayuden a definir los valores que contienen, como se representa en la Ilustración 11:

- Propiedades, representadas por bits, y que definen cómo debe usarse un valor que contiene la Característica. Algunos ejemplos de propiedades son: lectura, escritura, notificación, etc.
- Descripción, que contienen información relativa la Característica, además de definir el valor, la unidad y el formato que se usa para representarlo.
- Valor que representa, es un Atributo completo que contiene los datos de usuario en su campo de valor.

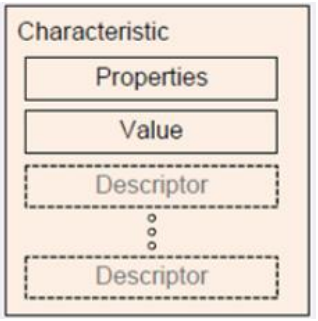


Ilustración 11: Jerarquía ATT, Característica [4].

2.3.3 Atributos

Los Atributos son la entidad de datos más pequeña definida, representando segmentos direccionables de información que pueden contener datos de usuario o metadatos. Tanto la capa G*ATT*, como *ATT* (que se describirán a lo largo del capítulo) pueden operar solo con Atributos, por lo que para que los *Clientes* y *Servidor*es interactúen, toda la información debe estar organizada y ajustada al protocolo BLE.

2.3.4 Attribute Protocol (*ATT*)

El protocolo *ATT* define la forma en la que el *Servidor* expone los datos al *Cliente* (Ilustración 12) y cómo se estructuran. Los datos están estructurados de acuerdo con una arquitectura *Cliente-Servidor*, permitiendo el intercambio de información. Los datos que gestiona el *Servidor* se denominan Atributos. Para configurar un Atributo se debe seguir la trama definida en la Ilustración 12 [4]:

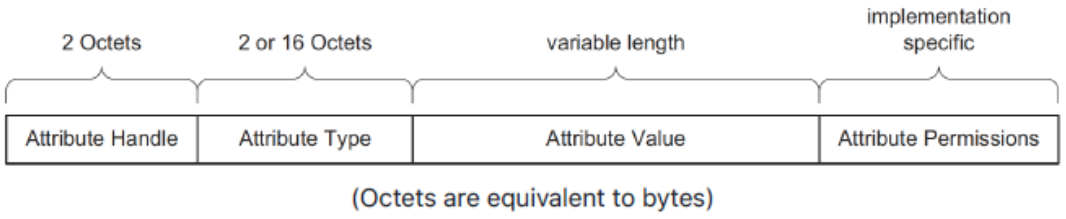


Ilustración 12: Trama de un Atributo [4].

- *Attribute Handle*: Representa un identificador único de 16 bits para cada Atributo en un *Servidor* G*ATT* particular. Es la parte de cada Atributo que hace que sea direccionable, y se garantiza que no cambiará entre transacciones o a través de las conexiones. El valor 0x0000 denota un identificador no válido, y la cantidad de identificadores disponibles para cada *Servidor* G*ATT* es 0xFFFE (65535), aunque en la práctica, el número de Atributos en un *Servidor* suele ser más cercano a unas pocas docenas. El *Cliente* debe usar la función de descubrimiento para obtener los identificadores de los Atributos de interés.
- *Attribute type*: Se corresponde con el tipo del Atributo, que no es más que un UUID. Puede ser un UUID de 16, 32 o 128 bits, ocupando 2, 4 o 16 bytes, respectivamente. Determina el tipo de datos presentes en el valor del Atributo, y hay mecanismos disponibles para descubrir Atributos basados exclusivamente en su tipo.
- *Attribute Permissions* : Los Permisos son metadatos que especifican qué operaciones *ATT* se pueden ejecutar en cada Atributo particular, y con qué requisitos de seguridad específicos. *ATT* y

G*ATT* definen los siguientes permisos, determinando si el *Cliente* puede leer o escribir (o ambos) un valor de Atributo. Cada Atributo puede tener uno de los siguientes permisos de acceso:

- *None*: El Atributo no puede ser leído ni escrito por un *Cliente*.
- *Readable*: El Atributo puede ser leído por un *Cliente*.
- *Writable*: El Atributo puede ser escrito por un *Cliente*.
- *Readable* and *Writable*: El Atributo puede ser leído y escrito por un *Cliente*. Determina si se requiere de un cierto nivel de cifrado para que el *Cliente* pueda acceder al Atributo. Estos son los permisos de cifrado permitidos, según los define G*ATT*:
  - *No encryption required*: El Atributo es accesible en una conexión de texto sin cifrar.
  - *Unauthenticated encryption required* (Security Mode 1, Level 2): La conexión debe cifrarse para acceder al Atributo, pero las claves de cifrado no necesitan ser autenticadas (aunque pueden serlo).
  - *Authenticated encryption required (Security Mode 1, Level 3*): La conexión debe cifrarse con una clave autenticada para acceder al Atributo.
  - *Authorization*: Determina si se requiere permiso del usuario para acceder al Atributo. Un Atributo puede elegir entre requerir o no requerir de autorización:
  - *No authorization required*: El acceso al Atributo no requiere de autorización.
  - *Authorization required:* El acceso al Atributo requiere de autorización.

Cuando un *Cliente* desea leer o escribir valores de/en los Atributos, desde o hacia un *Servidor*, emite una solicitud de lectura o escritura al *Servidor* a través del Controlador. 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 e interpretar el tipo de datos en función del UUID del Atributo. Por otro lado, durante una operación de escritura, se espera a que el *Cliente* proporcione datos que sean consistentes con el tipo de Atributo, pudiendo el *Servidor* rechazar la operación si no es ese el caso [7].

2.3.5 Perfiles

Los Perfiles abarcan la amplitud del protocolo, definiendo el comportamiento, tanto del *Cliente* como del *Servidor,* en cuanto a Servicios, Características e incluso conexiones y requisitos de seguridad. Además, llevan a cargo la implementación de estos Servicios y Características únicamente en el lado del *Servidor*. En una especificación de perfil se encuentra [4]:

- Definición de roles y relación entre el *Servidor* y el *Cliente* del G*ATT*.
- Servicios requeridos.
- Requisitos de Servicio.

- Cómo se utilizan los Servicios y Características.
- Detalles de los requisitos de establecimiento de conexión, incluido el proceso de *Advertising* y los parámetros de conexión.
- Consideraciones de Seguridad.

2.3.6 Generic Attribute Profile (G*ATT*)

El perfil *GATT* se construye sobre *ATT* y constituye un modelo de abstracción de datos. Se podrían considerar los cimientos de la transferencia de datos en BLE, ya que define cómo deben estar organizados los datos entre aplicaciones [4]. En este perfil, los dispositivos BLE pueden adquirir varios roles:

- *Cliente*, que envía solicitudes a un *Servidor* y recibe respuestas. El *Cliente* G*ATT* inicialmente no conoce los Atributos del *Servidor*, por lo que primero debe consultar acerca de la presencia y la naturaleza de estos, realizando para ello el proceso de descubrimiento de Servicios. Una vez completado este descubrimiento, comienza a leer y escribir los Atributos encontrados en el *Servidor*, así como a recibir actualizaciones iniciadas por el *Servidor*.

- *Servidor*, que recibe solicitudes de un *Cliente* y retorna la respuesta. También envía actualizaciones iniciadas por el *Servidor* cuando está 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* G*ATT* básico que pueda responder a solicitudes de los *Cliente*s, aunque solo sea para devolver una respuesta de error.

En un *Servidor* G*ATT*, los Atributos se agrupan en Servicios, cada uno de los cuales puede contener, o no, Características. Estas Características, a su vez, pueden incluir Descriptores. Esta jerarquía se aplica estrictamente a cualquier dispositivo BLE, lo que significa que todos los Atributos en un *Servidor* G*ATT* se incluirán en una de estas tres categorías, sin excepción. No puede existir ningún Atributo fuera de esta jerarquía, ya que el intercambio de datos entre dispositivos BLE depende de ello, tal y como se muestra en la Ilustración 13.

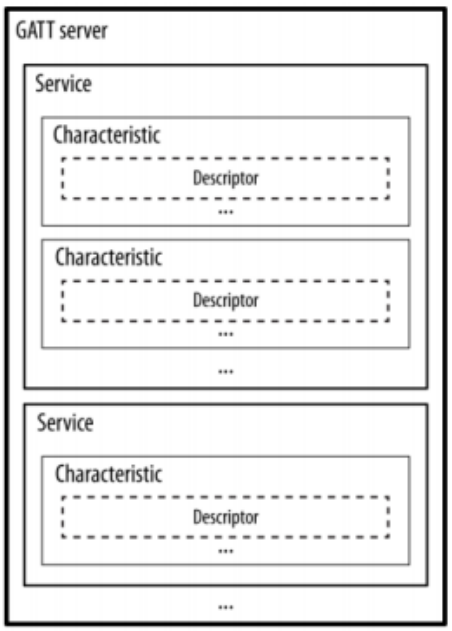


Ilustración 13: GATT Servidor [4].

2.3.7 Tipos de operaciones de Características

Las Características se pueden considerar como contenedores para los datos de usuario. Siempre incluyen al menos dos Atributos: Declaración de Característica (que proporciona metadatos sobre los datos de usuarios actuales) y el Valor de Característica (que es un Atributo). Además, el Valor de Característica puede ir seguido de Descriptores, que amplían aún más los metadatos contenidos en la Declaración de Característica. La Declaración, el Valor y los Descriptores forman la Definición de Característica.

El Atributo de Declaración de Característica es un UUID único y estandarizado de valor 0x2803 (descrito en la Ilustración 14) que se utiliza exclusivamente para indicar el comienzo de las Características. Este Atributo tiene permisos de solo lectura, ya que los *Clientes* solo pueden recuperar su valor, pero en ningún caso modificarlo.

Handle	Type	Permissions	Value	Value length
0xNNNN	$UUID_{characteristic}$	Read only	Properties, value handle (0xMMMM), characteristic UUID	5, 7, or 19 bytes
0xMMMM	Characteristic UUID	Any	Actual value	Variable

Ilustración 14: Declaración y valor de una Característica [8].

En la Ilustración 15, se describen los diferentes elementos que conforman una Característica, que son:

Name	Length in bytes	Description
Characteristic Properties	1	A bitfield listing the permitted operations on this characteristic
Characteristic Value Handle	2	The handle of the attribute containing the characteristic value
Characteristic UUID	2, 4, or 16	The UUID for this particular characteristic

Ilustración 15: Característica, propiedades y valor [8].

Characteristic Value Handle: El campo *Characteristic Value Handle* representa el identificador del Atributo que contiene el valor actual de la Característica.

Characteristic Properties: Este campo de 8 bits contiene la operación y el procedimiento que debe usarse con las Características. Son 9 tipos distintos y se codifican como un simple bit, tal como se enumera en la Tabla 1.

Tabla 1: Tipo de propiedades de las Características.

Propiedad	Descripción
Broadcast	Si se establece, permite colocar este valor de Característica en los paquetes publicitarios, utilizando el tipo AD de datos de Servicio
Read	Si se establece, permite a los clientes leer esta característica utilizando cualquiera de las operaciones de lectura ATT enumeradas
Write without response	Si se establece, permite a los clientes utilizar la operación Write Command ATT en esta característica
Write	Si se establece, permite a los clientes utilizar la operación Write Request/Response ATT en esta característica
Notify	Si se establece, permite que el servidor utilice la operación ATT de notificación del valor Handle en esta característica
Indicate	Si se establece, permite que el servidor utilice la operación ATT de indicación/confirmación del valor del Handle en esta característica
Signed Write Command	Si se establece, permite a los clientes utilizar la operación ATT de escritura firmada en esta característica
Queued Write	Si se establece, permite a los clientes utilizar las operaciones ATT de escritura en cola en esta característica
Writable Auxiliaries	Si se establece, un cliente puede escribir en el descriptor

El *Cliente* puede leer esas propiedades para averiguar qué operaciones puede realizar en cada Característica. Esto es particularmente importante para las propiedades Notificar e Indicar, porque estas operaciones son iniciadas por el *Servidor*.

Characteristic UUID: El UUID de la Característica puede ser un UUID aprobado por SIG (al hacer uso de las docenas de tipos de características incluidos en los perfiles estándar) o un UUID específico del proveedor, de 128 bits.

La estructura jerárquica de la unión entre Servicios y Características se muestra en la Ilustración 16.

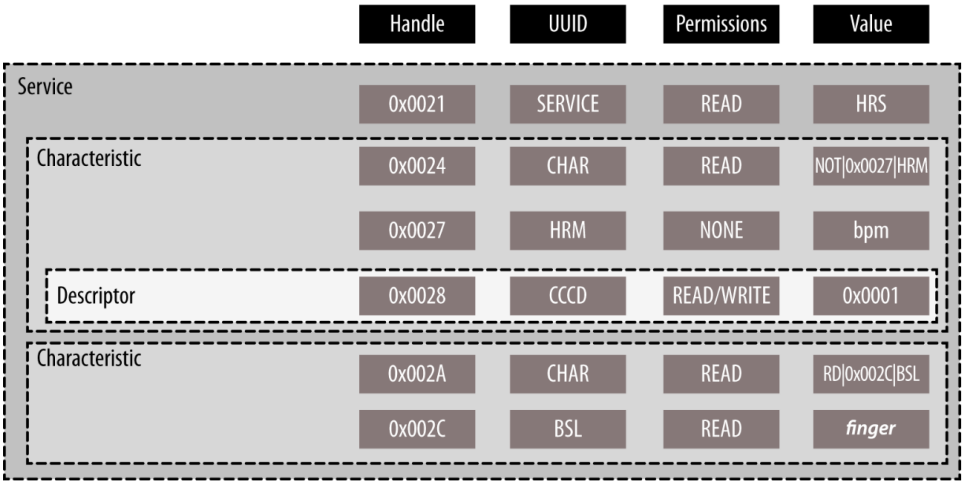


Ilustración 16: Estructura jerárquica de una característica [8].

2.3.8 Generic Access Profile (GAP)

El *Perfil de Acceso Genérico* determina el *framework* o la forma en la que los dispositivos BLE deben interactuar con otros. Incluye los siguientes aspectos:

- Modelo y roles de los dispositivos.
- *Advertisements.*
- Cómo se establece una conexión (iniciación, aceptación y parámetros de conexión).
- Seguridad.

La implementación de este entorno es obligatoria para la especificación oficial y permite a dos o más dispositivos *Bluetooth* interoperar entre sí, y comunicarse, así como la posibilidad de realizar intercambio de datos con otros dispositivos. En GAP, existen cuatro roles que un dispositivo BLE puede adoptar para unirse a una red:

- *Broadcaster*, optimizado para aplicaciones de solo transmisión, que distribuyen datos regularmente. Se envían periódicamente paquetes de *Advertising* con datos, y los datos son accesibles para cualquier dispositivo que esté escuchando. El rol *Broadcaster* utiliza el rol de *Advertiser* de la Capa de Enlace.

- *Observer,* optimizado para aplicaciones de solo recepción, que tienen como objetivo recopilar datos de dispositivos de *Broadcasting.* El rol de *Observer* utiliza el rol *Scanner* de la Capa de Enlace.

- *Central,* que corresponde al *Master* de la Capa de Enlace. Un dispositivo capaz de establecer múltiples conexiones con otros dispositivos. El rol *Central* es siempre el iniciador de las conexiones y esencialmente permite que los dispositivos entren en la red. El protocolo BLE es asimétrico, lo

que significa que los requisitos del *Master* de la Capa de Enlace son mayores que los de un *Slave*, por lo que el rol *Central* generalmente lo desempeña un *smartphone* o tableta, ya que tiene acceso a CPU potentes y recursos de almacenamiento. Esto le permite mantener conexiones con múltiples dispositivos. El dispositivo *Central* comienza por escuchar los paquetes de *Advertising* de otros dispositivos y luego inicia una conexión con el dispositivo seleccionado. Este proceso se puede repetir para incluir múltiples dispositivos en una sola red.

- *Peripheral,* que corresponde al dispositivo *Slave* de la Capa de Enlace. Emplea paquetes de *Advertising* para permitir que los dispositivos *Central* lo encuentren y posteriormente, establecer una conexión con ellos. El protocolo BLE está optimizado para requerir pocos recursos para la implementación del dispositivo *Peripheral,* al menos en términos de potencia de procesamiento y memoria.

Cada dispositivo particular puede operar en uno o más roles a la vez, y la especificación BLE no impone restricciones a este respecto. Muchos desarrolladores intentan asociar erróneamente los roles de *Cliente* y *Servidor* G*ATT* con los roles GAP.

2.4 Conexión y transferencia de datos del protocolo BLE

Una vez comprendidos los conceptos básicos del protocolo BLE, a lo largo de este apartado se recoge todo lo que concierne a las conexiones, tiempos y direcciones que definen el protocolo *Bluetooth Low Energy*.

2.4.1 Tipos de direcciones

El identificador fundamental de un dispositivo *Bluetooth* es similar a la dirección de acceso a medios Ethernet (MAC). Lo conforman 48 bits (6 bytes), e identifica de forma única un dispositivo entre pares. Existen dos tipos principales de direcciones: direcciones públicas y direcciones aleatorias [4].

- Dirección publica: Esta es una dirección fija que no cambia y viene programada de fábrica. Debe ser registrada con el IEEE (similar a la dirección MAC de un dispositivo WiFi o Ethernet).

- Dirección aleatoria: Dado que los fabricantes pueden elegir qué tipo de dirección utilizar (aleatoria o pública), las direcciones aleatorias son más usadas, ya que no requieren registro en el IEEE. La

dirección aleatoria se programa en el dispositivo o se genera en tiempo de ejecución. Puede ser uno de dos subtipos:

- Dirección estática: Se utiliza como reemplazo de direcciones públicas. Se puede generar en el arranque o permanecer inalterable durante toda la vida útil.

- Dirección privada: Se divide en dos subtipos adicionales:

  - Dirección privada irresoluble, aleatoria, temporal por un tiempo determinado. No se usa comúnmente.

  - Dirección privada resoluble, se usa para garantizar la privacidad. Son generadas utilizando la clave de resolución de identidad (IRK) y un número aleatorio. Cambia periódicamente (incluso durante la vida útil de la conexión) para evitar ser rastreado por dispositivos desconocidos. Solo dispositivos de confianza podrán resolverlo utilizando el IRK almacenado previamente.

2.4.2 UIID

Un identificador único universal (UUID) es un número de 128 bits (16 bytes) que está garantizado (o tiene una alta probabilidad) de ser globalmente único. Los UUID se utilizan en muchos protocolos y aplicaciones distintas de *Bluetooth*.

Para mayor eficiencia, y debido a que 16 bytes tomarían una gran parte de la longitud de carga útil de datos de 27 bytes la típica de la Capa de Enlace, la especificación BLE define dos formatos de UUID adicionales: UUID de 16 bits y de 32 bits. Estos formatos abreviados solo se pueden usar con UUID que se definen en la especificación de *Bluetooth* (es decir, que se enumeran en *Bluetooth* SIG como UUID de *Bluetooth* estándar) [10].

2.4.3 Paquetes y parámetros de *Advertising y Scanning*

BLE tiene un único formato de paquetes, y tan solo dos tipos de ellos, lo cual simplifica enormemente la implementación de la pila de protocolos.

Paquetes de *Advertising.*

Los paquetes de *Advertising* sirven para dos propósitos principales, transmitir datos para aplicaciones en el establecimiento de una conexión completa sin necesidad de cabecera, y para descubrir dispositivos *Slave* habilitando la posibilidad de conectarse a ellos. Cada paquete de *Advertising* soporta hasta 31 bytes de *playload* en la versión 4.0 y 4.1. Desde la versión 4.2 esta longitud será variable, y negociada mediante DEL (*Data Length Extension*). Tales paquetes son simplemente transmitidos a ciegas por el dispositivo *Advertiser* sin el conocimiento previo de la presencia de cualquier dispositivo *Scanner*. Se envían a una tasa fija definida por el parámetro *Advertising Interval*. Estos paquetes se envían tal y como se muestra en la Ilustración 17.

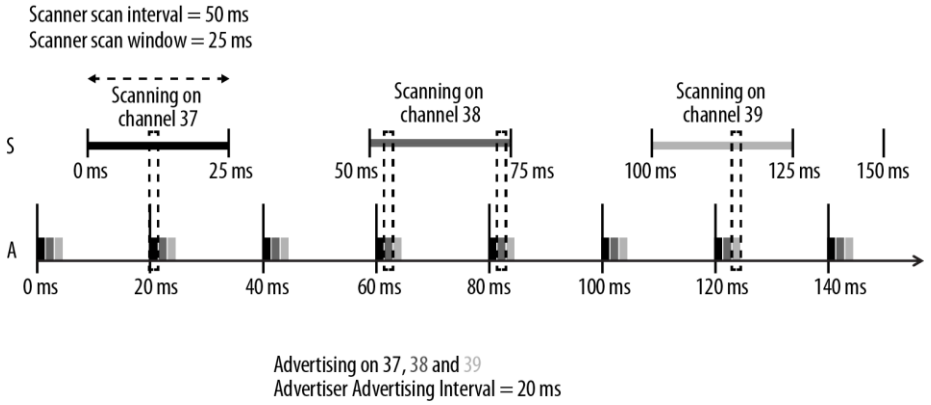


Ilustración 17: Paquetes y parámetros Advertising [8].

Parámetros de *Advertising:*

- *Advertising Interval, c*uanto menor es el valor de este intervalo de tiempo, mayor es la frecuencia con la que se emiten los paquetes de *Advertising,* lo que aumenta la probabilidad de que esos paquetes sean recibidos por un dispositivo *Scanner*, pero también se incurre en un mayor consumo de energía. Debido a que el proceso de *Advertising* emplea un máximo de tres canales de frecuencia, y el dispositivo *Advertiser* y el dispositivo *Scanner* no están sincronizados de ninguna manera, de forma que el dispositivo *Scanner* solo recibirá un paquete de *Advertising* cuando se solapen aleatoriamente.
- Datos de *Advertising:* siguiendo el formato de organización *TLV* (*Type-Length-Value*), los datos de *Advertising* entran en la porción de PDU del paquete BLE y contienen los siguientes campos:
  - Tipo de dato de *Advertising (AD Type)*, el tipo de A*dvertisement*, e incluye la especificación TLV que se puede observar en la Ilustración 18.
  - Longitud, la longitud de los datos que siguen al valor de longitud en sí (incluye el tipo de AD, así como los datos de AD).
  - Datos de *Advertising*, el valor actual que contiene.

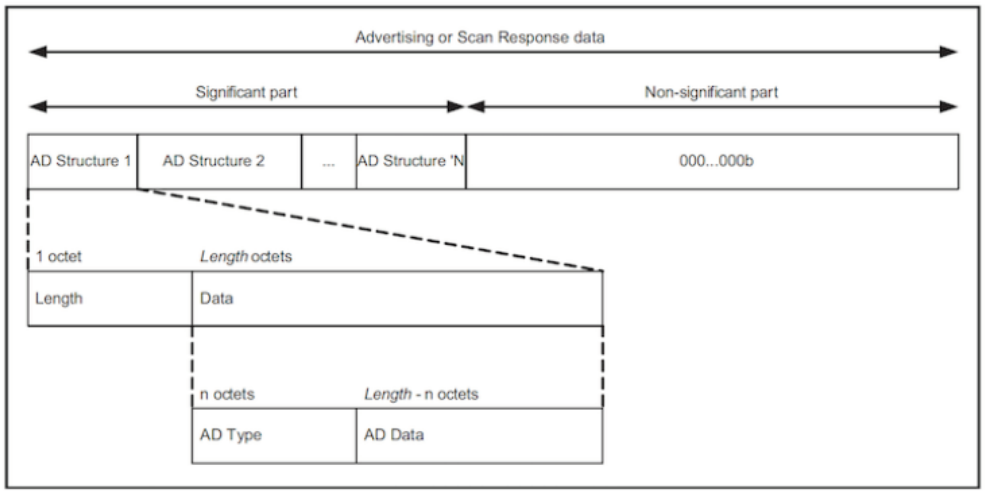


Ilustración 18: Especificación TLV [8].

Parámetros de *Scanning* (Ilustración 19):

Lo principales parámetros de *Scanning* son:

- *Scan type,* pasivo o activo.
- *Scanning Window,* indican cómo de largo será el escaneo de *Advertisements.*
- *Scanning Interval* indica cada cuanto se hará el escaneo de *Advertisements.*

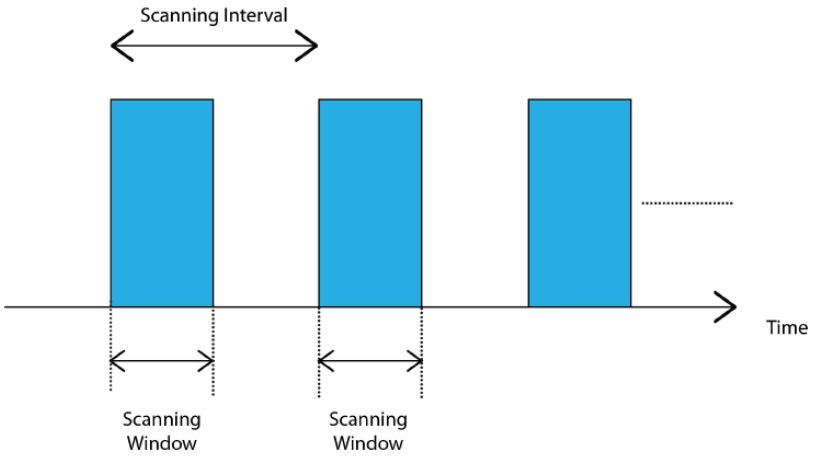


Ilustración 19: Parámetros de Scanning [4].

Los parámetros *Scanner Scan Interval y Scanner Scan Window* definen con qué frecuencia y durante cuánto tiempo un dispositivo *Scanner* escuchará posibles paquetes de *Advertising*, respectivamente. Al igual que con el parámetro *Advertising Interval*, estos valores tienen un gran impacto en el consumo de energía.

La especificación *BLE* define dos tipos básicos de procedimientos de *Scanning (Scan type):*

- *Passive Scanning: e*l dispositivo *Scanner* simplemente escucha paquetes de *Advertising.*

- *Active Scanning*, el dispositivo *Scanner* emite un paquete de solicitud de *Scanning (Scan Request)* después de recibir un paquete de *Advertising.* El dispositivo *Advertiser* lo recibe y responde con un paquete de respuesta de *Scanning (Scan Reponse Data).* En la Ilustración 20 se observan las diferencias entre ellos.

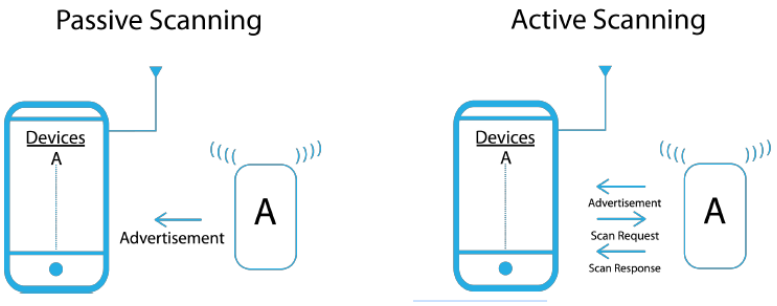


Ilustración 20: Passive Scanning vs Active Scanning [4].

2.4.4 Proceso de conexión

Para establecer una conexión, un dispositivo *Scanner* comienza en primer lugar a buscar dispositivos *Advertiser* que acepten solicitudes de conexión. Cuando detecta un dispositivo *Advertiser* adecuado, el dispositivo *Scanner* envía un paquete de solicitud de conexión (*Connect Request*) al dispositivo *Advertiser,* y siempre que éste responda, establece una conexión. El paquete de solicitud de conexión incluye el incremento de salto de frecuencia, que determina la secuencia de salto que seguirán los dispositivos *Master* y *Slave* durante la vida útil de la conexión. Una conexión es simplemente una secuencia de intercambio de paquetes de datos entre los dispositivos *Slave* y *Master* en tiempos predefinidos, como se muestra en la Ilustración 21, denominándose cada intercambio Evento de Conexión, (*Connection Event*). Además, por defecto, en cada Evento de Conexión, ambos dispositivos transmiten un paquete, aunque no tengan datos que enviarse, denominado *Empty Link Layer* PDU, con el fin de garantizar que la conexión aún está activa [7].

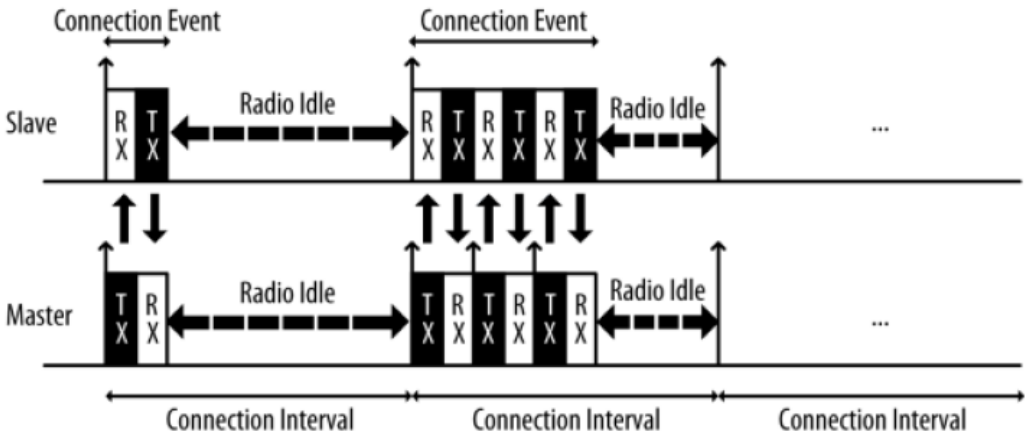


Ilustración 21: Evento de Conexión [8].

Durante el establecimiento de una conexión BLE, el dispositivo *Master* comunica al dispositivo *Slave* los siguientes parámetros:

- Intervalo de Conexión (*Connection Interval*): Tiempo entre el inicio de dos Eventos de Conexión consecutivos.

- Latencia del *Slave* (*Slave Latency*): Número de Eventos de Conexión que un dispositivo *Slave* puede elegir ignorar sin poner en riesgo la conexión.

- Tiempo de Espera de Supervisión de Conexión (*Connection Supervision Timeout*): Tiempo máximo entre dos paquetes de datos válidos recibidos, antes de que una conexión se considere perdida.

Todos los paquetes recibidos se comparan con un CRC (*Cyclic Redundancy Check*) de 24 bits, y se solicitan retransmisiones cuando la comprobación de errores detecta un error en la transmisión, sin haber un límite superior para el número de retransmisiones. La Capa de Enlace reenviará el paquete hasta que el receptor lo reconozca finalmente. Además de los procesos de *Advertising*, *Scanning*, establecimiento (y destrucción) de conexiones, y la transmisión y recepción de datos, la Capa de Enlace también es responsable de varios procedimientos de control (actualización de parámetros de conexión y cifrado).

2.4.5 Descubrimiento de Servicios y Características

El *Cliente* no tiene conocimiento sobre los Atributos que podrían estar presentes en un *Servidor*. Los Atributos no son conocidos por el *Cliente* cuando se conecta por primera vez a un *Servidor* GATT. Es por ello, que el *Cliente* realiza desde el principio de la conexión una serie de intercambios de paquetes con el

fin de determinar la ubicación, cantidad y naturaleza de todos los Atributos. Para el descubrimiento de Servicios Primarios, el perfil G*ATT* ofrece las siguientes opciones:

- *Descubrimiento de todos los* Servicios *Primarios:* El *Cliente* puede obtener una lista de todos los Servicios principales del *Servidor* G*ATT* (independientemente del UUID de Servicios). Se usa comúnmente cuando el *Cliente* admite más de un Servicio.
- Descubrimiento del Servicio Primario, por UUID de Servicio: Siempre que el *Cliente* sepa qué Servicio busca, simplemente puede buscar todas las instancias de un Servicio particular utilizando esta función.

Cada uno de estos genera rangos de identificadores, pertenecientes a los Atributos de cada uno de los Servicios descubiertos. Si se emplea la primera opción (*Descubrimiento de todos los Servicios Primarios*), también se obtienen los UUID de cada Servicio.

Al igual que con los Servicios, el procedimiento para realizar el descubrimiento se desglosa en dos opciones:

- *Descubrimiento de todas las Características de un* Servicio*:* una vez el *Cliente* obtiene el rango de Servicios que puede ser de interés, se procede a construir una lista completa de Características. La única entrada es el rango de identificadores, y a cambio, el *Servidor* devuelve, tanto el identificador como el valor de todos los Atributos de las Características incluidas dentro de ese Servicio.
- *Descubrimiento de Características por UUID:* Este procedimiento es idéntico al de los Servicios, con la única excepción que el *Cliente* descarta todas las respuestas no coincidentes con el UUID de la característica no especificada.

Una vez que se han establecido los límites (en términos de identificadores) de una Característica objetivo, el *Cliente* puede continuar con el descubrimiento del Descriptor de la Característica:

- Descubrimiento de todos los Descriptores: El *Cliente* puede usar esta función para recuperar todos los Descriptores asociados a una Característica específica. El *Servidor* responde con una lista de UUID.

2.5 Formato de paquete BLE

La Ilustración 22 muestra la estructura de un paquete BLE.

Ilustración 22: Estructura de paquete BLE [11].

El tamaño del campo de datos depende de la especificación *Bluetooth*. En la versión 4.0 y 4.1 el tamaño máximo de este campo es de 27 bytes. Sin embargo, desde *Bluetooth* v4.2, se agregó una nueva función para intercambiar la longitud de datos de campos. Esta función agregada a la Capa de Enlace se denomina *Data length Extension* (DLE) y permite aumentar el campo de carga útil de la unidad de datos del protocolo de canal de datos (PDU) de los 27 bytes predeterminados a hasta 251 bytes. Cada paquete de Capa de Enlace tiene 14 bytes de "*overhead*": un preámbulo de 1 byte, una dirección de acceso de 4 bytes, un encabezado de PDU de canal de datos de 2 bytes, una verificación de integridad de mensajes (MIC) de 4 bytes, y una comprobación de redundancia cíclica (CRC). El campo MIC se muestra como opcional, ya que solo se usa si el cifrado está habilitado. El campo de datos en BLE 5 sigue la estructura de la Ilustración 23.

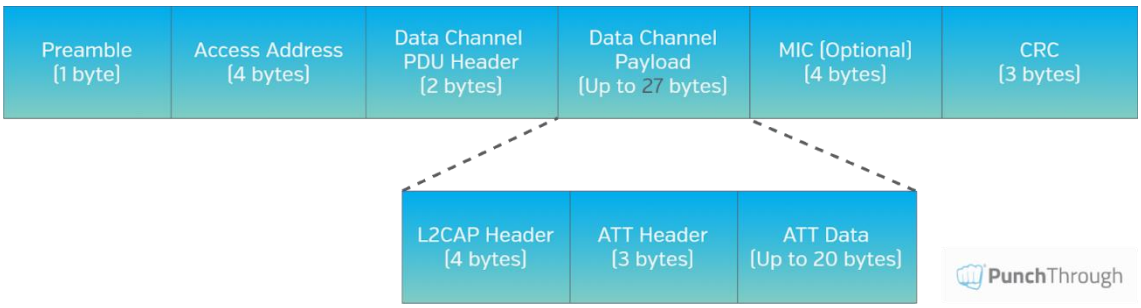


Ilustración 23: Estructura de paquetes de Capa de Enlace *Bluetooth 4.2/ 5.0* [11].

El encabezado *L2CAP* tiene un tamaño fijo (4 bytes) y se define para cumplir con sus requisitos funcionales, como el reensamblaje y la fragmentación de paquetes que sean de mayor tamaño que el campo de datos permitido, menos el tamaño del encabezado *L2CAP* de 4 bytes. Al aumentar de 27 bytes posibles (23 bytes de datos, si se restan los 4 bytes del encabezado *L2CAP* que son fijos), se debe añadir información extra de los Atributos que contiene, de ahí la cabecera de 3 bytes de encabezado para Atributos.

ATT MTU

La unidad de transmisión máxima *ATT* (MTU) es la longitud máxima de un paquete *ATT*. El parámetro *ATT* MTU está definido por *L2C*AP y su valor puede estar entre 23 e infinito. La implementación de la pila de *Bluetooth* es el factor clave para determinar el *ATT* MTU, tanto en el *Cliente* como en el *Peripheral*. Un paquete *ATT* estándar sigue la estructura de la Ilustración 24.

Ilustración 24: ATT MTU [11].

OP-Code indica la operación *ATT*, como comando de escritura, notificación, respuesta de lectura, etc. El campo de datos *ATT* contiene los datos de la aplicación [11]. Al enviar paquetes de Escritura, Lectura y Notificación o Indicación, el identificador de Atributo asociado (2 octetos) también deberá incluirse para la identificación de los datos, tal y como se muestra en Ilustración 25.

Ilustración 25: Attribute Handle [11].

Capítulo 3: Pila BLE *Cordio* Mbed

BLE *Cordio* Mbed es una solución para *Bluetooth Low Energy* (BLE) de código abierto que ofrece interfaces de abstracción para RTOS y hardware. El subsistema de *Host Cordio BLE* implementa un dispositivo BLE monomodo compatible con soluciones basadas en un único chip o en doble chip. Está compuesto por los siguientes componentes:

- Pila *Cordio*.
- Perfiles *Cordio*.

La pila *Cordio* implementa una *HCI* estándar para compatibilidad con Controladores BLE cuando se usa la configuración del sistema de doble chip. En este caso, puede utilizar conexiones con cable, como UART o SPI, externas a otro chip. La configuración de un solo chip (en otras palabras, SoC) utiliza una capa *HCI* para conectar el *Host Cordio* BLE al Controlador *Cordio* BLE. El subsistema del Controlador *Cordio* BLE implementa la Capa de Enlace BLE (*Link Layer*).

Tanto los sistemas *Host* como los Controladores utilizan las siguientes interfaces de abstracción para la compatibilidad entre varias funciones dependientes del sistema:

- *Wireless Softare Foundation* (WSF) para utilidades comunes y características RTOS.
- Capa de abstracción de plataforma (PAL) para interactuar con los periféricos o dispositivos integrados de una MCU.

Cuando se opera en un sistema de doble chip, la pila y los perfiles *Cordio* se ejecutan en un microControlador y se comunican con un chip Controlador BLE a través de una interfaz cableada como UART o SPI. Una capa *HCI* estándar basada en transporte gestiona la comunicación entre los dos dispositivos. En la Ilustración 26, se muestra una representación la pila *Cordio* [12].

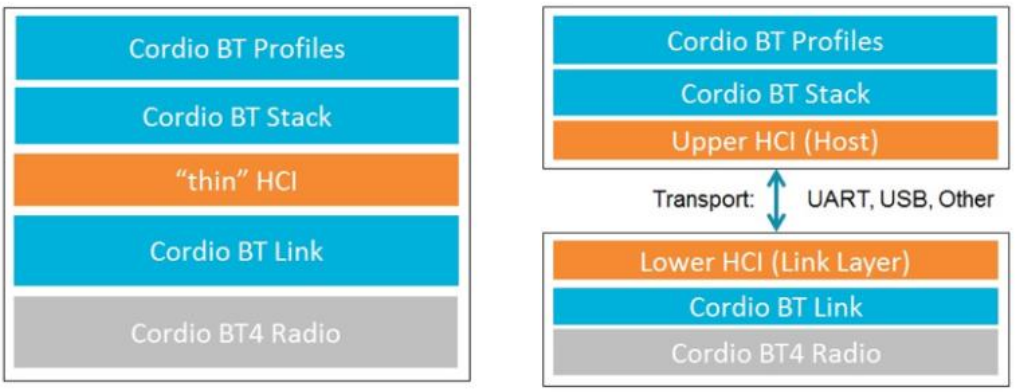


Ilustración 26: Pila Cordio en un único chipset y en un dual chipset [12].

3.1 Perfiles *Cordio*

Los perfiles *Cordio* de Arm constan de una aplicación, componentes de Servicios, perfiles *Bluetooth* interoperables, y un *Framework* para simplificar el desarrollo y la migración de aplicaciones.

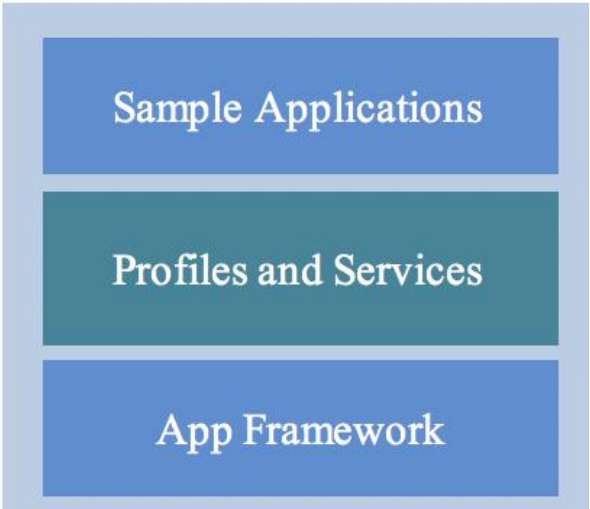


Ilustración 27: Perfiles Cordio [12].

Sistema software de Perfiles *Cordio*

- *Sample Applications,* proporcionan un código fuente de ejemplo para productos como un control remoto, un sensor de salud, o un reloj. Las aplicaciones de referencia están diseñadas con un enfoque orientado al producto, y cada aplicación admite uno o más perfiles BLE. Las aplicaciones de referencia interactúan con los perfiles, los Servicios y el *Framework.*
- *Profiles and Services,* son componentes interoperables diseñados para el Perfil de *Bluetooth* y los requisitos de especificación de Servicio. Los Perfiles y Servicios se utilizan en aplicaciones para implementar Características de Servicio y Perfiles específicos. Los Perfiles se implementan en archivos separados para cada rol de perfil. Sin embargo, los Servicios pueden agruparse en archivos según su función lógica y el perfil que los utiliza.
- *Application Framework,* la aplicación *Framework* actúa en algunas operaciones comunes en el ensamblado de aplicaciones BLE, tales como:
  - Dispositivos a nivel aplicación y gestión de seguridad.
  - Abstracciones simples de la interfaz de usuario para la gestión de pulsaciones de botones, sonidos, visualización, y otros comentarios del usuario.
  - Una base de datos de dispositivos abstractos para almacenar datos de enlace y otros parámetros del dispositivo.

El entorno de trabajo está definido tal y como se muestra en la Ilustración 28, donde cada subsistema contiene su propia API.

- *Main*: dispositivos, conexión y gestión de seguridad.
- *UI*: abstracción de la interfaz de usuario.
- *DB*: Base de Datos de dispositivos.
- *HW:* Abstracción de la interfaz del sensor hardware.

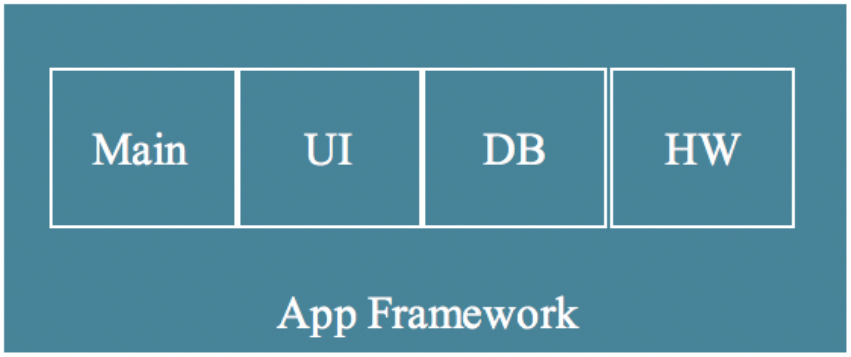


Ilustración 28: Subsistema software de Application Framework Cordio [12].

3.2 Pila *Cordio*

La pila *Cordio* constituye una solución completa de pila de protocolos de *Host* para dispositivos BLE monomodo. Consta de cinco capas de protocolo, como se muestra en la Ilustración 29.

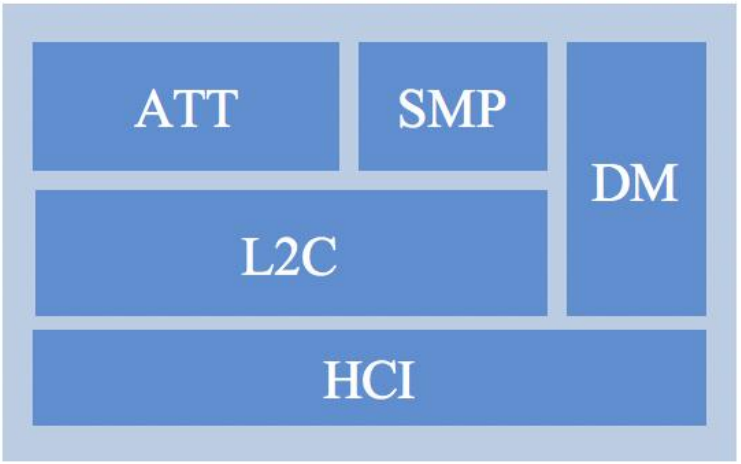


Ilustración 29: Arquitectura de la Pila Cordio [12].

ATT: El subsistema *ATT* implementa el protocolo *ATT* y el Perfil de Atributo Genérico (G*ATT*). Contiene dos subsistemas independientes: el *Cliente* de Protocolo de Atributo*s* (*ATTC*) y el *Servidor* de Protocolo de Atributo*s* (*ATTS*).

• *ATTC* implementa todas las Características del *Cliente* del protocolo *ATT* y está diseñado para cumplir con los requisitos del *Cliente* del perfil de Atributo genérico. *ATTC* puede admitir múltiples conexiones simultáneas a diferentes *Servidor*es.

• *ATTS* implementa todas las Características del *Servidor* de protocolo de Atributo*s*, y tiene soporte para múltiples conexiones de *Cliente* simultáneas. *ATTS* también implementa las Características del *Servidor* definidas por el Perfil de Atributo Genérico.

SMP: El subsistema *SMP* implementa el protocolo del administrador de seguridad. Contiene dos subsistemas independientes:

- El iniciador (*SMPI*). *SMPI* implementa las funciones de iniciador del protocolo del administrador de seguridad y tiene soporte para múltiples conexiones simultáneas.
- El respondedor (*SMPR*). *SMPR* implementa las funciones de respuesta del protocolo del administrador de seguridad y tiene soporte para una sola conexión (por diseño de especificación de *Bluetooth*).

SMP también implementa herramientas criptográficas, que usan *AES* (*Advanced Encryption Standard*). La interfaz de *AES* es asincrónica y se abstrae a través de WSF. *SMP* también implementa funciones para admitir la firma de datos.

L2C: El subsistema *L2C* implementa el protocolo *BLE L2CAP*. Es una versión sustancialmente reducida de *Bluetooth L2CAP* normal. En la ruta de transmisión de datos, la función principal de *L2C* es construir paquetes *L2CAP* y enviarlos a *HCI*. En la ruta de datos de recepción, su función principal es recibir paquetes de *HCI* y encaminarlos a *SMP* o *ATT*. *L2C* también implementa el procedimiento de actualización de parámetros de conexión.

HCI: El subsistema *HCI* implementa la especificación de la interfaz *Host*-Controlador. Esta especificación define comandos, eventos y paquetes de datos enviados entre una pila de protocolo BLE en un *Host* y una Capa de Enlace en un Controlador. La API de *HCI* está optimizada para ser una capa de interfaz para un sistema de un solo chip. Es configurable para un sistema de un solo chip o un sistema tradicional con *HCI* cableado. Esta capacidad de configuración se logra mediante una implementación en capas. Una capa de núcleo se puede configurar para un sistema de chip único o *HCI* cableado. Se puede configurar una Capa de Transporte y debajo de la capa *Central* para diferentes transportes por cable, como UART.

DM: El subsistema *DM* implementa los procedimientos de administración de dispositivos requeridos por la pila. Estos procedimientos están divididos por categoría de procedimiento y función de dispositivo (*Master* o *Slave*). En *DM* se implementan los siguientes procedimientos:

- *Advertising* y visibilidad del dispositivo: habilita / deshabilita el proceso de *Advertising*, establece parámetros y datos de *Advertising*, establece la conectividad y la visibilidad.
- Escaneo y descubrimiento de dispositivos: iniciar / detener proceso de *Scanning*, establecer parámetros de *Scanning*, informes de *Advertising*, descubrimiento de nombres.
- Gestión de conexiones: crear / aceptar / eliminar conexiones, configurar / actualizar parámetros de conexión, leer RSSI. Gestión de seguridad: *Bonding*, almacenamiento de parámetros de seguridad, autenticación, cifrado, autorización, gestión de direcciones aleatorias.
- Gestión de dispositivos locales: inicialización y reinicio, configuración de parámetros locales, comandos específicos del proveedor. Los procedimientos de DM admiten el perfil de acceso genérico (GAP) cuando corresponde [12].

3.3 Interfaces *Cordio*

El software usa diversas llamadas a funciones y funciones de devolución de llamada (*callback*) en sus API, como se desglosa en la Ilustración 30, y está conformado por los siguientes puntos:

- *Message Passing API Functions*. Las funciones de la API de paso de mensajes dan como resultado que se envíe un mensaje a la tarea que ejecuta la pila. Por lo general, estas funciones implican una operación compleja, como la creación de una conexión, y no acceden a datos internos (privados).
- *Direct Execute API Functions.* Se ejecuta completamente en el contexto de la función de llamada. Por lo general, estas funciones implican operaciones simples como leer o configurar datos internos. La programación de tareas está bloqueada al acceder a datos internos.
- *Callback Functions.* Las funciones *Callback* las implementa el *Cliente* mediante la pila de protocolos y se ejecutan en el contexto de la pila. Las funciones de devolución de llamada se utilizan para enviar eventos y datos al *Cliente*.

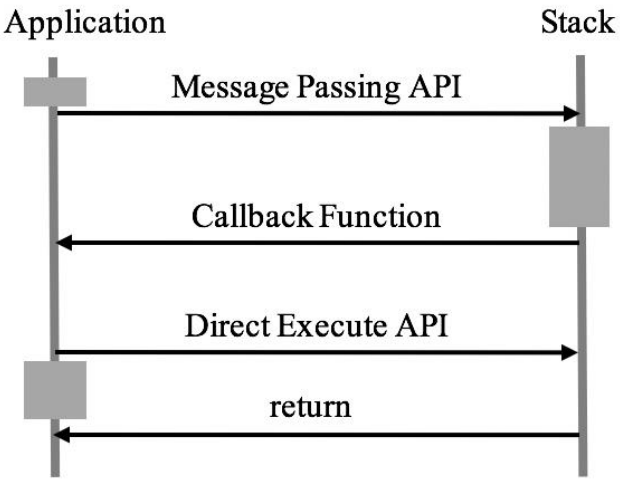


Ilustración 30: Ejecución de interfaces Cordio [12].

- *Event Handlers and Task. Cordio* define un Servicio de gestión de eventos que forma una base para los mecanismos de comunicación asincrónica utilizados en el sistema. Un Controlador de eventos puede recibir mensajes y eventos. Cada subsistema software normalmente tiene su propio Controlador de eventos. La pila está diseñada para ser flexible y permitir diferentes arquitecturas de tareas. El sistema de software *Cordio* no define ninguna tarea, pero define algunas interfaces para las tareas. Se basa en el sistema operativo de destino para implementar tareas y administrar los Servicio*s* de temporizador y Controlador de eventos del sistema operativo de destino. Un sistema de software típico de un solo chip utilizará tareas separadas para la aplicación, la pila y la Capa de Enlace. Sin embargo, no hay nada en el diseño de la pila o los perfiles de protocolos que impida que se ejecuten en la misma tarea que otros sistemas de software [12].

3.4 Ruta de datos

En esta sección se describe el flujo de datos entre aplicaciones y *HCI*.

Ruta de Transmisión:

La ruta de transmisión de datos cubre el flujo de datos que se envían desde la aplicación, a través de la pila, y posteriormente a *HCI*, pudiendo escribir dos copias de datos en la ruta de datos transmisión:

- Cuando los datos se envían desde la aplicación a la pila.
- Cuando los datos se envían desde la pila a *HCI*.

Como optimización, *Cordio* proporciona una API de copia cero (*zero-copy*) que utiliza un solo *buffer* de datos entre la aplicación y la pila. También se utiliza una *API* de copia cero entre la pila y *HCI* cuando se

ejecuta en una única arquitectura de *CPU.* La pila no copia datos internamente entre capas. La asignación y desasignación de *buffers* de datos tiene lugar en el punto donde se copian los datos. Cuando la aplicación envía datos a la pila, se asigna un *buffer* y los datos se copian en el *buffer*. Cuando los datos se envían desde la pila a la *HCI,* o la Capa de Enlace, los datos se copian en una *HCI* o en un *buffer* de la Capa de Enlace y se desasigna el *buffer* de la pila. En la Ilustración 31 se muestra el flujo de transmisión.

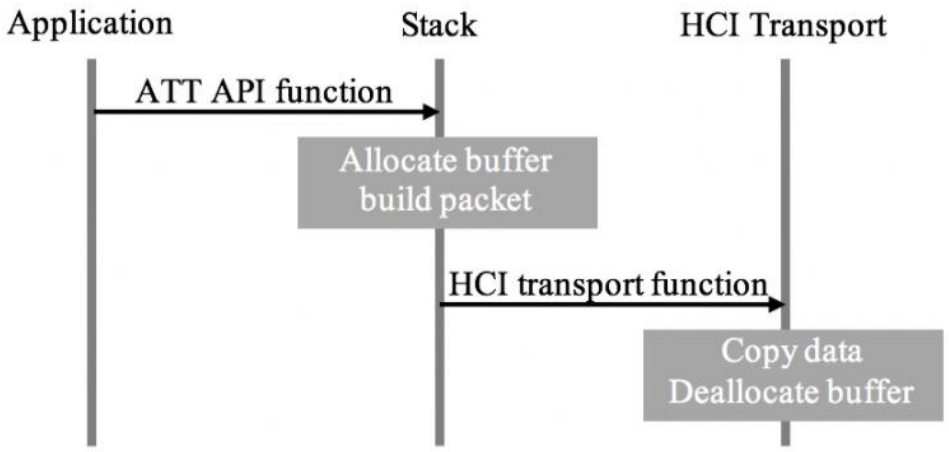


Ilustración 31: Ruta de transmisión Cordio [12].

En este escenario, la aplicación llama a una función de la *API ATTC* que inicia la transmisión de un paquete. La función *ATTC* asigna un *buffer* y construye un paquete de protocolo de Atributos, reservando espacio al principio del paquete para las cabeceras *L2CAP* y *HCI*. *ATTC* busca el manejador *(handle) HCI* para esta instancia y llama a una función *L2C*, pasando el manejador junto con el paquete y la longitud del paquete a *L2C.*

L2C comprueba que el enlace correspondiente a este gestor está conectado. Si no lo está, *L2C* descarta el paquete. Entonces *L2C* construye las cabeceras *L2CAP y HCI* para el paquete y llama a una función *HCI* para enviar el paquete a *HCI*. A continuación, *HCI* procesa el paquete. En general, *HCI* copiará los datos, desasignará el *buffer,* y pondrá en cola los datos [12].

Ruta recepción de datos:

La ruta de recepción datos cubre el flujo de datos cuando se envía desde *HCI*, a través de la pila, y luego a la aplicación. Al igual que la ruta de transmisión, puede haber dos copias de datos en la ruta de datos de recepción:

- Cuando los datos se envían desde la pila a la aplicación.
- Cuando los datos se envían desde *HCI* a la pila. La pila no copia datos internamente entre capas.

Los *buffers* son asignados por la capa *HCI* y posteriormente reasignados internamente por la pila. La Ilustración 32 muestra la ruta de recepción.

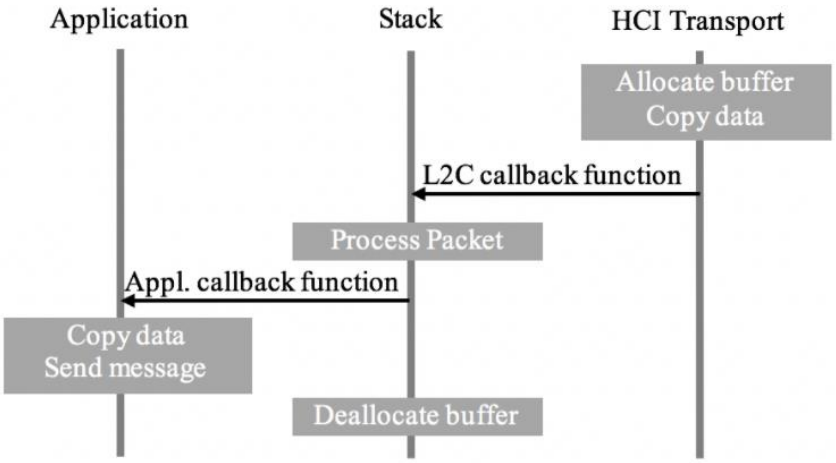


Ilustración 32: ruta de recepción Cordio [12].

HCI recibe un paquete de datos. Asigna un *buffer* WSF y copia todo el paquete de datos, incluyendo el *handle HCI* y la longitud, en el *buffer*. A continuación, *HCI* llama a una función de devolución de llamada de *L2C* para enviar los datos a *L2C*.

L2C recibe el paquete y realiza una comprobación básica de errores en la longitud y el PSM. Si se detecta un error, el paquete se descarta y el *buffer* se reasigna. Si el paquete es correcto, se encamina a *ATT* o *SMP*. En este ejemplo, el paquete se dirige a *ATT* y *L2C* llama a una función de retorno de *ATT*.

ATT recibe el paquete y realiza una comprobación de errores en la longitud y el *opcode* del Atributo. Si se detecta un error, el paquete se descarta y el *buffer* se reasigna. Si el paquete es correcto, se dirige a *ATTC* o *ATTS*. En este ejemplo, el paquete se dirige a *ATTS* y *ATT* llama a una función de retorno de *ATTS*.

ATTS procesa el paquete recibido según la especificación del protocolo de Atributos. En este ejemplo, el paquete se pasa a la aplicación para su posterior procesamiento. *ATTS* llama a la función de devolución de llamada de la aplicación, que asigna un *buffer* de mensajes WSF, copia los datos en el *buffer,* y envía el mensaje a la tarea de la aplicación. Después de llamar a la función de devolución de llamada de la aplicación, *ATTS* desasigna el *buffer* del paquete [12].

3.5 HCI

3.5.1 Gestor de Eventos HCI

Se utiliza un gestor de eventos de WSF para procesar eventos y mensajes, como el evento de tiempo de espera del comando *HCI* (si es aplicable), los eventos *HCI* recibidos, y los datos ACL (*Asynchronous Connection-Less)*. Un ejemplo de gestor de eventos para la implementación del chip dual está en la función

HCICoreHandler() en el archivo *HCI_core_ps.c* proporcionado en la pila *Cordio.* Esta función realiza las siguientes acciones:

- Gestiona el tiempo de espera del comando *HCI*.
- Procesa los datos entrantes en la cola *HCI* de transmisión.
- Gestiona la secuencia de reinicio durante el procesamiento de eventos *HCI*.
- Ejecuta la función de reensamblaje para los datos ACL entrantes.
- Pasa los paquetes ACL reensamblados a la pila mediante una devolución de llamada.

3.5.2 Gestión de conexiones

HCI necesita almacenar el estado de cada conexión para gestionar la fragmentación y el reensamblado de paquetes, así como para tener en cuenta adecuadamente los *buffers* de paquetes ACL *(Asynchronous Connection-Less*) del Controlador[12]. Las siguientes funciones de gestión de conexiones se implementan en el archivo *HCI_core.c* de la pila *Cordio:*

- *HCICoreConnOpen()*
- *HCICoreConnClose()*
- *HCICoreConnAlloc()*
- *HCICoreConnFree()*
- *HCICoreConnByHandle()*
- *HCICoreNextConnFragment()*

3.5.3 Comandos y eventos específicos del fabricante

El código *HCI* está diseñado para acomodar los comandos y eventos *HCI* específicos del fabricante. Las funciones para gestionar estos comandos y eventos se pueden añadir en el fichero *hci_vs.c*. La implementación de ejemplo para el chip dual contiene las siguientes funciones de marcador de posición:

- *HCICoreVSCmdCmplRcvd():* Gestiona eventos completos de comandos *HCI* específicos del proveedor.
- *HCICoreVsEvtRcvd():* Gestiona eventos *HCI* específicos del proveedor.
- *HCICoreHwErrorRcvd():* Realiza el procesamiento interno de *HCI* para los eventos de error de hardware.
- *HCIVsInit():* Inicialización del Controlador específico del proveedor.

3.5.4 Fragmentación

La fragmentación de paquetes ACL se lleva a cabo mediante una serie de funciones definidas en el archivo *hci_core.c*. Cuando se transmite un paquete, y su tamaño es mayor que el tamaño del paquete ACL del Controlador, se inicia el procedimiento de fragmentación. El paquete más grande se divide en múltiples paquetes ACL más pequeños hasta el tamaño del paquete del Controlador. Los fragmentos se envían desde el *buffer* del paquete ACL original, por lo que no es necesario asignar un nuevo *buffer* ni copiar datos para la fragmentación. Sin embargo, esto requiere una consideración especial para la reasignación del *buffer* de mayor tamaño de ACL. Cuando la transmisión de cada paquete de fragmentos ACL se completa (o tras la transmisión de un paquete ACL no fragmentado), la Capa de Transporte debe llamar a la función *HCICoreTxAclComplete()*. Esta función libera el *buffer* de paquetes ACL cuando se completa la fragmentación [12].

3.5.5 Reensamblado

El reensamblado de paquetes ACL se realiza en la función *HCICoreAclReassembly()*, definida en el archivo *HCI_core.c*. Esta función asigna un buffer de gran tamaño que contendrá el paquete reensamblado completo y posteriormente copia los fragmentos recibidos a este buffer, para reensamblar el paquete. Esta función también realiza una serie de comprobaciones de protocolo y longitud para verificar que los fragmentos de paquetes recibidos son válidos [12].

3.5.6 Configuración de la ruta de datos

Existen dos funciones definidas en el fichero *HCI_core.c* utilizadas para configurar las rutas de datos de recepción y transmisión. La función *HCISetAclQueueWatermarks()* establece los umbrales superior e inferior utilizados para el control de flujo en la ruta de datos de transmisión. Cuando el número de *buffers* en cola alcanza el umbral superior, se activa el control de flujo. Cuando el número de *buffers* alcanza el umbral inferior, se libera el control de flujo.

La función *HCISetMaxRxAclLen()* se utiliza para establecer el tamaño máximo del paquete ACL reensamblado. El valor mínimo se establece en 27 bytes, que es también el valor por defecto. Para recibir paquetes ACL de mayor tamaño, por ejemplo, cuando se utilizan conexiones seguras *SMP BLE* o se utilizan tamaños de *MTU ATT* mayor, se debe llamar a esta función para establecer un valor superior [12].

3.5.7 Ruta de datos de transmisión de ACL

La ruta de datos de transmisión de ACL cubre el flujo de datos de los paquetes ACL desde la pila hasta la capa de transporte o enlace por cable. La ruta de datos implementa varios procedimientos *HCI* que pueden incluirse opcionalmente en función de los requisitos de la plataforma. La Ilustración 33 muestra el funcionamiento típico de esta ruta de datos.

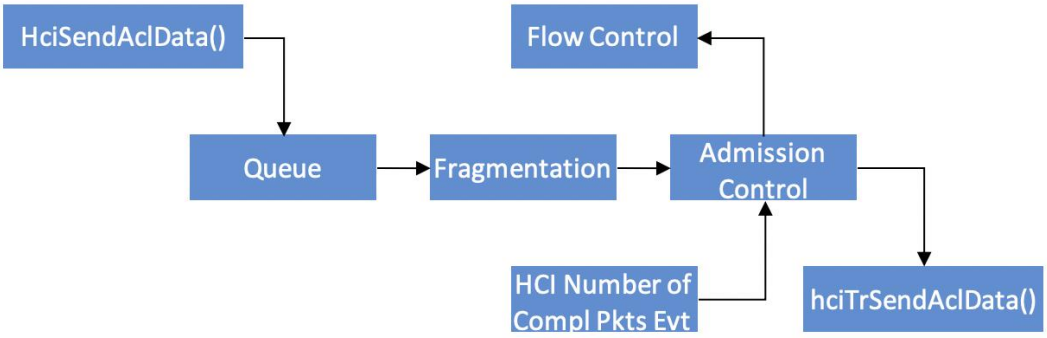


Ilustración 33: Ruta de transmisión de datos ACL[12].

Los paquetes de datos que se pasan a la función *HCISendAclData()* se ponen en cola. Si la longitud del paquete es mayor que el tamaño máximo del *buffer* de paquetes ACL del Controlador, el paquete se fragmenta. A continuación, los paquetes se pasan a un mecanismo de control de admisión que implementa el control de flujo de paquetes *HCI*, el procesamiento de eventos de número de paquetes completados *HCI* y el envío de eventos de control de flujo al *Cliente HCI*. La función *HCITrSendAclData()* es específica para el envío de un único paquete ACL a la capa de transporte o enlace por cable.

3.5.8 Ruta de datos de recepción de ACL

La ruta de datos de recepción de ACL cubre el flujo de paquetes de ACL desde la Capa de Enlace, o el transporte por cable, hasta la pila. La ruta de datos implementa varios procedimientos *HCI* que pueden incluirse opcionalmente en función de los requisitos de la plataforma. La Ilustración 34 muestra el funcionamiento típico de esta ruta de datos. En primer lugar, se recibe el paquete desde la Capa de Transporte por cable o de enlace. Si el paquete está fragmentado, se vuelve a ensamblar. A continuación, el paquete se pone en cola para el gestor de eventos *HCI*. Cuando el gestor de eventos *HCI* se ejecuta, procesa la cola y llama a la función *callback* del *Cliente* para enviar el paquete a la pila.

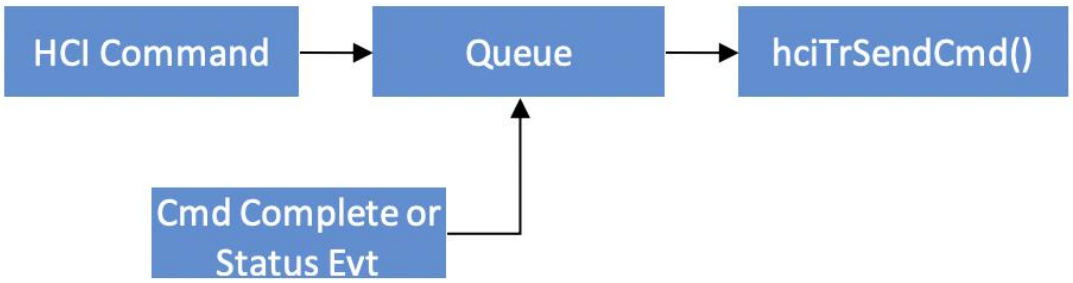


Ilustración 34: Ruta de recepción de datos ACL[12].

3.5.9 Ruta de datos de eventos

La ruta de datos de eventos cubre los eventos enviados desde la Capa de Enlace o el transporte por cable a la pila. La Ilustración 35 muestra el funcionamiento típico de esta ruta de datos.

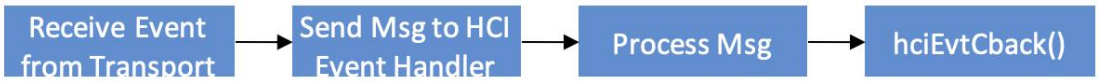


Ilustración 35: Ruta de datos de eventos[12].

Después de recibir un evento *HCI* desde la Capa de Enlace o transporte cableado, éste se envía en un mensaje al gestor de eventos *HCI*. Cuando el gestor de eventos *HCI* se ejecuta, procesa el mensaje y llama al *callback* del *Cliente HCI*, si es aplicable.

3.6 Organización de carpetas.

En la Tabla 2, la Tabla 3, la Tabla 4, la Tabla 5 y la Tabla 6 se presenta una descripción de las rutas de los diferentes directorios usados a lo largo del presente TFG, de acuerdo con la organización de *Cordio BLE Stack*.

Tabla 2: Directorio de los ficheros Cordio [12].

Directory	Description
ble-host	Cordio Stack software
ble-profiles	Cordio Profiles platform and sample projects
platform	Platform integration and example source
projects	Cordio sample applications
wsf	Wireless Software Foundation

Tabla 3: Directorio ble-Host [12].

Directory	Description
build	Build configuration / Makefiles
include	Host API
sources/hci	Host HCI source
sources/sec	Host security support (AES, ECC)
sources/stack	Host stack source

Tabla 4: Directorio perfiles [12].

Directory	Description
build	Build configuration / Makefiles
include	Profiles API
sources/apps	Application framework and sample applications
sources/profiles	BLE profiles
sources/services	BLE services

Tabla 5: Directorio espacios de trabajo Framework [12].

Directory	Description
app	App Framework
cycling	Cycling sensor sample application
datc	Proprietary data transfer client sample application
dats	Proprietary data transfer server sample application
fit	Fitness sensor sample application
gluc	Glucose sensor sample application
hidapp	HID sample application
medc	Health data collector sample application
meds	Health sensor sample application
sensor	Sensor sample application
tag	Proximity tag sample application
uribeacon	Uribeacon sample application
watch	Watch sample application
wdxs	Wireless data exchange application

Tabla 6: Directorio Perfiles Bluetooth [12].

Directory	Description
anpc	Alert Notification Profile client
bas	Battery Service server
blpc	Blood Pressure Profile client
blps	Blood Pressure Profile server
cpp	Cycling Power Profile server
cscp	Cycling Speed and Cadence Profile server
dis	Device Information Service client
fmpl	Find Me Profile locator
gap	GAP Profile
gatt	Generic Attribute Profile client
glpc	Glucose Profile client
glps	Glucose Profile server
hid	HID device
hrpc	Heart Rate Profile client
hrps	Heart Rate Profile server
htpc	Health Thermometer Profile client
htps	Health Thermometer Profile server
paspc	Phone Alert Status Profile client
plxpc	Pulse Oximeter Profile collector
plxps	Pulse Oximeter Profile sensor
rscp	Running Speed and Cadence Profile sensor
scpps	Scan Parameter Profile server
sensor	Example Temperature and Gyroscope Service Profile
tipc	Time Profile client
udsc	User Data Service Collector
uribeacon	Uribeacon Configuration Profile
wdxs	Proprietary Data Exchange Server Profile
wdxc	Proprietary Data Exchange Client Profile
wpc	Cordio proprietary profile client
wspc	Weight Scale Profile client
wsps	Weight Scale Profile server

3.7 Aplicación de referencia

En esta sección se analizan los parámetros configurables de la pila *Cordio BLE* a través de la aplicación de referencia *tag_main.c*. Conocer estos parámetros resulta de vital importancia para implementar los dispositivos *Peripheral y Central,* objetos del presente TFG.

Parámetros configurables:

- Parámetros del dispositivo *Slave.*

La configuración de los parámetros del dispositivo *Slave* se define en la estructura *appAdvCfg_t* ,que configura el intervalo y la duración del proceso de *Advertising.* La estructura contiene tres pares de valores de duración de intervalo de *Advertising* [12].

```
/*! \brief configurable parameters for Slave */
static const appAdvCfg_t tagSlaveCfg =
{
  {15000, 45000,     0},    /* Advertising durations in ms */
  {   56,   640,  1824}     /* Advertising intervals in 0.625 ms units */
};
```

Así, si la duración es cero, el proceso de *Advertising* no tendrá límite de tiempo, por lo que hasta que no se establezca una conexión, o sea finalizada por la aplicación, no se detendrá este proceso. En el ejemplo anterior se ha configurado el siguiente comportamiento (el intervalo de *Advertising* se especifica en unidades de 0,625 ms).

- 35 ms de intervalo por cada 15 segundos.
- 400 ms por cada 45 segundos.
- 1140 ms continuamente.

Parámetros de Seguridad:

La estructura de los parámetros de seguridad se encuentra en definida en la estructura de datos *appSecCfg_t*, donde se configuran las distintas opciones de Seguridad.

```
/*! \brief configurable parameters for security */
static const appSecCfg_t tagSecCfg =
{
```

```
  DM_AUTH_BOND_FLAG,  /* Authentication and bonding flags */
  0,                  /* Initiator key distribution flags */
  DM_KEY_DIST_LTK,    /* Responder key distribution flags */
  FALSE,              /* TRUE if Out-of-band pairing data is present */
  TRUE                /* TRUE to initiate security upon connection */
};
```

En el anterior ejemplo se define el siguiente comportamiento:

1) Se solicita la vinculación sin emparejamiento PIN.
2) Se distribuyen sólo las claves mínimas requeridas.
3) No existen datos fuera de banda.
4) Se debe iniciar una solicitud de seguridad al conectarse.

Parámetros de actualización de la conexión:

La estructura *appUpdateCfg_t* configura los parámetros de actualización de la conexión. Estos parámetros se utilizan después de establecer una conexión para reconfigurar una conexión de bajo consumo y/o baja latencia. La estructura *appUpdateCfg_t* se utiliza únicamente en los dispositivos *Slave.*

```
/*! \brief configurable parameters for connection parameter update */
static const appUpdateCfg_t tagUpdateCfg =
{
  6000,  /* Connection idle period in ms before ATTempting
            connection parameter update; set to zero to disable */
  640,   /* Minimum connection interval in 1.25 ms units */
  800,   /* Maximum connection interval in 1.25 ms units */
  0,     /* Connection latency */
  600,   /* Supervision timeout in 10 ms units */
  5      /* Number of update ATTempts before giving up */
};
```

En este ejemplo se define el siguiente comportamiento (en unidades de 1.25 ms):

1. Se solicita una actualización de los parámetros de conexión después de que la conexión haya permaneciendo inactiva durante, al menos, 6 segundos. Se considera que la conexión está inactiva cuando no hay ningún procedimiento de seguridad o de descubrimiento *ATT* pendiente.
2. Se solicita establecer un Intervalo de Conexión entre 800 y 1000 ms.
3. Se solicite una latencia de conexión de cero, lo que significa que los dispositivos *Slave* y *Master* tienen intervalos de conexión iguales.

4. Se establece el tiempo de espera de supervisión en 6 segundos. Si la conexión se pierde durante este intervalo de tiempo, los dispositivos se desconectarán.
5. Se intenta actualizar los parámetros de conexión 5 veces. El dispositivo *Master* puede rechazar una actualización de los parámetros de conexión si está ocupado. Si esto ocurre, la actualización de los parámetros de conexión se intentará de nuevo posteriormente.

Parámetros HID:

Las aplicaciones que utilizan el Servicio HID (como por ejemplo, el teclado, el ratón y el control remoto como en el caso de la apliación de referencia *tag)*, deben registrar un *hidConfig_t* con el perfil HID.

```
static const hidReportIdMap_t mouseReportIdSet[] =
{
  /* type                 ID           handle */
  {HID_REPORT_TYPE_INPUT, 0,           HIDM_INPUT_REPORT_HDL},  /* Input Report */
  {HID_REPORT_TYPE_INPUT, HID_BOOT_ID, HIDM_MOUSE_BOOT_IN_HDL}, /* Boot Input Report
*/
};

static const uint8_t mouseReportMap[] =
{
  0x05, 0x01,                   /* USAGE_PAGE (Generic Desktop) */
  0x09, 0x02,                   /* USAGE (Mouse) */
  0xa1, 0x01,                   /* COLLECTION (Application) */
  0x09, 0x01,                   /*   USAGE (Pointer) */
  0xa1, 0x00,                   /*   COLLECTION (Physical) */
  0x05, 0x09,                   /*     USAGE_PAGE (Button) */
  0x19, 0x01,                   /*     USAGE_MINIMUM (Button 1) */
  0x29, 0x03,                   /*     USAGE_MAXIMUM (Button 3) */
  0x15, 0x00,                   /*     LOGICAL_MINIMUM (0) */
  0x25, 0x01,                   /*     LOGICAL_MAXIMUM (1) */
  0x95, 0x03,                   /*     REPORT_COUNT (3) */
  0x75, 0x01,                   /*     REPORT_SIZE (1) */
  0x81, 0x02,                   /*     INPUT (Data,Var,Abs) */
  0x95, 0x01,                   /*     REPORT_COUNT (1) */
  0x75, 0x05,                   /*     REPORT_SIZE (5) */
  0x81, 0x03,                   /*     INPUT (Cnst,Var,Abs) */
  0x05, 0x01,                   /*     USAGE_PAGE (Generic Desktop) */
  0x09, 0x30,                   /*     USAGE (X) */
  0x09, 0x31,                   /*     USAGE (Y) */
  0x15, 0x81,                   /*     LOGICAL_MINIMUM (-127) */
  0x25, 0x7f,                   /*     LOGICAL_MAXIMUM (127) */
```

```
  0x75, 0x08,                    /*     REPORT_SIZE (8) */
  0x95, 0x02,                    /*     REPORT_COUNT (2) */
  0x81, 0x06,                    /*     INPUT (Data,Var,Rel) */
  0xc0,
  0xc0
};

/*! \brief HID Profile Configuration */
static const hidConfig_t mouseHidConfig =
{
  HID_DEVICE_TYPE_MOUSE,                              /* Type of HID device */
  (uint8_t*) mouseReportMap,                          /* Report Map */
  sizeof(mouseReportMap),                             /* Size of report map in bytes
*/
  (hidReportIdMap_t*) mouseReportIdSet,               /* Report ID to ATTribute Handle
map */
  sizeof(mouseReportIdSet)/sizeof(hidReportIdMap_t), /* ID to Handle map size (bytes)
*/
  NULL,                                               /* Output Report Callback */
  NULL,                                               /* Feature Report Callback */
  mouseInfoCback                                      /* Info Callback */
};

HidInit(&mouseHidConfig);
```

En este ejemplo se define el siguiente comportamiento:

1. Se configura un dispositivo *HID Mouse*, definido por HID_DEVICE_TYPE_MOUSE. Los ratones HID soportan informes *Boot Mouse HID*. Los tipos de dispositivos alternativos son HID_DEVICE_TYPE_KEYBOARD, que soporta informes de *Boot keyboard*, y HID_DEVICE_TYPE_GENERIC, que no soporta el protocolo *HDI Boot*, o informes *HDI Boot*.
2. Registra un mapa de informes HID definido por el *mouseReportMap*. El mapa de informes HID es un descriptor de informes HID para el dispositivo HID. En la especificación *USB HID* se encuentra una descripción detallada de los descriptores de informes HID.
3. Se realiza un *callback* de información que recibe la notificación de los mensajes del Punto de Control HID y mensajes del Modo de Protocolo HID a través de la función *mouseInfoCback().*
4. Se define como una aplicación que no recibe informes de características o de salida HID. Las aplicaciones que deseen recibir informes de Características o de salida HID deben indicar funciones de devolución de llamada a los parámetros *outputCback* o f*eatureCback* de la estructura *hidConfig_t.*

Datos de *Advertising*:

Los datos de *Advetising,* así como los datos de respuesta al proceso de *Scan,* se configuran a través de simples matrices de bytes. Existen conjuntos separados de datos *Advertising* y de respuesta de *Scan* para el modo conectable y descubrimiento.

El contenido de los datos de *Advertising* y de respuesta al proceso de *Scan* sigue un formato simple de longitud-tipo-valor, tal y como se define en la especificación *Bluetooth.* El byte de longitud contiene la longitud del byte de tipo, y los bytes de valor que le siguen. El byte de tipo contiene el tipo de datos de *Advertising*, o tipo AD, que especifica un tipo de datos concreto. Los bytes de valor, si están presentes, se establecen de acuerdo con el tipo AD.

```
/*! \brief Advertising data, discoverable mode */
static const uint8_t tagAdvDataDisc[] =
{
  /* flags */
  2,                                    /* length */
  DM_ADV_TYPE_FLAGS,                    /* AD type */
  DM_FLAG_LE_LIMITED_DISC |             /* flags */
  DM_FLAG_LE_BREDR_NOT_SUP,

  /* tx power */
  2,                                    /* length */
  DM_ADV_TYPE_TX_POWER,                 /* AD type */
  0,                                    /* tx power */

  /* device name */
  11,                                   /* length */
  DM_ADV_TYPE_LOCAL_NAME,               /* AD type */
  'c',
  'o',
  'r',
  'd',
  'i',
  'o',
  ' ',
  'a',
  'p',
  'p'
};
```

```
/*! \brief scan data, discoverable mode */
static const uint8_t tagScanDataDisc[] =
{
  /* service UUID list */
  7,                                    /* length */
  DM_ADV_TYPE_16_UUID,                  /* AD type */
  UINT16_TO_BYTES(ATT_UUID_LINK_LOSS_SERVICE),
  UINT16_TO_BYTES(ATT_UUID_IMMEDIATE_ALERT_SERVICE),
  UINT16_TO_BYTES(ATT_UUID_TX_POWER_SERVICE)
};
```

Los datos de *Advertising* constan de tres campos de tipo AD:

- *Flag*: Se establecen en modo descubrible limitado.
- Potencia de transmisión: La potencia de transmisión se establece en 0 dBm.
- Nombre del dispositivo: El nombre del dispositivo se establece como "*Cordio* app".

Los datos de la respuesta del proceso de *Scan* se establecen a la lista de Servicio*s* UUID. Esta contiene una lista de Servicio*s* soportados por el dispositivo. En este ejemplo, la lista contiene el Servicio de Pérdida de Enlace, el Servicio de Alerta Inmediata, y el Servicio de Potencia de Transmisión.

Datos de descubrimiento de *Cliente*s de *ATT*:

Los datos de descubrimiento de *Cliente*s de la *ATT* se utilizan para el descubrimiento de Servicio*s,* así como para gestionar la lista de manejadores (*handlers*) de los *Cliente*s, que contiene los manejadores de las Características y los Atributo*s* descubiertos.

La lista de manejadores es un vector de enteros definida por la aplicación de referencia. Las funciones de descubrimiento de *App Framework* que se utilizan para encontrar las Características y los Atributo*s* de los Servicio*s* deseados en un dispositivo homólogo, son las que establecen los gestores en la lista. En el caso de los dispositivos pares enlazados, la lista de manejadores se almacena en la base de datos del dispositivo, que se restaurará en conexiones posteriores sin tener que volver a realizar el proceso de descubrimiento. En el siguiente ejemplo, los datos de descubrimiento del *Cliente ATT* se configuran para descubrir el Servicio G*ATT* y el Servicio de Alerta Inmediata (IAS).

```
/*! \brief Discovery states: enumeration of services to be discovered */
enum
{
  TAG_DISC_GATT_SVC,      /*!< GATT service */
  TAG_DISC_IAS_SVC,       /*!< Immediate Alert service */
  TAG_DISC_SVC_MAX        /*!< Discovery complete */
};
```

```
/*! \brief the Client handle list, tagCb.hdlList[], is set as follows:
 *
 *  ------------------------------ <- TAG_DISC_GATT_START
 *  | GATT svc changed handle     |
 *  ------------------------------
 *  | GATT svc changed ccc handle |
 *  ------------------------------ <- TAG_DISC_IAS_START
 *  | IAS alert level handle      |
 *  ------------------------------
 */

/* Start of each service's handles in the handle list */
#define TAG_DISC_GATT_START       0
#define TAG_DISC_IAS_START        (TAG_DISC_GATT_START + GATT_HDL_LIST_LEN)
#define TAG_DISC_HDL_LIST_LEN     (TAG_DISC_IAS_START + FMPL_IAS_HDL_LIST_LEN)

/* Pointers into handle list for each service's handles */
static uint16_t *pTagGATTHdlList = &tagCb.hdlList[TAG_DISC_GATT_START];
static uint16_t *pTagIasHdlList = &tagCb.hdlList[TAG_DISC_IAS_START];
```

La enumeración del estado del proceso de descubrimiento consiste en una lista de los Servicios a descubrir. Estos valores se utilizan en el *callback* de descubrimiento de *App Framework*. A continuación, se definen algunas constantes y punteros para acceder a la lista de manejadores, como se muestra en la Ilustración 36.

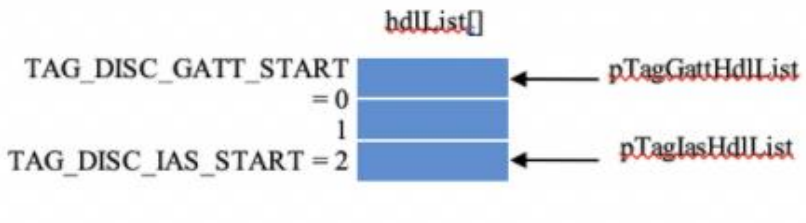


Ilustración 36: Ejemplo ATT Cordio [12].

En este ejemplo, la lista de manejadores almacena tres manejadores: Dos manejadores G*ATT* y un manejador IAS. Las constantes TAG_DISC_G*ATT*_START y TAG_DISC_IAS_START se establecen al índice inicial de los *handlers* para sus respectivos Servicios en la lista de manejadores. Los punteros *pTagGATTHdlList* y *pTagIasHdlList* apuntan al inicio de los manejadores para sus respectivos Servicios en la lista de *handlers*. Estos punteros son utilizados por las funciones de descubrimiento de Servicios del perfil (por ejemplo *GATTDiscover()* y *FmplIasDiscover()*) para acceder a la lista de manejadores.

Datos del *Cliente ATT*:

Cuando se completa el descubrimiento de Servicios y Características, un perfil suele requerir que se lean o escriban ciertas Características para configurar el perfil y los Servicios que utiliza. Por ejemplo, los Descriptores de Configuración de Características del *Cliente* (CCCD) suelen escribirse para habilitar las Indicaciones o Notificaciones de sus respectivas Características. Los datos del *Cliente ATT* consisten en constantes y estructuras de datos utilizadas para configurar una lista de Características descubiertas. Los datos se utilizan con la función *AppDiscConfigure()* de la API de *App Framework*.

La estructura de datos de tipo *ATTCDiscCfg_t* contiene una lista de Características para leer o escribir. Cada entrada de la lista contiene un valor (si se va a escribir), la longitud del valor y el índice del Atributo o Característica descubierta.

```
/*! \brief enumeration of client characteristic configuration descriptors used in
local ATT server */
enum
{
  TAG_GATT_SC_CCC_IDX,        /*!< GATT service, service changed characteristic */
  TAG_NUM_CCC_IDX             /*!< Number of ccc's */
};

/*! \brief client characteristic configuration descriptors settings, indexed by ccc
enumeration */
static const ATTSCccSet_t tagCccSet[TAG_NUM_CCC_IDX] =
{
  /* cccd handle         value range                 security level */
  {GATT_SC_CH_CCC_HDL,   ATT_CLIENT_CFG_INDICATE,    DM_SEC_LEVEL_ENC}
};
```

En este ejemplo, la base de datos del *Servidor* de la *ATT* contiene un único CCCD para la Característica de cambio de Servicio G*ATT*. La tabla de ajustes del CCCD tiene una sola entrada, que contiene el manejador del CCCD, el rango de valores, y el nivel de seguridad requerido para que se envíe una indicación o notificación con el valor de la Característica asociada al CCCD. En este ejemplo, el CCCD admite indicaciones, y se requiere la codificación antes de enviar una Indicación.

Llamadas de retorno de la pila de protocolos (*callback*):

- *Callback* DM. La función *callback DM* se ejecuta cuando la pila dispone de un evento de gestión de dispositivos para enviar a la aplicación. La función copia los parámetros del evento *callback* a un mensaje, y envía el mensaje al gestor de eventos de la aplicación de referencia.

- *Callback ATT*. La función de *callback ATT* se ejecuta cuando el *Cliente* o el *Servidor* del protocolo *ATT* tiene un evento que enviar a la aplicación. La función copia los parámetros del evento *callback* a un mensaje y envía el mensaje al gestor de eventos de la aplicación de referencia.
- *Calback ATT* CCC. La función *callback ATT CCC* se ejecuta cuando un dispositivo emparejado escribe un nuevo valor en un CCCD en el *Servidor ATT*. También se ejecuta al establecer la conexión si el CCCD se inicializa con un valor almacenado de una conexión anterior. La función comprueba primero si este nuevo valor de CCCD debe almacenarse en la base de datos del dispositivo. Si es así, el valor se almacena. A continuación, envía un mensaje al gestor de eventos de la aplicación de referencia con el valor del CCCD.

Funciones de acción del gestor de eventos:

En la aplicación de referencia se definen las funciones de las acciones de los gestores de eventos cuando un evento particular, como la apertura o el cierre de una conexión, requiere de acciones específicas en la aplicación. Las siguientes funciones son ejemplos de la aplicación referencia de *tag*.

- t*agClose*. Esta función realiza una alerta cuando se cierra la conexión.
- *tagSetup*. Esta función se ejecuta cuando se inicia la aplicación después de restablecer la pila, establece los datos de *Advertising y Scan response*, y a continuación inicia el proceso de *Advertising*.

```
/* set Advertising and scan response data for discoverable mode */
AppAdvSetData(APP_ADV_DATA_DISCOVERABLE, sizeof(tagAdvDataDisc),
            (uint8_t *) tagAdvDataDisc);
AppAdvSetData(APP_SCAN_DATA_DISCOVERABLE, sizeof(tagScanDataDisc),
            (uint8_t *) tagScanDataDisc);

/* set Advertising and scan response data for connectable mode */
AppAdvSetData(APP_ADV_DATA_CONNECTABLE, 0, NULL);
AppAdvSetData(APP_SCAN_DATA_CONNECTABLE, 0, NULL);

/* start Advertising; automatically set connectable/discoverable
mode and bondable mode */

AppAdvStart(APP_MODE_AUTO_INIT);
```

El dispositivo comienza a el proceso de *Advertising* llamando a la función *AppAdvStart().* Al utilizar el modo *auto init*, el modo de conexión/descubrimiento y de enlace del dispositivo se establece

automáticamente en función de si el dispositivo ya se ha vinculado. Si no se ha enlazado, el dispositivo se establece en modo detectable y enlazable. Si se ha enlazado, el dispositivo se establece en modo conectable y no enlazable.

Callback de descubrimiento:

Esta es la función de *callback* para la API de descubrimiento de *App Framework. App Framework* proporciona un conjunto de API de descubrimiento que simplifica el descubrimiento de Servicios y Características, así como la configuración de Servicios. *App Framework* ejecuta un *callback* en los momentos adecuados para que la aplicación realice una acción relacionada con el descubrimiento. El parámetro de estado de la función indica la acción a realizar, o el resultado del estado de una acción completada.

Los valores de estado y la acción asociada que realiza típicamente la función *callback,* son los siguientes:

1) *APP_DISC_INIT*. Este valor de estado se utiliza cuando se abre la conexión. La función debe llamar a *AppDiscSetHdlList(),* y pasar como parámetro un *buffer* de memoria para que *App Framework* almacene la lista de manejadores.
2) *APP_DISC_SEC_REQUIRED*. Este valor de estado se utiliza cuando se requiere de seguridad para completar la configuración. Esta función debe llamar a *AppSlaveSecurityReq()* y pasar como parámetro el *id* de conexión.
3) *APP_DISC_START*. Este valor de estado se utiliza cuando se inicia el proceso descubrimiento. La función debe iniciar el descubrimiento de Servicios para el primer Servicio a descubrir, por ejemplo llamar a *GATTDiscover().*
4) *APP_DISC_CMPL* y *APP_DISC_FAILED*. Estos valores de estado se utilizan cuando el procedimiento de descubrimiento previamente iniciada, ha finalizado. Si hay más Servicios por descubrir, inicia el descubrimiento para el siguiente Servicio. En caso contrario, llama a *AppDiscComplete(*APP_DISC_CMPL) para notificar a *App Framework* que todos los procedimientos de descubrimiento han finalizado. Si hay un procedimiento de configuración que realizar, inicia el procedimiento de configuración llamando a *AppDiscConfigure()* y utilizando los datos del *Cliente ATT*.
5) *APP_DISC_CFG_START.* Este valor de estado se usa para iniciar un procedimiento de configuración. Se utiliza cuando se han completado todos los procedimientos de descubrimiento, pero la configuración no se ha completado. Si hay un procedimiento de configuración que realizar, inicia el procedimiento. En caso contrario, llama a la función *AppDiscComplete(APP_DISC_CFG_CMPL*) para notificar a *App Framework* que todos los procedimientos de detección han finalizado.

6) *APP_DISC_CFG_CONN_START*. Este valor de estado se utiliza para iniciar un procedimiento de configuración de la conexión. Se utiliza cuando una aplicación necesita leer o escribir ciertas características del dispositivo homólogo cada vez que se establece una conexión. Si procede, llama a *AppDiscConfigure()* para realizar el procedimiento de configuración.
7) *APP_DISC_CFG_CMP.*: Esta función se llama cuando finaliza un procedimiento de configuración. Llama a *AppDiscComplete*(APP_DISC_CFG_CMPL) para notificar a *App Framework* que todos los procedimientos de descubrimiento han finalizado.

Función de procesamiento del gestor de eventos:

Esta función decodifica los eventos DM o *ATT* recibidos, y a continuación ejecuta una función de acción para realizar un procedimiento específico de la aplicación. El código de la aplicación de referencia también demuestra cómo los eventos DM se mapean a eventos UI que posteriormente se pasan a *AppUiAction()* para realizar una acción UI específica de la plataforma, por ejemplo hacer parpadear un LED cuando se establece una conexión.

Función de inicialización de la aplicación:

La función de inicialización de la aplicación se ejecuta en el arranque del sistema cuando se inicializan los gestores de eventos WSF. Esta función inicializa los punteros de configuración de *App Framework* e inicializa cualquier componente utilizado que requiera inicialización.

Función del gestor de eventos de la aplicación:

Este es el gestor de eventos WSF de la aplicación. Es ejecutada por el SO de WSF. Los mensajes recibidos se pasan a los componentes apropiados del *App Framework,* y posteriormente se pasan a la función de procesamiento del gestor de eventos de la aplicación.

Función de inicio de la aplicación:

Esta es la función que une todo para la aplicación. Así, esta función se ejecuta al iniciar el sistema después de que los gestores de eventos del WSF hayan inicializado. La función registra las funciones de *callback* de la pila de protocolos de la aplicación y las funciones de *callback* de *App Framework*.

A continuación, inicializa los Servicios utilizados en la base de datos del *Servidor ATT* local. Finalmente, se llama a la función *DmDevReset()* para restablecer la pila y el Controlador BLE, y posteriormente activar el inicio de la aplicación [12].

3.8 Plataforma *Artemis Thing Plus*

La plataforma *Artemis Thing Plus,* representada en la Ilustración 37, es un producto de la empresa *Sparkfun*, que se utilizará para implementar la funcionalidad, tanto del dispositivo *Peripheral* como del dispositivo Central en el presente TFG. Por tanto, será de vital importancia comprender todas las funcionalidades que a priori ofrece y de las que se hará uso.

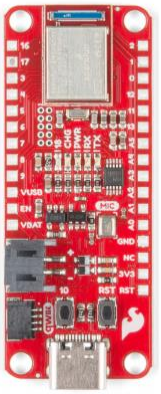


Ilustración 37: Artemis Thing Plus.

Características:

- Flash de 1M / RAM de 384k.
- 21 pines GPIO - todos con capacidad de interrupción.
- 21 canales PWM.
- Radio BLE incorporada.
- 8 canales ADC con precisión de 14 bits.
- Buses I2C.
- 1 bus SPI.
- UART.
- Micrófono digital PDM.
- Interfaz I2S.
- Conector Qwiic.

Programación:

La plataforma *Artemis Thing Plus* tiene dos métodos de programación. El más común es USB con conector tipo C (Ilustración 38) que opera como puente serie. La programación se puede realizar presionando el botón *Upload* en *Arduino* IDE, o bien a partir del comando "*make bootload*" desde el *SDK* .

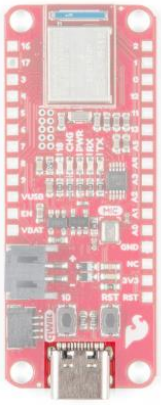


Ilustración 38: Conexión serial plataforma Artemis Thing Plus [13].

El chip integrado en la plataforma *Artemis Thing Plus* para esta conexión es el CH340C, como se muestra en la Ilustración 39, instalándose los *drivers* automáticamente en la mayoría de los sistemas operativos.

Ilustración 39: Chip CH340C [14].

Por otro lado, el segundo método disponible para la programación de la placa *Artemis Thing Plus* a través de comunicación serie, es JTAG. A lo largo del presente TFG, esta comunicación se usa para transferir el código compilado, presentando compatibilidad con cualquier programador o *debugger* JTAG.

Una interfaz JTAG es una interfaz especial de cuatro o cinco pines agregados a un chip, diseñada de tal manera que varios chips en una tarjeta puedan tener sus líneas JTAG conectadas de acuerdo con una política *daisy chain*, de manera que una sonda de testeo JTAG necesitaría conectarse a un solo "puerto JTAG" para acceder a todos los chips en un circuito impreso.

GPIO:

Artemis Thing Plus proporciona 21 pines GPIO situados en los laterales de la placa, separando salidas de entradas. Como se puede deducir en la Ilustración 40, cada una corresponde a la salida del MCU que se detallará en el siguiente apartado.

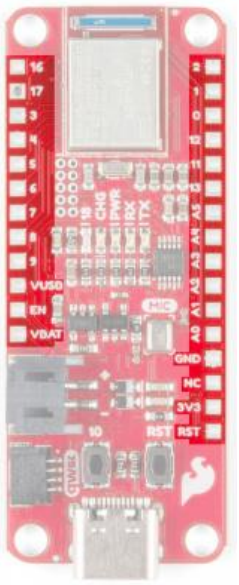


Ilustración 40: Gpio en la paltaforma Artemis Thing Plus [14].

I2C:

Esta placa permite el uso de I2C, un bus de comunicaciones serie que opera a una velocidad de 100 kbit/s en modo estándar, pudiendo llegar a 3.4 Mbits/s. Se caracteriza por utilizar dos líneas para transmitir la información, una para datos, y otra para la señal de reloj, siendo necesaria una tercera para referencia de masa. En este TFG, será el bus elegido para conectar un sensor de temperatura a la placa [15].

Los pines I2C en la plataforma *Artemis Thing Plus* están etiquetados como SDA y SCL en la parte posterior de la placa (Ilustración 41). Estos pines pueden ser controlados, por ejemplo, desde *Arduino IDE* usando sentencias de la librería *Wire,* como *Wire.begin(), Wire.read*(), aunque no será el caso del presente TFG [14].

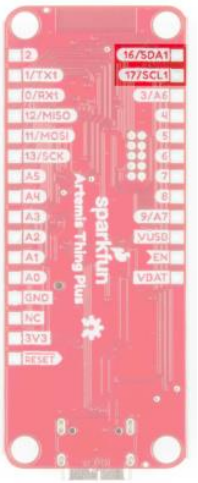


Ilustración 41: Pines I2C correspondiente a la plataforma Artemis Thing Plus [14].

3.9 MCU Apollo 3

Apollo3 blue es un MCU que expande la familia de microControladores Apollo de ultrabajo consumo de energía. Gestiona una reducción adicional en el consumo de energía sin precedentes hasta ahora en las generaciones anteriores del procesador Apollo ARM Cortex-M4 de alto rendimiento. Su frecuencia de reloj nominal es de 48 MHz, con la opción de alto rendimiento a 96 MHz [16].

Como se puede observar en la Ilustración 42, contiene un submódulo de comunicación serie con varias opciones, a elección del usuario, así como un submódulo que implementa el protocolo *Bluetooth*, lo que hace que represente un MCU idóneo para el objetivo de Implementar un dispositivo *Peripheral* BLE con la integración de un sensor I2C planteada inicialmente en el presente TFG. Con el objetivo de implementar la conexión *Bluetooth* entre los dispositivos Peripheral y *Central*, se deben estudiar las diferentes opciones que proporciona este MCU.

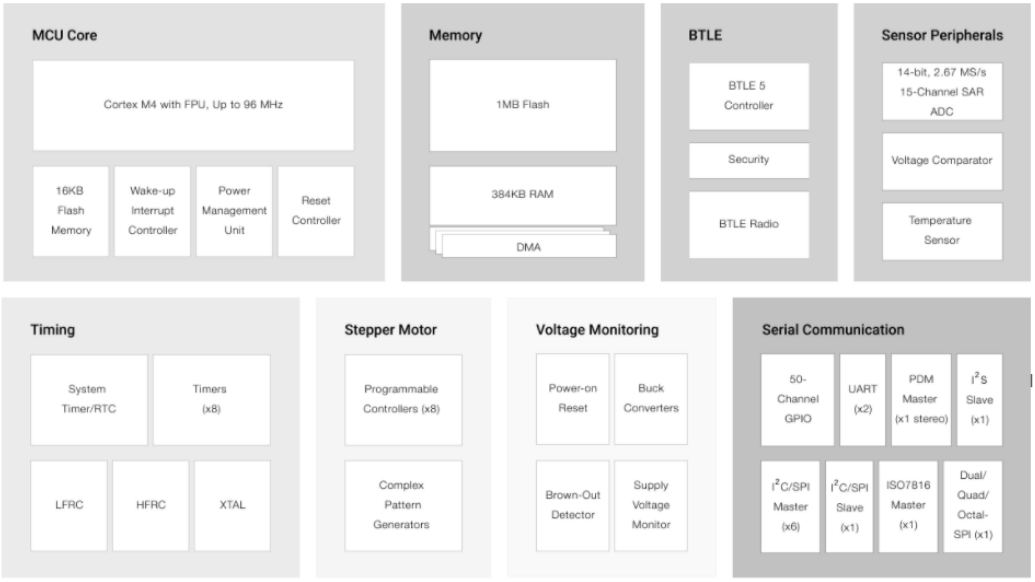


Ilustración 42: Módulo Ambiq Apollo 3 [16].

Comunicación serie:

Como se puede apreciar en la Ilustración 43, presenta un amplio abanico de posibles conexiones serie. Aunque se tratará en siguientes puntos con más detenimiento, se usará I2C. Se eligió esta interfaz, no solo porque ya se ha trabajado con este protocolo en otros proyectos dentro del grupo de investigación, sino por la sencillez que presenta aparentemente a la hora de su uso y configuración (por parte del fabricante).

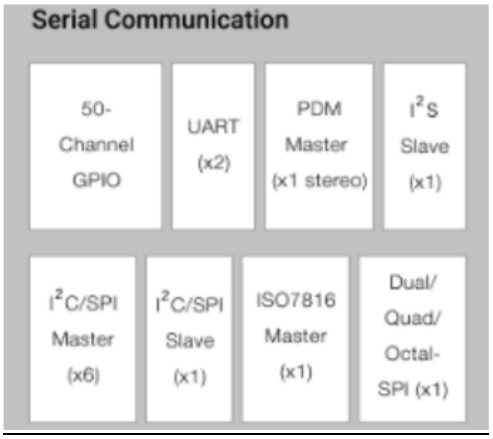


Ilustración 43: Comunicación Serial Apollo3 [16].

El *I2C / SPI Master* del *Apollo3 blue*, admite un conjunto flexible de comandos para implementar una variedad de interfaces I2C. La interfaz I2C consta de dos líneas: una línea de datos bidireccional (SDA) y una línea de reloj (SCL). Tanto la línea SDA como la SCL deben estar conectadas a una tensión de alimentación positiva mediante una resistencia de *pull-up*. Por definición, el dispositivo que envía un mensaje se denomina "transmisor", y el dispositivo que acepta el mensaje se llama "receptor". El dispositivo que controla la transferencia de mensajes mediante SCL se denomina *Master*. Los dispositivos controlados por el *Master* se denominan *Slave*s. El MCU *Apollo3 Blue I2C Master* es siempre un dispositivo *Master*.

Por otro lado, se define el siguiente protocolo:

- La transferencia de datos puede iniciarse solo cuando el bus no está ocupado.

- Durante la transferencia de datos, la línea de datos debe permanecer estable siempre que la línea del reloj esté a nivel alto.

- Los cambios en la línea de datos mientras la línea del reloj se encuentra a nivel alto se interpretarán como señales de control [17].

El funcionamiento descrito se muestra detalladamente en la Ilustración 44.

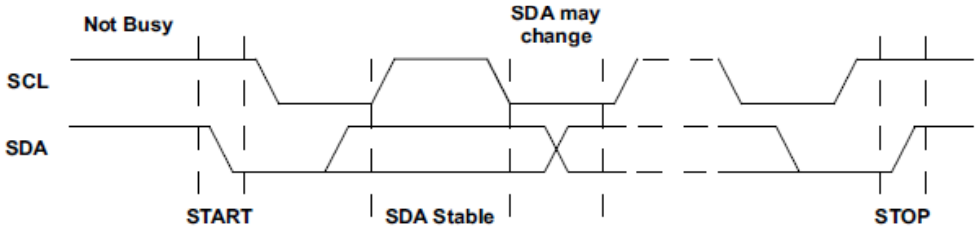


Ilustración 44: Funcionamiento SCL y SDA (I2C) [17].

Bluetooth Low Energy (BLE):

La MCU *Apollo3 Blue* incluye un subsistema de baja energía *Bluetooth* de baja potencia representado en la Ilustración 45. El Controlador BLE y el *Host* se pueden configurar para admitir hasta ocho conexiones seguras simultáneas, además de paquetes de longitud extendida.

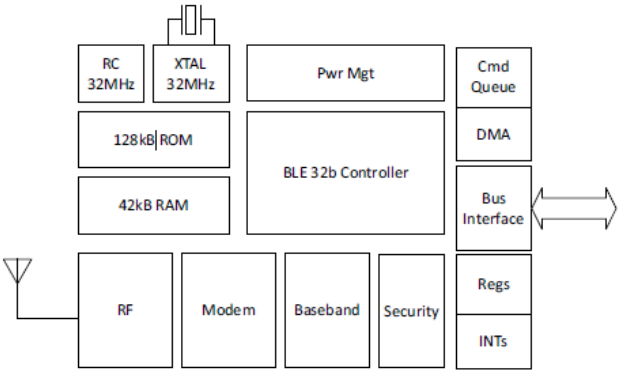


Ilustración 45: Módulo Bluetooth Apollo3 [16].

El subsistema BLE contiene un transceptor de RF de 2,4 GHz, un módem, una banda base y un procesador de 32 bits. Eso admite una fuente de reloj de cristal externa de 32 MHz, así como una fuente de reloj de oscilador RC interna de 32 MHz. El cristal de 32 MHz es necesario como referencia de frecuencia para la radio, y además de como fuente de reloj principal para los bloques del Controlador. El RC interno de 32 MHz se puede utilizar como fuente de reloj para el procesador de RF.

3.10 Configuración de la plataforma

3.10.1 Herramientas software

En el desarrollo del presente TFG, ha sido necesario instalar una serie las herramientas *software* (*toolchain)* para poder ejecutar el SDK de la plataforma *Artemis Thing Plus,* en el presente TFG se ha usado la versión 2.4.2 del SDK. En primer lugar, se justifica y se explica la lista de funcionalidades necesarias, para entender el motivo por el que se requiere cada elemento.

Compilador:

Convierte la librería que contiene todas las instrucciones útiles y sus diferentes definiciones para el uso del microControlador; en este caso, el SDK es proporcionado por *Sparkfun* en lenguaje máquina. Software usado*: GNU-RM* (compilador) *y Ambiq mircro Apollo3 blue SDK rel 2.4.2* [14].

Bash:

Lenguaje de órdenes y shell de Unix escrito para el Proyecto GNU como un reemplazo de software libre. Para SO *Windows* se usa *Git Bash.*

Makefile :

Será el archivo que defina el conjunto de tareas a ejecutar, en los SO tipo UNIX usados en *Artemis Thing Plus*.

*Bootloade*r propio de *Sparkfun:*

Un *bootloader* o gestor de arranque es un software especial que carga en la memoria interna el sistema operativo instalado en el sistema. Para conseguirlo, el *bootloader* suele ejecutarse directamente al arrancar un dispositivo usando algún medio que sea *booteable,* es decir, que sirva como unidad de arranque, como puede ser un disco duro, un CD o DVD, o un pendrive USB. El medio de arranque recibe la información acerca de dónde se encuentra el *bootloader* por parte del *firmwar*e. El sistema de arranque que se usará en el presente TFG es *SVL,* propio de *Sparkfun*, que permite realizar una carga a 921600 bps.

Python:

Será otro elemento a instalar, necesario para el desarrollo del presente TFG. En este caso, dentro del SDK algunas secuencias requieren de *pyhon3*, por lo que si se trabaja desde *Windows,* se debe cambiar el nombre del ejecutable *pyhton.exe* por *python3.exe.*

Al ser una programación a más bajo nivel, es necesario instalar la librería de datos que proporciona *python3*:

- *Pycryptodome*, es una librería que utiliza primitivas criptográficas, escrita principalmente en *python*, si bien para uso crítico (como el rendimiento) se utilizan algunas extensiones.
- *pyserial*, permite la comunicación serie RS232.

Sparkfun proporciona una guía de instalación del *toolchain* con enlaces a todos los programas y un pequeño manual para su instalación [18].

3.10.2 Configuración del repositorio

Para el uso de la pila *Cordio BLE* se utilizó *Windows* como sistema operativo. El proceso varía bastante con respecto a otros sistemas operativos más abiertos y compatibles con *UNIX*, como es Linux. Estas diferencias se explican detalladamente en la página oficial de *Sparkfun* [18].

Como primer paso, *Sparkfun* facilita unas referencias de *makefiles* y diferentes funcionalidades compatibles con el procesador *Apollo3 blue* [19], además de una librería maestra que contiene el *freeRTOS*,

y desde donde se realizarán todas las adaptaciones pertinentes para implementar los dispositivos *Peripheral y Central*.

Se Recomienda que el *SDK master* desde el que se trabaje se clone, iniciando el *bash* y ejecutando los siguientes comandos [20]:

git clone https://github.com/Sparkfun/Sparkfun_Apollo3_AmbiqSuite_BSPs boards_sfe

Una vez instalado el SDK, se debe descargar el BSP, clonando el contenido del repositorio *Github*, en la ruta del *SDK master*, dentro de la carpeta *boards_sfe*, desde el *bash:*

git clone https://github.com/Sparkfun/Sparkfun_Apollo3_AmbiqSuite_BSPs boards_sfe

3.10.3 Ejecución del primer ejemplo: *Blinky*

Una vez instalados todos los programas y configurado el repositorio hardware, el primer paso lógico será ejecutar uno de los ejemplos de referencia proporcionados por *Sparkfun*, en este caso, el denominado *blinky*. Las aplicaciones de muestra proporcionadas se ubican en el siguiente directorio, como se observa en la Ilustración 46.

AmbiqSuiteSDK\boards_sfe\Sparkfun_Apollo3_AmbiqSuite_BSPs\common

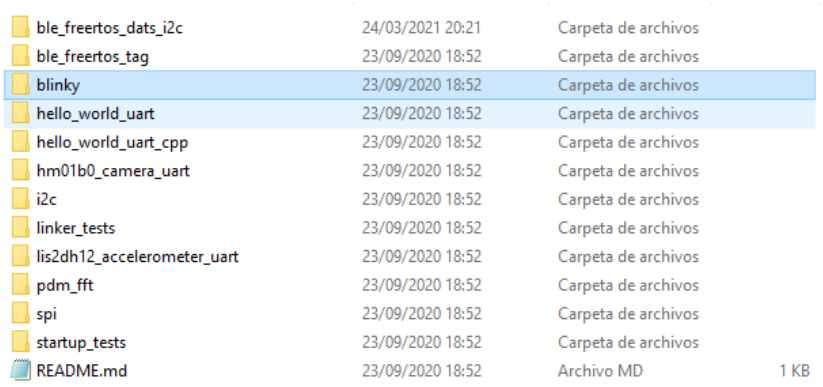


Ilustración 46: Ejemplos de referencia proporcionados por Sparkfun.

Al Abrir la carpeta del ejemplo *blinky* se encontrarán dos ficheros, el fichero *main* que contiene el código C de ejecución del ejemplo, y el código *gcc*, que contiene el *makefile* correspondiente. Para compilar el *makefile*, se ejecuta desde el *Git Bash* en esta ruta, para lo cual se debe usar el siguiente comando, como se observa en la Ilustración 47:

```
$ make BOARD=Artemis_thing_plus
```

```
rds_sfe/common/examples/blinky/gcc (master)
$ make BOARD=Artemis_thing_plus
Makefile:112: Using BOARD=Artemis_thing_plus at ../../../../Artemis_thing_plus
Makefile:117: warning: you have not defined COM_PORT. Assuming it is COM4
Makefile:121: warning: you have not defined PYTHON3. assuming it is accessible b
y 'python3'
Makefile:125: defaulting to 115200 baud for ASB
Makefile:129: defaulting to 921600 baud for SVL
Makefile:134: warning: you have not defined SDKPATH so will continue assuming th
at the SDK root is at ../../../../..
Makefile:142: warning: you have not defined COMMONPATH so will continue assuming
 that the COMMON root is at ../../../../common
Makefile:163: warning: you have not defined PROJECTPATH so will continue assumin
g that the PROJECT root is at ..
Makefile:170: CONFIG=../gcc/Artemis_thing_plus/bin
 Compiling gcc ../../../../common/examples/blinky/main.c
 Compiling gcc ../../../../common/tools_sfe/templates/startup_gcc.c
 Compiling gcc ../../../../../utils/am_util_delay.c
 Compiling gcc ../../../../../utils/am_util_faultisr.c
 Compiling gcc ../../../../../utils/am_util_stdio.c
 Compiling gcc ../../../../../devices/am_devices_led.c
 Linking gcc ../gcc/Artemis_thing_plus/bin/blinky_asb.axf with script ../../../.
./common/tools_sfe/templates/asb_linker.ld
 Copying gcc ../gcc/Artemis_thing_plus/bin/blinky_asb.bin...
```

Ilustración 47: Proceso de compilar desde el Git Bash.

Si al ejecutar el comando, no se producen errores, se genera una carpeta con los archivos compilados, lo que indica que no ha habido ningún problema en el proceso de compilación. El resultado debería ser semejante al mostrado en la Ilustración 48.

am_devices_led.d	26/03/2021 19:47	Archivo D	6 KB
am_devices_led.o	26/03/2021 19:47	Archivo O	10 KB
am_util_delay.d	26/03/2021 19:47	Archivo D	6 KB
am_util_delay.o	26/03/2021 19:47	Archivo O	5 KB
am_util_faultisr.d	26/03/2021 19:47	Archivo D	6 KB
am_util_faultisr.o	26/03/2021 19:47	Archivo O	6 KB
am_util_stdio.d	26/03/2021 19:47	Archivo D	1 KB
am_util_stdio.o	26/03/2021 19:47	Archivo O	19 KB
blinky.lst	26/03/2021 19:47	Archivo LST	82 KB
blinky.map	26/03/2021 19:47	Archivo MAP	68 KB
blinky_asb.axf	26/03/2021 19:47	Archivo AXF	90 KB
blinky_asb.bin	26/03/2021 19:47	Archivo BIN	5 KB
main.d	26/03/2021 19:47	Archivo D	7 KB
main.o	26/03/2021 19:47	Archivo O	6 KB
startup_gcc.d	26/03/2021 19:47	Archivo D	1 KB
startup_gcc.o	26/03/2021 19:47	Archivo O	7 KB

Ilustración 48: Archivos tras compilación.

Una vez compilado todo correctamente, se ejecuta el comando de arranque.

3.10.4 Conversión de ficheros genéricos, para su ejecución en la plataforma *Artemis Thing Plus*

Para poder implementar un dispositivo *Peripheral* funcional, se planteó inicialmente el uso de plantillas con funcionalidad BLE. Sin embargo, ninguno de los ficheros disponibles era apto como referencia, por lo que fue necesario buscar la forma de adaptar inicialmente las plantillas facilitadas por la propia pila *Cordio* BLE dentro de la librería, que sí cumplían con los requisitos necesarios. El fabricante *Sparkfun* no específica cómo hacerlo, por lo que fue necesario investigar y probar varios métodos.

A continuación, se describe paso a paso el proceso de conversión obtenido a partir del trabajo realizado en el presente TFG. En esta guía se convierte como referencia el fichero *ble_freertos_amdtps.*

1) En la siguiente ruta se encuentran los ficheros (Ilustración 49) con diversas funcionalidades propias de la pila *Cordio* BLE *C:\...\AmbiqSuiteSDK-master\boards\apollo3_evb\examples*.

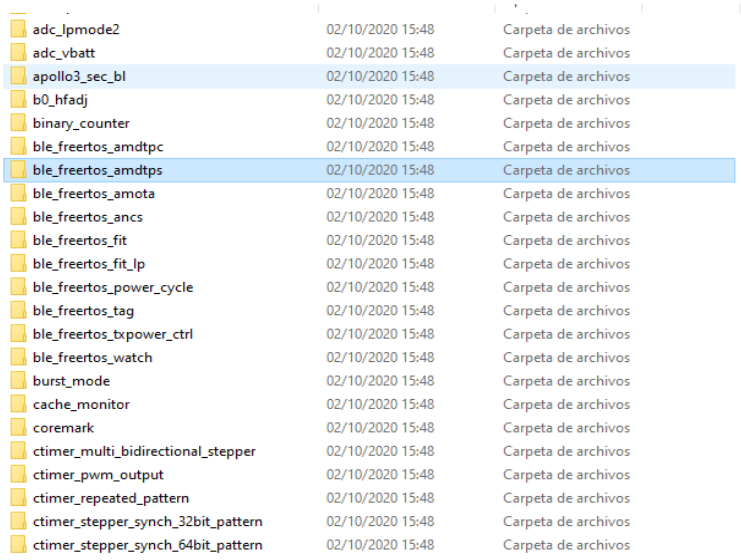


Ilustración 49: Ejemplos Apollo 3_evb.

2) Se traslada el directorio del ejemplo a convertir, desde *SDK/board/apollo3_evb/examples* (Librería del SDK del procesador), a la carpeta de los ejemplos del SDK de la placa *ArtemisThing Plus* (Librería propia de la placa).
3) Se copia el archivo raíz, observándose que existen dos carpetas, *iar* y *keil* (Ilustración 50), que se deberán eliminar.

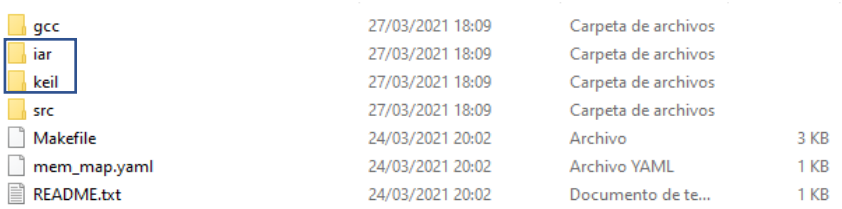


Ilustración 50: ficheros iar y keil.

4) En el mismo subdirectorio del SDK de la plataforma *Arthemis Thing Plus*, se buscan los ejemplos en la versión que se está usando, ubicados en la carpeta en *SDK/boards_sfe/common/examples*. De todos los ejemplos, se selecciona por ejemplo el *ble_freetos_tag,* y en *gcc* se copian los ficheros *ble_freertos_tag_asb.ld (*Ilustración 51*) y ble_freertos_tag_svl.ld* (funcionalidades *Bluetooth*), además de añadirlos al *gcc* del ejemplo que requiera la conversión.

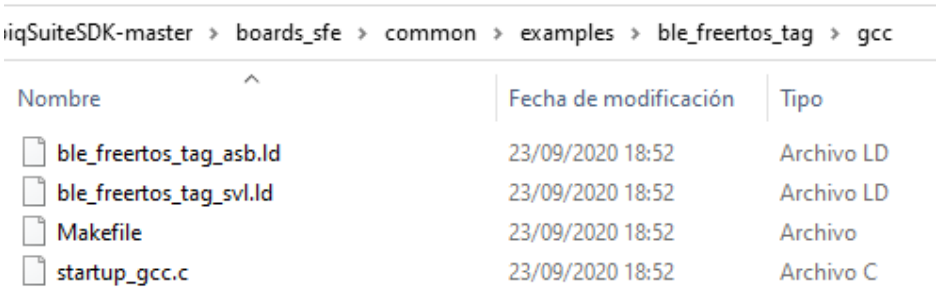


Ilustración 51: ficheros .ld a copiar.

5) Desde el ejemplo *ble_freetos_tag* se copia el *Makefile* y se sustituye por el que viene en el directorio del ejemplo a convertir. En la Ilustración 52 se muestra cómo debe estructurarse el directorio tras realizar las conversiones pertinentes.

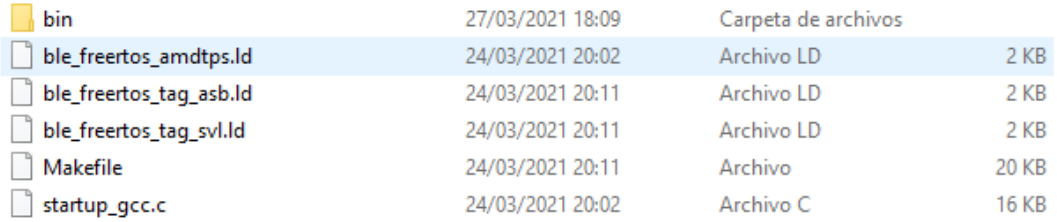


Ilustración 52: Cambio en el fichero a convertir.

6) Una vez copiados los ficheros del *tag* al *gcc*, y el *Makefile*, se ejecuta y compila siguiendo el proceso explicado en el punto 3.10.3.

3.10.5 Depuración

Una de las carencias que presentan las aplicaciones de referencia de la plataforma *Artemis Thing Plus* es la de no permitir mostrar mensajes de ejecución, pudiendo verse únicamente desde un dispositivo *Central* si existe algún dispositivo *Peripheral,* lo cual resulta insuficiente para poder realizar la implementación objeto del presente TFG.

Una solución que se planteó sería usar un depurador que, conectado con la placa y un software específico, permitiera visualizar la línea de ejecución del programa. Tras investigar con los GPIO compatibles para la plataforma *Artemis Thing Plus,* se encontró, de la mano de la empresa *Segger,* un depurador que mediante conexión tipo J-LINK propia de esta empresa, hace posible interconectar placa y ordenador con un depurador. El dispositivo elegido ha sido J-LINK EDU MINI, representado en la Ilustración 53.

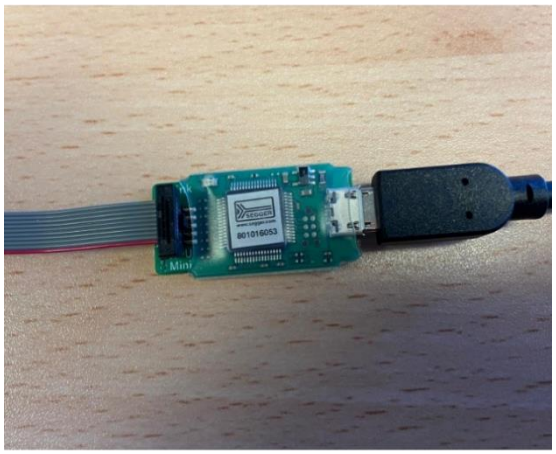


Ilustración 53: J-LINK EDU MINI.

Así, mediante el software *SWO viewer* propio de SEGGER, es posible visualizar la línea de ejecución, para lo cual simplemente se deberá configurar que procesador se usa, y la velocidad, tal y como se muestra en la Ilustración 54.

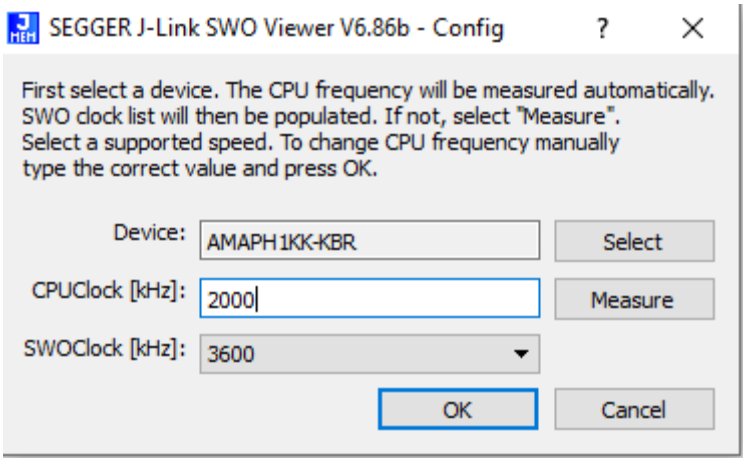


Ilustración 54: Segger J-Link SWO Viewer.

Capítulo 4: Implementación de los dispositivos *Peripheral* y *Central*

A lo largo de este capítulo se describe del proceso de diseño e implementación de los distintos dispositivos *Peripheral* y *Central*, esenciales para la comunicación BLE. El contenido se expone de manera cronológica para llevar un hilo temporal de proyecto y los diferentes problemas que se han tenido que solucionar para completar su implementación.

4.1 Implementación del dispositivo *Peripheral*

Para poder diseñar e implementar el dispositivo *Peripheral* se han estudiado alteravitas, siendo el objetivo inicial realizar un *Peripheral* básico, al cual se añadirá complejidad a lo largo del transcurso del presente TFG. Se partió de los ficheros de referencia incluidos en el SDK, siendo los ejemplos del procesador *amdtpc y amdtps,* propios de un dispositivo *Central* y *Peripheral*, respectivamente, los primeros en evaluarse como candidatos a ser la base de la implementación a realizar. Sin embargo, tras estudiar su funcionamiento y la poca información que ofrecían sobre el propio protocolo de comunicación AMDTP que implementaban (propietario de ARM), se decidió cambiar de estrategia.

El siguiente paso consistió en estudiar la referencia *ble_freertos_tag,* disponible en *AmbiqSuiteSDK**boards_sfe**common*\examples, donde se contempla el protocolo BLE y los diferentes parámetros que manejan para poder tener una comunicación exitosa basada en el perfil *Tag*. Esta funcionalidad ya ha sido compilada y ejecutada con éxito, siendo además detectado desde un dispositivo *Central*, por lo que se consideró bastante interesante estudiar su funcionamiento. Dentro del mismo ejemplo de referencia, en la carpeta *src* se encuentran otros ficheros usados para la correcta ejecución de *tag_main,* como *radio_task.c y ble_freertos_tag.c*, los ficheros principales para la comunicación BLE.

Tras su estudio y verificación, se llegó a la conclusión de que, en cuanto a conectividad *Bluetooth*, estos ficheros son viables para el intercambio de la información que se requiere. En consecuencia, el siguiente paso lógico, consistó en buscar una funcionalidad básica del dispositivo *Peripheral*, al fin y al cabo, entre los objetivos de este TFG no se contempla desarrollar una aplicación, sino estudiar el rendimiento y comportamiento de la placa *Artemis Thing Plus* usando el protocolo BLE a partir de la pila *Cordio* BLE.

La funcionalidad principal del dispositivo *Peripheral* que se desarrollará a lo largo del presente TFG será el de un *Peripheral* que realiza una medida de temperatura a partir de un sensor. Será la interacción con el dispositivo *Central* la que defina cuándo se deberá transmitir la información de temperatura obtenida, ya que el dispositivo *Central* solicitará su envío mediante una interfaz de usuario. La configuración de este funcionamiento será analizado y explicado a lo largo del presente capítulo.

4.1.1 Configuración del sensor

En cuanto a la funcionalidad básica del dispositivo *Peripheal*, se pensó en diferentes alternativas, pero finalmente se optó por implementar un dispositivo *Peripheral* con un sensor de temperatura, que capta la temperatura y la envía un a un dispositivo *Central,* cuando éste lo solicite.

Para ello se empezó analizando la aplicación I2C implementada en el fichero *main.c* disponible en el subdirectorio *AmbiqSuiteSDK-master/boards_sfe/common/examples/I2C*, con el fin de poder implementar una aplicación que permitiera la lectura de la temperatura a partir de la integración de un sensor MLX90614, representado en la ilustración 55, mediante una interfaz I2C.

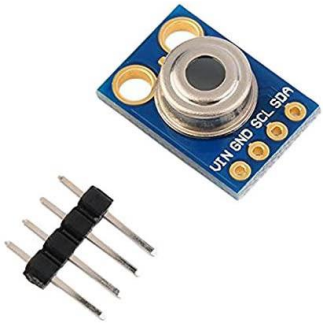


Ilustración 55 : Sensor Mlx90614

Si bien inicialmente se tomó como referencia el ejemplo implementado en el fichero *main.c,* en éste se realiza únicamente la función de escritura I2C, no permitiendo la operación de lectura. Por tanto, el primer paso consistió en lograr visualizar a través del puerto serie, mediante la aplicación *Putty,* los mensajes que se enviaban desde el código de *main.c* a través de directivas *am_util_studio_printf().*

Para ello, en el fichero *am_bsp.c,* disponible en *AmbiqSuiteSDK-master/boards_sfe /Artemis_thing_plus /bsp,* se determinó que, en la configuración por defecto de la UART, el parámetro *ui32Baudrate* se debía establecer a valor 115200 baudios, tal y como se muestra en la Ilustración 56.

```
//
// Standard UART settings: 115200-8-N-1
//
.ui32BaudRate = 115200,
.ui32DataBits = AM_HAL_UART_DATA_BITS_8,
.ui32Parity = AM_HAL_UART_PARITY_NONE,
.ui32StopBits = AM_HAL_UART_ONE_STOP_BIT,
.ui32FlowControl = AM_HAL_UART_FLOW_CTRL_NONE,
```

Ilustración 56: Configuración UART.

En consecuencia, en la aplicación *Putty*, además del puerto COM correspondiente, es necesario establecer el parámetro *Speed (baud*) al valor 115200. El resto de parámetros está por defecto en 8-N-1, por lo que no fue necesario modificar ningún otro parámetro.

4.1.2 Configuración del fichero I2C

A partir de este punto, se procede a modificar el código original de la aplicación I2C para lograr el objetivo de leer el valor de la temperatura de un objeto desde el sensor MLX90614 a través de I2C. Con este propósito, el primer paso consiste en analizar la implementación de la librería de referencia *Adafruit_MLX90614* para la integración del sensor MLX90624 [21]. En esta librería se puede obtener la dirección I2C del sensor, que es *0x5A,* así como la dirección del registro correspondiente a la medida de la temperatura de un objeto en grados, que es *0x07,* como se observa en la definición de los parámetros *MLX90614-12cadrr* y *MLX90614_TOBJ1*, respectivamente, en el fichero *Adalfruit_MLX90614.h* de la librería.

Por otro lado, en el fichero *Adafruit_MLX90614.cpp* de la librería se indica, en la función *read16(),* el número de bytes asociados a la lectura I2C del valor de la medida del sensor MLX90614. Esta función define qué bytes son recibidos, y el valor base (de tipo *uint16_t*).

A partir de este valor base, en la función *readTemp()* se indica el procedimiento que hay que seguir para obtener finalmente el valor de la temperatura en grados centígrados, que se corresponde con la Ilustración 57.

```
float Adafruit_MLX90614::readTemp(uint8_t reg) {
  float temp;

  temp = read16(reg);
  temp *= .02;
  temp -= 273.15;
  return temp;
```

Ilustración 57:readTemp I2C.

Una vez configurado el sensor, se retoma la implementación de la aplicación en el fichero *main.c.* Como referencia de un proceso de lectura I2C, se partió de un código genérico para la implementación de una aplicación capaz de realizar operaciones de lectura y una escritura I2C con *AmbiqSDK* [22].

Por otro lado, en la página oficial de la plataforma *Artemis Thing Plus* [22], se establecen ciertas claves para su configuración, indicándose que la interfaz I2C está asociada a los pines 17/SCL1 y 16/SDA1(Ilustración 58). Así, a partir de esta referencia, se pudo determinar con el esquemático de la plataforma *Artemis Thing Plus* disponible en el fichero *ArtemisThingPlusSchematic.pdf,* y representada en

la Ilustración 59, que los pines D17/SCL Y D16/SDA de la placa se correspondían con los pines del *MCU Apollo3 Blue*D42-SCL3 y D43-SDA3, respectivamente.

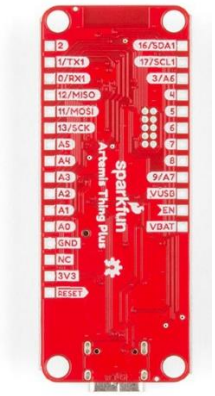


Ilustración 58: Sparkfun Artemis Thing Plus.

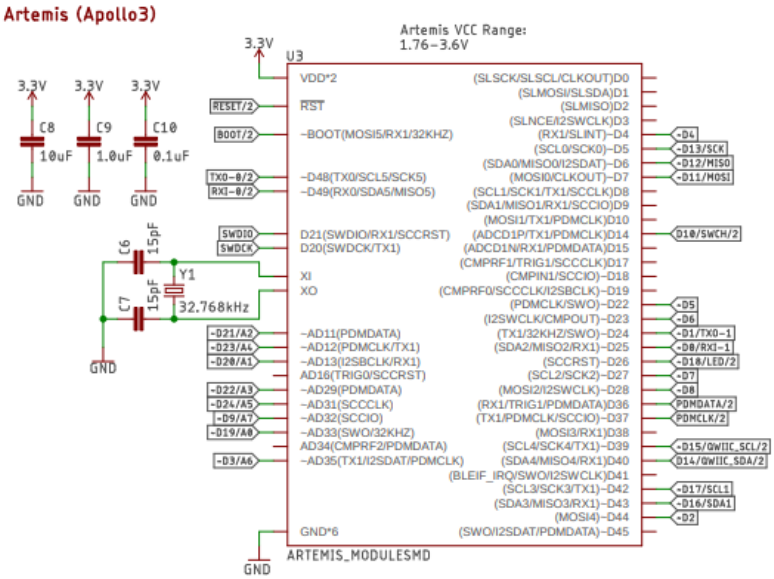


Ilustración 59: Esquemático Artemis y Apollo3

A partir de esta información, se pudo determinar en el fichero *am_bsp_pins.h* disponible en *ArtemisSDKSuite-master/boards_sfe/artemis_thing_plus/bsp,* que la interfaz I2C asociada a estos pines se correspondía con la del identificador 3. En la Ilustración 60 se muestra por defecto dónde se encuentran estos pines.

```
//*****************************************************************************
//
//  IOM3_SCL pin: I/O Master 3 I2C clock signal.
//
//*****************************************************************************
#define AM_BSP_GPIO_IOM3_SCL                42
extern const am_hal_gpio_pincfg_t       g_AM_BSP_GPIO_IOM3_SCL;

//*****************************************************************************
//
//  IOM3_SDA pin: I/O Master 3 I2C data signal.
//
//*****************************************************************************
#define AM_BSP_GPIO_IOM3_SDA                43
extern const am_hal_gpio_pincfg_t       g_AM_BSP_GPIO_IOM3_SDA;
```

Ilustración 60: Pines asociados a I2C.

Por tanto, en el fichero *main.c* ya se puede establecer el valor de DEVICE-ADDR (0x5A). DEVICE_ADDR_R(0X07), así como IOMN a '3,' además de establecer la frecuencia del reloj de la i/F I2C a 100KHz, de modo que I2C_FREQ (AM_HAL_IOM_100 KHz). Por otro lado, en la función de configuración *int_iom()* se estableció en las líneas 39 y 42 que los pines GPIO a configurar se correspondían con las señales SCL y SDA asociadas a la interfaz I2C con identificador 3 como se epresenta en la Ilustración 61.

```
#define DEVICE_ADDR (0x5A) // FBTG // MLX90614_I2CADDR en Adafruit_MLX90614 library
#define DEVICE_ADDR_R (0X07) // FBTG // MLX90614_TOBJ1 en Adafruit_MLX90614 library

#define IOMN 3 // FBTG // Pins: 17/SCL,16/SDA en ArtemisThingPlusChematic.pdf y /boards_sfe/artemis_thing_plys/bsp/asm_bsp_pins.h
#define I2C_FREQ (AM_HAL_IOM_100KHZ)
```

Ilustración 61: IOMN, pines I2C.

Por otro lado, los parámetros asociados a la lectura I2C se establecieron, tal y como se muestra en la Ilustración 62.

```
xfer.uPeerInfo.ui32I2CDevAddr = DEVICE_ADDR_W;
xfer.ui32InstrLen = 0;
xfer.ui32Instr = 0;
xfer.ui32NumBytes = 2;
xfer.eDirection = AM_HAL_IOM_TX;
xfer.pui32TxBuffer = (uint32_t*)cmd;
xfer.pui32RxBuffer = NULL;
xfer.bContinue = false;
xfer.ui8RepeatCount = 0;
xfer.ui8Priority = 1;
xfer.ui32PauseCondition = 0;
xfer.ui32StatusSetClr = 0;
```

Ilustración 62: Parámetros de lectura I2C.

A partir de estos parámetros, en la sentencia *status= am_hal_iom_blocky_transfer (iom_handle, 8xfa)* se invoca la lectura I2C, realizándose a partir de los valores recibidos en el *array cmd*, los cálculos

necesarios para obtener el valor de *tipo float* correspondiente a la temperatura en grados centígrados del objeto, medida por el sensor MLX90614 (Ilustración 63 e Ilustración 64).

```
am_hal_iom_blocking_transfer(void *pHandle,
                             am_hal_iom_transfer_t *psTransaction)
{
    uint32_t ui32Cmd, ui32Offset, ui32OffsetCnt, ui32Dir, ui32Cont;
    uint32_t ui32FifoRem, ui32FifoSiz;
    uint32_t ui32Bytes;
    uint32_t ui32IntConfig;
    uint32_t *pui32Buffer;
    am_hal_iom_state_t *pIOMState = (am_hal_iom_state_t*)pHandle;
    uint32_t ui32Module;
    uint32_t ui32Status = AM_HAL_STATUS_SUCCESS;
    bool     bCmdCmp = false;
    uint32_t numWait = 0;
```

Ilustración 63: iom_blocking_transfer I2C.

```
status = am_hal_iom_blocking_transfer(iom_handle, &xfer);
```

Ilustración 64: main_org.c de I2C.

Con todo esto, al compilar y programar la placa *Artemis_Thing_Plus* con el fichero *main.c* modificado, mediante el comando *make bootload_svl BOARD= artemis_thing_plus COM_PORT= COM6,* se obtuvo en el terminal de la aplicación *Putty* la correcta lectura de la temperatura en grados centígrados, medida por el sensor MLX90614, y tansferida a través de I2C.

4.1.3 Fichero *dats.c*

En esta sección se presentan los ficheros más importantes en la implementación del dispositivo *Peripheral* desarrollado en el presente TFG. Para ello se usó como referencia el fichero *dats,* que se encuentra en la ruta *AmbiqSuiteSDK-master\third_party\exactle\ble-profiles\sources\apps*. A continuación se muestran los parámetros de configuración establecidos en el dispositivo *Peripheral* implementado en el presente TFG, para su integración con el sensor MLX90614. Todos estos parámetros se configuran siguiendo el planteamiento expuesto en la Ilustración 65, la Ilustración 66, la Ilustración 67, la Ilustración 68, la Ilustración 69, la Ilustración 70, y la Ilustración 71.

Parámetros de *Advertising:*

```
/*! configurable parameters for advertising */
static const appAdvCfg_t datsAdvCfg =
{
  {30000,     0,     0},                  /*! Advertising durations in ms */
  {   96,  1600,     0}                   /*! Advertising intervals in 0.625 ms units */
};

/*! configurable parameters for slave */
static const appSlaveCfg_t datsSlaveCfg =
{
  1,                                      /*! Maximum connections */
};
```

Ilustración 65: Configuración de parametros Advertising en dats.c

- Se realizarán intervalos de *Advertising* durante 60 ms por cada 30 segundos.
- Intervalos de 1000 ms continuamente.
- Se establece el máximo de conexiones del dispositivo que, al tratarse de un dispositivo *Peripheral,* puede ser de un máximo de 1 dispositivo.

Parámetros de Seguridad:

```
/*! configurable parameters for security */
static const appSecCfg_t datsSecCfg =
{
  DM_AUTH_BOND_FLAG | DM_AUTH_SC_FLAG,    /*! Authentication and bonding flags */
  DM_KEY_DIST_IRK,                        /*! Initiator key distribution flags */
  DM_KEY_DIST_LTK | DM_KEY_DIST_IRK,      /*! Responder key distribution flags */
  FALSE,                                  /*! TRUE if Out-of-band pairing data is present */
  TRUE                                    /*! TRUE to initiate security upon connection */
};
```

Ilustración 66: Parámetros de seguridad dats.c

1) Se solicita la vinculación sin emparejamiento PIN.
2) Se distribuyen únicamente las claves mínimas requeridas.
3) No hay datos fuera de banda.
4) Se debe iniciar una solicitud de seguridad al conectarse.

Parámetros *SMP*:

```
/*! SMP security parameter configuration */
static const smpCfg_t datsSmpCfg =
{
  500,                                    /*! 'Repeated attempts' timeout in msec */
  SMP_IO_NO_IN_NO_OUT,                    /*! I/O Capability */
  7,                                      /*! Minimum encryption key length */
  16,                                     /*! Maximum encryption key length */
  1,                                      /*! Attempts to trigger 'repeated attempts' timeout */
  0,                                      /*! Device authentication requirements */
  64000,                                  /*! Maximum repeated attempts timeout in msec */
  64000,                                  /*! Time msec before attemptExp decreases */
  2                                       /*! Repeated attempts multiplier exponent */
};
```

Ilustración 67: Parámetros SMP dats.

Se definen otras características que se definen a la hora de realizar una conexión, como máximo número de intentos, longitud máxima y mínima del código encriptado, etc.

Parámetros de actualización de la conexión:

```
/*! configurable parameters for connection parameter update */
static const appUpdateCfg_t datsUpdateCfg =
{
  0,                                    /*! Connection idle period in ms before attempting
                                            connection parameter update; set to zero to disable */
  640,                                  /*! Minimum connection interval in 1.25ms units */
  800,                                  /*! Maximum connection interval in 1.25ms units */
  3,                                    /*! Connection latency */
  600,                                  /*! Supervision timeout in 10ms units */
  5                                     /*! Number of update attempts before giving up */
};
```

Ilustración 68: Parámetros de actualización de conexión.

1. Se solicita un Intervalo de Conexión entre 800 y 1000 ms.

2. Latencia de conexión 0 con lo que, en este caso, los dispositivos *Master* y *Slave* no tienen que tener el mismo valor de intervalo.

3. Se establece el tiempo de espera de supervisión en 6 segundos. Si la conexión se pierde durante 6 segundos, los dispositivos se desconectarán.

4. Se intenta actualizar los parámetros de conexión 5 veces. El dispositivo *Master* puede rechazar una actualización de los parámetros de conexión si está ocupado. Si esto ocurre, se intentará de nuevo la actualización de los parámetros de conexión.

Configuración *ATT*:

```
/*! ATT configurable parameters (increase MTU) */
static const attCfg_t datsAttCfg =
{
  15,                                   /* ATT server service discovery connection idle timeout in seconds */
  241,                                  /* desired ATT MTU */
  ATT_MAX_TRANS_TIMEOUT,                /* transcation timeout in seconds */
  4                                     /* number of queued prepare writes supported by server */
};

/*! local IRK */
static uint8_t localIrk[] =
{
  0x95, 0xC8, 0xEE, 0x6F, 0xC5, 0x0D, 0xEF, 0x93, 0x35, 0x4E, 0x7C, 0x57, 0x08, 0xE2, 0xA3, 0x85
};
```

Ilustración 69: Configuración parámetros ATT dats.

Se definen los parámetros más importantes del *Servidor ATT* y se establece la Clave de Resolución de Identidad (IRK).

Parámetros de *Advertising:*

```
/**************************************************************************************************
  Advertising Data
**************************************************************************************************/

/*! advertising data, discoverable mode */
static const uint8_t datsAdvDataDisc[] =
{
  /*! flags */
  2,                                    /*! length */
  DM_ADV_TYPE_FLAGS,                    /*! AD type */
  DM_FLAG_LE_GENERAL_DISC |             /*! flags */
  DM_FLAG_LE_BREDR_NOT_SUP,

  /*! manufacturer specific data */
  3,                                    /*! length */
  DM_ADV_TYPE_MANUFACTURER,             /*! AD type */
  UINT16_TO_BYTES(HCI_ID_ARM)           /*! company ID */
};

/*! scan data, discoverable mode */
static const uint8_t datsScanDataDisc[] =
{
  /*! device name */
  8,                                    /*! length */
  DM_ADV_TYPE_LOCAL_NAME,               /*! AD type */
  'D',
  'a',
  't',
  'a',
  ' ',
  'T',
  'X'
};
```

Ilustración 70: Estructura datos Advertising dats.

Los datos *Advertising* constan de tres campos de tipo AD:

- La longitud. En este caso es de 2.
- El tipo. Se especifica que es de tipo *Advertising.*
- *Flag.* Definen el modo de descubrimiento, se utiliza la bandera genérica para el descubrimiento, y el Indicador de *BR/EDR* no compatible.

Además, por defecto, el fabricante proporciona una configuración estándar para estos datos. Por último, se establece un nombre para identificar el dispositivo, que en este caso es *"Data TX"*.

Función *ATTS*Ccset_t:

```
/*! client characteristic configuration descriptors settings, indexed by above enumeration */
static const attsCccSet_t datsCccSet[DATS_NUM_CCC_IDX] =
{
  /* cccd handle          value range               security level */
#if WDXS_INCLUDED == TRUE
  {WDXS_DC_CH_CCC_HDL,    ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_DC_CH_CCC_IDX */
  {WDXS_FTC_CH_CCC_HDL,   ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_FTC_CH_CCC_IDX */
  {WDXS_FTD_CH_CCC_HDL,   ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_FTD_CH_CCC_IDX */
  {WDXS_AU_CH_CCC_HDL,    ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_AU_CH_CCC_IDX */
#endif /* WDXS_INCLUDED */

  {GATT_SC_CH_CCC_HDL,    ATT_CLIENT_CFG_INDICATE,  DM_SEC_LEVEL_NONE},   /* DATS_GATT_SC_CCC_IDX */
  {WP_DAT_CH_CCC_HDL,     ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE}    /* DATS_WP_DAT_CCC_IDX */
};
```

Ilustración 71: Estructura de los datos ATT dats.

Se configura cómo son gestionadas las Características, siguiendo la estructura que muestra la Ilustración 72.

Ilustración 72: Gestión de las Características ATT dats.

La base de datos del *Servidor ATT* contiene dos CCCD que gestionan la Característica de Cambio de Servicio G*ATT* y la configuración de las Características de los *Client*es de datos propietarios. La tabla de ajustes del CCCD tiene una sola entrada, que contiene el manejador del CCCD, el rango de valores, y el nivel de seguridad requerido para que se envíe una Indicación o una Notificación con el valor de la Característica asociada al CCCD. En este caso, el CCCD admite Indicaciones y Notificaciones, sin requerir codificación antes de ser enviadas.

Variables locales:

A continuación, se genera una estructura que posee el valor del identificador (Ilustración 73) y qué tipo de *Phymode* se configura, de entre los inicialmente soportados por la pila *Cordio* (Ilustración 74).

```
/*************************************************************************************************
  Local Variables
*************************************************************************************************/

/*! application control block */
static struct
{
  wsfHandlerId_t    handlerId;        /* WSF handler ID */
#if CS50_INCLUDED == TRUE
  uint8_t           phyMode;          /*! PHY Test Mode */
#endif /* CS50_INCLUDED */
} datsCb;
```

Ilustración 73: Esctructura de control WSFhandler dats.

```
/**************************************************************************************
  Macros
**************************************************************************************/
#if CS50_INCLUDED == TRUE

/* PHY Test Modes */
#define DATS_PHY_1M                     1
#define DATS_PHY_2M                     2
#define DATS_PHY_CODED                  3

#endif /* CS50_INCLUDED */
```

Ilustración 74: Phymode dats.

Inicialización I2C:

En la Ilustración 75, se muestra el código que implementa todo el proceso descrito en el punto 4.1.2.

```
void init_iom( void ){
    am_util_stdio_printf("--- ESTOY EJECUTANDO LA FUNCIÓN init_iom( void )\r\n");

    uint32_t status = AM_HAL_STATUS_SUCCESS;

    iom_cfg.eInterfaceMode = AM_HAL_IOM_I2C_MODE;
    iom_cfg.ui32ClockFreq = I2C_FREQ;

    status = am_hal_iom_initialize(IOMN, &iom_handle);
    if(status != AM_HAL_STATUS_SUCCESS){ report(status); }
    am_util_stdio_printf("iom_handle = %d \r\n");

    status = am_hal_iom_power_ctrl(iom_handle, AM_HAL_SYSCTRL_WAKE, false);
    if(status != AM_HAL_STATUS_SUCCESS){ report(status); }

    status = am_hal_iom_configure(iom_handle, &iom_cfg);
    if(status != AM_HAL_STATUS_SUCCESS){ report(status); }

    status = am_hal_iom_enable(iom_handle);
    if(status != AM_HAL_STATUS_SUCCESS){ report(status); }

    // config pins
    status = am_hal_gpio_pinconfig(AM_BSP_GPIO_IOM3_SCL, g_AM_BSP_GPIO_IOM3_SCL); //FBTG // en /boards_sfe/artemis_thing_plys/bsp/asm_bsp_pins.h
    if(status != AM_HAL_STATUS_SUCCESS){ report(status); }

    status = am_hal_gpio_pinconfig(AM_BSP_GPIO_IOM3_SDA, g_AM_BSP_GPIO_IOM3_SDA); //FBTG // en /boards_sfe/artemis_thing_plys/bsp/asm_bsp_pins.h
    if(status != AM_HAL_STATUS_SUCCESS){ report(status); }
}
```

Ilustración 75: inicialización I2C dats.

Función *datsSendData*:

Se elabora el mensaje a enviar que, en este caso, se trata de la información que se obtiene del sensor MLX90614 a través de I2C (Ilustración 76).

```
static void datsSendData(dmConnId_t connId)
{
    am_util_stdio_printf("--- ESTOY EJECUTANDO LA FUNCIÓN datsSendData(dmConnId_t connId)\r\n");

//  uint8_t str[] = "hello hello hello hello hello hello.....from dats";
//  char str[] = "hello hello hello hello hello hello.....from dats";
  char str[] = "hello from dats";
  uint8_t measBattLevel = 0x00;
  uint8_t measBattLevel2[] = {0x95, 0xC8, 0xEE, 0x6F, 0xC5, 0x0D, 0xEF, 0x93, 0x35, 0x4E, 0x7C, 0x57, 0x08, 0xE2, 0xA3, 0x85};

  APP_TRACE_INFO1("ESTOY EN datsSendData(connId) - %d", sizeof(str));



    // i2c
    uint32_t status = AM_HAL_STATUS_SUCCESS;

    init_iom();

    uint8_t cmd[3] = {0,0,0};

    // i2c.read; // FBTG //
    xfer.uPeerInfo.ui32I2CDevAddr = DEVICE_ADDR;
    xfer.ui32InstrLen = 1;
    xfer.ui32Instr = DEVICE_ADDR_R; //MLX90614_TOBJ1
    xfer.ui32NumBytes = (uint32_t)3;
    xfer.eDirection = AM_HAL_IOM_RX;
    xfer.pui32RxBuffer = (uint32_t*)cmd;
    xfer.bContinue = false;
    xfer.ui8RepeatCount = 0;
    xfer.ui32PauseCondition = 0;
    xfer.ui32StatusSetClr = 0;
    xfer.ui8Priority = 1;
```

Ilustración 76: Configuración del mensaje I2C dats.

Se declara una variable denominada *cmd* en la que, si la lectura se realiza correctamente, se almacenan los datos que proporciona el sensor. Se puede observar este comportamiento en la Ilustración 77.

```
const char* stat_msg = NULL;
switch(status){
    case AM_HAL_STATUS_SUCCESS : stat_msg = "success"; break;
    case AM_HAL_IOM_ERR_I2C_NAK : stat_msg = "NAK"; break;
    case AM_HAL_IOM_ERR_I2C_ARB : stat_msg = "ARB"; break;

    default:
        stat_msg = "UNKNOWN ERROR";
        break;
}
am_util_stdio_printf("I2C READ result: %s\r\n", stat_msg);
am_util_stdio_printf("cmd[0]: 0x%02X\r\n", cmd[0]);
am_util_stdio_printf("cmd[1]: 0x%02X\r\n", cmd[1]);
am_util_stdio_printf("cmd[2]: 0x%02X\r\n", cmd[2]);

uint16_t ret;
float temp;

ret = cmd[0];
ret |= cmd[1] << 8;
temp = ret;
temp *= .02;
temp -= 273.15;
am_util_stdio_printf("- temp: %f\r\n", temp);
```

Ilustración 77: Lectura cmd I2C.

La variable *status*, tras realizar la lectura, contiene el estado de la misma, notificando si se hubiera producido algún error o si en su defecto, la lectura se ha realizado con éxito. Por último, se realiza la conversión pertinente a grados centígrados y se muestra, a través del terminal serie, el valor de la temperatura obtenida.

Función *ATTSCcccEnabled*:

Comprueba si está habilitado algún CCCD y si el nivel de seguridad de la Característica se ha cumplido (*ATTSCccEnabled*). Finalmente, si cumplen los requisitos, se envía la Notificación.

```
if (AttsCccEnabled(connId, DATS_WP_DAT_CCC_IDX))
{
  APP_TRACE_INFO0("ESTOY EN EL IF DE datsSendData(connId)");
  /* send notification */
 AttsHandleValueNtf(connId, WP_DAT_HDL, sizeof(cmd), (uint8_t *)cmd);
}
```

Ilustración 78: ATTSCccEnabled () dats.

Función *datsDmCback*:

Se ejecuta cuando la pila tiene un evento de gestión de dispositivo para enviar a la aplicación. La función copia los parámetros del evento a un mensaje, y envía el mensaje al gestor de eventos (Ilustración 79).

```
static void datsDmCback(dmEvt_t *pDmEvt)
{
  dmEvt_t   *pMsg;
  uint16_t  len;

  if (pDmEvt->hdr.event == DM_SEC_ECC_KEY_IND)
  {
    DmSecSetEccKey(&pDmEvt->eccMsg.data.key);

    /* If the local device sends OOB data. */
    if (datsSendOobData)
    {
      uint8_t oobLocalRandom[SMP_RAND_LEN];
      SecRand(oobLocalRandom, SMP_RAND_LEN);
      DmSecCalcOobReq(oobLocalRandom, pDmEvt->eccMsg.data.key.pubKey_x);
    }
  }
  else if (pDmEvt->hdr.event == DM_SEC_CALC_OOB_IND)
  {
    if (datsOobCfg == NULL)
    {
      datsOobCfg = WsfBufAlloc(sizeof(dmSecLescOobCfg_t));
    }

    if (datsOobCfg)
    {
      Calc128Cpy(datsOobCfg->localConfirm, pDmEvt->oobCalcInd.confirm);
      Calc128Cpy(datsOobCfg->localRandom, pDmEvt->oobCalcInd.random);
    }
  }
  else
  {
    len = DmSizeOfEvt(pDmEvt);

    if ((pMsg = WsfMsgAlloc(len)) != NULL)
    {
```

Ilustración 79: datsDmCback().

Función *datsCccCback*:

Esta función se ejecuta cuando un dispositivo emparejado escribe un nuevo valor en un CCCD en el *Servidor ATT*. También cabe la posibilidad de que la función se ejecute al establecer la conexión si el CCCD se inicializa con un valor almacenado de una conexión anterior. La función comprueba en primer lugar si este nuevo valor CCCD debe almacenarse en la base de datos del dispositivo. Si es sí, el valor se almacena. A continuación, envía un mensaje al gestor de eventos de la aplicación, con el valor CCCD. En la Ilustración 80 se muestra el código de esta función.

```
static void datsCccCback(attsCccEvt_t *pEvt)
{
  appDbHdl_t    dbHdl;
  /* If CCC not set from initialization and there's a device record and currently bonded */
  if ((pEvt->handle != ATT_HANDLE_NONE) &&
      ((dbHdl = AppDbGetHdl((dmConnId_t) pEvt->hdr.param)) != APP_DB_HDL_NONE) &&
      AppCheckBonded((dmConnId_t)pEvt->hdr.param))
  {
    /* Store value in device database. */
    AppDbSetCccTblValue(dbHdl, pEvt->idx, pEvt->value);
  }
}
```

Ilustración 80: datsCccCback().

Función *datsWpWriteCback:*

Se puede observar en la Ilustración 81, desde que recibe un dato (en este caso a través de I2C), lo escribe y lo transmite.

```
uint8_t datsWpWriteCback(dmConnId_t connId, uint16_t handle, uint8_t operation,
                         uint16_t offset, uint16_t len, uint8_t *pValue, attsAttr_t *pAttr)
{
  /* print received data */
  APP_TRACE_INFO0((const char*) pValue);

  /* send back some data */
  datsSendData(connId);

  return ATT_SUCCESS;
}
```

Ilustración 81: datsWpWriteCback().

Función *dmSecKey_t:*

Esta función, cuyo código se muestra en la Ilustración 82, se ejecuta cuando tiene un evento de gestión, extrayendo la clave de emparejamiento de un dispositivo ya almacenado, o devolviendo NULL en caso de que este dispositivo no haya sido emparejado nunca.

```
static dmSecKey_t *datsGetPeerKey(dmEvt_t *pMsg)
{
  appDbHdl_t dbHdl;

  /* get device database record handle */
  dbHdl = AppDbGetHdl((dmConnId_t) pMsg->hdr.param);

  /* if database record handle valid */
  if (dbHdl != APP_DB_HDL_NONE)
  {
    return AppDbGetKey(dbHdl, DM_KEY_IRK, NULL);
  }

  return NULL;
}
```

Ilustración 82: dmSecKey_t() dats.

Función *datsPrivAddDevToResListInd*:

Esta función se ejecuta cuando tiene un evento de gestión, determinando si se ha de añadir el dispositivo a la lista de resolución, para su inicialización (Ilustración 83).

```
static void datsPrivAddDevToResListInd(dmEvt_t *pMsg)
{
  dmSecKey_t *pPeerKey;
  /* if peer IRK present */
  if ((pPeerKey = datsGetPeerKey(pMsg)) != NULL)
  {
    /* set advertising peer address */
    AppSetAdvPeerAddr(pPeerKey->irk.addrType, pPeerKey->irk.bdAddr);
  }
}
```

Ilustración 83: datsPrivAddDevToResListInd ().

Función *datsPrivRemDevFromResListInd*:

La función se ejecuta cuando es llamada desde el gestor de eventos, siempre y cuando se necesite eliminar una dirección almacenada en la base de datos, añadiéndose ceros para inicializarlo en la base de datos (Ilustración 84).

```
static void datsPrivRemDevFromResListInd(dmEvt_t *pMsg)
{

  if (pMsg->hdr.status == HCI_SUCCESS)
  {
    if (AppDbGetHdl((dmConnId_t) pMsg->hdr.param) != APP_DB_HDL_NONE)
    {
      uint8_t addrZeros[BDA_ADDR_LEN] = { 0 };

      /* clear advertising peer address and its type */
      AppSetAdvPeerAddr(HCI_ADDR_TYPE_PUBLIC, addrZeros);
    }
  }
}
```

Ilustración 84: datsPrivRemDevFromResListInd().

Función *datsSetup*:

Esta función se ejecuta cuando se inicia la aplicación, después de restablecer la pila. Configura el proceso de *Advertising* y los datos de respuesta del proceso de *Scan*, iniciando posteriormente el proceso de *Advertising*. El dispositivo comienza el proceso *Advertising* llamando a la función *AppAdvStart().* Al utilizar el modo *auto init,* el modo para conectar, detectar o enlazar un dispositivo se establece automáticamente en función de si el dispositivo ya se ha vinculado. Si no se ha enlazado, el dispositivo se establece en modo detectable y enlazable. Si se ha enlazado, el dispositivo se establece en modo conectable y no enlazable (Ilustración 85).

```
static void datsSetup(dmEvt_t *pMsg)
{
  /* set advertising and scan response data for discoverable mode */
  AppAdvSetData(APP_ADV_DATA_DISCOVERABLE, sizeof(datsAdvDataDisc), (uint8_t *) datsAdvDataDisc);
  AppAdvSetData(APP_SCAN_DATA_DISCOVERABLE, sizeof(datsScanDataDisc), (uint8_t *) datsScanDataDisc);

  /* set advertising and scan response data for connectable mode */
  AppAdvSetData(APP_ADV_DATA_CONNECTABLE, sizeof(datsAdvDataDisc), (uint8_t *) datsAdvDataDisc);
  AppAdvSetData(APP_SCAN_DATA_CONNECTABLE, sizeof(datsScanDataDisc), (uint8_t *) datsScanDataDisc);

  /* start advertising; automatically set connectable/discoverable mode and bondable mode */
  AppAdvStart(APP_MODE_AUTO_INIT);
}
```

Ilustración 85: datsSetup().

Función *datsProcMsg()*

Se trata de la función principal para gestionar eventos de mensajes. En esta función se extrae del mensaje el tipo de evento, y se identifica de qué tipo de evento se trata, para en función del evento a tratar, llamar a las funciones correspondientes. El código de esta función se muestra en la Ilustración 86 y la Ilustración 87.

```
static void datsProcMsg(dmEvt_t *pMsg)
{
  uint8_t uiEvent = APP_UI_NONE;

  switch(pMsg->hdr.event)
  {
    case ATT_MTU_UPDATE_IND:
      APP_TRACE_INFO1("Negotiated MTU %d", ((attEvt_t *)pMsg)->mtu);
      break;

    case DM_RESET_CMPL_IND:
      AttsCalculateDbHash();
      DmSecGenerateEccKeyReq();
      datsSetup(pMsg);
      uiEvent = APP_UI_RESET_CMPL;
      break;

    case DM_ADV_START_IND:
      uiEvent = APP_UI_ADV_START;
      break;

    case DM_ADV_STOP_IND:
      uiEvent = APP_UI_ADV_STOP;
      break;

    case DM_CONN_CLOSE_IND:
      uiEvent = APP_UI_CONN_CLOSE;
      break;

    case DM_SEC_PAIR_CMPL_IND:
      DmSecGenerateEccKeyReq();
      uiEvent = APP_UI_SEC_PAIR_CMPL;
      break;

    case DM_SEC_PAIR_FAIL_IND:
      DmSecGenerateEccKeyReq();
      uiEvent = APP_UI_SEC_PAIR_FAIL;
      break;

    case DM_SEC_ENCRYPT_IND:
      uiEvent = APP_UI_SEC_ENCRYPT;
      break;

    case DM_SEC_ENCRYPT_FAIL_IND:
      uiEvent = APP_UI_SEC_ENCRYPT_FAIL;
      break;

    case DM_SEC_AUTH_REQ_IND:

      if (pMsg->authReq.oob)
      {
        dmConnId_t connId = (dmConnId_t) pMsg->hdr.param;
```

Ilustración 86: datsProcMsg().

Una vez se identifica el tipo de evento, se almacena en la variable *uiEvent* el tipo, y se llama a la función *AppUiAction*() para realizar una acción *UI* específica.

```
  if (uiEvent != APP_UI_NONE)
  {
    AppUiAction(uiEvent);
  }
}
```

Ilustración 87: uiEvent datsc.

Función *DatsHandlerInit*:

En esta función se inicializa el gestor de la aplicación, que es llamado durante la inicialización del sistema. Se introducen las diferentes configuraciones y sus punteros, además de inicializar el *Framework*, y por último, se inserta la clave IRK del dispositivo (Ilustración 88).

```
void DatsHandlerInit(wsfHandlerId_t handlerId)
{

  APP_TRACE_INFO0("DatsHandlerInit");

  /* store handler ID */
  datsCb.handlerId = handlerId;

  /* Set configuration pointers */
  pAppSlaveCfg = (appSlaveCfg_t *) &datsSlaveCfg;
  pAppAdvCfg = (appAdvCfg_t *) &datsAdvCfg;
  pAppSecCfg = (appSecCfg_t *) &datsSecCfg;
  pAppUpdateCfg = (appUpdateCfg_t *) &datsUpdateCfg;
  pSmpCfg = (smpCfg_t *) &datsSmpCfg;
  pAttCfg = (attCfg_t *) &datsAttCfg;

  /* Initialize application framework */
  AppSlaveInit();
  AppServerInit();

  /* Set IRK for the local device */
  DmSecSetLocalIrk(localIrk);
}
```

Ilustración 88: DatsHandlerInit().

Función *DatsWsfBufDiagnostics*:

Si existe un error en el buffer *WSF*, se diagnostica representando el tipo, la tarea, y su longitud (Ilustración 89).

```
static void datsWsfBufDiagnostics(WsfBufDiag_t *pInfo)

  if (pInfo->type == WSF_BUF_ALLOC_FAILED)
  {
    APP_TRACE_INFO2("Dats got WSF Buffer Allocation Failure - Task: %d Len: %d",
                    pInfo->param.alloc.taskId, pInfo->param.alloc.len);
  }
```

Ilustración 89: DatsWsfBufDiagnostics().

Función *DatsHandler*:

Se trata del gestor de la aplicación, que se encarga de procesar los mensajes *DM* relacionados con la conexión del dispositivo *Slave*, siempre que es llamada por el gestor de eventos *WSF*, como se muestra en la Ilustración 90.

```
void DatsHandler(wsfEventMask_t event, wsfMsgHdr_t *pMsg)
{
  if (pMsg != NULL)
  {
    APP_TRACE_INFO1("Dats got evt %d", pMsg->event);

    /* process ATT messages */
    if (pMsg->event >= ATT_CBACK_START && pMsg->event <= ATT_CBACK_END)
    {
      /* process server-related ATT messages */
      AppServerProcAttMsg(pMsg);
    }
    /* process DM messages */
    else if (pMsg->event >= DM_CBACK_START && pMsg->event <= DM_CBACK_END)
    {
      /* process advertising and connection-related messages */
      AppSlaveProcDmMsg((dmEvt_t *) pMsg);

      /* process security-related messages */
      AppSlaveSecProcDmMsg((dmEvt_t *) pMsg);
```

Ilustración 90: DatsHandler().

Función *DataStart*:

Se inicializa la pila de protocolos que se usa, tal y como se muestra en la Ilustración 91.

```
void DatsStart(void)
{
    am_util_stdio_printf("--- ESTOY JECUTANDO LA FUNCIÓN DatsStart(void)\r\n");

  /* Register for stack callbacks */
  DmRegister(datsDmCback);
  DmConnRegister(DM_CLIENT_ID_APP, datsDmCback);
  AttRegister(datsAttCback);
  AttConnRegister(AppServerConnCback);
  AttsCccRegister(DATS_NUM_CCC_IDX, (attsCccSet_t *) datsCccSet, datsCccCback);

  /* Initialize attribute server database */
  SvcCoreGattCbackRegister(GattReadCback, GattWriteCback);
  SvcCoreAddGroup();
  SvcWpCbackRegister(NULL, datsWpWriteCback);
  SvcWpAddGroup();

  /* Set Service Changed CCCD index. */
  GattSetSvcChangedIdx(DATS_GATT_SC_CCC_IDX);

  /* Register for app framework button callbacks */
  AppUiBtnRegister(datsBtnCback);

  WsfBufDiagRegister(datsWsfBufDiagnostics);

  /* Reset the device */
  DmDevReset();
}
```

Ilustración 91: DatsStart().

Flujograma del proceso de *Advertising:*

Para entender con mayor profundidad algunas de las funciones del dispositivo *Peripheral* implementado en el fichero *dats.c,* en la Ilustración 92 se muestra el proceso de *Advertising* en un dispositivo *Peripheral.*

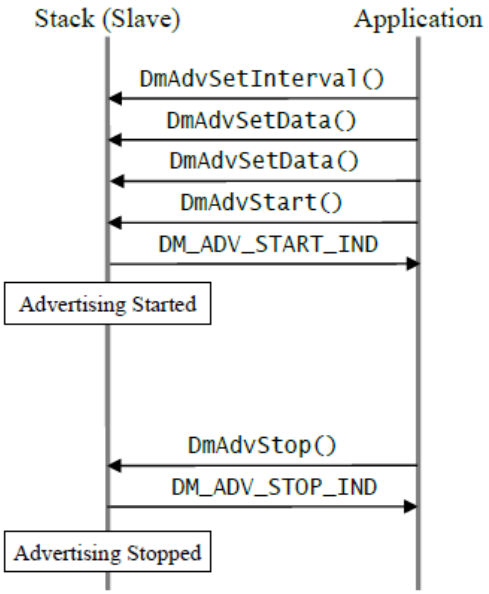


Ilustración 92: Flujograma de Advertising (DM)

Para realizar el proceso de *Advertising*, el gestor *DM* llama a las diferentes funciones que configuran el dispositivo (*datsStart()* y *datsSetup()).* Una vez configurado, la propia pila ejecuta la función *datsProcMsg(),* en la que tras analizar la información contenida en la cabecera, se localiza el tipo de evento.

Interactúa directamente con la aplicación e indica qué debe iniciar el proceso de *Advertising* (Ilustración 93). De manera continua, y tal como se configuró el procedimiento de *Adversiting* del dispositivo, el gestor *DM* detiene y activa el proceso dependiendo del intervalo configurado, haciendo llamadas a la función *datsProcMsg(),* como se observa en la ilustración 94.

```
case DM_ADV_START_IND:
  uiEvent = APP_UI_ADV_START;
  break;
```

Ilustración 93: Evento DM inicio de Advertising.

```
case DM_ADV_STOP_IND:
  uiEvent = APP_UI_ADV_STOP;
  break;
```

Ilustración 94: Evento DM finaliza Advertising.

4.1.4 Fichero *radiotask.c*

Radiotask pertenece a la serie de ficheros ubicados en la carpeta *src*, tanto para el dispositivo *Central* como para dispositivo *Peripheral*. *RadioTask* inicializa la pila, configura el subsistema BLE, e inicia el perfil BLE.

Función *Exactle_stack_init* :

Se inicia el subsistema *WSF* y se configuran los temporizadores (Ilustración 95).

```
void
exactle_stack_init(void)
{
    wsfHandlerId_t handlerId;
    uint16_t       wsfBufMemLen;
    //
    // Set up timers for the WSF scheduler.
    //
    WsfOsInit();
    WsfTimerInit();
```

Ilustración 95: exactle_stack_init Peripheral.

Se genera un conjunto de *buffers* para satisfacer las necesidades de memoria dinámica (Ilustración 96).

```
wsfBufMemLen = WsfBufInit(sizeof(g_pui32BufMem), (uint8_t *)g_pui32BufMem, WSF_BUF_POOLS,
        g_psPoolDescriptors);

if (wsfBufMemLen > sizeof(g_pui32BufMem))
{
    am_util_debug_printf("Memory pool is too small by %d\r\n",
                         wsfBufMemLen - sizeof(g_pui32BufMem));
}
```

Ilustración 96: creación buffers WSF Peripheral.

Se inicializa el Servicio de Seguridad, tal y como se muestra en la Ilustración 97 y la Ilustración 98 .

```
SecInit();
SecAesInit();
SecCmacInit();
SecEccInit();
```

Ilustración 97: Inicialización Servicio *seguridad Peripheral.*

```
handlerId = WsfOsSetNextHandler(HciHandler);
HciHandlerInit(handlerId);

handlerId = WsfOsSetNextHandler(DmHandler);
DmDevVsInit(0);
DmAdvInit();
DmConnInit();
DmConnSlaveInit();
DmSecInit();
DmSecLescInit();
DmPrivInit();
DmHandlerInit(handlerId);

handlerId = WsfOsSetNextHandler(L2cSlaveHandler);
L2cSlaveHandlerInit(handlerId);
L2cInit();
L2cSlaveInit();

handlerId = WsfOsSetNextHandler(AttHandler);
AttHandlerInit(handlerId);
AttsInit();
AttsIndInit();
AttcInit();

handlerId = WsfOsSetNextHandler(SmpHandler);
SmpHandlerInit(handlerId);
SmprInit();
SmprScInit();

handlerId = WsfOsSetNextHandler(AppHandler);
AppHandlerInit(handlerId);

ButtonHandlerId = WsfOsSetNextHandler(button_handler);

handlerId = WsfOsSetNextHandler(HciDrvHandler);
HciDrvHandlerInit(handlerId);

handlerId = WsfOsSetNextHandler(DatsHandler);
DatsHandlerInit(handlerId);
```

Ilustración 98: Inicialización de las funciones de devolución de llamada para la pila BLE Peripheral.

Función *RadioTask*:

Esta es la función principal, que se ejecutará al compilar el proyecto. Llama a la función *exactle_stack_init()* (expuesta en el punto anterior), y una vez la pila está lista, se inicia el perfil BLE configurado, como se muestra en la ilustración 99.

```
void
RadioTask(void *pvParameters)
{
    //
    // Initialize the main ExactLE stack.
    //
    exactle_stack_init();

    //
    // Enable BLE data ready interrupt
    //
//    am_hal_interrupt_enable(AM_HAL_INTERRUPT_GPIO);

    setup_buttons();

    //
    // Start the "Dats" profile.
    //
    DatsStart();

    while (1)
    {
        //
        // Calculate the elapsed time from our free-running timer, and update
        // the software timers in the WSF scheduler.
        //
        wsfOsDispatcher();

    }
```

Ilustración 99: RadioTask() Peripheral.

Fichero *Ble_freetos_dats_I2C.c:*

Contiene la función *main* del proyecto (Ilustración 100). Al igual que el fichero *RadioTask.c* se encuentra en la ruta *src*, y todos estos archivos son compilados cuando se ejecuta el *makefile*. En concreto, en este fichero se une la aplicación con el sistema operativo (RTOS).

```
int
main(void)
{
    //
    // Set the clock frequency
    //
    am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_SYSCLK_MAX, 0);

    //
    // Set the default cache configuration
    //
    am_hal_cachectrl_config(&am_hal_cachectrl_defaults);
    am_hal_cachectrl_enable();

#ifndef NOFPU
    //
    // Enable the floating point module, and configure the core for lazy
    // stacking.
    //
    am_hal_sysctrl_fpu_enable();
    am_hal_sysctrl_fpu_stacking_enable(true);
#else
    am_hal_sysctrl_fpu_disable();
#endif

    //
    // Configure the board for low power.
    //
    am_bsp_low_power_init();


#ifdef AM_PART_APOLLO
    //
    // SRAM bank power setting.
    // Need to match up with actual SRAM usage for the program
    // Current usage is between 32K and 40K - so disabling upper 3 banks
    //
    am_hal_mcuctrl_sram_power_set(AM_HAL_MCUCTRL_SRAM_POWER_DOWN_5 |
                                  AM_HAL_MCUCTRL_SRAM_POWER_DOWN_6 |
                                  AM_HAL_MCUCTRL_SRAM_POWER_DOWN_7,
                                  AM_HAL_MCUCTRL_SRAM_POWER_DOWN_5 |
                                  AM_HAL_MCUCTRL_SRAM_POWER_DOWN_6 |
                                  AM_HAL_MCUCTRL_SRAM_POWER_DOWN_7);

#if 0 // Not turning off the Flash as it may be needed to download the image
    //
    // Flash bank power set.
    //
    am_hal_mcuctrl_flash_power_set(AM_HAL_MCUCTRL_FLASH_POWER_DOWN_1);
#endif
#endif // AM_PART_APOLLO

#ifdef AM_PART_APOLLO2
#if 0 // Not turning off the Flash as it may be needed to download the image
    am_hal_pwrctrl_memory_enable(AM_HAL_PWRCTRL_MEMEN_FLASH512K);
#endif
#ifdef AM_DEBUG_PRINTF
    enable_print_interface();
#endif

    //
    // Initialize plotting interface.
    //
    am_util_debug_printf("BLE FreeRTOS DATS Example\n");

    // FBTG
    // Initialize the printf interface for UART output
    //
    am_bsp_uart_printf_enable();


    //
    // Run the application.
    //
    run_tasks();

    //
    // We shouldn't ever get here.
    //
    while (1)
    {
    }

}
```

Ilustración 100: Main() Ble_freetos_dats_i2c Peripheral.

Se configuran todos los aspectos físicos, como la frecuencia de reloj, o la memoria dinámica, y se llama la función *run_task()* implementada en el fichero *RTOS.c.*

4.1.5 Fichero *RTOS.c*

Este fichero determina todos los aspectos relacionados con la gestión y la configuración del sistema operativo en tiempo real (*Real Time Operative System*), siendo el encargado de llevar la línea de ejecución decidiendo cuándo se ejecuta una tarea de acuerdo a una serie de prioridades.

En la Ilustración 101 se muestra la función *am_ctimer_isr*, en la que se definen una serie de interrupciones asociadas a diferentes temporizadores, así como los manejadores asociados a cada uno de ellos.

```
void
am_ctimer_isr(void)
{
    uint32_t ui32Status;

    //
    // Check the timer interrupt status.
    //
    ui32Status = am_hal_ctimer_int_status_get(false);
    am_hal_ctimer_int_clear(ui32Status);

    //
    // Run handlers for the various possible timer events.
    //
    am_hal_ctimer_int_service(ui32Status);
}
```

Ilustración 101: timer interrupciones Peripheral.

Función para gestionar las tareas IDLE. Realiza las operaciones de apagado específicas de la aplicación, devolviendo 0 si esta función incorpora WFI, y en caso contrario devuelve el valor introducido como parámetro.

```
uint32_t am_freertos_sleep(uint32_t idleTime)
{
    am_hal_sysctrl_sleep(AM_HAL_SYSCTRL_SLEEP_DEEP);
    return 0;
}
```

Ilustración 102: Función IDLE.

Al contrario de la anterior función, realiza la operación de “despertar”, necesaria por ejemplo, para encender/activar periféricos, etc (Ilustración 103).

```
void am_freertos_wakeup(uint32_t idleTime)
{
    return;
}
```

Ilustración 103: am_freertos_wakeup().

En caso de fallo ante una llamada a la función *pvPortMalloc*() porque no haya suficiente memoria libre disponible en el *heap* de *FreeRTOS*, esta función es llamada internamente por las funciones de la API de *FreeRTOS* que crean tareas, colas, temporizadores de software y semáforos. El tamaña del *heap* se establece a partir del parámetro *TOTAL_HEAP_SIZE* definido en *FreeRTOSConfig.h* (Ilustración 104 e Ilustración 105).

```
void
vApplicationMallocFailedHook(void)
{
    while (1);
}
```

Ilustración 104: Malloc() Peripheral.

```
void
vApplicationStackOverflowHook(TaskHandle_t pxTask, char *pcTaskName)
{
    (void) pcTaskName;
    (void) pxTask;

    //
    // Run time stack overflow checking is performed if
    // configconfigCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2.  This hook
    // function is called if a stack overflow is detected.
    //
    while (1)
    {
        __asm("BKPT #0\n") ; // Break into the debugger
    }
}
```

Ilustración 105: overFlowHook().

Función *setup_task*:

La función mostrada en la Ilustración 106 se usa para cualquier inicialización global, ejecutándose después de inicializar el *Sheduler* (Planificador), pero antes de que se inicie cualquier tarea funcional. Puede ser útil para habilitar eventos, semáforos y otras características específicas del RTOS. Concretamente, ejecutará la función *RadiotaskSetup().*

```
void
setup_task(void *pvParameters)
{
    //
    // Print a debug message.
    //
    am_util_debug_printf("Running setup tasks...\r\n");

    //
    // Run setup functions.
    //
    RadioTaskSetup();

    //
    // Create the functional tasks
    //
    xTaskCreate(RadioTask, "RadioTask", 512, 0, 3, &radio_task_handle);
    //
    // The setup operations are complete, so suspend the setup task now.
    //
    vTaskSuspend(NULL);

    while (1);
}
```

Ilustración 106: setup_task ().

Función *run_task*:

La función *run_task()* establece algunas prioridades de interrupción antes de crear tareas, e inicia el *Sheduler* (planificador de tareas), como se observa en la ilustración 107.

```
void
run_tasks(void)
{
    //
    // Set some interrupt priorities before we create tasks or start the scheduler.
    //
    // Note: Timer priority is handled by the FreeRTOS kernel, so we won't
    // touch it here.
    //

    //
    // Create essential tasks.
    //
    xTaskCreate(setup_task, "Setup", 512, 0, 3, &xSetupTask);


    //
    // Start the scheduler.
    //
    vTaskStartScheduler();
}
```

Ilustración 107: run_task() Peripheral.

4.1.6 Dispositivo *Peripheral* funcional

Tras analizar los ficheros desarrollados para la implementación del dispositivo *Peripheral* en el presente TFG, en esta sección se mostrarán los resultados obtenidos. La Ilustración 108 se muestra cómo se ha conectado la placa *Artemis Thing Plus* al *SWO J-Link* y al sensor de temperatura MLX90614.

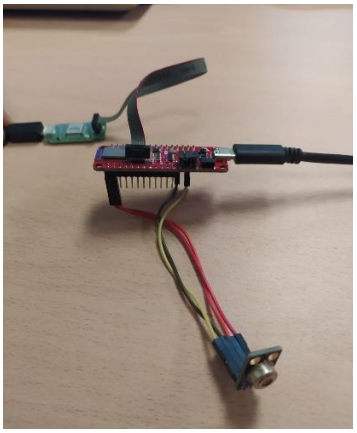


Ilustración 108: Dispositivo Peripheral para compilar y programar la plataforma con el código desarrollado.

A continuación, se ejecuta el *makefile* situado en la ruta *master\boards_sfe\common\examples\ble_freertos_dats_I2C_ok_v01\gcc,* como se muestra en el capítulo anterior. Las pruebas funcionales del dispositivo *Peripheral* se realizan mediante la aplicación *NRF Connect,* que simulará un dispositivo *Central* en un teléfono móvil, además de abrir el depurador y la aplicación *Putty* para visualizar todo el proceso.

Al conectar y ejecutar el dispositivo *Peripheral,* éste empezará a realizar el proceso de *Advertising*, pudiendo realizarse con la aplicación *NRF connect* se el *Scanning* en el dispositivo *Central,* como se muestra en la Ilustración 109.

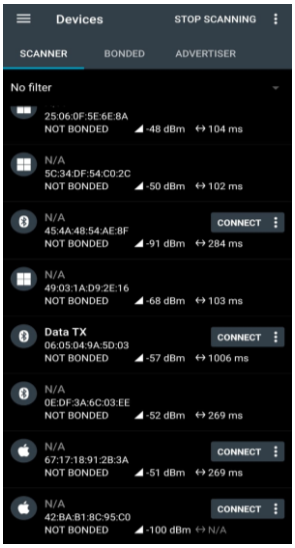


Ilustración 109: NRF Connect Scanner

A continuación se prueba a conectar al dispositivo *Peripheral* llamado "*Data TX",* como se estableció en el fichero *dats_I2C.c*. Tras realizar la conexión, se pueden visualizar los Servicios y Características obtenidas tras el proceso de descubrimiento, siendo el último Servicio el que permite indicarle al dispositivo *Peripheral* que envíe la información proporcionada por el sensor de temperatura, como se observa en la Ilustración 110.

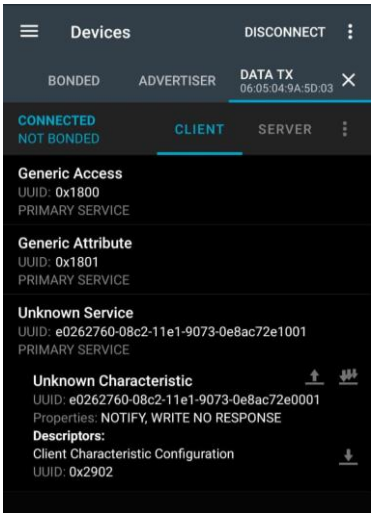


Ilustración 110: NRF Connect Servicios *disponibles*

Para ello se debe escribir un valor cualquiera en la Característica asociada a este Servicio, y enviarlo al dispositivo *Peripheral*. Se sigue el proceso descrito en la Ilustración 111.

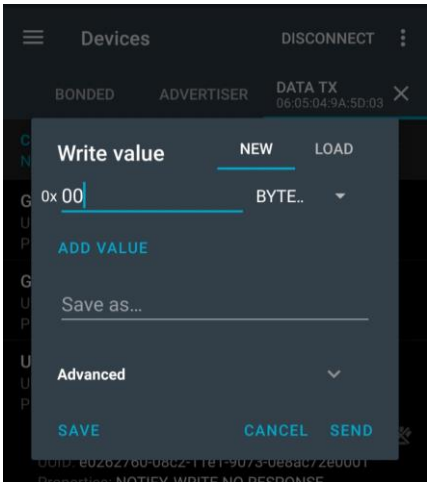


Ilustración 111: NRF Connect escritura desde el dispositivo Central

Tal y como está configurado el dispositivo *Peripheral*, enviará como respuesta una Notificación que contiene el valor de la temperatura medida, como se muestra en la Ilustración 112.

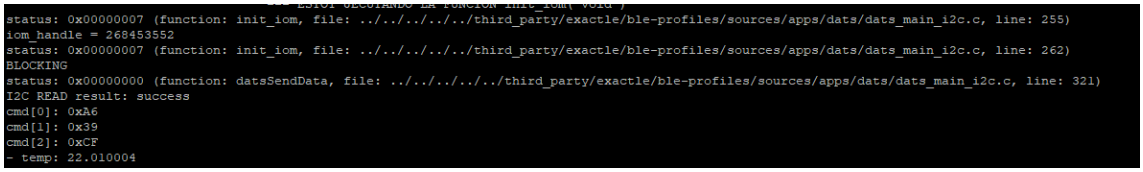


Ilustración 112: UART desde el dispositivo Peripheral

4.2 Implementación del dispositivo *Central*

En este apartado se analizan los ficheros desarrollados para implementar la funcionalidad al dispositivo *Central*. Al igual que el dispositivo *Peripheral,* se basa en diferentes ficheros de perfil BLE *Cordio*, aunque en este caso se deberá configurar como *Master* de la conexión. En los siguientes apartados se presentarán los ficheros que contienen el código que implementa la funcionalidad del dispositivo *Central* en el presente TFG.

4.2.1 Fichero *Datc.c*

En muchos aspectos, es similar al código usado para implementar el dispositivo *Peripheral,* con la diferencia principal asociada al funcionamiento de un dispositivo *Central*, en el que se deberá extraer la información que recibe del dispositivo *Peripheral* mediante la interacción con la pila de protocolos *BLE*, además de la configuración de diferentes parámetros. La Ilustración 113, la Ilustración 114, la Ilustración 115, la Ilustración 116, la Ilustración 117, la Ilustración 118 y la Ilustración 119 muestran todos los parámetros configurados en el dispositivo *Central* desarrollado.

Parámetros principales del dispositivo *Master*:

```
static const appMasterCfg_t datcMasterCfg =
{
  96,                                   /*! The scan interval, in 0.625 ms units */
  48,                                   /*! The scan window, in 0.625 ms units  */
  4000,                                 /*! The scan duration in ms */
  DM_DISC_MODE_NONE,                    /*! The GAP discovery mode */
  DM_SCAN_TYPE_ACTIVE                   /*! The scan type (active or passive) */
};
```

Ilustración 113: Parámetros dispositvo Master.

Se definen los parámetros del proceso de *Scanning*, cómo y cada cuánto tiempo atenderá el dispositivo *Central* a los paquetes de *Advertising.*

- Intervalo de *Scan* de duración 60 ms.
- Ventana de *Scan* de 30 ms.
- Duración del *Scan* de 4000 ms.
- Los dispositivos que realizan la detección de GAP no pueden descubrir este dispositivo.
- Tipo de *Scan,* Activo.

```
static const appSecCfg_t datcSecCfg =
{
  DM_AUTH_BOND_FLAG | DM_AUTH_SC_FLAG,    /*! Authentication and bonding flags */
  DM_KEY_DIST_IRK,                        /*! Initiator key distribution flags */
  DM_KEY_DIST_LTK | DM_KEY_DIST_IRK,      /*! Responder key distribution flags */
  FALSE,                                  /*! TRUE if Out-of-band pairing data is present */
  FALSE                                    /*! TRUE to initiate security upon connection */
};
```

Ilustración 114: Parámetros de seguridad Central.

Parámetros de seguridad usados para la conexión, establecidos por defecto. A destacar que los emparejamientos fuera de banda (*out of band)* están desactivados, ya que se trabaja en las bandas por defecto.

```
static const hciConnSpec_t datcConnCfg =
{
  40,                                     /*! Minimum connection interval in 1.25ms units */
  40,                                     /*! Maximum connection interval in 1.25ms units */
  0,                                      /*! Connection latency */
  600,                                    /*! Supervision timeout in 10ms units */
  0,                                      /*! Unused */
  0                                       /*! Unused */
};
```

Ilustración 115: Parámetros de conexión Central.

Los parámetros de la conexión, que tras el proceso de emparejamiento impone el dispositivo *Central* al dispositivo *Peripheral*.

```
static const appDiscCfg_t datcDiscCfg =
{
  FALSE,                                  /*! TRUE to wait for a secure connection before initiating discovery */
  TRUE                                    /*! TRUE to fall back on database hash to verify handles when no bond exists. */
};

static const appCfg_t datcAppCfg =
{
  FALSE,                                  /*! TRUE to abort service discovery if service not found */
  TRUE                                    /*! TRUE to disconnect if ATT transaction times out */
};
```

Ilustración 116: Descubrimiento de Servicios*.*

Parámetros usados para el descubrimiento de Servicios y Características del dispositivo *Slave*. Siempre que *appDiscCfg_t* adquiera el valor TRUE, se debe esperar una conexión segura antes de iniciar la detección. La variable *appCfg_t*, relacionada con los parámetros de configuración de la aplicación, adquiere el valor TRUE si se debe abortar el descubrimiento de un Servicio o desconectar cuando en el *ATT* se cumple el *timeout* establecido.

```
static const attCfg_t datcAttCfg =
{
  15,                                     /* ATT server service discovery connection idle timeout in seconds */
  241,                                    /* desired ATT MTU */
  23,                                     /* desired ATT MTU */
  ATT_MAX_TRANS_TIMEOUT,                  /* transcation timeout in seconds */
  4                                       /* number of queued prepare writes supported by server */
};

/*! local IRK */
static uint8_t localIrk[] =
{
  0xA6, 0xD9, 0xFF, 0x70, 0xD6, 0x1E, 0xF0, 0xA4, 0x46, 0x5F, 0x8D, 0x68, 0x19, 0xF3, 0xB4, 0x96
};
```

Ilustración 117: Parámetros ATT Central.

Configuración de los parámetros que usa el *Servidor ATT*, tales como el tamaño del MTU a utilizar, la duración de los *timeouts,* o el número máximo de colas soportadas para escribir por el *Servidor ATT*. Por último, se declara la clave *IRK* de este dispositivo.

Cliente ATT:

```
/**************************************************************************************************
  ATT Client Discovery Data
**************************************************************************************************/

/*! Discovery states:  enumeration of services to be discovered */
enum
{
  DATC_DISC_GATT_SVC,      /*! GATT service */
  DATC_DISC_GAP_SVC,       /*! GAP service */
  DATC_DISC_WP_SVC,        /*! Arm Ltd. proprietary service */
};

/*! the Client handle list, datcCb.hdlList[], is set as follows:
 *
 *  ------------------------------- <- DATC_DISC_GATT_START
 *  | GATT svc changed handle     |
 *  -------------------------------
 *  | GATT svc changed ccc handle |
 *  ------------------------------- <- DATC_DISC_GAP_START
 *  | GAP central addr res handle |
 *  -------------------------------
 *  | GAP RPA Only handle         |
 *  ------------------------------- <- DATC_DISC_WP_START
 *  | WP handles                  |
 *  | ...                         |
 *  -------------------------------
 */

/*! Start of each service's handles in the the handle list */
#define DATC_DISC_GATT_START       0
#define DATC_DISC_GAP_START        (DATC_DISC_GATT_START + GATT_HDL_LIST_LEN)
#define DATC_DISC_WP_START         (DATC_DISC_GAP_START + GAP_HDL_LIST_LEN)

/*! Pointers into handle list for each service's handles */
static uint16_t *pDatcGattHdlList[DM_CONN_MAX];
static uint16_t *pDatcGapHdlList[DM_CONN_MAX];
static uint16_t *pDatcWpHdlList[DM_CONN_MAX];
```

Ilustración 118: Cliente ATT datc.

Define e inicializa los Servicios usados para el descubrimiento de datos, además de generar los punteros de los tres Servicios que se usarán.

```
/**************************************************************************************************
  ATT Client Configuration Data
**************************************************************************************************/

/*
 * Data for configuration after service discovery
 */

/* Default value for CCC indications */
const uint8_t datcCccIndVal[2] = {UINT16_TO_BYTES(ATT_CLIENT_CFG_INDICATE)};

/* Default value for CCC notifications */
const uint8_t datcCccNtfVal[2] = {UINT16_TO_BYTES(ATT_CLIENT_CFG_NOTIFY)};

/* Default value for Client Supported Features (enable Robust Caching) */
const uint8_t datcCsfVal[1] = {ATTS_CSF_ROBUST_CACHING};

/* List of characteristics to configure after service discovery */
static const attcDiscCfg_t datcDiscCfgList[] =
{
  /* Write:  GATT service changed ccc descriptor */
  {datcCccIndVal, sizeof(datcCccIndVal), (GATT_SC_CCC_HDL_IDX + DATC_DISC_GATT_START)},

  /* Write:  GATT client supported features */
  {datcCsfVal, sizeof(datcCsfVal), (GATT_CSF_HDL_IDX + DATC_DISC_GATT_START)},

  /* Write:  Proprietary data service changed ccc descriptor */
  {datcCccNtfVal, sizeof(datcCccNtfVal), (WPC_P1_NA_CCC_HDL_IDX + DATC_DISC_WP_START)},

/* Characteristic configuration list length */
#define DATC_DISC_CFG_LIST_LEN   (sizeof(datcDiscCfgList) / sizeof(attcDiscCfg_t))

/* sanity check:  make sure configuration list length is <= handle list length */
WSF_CT_ASSERT(DATC_DISC_CFG_LIST_LEN <= DATC_DISC_HDL_LIST_LEN);

/**************************************************************************************************/
```

Ilustración 119: Configuración de datos ATT datc.

Se configura cómo se tratan los datos, una vez finalizado el proceso de descubrimiento.

Función *datcDmCback:*

Se ejecuta cuando la pila tiene un evento de gestión de dispositivo para enviar a la aplicación. La función copia los parámetros del evento a un mensaje, y envía el mensaje al gestor de eventos (Ilustración 120).

```
static void datcDmCback(dmEvt_t *pDmEvt)
{
  dmEvt_t   *pMsg;
  uint16_t  len;
  uint16_t  reportLen;

  if (pDmEvt->hdr.event == DM_SEC_ECC_KEY_IND)
  {
    DmSecSetEccKey(&pDmEvt->eccMsg.data.key);

  }
  else if (pDmEvt->hdr.event == DM_SEC_CALC_OOB_IND)
  {
    if (datcOobCfg == NULL)
    {
      datcOobCfg = WsfBufAlloc(sizeof(dmSecLescOobCfg_t));
    }

    if (datcOobCfg)
    {
      Calc128Cpy(datcOobCfg->localConfirm, pDmEvt->oobCalcInd.confirm);
      Calc128Cpy(datcOobCfg->localRandom, pDmEvt->oobCalcInd.random);
    }
  }
  else
  {
    len = DmSizeOfEvt(pDmEvt);

    if (pDmEvt->hdr.event == DM_SCAN_REPORT_IND)
    {
      reportLen = pDmEvt->scanReport.len;
    }
    else
    {
      reportLen = 0;
    }

    if ((pMsg = WsfMsgAlloc(len + reportLen)) != NULL)
    {
      memcpy(pMsg, pDmEvt, len);
      if (pDmEvt->hdr.event == DM_SCAN_REPORT_IND)
      {
        pMsg->scanReport.pData = (uint8_t *) ((uint8_t *) pMsg + len);
        memcpy(pMsg->scanReport.pData, pDmEvt->scanReport.pData, reportLen);
      }
      WsfMsgSend(datcCb.handlerId, pMsg);
    }
  }
}
```

Ilustración 120: datcDmCback().

Función *datcATTCback:*

Esta función de *callback* envía los eventos *ATT* a la aplicación *Cliente*. Se utiliza una única función de *callback* tanto para *ATTS* como para *ATTC*, función descrita en la Ilustración 121.

```
static void datcAttCback(attEvt_t *pEvt)
{
  attEvt_t *pMsg;

  if ((pMsg = WsfMsgAlloc(sizeof(attEvt_t) + pEvt->valueLen)) != NULL)
  {
    memcpy(pMsg, pEvt, sizeof(attEvt_t));
    pMsg->pValue = (uint8_t *) (pMsg + 1);
    memcpy(pMsg->pValue, pEvt->pValue, pEvt->valueLen);
    WsfMsgSend(datcCb.handlerId, pMsg);
  }
}
```

Ilustración 121: datcATTCback().

Función datcScanStart:

Cuando es llamada por el gestor de eventos, inicia el proceso de *Scanning* (Ilustración 122*).*

```
static void datcScanStart(dmEvt_t *pMsg)
{
  if (pMsg->hdr.status == HCI_SUCCESS)
  {
    datcCb.scanning = TRUE;
  }
}
```

Ilustración 122: dat cScanStart().

Función *datscScanStop:*

Cuando el gestor de eventos invoca a esta función, el proceso de *Scanning* se detiene, y abre la conexión con el objetivo de emparejar ambos dispositivos, como se muestra en la Ilustración 123.

```
tatic void datcScanStop(dmEvt_t *pMsg)

 if (pMsg->hdr.status == HCI_SUCCESS)
 {
   datcCb.scanning = FALSE;
   datcCb.autoConnect = FALSE;

   /* Open connection */
   if (datcConnInfo.doConnect)
   {
     AppConnOpen(datcConnInfo.addrType, datcConnInfo.addr, datcConnInfo.dbHdl);
     datcConnInfo.doConnect = FALSE;
   }
 }
```

Ilustración 123: datcScanStop().

Función *datcScanReport*:

Analiza el tipo de conexión, y en función de ello realiza la acción correspondiente (Ilustración 124). Si ya se ha vinculado con este dispositivo, entonces realiza directamente la conexión.

```
static void datcScanReport(dmEvt_t *pMsg)
{
  uint8_t *pData;
  appDbHdl_t dbHdl;
  bool_t  connect = FALSE;

  /* disregard if not scanning or autoconnecting */
  if (!datcCb.scanning || !datcCb.autoConnect)
  {
    return;
  }

  /* if we already have a bond with this device then connect to it */
  if ((dbHdl = AppDbFindByAddr(pMsg->scanReport.addrType, pMsg->scanReport.addr)) != APP_DB_HDL_NONE)
  {
    /* if this is a directed advertisement where the initiator address is an RPA */
    if (DM_RAND_ADDR_RPA(pMsg->scanReport.directAddr, pMsg->scanReport.directAddrType))
    {
      /* resolve direct address to see if it's addressed to us */
      AppMasterResolveAddr(pMsg, dbHdl, APP_RESOLVE_DIRECT_RPA);
    }
    else
    {
      connect = TRUE;
    }
```

Ilustración 124: datcScanReport().

Si se trata de un dispositivo con RPA (*Resolvable Private Address*), la aplicación intentará resolverla (Ilustración 125).

```
else if (DM_RAND_ADDR_RPA(pMsg->scanReport.addr, pMsg->scanReport.addrType))
{
  /* resolve advertiser's RPA to see if we already have a bond with this device */
  AppMasterResolveAddr(pMsg, APP_DB_HDL_NONE, APP_RESOLVE_ADV_RPA);
}
```

Ilustración 125: RPA.

Evalúa si se ha podido conectar, y en tal caso se interrumpe el proceso de *Scanning* y se almacena la información esencial para realizar la conexión (Ilustración 126).

```
  if (connect)
  {
    /* stop scanning and connect */
    datcCb.autoConnect = FALSE;
    AppScanStop();

    /* Store peer information for connect on scan stop */
    datcConnInfo.addrType = DmHostAddrType(pMsg->scanReport.addrType);
    memcpy(datcConnInfo.addr, pMsg->scanReport.addr, sizeof(bdAddr_t));
    datcConnInfo.dbHdl = dbHdl;
    datcConnInfo.doConnect = TRUE;
  }
}
```

Ilustración 126: Conexión y guardado.

Función *datcValueNtf:*

Función que procesa y muestra por la interfaz serie la información recibida del dispositivo *Peripheral*. Específicamente sigue el mismo proceso que la función *datsSendata()* (fichero *dats.c*). Se muestra un segmento del código correspondiente en la Ilustración 127.

```
static void datcValueNtf(attEvt_t *pMsg)
{
  /* print the received data */
  //FBIG
    APP_TRACE_INFO0("");
    APP_TRACE_INFO1("I2C READ result (valueLen = %d):", pMsg->valueLen);
    APP_TRACE_INFO1("cmd[0]: 0x%02X", pMsg->pValue[0]);
    APP_TRACE_INFO1("cmd[1]: 0x%02X", pMsg->pValue[1]);
    APP_TRACE_INFO1("cmd[2]: 0x%02X", pMsg->pValue[2])
    uint16_t ret;
    float temp;
    ret = pMsg->pValue[0];
    ret |= pMsg->pValue[1] << 8;
    temp = ret;
    temp *= .02;
    temp -= 273.15;
    APP_TRACE_INFO1("- temp: %f\n", temp);
}
```

Ilustración 127: datcValueNtf ().

Función *datcSetup*:

En esta función se establecen todos los parámetros que se deben modificar después de inicializar el dispositivo, tal y como se muestra Ilustración 128.

```
static void datcSetup(dmEvt_t *pMsg)
{
  datcCb.scanning = FALSE;
  datcCb.autoConnect = FALSE;
  datcConnInfo.doConnect = FALSE;

  DmConnSetConnSpec((hciConnSpec_t *) &datcConnCfg);
}
```

Ilustración 128: datcSetup().

Función *datcSendData*:

Cuando el gestor del dispositivo llame a esta función, se escribe en el *ATT* la respuesta "ok" (Ilustración 129), que se envía al dispositivo *Peripheral*. Como se comentó en el punto anterior, una vez el dispositivo *Peripheral* recibe cualquier mensaje, devolverá el valor que le llegue desde el sensor de temperatura a través de la interfaz I2C. Cada vez que el dispositivo *Central* necesite obtener la temperatura medida desde el sensor conectado al dispositivo *Peripheral,* llama a esta función.

```
static void datcSendData(dmConnId_t connId)
{
  uint8_t str[] = "ok";

  if (pDatcWpHdlList[connId-1][WPC_P1_DAT_HDL_IDX] != ATT_HANDLE_NONE)
  {
    AttcWriteCmd(connId, pDatcWpHdlList[connId-1][WPC_P1_DAT_HDL_IDX], sizeof(str), str);
  }
}
```

Ilustración 129: datcSendData().

Función *datcDiscGapCmpl:*

La pila llamará a esta función cuando se haya completado el descubrimiento de Servicios y se trate de una RPA ya resuelta, introduciendo en la base de datos la dirección (Ilustración 130).

```
static void datcDiscGapCmpl(dmConnId_t connId)
{
  appDbHdl_t dbHdl;

  /* if RPA Only attribute found on peer device */
  if ((pDatcGapHdlList[connId-1][GAP_RPAO_HDL_IDX] != ATT_HANDLE_NONE) &&
      ((dbHdl = AppDbGetHdl(connId)) != APP_DB_HDL_NONE))
  {
    /* update DB */
    AppDbSetPeerRpao(dbHdl, TRUE);
  }
}
```

Ilustración 130: datcDiscGapCmpl().

Función *DatcDiscCback*:

Lee el estado de la aplicación y realiza acciones acordes al estado en el que se encuentra el proceso de descubrimiento (Ilustración 131 e Ilustración 132).

```
static void datcDiscCback(dmConnId_t connId, uint8_t status)
{
  switch(status)
  {
    case APP_DISC_INIT:
      /* set handle list when initialization requested */
      AppDiscSetHdlList(connId, datcCb.hdlListLen, datcCb.hdlList[connId-1]);
      break;

    case APP_DISC_READ_DATABASE_HASH:
      /* Read peer's database hash */
      AppDiscReadDatabaseHash(connId);
      break;

    case APP_DISC_SEC_REQUIRED:
      /* initiate security */
      AppMasterSecurityReq(connId);
      break;

    case APP_DISC_START:
      /* initialize discovery state */
      datcCb.discState[connId-1] = DATC_DISC_GATT_SVC;

      /* discover GATT service */
      GattDiscover(connId, pDatcGattHdlList[connId-1]);
      break;

    case APP_DISC_FAILED:
      if (pAppCfg->abortDisc)
      {
        /* if discovery failed for proprietary data service then disconnect */
        if (datcCb.discState[connId-1] == DATC_DISC_WP_SVC)
        {
          AppConnClose(connId);
          break;
        }
      }
      /* Else falls through. */

    case APP_DISC_CMPL:
      /* next discovery state */
      datcCb.discState[connId-1]++;
```

Ilustración 131: datcDiscCback().

```
  else if (datcCb.discState[connId-1] == DATC_DISC_WP_SVC)
  {
    /* discover proprietary data service */
    WpcP1Discover(connId, pDatcWpHdlList[connId-1]);
  }
  else
  {
    /* discovery complete */
    AppDiscComplete(connId, APP_DISC_CMPL);

    /* GAP service discovery completed */
    datcDiscGapCmpl(connId);

    /* start configuration */
    AppDiscConfigure(connId, APP_DISC_CFG_START, DATC_DISC_CFG_LIST_LEN,
                     (attcDiscCfg_t *) datcDiscCfgList, DATC_DISC_HDL_LIST_LEN, datcCb.hdlList[connId-1]);
  }
  break;

case APP_DISC_CFG_START:
    /* start configuration */
    AppDiscConfigure(connId, APP_DISC_CFG_START, DATC_DISC_CFG_LIST_LEN,
                     (attcDiscCfg_t *) datcDiscCfgList, DATC_DISC_HDL_LIST_LEN, datcCb.hdlList[connId-1]);
  break;

case APP_DISC_CFG_CMPL:
  AppDiscComplete(connId, status);
  break;

case APP_DISC_CFG_CONN_START:
  /* no connection setup configuration */
  break;

default:
  break;
}
```

Ilustración 132: datcDiscCback() (2).

En la Tabla 7 se enumeran los diferentes estados de la aplicación.

Tabla 7 : APP_DISC

Nombre	Descripción
APP_DISC_INIT	Descubrimiento o configuración completo.
APP_DISC_SEC_REQUIRED	Requiere proceso de seguridad para completar configuración
APP_DISC_START	Comienza el Servicio de descubrimiento
APP_DISC_CMPL	Completado el Servicio de descubrimiento
APP_DISC_FAILED	Fallo en el Servicio de descubrimiento
APP_DISC_CFG_START	Comienza el Servicio de configuración
APP_DISC_CFG_CONN_START	Comienza la instalación de la configuración para la conexión
APP_DISC_CFG_CMPL	Completado el Servicio de configuración

Función *datcProcMsg*:

Como ocurre en el fichero del dispositivo *Peripheral,* en esta función, dependiendo del evento que active el gestor del dispositivo, se analiza y se realizan acciones dependiendo del tipo de evento que se produzca, como se muestra en el código de la Ilustración 133. Tras identificar el tipo de evento, introduce en la variable *uiEvent* el evento equivalente en la aplicación para que se procesado. En la Tabla 8 se muestran los eventos más importantes para interactuar con la aplicación.

```
static void datcProcMsg(dmEvt_t *pMsg)
{
  uint8_t uiEvent = APP_UI_NONE;

  switch(pMsg->hdr.event)
  {
    case ATTC_HANDLE_VALUE_NTF:
      datcValueNtf((attEvt_t *) pMsg);
      break;

    case DM_RESET_CMPL_IND:
      AttsCalculateDbHash();
      DmSecGenerateEccKeyReq();
      datcSetup(pMsg);
//FBTG
      BleMenuInit();
      uiEvent = APP_UI_RESET_CMPL;
      break;

    case DM_SCAN_START_IND:
      datcScanStart(pMsg);
      uiEvent = APP_UI_SCAN_START;
      break;

    case DM_SCAN_STOP_IND:
      datcScanStop(pMsg);
      uiEvent = APP_UI_SCAN_STOP;
      am_menu_printf("scan stop\r\n");
      break;

    case DM_SCAN_REPORT_IND:
      datcScanReport(pMsg);
      break;

    case DM_CONN_OPEN_IND:
      datcOpen(pMsg);
      am_menu_printf(" Connection opened\r\n");
      uiEvent = APP_UI_CONN_OPEN;
      break;

    case DM_CONN_CLOSE_IND:
      am_menu_printf(" Connection closed\r\n");

      uiEvent = APP_UI_CONN_CLOSE;
      break;

    case DM_SEC_PAIR_CMPL_IND:
      DmSecGenerateEccKeyReq();
      uiEvent = APP_UI_SEC_PAIR_CMPL;
      break;

    case DM_SEC_PAIR_FAIL_IND:
      DmSecGenerateEccKeyReq();
      uiEvent = APP_UI_SEC_PAIR_FAIL;
      break;
```

Ilustración 133: datcProcMsg().

Tabla 8: Eventos de aplicación.

Nombre	Descripción
APP_UI_NONE	No hay evento
APP_UI_RESET_CMPL	Reseteo completado
APP_UI_DISCOVERABLE	Activo el modo descubrimiento
APP_UI_ADV_START	Comienza el proceso de *Advertising*
APP_UI_ADV_STOP	Finaliza el proceso de *Advertising*
APP_UI_SCAN_START	Comienza proceso de *Scanning*
APP_UI_SCAN_STOP	Finaliza proceso de *Scanning*
APP_UI_SCAN_REPORT	Datos de escaneo recibidos del dispositivo emparejado
APP_UI_CONN_OPEN	Conexión abierta
APP_UI_CONN_CLOSE	Conexión cerrada
APP_UI_SEC_PAIR_CMPL	Emparejamiento completado
APP_UI_SEC_PAIR_FAIL	Emparejamiento fallido
APP_UI_SEC_ENCRYPT	Conexión encriptada
APP_UI_SEC_ENCRYPT_FAIL	Fallo al encriptar la conexión.
APP_UI_PASSKEY_PROMPT	Pedir al usuario que introduzca la clave de acceso
APP_UI_ALERT_CANCEL	Cancelar alerta
APP_UI_ALERT_LOW	Alerta baja
APP_UI_ALERT_HIGH	Alerta alta

Función *Datchandler:*

Se trata del gestor de la aplicación que se encarga de procesar los mensajes *DM* relacionados con la conexión del dispositivo *Master*. Está definido por la sección de código expuesta en la Ilustración 134.

```
void DatcHandler(wsfEventMask_t event, wsfMsgHdr_t *pMsg)
{
  if (pMsg != NULL)
  {
    {
      APP_TRACE_INFO1("Datc got evt %d", pMsg->event);
    }

    /* process ATT messages */
    if (pMsg->event <= ATT_CBACK_END)
    {
      /* process discovery-related ATT messages */
      AppDiscProcAttMsg((attEvt_t *) pMsg);

      /* process server-related ATT messages */
      AppServerProcAttMsg(pMsg);
    }
    /* process DM messages */
    else if (pMsg->event <= DM_CBACK_END)
    {
      /* process advertising and connection-related messages */
      AppMasterProcDmMsg((dmEvt_t *) pMsg);

      /* process security-related messages */
      AppMasterSecProcDmMsg((dmEvt_t *) pMsg);

      /* process discovery-related messages */
      AppDiscProcDmMsg((dmEvt_t *) pMsg);
    }

    /* perform profile and user interface-related operations */
    datcProcMsg((dmEvt_t *) pMsg);

  }
}
```

Ilustración 134: DatcHandler().

Función *datcInitSvcHdlList*:

Se inicializan los punteros de cada Servicio (Ilustración 135).

```
static void datcInitSvcHdlList()
{
  uint8_t i;

  for (i=0; i<DM_CONN_MAX; i++)
  {
    /*! Pointers into handle list for each service's handles */
    pDatcGattHdlList[i] = &datcCb.hdlList[i][DATC_DISC_GATT_START];
    pDatcGapHdlList[i] = &datcCb.hdlList[i][DATC_DISC_GAP_START];
    pDatcWpHdlList[i] = &datcCb.hdlList[i][DATC_DISC_WP_START];
}
```

Ilustración 135: datcInitSvcHdlList().

Función *DatcStart:*

Llama a las funciones que inicializan y registran todos los procesos necesarios para dar comienzo al funcionamiento del dispositivo *Central.*

```
void DatcStart(void)
{
  /* Initialize handle pointers */
  datcInitSvcHdlList();

  /* Register for stack callbacks */
  DmRegister(datcDmCback);
  DmConnRegister(DM_CLIENT_ID_APP, datcDmCback);
  AttRegister(datcAttCback);
  /* Register for app framework discovery callbacks */
  AppDiscRegister(datcDiscCback);

  /* Initialize attribute server database */
  SvcCoreAddGroup();


  /* Reset the device */
  DmDevReset();
}
```

Ilustración 136: DatcStart().

4.2.2 Fichero Ble_menu.c

A diferencia del dispositivo *Peripheral*, en el dispositivo *Central* se ha creado un menú a partir del que, mediante la conexión UART, se puede interactuar con la conexión BLE y solicitar la lectura de diferentes Características y Servicios definidos en el dispositivo *Peripheral*.

Variables y texto del menú:

Se declaran las variables que contienen el texto plano del menú con las diferentes opciones, como se representa en la Ilustración 137.

```
char menuRxData[20];
uint32_t menuRxDataLen = 0;

sBleMenuCb bleMenuCb;

char mainMenuContent[BLE_MENU_ID_MAX][32] = {
    "1. BLE_SCAN+CONN",
    "2. BLE_SEND",
};

char gapMenuContent[GAP_MENU_ID_MAX][32] = {
    "1. Start BLE_SCAN",
    "2. Show BLE_SCAN Results",
    "3. Open CONNECTION",
    "4. Close CONNECTION"
};

char datMenuContent[DAT_MENU_ID_MAX][64] = {
    "1. Send BLE_DATA 'OK'",
};

static void BleMenuShowMenu(void);
```

Ilustración 137: Texto plano menú BLE.

Función *showScanResults:*

Se muestran por pantalla, mediante la directiva *am_menu_printf,* las direcciones identificadas tras el proceso de *Scanning* (Ilustración 138).

```
static void showScanResults(void)
{
    appDevInfo_t *devInfo;
    uint8_t num = AppScanGetNumResults();

    am_menu_printf("--------------------Scan Results--------------------\r\n");
    for (int i = 0; i < num; i++)
    {
        devInfo = AppScanGetResult(i);
        if (devInfo)
        {
            am_menu_printf("%d : %d %02x%02x%02x%02x%02x%02x \r\n", i, devInfo->addrType,
            devInfo->addr[0], devInfo->addr[1], devInfo->addr[2], devInfo->addr[3], devInfo->addr[4], devInfo->addr[5]);
        }
    }
    am_menu_printf("----------------------------------------------------\r\n");
}
```

Ilustración 138: showScanResults().

Función *isSelectionHome:*

Siempre que se reciba por la UART una letra *'h',* se representa el menú principal, para lo cual se asigna a las dos opciones la dirección y el Servicio (Ilustración 139).

```
static bool isSelectionHome(void)
{
    if (menuRxData[0] == 'h')
    {
        bleMenuCb.menuId = BLE_MENU_ID_MAIN;
        bleMenuCb.prevMenuId = BLE_MENU_ID_MAIN;
        bleMenuCb.gapMenuSelected = GAP_MENU_ID_NONE;
        bleMenuCb.datMenuSelected = DAT_MENU_ID_NONE;

        return true;
    }
    return false;
}
```

Ilustración 139: isSelectionHome().

Función *handleGAPSlection*:

Actúa como gestor principal del Servicio GAP dentro del menú, realizando la acción tras evaluar el identificador de la opción seleccionada, como se muestra en la Ilustración 140 y la Ilustración 141.

```
static void handleGAPSlection(void)
{
    //FBTG
    dmConnId_t  connIdList[DM_CONN_MAX];
    uint8_t     numConnections = AppConnOpenList(connIdList);

    eGapMenuId id;
    if (bleMenuCb.gapMenuSelected == GAP_MENU_ID_NONE)
    {
        id = (eGapMenuId)(menuRxData[0] - '0');
    }
    else
    {
        id = bleMenuCb.gapMenuSelected;
    }

    switch (id)
    {
        case GAP_MENU_ID_SCAN_START:
            am_menu_printf("scan start\r\n");
            DatcScanStart();
            break;
        case GAP_MENU_ID_SCAN_RESULTS:
            showScanResults();
            break;
        case GAP_MENU_ID_OPEN_CONNECT:
            if (bleMenuCb.gapMenuSelected == GAP_MENU_ID_NONE)
            {
                am_menu_printf("choose an idx from scan results to connect:\r\n");
                showScanResults();
                bleMenuCb.gapMenuSelected = GAP_MENU_ID_OPEN_CONNECT;
            }
            else
            {
                uint8_t idx = menuRxData[0] - '0';
                bleMenuCb.gapMenuSelected = GAP_MENU_ID_NONE;
                DatcConnOpen(idx);
            }
            break;
        case GAP_MENU_ID_CLOSE_CONNECT:
            if (bleMenuCb.gapMenuSelected == GAP_MENU_ID_NONE)
            {
                am_menu_printf("choose a conn_id from open connections (%d):\r\n", numConnections);
                bleMenuCb.gapMenuSelected = GAP_MENU_ID_CLOSE_CONNECT;
            }
            else
            {
                uint8_t cid = menuRxData[0] - '0';
//              am_menu_printf("close connection with conn_id = %d\r\n", cid);
                DatcConnClose(cid);
                bleMenuCb.gapMenuSelected = GAP_MENU_ID_NONE;
            }
            break;
        default:
            break;
    }
```

Ilustración 140: HandlerGapSelection().

```
"1. Start BLE_SCAN",
"2. Show BLE_SCAN Results",
"3. Open CONNECTION",
"4. Close CONNECTION"
```

Ilustración 141: menú princpial.

Función *handleDATselection:*

Evalúa en el menú DAT, qué opción ha sido seleccionada, y a partir de ella, realiza una acción concreta. En este caso, solo existe una función que consiste en enviar el mensaje “ok” equivalente a solicitar al dispositivo *Peripheral* información sobre la temperatura medida por el sensor. Tras realizar el envío, muestra por pantalla el identificador de la conexión (Ilustración 142).

```
static void handleDATselection(void)
{
    //FBTG
    dmConnId_t  connIdList[DM_CONN_MAX];
    uint8_t     numConnections = AppConnOpenList(connIdList);

    eDatMenuId id;
    if (bleMenuCb.datMenuSelected == DAT_MENU_ID_NONE)
    {
        id = (eDatMenuId)(menuRxData[0] - '0');
    }
    else
    {
        id = bleMenuCb.datMenuSelected;
    }

    switch (id)
    {
        case DAT_MENU_ID_SEND_OK:
            if (bleMenuCb.datMenuSelected == DAT_MENU_ID_NONE)
            {
                am_menu_printf("choose a conn_id from open connections (%d):\r\n", numConnections);
                bleMenuCb.datMenuSelected = DAT_MENU_ID_SEND_OK;
            }
            else
            {
                uint8_t cid = menuRxData[0] - '0';
                am_menu_printf("send 'ok' data to server with conn_id = %d\r\n", cid);
                DatcSendOKData(cid);
                bleMenuCb.datMenuSelected = DAT_MENU_ID_NONE;
            }
            break;
        default:
            break;
    }
}
```

Ilustración 142: HandlerDATselection().

Función *handleSelection:*

El gestor de selección evalúa la dirección, y muestra el menú correspondiente, dependiendo de la acción que esté realizándose (Ilustración 143).

```
static void handleSelection(void)
{
    if (isSelectionHome())
    {
        BleMenuShowMenu();
        return;
    }

    switch (bleMenuCb.menuId)
    {
        case BLE_MENU_ID_MAIN:
            bleMenuCb.menuId = (eBleMenuId)(menuRxData[0] - '0');
            BleMenuShowMenu();
            break;
        case BLE_MENU_ID_GAP:
            handleGAPSlection();
            break;
        case BLE_MENU_ID_DAT:
            handleDATselection();
            break;
        default:
            am_util_debug_printf("handleSelection() unknown input\n");
            break;
    }
}
```

Ilustración 143: handleSelection().

Función *BleMenuRx:*

Muestra por el terminal serie la información recibida desde el dispositivo *Peripheral,* y limpia el buffer del mensaje ya representado.

```
)id
.eMenuRx(void)

   if (menuRxDataLen == 0)
   {

      return;
   }

   menuRxData[menuRxDataLen] = '\0';
   am_util_debug_printf("BleMenuRx data = %s\n", menuRxData);

   handleSelection();
   // clear UART rx buffer
   memset(menuRxData, 0, sizeof(menuRxData));
   menuRxDataLen = 0;
```

Ilustración 144: BleMenuRx().

Función *BLEMenuShowMainMenu:*

Función que representa el menú principal, mediante un bucle que recorre carácter a carácter el menú principal. Los diferentes textos a mostrar se describen en la Ilustración 145 e Ilustración 146.

```
static void
BLEMenuShowMainMenu(void)
{
    am_menu_printf("--------------------BLE main menu--------------------\r\n");
    for (int i = 0; i < BLE_MENU_ID_MAX; i++)
    {
        am_menu_printf("%s\r\n", mainMenuContent[i]);
        uart_transmit_delay(AM_BSP_UART_PRINT_INST);
    }
    am_menu_printf("hint: use 'h' to do main menu\r\n");
    am_menu_printf("-----------------------------------------------------\r\n");
}
```

Ilustración 145: BLEmenuShowMainMenu().

```
static void
BLEMenuShowGAPMenu(void)
{
    for (int i = 0; i < GAP_MENU_ID_MAX; i++)
    {
        am_menu_printf("%s\r\n", gapMenuContent[i]);
        uart_transmit_delay(AM_BSP_UART_PRINT_INST);
    }
}


static void
BLEMenuShowDATMenu(void)
{
    for (int i = 0; i < DAT_MENU_ID_MAX; i++)
    {
        am_menu_printf("%s\r\n", datMenuContent[i]);
        uart_transmit_delay(AM_BSP_UART_PRINT_INST);
    }
}
```

Ilustración 146: BleMenuShowGAPmenu() y BleMenuShowDATmenu() .

Función *BleMenuShowMenu:*

Llama a alguna de las funciones anteriores cuando determina el menú que debe representarse (Ilustración 147).

```
static void
BleMenuShowMenu(void)
{
    switch (bleMenuCb.menuId)
    {
        case BLE_MENU_ID_MAIN:
            BLEMenuShowMainMenu();
            break;
        case BLE_MENU_ID_GAP:
            BLEMenuShowGAPMenu();
            break;
        case BLE_MENU_ID_DAT:
            BLEMenuShowDATMenu();
            break;
        default:
            break;
    }
}
```

Ilustración 147: BleMenuShowMenu().

4.2.3 Fichero Ble_Freertos.c

En lo que respecta al código de este fichero, se analizan únicamente las diferencias con respecto al equivalente en el dispositivo *Peripheral*.

Al tratarse del dispositivo *Central*, éste debe mostrar por pantalla todo el proceso de conexión y la información que le llega desde la UART, inicializando las interfaces usadas.

```
void
enable_print_interface(void)
{
#if 0
    //
    // Initialize the printf interface for UART output.
    //
    am_bsp_uart_printf_enable();
#else
    //
    // Initialize a debug printing interface.
    //
    am_bsp_itm_printf_enable();
#endif
    //
    // Print the banner.
    //
    am_util_stdio_terminal_clear();
}
```

Ilustración 148: Inicialización de interfaces para mostrar, Central.

Configuración de la UART

Esta función (Ilustración 149) habilita el uso de la UART, tanto de salida como de entrada, tratándola como interrupciones para interactuar con ella. Además, configura los parámetros, siendo en este caso la configuración UART por defecto 115200 – 8 – N – 1.

```
void
setup_serial(int32_t i32Module)
{
    //
    // Enable a UART to use for the menu.
    //
    const am_hal_uart_config_t sUartConfig =
    {
        //
        // Standard UART settings: 115200-8-N-1
        //
        .ui32BaudRate = 115200,
        .ui32DataBits = AM_HAL_UART_DATA_BITS_8,
        .ui32Parity = AM_HAL_UART_PARITY_NONE,
        .ui32StopBits = AM_HAL_UART_ONE_STOP_BIT,
        .ui32FlowControl = AM_HAL_UART_FLOW_CTRL_NONE,

        //
        // Set TX and RX FIFOs to interrupt at half-full.
        //
        .ui32FifoLevels = (AM_HAL_UART_TX_FIFO_1_2 |
                           AM_HAL_UART_RX_FIFO_1_2),

        //
        // Buffers
        //
        .pui8TxBuffer = 0,
        .ui32TxBufferSize = 0,
        .pui8RxBuffer = 0,
        .ui32RxBufferSize = 0,
    };

    am_hal_uart_initialize(0, &UART);
    am_hal_uart_power_control(UART, AM_HAL_SYSCTRL_WAKE, false);
    am_hal_uart_configure(UART, &sUartConfig);

    //
    // Make sure the UART interrupt priority is set low enough to allow
    // FreeRTOS API calls.
    //
    NVIC_SetPriority(UART0_IRQn, NVIC_configMAX_SYSCALL_INTERRUPT_PRIORITY);


    am_hal_gpio_pinconfig(AM_BSP_GPIO_COM_UART_TX, g_AM_BSP_GPIO_COM_UART_TX);
    am_hal_gpio_pinconfig(AM_BSP_GPIO_COM_UART_RX, g_AM_BSP_GPIO_COM_UART_RX);

    //
    // Enable UART RX interrupts manually.
    //
    am_hal_uart_interrupt_clear(UART, AM_HAL_UART_INT_RX | AM_HAL_UART_INT_RX_TMOUT);
    am_hal_uart_interrupt_enable(UART, AM_HAL_UART_INT_RX | AM_HAL_UART_INT_RX_TMOUT);
    NVIC_EnableIRQ(UART0_IRQn);
}
```

Ilustración 149: setup_serial().

En esta función se llevan a cabo las conversiones necesarias para escribir por la *UART,* y se crea el *buffer* de envío como una estructura en la que cada parámetro está configurado para su transmisión. Esta función es mostrada en la Ilustración 150.

```
// buffer for printf
static char g_prfbuf[AM_PRINTF_BUFSIZE];

uint32_t
am_menu_printf(const char *pcFmt, ...)
{
    uint32_t ui32NumChars;

    //
    // Convert to the desired string.
    //
    va_list pArgs;
    va_start(pArgs, pcFmt);
    ui32NumChars = am_util_stdio_vsprintf(g_prfbuf, pcFmt, pArgs);
    va_end(pArgs);

    //
    // This is where we print the buffer to the configured interface.
    //
    am_hal_uart_transfer_t sSend =
    {
        .ui32Direction = AM_HAL_UART_WRITE,
        .pui8Data = (uint8_t *)g_prfbuf,
        .ui32NumBytes = ui32NumChars,
        .ui32TimeoutMs = AM_HAL_UART_WAIT_FOREVER,
        .pui32BytesTransferred = 0,
    };

    am_hal_uart_transfer(UART, &sSend);

    //
    // return the number of characters printed.
    //
    return ui32NumChars;
}
```

Ilustración 150: am_menu_printf().

Con respecto a la función *main* (Ilustración 151) del dispositivo *Central,* el único cambio que se puede

observar es la configuración de la *UART,* mediante la llamada a la función *setup_serial ()* ya mencionada.

```
//FBTG
    setup_serial(0);

    //
    // Run the application.
    //
    run_tasks();

    //
    // We shouldn't ever get here.
    //
    while (1)
```

Ilustración 151: llamada a setup_serial().

4.3 Funcionamiento de los dispositivos *Peripheral* y *Central*

En esta sección se muestran los resultados experimentales obtenidos a partir de de la implementación de ambos dispositivos, uno de los hitos del proyecto. Para ello, se usan dos placas *Artemis Thing Plus* (una actuando como dispositivo *Central* y otra como dispositivo *Peripheral)*, además de utilizar programa *Putty* para visualizar la información que llega desde la UART y, como se describió anteriormente, a través de la que se configura el menú que interactúa con la conexión.

En primer lugar, se ejecutan los diferentes *makefiles* correspondientes a, *dats y datc.* Se configuran para visualizar la línea de ejecución, tanto con la aplicación *Putty (UART)* como con el *SWO (viewer)* del dispositivo *Central*. Al conectar el dispositivo *Peripheral* comienza el proceso de *Advertising*. Desde el dispositivo *Central* se presenta el menú representando en la ilustración 152, en el que se muestran las dos opciones disponibles.

Ambas opciones, como se presentó anteriormente, acceden al Servicio GAP y al *Dat* respectivamente. Para poder enlazar un dispositivo desde un dispositivo *Central,* se comienza a realizar el proceso de *Scanning* desde el dispositivo *Central* (Ilustración 153), hasta que se cumple el intervalo establecido, y lo interrumpe.

```
--------------------BLE main menu--------------------
1. BLE_SCAN+CONN
2. BLE_SEND

hint: use 'h' to do main menu
-----------------------------------------------------
```

Ilustración 152: Menú principal desde Putty (UART)

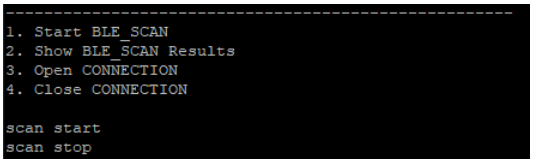


Ilustración 153: Proceso de Scanning desde el menú del dispositivo Central

En Ilustración 154 se observa a través de *SWO* el hilo de ejecución del dispositivo *Central* tras introducir el valor 1, correspondiente al inicio del proceso de *Scan*.

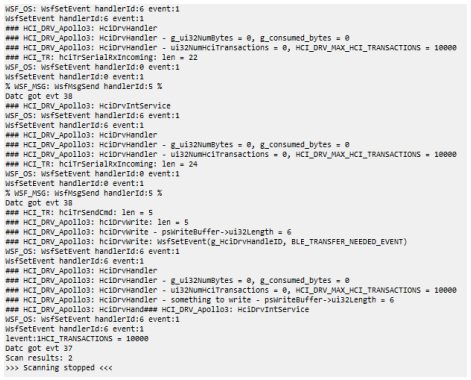


Ilustración 154: Captura de línea de ejecución del dispositivo Central tras Scanning

Para poder realizar la conexión, aún queda indicarle el dispositivo *Peripheral,* desde el dispositivo *Central* que se desea establecer una conexión. Para ello se abre la conexión y se elige el dispositivo anteriormente escaneado, tal y como se muestra en la Ilustración 155.

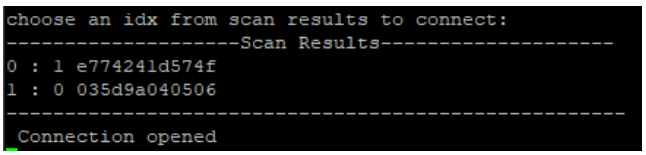


Ilustración 155: Abrir conexión desde el menú del dispositivo Central

Una vez conectados ambos dispositivos, se accede al menú *dat* para enviar un mensaje al *Peripheral* (Ilustración 156) con el fin de que éste envíe, en forma de notificación, una medida de la temperatura proporcionada por el sensor (Ilustración 157).

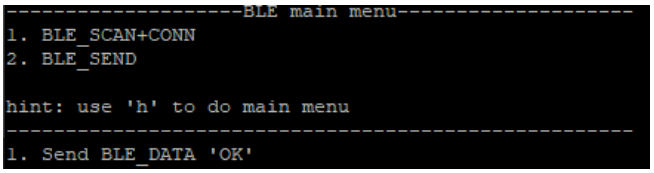


Ilustración 156: Envío BLE_DATA 'ok' desde el dispositivo Central

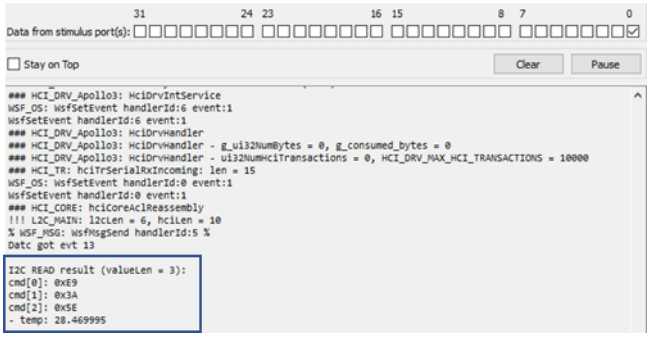


Ilustración 157: Lectura de la temperatura desde el dispositivo Central

La recepción del valor de la temperatura de forma exitosa confirma que la implementación de ambos dispositivos es completamente satisfactoria. Por lo tanto, ya implementada esta conexión se procede a incorporar a este desarrollo inicial, una serie de mejoras con opciones que ofrece BLE 5, así como el estudio de las prestaciones que proporciona esta conexión.

Capítulo 5: Implementación de la transferencia de datos WDX

En el presente TFG se ha implementado con éxito hasta ahora la funcionalidad básica de los dispositivos *Peripheral* y *Central BLE*, presentándose en este capítulo la implementación de la transferencia de *Stream* de datos a partir de la integración del perfil WDX [23]. El envío de datos en *Streaming* permitirá determinar el valor del *Throughput* que se puede obtener a partir de los dispositivos *Central/Peripheral* implementados en la plataforma *Artemis Thing Plus*, de *Sparkfun* a partir de la pila BLE *Cordio*.

5.1 Perfil WDX

El Perfil WDX (*Wireless Data Exchange)* es un perfil que se utiliza para transmitir y recibir archivos de datos y flujos de datos a través de una conexión *BLE*. El perfil WDX tiene los siguientes subsistemas:

- *File transfer (FT).* El subsistema FT se utiliza para enviar y recibir archivos de datos desde y hacia un sistema de archivos integrado (*Embedded File System, EFS*).
- *Authentication (AU).* El subsistema AU se utiliza para garantizar un enlace con un *Servidor*.
- *Device Configuration (DC).* El subsistema DC se utiliza para el intercambio de los siguientes parámetros de configuración con un *Servidor*:
    - Parámetros de actualización de la conexión.
    - Parámetros de autenticación.
    - Parámetros de MTU.
    - Nivel de batería.
    - Información sobre el modelo del dispositivo.
    - Información del *firmware* del dispositivo.
    - Diagnóstico del dispositivo.
- *Data Streaming (Stream).* El subsistema *Stream* puede utilizarse para transferir flujos de datos utilizados en aplicaciones de audio, vídeo, salud y *fitness,* entre otras [23].

Para implementar esta funcionalidad con el fin de poder transferir un flujo de datos entre los dispositivos *Central/Peripheral* implementados, ha sido necesario modificar los ficheros presentados en el anterior capítulo. Así, para la implementación de ambos dispositivos, se seguirá un formato idéntico, explicando las diversas adaptaciones realizadas.

5.1.1 Perfil WDXS (dispositivo *Peripheral)*

Existen dos Servicios BLE asociados al perfil WDX con esta funcionalidad, *Wdxs (Servidor*) y *Wdxc* (*Cliente*), realizándose el estudio de los ficheros usados para implementar el *Stream* en la conexión, comenzando por el *Servidor*. Las funciones prototipadas en el fichero *wdxs_main.c* están destinadas al uso de RAM, y habilitar la Lectura/Escritura/Borrado, como se muestra en la Ilustración 158.

```
/**************************************************************************************************
  Local Function Prototypes
**************************************************************************************************/
static uint8_t WdxsRamErase(uint32_t address, uint32_t size);
static uint8_t WdxsRamRead(uint8_t *pBuf, uint32_t address, uint32_t size);
static uint8_t WdxsRamWrite(const uint8_t *pBuf, uint32_t address, uint32_t size);
```

Ilustración 158: Funciones prototipadas en el fichero Wdxs.

En la sentencia mostrada en la Ilustración 159, se da formato a la información de la lista de archivos para el caso del archivo cuyo manejador se proporciona como parámetros desde la aplicación.

```
static void wdxsFormatFileResource(uint8_t *pData, wsfEfsHandle_t handle)
{
        APP_TRACE_INFO1("WDXS: --- wdxsFormatFileResource (handle = %d)", handle);

  UINT16_TO_BSTREAM(pData, handle);
  UINT8_TO_BSTREAM(pData, WsfEfsGetFileType(handle));
  UINT8_TO_BSTREAM(pData, WsfEfsGetFilePermissions(handle) & WSF_EFS_REMOTE_PERMISSIONS_MASK);
  UINT32_TO_BSTREAM(pData, WsfEfsGetFileSize(handle));
  WstrnCpy((char *)pData, WsfEfsGetFileName(handle), WSF_EFS_NAME_LEN);
  WstrnCpy((char *)pData+WSF_EFS_NAME_LEN, WsfEfsGetFileVersion(handle), WSF_EFS_VERSION_LEN);
}
```

Ilustración 159: wdxsFormatFileResoursce().

Función *WdxsUpdateListing:*

Esta función crea la lista de archivos a partir de los que se generarán los datos de usuario de los paquetes BLE que se transmitirán entre los dispositivos *Central/Peripheral* con las opciones *WDXC_INCLUDED = TRUE y WDXS_INCLUDED = TRUE*, según corresponda en cada caso, como se observa en el extracto de código mostrado en la Ilustración 160.

```
void WdxsUpdateListing(void)
{
  uint8_t *pTmp;
  uint8_t header[WDX_FLIST_HDR_SIZE];
  uint8_t record[WDX_FLIST_RECORD_SIZE];
  uint32_t position = 0, totalSize = 0;
  uint32_t fileCount = 0;
  uint8_t i;

    APP_TRACE_INFO0("WDXS: --- WdxsUpdateListing");

  position = WDX_FLIST_HDR_SIZE;

  for (i=0; i<WSF_EFS_MAX_FILES; i++)
  {
    if (WsfEfsGetFileByHandle(i) && (WsfEfsGetFilePermissions(i) & WSF_EFS_REMOTE_VISIBLE))
    {
      APP_TRACE_INFO1("WDXS: --- WdxsUpdateListing (i = %d)", i);

      /* Update the total size and file count */
      totalSize += WsfEfsGetFileSize(i);
      fileCount++;

      APP_TRACE_INFO3("WDXS: --- WdxsUpdateListing (i = %d) - totalSize = %d - fileCount = %d", i, totalSize, fileCount);

      wdxsFormatFileResource(record, i);

      /* Write the record */
          APP_TRACE_INFO0("WDXS: --- WdxsUpdateListing - WRITE THE RECORD");

      WsfEfsPut(WDX_FLIST_HANDLE, position, record, WDX_FLIST_RECORD_SIZE);

      position += WDX_FLIST_RECORD_SIZE;
    }
    else
    {
      APP_TRACE_INFO1("WDXS: --- WdxsUpdateListing NO UPDATE", i);

    }
  }

  /* Add the header after calculating the total_size and file_count */
  pTmp = header;
  UINT8_TO_BSTREAM(pTmp, WDX_FLIST_FORMAT_VER);
  UINT16_TO_BSTREAM(pTmp, fileCount);
  UINT32_TO_BSTREAM(pTmp, totalSize);

      APP_TRACE_INFO1("WDXS: --- WdxsUpdateListing - fileCount = %d", fileCount);
      APP_TRACE_INFO1("WDXS: --- WdxsUpdateListing - totalSize = %d", totalSize);

  /* Write the header */
      APP_TRACE_INFO0("WDXS: --- WdxsUpdateListing - WRITE THE HEADER");

  WsfEfsPut(WDX_FLIST_HANDLE, 0, header, WDX_FLIST_HDR_SIZE);
}
```

Ilustración 160: WdxsUpdateListing().

Función *wdxsSetCccIdx:*

En esta función se establece el CCCD utilizado por la aplicación para las características del Servicio WDXS (Ilustración 161).

```
void WdxsSetCccIdx(uint8_t dcCccIdx, uint8_t auCccId
{
     APP_TRACE_INFO0("WDXS: --- WdxsSetCccIdx");

  wdxsCb.dcCccIdx = dcCccIdx;
  wdxsCb.auCccIdx = auCccIdx;
  wdxsCb.ftcCccIdx = ftcCccIdx;
  wdxsCb.ftdCccIdx = ftdCccIdx;
}
```

Ilustración 161: WdxsSetCccIdx() .

Función *WdxsProcDmMsg:*

Esta función se invoca desde la aplicación para notificar algún evento *DM* al Servicio WDXS. Así, procesa los mensajes del gestor de eventos, llamando la pila a esta función para hacer uso de los diferentes

subsistemas. En la Ilustración 162 se pueden evaluar los casos que requieren un procedimiento distinto al usado hasta ahora.

```
void WdxsProcDmMsg(dmEvt_t *pEvt)
{
//  APP_TRACE_INFO0("WDXS: --- WdxsProcDmMsg");


  switch (pEvt->hdr.event)
  {
    case DM_CONN_CLOSE_IND:
  APP_TRACE_INFO0("WDXS: --- WdxsProcDmMsg - DM_CONN_CLOSE_IND");

      if (wdxsDcCb.doReset)
      {
        WdxsResetSystem();
      }
      break;

    case DM_CONN_OPEN_IND:
  APP_TRACE_INFO0("--- WdxsProcDmMsg - DM_CONN_OPEN_IND");
      /* Initialize connection parameters */
      wdxsCb.txReadyMask = WDXS_TX_MASK_READY_BIT;
      wdxsCb.ftInProgress = WDX_FTC_OP_NONE;
      wdxsCb.ftLen = 0;
      wdxsCb.ftOffset = 0;
#if WDXS_AU_ENABLED == TRUE
      wdxsAuCb.authLevel = WDX_AU_LVL_NONE;
      wdxsAuCb.authState = WDXS_AU_STATE_UNAUTHORIZED;
#endif /* WDXS_AU_ENABLED */
  APP_TRACE_INFO1("--- WdxsProcDmMsg - pEvt->connOpen.connInterval = %d", pEvt->connOpen.connInterval);
  APP_TRACE_INFO1("--- WdxsProcDmMsg - pEvt->connOpen.connLatency = %d", pEvt->connOpen.connLatency);
  APP_TRACE_INFO1("--- WdxsProcDmMsg - pEvt->connOpen.supTimeout = %d", pEvt->connOpen.supTimeout);

      wdxsCb.connInterval = pEvt->connOpen.connInterval;
      wdxsCb.connLatency = pEvt->connOpen.connLatency;
      wdxsCb.supTimeout = pEvt->connOpen.supTimeout;
      wdxsCb.txPhy= HCI_PHY_LE_1M_BIT;
      wdxsCb.rxPhy = HCI_PHY_LE_1M_BIT;
      break;


    case DM_CONN_UPDATE_IND:
   APP_TRACE_INFO0("WDXS: --- WdxsProcDmMsg - DM_CONN_UPDATE_IND");

      if (pEvt->hdr.status == HCI_SUCCESS)
      {
        wdxsCb.connInterval = pEvt->connUpdate.connInterval;
        wdxsCb.connLatency = pEvt->connUpdate.connLatency;
        wdxsCb.supTimeout = pEvt->connUpdate.supTimeout;
      }
      wdxsDcUpdateConnParam((dmConnId_t) pEvt->hdr.param, pEvt->hdr.status);
      break;

    case DM_PHY_UPDATE_IND:
   APP_TRACE_INFO0("WDXS: --- WdxsProcDmMsg - DM_PHY_UPDATE_IND");
      if (pEvt->hdr.status == HCI_SUCCESS)
      {
        wdxsCb.txPhy = pEvt->phyUpdate.txPhy;
        wdxsCb.rxPhy = pEvt->phyUpdate.rxPhy;
      }
      wdxsDcUpdatePhy((dmConnId_t) pEvt->hdr.param, pEvt->hdr.status);
      break;

    default:
  //  APP_TRACE_INFO0("WDXS: --- WdxsProcDmMsg - default");
      break;
  }
```

Ilustración 162: WdxsPocMsg().

Los casos adicionales que deben ser procesados por esta función son los siguientes:

1) *DM_CONN_CLOSE_IND*, al cerrar conexión se inicializa el sistema WDX a través de la llamada a la función *WdxsResetSystem().*
2) *DM_CONN_OPEN_IND*, cuando se abre conexión, se establecen los parámetros de conexión en las variables *wdxsCb*, además del subsistema de autentificación, en su caso.
3) *DM_CONN_UPDATE_IND,* cuando se actualizan los parámetros de conexión.
4) *DM_PHY_UPDATE_IND*, cuando es necesario cambiar el modo *PHY*.

Como se puede observar, este fichero sigue una estructura similar a la del fichero *dats_main.c,* radicando las principales diferencias en el uso de variables propias y la forma de establecer la conexión.

5.1.2 Fichero *wdxs_Stream.c* (dispositivo *Peripheral)*

En el fichero *wdxs_Stream.c* se implementan las funciones que permiten la generación de un *Stream* WDX (WDXS). Los flujos de datos WDXS se implementan como medios físicos virtuales en el sistema de archivos integrados (EFS). Así, un flujo de datos se crea a partir de los siguientes pasos:

1) Se crea la estructura *FileMedia_t* (*EFS Media Control Structure*) para el flujo que contiene la función de lectura creada en el paso 2.

2) Se implementa una función de lectura para el *Stream*. WDXS y EFS llamarán a la función de lectura para obtener los datos del flujo.

3) Se registra el contenido en el EFS, y se añade un archivo al sistema de archivos integrado que usa la comunicación creada en el paso 2.

El subsistema *Stream* del perfil WDX es el más importante para el presente TFG, ya que es la funcionalidad que se desea implementar en los dispositivos *Central/Peripheral*, motivo por el que se desglosan todas las funcionalidades que ofrece como *Servidor* WDX.

Función *wdxsSineRead:*

```
static uint8_t wdxsSineRead(uint8_t *pBuf, uint32_t address, uint32_t len)
{
  static int8_t incr = 1;
  static uint8_t dataVal = 0;

        APP_TRACE_INFO2("WDXS_STREAM: --- wdxsSineRead - len = %d, dataval = %d", len, dataVal);

  /* Build data in sine waveform */
  memset(pBuf, dataVal, len);
      APP_TRACE_INFO2("WDXS_STREAM: --- wdxsSineRead - pBuf (first byte) = %x (%d)", pBuf[0], pBuf[0]);

  if (dataVal <= 127)
  {
    incr++;
  }
  else
  {
    incr--;
  }

  dataVal += (incr / 2);

  return TRUE;
}
```

Ilustración 163: wdxsSineRead().

En esta función de lectura se determina el valor a partir del cual se generará el *payload* de cada uno de los paquetes BLE enviados en el flujo de datos que se transmitirá entre los dispositivos *Central* y *Peripheral*. Se trata de una función de lectura, que como se verá a lo largo de este capítulo, sigue una forma

de onda sinusoidal, a partir la cual se incrementa el valor hasta 127, e irá decreciendo hasta un valor mínimo de 0, y así sucesivamente. Existen otras dos formas de onda que no se usarán a lo largo del presente TFG, que son la forma de onda en escalón y en diente de sierra. En la Ilustración 164, se configura la forma de onda sinusoidal.

```
static uint8_t wdxsStreamRead(uint8_t *pBuf, uint32_t address, uint32_t len)
{
        APP_TRACE_INFO0("WDXS_STREAM: --- wdxsStreamRead");

  switch(wdxsStreamWaveform)
  {
  case WDXS_STREAM_WAVEFORM_SINE:
    wdxsSineRead(pBuf, address, len);
    break;

  case WDXS_STREAM_WAVEFORM_STEP:
    wdxsStepRead(pBuf, address, len);
    break;

  case WDXS_STREAM_WAVEFORM_SAWTOOTH:
    wdxsSawtoothRead(pBuf, address, len);
    break;
  }

  return TRUE;
}
```

Ilustración 164: wdxsStreamRead().

La función *wdxsStreamInit* configura en el fichero *dats_main.c* el tipo de forma de onda a usar para generar datos, si bien se datallará posteriormente. Llama a la función *wdxsStreamRead()* y realiza la lectura del valor correspondiente.

En la función mostrada en la Ilustración 165 se crea un *Stream* WDXS, inicializando todos los procesos necesarios para realizar el *Stream* de datos, registrando el tipo de *Stream* por defecto, estableciendo el valor de los Atributos del *Stream,* y añadiendo el fichero asociado al *Stream*.

```
void wdxsStreamInit(void)
{
  wsfEsfAttributes_t attr;

     APP_TRACE_INFO0("WDXS_STREAM: --- wdxsStreamInit");


  /* Register the media for the stream */
  WsfEfsRegisterMedia(&WDXS_StreamMedia, WDX_STREAM_MEDIA);

  /* Set the attributes for the stream */
  attr.permissions = WSF_EFS_REMOTE_VISIBLE | WSF_EFS_REMOTE_GET_PERMITTED;
  attr.type = WSF_EFS_FILE_TYPE_STREAM;

  /* Potential buffer overrun is intentional to zero out fixed length field */
  /* coverity[overrun-buffer-arg] */
  WstrnCpy(attr.name, "Stream", WSF_EFS_NAME_LEN);
  /* coverity[overrun-buffer-arg] */
  WstrnCpy(attr.version, "1.0", WSF_EFS_VERSION_LEN);

  /* Add a file for the stream */
  WsfEfsAddFile(0, WDX_STREAM_MEDIA, &attr, 0);
```

Ilustración 165: wdxsStreamInit().

5.1.3 Perfil WDXC (dispositivo *Central)*

En el fichero *wdxc_main.c* se incluye el registro de los cuatros subsistemas propios del perfil WDX, así como las funciones que implementan la funcionalidad de un *Cliente* WDX. Como se puede observar en

la Ilustración 166, inicialmente se configuran las Características y los Descriptores CCCD asociados a cada uno de los diferentes subsistemas WDX, como se estipula en la pila de protocolos BLE. Los datos que se envían y reciben en *Stream* se denominan FTD (*File Transfer Data*).

```
static const uint8_t wdxcDcUuid[ATT_128_UUID_LEN] = {WDX_DC_UUID};     /* WDX Device Configuration Characteristic */
static const uint8_t wdxcFtcUuid[ATT_128_UUID_LEN] = {WDX_FTC_UUID};   /* WDX File Transfer Control Characteristic */
static const uint8_t wdxcFtdUuid[ATT_128_UUID_LEN] = {WDX_FTD_UUID};   /* WDX File Transfer Data Characteristic */
static const uint8_t wdxcAuUuid[ATT_128_UUID_LEN] = {WDX_AU_UUID};     /* WDX Authentication Characteristic */

/*! WDXC Device Configuration */
static const attcDiscChar_t wdxcWdxsDc =
{
  wdxcDcUuid,
  ATTC_SET_UUID_128
};

/*! WDXC Device Configuration CCC descriptor */
static const attcDiscChar_t wdxcWdxsDcCcc =
{
  attCliChCfgUuid,
  ATTC_SET_DESCRIPTOR
};

/*! WDXC File Transfer Control */
static const attcDiscChar_t wdxcWdxsFtc =
{
  wdxcFtcUuid,
  ATTC_SET_UUID_128
};

/*! WDXC File Transfer Control CCC descriptor */
static const attcDiscChar_t wdxcWdxsFtcCcc =
{
  attCliChCfgUuid,
  ATTC_SET_DESCRIPTOR
};

/*! WDXC File Transfer Data */
static const attcDiscChar_t wdxcWdxsFtd =
{
  wdxcFtdUuid,
  ATTC_SET_UUID_128
};

/*! WDXC File Transfer Data CCC descriptor */
static const attcDiscChar_t wdxcWdxsFtdCcc =
{
  attCliChCfgUuid,
  ATTC_SET_DESCRIPTOR
};

/*! WDXC Authentication */
static const attcDiscChar_t wdxcWdxsAu =
{
  wdxcAuUuid,
  ATTC_SET_UUID_128
};


static const attcDiscChar_t wdxcWdxsAuCcc =
{
  attCliChCfgUuid,
  ATTC_SET_DESCRIPTOR
};

/*! List of characteristics to be discovered; order matches handle index enumeration  */
static const attcDiscChar_t *wdxcWdxsDiscCharList[] =
{
  &wdxcWdxsDc,                    /*! WDXC Device Configuration */
  &wdxcWdxsDcCcc,                 /*! WDXC Device Configuration CCC descriptor */
  &wdxcWdxsFtc,                   /*! WDXC File Transfer Control */
  &wdxcWdxsFtcCcc,                /*! WDXC File Transfer Control CCC descriptor */
  &wdxcWdxsFtd,                   /*! WDXC File Transfer Data */
  &wdxcWdxsFtdCcc,                /*! WDXC File Transfer Data CCC descriptor */
  &wdxcWdxsAu,                    /*! WDXC Authentication */
  &wdxcWdxsAuCcc,                 /*! WDXC Authentication CCC descriptor */
};

/*! sanity check:  make sure handle list length matches characteristic list length */
WSF_CT_ASSERT(WDXC_HDL_LIST_LEN == ((sizeof(wdxcWdxsDiscCharList) / sizeof(attcDiscChar_t *))));

const uint8_t wdxcSvcUuid[ATT_16_UUID_LEN] =    {UINT16_TO_BYTES(WDX_SVC_UUID)};
```

Ilustración 166: Variables wdxc.

Por una parte, en la función *wdxcWdxsDiscover*, se realiza el descubrimiento de Servicios y Características asociados al perfil WDX. El parámetro *pHdlList* debe apuntar a un vector de longitud *WDXC_HDL_LIST_LEN*. Si el descubrimiento se realiza con éxito, los manejadores de las Características y Descriptores descubiertos se establecerán en *pHdlList*, tal y como se muestra en Ilustración 167.

```
void WdxcWdxsDiscover(dmConnId_t connId, uint16_t *pHdlList)
{
    APP_TRACE_INFO0("WDXC: WdxcWdxsDiscover.");

  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];


  /* Store pointer to the attribute handles in the control block */
  pConnCb->pHdlList = pHdlList;

  /* Perform service discovery */
  AppDiscFindService(connId, ATT_16_UUID_LEN, (uint8_t *) wdxcSvcUuid,
                     WDXC_HDL_LIST_LEN, (attcDiscChar_t **) wdxcWdxsDiscCharList, pHdlList);
}
```

Ilustración 167 WdxcWdxsDiscover.

En Ilustración 168 se observa la función en la que se realiza el descubrimiento de los ficheros asociados a la transferencia de datos WDX, en su caso. Una vez completado el proceso de descubrimiento, se llamará a la función *ftcCallback* con el puntero a *WDX_FTC_OP* y el manejador de fichero *WDX_FLIST_HANDLE*.

```
void WdxcDiscoverFiles(dmConnId_t connId, wsfEfsFileInfo_t *pFileInfo, uint8_t maxFiles)
{
  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];

      APP_TRACE_INFO0("WDXC: WdxcDiscoverFiles.");


  /* Verify WDXC is Idle */
  if (pConnCb->fileHdl == WSF_EFS_INVALID_HANDLE)
  {
      APP_TRACE_INFO0("&&& pConnCb->fileHdl == WSF_EFS_INVALID_HANDLE");

    uint16_t len = maxFiles * WDX_FLIST_RECORD_SIZE + WDX_FLIST_HDR_SIZE;

    /* Update control information */
    pConnCb->pFileList = pFileInfo;
    pConnCb->fileCount = 0;
    pConnCb->maxFiles = maxFiles;
    pConnCb->fileHdl = WDX_FLIST_HANDLE;
    pConnCb->fDlPos = 0;

    /* Perform a get on file zero (the file list handle) where the length of the get is
     * the most file information pFileInfo can hold */
    WdxcFtcSendGetReq(connId, WDX_FLIST_HANDLE, 0, len, 0);
  }
}
```

Ilustración 168: WdxcDiscoverFiles.

En la función mostrada en la Ilustración 169, se procesa byte a byte el listado de ficheros asociados a la transferencia de un flujo de datos WDX (FTD), así como el manejador asociado, el tipo de transferencia, los permisos, o el tamaño de los ficheros.

```
static void wdxsParseFileList(dmConnId_t connId, uint8_t *pValue, uint16_t len)
{
        APP_TRACE_INFO0("WDXC: wdxsParseFileList.");

  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];
  uint16_t pos = 0;

  /* Depending on the MTU, blocks of FTD data from the file list may end mid-value.
   * Maintain a global position called wdxcCb.fDlPos, and process FTD data byte by byte */

  while (pos < len)
  {
    if (pConnCb->fDlPos < WDX_FLIST_HDR_SIZE)
    {
      /* Ignore file list header */
    }
    else
    {
      /* Find the file index and the position within the file index (the mark) */
      uint8_t mark = (pConnCb->fDlPos - WDX_FLIST_HDR_SIZE) % WDX_FLIST_RECORD_SIZE;
      uint8_t file = (pConnCb->fDlPos - WDX_FLIST_HDR_SIZE) / WDX_FLIST_RECORD_SIZE;
      wsfEfsFileInfo_t *pInfo;

      /* Ignore data if there is insufficient space in pConnCb->pFileList */
      if (file >= pConnCb->maxFiles)
      {
        return;
      }

      pInfo = &pConnCb->pFileList[file];

      /* Process a byte of data */
      switch (mark)
      {
      case 0: pConnCb->fileCount++; pInfo->handle = pValue[pos];  break;
      case 1: pInfo->handle |= ((uint16_t)pValue[pos]) << 8;    break;
      case 2: pInfo->attributes.type = pValue[pos];             break;
      case 3: pInfo->attributes.permissions = pValue[pos];      break;
      case 4: pInfo->size = pValue[pos];                        break;
      case 5: pInfo->size |= ((uint32_t)pValue[pos]) << 8;      break;
      case 6: pInfo->size |= ((uint32_t)pValue[pos]) << 16;     break;
      case 7: pInfo->size |= ((uint32_t)pValue[pos]) << 24;     break;
      default:
        if (mark > 7 && mark < 8 + WSF_EFS_NAME_LEN)
        {
          pInfo->attributes.name[mark - 8] = pValue[pos];
        }
        else
        {
          pInfo->attributes.version[mark - (8 + WSF_EFS_NAME_LEN)] = pValue[pos];
        }
        break;
      }
    }
```

Ilustración 169: wdxsParseFileL

Por su parte, en la función *wdxcParseFtc,* representada en la Ilustración 170, se procesan los mensajes asociados al control de transferencia de ficheros (FTC), mientras que en la función *wdxParseFtd*, cuyo código se muestra en la Ilustración 171, se procesan los mensajes asociados a la transferencia de datos (FTD).

```
static void wdxcParseFtc(dmConnId_t connId, uint8_t *pValue, uint16_t len)
{
    APP_TRACE_INFO0("WDXC: wdxcParseFtc.");

  /* Notification on a File Transfer Control (FTC) Attribute */
  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];
  uint8_t *p = pValue;
  uint8_t op, status;
  uint16_t handle;

  BSTREAM_TO_UINT8(op, p);
  BSTREAM_TO_UINT16(handle, p);

  if (op == WDX_FTC_OP_ABORT || op == WDX_FTC_OP_EOF)
  {
    pConnCb->fileHdl = WSF_EFS_INVALID_HANDLE;
    status = WDX_FTC_ST_SUCCESS;
  }
  else
  {
    BSTREAM_TO_UINT8(status, p)
  }

  /* Call appliation callback */
  if (wdxcCb.pFtdCallback)
  {
    (*wdxcCb.pFtcCallback)(connId, handle, op, status);
  }
}
```

Ilustración 170: wdxcParseFtc().

```
static void wdxcParseFtd(dmConnId_t connId, uint8_t *pValue, uint16_t len)
{
    APP_TRACE_INFO0("WDXC: wdxcParseFtd.");

  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];

  if (pConnCb->fileHdl == WDX_FLIST_HANDLE)
  {
    /* File Discovery is in progress */
    wdxsParseFileList(connId, pValue, len);
  }
  else if (pConnCb->fileHdl != WSF_EFS_INVALID_HANDLE)
  {
    /* Notify application of File Transfer Data (FTD) */
    if (wdxcCb.pFtdCallback)
    {
      (*wdxcCb.pFtdCallback)(connId, pConnCb->fileHdl, len, pValue);
    }
  }
}
```

Ilustración 171: wdxcParseFtd

En la función *wdxcWdxsValueUpdate*, cuyo códido se representa en la Ilustración 172, se procesa un valor recibido en un mensaje de respuesta de lectura en foma de Notificación o Indicación. El parámetro *pHdlList* debe apuntar a un vector de longitud *WDXC_WDX_HDL_LIST_LEN*. Si el manejador de Atributos del mensaje coincide con el manejador de la lista de asociados a cada uno de los subsistemas WDX, el valor debe procesarse, en caso contrario, se ignora.

```
static uint8_t wdxcWdxsValueUpdate(attEvt_t *pMsg)
{
    APP_TRACE_INFO0("WDXC: wdxcWdxsValueUpdate.");

  uint16_t *pHdlList = wdxcCb.conn[pMsg->hdr.param-1].pHdlList;
  uint8_t   status = ATT_SUCCESS;

  /* device configuration */
  if (pMsg->handle == pHdlList[WDXC_DC_HDL_IDX])
  {
    /* Not Supported */
  }
  /* file transfer control */
  else if (pMsg->handle == pHdlList[WDXC_FTC_HDL_IDX])
  {
    APP_TRACE_INFO0("WDXC file transfer control.");
    wdxcParseFtc((dmConnId_t) pMsg->hdr.param, pMsg->pValue, pMsg->valueLen);
  }
  /* file transfer data */
  else if (pMsg->handle == pHdlList[WDXC_FTD_HDL_IDX])
  {
    APP_TRACE_INFO0("WDXC file transfer data.");
    wdxcParseFtd((dmConnId_t) pMsg->hdr.param, pMsg->pValue, pMsg->valueLen);
  }
  /* authentication  */
  else if (pMsg->handle == pHdlList[WDXC_AU_HDL_IDX])
  {
    /* Not Supported */
  }
  else
  {
    status = ATT_ERR_NOT_FOUND;
  }

  return status;
}
```

Ilustración 172: wdxcWdxsValueUpdate.

Por su parte, la función *WdxcProcMsg* (Ilustración 173) se invoca desde la aplicación para notificar al *Cliente* WDX (WDXC) los eventos del sistema. Al igual que en otros ficheros, esta función realiza acciones dependiendo el mensaje *DM* a procesar.

```
void WdxcProcMsg(wsfMsgHdr_t *pEvt)
{
//                APP_TRACE_INFO0("WDXC: WdxcProcMsg.");

  switch (pEvt->event)
  {
    case ATTC_HANDLE_VALUE_NTF:
    {
      APP_TRACE_INFO0("WdxcProcMsg - ATTC_HANDLE_VALUE_NTF");

      wdxcWdxsValueUpdate((attEvt_t *)pEvt);
    }
      break;

    case DM_CONN_CLOSE_IND:
      {
        APP_TRACE_INFO0("WdxcProcMsg - DM_CONN_CLOSE_IND");
    wdxcConnCb_t *pConnCb = &wdxcCb.conn[pEvt->param - 1];

        /* Set file handle to invalid to indicate no file operation is in progress */
        pConnCb->fileHdl = WSF_EFS_INVALID_HANDLE;
      }
      break;

    default:
    {
//          APP_TRACE_INFO0("WdxcProcMsg - default");
    }
    break;
  }
}
```

Ilustración 173: WdxProcMsg Stream.

1) *APP_HANDLE_VALUE_NTF*, si se recibe un mensaje de este tipo se llama a la función *WdxcWdxsValueUpdate* para procesar el valor recibido.
2) *DM_CONN_CLOSE_IND*, si se cierra la conexión con el dispositivo, se finalizan las opreaciones de transferencia de datos en curso.

5.1.4 Fichero *Wdxc_Stream.c*

En este fichero, al corresponder a la implementación del dispositivo *Central,* se configura el inicio y el fin de la transferencia de un flujo de datos. La Ilustración 174 muestra el código correspondiente a la función *WdxcStreamStart*, en la que se envía una petición para iniciar la transferencia de un flujo de datos asociado al manejador de fichero proporcionado como parámetro, para una conexión determinada.

```
void WdxcStreamStart(dmConnId_t connId, uint16_t fileHdl)
{
  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];

        APP_TRACE_INFO1("WDXC_STREAM: --- WdxcStreamStart - connId = %d", connId);


  /* Verify WDXC is Idle */
  if (pConnCb->fileHdl == WSF_EFS_INVALID_HANDLE)
  {
    pConnCb->fileHdl = fileHdl;
        APP_TRACE_INFO0("WDXC_STREAM: --- WdxcStreamStart - WDXC is idle");

    /* Send a get request with len set the MTU length (offset is ignored for streams) */
    WdxcFtcSendGetReq(connId, fileHdl, 0, AttGetMtu(connId), 0);
  }
}
```

Ilustración 174: WdxcStramStart.

Por su parte, en la función *WdxcStreamStop* se finaliza la transferencia de flujo de datos activa mediante el envío de un mensaje de FTC tipo *WDX_FTC_OP_ABORT*.

```
void WdxcStreamStop(dmConnId_t connId)
{
  wdxcConnCb_t *pConnCb = &wdxcCb.conn[connId - 1];

        APP_TRACE_INFO0("WDXC_STREAM: --- WdxcStreamStop - WDXC is idle");

  /* Send an abort to stop the stream */
  WdxcFtcSendAbort(connId, pConnCb->fileHdl);

  /* Set file handle to invalid to indicate no file operation is in progress */
  pConnCb->fileHdl = WSF_EFS_INVALID_HANDLE;
}
```

Ilustración 175: WdxcStreamStop.

5.2 Implementación de la funcionalidad WDX en el dispositivo *Peripheral*

La aplicación *dats* implementa el papel de dispositivo *Slave* de una aplicación propietaria de WDX. En primer lugar, antes de realizar las modificaciones pertinentes sobre la implementación del dispositivo *Peripheral* desarrollado inicialmente, se añade en el fichero *makefile* el perfil situado en la ruta *AmbiqSuiteSDK-master\boards_sfe\common\examples\ble_freertos_datc_menu_ok_v01\gcc*, en la que se encuentra el perfil WDXC y el perfil WDXS, como se muestra en la Ilustración 176.

```
#FBTG
INCLUDES+= -I$(COMMONPATH)/examples/ble_freertos_datc_menu_ok_v01/src

INCLUDES+= -I$(SDKPATH)/CMSIS/ARM/Include
INCLUDES+= -I$(SDKPATH)/third_party/exactle/wsf/sources/port/freertos
INCLUDES+= -I$(SDKPATH)/third_party/FreeRTOSv10.1.1/Source/include
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-host/sources/hci/ambiq/apollo3
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-host/sources/stack/dm
INCLUDES+= -I$(SDKPATH)/third_party/exactle/wsf/sources/util
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-profiles/sources/profiles/wdxs
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-profiles/sources/profiles/wdxc
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-profiles/sources/profiles/gatt
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-profiles/include/app
INCLUDES+= -I$(SDKPATH)/third_party/uecc
INCLUDES+= -I$(BOARDPATH)/bsp
INCLUDES+= -I$(SDKPATH)/CMSIS/AmbiqMicro/Include
INCLUDES+= -I$(SDKPATH)/third_party/exactle/ble-profiles/sources/services
```

Ilustración 176: Makefile dats_wdxs

5.2.1 Fichero *dats_main_I2C.c*

En este fichero, localizado en la ruta *AmbiqSuiteSDKmaster* *boards_sfe*\ *common*\ *examples* *ble_freertos_dats_I2C_ok_v01**dats*, se implementa la funcionalidad del dispositivo *Peripheral,* incluyendo la transferencia de flujo de datos a partir los perfiles WDXC/WDXS proporcionados en la pila *Cordio* BLE. Así, en primer lugar se ha modificado el fichero inicial incluyendo las definiciones y los ficheros asociados al perfil WDXS, como se muestra en la Ilustración 177.

```
#include "dats/dats_api_i2c_v01.h"
#if WDXS_INCLUDED  == TRUE
#include "wsf_efs.h"
#include "svc_wdxs.h"
#include "wdxs/wdxs_api.h"
#include "wdxs/wdxs_main.h"
#include "wdxs/wdxs_stream.h"
#endif
```

Ilustración 177: Includes wdxs

Adicionalmente, se incluyen los descriptores CCCD asociados a cada uno de los subsistemas del Servicio WDX, asignando así un índice a cada uno de ellos, tal y como se muestra en la Ilustración 178.

```
enum
{
#if WDXS_INCLUDED == TRUE
  WDXS_DC_CH_CCC_IDX,             /*! WDXS DC service, service changed characteristic */
  WDXS_FTC_CH_CCC_IDX,            /*! WDXS FTC  service, service changed characteristic */
  WDXS_FTD_CH_CCC_IDX,            /*! WDXS FTD service, service changed characteristic */
  WDXS_AU_CH_CCC_IDX,             /*! WDXS AU service, service changed characteristic */
#endif /* WDXS_INCLUDED */
```

Ilustración 178: Subsistemas Stream Servidor.

A continuación, se configura cada uno de estos nuevos Servicios asociados al perfil WDX, siguiendo la misma estructura que la expuesta anteriormente, es decir, el manejador con el que se identifica, así como sus propiedades y nivel de seguridad (Ilustración 179).

```
/*! client characteristic configuration descriptors settings, indexed by above enumeration */
static const attsCccSet_t datsCccSet[DATS_NUM_CCC_IDX] =
{
  /* cccd handle          value range               security level */
#if WDXS_INCLUDED == TRUE
  {WDXS_DC_CH_CCC_HDL,    ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_DC_CH_CCC_IDX */
  {WDXS_FTC_CH_CCC_HDL,   ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_FTC_CH_CCC_IDX */
  {WDXS_FTD_CH_CCC_HDL,   ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_FTD_CH_CCC_IDX */
  {WDXS_AU_CH_CCC_HDL,    ATT_CLIENT_CFG_NOTIFY,    DM_SEC_LEVEL_NONE},   /* WDXS_AU_CH_CCC_IDX */
#endif /* WDXS_INCLUDED */
```

Ilustración 179: Estructura CCCD Stream Servidor.

Función *datsATTCback*:

En esta función se implementa la función *callback* que es llamada por la aplicación para enviar notificaciones al *Cliente* WDX (WDXS). Se describe su funcionamiento en la sección de código mostrada en la Ilustración 180.

```
static void datsAttCback(attEvt_t *pEvt)
{
#if WDXS_INCLUDED == TRUE
  bool_t wdxsAttCback_return;

  wdxsAttCback_return = WdxsAttCback(pEvt);
  APP_TRACE_INFO1("%%% wdxsAttCback_return = %d", wdxsAttCback_return);

#endif /* WDXS_INCLUDED */
}
```

Ilustración 180: datsATTCback Stream.

Función *datsProcMsg*:

La única diferencia con el código anteriormente desarrollado para implantar esta función, reside en que, a la hora de establecer una conexión, no sólo se notificará a la aplicación que se debe abrir (*APP_UI_CONN_OPEN*) sino que se establece el tipo de forma de onda a partir de la cuál se generará el flujo de datos, siendo en este caso sinusoidal. En la función *wdxsSetStreamWaveform()*, que como se indicó anteriormente es propia del perfil WDX, se especifica la forma de onda a seguir a la hora de la generar los datos que se transferirán, en este caso, de acuerdo una forma de onda sinusoidal. Esta función se representa en la Ilustración 181.

```
static void datsProcMsg(dmEvt_t *pMsg)
{
  uint8_t uiEvent = APP_UI_NONE;

  switch(pMsg->hdr.event)
  {
    case ATT_MTU_UPDATE_IND:
      APP_TRACE_INFO1("Negotiated MTU %d", ((attEvt_t *)pMsg)->mtu);
      break;

    case DM_RESET_CMPL_IND:
      AttsCalculateDbHash();
      DmSecGenerateEccKeyReq();
      datsSetup(pMsg);
      uiEvent = APP_UI_RESET_CMPL;
      break;

    case DM_ADV_START_IND:
      uiEvent = APP_UI_ADV_START;
      break;

    case DM_ADV_STOP_IND:
      uiEvent = APP_UI_ADV_STOP;
      break;

    case DM_CONN_OPEN_IND:
      uiEvent = APP_UI_CONN_OPEN;
#if CS50_INCLUDED == TRUE
//FBTG
      datsCb.phyMode = DATS_PHY_1M;
#endif /* CS50_INCLUDED */

#if WDXS_INCLUDED == TRUE
        static uint8_t waveform = WDXS_STREAM_WAVEFORM_SINE;
        wdxsSetStreamWaveform(waveform);
#endif /* WDXS_INCLUDED */
```

Ilustración 181:datsProcMsg Stream.

Función *DatsHandler:*

Las modificaciones realizadas en la implementación de esta función se centran en incluir el procesamiento de los mensajes relativos al perfil WDX mediante la llamada a la función *WdxsProcDmMsg()* descrita anteriormente, como se muestra en la Ilustración 182.

```
void DatsHandler(wsfEventMask_t event, wsfMsgHdr_t *pMsg)
{
  if (pMsg != NULL)
  {
//    APP_TRACE_INFO0("--- DatsHandler");
    APP_TRACE_INFO1("--- DatsHandler - Dats got evt %d", pMsg->event);

    /* process ATT messages */
    if (pMsg->event >= ATT_CBACK_START && pMsg->event <= ATT_CBACK_END)
    {
      /* process server-related ATT messages */
      AppServerProcAttMsg(pMsg);
    }
    /* process DM messages */
    else if (pMsg->event >= DM_CBACK_START && pMsg->event <= DM_CBACK_END)
    {
      /* process advertising and connection-related messages */
      AppSlaveProcDmMsg((dmEvt_t *) pMsg);

      /* process security-related messages */
      AppSlaveSecProcDmMsg((dmEvt_t *) pMsg);

#if WDXS_INCLUDED == TRUE
      /* process WDXS-related messages */
      WdxsProcDmMsg((dmEvt_t*) pMsg);
#endif /* WDXS_INCLUDED */
    }

    /* perform profile and user interface-related operations */
    datsProcMsg((dmEvt_t *) pMsg);
  }
}
```

Ilustración 182: DatsHandle Stream.

Función *DatsStart*:

Como se describió en el punto anterior, dentro de las funcionalidades asociadas al perfil WDX, se pretende dar uso al *Stream*, por lo que se debe incluir la inicialización del subsistema *Stream* mediante la llamada a la función *WdxsStreamInity* y adicionalmente el *OTA file media*, que define los archivos gestionados por el subsistema *Stream.* Además, se inicializa el modo PHY que se acuerda para la transferencia de los ficheros WDX.

```
void DatsStart(void)
{
      APP_TRACE_INFO0("DatsStart");

  /* Register for stack callbacks */
  DmRegister(datsDmCback);
  DmConnRegister(DM_CLIENT_ID_APP, datsDmCback);
  AttRegister(datsAttCback);
  AttConnRegister(AppServerConnCback);
  AttsCccRegister(DATS_NUM_CCC_IDX, (attsCccSet_t *) datsCccSet, datsCccCback);

  /* Initialize attribute server database */
  SvcCoreGattCbackRegister(GattReadCback, GattWriteCback);
  SvcCoreAddGroup();
  SvcWpCbackRegister(NULL, datsWpWriteCback);
  SvcWpAddGroup();

  /* Set Service Changed CCCD index. */
  GattSetSvcChangedIdx(DATS_GATT_SC_CCC_IDX);

  /* Register for app framework button callbacks */
//  AppUiBtnRegister(datsBtnCback);


#if WDXS_INCLUDED == TRUE

  /* Initialize the OTA File Media */
  WdxsOtaMediaInit();

  /* Initialize the WDXS Stream */
  wdxsStreamInit();

  /* Set the WDXS CCC Identifiers */
  WdxsSetCccIdx(WDXS_DC_CH_CCC_IDX, WDXS_AU_CH_CCC_IDX, WDXS_FTC_CH_CCC_IDX, WDXS_FTD_CH_CCC_IDX);

#if CS50_INCLUDED == TRUE
  APP_TRACE_INFO0("--- WdxsPhyInit (CS50)");
  WdxsPhyInit();
#endif /* CS50_INCLUDED */
```

Ilustración 183: DatsStart Stream.

5.2.2 Fichero *Radio_task.c*

Función *DatsStart:*

A partir del fichero *radio_Task.c* inicialmente desarrollado en el presente TFG, se inicializan los dos Servicios que se usan del perfil WDX (Ilustración 184), OTA y *Stream*, analizados en el punto anterior.

```
void DatsStart(void)
{
      APP_TRACE_INFO0("DatsStart");

  /* Register for stack callbacks */
  DmRegister(datsDmCback);
  DmConnRegister(DM_CLIENT_ID_APP, datsDmCback);
  AttRegister(datsAttCback);
  AttConnRegister(AppServerConnCback);
  AttsCccRegister(DATS_NUM_CCC_IDX, (attsCccSet_t *) datsCccSet, datsCccCback);

  /* Initialize attribute server database */
  SvcCoreGattCbackRegister(GattReadCback, GattWriteCback);
  SvcCoreAddGroup();
  SvcWpCbackRegister(NULL, datsWpWriteCback);
  SvcWpAddGroup();

  /* Set Service Changed CCCD index. */
  GattSetSvcChangedIdx(DATS_GATT_SC_CCC_IDX);

  /* Register for app framework button callbacks */
//  AppUiBtnRegister(datsBtnCback);

#if WDXS_INCLUDED == TRUE

  /* Initialize the OTA File Media */
  WdxsOtaMediaInit();

  /* Initialize the WDXS Stream */
  wdxsStreamInit();

  /* Set the WDXS CCC Identifiers */
  WdxsSetCccIdx(WDXS_DC_CH_CCC_IDX, WDXS_AU_CH_CCC_IDX, WDXS_FTC_CH_CCC_IDX, WDXS_FTD_CH_CCC_IDX);

#if CS50_INCLUDED == TRUE
  APP_TRACE_INFO0("--- WdxsPhyInit (CS50)");
  WdxsPhyInit();
#endif /* CS50_INCLUDED */

#endif /* WDXS_INCLUDED */

#if CS50_INCLUDED == TRUE
  APP_TRACE_INFO0("--- DmPhyInit (CS50)");
  DmPhyInit();
#endif /* CS50_INCLUDED */

  WsfBufDiagRegister(datsWsfBufDiagnostics);

  /* Reset the device */
  DmDevReset();
```

Ilustración 184: DatsStart().

Función *exactle_stack_init:*

En la sección de código que se muestra en la Ilustración 185, se incluye en esta función, un *handle* adicional con el fin de registrar un manejador de tarea asociado a la funcionalidad del perfil WDX, en la sección de inicialización de la aplicación.

```
//FBTG
#if WDXS_INCLUDED  == TRUE
    handlerId = WsfOsSetNextHandler(WdxsHandler);
    am_util_debug_printf("WsfOsSetNextHandler(WdxsHandler) - handlerId = %d\r\n", handlerId);
    WdxsHandlerInit(handlerId);
#endif
```

Ilustración 185: Radio_Task configuración longitud y gestor para Stream.

Por último, se realizó una modificación en el fichero *Wsf_os.c* ubicado en la ruta *AmbiqSuiteSDK-master\third_party\exactle\wsf\sources\port\freertos.* Esta modificación está relacionada con el número de manejadores máximo de la aplicación, ya que por defecto soporta 9, y en el caso del presente TFG se

necesitan 10 para llevar a cabo adecuadamente la transferencia de flujos de datos por *Stream*. Este cambio se ve reflejado en la Ilustración 186.

```
/* maximum number of event handlers per task */
#ifndef WSF_MAX_HANDLERS
#define WSF_MAX_HANDLERS      10
#endif
```

Ilustración 186: Wsf_os.C modificación de WSF_MAX_HANDLERS.

5.3 Implementación de la funcionalidad WDX en el dispositivo *Central*

En este caso se hace uso de una función de conexión automática para escanear y conectarse automáticamente a una aplicación *Slave*. Una vez conectada, la aplicación puede enviar y recibir mensajes de datos simples. La aplicación *datc* es capaz de conectarse a múltiples dispositivos *Slave*.

Cuando la aplicación *datc* se compila con la opción *WDXC_INCLUDED=TRUE*, puede transmitir *Stream* datos a un dispositivo WDXS. Para comenzar el *Streaming*, se requiere en primer lugar descubrir los archivos en el dispositivo par, y posteriormente, iniciar/detener el *Streaming* de datos.

5.3.1 Fichero *Datc_main.c*

Este fichero se encuentra en la ruta *AmbiqSuiteSDK-master**boards_sfe**common**example*, e incluye la transferencia de flujo de datos a partir los perfiles WDXC/WDXS proporcionados en la pila *Cordio* BLE. Así, se ha modificado el fichero inicial del dispositivo *Central* incluyendo las definiciones y los ficheros asociados al perfil WDXC, como se muestra en la Ilustración 187.

```
#include "dats/dats_api_i2c_v01.h"
#if WDXS_INCLUDED  == TRUE
#include "wsf_efs.h"
#include "svc_wdxs.h"
#include "wdxs/wdxs_api.h"
#include "wdxs/wdxs_main.h"
#include "wdxs/wdxs_stream.h"
#endif
```

Ilustración 187: Includes wdxc.

En este sentido, en el apartado de macros se definen los parámetros del perfil WDX, que se muestran en la Ilustración 188.

```
#if WDXC_INCLUDED == TRUE
/* Size of WDXC file discovery dataset */
#define DATC_WDXC_MAX_FILES               4

/*! WSF message event starting value */
#define DATC_MSG_START                    0xA0

/*! Data rate timer period in seconds */
#define DATC_WDXS_DATA_RATE_TIMEOUT       4

/*! WSF message event enumeration */
enum
{
  DATC_WDXS_DATA_RATE_TIMER_IND = DATC_MSG_START, /*! Data rate timer expired */
};
#endif /* WDXC_INCLUDED */
```

Ilustración 188: Macro WDXC.

A la hora de configurar el *Cliente ATT* se considera WDX como un Servicio más a descubrir e inicializar, por lo que debe tenerse en cuenta. En la Ilustración 189 se muestran los cambios introducidos con respecto al *Cliente ATT* básico. Adicionalmente, se ha incluido el código necesario para añadir el Servicio WDX entre los que deben ser descubiertos desde el dispositivo *Central*, como se muestran en la Ilustración 190.

```
/**************************************************************************************************
  ATT Client Discovery Data
**************************************************************************************************/

/*! Discovery states:  enumeration of services to be discovered */
enum
{
  DATC_DISC_GATT_SVC,      /*! GATT service */
  DATC_DISC_GAP_SVC,       /*! GAP service */
  DATC_DISC_WP_SVC,        /*! Arm Ltd. proprietary service */
#if WDXC_INCLUDED == TRUE
  DATC_DISC_WDXC_SCV,      /*! Arm Ltd. Wireless Data Exchange service */
#endif /* WDXC_INCLUDED */
  DATC_DISC_SVC_MAX        /*! Discovery complete */
};
```

Ilustración 189: Configuración Cliente ATT Stream.

```
/*! Start of each service's handles in the the handle list */
#define DATC_DISC_GATT_START       0
#define DATC_DISC_GAP_START        (DATC_DISC_GATT_START + GATT_HDL_LIST_LEN)
#define DATC_DISC_WP_START         (DATC_DISC_GAP_START + GAP_HDL_LIST_LEN)
#if WDXC_INCLUDED == TRUE
#define DATC_DISC_WDXC_START       (DATC_DISC_WP_START + WPC_P1_HDL_LIST_LEN)
#define DATC_DISC_HDL_LIST_LEN     (DATC_DISC_WDXC_START + WDXC_HDL_LIST_LEN)
#else
#define DATC_DISC_HDL_LIST_LEN     (DATC_DISC_WP_START + WPC_P1_HDL_LIST_LEN)
#endif /* WDXC_INCLUDED */
```

Ilustración 190: Definición Servicio *descubrimiento para sistema WDX.*

Por otro lado, se incluye el puntero a la lista de manejadores asociados a los Servicios WDX, como se muestra en la Ilustración 191.

```
/*! Pointers into handle list for each service's handles */
static uint16_t *pDatcGattHdlList[DM_CONN_MAX];
static uint16_t *pDatcGapHdlList[DM_CONN_MAX];
static uint16_t *pDatcWpHdlList[DM_CONN_MAX];
#if WDXC_INCLUDED == TRUE
static uint16_t *pDatcWdxHdlList[DM_CONN_MAX];
#endif /* WDXC_INCLUDED */
```

Ilustración 191: Lista de punteros Cliente ATT Stream.

Además, en la configuración de datos se deben incluir, en la lista de Características a configurar tras finalizar el proceso de descubrimiento de Servicios, los descriptores CCCD asociados a los subsistemas del perfil WDX, como se muestra en la figura Ilustración 192.

```
#if WDXC_INCLUDED == TRUE
  /* Write:  WDXC ccc descriptors */
  {datcCccNtfVal, sizeof(datcCccNtfVal), (WDXC_DC_CCC_HDL_IDX + DATC_DISC_WDXC_START)},
  {datcCccNtfVal, sizeof(datcCccNtfVal), (WDXC_FTC_CCC_HDL_IDX + DATC_DISC_WDXC_START)},
  {datcCccNtfVal, sizeof(datcCccNtfVal), (WDXC_FTD_CCC_HDL_IDX + DATC_DISC_WDXC_START)},
  {datcCccNtfVal, sizeof(datcCccNtfVal), (WDXC_AU_CCC_HDL_IDX + DATC_DISC_WDXC_START)},
#endif /* WDXC_INCLUDED */
};
```

Ilustración 192: Servicios *CCCD Cliente ATT Stream.*

Por otro lado, en la función *dataOpen* se contempla el modo *PHY* que se configura para abrir la conexión, siendo aquí donde se modifica y especifica cúal se usa. Inicialmente se usa el modo *Classic* (los tipos de modalidad *PHY* se explicaron en el Capítulo 2 del presente TFG).

```
static void datcOpen(dmEvt_t *pMsg)
{
#if CS50_INCLUDED == TRUE
//FBTG
  datcCb.phyMode[pMsg->hdr.param-1] = DATC_PHY_1M;
//  datcCb.phyMode[pMsg->hdr.param-1] = DATC_PHY_2M;
#endif /* CS50_INCLUDED */
#if WDXC_INCLUDED == TRUE
      APP_TRACE_INFO0("*** datcOpen.");
  datcCb.streamHandle[pMsg->hdr.param-1] = WSF_EFS_INVALID_HANDLE;
#endif /* WDXC_INCLUDED */
}
```

Ilustración 193: DatcOpen Stream.

Con el objetivo de calcular el *Throughput,* en la transferencia de flujo de datos WDX se implementa una nueva función. Se trata de una función que calcula el *data rate* y determina mediante un *timer* el intervalo de tiempo para medir los resultados parciales. En la Ilustración 194, se observa la justificación de todos estos cálculos. El parámetro *DATC_WDXS_DATA_RATE_TIMEOUT* se establece con un valor de 4 segundos, por lo que, transcurrido este periodo de tiempo, se calculará el *data rate* parcial, en bps, iniciándose de nuevo el *timer*.

```
static void datcWdxcProcessDataRateTimeout(dmConnId_t connId)
{
  uint32_t bps;

        APP_TRACE_INFO0("*** datcWdxcProcessDataRateTimeout.");

  /* Suppress warning when APP_TRACE disabled */
  (void) bps;

  /* Calculate data rate */
  bps = (datcCb.rxCount[connId-1] - datcCb.lastCount[connId-1]) * 8 / DATC_WDXS_DATA_RATE_TIMEOUT;
  datcCb.lastCount[connId-1] = datcCb.rxCount[connId-1];

  datcCb.streamSeconds[connId-1] += DATC_WDXS_DATA_RATE_TIMEOUT;
  datcCb.streamRate[connId-1] = datcCb.rxCount[connId-1] * 8 / datcCb.streamSeconds[connId-1];

  APP_TRACE_INFO2("Streaming data rate %d bps, overall %d bps", bps, datcCb.streamRate[connId-1]);
  am_menu_printf(" -- Streaming data rate %d bps, overall %d bps\r\n", bps, datcCb.streamRate[connId-1]);


  /* Restart the timer */
  datcCb.dataRateTimer[connId-1].msg.event = DATC_WDXS_DATA_RATE_TIMER_IND;
  datcCb.dataRateTimer[connId-1].msg.param = connId;
  datcCb.dataRateTimer[connId-1].handlerId = datcCb.handlerId;

  WsfTimerStartSec(&datcCb.dataRateTimer[connId-1], DATC_WDXS_DATA_RATE_TIMEOUT);
}
```

Ilustración 194: DatcWdxProcessDataRateTimeout.

La función mostrada en la Ilustración 195 realiza la suscripción al envío de notificaciones, equivalente al envío de cualquier dato desde el dispositivo *Peripheral.* Solo se ejecutará en caso de que se configure la opción *WDXC_INCLUDED = FALSE* en el *Makefile.*

```
#if WDXC_INCLUDED == FALSE
void DatcSendOKData(dmConnId_t connId)
{
    datcSendData(connId);
}
#endif
```

Ilustración 195: DatcSendOKData.

El descubrimiento de los ficheros del EFS en el dispositivo *Peripheral* se realiza desde el sistema WDX, invocando la función *WdxcDiscoverFile ()* para realizar el descubrimiento de los Servicios y Características, como se puede observar en Ilustración 196.

```
#if WDXC_INCLUDED == TRUE
void WdxcDiscover(dmConnId_t connId)
{
        if (datcCb.discState[connId - 1] > DATC_DISC_WDXC_SCV)
        {
          APP_TRACE_INFO0("*** Discover files");
          WdxcDiscoverFiles(connId, datcCb.fileList[connId - 1], DATC_WDXC_MAX_FILES);
          am_menu_printf(" - WDXC File discovery completed from peripheral with conn_id = %d\r\n", connId);
        }
}
#endif
```

Ilustración 196:WdxcDiscover.

Por otro lado, se ha incluido en el código que implementa la funcionalidad del dispositivo *Central* la función *WdxcStartStopStrm,* desde la que, o bien se finalizará o bien se iniciará el flujo de datos WDX desde el dispositivo *Peripheral*, a partir de la llamada a la función *WdxcStremStop* o *WdxStreamStart*, respectivamente, como se muestra en la Ilustración 197.

```
#if WDXC_INCLUDED == TRUE
void WdxcStartStopStrm(dmConnId_t connId)
{
        if (datcCb.streaming[connId - 1])
        {
          APP_TRACE_INFO0("*** Stream stopped");
          am_menu_printf(" - Stop WDXC Streaming from peripheral with conn_id = %d\r\n", connId);
         /* Stop the WDXC data stream */
          WdxcStreamStop(connId);
        }
        else if (datcCb.discState[connId - 1] > DATC_DISC_WDXC_SCV)
        {
          APP_TRACE_INFO0("*** Stream started");
          am_menu_printf(" - Start WDXC Streaming from peripheral with conn_id = %d\r\n", connId);
          /* Start the WDXC data stream */
          WdxcStreamStart(connId, datcCb.streamHandle[connId - 1]);
        }
}
#endif
```

Ilustración 197: WdxcStartStopStream.

Finalmente, en la función *datcProcMsg*, que se encarga de gestionar los diferentes eventos *DM* que contempla la pila *Cordio BLE*, se ha incluido el código mostrado en la Ilustración 198 con el fin de procesar los eventos DM asociados a la expiración del *timer* utilizado para el cálculo del data *rate,* así como su detención, junto con la del envío del flujo de datos en caso de que se cierre la conexión.

```
static void datcProcMsg(dmEvt_t *pMsg)
{
  uint8_t uiEvent = APP_UI_NONE;

  switch(pMsg->hdr.event)
  {
#if WDXC_INCLUDED == TRUE
    case DATC_WDXS_DATA_RATE_TIMER_IND:
      APP_TRACE_INFO0("*** datcProcMsg - DATC_WDXS_DATA_RATE_TIMER_IND");
      datcWdxcProcessDataRateTimeout((dmConnId_t) pMsg->hdr.param);
      break;
#endif /* WDXC_INCLUDED */


#if WDXC_INCLUDED == TRUE
      WsfTimerStop(&datcCb.dataRateTimer[pMsg->hdr.param - 1]);
      datcCb.streaming[pMsg->hdr.param - 1] = FALSE;
#endif /* WDXC_INCLUDED */
      uiEvent = APP_UI_CONN_CLOSE;
      break;

#if CS50_INCLUDED == TRUE
    case DM_PHY_UPDATE_IND:
      APP_TRACE_INFO0("*** datcProcMsg - DM_PHY_UPDATE_IND");
      APP_TRACE_INFO2("DM_PHY_UPDATE_IND - RX: %d, TX: %d", pMsg->phyUpdate.rxPhy, pMsg->phyUpdate.txPhy);
      datcCb.phyMode[pMsg->hdr.param-1] = pMsg->phyUpdate.rxPhy;
      break;
#endif /* CS50_INCLUDED */

    default:
      break;
  }

  if (uiEvent != APP_UI_NONE)
  {
    AppUiAction(uiEvent);
  }
}
```

Ilustración 198: datcProcMsg.

5.3.2 Fichero *Ble_menu.c*

El menú es la interfaz directa con el usuario, que como se ha explicado a lo largo del documento, se realizará a través de la *UART,* mostrándose las distintas opciones que se pueden realizar desde el dispositivo *Central.* Al añadir la funcionalidad de la transferencia de flujo de datos WDX, se han incluido

más opciones a partir de las cuales será posible invocar desde el menú de acciones todas las que se han visto a lo largo de este capítulo. La Ilustración 199 muestra el nuevo menú a la hora de enviar datos, permitiendo que se realice de la manera convencional, o como *Stream*.

```
char datMenuContent[DAT_MENU_ID_MAX][64] = {
    "1. WDXC Discover files on periheral (WDXC/WDXS_INCLUDED = TRUE)",
    "2. WDXC Start/Stop Streaming (WDXC/WDXS_INCLUDED =  TRUE)",
    "3. Send BLE_DATA 'OK' (WDXC/WDXS_INCLUDED =  FALSE)",
};
```

Ilustración 199: Menu Stream.

Así, en el menú inicialmente implementado se incluyen dos nuevas opciones; una para realizar el descubrimiento de los ficheros del EFS que contiene el dispositivo *Peripheral,* y otra para comenzar/finalizar el envío de flujo de datos WDX. Como se observa en la Ilustración 200 , al añadir estas dos opciones, en el gestor de la función *dats* se añaden las funciones explicadas anteriormente, equivalentes a estas.

```
#if WDXC_INCLUDED == TRUE
            if (bleMenuCb.datMenuSelected == DAT_MENU_ID_NONE)
            {
                am_menu_printf("Choose a conn_id to perform WDXC Discover files (%d):\r\n", numConnections);
                bleMenuCb.datMenuSelected = DAT_MENU_DISCOVER_STRM;
            }
            else
            {
                uint8_t cid = menuRxData[0] - '0';
//              am_menu_printf(" - WDXC Discover files from peripheral with conn_id = %d\r\n", cid);
                WdxcDiscover(cid);
                bleMenuCb.datMenuSelected = DAT_MENU_ID_NONE;
            }
#else
            am_menu_printf(" !!! Warning - WDXC_INCLUDED / WDXS_INCLUDED = FALSE\r\n");
#endif
            break;
        case DAT_MENU_STARTSTOP_STRM:
#if WDXC_INCLUDED == TRUE
            if (bleMenuCb.datMenuSelected == DAT_MENU_ID_NONE)
            {
                am_menu_printf("Choose a conn_id to Start/Stop WDXC Streaming (%d):\r\n", numConnections);
                bleMenuCb.datMenuSelected = DAT_MENU_STARTSTOP_STRM;
            }
            else
            {
                uint8_t cid = menuRxData[0] - '0';
//              am_menu_printf(" - Start/Stop WDXC Streaming from peripheral with conn_id = %d\r\n", cid);
                WdxcStartStopStrm(cid);
                bleMenuCb.datMenuSelected = DAT_MENU_ID_NONE;
            }
#else
            am_menu_printf(" !!! Warning - WDXC_INCLUDED / WDXS_INCLUDED = FALSE\r\n");
#endif
            break;
        case DAT_MENU_ID_SEND_OK:
#if WDXC_INCLUDED == FALSE
            if (bleMenuCb.datMenuSelected == DAT_MENU_ID_NONE)
            {
                am_menu_printf("Choose a conn_id from open connections (%d):\r\n", numConnections);
                bleMenuCb.datMenuSelected = DAT_MENU_ID_SEND_OK;
            }
            else
            {
                uint8_t cid = menuRxData[0] - '0';
                am_menu_printf(" - Send 'ok' data to server with conn_id = %d\r\n", cid);
                DatcSendOKData(cid);
                bleMenuCb.datMenuSelected = DAT_MENU_ID_NONE;
            }
#else
            am_menu_printf(" !!! Warning - WDXC_INCLUDED / WDXS_INCLUDED = TRUE\r\n");
#endif
            break;
        default:
            break;
```

Ilustración 200: handleDatSelection Stream menu.

Capítulo 6: Estudio y optimización del *Data Throughput*

En este capítulo se presentan las pruebas y mejoras necesarias para intentar obtener las mejores prestaciones posibles en términos de *Throughput* a partir de los dispositivos *Peripheral* y *Central* implementados para la transferencia de *Stream* de datos WDX en la plataforma *Artemis Thing Plus* de *Sparkfun*.

6.1 *Data Throughput*

Entre los diferentes factores que determinan el *Througthput* de datos que puede alcanzar *BLE 5,* destacan los siguientes:

1) El modo PHY utilizado (*LE 1M, LE2M* o LE *Coded*).
2) El valor del Intervalo de Conexión (CI).
3) El valor de la unidad máxima de transferencia *ATT* (*ATT* MTU).
4) La característica de extensión de la longitud de datos (DLE).
5) El tipo de operación: *Write with response/ Write without response, Indication/Notification*.
6) El valor del parámetro IFS (*Inter Frame Space*) de la especificación BLE, que corresponde al lapso de tiempo entre la transmisión paquetes consecutivos (150 µs).
7) La transmisión de paquetes vacíos (Empty PDU).
8) El *overhead* de los paquetes, ya que no todos los bytes de un paquete se utilizan para la transferencia del *payload* de la aplicación.

6.1.1 Modo PHY

En BLE 5 se definen básicamente tres modos PHY, correspondientes a *Classic 1 Mbps* PHY, *2 Mbps* PHY, y *Coded* PHY. El PHY utilizado influye directamente en el máximo rendimiento que puede obtenerse, puesto que determina la tasa a la que los paquetes se envían a través del aire. Sin embargo, el modo 2 Mbps PHY requiere de un hardware actualizado, y no todos los dispositivos lo poseen. Tanto la opción *LE 1M PHY* como *LE 2M PHY* se denominan *Uncoded* PHY, ya que en ellos se usa una representación de 1 símbolo por bit de datos (en comparación con la opción *LE Coded PHY,* en la que se usa una representación de 2 u 8 símbolos por bit de datos).

No obstante, a pesar de lo que pueda parecer, las tasas de 1 Mbps (*LE 1M PHY*), 2Mbps (*LE 2M PHY*), 125 Kbps y 500 Kbps (*LE Coded* PHY con S=8 y S=2), respectivamente, representan la tasa a la que la

radio transmite datos, pero en la práctica no es alcanzable para el *throughput* de la aplicación, debido a factores que se comentarán posteriormente, como el límite en el número de paquetes que se transmiten por Intervalo de Conexión, el retardo entre paquetes consecutivos (IFS), los paquetes vacíos, etc. En la Ilustración 201 se muestra el formato de un paquete BLE 5 para el caso *Uncoded LE*.

Packet format for Uncoded LE data packets

<table>
<tr><th>Preamble</th><th>Access Address</th><th colspan="6">PDU (2-257 bytes)</th><th>CRC</th></tr>
<tr><td rowspan="3">1 byte
(1M PHY)
2 bytes
(2M PHY)</td><td rowspan="3">4 bytes</td><td>LL Header</td><td colspan="4">Payload (0-251 bytes)</td><td rowspan="2">MIC
(Optional)</td><td rowspan="3">3 bytes</td></tr>
<tr><td rowspan="2">2 bytes</td><td>L2CAP Header</td><td colspan="3">ATT Data (0-247 bytes)</td></tr>
<tr><td rowspan="2">4 bytes</td><td colspan="2">ATT Header</td><td>ATT Payload</td><td rowspan="2">4 bytes</td></tr>
<tr><td>Op Code
1 byte</td><td>Attribute Handle
2 bytes</td><td>Up to 244 bytes</td></tr>
</table>

Ilustración 201: Formato de paquete Uncoded LE.

De los campos definidos en este formato de paquete, el que determina realmente el tamaño de los datos de la aplicación, es *ATT Payload*, donde:

- En BLE 4.0 y 4.1, el máximo *ATT Payload* es de 20 bytes.
- En BLE 4.2 Y 5.0, una característica vista anteriormente, denominada DLE, permite que el campo *ATT Payload* incluya hasta 244 bytes de datos.

6.1.2 Intervalo de Conexión y número máximo de paquetes por conexión.

El Intervalo de Conexión, como ya se ha comentado con anterioridad, determina el número máximo de paquetes que pueden enviarse durante un Evento de Conexión. En principio, cuanto mayor sea este valor, más paquetes pueden enviarse en un Evento de Conexión (hasta un determinado límite).

Así, el Evento de Conexión representa el tiempo entre dos eventos de transferencia de datos entre los dispositivos *Central* y *Peripheral*. Un Evento de Conexión es iniciado por el dispositivo *Central* mediante la transmisión de un paquete. En caso de que el dispositivo *Peripheral* reciba satisfactoriamente el paquete desde el dispositivo *Central,* el dispositivo *Peripheral* transmitirá un paquete hacia el dispositivo *Central.* Por defecto, los dispositivos *Central* y *Peripheral* transmiten un paquete incluso en caso de que no dispongan de datos que enviar (*Empty PDU).* En la Ilustración 202, se representa el envío de paquetes simples en un Evento de Conexión.

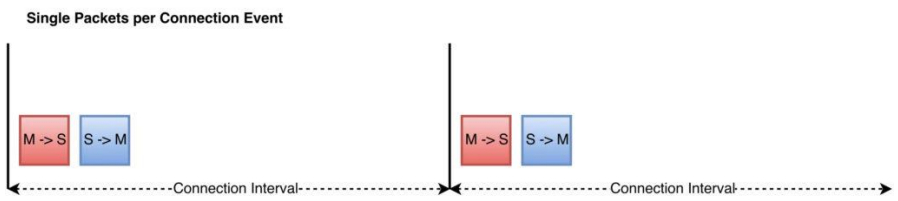


Ilustración 202: Paquetes simples por Evento de Conexión [24].

En los casos en los que los dispositivos *Central/Peripheral* disponen de datos almacenados, pueden solicitar o indicar al otro dispositivo que dispone de más datos para enviar antes del final del Evento de Conexión. Esto puede darse en el intercambio de múltiples paquetes durante un Intervalo de Conexión. El Evento de Conexión continua hasta que el envío de un paquete falle o éste se reciba incorrectamente, el dispositivo emisor esté de acuerdo con finalizar el Evento de Conexión, o en un escenario atípico, se alcance el final del Intervalo de Conexión. Así, en un Evento de Conexión es posible intercambiar más de 2 paquetes, lo que permitiría incrementar el *Throughput*.

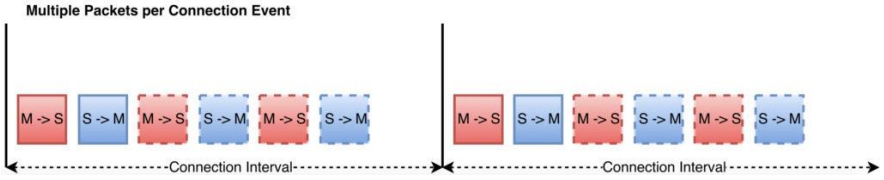


Ilustración 203: Paquetes múltiples por Evento de Conexión [24].

Sin embargo, el envío de paquetes por Evento de Conexión depende de cada dispositivo y de la pila BLE, pudiendo estar limitado entre dispositivos y versiones de la pila en un dispositivo específico. Este valor también depende de la operación del dispositivo, de forma que la radio pueda tener que atender a otros eventos, y el número de paquetes enviados por Evento de Conexión puede no alcanzar el máximo permitido por la pila BLE.

El valor mínimo y el valor máximo del Intervalo de Conexión está limitado a 7,5 ms y 4s, respectivamente, estableciéndose en unidades de 1,25 ms. Así, el Intervalo de Conexión representa un retardo fijo entre el inicio de los Eventos de Conexión, mientras que este último representa la transferencia activa de paquetes entre los dispositivos *Maestro/Esclavo*. El final de un Evento de Conexión está determinado por el campo *More Data* (MD) de la cabecera de un paquete de la capa LL (*Link Layer)*. En la Ilustración 204 se representan los diferentes campos que introduce la capa LL al paquete BLE.

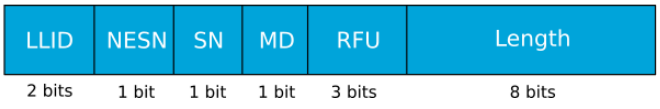


Ilustración 204: Cabecera header DLE paquete LL [24].

Así, la cantidad de datos transmitidos en un único Evento de Conexión está limitada por lo general, por la cantidad de tiempo que las radios de los dispositivos *Central/Peripheral* están asignadas a la transmisión/recepción de datos (que normalmente se referencia como longitud del Evento de Conexión LE *length*). Por esto, un cambio de canal es equivalente a un cambio de Evento de Conexión. La duración del Evento de Conexión no puede ser mayor que el Intervalo de Conexión, pero es un parámetro que no se negocia entre los dispositivos *Central/Peripheral.* El valor de la longitud del Evento de Conexión puede establecerse entre un valor de 0 y 65535, en unidades de 1,25 ms.

6.1.3 Máxima *ATT* MTU

El parámetro *ATT* MTU determina la máxima cantidad de datos que puede ser gestionada por los dispositivos *Central/Peripheral*, y que puede almacenarse en sus buffers. *ATT* MTU representa la longitud máxima de un paquete *ATT*. El parámetro *ATT* MTU se define en la capa *L2CAP*, y su valor puede estar entre 23 bytes e infinito.

El campo *OP-Code* indica la operación *ATT* (Notificación, *Response*…), mientras que, en el envío de una Notificación, *Read, Write, Read* o *Indication*, se necesita introducir adicionalmente el campo *Attribute Handle*, de forma que en este caso, el tamaño de datos de aplicación enviados en un paquete BLE, es de (*ATT* MTU – 3 octets). En la Ilustración 205, se desglosan los campos dentro de *ATT Data* se usan para indicar la operación.

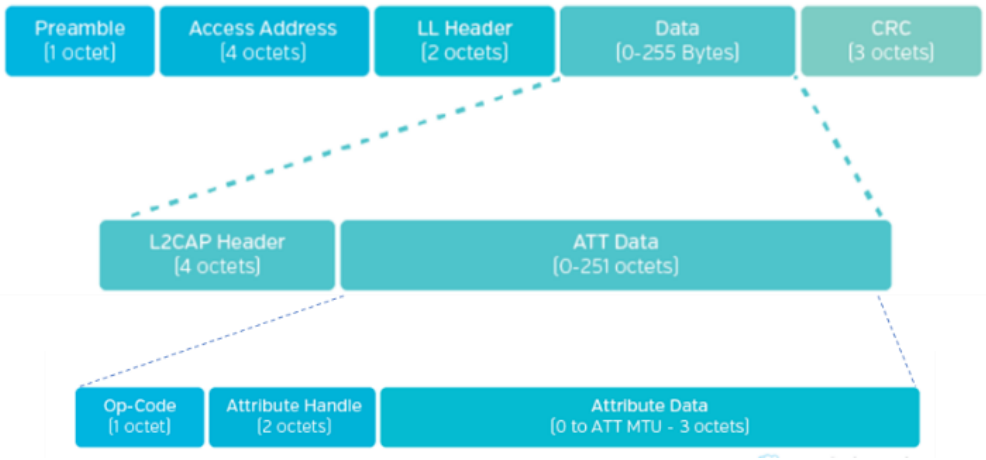


Ilustración 205: Paquete ATT MTU 3 Octets.

El mínimo valor del MTU soportado es de 27 bytes, lo cual permite la transferencia de hasta 20 bytes de *ATT Payload* por paquete BLE (3 bytes se usan para la cabecera *ATT*, y 4 bytes para la cabecera *L2CAP*).

Si bien no existe límite en la especificación del estándar sobre el valor máximo del MTU, cada pila puede tener sus propias limitaciones. Por ejemplo, si se habilita la característica DLE, se pueden transferir hasta 251-4 = 247 bytes (tras considerar el tamaño de la cabecera *L2CAP*). Tras tener en consideración la cabecera *ATT* (3 bytes), restan 244 bytes para el *Payload ATT* en un paquete BLE. Así, en caso de que el MTU sea al menos de 247 bytes, ocupa un único paquete, pero si el MTU es mayor de 247 bytes, será

dividido en múltiples paquetes, lo que hará que el *Throghput* se reduzca. El MTU efectivo se determina al mínimo valor de *ATT* MTU que requieren los dispositivos *Central/Peripheal.*

6.1.4 DLE

En la especificación 4.2 del estándar BLE, el SIG introducía como novedad la característica DLE (*Data Lenght Extension*). Esta característica, añadida en la capa LL, permite que el campo PDU pueda incrementarse, de los 27 bytes por paquete, hasta los 251 bytes, lo que dejaría 244 bytes de datos, como máximo, por paquete BLE. La configuración de la característica DLE dependerá en cada caso del chip, la pila BLE, y el SDK utilizado.

En todo caso, la habilitación de la característica DLE se produce cuando se inicia el procedimiento *Data Lenght Update* a partir del envío de un comando de tipo *LL_LENGTH_REQ*, que debe ser respondido con un comando de tipo *LL_LENGTH_RESP* para completar el procedimiento. En este procedimiento, se negocian los siguientes parámetros entre los dispositivos *Central/Peripheral*:

- *connMaxTXOctets*, el número máximo de bytes con datos (*payload*) que la pila puede enviar en un paquete de datos simple.
- *connMaxTxTime*, la longitud del tiempo, en microsegundos, durante la cual la pila puede trasmitir activamente un paquete de datos simple.
- *connMaxRxOctets*, el máximo número de bytes con datos que la pila puede recibir en un paquete de datos simple.
- *connMaxRxTime*, la longitud del tiempo, en microsegundos, durante la cual la pila puede recibir activamente un paquete de datos.

En la Ilustración 206 se muestra el flujo entre *Maestro/Esclavo* para solicitar y configurar un DLE determinado.

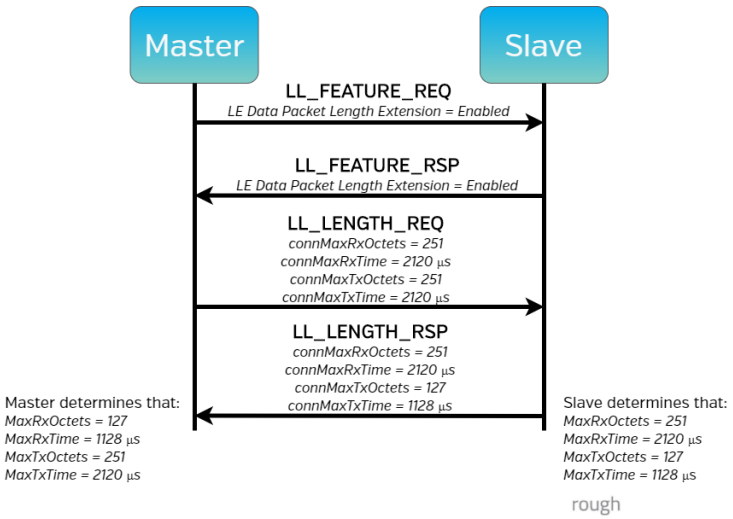


Ilustración 206: DLE Maestro/Esclavo [24].

En el caso expuesto en la Ilustración 207, se puede observar un Evento de Conexión en el que tanto el dispositivo *Central* como el *Peripheral,* envían un paquete de máximo tamaño de *payload*.

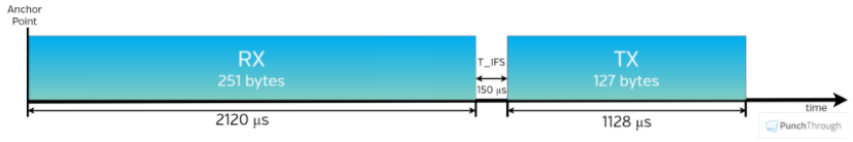


Ilustración 207: Emparejamiento TX/RX en el que cada lado transmite un paquete con un Payload [24].

El valor de los parámetros *connMaxTxTime* y *connMaxRxTime* se calcula normalmente a partir del tamaño máximo en bits de los paquetes de datos en *Link Layer Data Channel*, incluyendo el *overhead*, a la tasa de datos de *LE 1M PHY* (1Mbps). Por ejemplo, cuando el valor del parámetro *connMaxTxOctets* es de 251 bytes, más la cabecera estándar de 14 bytes de un paquete LL, se transmiten (251 bytes + 14 bytes) x 8 bytes = 273 bytes a una tasa de 1Mbps, lo que se completa en 2120µs.

6.1.5 Tipo de operación

Si se desea obtener un elevado *Throughput*, se pueden usar operaciones de tipo Notificación (como en el caso del presente TFG), o *Write Wihout Response* para transferir datos entre los dispositivos, ya que este tipo de operaciones elimina la necesidad de recepción de confirmar los datos, así como de responder antes del envío de un nuevo bloque de datos.

6.1.6 IFS

En la especificación del estándar *Bluetooth* se establece que el intervalo de tiempo entre dos paquetes consecutivos en el mismo canal se denomina *Inter Fram Space* (IFS). Se define como el tiempo desde el final del último bit anterior, hasta el principio del primer bit de la siguiente subsecuencia. El espacio entre tramas (IFS), se denomina "*T_IFS*", y será de 150 µs.

6.1.7 Empty PDU

Si el dispositivo receptor dispone de datos para enviar al otro dispositivo, necesita igualmente enviar un paquete vacío, de acuerdo con lo que se establece en la especificación *Bluetooth,* denominado *Empty* PDU.

6.1.8 *Overhead*

Un paquete BLE incluye datos de *overhead* que no cuentan como datos *ATT*, y que básicamente consumen parte de la tasa de datos de transferencia, sin que cuenten como bytes enviados asociados a los datos de la aplicación. Este *overhead* se corresponde con diferentes campos e información de cabecera definidos en la especificación *Bluetooth.*

6.2 Cálculo del *Data Throughput*

En la Ilustración 208 se observa que la cantidad de *overhead* para cada PHY es ligeramente diferente. El *Preamble* es de 1 byte en el caso de *1M PHY,* y de 2 bytes en el caso de *2M PHY*. El campo *MIC* es un campo opcional que se usa para encriptar la conexión. Para *LE coded*, sigue un formato de paquete, como se muestra en la Ilustración 209.

Packet format for Uncoded LE data packets

<table>
<tr><th>Preamble</th><th>Access Address</th><th colspan="6">PDU (2-257 bytes)</th><th>CRC</th></tr>
<tr><td rowspan="3">1 byte
(1M PHY)
2 bytes
(2M PHY)</td><td rowspan="3">4 bytes</td><td>LL Header</td><td colspan="4">Payload (0-251 bytes)</td><td rowspan="2">MIC
(Optional)</td><td rowspan="3">3 bytes</td></tr>
<tr><td rowspan="2">2 bytes</td><td>L2CAP Header</td><td colspan="3">ATT Data (0-247 bytes)</td></tr>
<tr><td rowspan="2">4 bytes</td><td colspan="2">ATT Header</td><td>ATT Payload</td><td rowspan="2">4 bytes</td></tr>
<tr><td>Op Code
1 byte</td><td>Attribute Handle
2 bytes</td><td>Up to 244 bytes</td></tr>
</table>

Ilustración 208: Formato de paquete Playload LE [25].

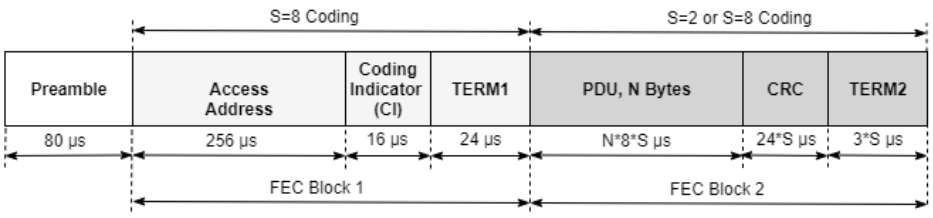


Ilustración 209: Formato de paquete para LE Coded PHY Link Layer [25].

Para la implementación se asumen las siguientes consideraciones:

- Encriptado no activado, por ende, campo MIC vacío.
- Es de interés para el cálculo del *Throughput*, hacer el cálculo en un flujo de envío (por ejemplo, de *Master* a *Slave*), por lo que se asume una sola dirección de transferencia.
- Escrituras sin respuestas (lo cual ayuda a maximizar el *Throughput* cuando la cantidad de datos a transferir es elevada).

Así, los pasos que se establecen para calcular el máximo *Data Throughput,* son:

1º) Determinar el PHY que se está usando, y anotar la tasa de transferencia de datos en bruto.

2º) Determinar el tiempo necesario para enviar un paquete de datos y un paquete vacío desde el receptor.

El tiempo en el cual un paquete de datos puede ser enviado, incluirá los siguientes factores:

Tiempo de transmisión de un paquete de datos = tiempo para transmitir un paquete vacío + IFS + tiempo de transmisión del paquete de datos actual + IFS.

El tiempo de transmisión de un paquete vacío puede calcularse de la siguiente manera:

Tiempo en transmitir paquete vacío = tamaño paquete vacío/ tasa de datos.

Un paquete vacío contiene los siguientes campos:

Preamble + Access Address + LL header + CRC

Para el modo *1M PHY*, el preámbulo está formado por 1 byte, y el total de tamaño de un paquete vacío es igual a 1 + 4 +2 +3 bytes = 10 bytes * 8 bytes = 80 bits: Para *2M PHY*, se incrementa en 1 byte el *preamble*, dando lugar paquetes de 88 bits. Basado en estos cálculos, el tiempo para transmitir un paquete vacío en el modo *1M PHY* será:

80 bits/ 1Mbps = 80 µs.

Un paquete de datos contiene todos los campos que se muestran en la Ilustración 208.

Tiempo para transmitir paquetes de datos = tamaño del paquete/ tasa de datos.

Si se supone que la característica DLE está activa, y el *ATT* MTU es equivalente al máximo número de bits permitidos en un paquete de datos, entonces:

Longitud del paquete = 1 + 4 + 2 + 4 + 247 + 3 bytes = 265 bytes = 256*8 = 2088 bits.

Tiempo para transmitir paquetes de datos = 2088 bits/ 1 Mbps = 2,088 ms.

Tiempo de transmisión de un paquete de datos = tiempo para transmitir un paquete vacío + IFS + tiempo de transmisión del paquete de datos actual + IFS = 2,468 ms

En el caso del modo *2M PHY*, siguiendo el mismo proceso, el tiempo de transmisión de un paquete será de 1,392 ms. Cuando la característica el DLE está activa y el MTU se ajusta a menos de 247 bytes, se tiene un mayor *overhead* (ya que ahora los datos son mayores que la MTU *ATT*, dividiéndose en más paquetes). Por ejemplo, si se tiene una *ATT* MTU configurada a 158 bytes, para transferir 244 bytes de datos de aplicación, se necesita enviar dos paquetes, haciendo que el *Thoughput* se reduzca debido a un

incremento en el *overhead,* además de añadir más IFS entre paquetes. Otro escenario posible sería desactivar la característica DLE (27 bytes de *payload*) con una MTU mayor que 27 bytes. Aquí, se tienen que enviar más paquetes equivalentes a la cantidad de datos, provocando una previsible bajada en el *Throughput.*

3º) Calcular cuántos paquetes pueden transmitirse durante un Intervalo de Conexión.

Estos cálculos no son puramente matemáticos, ya que se necesita tener en cuenta las limitaciones de la pila y el dispositivo usado. Dicho esto, en una MCU el SDK del proveedor suele indicar este máximo en su documentación, aunque también es útil hacer una validación experimental y determinar las limitaciones de cada dispositivo específico.

Una vez calculado este máximo, se puede calcular el máximo teórico del número de paquetes enviados para un intervalo determinado. Por ejemplo, si se tiene un Intervalo de Conexión de 7,5 ms (el mínimo permitido impuesto en la especificación) usando el modo *1M PHY*:

Máximo número de paquetes de datos por intervalo = Intervalo de Conexión / tiempo de envío de un paquete = 7.5ms/ 2,468 ms = 3 paquetes. (redondeo a la baja)

Normalmente, este número no es realista, ya que no cuenta el tiempo de retardo entre paquetes consecutivos en un Evento de Conexión. En la práctica, seguramente se podrán enviar 2 paquetes en un Intervalo de Conexión, en lugar de 3.

4º) Cálculo del número máximo paquetes de datos que pueden ser transmitidos en un Intervalo de Conexión.

***Data Throughput* = datos por Intervalo de Conexión / Intervalo de Conexión = (número de paquetes de datos por Intervalo de Conexión x tamaño del paquete) / Intervalo de Conexión = 2 x 244 x 8 (bits) / 7.5 ms = 520,533 bits/s ≈ 508 kbps.**

6.2.1 Cálculos iniciales del máximo *Data Throughput* estimado

Para realizar los cálculos, hasta ahora, se han supuesto parámetros aleatorios. En este apartado, se realizan los cálculos teóricos con los parámetros iniciales a usar en el presente TFG:

- Modo PHY: 1 Mbps.
- *ATT* MTU: 251 bytes.
- DLE: activo.
- Intervalo de Conexión: 100 x 1,25ms = 125 ms.

Tiempo necesario para transmitir un paquete:

Tamaño del paquete BLE / tasa de datos = (1 + 4 + 2 + 4 + 251 + 3) x 8 bits / 1Mbps = 2120 µs.

Tiempo necesario para transmitir un paquete BLE hasta recibir un paquete vacío:

Tiempo para transmitir un paquete vacío + IFS + tiempo de transmisión de un paquete + IFS = 80 µs + 150 µs + 2120 µs + 150 µs = 2500 µs.

Máximo número de paquetes BLE por Intervalo de Conexión:

(Intervalo de Conexión / tiempo para transmitir un paquete BLE y recibir un mensaje vacío) = 125 ms / 2120 µs = 58,96

Esto hace 58 paquetes, aunque la plataforma *Artemis Thing Plus* limita este valor a 4 paquetes, por lo que, todos los cálculos se harán con este valor.

Data Throughput:

Datos Transferidos por Intervalo de Conexión / Intervalo de Conexión = 992 x 8 (bits) / 125 ms = 63,48 Kbps.

Si se compara este *Data Throughput* con el anteriormente calculado, teniendo en cuenta que se aumenta el Intervalo de Conexión, resulta evidente que no es óptimo.

6.2.2 Optimización de Data *Throughput* en los dispositivos *Central/Peripheral* implementados

Para maximizar el rendimiento de la conexión entre los dispositivos *Central/Peripheral* implementados, se deben tener en cuenta las siguientes consideraciones:

- Activar siempre la característica DLE, obviamente si se usa *Bluetooth v4.1* o anterior, esta opción no será viable.
- Usar el modo *LE 2M PHY*. Si ambos dispositivos soportan BLE 5, normalmente, la opción PHY más adecuada es *LE 2M PHY*. Usar *LE 2M PHY* es el modo en el que se consigue el mejor compromiso entre consumo y rendimiento.
- Usar en la conexión solo *Notificaciones* y *Escrituras sin respuestas*, eliminando así paquetes innecesarios ya que, a diferencia de las Indicaciones o las Escrituras, no requieren de *acknowledge (*confirmación de recepción).
- Elegir *ATT* MTU de 247 bytes.
- Elegir un Intervalo de Conexión que permita el máximo envío de paquetes por intervalo, pero siempre teniendo en cuenta que cuanto más frecuente es el intervalo, mayor será el consumo de energía. Elegir un Intervalo de Conexión demasiado elevado afectará al rendimiento directamente. A la hora de elegir el valor del Intervalo de Conexión, también se deben tener

en cuenta las limitaciones físicas de los dispositivos (*Central y Peripheral)*, como en el caso del presente TFG.

Para maximizar el *Thoughput* a nivel de aplicación conviene maximizar el uso de los diferentes *buffers.* Para ello, lo idóneo es adecuar la longitud de los paquetes a controlar minimizando el tamaño del *overhead,* siempre que sea posible. Hay que entender que la transmisión y recepción de paquetes siempre interactúa con la *HCI (Interfaz Host/*Controlador). Por lo tanto, para leer la información de esta capa se define en una variable denominada *LE_READ_BUF SIZE*. Esto supone una limitación en el tamaño que el Controlador puede aceptar, siendo una de las razones por las que se fragmenta la información que llega. Se necesita configurar la aplicación para mejorar el MTU en determinados *buffers*, ya que el tamaño máximo de MTU es directamente proporcional al máximo número de paquetes soportados por el Controlador, y el número de *buffers* que soporta. Por ejemplo, si el Controlador soporta como máximo paquetes de 256 bytes, y hay 4 *buffers* disponibles, entonces la aplicación podría enviar un *payload* de 256 x 4 – 4 (*overhead)* = 1020 bytes, los cuales podrían ser fragmentados por la interfaz *HCI* y pasados al Controlador en 4 *payloads* diferentes. Esto esencialmente mantiene los *buffers* llenos y maximiza el uso del canal implementado.

6.3 Configuración de los parámetros en los dispositivos *Central/Peripheral*

Según lo presentado en apartados anteriores, se expone a continuación cómo se han configurado los principales parámetros que afectan directamente al *Throughput*, tanto en el dispositivo *Central,* como en el dispositivo *Peripheral* implementados en el presente TFG.

6.3.1 Configuración PHY

- Dispositivo *Central:*

En el *makefile,* se define el parámetro de configuración *DCS50_ INCLUDED* (Ilustración 210) activando los diferentes modos PHY soportado por la plataforma.

```
#FBTG
DEFINES+= -DL2C_TRACE_ENABLED
DEFINES+= -DWDXC_INCLUDED
DEFINES+= -DCS50_INCLUDED
DEFINES+=
```

Ilustración 210: Define PHY en el Central

Dentro del fichero *main.c*, se definen los tres PHY posibles, con un identificador diferente, como se observa en la Ilustración 212.

```
/**********************************************
Macros
**********************************************
#if CS50_INCLUDED == TRUE
/* PHY Test Modes */
#define DATC_PHY_1M                     1
#define DATC_PHY_2M                     2
#define DATC_PHY_CODED                  3
#endif /* CS50_INCLUDED */
```

Ilustración 211: datc_main define PHY.

En la función *datcOpen,* al establecer la conexión dentro del archivo *datc_main.c*, se configura el modo de PHY a usar, como se indica en la Ilustración 212.

```
static void datcOpen(dmEvt_t *pMsg)
{
#if CS50_INCLUDED == TRUE
//FBTG
  datcCb.phyMode[pMsg->hdr.param-1] = DATC_PHY_1M;
#endif /* CS50_INCLUDED */
#if WDXC_INCLUDED == TRUE
        APP_TRACE_INFO0("*** datcOpen.");
  datcCb.streamHandle[pMsg->hdr.param-1] = WSF_EFS_INVALID_HANDLE;
#endif /* WDXC_INCLUDED */
}
```

Ilustración 212: datcOpen configuración PHY.

- Dispositivo *Peripheral.*

En el *makefile* y en el fichero *dats_main*, se llevan a cabo las mismas declaraciones que en el dispositivo *Central*. Como se muestra en la Ilustración 213, al atender el evento *DM_CONN_OPEN_IND*, el dispositivo *Peripheral* indica qué modos *PHY* soporta. Como se expone en la Ilustración 212 , el dispositivo *Central* impone el PHY, de manera que siempre que sea físicamente posible, será el PHY que se use en la conexión.

```
    case DM_CONN_OPEN_IND:
      uiEvent = APP_UI_CONN_OPEN;

#if CS50_INCLUDED == TRUE
//FBTG
      datsCb.phyMode = DATS_PHY_1M;
      datsCb.phyMode = DATS_PHY_2M;
#endif /* CS50_INCLUDED */
```

Ilustración 213: PHY en el evento DM_COMN_OPEN_IND.

6.3.2 Configuración del Intervalo de Conexión y del número máximo de paquetes por Evento de Conexión

- Dispositivo *Central*. Como se ha explicado detalladamente con anterioridad, el dispositivo *Central* establece los valores que se deberán usar como parámetros de conexión, una vez el dispositivo *Peripheral* es descubierto. Se describen las diferentes situaciones que se pueden dar a la hora de su configuración.

Usar parámetros por defecto:

En el fichero de cabecera disponible en la ruta *AmbiqSDK\third-party\exactle\ble-Host\include\dm_api.h* se establecen los parámetros de conexión que soporta GAP con esta librería. Los parámetros GAP por defecto se representan en la Ilustración 214.

```
/** \name GAP Connection Slave Latency
 *
 */
/**@{*/
/*! \brief GAP connection establishment slaves latency */
#define DM_GAP_CONN_EST_LATENCY     0
/**@}*/

/** \name GAP Connection Interval
 * GAP connection interval in 1.25ms units.
 */
/**@{*/
#define DM_GAP_INITIAL_CONN_INT_MIN 24      /*!< \brief Minimum initial connection interval */
#define DM_GAP_INITIAL_CONN_INT_MAX 40      /*!< \brief Maximum initial connection interval */
/**@}*/

/** \name GAP Connection Event Lengths
 * GAP connection establishment minimum and maximum connection event lengths.
 */
/**@{*/
#define DM_GAP_CONN_EST_MIN_CE_LEN  0  /*!< \brief Connection establishment minimum event length */
//FBTG
//#define DM_GAP_CONN_EST_MAX_CE_LEN  0  /*!< \brief Connection establishment maximum event length */
#define DM_GAP_CONN_EST_MAX_CE_LEN  0xffff  /*!< \brief Connection establishment maximum event length */
/**@}*/
```

Ilustración 214: parámetros predeterminados de conexión GAP.

En la pila, para atender los eventos *DM* de conexión, se usa el fichero *dm_conn.c,* en el que se establecen los valores por defecto, como se muestra en la Ilustración 215 y la Ilustración 216.

```
static const hciConnSpec_t dmConnSpecDefaults =
{
  DM_GAP_INITIAL_CONN_INT_MIN,
  DM_GAP_INITIAL_CONN_INT_MAX,
  DM_GAP_CONN_EST_LATENCY,
  DM_DEFAULT_EST_SUP_TIMEOUT,
  DM_GAP_CONN_EST_MIN_CE_LEN,
  DM_GAP_CONN_EST_MAX_CE_LEN
};
```

Ilustración 215: Parámetros por defecto conn.

En el módulo *DM*, se configuran tras un *reset* los parámetros por defecto en la conexión. La Ilustración 216 muestra cómo se definen tras inicializar la estructura definida por defecto en el bloque de control.

```
void dmConnReset(void)

    APP_TRACE_INFO0("%%%% DM_CONN: dmConnReset %%%%");

dmConnCcb_t             *pCcb = dmConnCb.ccb;
hciDisconnectCmplEvt_t  disconnectCmpl;
uint8_t                 i;

/* generate HCI disconnect complete event */
disconnectCmpl.hdr.event = HCI_DISCONNECT_CMPL_CBACK_EVT;
disconnectCmpl.hdr.status = disconnectCmpl.status = HCI_SUCCESS;
disconnectCmpl.reason = HCI_ERR_LOCAL_TERMINATED;

for (i = DM_CONN_MAX; i > 0; i--, pCcb++)
{
  if (pCcb->inUse)
  {
    /* set connection id */
    disconnectCmpl.hdr.param = disconnectCmpl.handle = pCcb->handle;

    /* handle the event */
    dmConnHciHandler((hciEvt_t *) &disconnectCmpl);
  }
}

/* initialize control block */
for (i = 0; i < DM_NUM_PHYS; i++)
{
  dmConnCb.scanInterval[i] = DM_GAP_SCAN_FAST_INT_MIN;
  dmConnCb.scanWindow[i] = DM_GAP_SCAN_FAST_WINDOW;
  dmConnCb.connSpec[i] = dmConnSpecDefaults;
    APP_TRACE_INFO2("%%%% DM_CONN: dmConnReset initialize i=%d with dmConnSpecDefaults->minCeLen = %d %%%%", i, dmConnSpecDefaults.minCeLen);
    APP_TRACE_INFO2("%%%% DM_CONN: dmConnReset initialize i=%d with dmConnSpecDefaults->maxCeLen = %d %%%%", i, dmConnSpecDefaults.maxCeLen);

}

dmCb.initFiltPolicy = HCI_FILT_NONE;
dmCb.connAddrType = DM_ADDR_PUBLIC;
```

Ilustración 216: Configuración de conexión desde dmConnReset().

Establecimiento de los parámetros preferidos:

Los parámetros de conexión de *HCI* se definen en el fichero de cabecera disponible en el directorio *AmbiqsuitSDK\third_party\exactle\wsf\include\HCI_defs.h.* En la Ilustración 217 se muestran los diferentes parámetros y su dirección, desde *hc_defs.h.*

```
/** \name Packet definitions
 *
 */
/**@{*/
#define HCI_CMD_HDR_LEN                     3       /*!< \brief Command packet header length */
#define HCI_ACL_HDR_LEN                     4       /*!< \brief ACL packet header length */
#define HCI_ISO_HDR_LEN                     4       /*!< \brief ISO packet header length */
#define HCI_EVT_HDR_LEN                     2       /*!< \brief Event packet header length */
#define HCI_EVT_PARAM_MAX_LEN               255     /*!< \brief Maximum length of event packet parameters */
//FBTG
//#define HCI_ACL_DEFAULT_LEN                 27      /*!< \brief Default maximum ACL packet length */
#define HCI_ACL_DEFAULT_LEN                 255     /*!< \brief Default maximum ACL packet length */
```

Ilustración 217: Parámetros de conexión HCI.

En la Ilustración 218 se muestra el establecimiento de los parámetros de conexión.

```
/*! Connection parameters */
static const hciConnSpec_t datcConnCfg =
{
//FBTG
//  40,                                 /*! Minimum connection interval in 1.25ms units */
//  40,                                 /*! Maximum connection interval in 1.25ms units */
  100,                                /*! Minimum connection interval in 1.25ms units */ //- minimo = 6
  100,                                /*! Maximum connection interval in 1.25ms units */
  0,                                  /*! Connection latency */
  600,                                /*! Supervision timeout in 10ms units */
  0,                                  /*! Unused */ //FBTG - minCElen
//  0                                   /*! Unused */ //FBTG - maxCElen
  65535                                 /*! Unused */ //FBTG - maxCElen
};
```

Ilustración 218: configuración conexión desde el Central.

Cuando llega un evento desde *DM,* indicando que se ha completado la conexión, se llama a la función *datcSetup(),* como se muestra en la Ilustración 219.

```
case DM_RESET_CMPL_IND:
  APP_TRACE_INFO0("*** datcProcMsg - DM_RESET_CMPL_IND");
  AttsCalculateDbHash();
  DmSecGenerateEccKeyReq();
  datcSetup(pMsg);
```

Ilustración 219: datcProcMsg() evento DLE reset.

En ella, se llama a la función propia de *DM DmConnSetConnSpec* (Ilustración 220).

```
static void datcSetup(dmEvt_t *pMsg)
{
  datcCb.scanning = FALSE;
  datcCb.autoConnect = FALSE;
  datcConnInfo.doConnect = FALSE;

  DmConnSetConnSpec((hciConnSpec_t *) &datcConnCfg);
}
```

Ilustración 220: DatcSetup() Central.

La Ilustración 221 muestra la función *DmConnsetConnSpec*, propia de la pila *Cordio*, y ubicada en *ble_Host/sources/stack/dm/dm_conn*. Esta función establece los parámetros de especificación de la conexión (Ilustración 222) para las conexiones que se crearán mediante la función *DmConnOpen().* Estas dos funciones actualizan la estructura configurada en el archivo *main*.c que se pasa como parámetro para ser usada como parámetros de conexión.

```
void DmConnSetConnSpec(hciConnSpec_t *pConnSpec)
{
  dmConnSetConnSpec(HCI_INIT_PHY_LE_1M_BIT, pConnSpec);

      APP_TRACE_INFO0("%%%% DM_CONN: DmConnSetConnSpec %%%%");
    APP_TRACE_INFO1("%%%% DM_CONN: DmConnSetConnSpec pMsg->connSpec->minCeLen = %d", pConnSpec->minCeLen);
    APP_TRACE_INFO1("%%%% DM_CONN: DmConnSetConnSpec pMsg->connSpec->maxCeLen = %d", pConnSpec->maxCeLen);
}
```

Ilustración 221: DmConnSetConnSpec().

```
static void dmConnSetConnSpec(uint8_t initPhy, hciConnSpec_t *pConnSpec)
{
  WSF_ASSERT((initPhy == HCI_INIT_PHY_LE_1M_BIT) ||
             (initPhy == HCI_INIT_PHY_LE_2M_BIT) ||
             (initPhy == HCI_INIT_PHY_LE_CODED_BIT));

  WsfTaskLock();
  dmConnCb.connSpec[DmInitPhyToIdx(initPhy)] = *pConnSpec;
  WsfTaskUnlock();

      APP_TRACE_INFO0("%%%% DM_CONN: dmConnSetConnSpec %%%%");
}
```

Ilustración 222: dmConnSetConnSpec().

<u>Establecimiento de los parámetros de conexión en el proceso de conexión entre los dispositivos *Central/Peripheral:*</u>

El proceso de establecimiento de conexión sigue el siguiente flujo: para abrir la conexión se usa la función *dmConnOpen()* (Ilustración 223), esto ocurre una vez identificada la dirección del dispositivo a emparejar. Esta función crea la conexión, llamando a su vez a la función *HCILeCreateConnnCmd()*, como se puede observar en la Ilustración 224, creando la conexión interactuando con la interfaz *HCI*. En ésta se configuran y almacenan en su *buffer* los parámetros de conexión realizando un *parse* para ser compatible con el *Stream* de datos (*UINT16_TO_BSTREAM*).

```
static void dmConnOpen(uint8_t initPhys, uint8_t addrType, uint8_t *pAddr)
{
  uint8_t phyIdx = DmScanPhyToIdx(HCI_SCAN_PHY_LE_1M_BIT);

  /* Create connection */
  HciLeCreateConnCmd(dmConnCb.scanInterval[phyIdx], dmConnCb.scanWindow[phyIdx], dmCb.initFiltPolicy,
                     addrType, pAddr, DmL1AddrType(dmCb.connAddrType), &(dmConnCb.connSpec[phyIdx]));

  /* pass connection initiation started to dev priv */
  dmDevPassEvtToDevPriv(DM_DEV_PRIV_MSG_CTRL, DM_DEV_PRIV_MSG_CONN_INIT_START, 0, 0);
}
```

Ilustración 223: DmConnOpen() en dm_conn_master_leg.c.

```
void HciLeCreateConnCmd(uint16_t scanInterval, uint16_t scanWindow, uint8_t filterPolicy,
                        uint8_t peerAddrType, uint8_t *pPeerAddr, uint8_t ownAddrType,
                        hciConnSpec_t *pConnSpec)
{
  uint8_t *pBuf;
  uint8_t *p;

  if ((pBuf = hciCmdAlloc(HCI_OPCODE_LE_CREATE_CONN, HCI_LEN_LE_CREATE_CONN)) != NULL)
  {
    APP_TRACE_INFO0("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");
    APP_TRACE_INFO0("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");
    APP_TRACE_INFO0("%%%%%%%%%%%%%%%%%%% HciLeCreateConnCmd - hciCmdSend(pBuf) %%%%%%%%%%%%%%%%%%");
    APP_TRACE_INFO1("%%%%%%%%%%%%%%%%%%% pConnSpec->minCeLen = %d", pConnSpec->minCeLen);
    APP_TRACE_INFO1("%%%%%%%%%%%%%%%%%%% pConnSpec->maxCeLen = %d", pConnSpec->maxCeLen);
    APP_TRACE_INFO0("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");
    APP_TRACE_INFO0("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");

    p = pBuf + HCI_CMD_HDR_LEN;
    UINT16_TO_BSTREAM(p, scanInterval);
    UINT16_TO_BSTREAM(p, scanWindow);
    UINT8_TO_BSTREAM(p, filterPolicy);
    UINT8_TO_BSTREAM(p, peerAddrType);
    BDA_TO_BSTREAM(p, pPeerAddr);
    UINT8_TO_BSTREAM(p, ownAddrType);
    UINT16_TO_BSTREAM(p, pConnSpec->connIntervalMin);
    UINT16_TO_BSTREAM(p, pConnSpec->connIntervalMax);
    UINT16_TO_BSTREAM(p, pConnSpec->connLatency);
    UINT16_TO_BSTREAM(p, pConnSpec->supTimeout);
    UINT16_TO_BSTREAM(p, pConnSpec->minCeLen);
    UINT16_TO_BSTREAM(p, pConnSpec->maxCeLen);
    hciCmdSend(pBuf);
  }
}
```

Ilustración 224: HCILeCreateConnCmd() en HCI_cmd.c

Una vez los datos que envía el dispositivo *Central* al abrir la conexión se gestionan como datos de *Stream,* para establecerlos en el *Peripheral* se debe realizar el proceso contrario, convertir estos datos *Stream* en *Uint.* En la Ilustración 225 se observa cómo se realiza esta conversión.

```
static void hciEvtParseLeEnhancedConnCmpl(hciEvt_t *pMsg, uint8_t *p, uint8_t len)
{


  BSTREAM_TO_UINT8(pMsg->leConnCmpl.status, p);
  BSTREAM_TO_UINT16(pMsg->leConnCmpl.handle, p);
  BSTREAM_TO_UINT8(pMsg->leConnCmpl.role, p);
  BSTREAM_TO_UINT8(pMsg->leConnCmpl.addrType, p);
  BSTREAM_TO_BDA(pMsg->leConnCmpl.peerAddr, p);
  BSTREAM_TO_BDA(pMsg->leConnCmpl.localRpa, p);
  BSTREAM_TO_BDA(pMsg->leConnCmpl.peerRpa, p);
  BSTREAM_TO_UINT16(pMsg->leConnCmpl.connInterval, p);
  BSTREAM_TO_UINT16(pMsg->leConnCmpl.connLatency, p);
  BSTREAM_TO_UINT16(pMsg->leConnCmpl.supTimeout, p);
  BSTREAM_TO_UINT8(pMsg->leConnCmpl.clockAccuracy, p);

  pMsg->hdr.param = pMsg->leConnCmpl.handle;
  pMsg->hdr.status = pMsg->leConnCmpl.status;
```

Ilustración 225: HCI conversión de los parámetros de conexión a Streaming.

- Dispositivo *Peripheral*. Existe un escenario que aún no ha sido contemplado. A la hora de establecer la conexión existen casos en los que es el dispositivo *Peripheral* el que propone ciertos parámetros de conexión, aunque la última palabra la tiene siempre el dispositivo *Central.* Esto puede ocurrir cuando el dispositivo *Central* intenta establecer unos parámetros de conexión físicamente imposibles de soportar por parte del dispositivo *Peripheral.*

Establecimiento de los parámetros de conexión en el proceso de conexión entre los dispositivos *Central/Peripheral:*

En el fichero *main* del dispositivo *Peripheral,* se configuran los parámetros de conexión. El primer elemento de la estructura se podrá modificar a 1, si el *Peripheral* establece los parámetros de conexión que, como se ha mencionado con anterioridad, por defecto son establecidos por el dispositivo *Central*. (Ilustración 226).

```
/*! configurable parameters for connection parameter update */
static const appUpdateCfg_t datsUpdateCfg =
{
//FBTG
  0,                                    // Connection idle period in ms before attempting
                                        // connection parameter update; set to zero to disable
//  1,                                    // Connection idle period in ms before attempting
//                                        // connection parameter update; set to zero to disable */
  640,                                  /*! Minimum connection interval in 1.25ms units */
  800,                                  /*! Maximum connection interval in 1.25ms units */
  0,                                    /*! Connection latency */
  600,                                  /*! Supervision timeout in 10ms units */
  5                                     /*! Number of update attempts before giving up */
};
```

Ilustración 226: Configuración de parámetros desde el Peripheral.

Si se establece que sea el dispositivo *Peripheral* el que establezca los parámetros de conexión, se seguirá el siguiente flujo: al procesar el evento de abrir conexión, la pila, desde la aplicación del dispositivo *Slave* (Ilustración 227), ejecuta la función *appSlaveConnOpen*(Ilustración 228). Ésta, comprueba si los parámetros de conexión se deben modificar (si es el valor es diferente a 1) y, si es así, entra en la función y se configura el valor de las variables *connInterval, connLatency* y el *timeout* introducidas en el dispositivo *Peripheral.* En el presente TFG, este valor siempre será 0, por lo que en ningún caso será el dispositivo *Peripheral* el que establezca los parámetros de conexión.

```
void AppSlaveProcDmMsg(dmEvt_t *pMsg)
{
  appConnCb_t *pCb = NULL;

  /* look up app connection control block from DM connection ID */
  if ((pMsg->hdr.event != DM_ADV_STOP_IND) &&
      (pMsg->hdr.event != DM_ADV_SET_STOP_IND))
  {
    pCb = &appConnCb[pMsg->hdr.param - 1];
  }

  switch (pMsg->hdr.event)
  {
    case DM_RESET_CMPL_IND:
      appSlaveAdvModeInit();
      break;

    case DM_ADV_SET_STOP_IND:
    case DM_ADV_STOP_IND:
      if (appSlaveCb.advStopCback != NULL)
      {
        (*appSlaveCb.advStopCback)(pMsg);
      }
      break;

    case DM_CONN_OPEN_IND:
      appSlaveConnOpen(pMsg, pCb);
      break;
```

Ilustración 227: AppSlaveProcMsg() en app_Slave.c

```
static void appSlaveProcConnOpen(dmEvt_t *pMsg, appConnCb_t *pCb)
{
  /* store connection ID */
  pCb->connId = (dmConnId_t) pMsg->hdr.param;

    APP_TRACE_INFO0("%%%% appSlaveProcConnOpen %%%%");
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pMsg->connOpen.connInterval = %d", pMsg->connOpen.connInterval);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pMsg->connOpen.connLatency = %d", pMsg->connOpen.connLatency);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pMsg->connOpen.supTimeout = %d", pMsg->connOpen.supTimeout);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pAppUpdateCfg->connIntervalMin = %d", pAppUpdateCfg->connIntervalMin);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pAppUpdateCfg->connIntervalMax = %d", pAppUpdateCfg->connIntervalMax);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pAppUpdateCfg->connLatency = %d", pAppUpdateCfg->connLatency);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pAppUpdateCfg->supTimeout = %d", pAppUpdateCfg->supTimeout);
    APP_TRACE_INFO1("%%%% appSlaveProcConnOpen - pAppUpdateCfg->idlePeriod = %d", pAppUpdateCfg->idlePeriod);



  /* check if we should do connection parameter update */
  if ((pAppUpdateCfg->idlePeriod != 0) &&
      ((pMsg->connOpen.connInterval < pAppUpdateCfg->connIntervalMin) ||
       (pMsg->connOpen.connInterval > pAppUpdateCfg->connIntervalMax) ||
       (pMsg->connOpen.connLatency != pAppUpdateCfg->connLatency) ||
       (pMsg->connOpen.supTimeout != pAppUpdateCfg->supTimeout)))
  {
    pCb->connWasIdle = FALSE;
    pCb->attempts = 0;
    APP_TRACE_INFO0("%%%% appSlaveConnOpen - appConnUpdateTimerStart%%%%");
    appConnUpdateTimerStart(pCb->connId);

  }
}
```

Ilustración 228: appSlaveProcConnOpen() en app_Slave.c

6.3.3 *ATT* MTU

Con el fin de establecer un determinado valor de MTU en los dispositivos *Central/Peripheral*, se configurará el mismo valor máximo en ambos, correspondiente a una MTU de 251 bytes, ya que, durante el emparejamiento, el intercambio del valor MTU determinará el tamaño máximo soportado, que se corresponderá con el menor de los MTU definidos por ambos dispositivos. La Ilustración 230 y la Ilustración 229 muestran dónde se configura la MTU en el protocolo *ATT*.

```
/*! \brief ATT run-time configurable parameters */
typedef struct
{
  wsfTimerTicks_t   discIdleTimeout;  /*!< \brief ATT server service discovery connection idle timeout in seconds */
  uint16_t          mtu;              /*!< \brief desired ATT MTU */
  uint8_t           transTimeout;     /*!< \brief transcation timeout in seconds */
  uint8_t           numPrepWrites;    /*!< \brief number of queued prepare writes supported by server */
} attCfg_t;

/*!
 * \brief ATT callback event
 *
 * \param hdr.event     Callback event
 * \param hdr.param     DM connection ID
 * \param hdr.status    Event status:  ATT_SUCCESS or error status
 * \param pValue        Pointer to value data, valid if valueLen > 0
 * \param valueLen      Length of value data
 * \param handle        Attribute handle
 * \param continuing    TRUE if more response packets expected
 * \param mtu           Negotiated MTU value
 */
typedef struct
{
  wsfMsgHdr_t           hdr;          /*!< \brief Header structure */
  uint8_t               *pValue;      /*!< \brief Value */
  uint16_t              valueLen;     /*!< \brief Value length */
  uint16_t              handle;       /*!< \brief Attribute handle */
  bool_t                continuing;   /*!< \brief TRUE if more response packets expected */
  uint16_t              mtu;          /*!< \brief Negotiated MTU value */
} attEvt_t;
```

Ilustración 229: ATT_api.h Estructura MTU.

```
/** \name ATT PDU Format
 * ATT PDU defaults and constants
 */
/**@{*/
#define ATT_HDR_LEN                   1         /*!< \brief Attribute PDU header length */
#define ATT_AUTH_SIG_LEN              12        /*!< \brief Authentication signature length */
//FBTG
//#define ATT_DEFAULT_MTU               23        /*!< \brief Default value of ATT_MTU */
#define ATT_DEFAULT_MTU               251       /*!< \brief Default value of ATT_MTU */
#define ATT_MAX_MTU                   517       /*!< \brief Maximum value of ATT_MTU */
//FBTG
//#define ATT_DEFAULT_PAYLOAD_LEN       20        /*!< \brief Default maximum payload length for most PDUs */
#define ATT_DEFAULT_PAYLOAD_LEN       251       /*!< \brief Default maximum payload length for most PDUs */
/**@}*/

/** \name ATT Maximum Value Parameters
 * maximum values for ATT attribute length and offset
 */
/**@{*/
#define ATT_VALUE_MAX_LEN             512       /*!< \brief Maximum attribute value length */
#define ATT_VALUE_MAX_OFFSET          511       /*!< \brief Maximum attribute value offset */
/**@}*/
```

Ilustración 230: ATT_defs.h Formato PDU.

En la Capa de Enlace (*Link Layer)* se definen, como se puede observar en Ilustración 231, los parámetros indicados.

```
#define LL_MAX_DATA_LEN_MIN           27      /*!< Minimum value for maximum Data PDU length */
#define LL_MAX_DATA_LEN_ABS_MAX       251     /*!< Absolute maximum limit for maximum Data PDU length */

#define LL_MAX_DATA_TIME_MIN          328     /*!< Minimum value for maximum Data PDU time */
#define LL_MAX_DATA_TIME_ABS_MAX      17040   /*!< Absolute maximum limit for maximum Data PDU time */
#define LL_MAX_DATA_TIME_ABS_MAX_1M   2120    /*!< Absolute maximum limit for maximum Data PDU time (LE 1M PHY) */
```

Ilustración 231: ll_defs.h MTU.

Como se mencionó anteriormente, en los dispositivos *Central* y *Peripheral* se configurará el mismo valor de MTU. La Ilustración 232 muestra cómo se definió el parámetro MTU desde los parámetros *ATT* en el presente TFG.

```
/*! ATT configurable parameters (increase MTU) */
static const attCfg_t datsAttCfg =
{
  15,                               /* ATT server service discovery connection idle timeout in seconds */
  247,                              /* desired ATT MTU */
//  241,                              /* desired ATT MTU */
  ATT_MAX_TRANS_TIMEOUT,            /* transcation timeout in seconds */
  5                                 /* number of queued prepare writes supported by server */
};
```

Ilustración 232: MTU parámetros configurables Dispositivo Central.

Adicionalmente, puesto que la pila no permite enviar paquetes a la interfaz *HCI*, a menos que sea posible la recepción de paquetes del mismo tamaño, es necesario configurar el *HCI* del dispositivo receptor para ser capaz de gestionar un paquete del tamaño configurado, más 4 bytes de *overhead ATT*, es decir, 255 bytes.

Analizando las funciones implementadas por librería *Cordio BLE*, se identificó que la función necesaria para este propósito era *HCIsetMaxRxAclLen*, que se incluyó en las funciones de inicialización de ambos dispositivos. La Ilustración 233, la Ilustración 234 y la Ilustración 235 muestran el lugar donde se configura, llama e inicializa la pila, respectivamente. Aunque solo se muestre el caso del dispositivo *Central,* se realiza este proceso en ambos dispositivos.

```
/*************************************************************************************************/
/*!
 *  \fn     HciSetMaxRxAclLen
 *
 *  \brief  Set the maximum reassembled RX ACL packet length.  Minimum value is 27.
 *
 *  \param  len     ACL packet length.
 *
 *  \return None.
 */
/*************************************************************************************************/
void HciSetMaxRxAclLen(uint16_t len)
{
  hciCoreCb.maxRxAclLen = len;
}
```

Ilustración 233: HCISetMaxRxAcllen() en HCI_core.c

```
/*************************************************************************************************/
*!
*  \brief  Connection callback for ATTC.
*
*  \param  pCcb    ATT control block.
*  \param  pDmEvt  DM callback event.
*
*  \return None.
*/
/*************************************************************************************************/
tatic void attcConnCback(attCcb_t *pCcb, dmEvt_t *pDmEvt)

 attcCcb_t *pClient;
 uint16_t  localMtu;
 uint8_t   status;

 /* if connection opened */
 if (pDmEvt->hdr.event == DM_CONN_OPEN_IND)
 {
   /* if we initiated connection send MTU request */
   if (DmConnRole(pCcb->connId) == DM_ROLE_MASTER)
   {
     localMtu = WSF_MIN(pAttCfg->mtu, (HciGetMaxRxAclLen() - L2C_HDR_LEN));

         APP_TRACE_INFO2("ATTC_MAIN: attcConnCback pAttCfg->mtu = %d, (HciGetMaxRxAclLen() - L2C_HDR_LEN) = %d", pAttCfg->mtu, (HciGetMaxRxAclLen() - L2C_HDR_LEN));
         APP_TRACE_INFO1("ATTC_MAIN: attcConnCback localMtu = %d", localMtu);

     /* if desired MTU is not the default */
     if (localMtu != ATT_DEFAULT_MTU)
     {
       AttcMtuReq(pCcb->connId, localMtu);
     }
```

Ilustración 234: Callback Central en ATTC_main.c

```
#if WDXS_INCLUDED == TRUE
    HciSetMaxRxAclLen(255);
#endif
```

Ilustración 235: exactle_stack_init() en Radio_task.c.

6.3.4 DLE

En cuanto la característica DLE, se habilita en ambos dispositivos, además de establecer el mismo tamaño del paquete de datos PDU en el proceso de inicialización, correspondiente al valor 251 (244 bytes de datos en cada paquete BLE), a través de los parámetros *maxTXOctets/maxRxOtets*, además de establecer el valor de los parámetros *maxTxTime/mxRxTime* al valor de 2120 µs (tiempo necesario para transmitir un paquete según los cálculos realizados). Se definen en la *HCI* los siguientes valores para estos parámetros (Ilustración 236, Ilustración 237 y Ilustración 238).

```
#define HCI_CMD_HDR_LEN                     3       /*!< \brief Command packet header length */
#define HCI_ACL_HDR_LEN                     4       /*!< \brief ACL packet header length */
#define HCI_ISO_HDR_LEN                     4       /*!< \brief ISO packet header length */
#define HCI_EVT_HDR_LEN                     2       /*!< \brief Event packet header length */
#define HCI_EVT_PARAM_MAX_LEN               255     /*!< \brief Maximum length of event packet parameters */
//FBTG
//#define HCI_ACL_DEFAULT_LEN                 27      /*!< \brief Default maximum ACL packet length */
#define HCI_ACL_DEFAULT_LEN                 255     /*!< \brief Default maximum ACL packet length */
```

Ilustración 236:HCI_defs.h tamaño ACL.

```
/* Default maximum ACL packet size for reassembly */
#ifndef HCI_MAX_RX_ACL_LEN
#define HCI_MAX_RX_ACL_LEN        HCI_ACL_DEFAULT_LEN
#endif
```

Ilustración 237: HCI_core.c tamaño ACL.

```
#define HCI_LE_SUP_FEAT_DATA_LEN_EXT                0x00000020
#define HCI_LE_SUP_FEAT_PRIVACY                     0x00000040
#define HCI_LE_SUP_FEAT_EXT_SCAN_FILT_POLICY        0x00000080
```

Ilustración 238: DLE en HCI_defs.h.

En este caso, se identifica la constante asociada que habilitará la característica DLE, tanto en el dispositivo *Central* como en el dispositivo *Peripheral.* A continuación, en la Ilustración 238, se muestra cómo en la configuración de la máscara que determina las características LE en la pila *Cordio BLE Stack*, se incluye por defecto la característica DLE.

```
/* LE supported features configuration mask */
uint32_t hciLeSupFeatCfg =
  HCI_LE_SUP_FEAT_ENCRYPTION                |    /* LE Encryption */
  HCI_LE_SUP_FEAT_CONN_PARAM_REQ_PROC       |    /* Connection Parameters Request Procedure */
  HCI_LE_SUP_FEAT_EXT_REJECT_IND            |    /* Extended Reject Indication */
  HCI_LE_SUP_FEAT_SLV_INIT_FEAT_EXCH        |    /* Slave-initiated Features Exchange */
  HCI_LE_SUP_FEAT_LE_PING                   |    /* LE Ping */
  HCI_LE_SUP_FEAT_DATA_LEN_EXT              |    /* LE Data Packet Length Extension */
  HCI_LE_SUP_FEAT_PRIVACY                   |    /* LL Privacy */
  HCI_LE_SUP_FEAT_EXT_SCAN_FILT_POLICY      |    /* Extended Scanner Filter Policies */
  HCI_LE_SUP_FEAT_LE_2M_PHY                 |    /* LE 2M PHY supported */
  HCI_LE_SUP_FEAT_STABLE_MOD_IDX_TRANSMITTER |   /* Stable Modulation Index - Transmitter supported */
  HCI_LE_SUP_FEAT_STABLE_MOD_IDX_RECEIVER   |    /* Stable Modulation Index - Receiver supported */
  HCI_LE_SUP_FEAT_LE_EXT_ADV                |    /* LE Extended Advertising */
  HCI_LE_SUP_FEAT_LE_PER_ADV;                    /* LE Periodic Advertising */
```

Ilustración 239:Configuración de la máscara en HCI_core.c

A continuación, en la Ilustración 240 se muestra cómo en la función *HCICoreResetSequence(),* que implementa la secuencia de acciones a ejecutar en la inicialización de la capa *HCI*, se incluye el comando *HCICoreReadMaxDataLen(),* que permite obtener los valores establecidos en el Controlador para *maxTxoctets y maxTxTime.*

```
case HCI_OPCODE_LE_READ_MAX_DATA_LEN:
  {
    uint16_t maxTxOctets;
    uint16_t maxTxTime;

    BSTREAM_TO_UINT16(maxTxOctets, pMsg);
    BSTREAM_TO_UINT16(maxTxTime, pMsg);
```

Ilustración 240: HCICoreResetSequence() en haci_vs.c.

Tras ejecutar este comando, se opera el *opcode HCI_OPCODE_LE_READ_MAX_DATA_LEN,* en el que se asignan los valores obtenidos en las variables *maxTxOctets/maxTxtime*. Por último, se llama a la función *HCILeWriteDefDataLen* para almacenar los valores obtenidos del Controlador para *maxTxOctets/maxTxTime* (Ilustración 241).

```
/*************************************************************************************************/
/*!
 *  \fn     HciLeWriteDefDataLen
 *
 *  \brief  HCI LE write suggested default data len command.
 *
 *  \return None.
 */
/*************************************************************************************************/
void HciLeWriteDefDataLen(uint16_t suggestedMaxTxOctets, uint16_t suggestedMaxTxTime)
{
  uint8_t *pBuf;
  uint8_t *p;

  APP_TRACE_INFO1("() HCI_CMD: HciLeWriteDefDataLen - suggestedMaxTxOctets = %d", suggestedMaxTxOctets);


  if ((pBuf = hciCmdAlloc(HCI_OPCODE_LE_WRITE_DEF_DATA_LEN, HCI_LEN_LE_WRITE_DEF_DATA_LEN)) != NULL)
  {
    p = pBuf + HCI_CMD_HDR_LEN;
    UINT16_TO_BSTREAM(p, suggestedMaxTxOctets);
    UINT16_TO_BSTREAM(p, suggestedMaxTxTime);
    hciCmdSend(pBuf);
  }
}
```

Ilustración 241: HCILewriteDetfDataLen()

La ejecución de la función *HCICoreReadMaxDataLen* (Ilustración 242) conlleva que la función *HCIEvtParseReadMaxDatalencmdCmp1()* realice un *parse* de los eventos de respuesta obtenidos desde el Controlador, con el fin de obtener los valores establecidos (Ilustración 243).

```
/*************************************************************************************************/
/*!
 *  \fn     HciLeReadMaxDataLen
 *
 *  \brief  HCI LE read maximum data len command.
 *
 *  \return None.
 */
/*************************************************************************************************/
void HciLeReadMaxDataLen(void)
{
  uint8_t *pBuf;

    APP_TRACE_INFO0("%%% HCI_CMD - HciLeReadMaxDataLen");

  if ((pBuf = hciCmdAlloc(HCI_OPCODE_LE_READ_MAX_DATA_LEN, HCI_LEN_LE_READ_MAX_DATA_LEN)) != NULL)
  {
    hciCmdSend(pBuf);
  }
}
```

Ilustración 242: HCILeReadMaxDataLen() en HCI_cmd.c

```
/*************************************************************************************************/
/*!
 *  \fn     hciEvtParseReadMaxDataLenCmdCmpl
 *
 *  \brief  Parse an HCI event.
 *
 *  \param  pMsg    Pointer to output event message structure.
 *  \param  p       Pointer to input HCI event parameter byte stream.
 *  \param  len     Parameter byte stream length.
 *
 *  \return None.
 */
/*************************************************************************************************/
static void hciEvtParseReadMaxDataLenCmdCmpl(hciEvt_t *pMsg, uint8_t *p, uint8_t len)
{
        APP_TRACE_INFO0("HCI_EVT: hciEvtParseReadMaxDataLenCmdCmpl");

  BSTREAM_TO_UINT8(pMsg->leReadMaxDataLenCmdCmpl.status, p);
  BSTREAM_TO_UINT16(pMsg->leReadMaxDataLenCmdCmpl.supportedMaxTxOctets, p);
  BSTREAM_TO_UINT16(pMsg->leReadMaxDataLenCmdCmpl.supportedMaxTxTime, p);
  BSTREAM_TO_UINT16(pMsg->leReadMaxDataLenCmdCmpl.supportedMaxRxOctets, p);
  BSTREAM_TO_UINT16(pMsg->leReadMaxDataLenCmdCmpl.supportedMaxRxTime, p);

        APP_TRACE_INFO2("HCI_EVT: hciEvtParseReadMaxDataLenCmdCmpl - supportedMaxTxOctets = %d, supportedMaxRxOctet
        APP_TRACE_INFO2("HCI_EVT: hciEvtParseReadMaxDataLenCmdCmpl - supportedMaxTxTime = %d, supportedMaxRxTime = 

  pMsg->hdr.status = pMsg->leReadMaxDataLenCmdCmpl.status;
}
```

Ilustración 243: HCIEvtParseReadMaxDataLenCmdCmpl() en HCI_evt.c.

6.4 Pruebas experimentales iniciales (*nosetDatalength*)

Una vez configurados todos los parámetros en la implementación de los dispositivos *Central/Peripheral* realizada en el presente TFG a partir de la librería *Cordio BLE,* se empezaron a realizar pruebas orientadas fundamentalmente a la medida del *Throughput* obtenido para una serie de casos con *WDXS_ENABLED y CS5C_ENABLED* iguales a *TRUE* (*Stream* y *1M LE).*

CASO 1:

Así, el primer caso que se consideró se basaba en el uso de los parámetros de conexión mostado en la Ilustración 244 en el dispositivo *Central*.

```
/*! Connection parameters */
static const hciConnSpec_t datcConnCfg =
{
  100,                                  /*! Minimum connection interval in 1.25ms units */ //- mínimo = 6
  100,                                  /*! Maximum connection interval in 1.25ms units */
  0,                                   /*! Connection latency */
  600,                                 /*! Supervision timeout in 10ms units */
  0,                                   /*! Unused */ //FBTG - minCElen
//  0                                     /*! Unused */ //FBTG - maxCElen
  65535                                   /*! Unused */ //FBTG - maxCElen
};
```

Ilustración 244: Parámetro de conexión para el 1º Caso.

A partir del menú que se ejecuta en el dispositivo *Central* a través de la interfaz serie, se estableció la conexión con el dispositivo *Peripheral*, y tras descubrir los ficheros WDXS desde el dispositivo *Central*, puesto que evidentemente en este caso *WDXC/WDXS_INCLUDED=TRUE*, se inició un *Stream*ing WDXC/WDXS desde el dispositivo *Peripheral*, obteniéndose los valores de *Thoughput* que se muestran en la Ilustración 245, alrededor de 8kbps, lo que representa un valor anormalmente reducido para los parámetros de configuración establecidos en la implementación de los dispositivos *Central/Peripheral*.

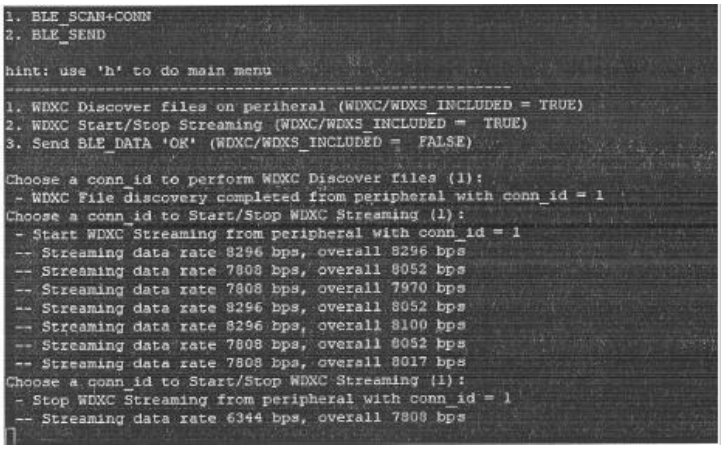


Ilustración 245: Putty medida Througthput Caso 1, inicialmente.

Este cálculo se realiza a partir de la función creada en *datc_main* siguiendo la siguiente secuencia:

1) Se establece cada cuánto se realiza el cálculo del *Throughput* mediante *timers,* como se muestra en la *Ilustración 246*. Se define cada 4 segundos.

```
/*! Data rate timer period in seconds */
#define DATC_WDXS_DATA_RATE_TIMEOUT       4
```

Ilustración 246: Establecimiento de Timeout para cálculo de Throughput.

2) Se crea un *timer* en la aplicación con el periodo de tiempo establecido anteriormente (*Ilustración 247*).

```
WsfTimerStartSec(&datcCb.dataRateTimer[connId-1], DATC_WDXS_DATA_RATE_TIMEOUT);
```

Ilustración 247: Creación del Timer con el Timeout establecido.

3) Por último, se implementa la función que calcula, a partir de los datos recibidos cada 4 segundos en la aplicación, de forma que, cuando estos 4 segundos se cumplen, se realiza el cálculo del *Throughput* parcial. Los datos en bits almacenados durante este periodo de tiempo, se pueden observar en la *Ilustración 248*.

```
static void datcWdxcProcessDataRateTimeout(dmConnId_t connId)
{
  uint32_t bps;

        APP_TRACE_INFO0("*** datcWdxcProcessDataRateTimeout.");

  /* Suppress warning when APP_TRACE disabled */
  (void) bps;

  /* Calculate data rate */
  bps = (datcCb.rxCount[connId-1] - datcCb.lastCount[connId-1]) * 8 / DATC_WDXS_DATA_RATE_TIMEOUT;
  datcCb.lastCount[connId-1] = datcCb.rxCount[connId-1];

  datcCb.streamSeconds[connId-1] += DATC_WDXS_DATA_RATE_TIMEOUT;
  datcCb.streamRate[connId-1] = datcCb.rxCount[connId-1] * 8 / datcCb.streamSeconds[connId-1];

  APP_TRACE_INFO2("Streaming data rate %d bps, overall %d bps", bps, datcCb.streamRate[connId-1]);
  am_menu_printf(" -- Streaming data rate %d bps, overall %d bps\r\n", bps, datcCb.streamRate[connId-1]);


  /* Restart the timer */
  datcCb.dataRateTimer[connId-1].msg.event = DATC_WDXS_DATA_RATE_TIMER_IND;
  datcCb.dataRateTimer[connId-1].msg.param = connId;
  datcCb.dataRateTimer[connId-1].handlerId = datcCb.handlerId;

  WsfTimerStartSec(&datcCb.dataRateTimer[connId-1], DATC_WDXS_DATA_RATE_TIMEOUT);
}
```

Ilustración 248: ProcessDataRateTimeout función para el cálculo del Throughput desde la aplicación.

Con el fin de poder determinar la causa de esta reducción inusual de *Thoughput,* se decidió hacer uso del *Snnifer NRF52840* de la empresa *Nordic Semiconductor* [26], junto con la herramienta *NRF Sniffer* [27] de la mima empresa, y la aplicación de escritorio *WireShark* [27] para *Windows*.

A partir del análisis de los paquetes transmitidos entre los dispositivos *Central/Peripheral* con la opción *WDXC/WDXS_ENABLED = TRUE,* una vez iniciado el *Streaming* de datos, se pudo determinar que, si bien el MTU negociado entre ambos dispositivos durante el proceso de conexión se correspondía con el establecido, al igual que los parámetros de conexión, el tamaño de los paquetes transmitidos a través del aire entre ambos dispositivos no se ajustaba al valor establecido (255 bytes), sino que este *payload* se envía en múltiples paquetes BLE (Ilustración 249).

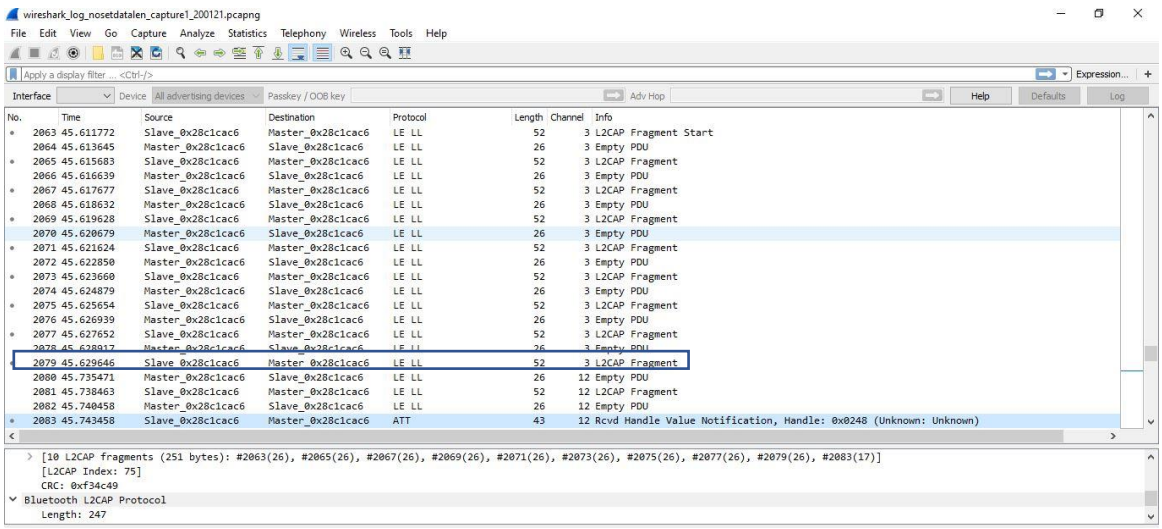


Ilustración 249: Wireshark nosetdatalength

A partir del análisis de estos paquetes, se observa cómo por ejemplo en el paquete 2083 se realiza el proceso de reemsablado del paquete enviado desde el dispositivo *Central* (de tamaño 251 bytes), a partir

de los datos reubicados en los paquetes 2063, 2065, 2067, 2069, 2071,2073. En consecuencia, el tamaño de los paquetes de datos PDU transmitidos a través del aire, si bien se corresponde con valores que evidencian que la característica DLE está habilitada (Ilustración 250), no se ajustan con los valores establecidos en las variables *MaxTxOctets* en el proceso de inicialización de la capa *HCI*.

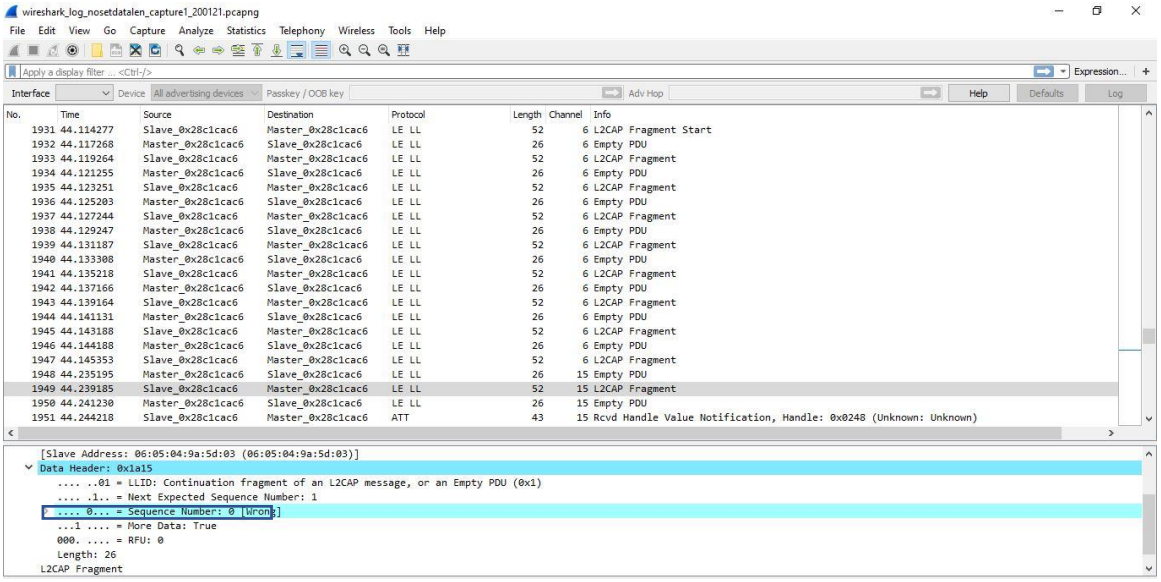


Ilustración 250: Wireshark Wrong sequence number.

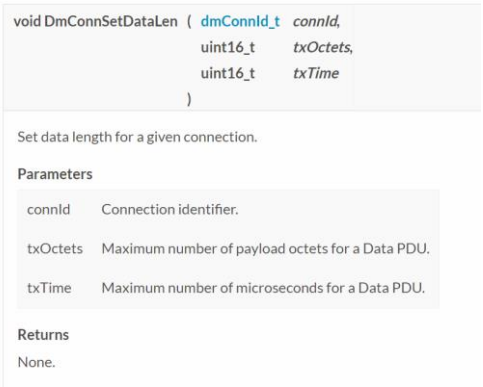
6.5 Solución al problema del tamaño del PDU (*DMconnSetdatalen*)

Finalmente se consideró que este problema podría solucionarse en caso de que existiera una función que permitiera establecer explícitamente el valor de la longitud del PDU, una vez establecida la conexión entre los dispositivos *Peripheral/Central*.

Tras realizar un análisis de las funciones proporcionadas en la librería BLE *Cordio*, se identificó en el módulo DM (*GAP Device Manager*), en el que se implementan los procedimientos requeridos por la pila para la gestión de los dispositivos (divididos por categoría y rol del dispositivo *Master/Slave*), entre ellos, los relacionados con la creación/aceptación/eliminación de conexión y actualización de los parámetros [12].

De este análisis se obtuvo que en la librería BLE *Cordio Stack* se incluye la función *DmConnSetDataLen(),* implementada en el fichero *dm_conn.c* ubicado en el directorio *third-party /exactle /ble_Host /sources/stack/dm*.

En la Ilustración 251 y la Ilustración 252 se muestra la función *DmConnSetDataLen.* Esta función permite establecer la longitud máxima de datos transmitidos por el aire para una determinada conexión, es decir, el mínimo o máximo número de bytes que limitan el *Data PDU*, así como el máximo tiempo en µs para transmitir un *Data PDU*.



void DmConnSetDataLen (dmConnId_t *connId*,
uint16_t *txOctets*,
uint16_t *txTime*
)

Set data length for a given connection.

Parameters

connId	Connection identifier.
txOctets	Maximum number of payload octets for a Data PDU.
txTime	Maximum number of microseconds for a Data PDU.

Returns

None.

Ilustración 251: DmConnSetDataLen parámetros.

```
void DmConnSetDataLen(dmConnId_t connId, uint16_t txOctets, uint16_t txTime)
{
  dmConnApiSetDataLen_t *pMsg;

        APP_TRACE_INFO0("%%%% DM_CONN: DmConnSetDataLen %%%%");


  if ((pMsg = WsfMsgAlloc(sizeof(dmConnApiSetDataLen_t))) != NULL)
  {
    pMsg->hdr.event = DM_CONN_MSG_API_SET_DATA_LEN;
    pMsg->hdr.param = connId;
    pMsg->txOctets = txOctets;
    pMsg->txTime = txTime;

    WsfMsgSend(dmCb.handlerId, pMsg);
  }
}
```

Ilustración 252: DmConnSetDataLen().

Así, en el fichero *dats_main_i2c.c* del *subdirectorio third-party/excatle/ble-profiles/sources/apps/dats*, se introduce la llamada a la función *DmConnSetDataLen* en la función *datsProcMsg*, cuando se indique que la conexión entre los dispositivos *Central*/Peripheral esté establecida (*pMsg -> hdr.event = DM_CONN-OPEN_ID*) con el fin de establecer el valor del parámetro *maxData PDU* a 251, y el máximo tiempo para transmitir un *Data PDU* a 0x848 (2120 µs), tal y como se presenta en la Ilustración 253.

```
#if WDXS_INCLUDED == TRUE
      // PDU length, TX interval (0x148 ~ 0x848)
      //DmConnSetDataLen(1, 27, 0x148);

      DmConnSetDataLen(1, 251, 0x848);
#endif /* WDXS_INCLUDED */
```

Ilustración 253: datsProcMsg() del Central DmconnsetDatalen.

Por lo tanto, del análisis del código del módulo *DM* de la librería BLE *Cordio* se deduce que a partir de la llamada a la función *DmConnSetDatalen* se genera un mensaje WSF correspondiente al evento *DM_CONN_MSG_API_SET_DATA_LEN*, que es procesado por la función *dmConn2MsgHandler* (Ilustración 254), implementada en el fichero *dm_conn.c*, en la que se que crea una estructura de tipo

dmConnApiSetDataLen_t (Ilustración 255) a la que se asigna el valor de los parámetros definidos en la función *DmConnSetDataLen,* que se usan a su vez como parámetros en la llamada a la función *HCIeLeSetDataLen*.

```
void dmConn2MsgHandler(wsfMsgHdr_t *pMsg)
{
  dmConnCcb_t *pCcb;

        APP_TRACE_INFO0("%%%% DM_CONN: dmConn2MsgHandler %%%%");

  /* look up ccb from conn id */
  if ((pCcb = dmConnCcbById((dmConnId_t) pMsg->param)) != NULL)
  {
    dmConn2Msg_t *pConn2Msg = (dmConn2Msg_t *) pMsg;

    /* handle incoming message */
    switch (pMsg->event)
    {
      case DM_CONN_MSG_API_READ_RSSI:
        HciReadRssiCmd(pCcb->handle);
        break;

      case DM_CONN_MSG_API_REM_CONN_PARAM_REQ_REPLY:
        {
          APP_TRACE_INFO0("%%%% DM_CONN: dmConn2MsgHandler - DM_CONN_MSG_API_REM_CONN_PARAM_REQ_REPLY %%%%");

        hciConnSpec_t *pConnSpec = &pConn2Msg->apiRemConnParamReqReply.connSpec;

        HciLeRemoteConnParamReqReply(pCcb->handle, pConnSpec->connIntervalMin,
                                     pConnSpec->connIntervalMax, pConnSpec->connLatency,
                                     pConnSpec->supTimeout, pConnSpec->minCeLen,
                                     pConnSpec->maxCeLen);
        }
        break;

      case DM_CONN_MSG_API_REM_CONN_PARAM_REQ_NEG_REPLY:
        HciLeRemoteConnParamReqNegReply(pCcb->handle, pConn2Msg->apiRemConnParamReqNegReply.reason);
        break;

      case DM_CONN_MSG_API_SET_DATA_LEN:
        {
          APP_TRACE_INFO0("%%%% DM_CONN: dmConn2MsgHandler - DM_CONN_MSG_API_SET_DATA_LEN %%%%");

        dmConnApiSetDataLen_t *pDataLen = &pConn2Msg->apiSetDataLen;

        HciLeSetDataLen(pCcb->handle, pDataLen->txOctets, pDataLen->txTime);
        }
        break;
```

Ilustración 254: dmConn2MsgHandler.

```
typedef union
{
  wsfMsgHdr_t               hdr;
  dmConnApiOpen_t           apiOpen;
  dmConnApiClose_t          apiClose;
  dmConnApiUpdate_t         apiUpdate;
  dmConnL2cUpdateInd_t      l2cUpdateInd;
  dmConnL2cUpdateCnf_t      l2cUpdateCnf;
  hciLeConnCmplEvt_t        hciLeConnCmpl;
  hciDisconnectCmplEvt_t    hciDisconnectCmpl;
  hciLeConnUpdateCmplEvt_t  hciLeConnUpdateCmpl;
} dmConnMsg_t;
```

Ilustración 255: Estructura DM messages.

Por una parte, en la función *HCILeSetDataLen* (Ilustración 256) se crea el paquete *HCI_CMD* conteniendo en el *Stream* los parámetros que se desean establecer, y se envía este comando el dispositivo *Central* mediante la llamada a la función *HCICmdSend*.

```
void HciLeSetDataLen(uint16_t handle, uint16_t txOctets, uint16_t txTime)
{
  uint8_t *pBuf;
  uint8_t *p;

    APP_TRACE_INFO0("%%% HCI_CMD - HciLeSetDataLen");

  if ((pBuf = hciCmdAlloc(HCI_OPCODE_LE_SET_DATA_LEN, HCI_LEN_LE_SET_DATA_LEN)) != NULL)
  {
    p = pBuf + HCI_CMD_HDR_LEN;
    UINT16_TO_BSTREAM(p, handle);
    UINT16_TO_BSTREAM(p, txOctets);
    UINT16_TO_BSTREAM(p, txTime);
    hciCmdSend(pBuf);
  }
}
```

Ilustración 256: HCILeSetDataLen.

Por una parte, en el dispositivo *Central,* al recibirse el paquete *HCI_CMD* desde el dispositivo *Peripheral,* se genera un evento *HCI* gestionado por la función *HCIEvtParseDatalenChange,* en la que se obtiene del *Stream* el valor de los parámetros *mxTxOctects/maxTxTime/maxRxOctests/maxRxTime*, como se observa en la Ilustración 257.

```
static void hciEvtParseDataLenChange(hciEvt_t *pMsg, uint8_t *p, uint8_t len)
{

  APP_TRACE_INFO2("- HCI_EVT (BEFORE): hciEvtParseDataLenChange - leDataLenChang

  BSTREAM_TO_UINT16(pMsg->leDataLenChange.handle, p);
  BSTREAM_TO_UINT16(pMsg->leDataLenChange.maxTxOctets, p);
  BSTREAM_TO_UINT16(pMsg->leDataLenChange.maxTxTime, p);
  BSTREAM_TO_UINT16(pMsg->leDataLenChange.maxRxOctets, p);
  BSTREAM_TO_UINT16(pMsg->leDataLenChange.maxRxTime, p);

  APP_TRACE_INFO2("- HCI_EVT (AFTER): hciEvtParseDataLenChange - leDataLenChange

  pMsg->hdr.param = pMsg->leDataLenChange.handle;
}
```

Ilustración 257: HCIEvtParseDataLenChange().

Este evento es atendido en el módulo DM de la librería BLE *Cordio* por la función *dmConn2ActDatalenChange* (Ilustración 258), que asigna a una estructura de tipo *HCILeDataLenChangeEvt_t* (Ilustración 259) los parámetros convertidos por la función *HCIEvtParseDataLenChange* desde *el Stream* del comando *HCI* recibido.

```
static void dmConn2ActDataLenChange(dmConnCcb_t *pCcb, hciEvt_t *pEvent)
{
  hciLeDataLenChangeEvt_t evt;


  /* call callback */
  evt.hdr.event = DM_CONN_DATA_LEN_CHANGE_IND;
  evt.hdr.param = pCcb->connId;
  evt.hdr.status = HCI_SUCCESS;
  evt.handle = pCcb->handle;
  evt.maxTxOctets = pEvent->leDataLenChange.maxTxOctets;
  evt.maxTxTime = pEvent->leDataLenChange.maxTxTime;
  evt.maxRxOctets = pEvent->leDataLenChange.maxRxOctets;
  evt.maxRxTime = pEvent->leDataLenChange.maxRxTime;

  APP_TRACE_INFO2("- DM_CONN: dmConn2ActDataLenChange - leDataLenChange.m
  APP_TRACE_INFO2("- DM_CONN: dmConn2ActDataLenChange - leDataLenChange.m

  (*dmConnCb.connCback[DM_CLIENT_ID_APP])((dmEvt_t *) &evt);
}
```

Ilustración 258: dmConn2ActDataLenChange().

```
typedef struct
{
  wsfMsgHdr_t        hdr;         /*!< \brief Event header. */
  uint16_t           handle;      /*!< \brief Connection handle. */
  uint16_t           maxTxOctets; /*!< \brief Maximum Tx octets. */
  uint16_t           maxTxTime;   /*!< \brief Maximum Tx time. */
  uint16_t           maxRxOctets; /*!< \brief Maximum Rx octets. */
  uint16_t           maxRxTime;   /*!< \brief Maximum Rx time. */
} hciLeDataLenChangeEvt_t;
```

Ilustración 259:Estructura LE data length change event.

El procedimiento descrito puede comprobarse a partir de los mensajes incluidos mediante la sentencia *APP_TRACEX* en los ficheros de la librería BLE *Cordio* que se obtiene, tanto del dispositivo *Central* como del dispositivo *Peripheral*, a través de *SWO*, y capturas mediante la aplicación *SEGGER J-LINK SWO viewer*, una vez establecida la conexión entre ambos dispositivos. En la Ilustración 260 y la Ilustración 261 se observan los resultados obtenidos, tanto del dispositivo *Peripheral* como del dispositivo *Central,* respectivamente.

```
% DM_CONN: DmConnSetDataLen %
% WSF_MSG: WsfMsgSend handlerId:1 %
WDXS_STREAM: --- wdxsSetStreamWaveform
>>> Connection opened <<<

% DM_CONN: dmConn2MsgHandler %
% DM_CONN: dmConn2MsgHandler - DM_CONN_MSG_API_SET_DATA_LEN %
 HCI_CMD - HciLeSetDataLen
### HCI_DRV_Apollo3: HciDrvHandler
### HCI_DRV_Apollo3: HciDrvHandler - g_ui32NumBytes = 0, g_consumed_bytes = 0
### HCI_DRV_Apollo3: HciDrvHandler - ui32NumHciTransactions = 0, HCI_DRV_MAX_HCI_TRANSACTIONS = 10000
### HCI_DRV_Apollo3: HciDrvHandler - something to write - psWriteBuffer->ui32Length = 6
### HCI_DRV_Apollo3: HciDrvHandler - we managed to actually send a packet
### HCI_DRV_Apollo3: HciDrvIntService
WSF_OS: WsfSetEvent handlerId:7 event:1
WsfSetEvent handlerId:7 event:1
### HCI_DRV_Apollo3: HciDrvHandler
### HCI_DRV_Apollo3: HciDrvHandler - g_ui32NumBytes = 0, g_consumed_bytes = 0
### HCI_DRV_Apollo3: HciDrvHandler - ui32NumHciTransactions = 0, HCI_DRV_MAX_HCI_TRANSACTIONS = 10000
### HCI_TR: hciTrSerialRxIncoming: len = 7
WSF_OS: WsfSetEvent handlerId:0 event:1
WsfSetEvent handlerId:0 event:1
### HCI_TR: hciTrSendCmd: len = 9
### HCI_DRV_Apollo3: hciDrvWrite: len = 9
### HCI_DRV_Apollo3: hciDrvWrite - psWriteBuffer->ui32Length = 10
### HCI_DRV_Apollo3: hciDrvWrite: WsfSetEvent(g_HciDrvHandleID, BLE_TRANSFER_NEEDED_EVENT)
WSF_OS: WsfSetEvent handlerId:7 event:1
WsfSetEvent handlerId:7 event:1
### HCI_DRV_Apollo3: HciDrvHandler
### HCI_DRV_Apollo3: HciDrvHandler - g_ui32NumBytes = 0, g_consumed_bytes = 0
### HCI_DRV_Apollo3: HciDrvHandler - ui32NumHciTransactions = 0, HCI_DRV_MAX_HCI_TRANSACTIONS = 10000
### HCI_DRV_Apollo3: HciDrvHandler - something to write - psWriteBuffer->ui32Length = 10
### HCI_DRV_Apollo3: HciDrvIntService
WSF_OS: WsfSetEvent handlerId:7 event:1
WsfSetEvent handlerId:7 event:1


### HCI_TR: hciTrSerialRxIncoming: len = 9
WSF_OS: WsfSetEvent handlerId:0 event:1
WsfSetEvent handlerId:0 event:1
HCI_EVT: hciEvtParseSetDa### HCI_DRV_Apollo3: HciDrvIntService
```

Ilustración 260: SWO SetDataLen Peripheral.

```
HCI_EVT (BEFORE): hciEvtParseDataLenChange - leDataLenChange.maxTxOctets = 0, leDataLenChange.maxRxOctets = 1
HCI_EVT (AFTER): hciEvtParseDataLenChange - leDataLenChange.maxTxOctets = 251, leDataLenChange.maxRxOctets = 251
DM_CONN: dmConn2ActDataLenChange - leDataLenChange.maxTxOctets = 251, leDataLenChange.maxRxOctets = 251
DM_CONN: dmConn2ActDataLenChange - leDataLenChange.maxTxTime = 2120, leDataLenChange.maxRxTime = 2120
WSF_MSG: WsfMsgSend handlerId:5 %
```

Ilustración 261: SWO SetDataLen Central.

6.5.1 Pruebas Experimentales con *DmConnSetDataLen()*

Una vez actualizado desde el dispositivo *Peripheral* el tamaño máximo del *Data PDU* mediante la incorporación de la función *DmConnSetDatalen* en el fichero *dats_main_I2C.c*, el siguiente paso lógico es, mediante el uso del *sniffer nRF52840*, verificar que el tamaño de los paquetes enviados desde el dispostivio *Peripheral* al dispositivo *Central* con la opción *WDXC_WPXS_INCLUDED = TRUE*, se ajustan ahora al valor establecido (251 bytes), y en consecuencia se envía en un único paquete cada Notificación generada al iniciar el *Stream* de datos, lo que debería conllevar un aumento considerable del *Throughput* obtenido.

CASO 1:

Con un MTU de 247 bytes, se obtiene que los valores de *Throughput*, según se observa en la Ilustración 262, están en torno a 50 kbps, representando un aumento del valor experimental obtenido

inicialmente para los mismos parámetros de configuración, pero habiendo establecido ahora el valor de PDU a 251 bytes desde el dispositivo *Peripheral*. Este valor corresponde a un *Throughput* resultante de enviar 3 o 4 paquetes por Intervalo de Conexión:

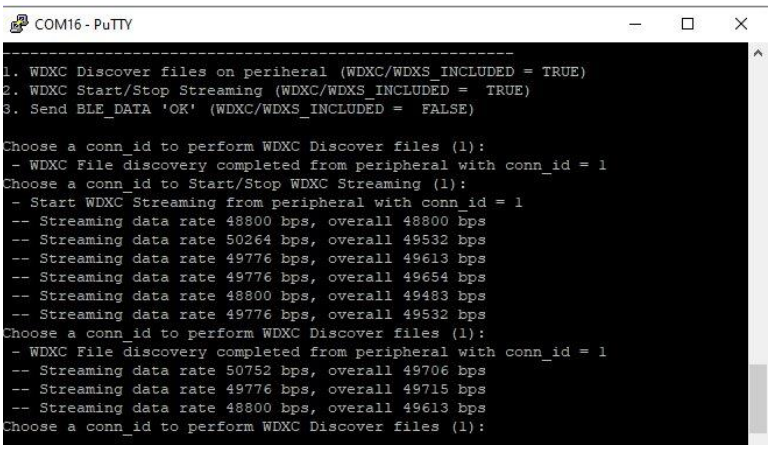


Ilustración 262: Putty Througthput caso 1 tras setDataLen.

- Para 3 paquetes:

$$Throughput = \frac{8\,(bits) x\ 3\,(paquetes\ por\ ic)\ x\ 244\,(bytes\ de\ payload)}{Ic\ (\ Intervalo\ de\ conexión)} \approx 46{,}85\ Kbps$$

- Para 4 paquetes:

$$Throughput = \frac{8\,(bits) x\ 4\,(paquetes\ por\ ic)\ x\ 244\,(bytes\ de\ payload)}{Ic\ (\ Intervalo\ de\ conexión)} \approx 62{,}46\ Kbps$$

A partir del análisis de los paquetes BLE capturador por el *Sniffer nRP52840* se pudo comprobar el envío de los paquetes de tipo *LL_LENGTH_REQ* intercambiados entre los dispositivos *Central*/*Peripheral* con el fin de establecer el valor del máximo *Data* PDU a 251 bytes, mediante la llamada a la función *DmConnSetDataLen()* en el dispositivo *Peripheral*. Por otro lado, se pudo verificar que, en este caso, efectivamente el número de paquetes transmitidos entre los dispositivos *Central/Peripheral* en cada Evento de Conexión se corresponde con el envío 3 a 4 paquetes BLE del tamaño especificado de 251 bytes, siendo el tamaño del paquete BLE asociado a cada Notificación de 277 bytes (Ilustración 263).

	Acces Addr	LLH	ATTCapH	ATTH	Payload	CRC	Sniffer	
Notificarion (bytes)	4	2	4	3	244	3	17	
								Total 277

Ilustración 263: Paquete Notificación BLE Caso 1.

Por otro lado, se observó que había numerosos paquetes BLE de Notificación que incluían un número de secuencia erróneo (*SEQUENCE NUMBER = WRONG* en *WireShark*) , como el ejemplo mostrado en la Ilustración 264, lo que evidentemente influye también en que el *Throughput* obtenido experimentalmente se vea reducido. En todo caso, con el fin de determinar los factores que afectaban al hecho de que únicamente se enviasen en el mayor de los casos 3- 4 paquetes BLE por cada dispositivo en un Evento de Conexión, aún cuando como este caso, el valor del Intervalo de Conexión configurado sea tan elevado como 125 ms, se decidió analizar en detalle el formato de los paquetes BLE enviados por el dispositivo *Peripheral*, una vez activada la opción de *WDXStream* desde el menú.

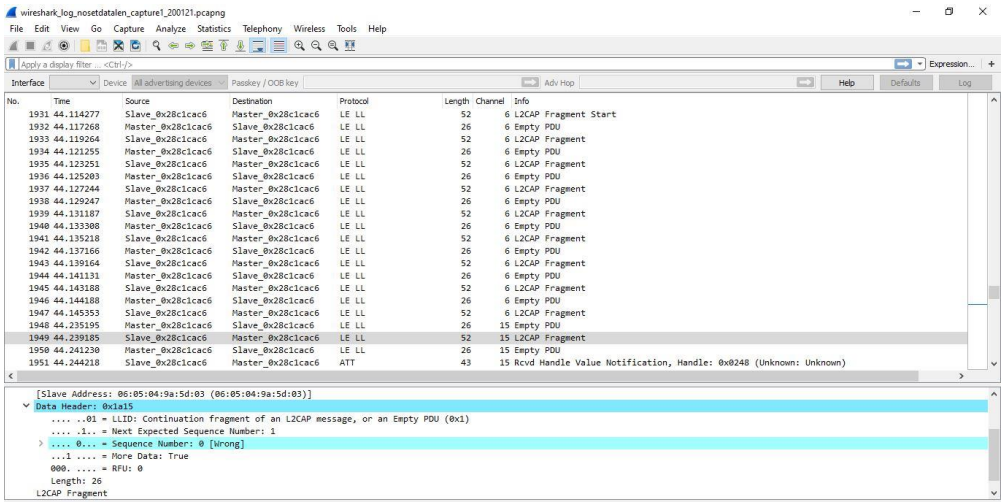


Ilustración 264: Wireshark Wrong Sequence Number.

A partir de este análisis, inicialmente se prestó atención al valor del bit MD (*More Data*) de la cabecera *Data Header* del PDU, que como se vio, se utiliza para indicar que un dispositivo dispone de más datos para enviar en un determinado Evento de Conexión. Así, si alguno o ambos dispositivos establecen a nivel alto el valor del bit MD, el dispositivo *Master* puede continuar el Evento de Conexión mediante el envío de otro paquete BLE, y el dispositivo *Peripheral* debe permanecer a la escucha tras enviar un paquete BLE.

En caso de que el dispositivo *Master* no reciba un paquete BLE del dispositivo *Peripheral,* el dispositivo *Master* cerrará el Evento de Conexión. En caso de que el dispositivo *Slave* no reciba un paquete BLE del dispositivo *Master,* el dispositivo *Peripheral* será el que cierre el Evento de Conexión. Así, el intercambio de dos paquetes con el bit MD a nivel bajo, implicará el cierre del Evento de Conexión.

A partir de esta situación, se observa cómo efectivamente, de la información obtenida de *Wireshark* (Ilustración 265) para dos paquetes BLE intercambiados entre los dispositivos *Central/Peripheral* en un Evento de Conexión, el valor del bit MD en el paquete indicado, enviado desde el dispositivo *Peripheral*, finaliza el Evento de Conexión tras la transferencia de 3 paquetes BLE de notificación.

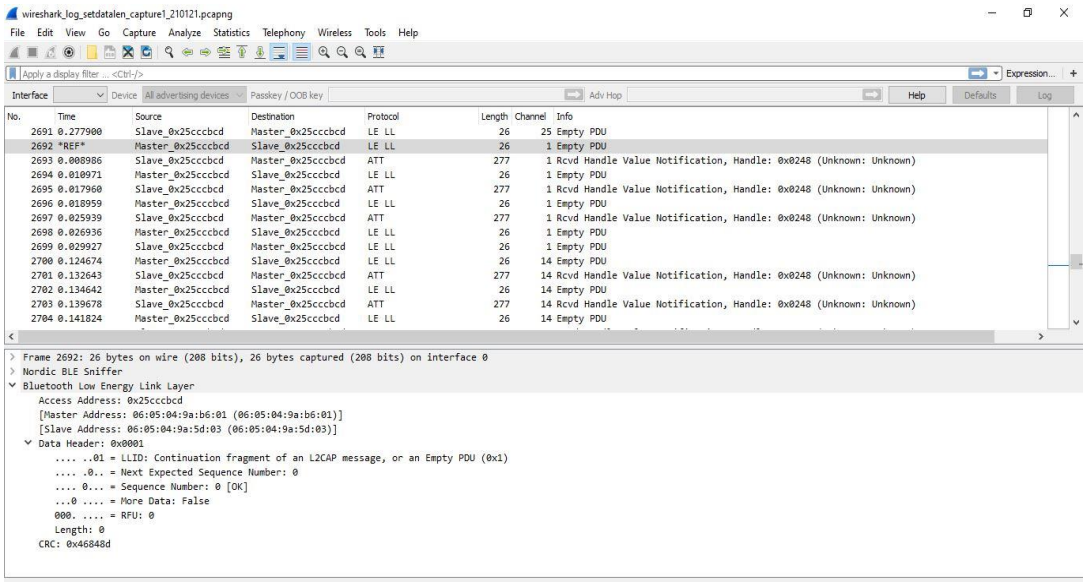


Ilustración 265: Wireshark More Data False.

A partir de esta constatación, se dedujo que este hecho podía estar relacionado con el número y tamaño de los elementos de almacenamiento dispuestos en el Controlador implementado en el MCU *Ambiq 3 Blue* integrado en la placa *Artemis Thing Plus*. Después de revisar la poca documentación existente sobre este aspecto en la distribución de la librería *Cordio BLE*, se pasó a analizar en detalle el código de los ficheros que implementaban la capa *HCI*. De este proceso se pudo determinar que el número y tamaño de *buffers* disponibles en el Controlador se podía obtener mediante la función *HCILeReadBufSizeCmd* (Ilustración 266), por lo que se incluyó la llamada a esta función en la secuencia de inicialización de la capa *HCI* (Ilustración 267), implementada en el fichero *HCI_vs.c*.

```
void HciLeReadBufSizeCmd(void)
{
  uint8_t *pBuf;

  if ((pBuf = hciCmdAlloc(HCI_OPCODE_LE_READ_BUF_SIZE, HCI_LEN_LE_READ_BUF_SIZE)) != NULL)
  {
    hciCmdSend(pBuf);
  }
}
```

Ilustración 266: HCILeReadBufSizeCmd.

```
  case HCI_OPCODE_READ_BD_ADDR:
    /* parse and store event parameters */
    BdaCpy(hciCoreCb.bdAddr, pMsg);

    /* send next command in sequence */
    HciLeReadBufSizeCmd();
APP_TRACE_INFO0("%%% HCI_VS - hciCoreResetSequence - HCI_OPCODE_READ_BD_ADDR");
    break;

  case HCI_OPCODE_LE_READ_BUF_SIZE:
    /* parse and store event parameters */
    BSTREAM_TO_UINT16(hciCoreCb.bufSize, pMsg);
    BSTREAM_TO_UINT8(hciCoreCb.numBufs, pMsg);

APP_TRACE_INFO1("%%% HCI_VS - hciCoreResetSequence - hciCoreCb.bufSize = %u", hciCoreCb.bufSize);
APP_TRACE_INFO1("%%% HCI_VS - hciCoreResetSequence - hciCoreCb.numBufs = %u", hciCoreCb.numBufs);

    /* initialize ACL buffer accounting */
    #if defined(AM_PART_APOLLO3) || defined(AM_PART_APOLLO3P)
    // B0 has less data buffer compared to A1 and A0
    if (!APOLLO3_GE_B0)
    {
      hciCoreCb.numBufs--;
    }
    #endif
    hciCoreCb.availBufs = hciCoreCb.numBufs;

    /* send next command in sequence */
    HciLeReadSupStatesCmd();
    break;
```

Ilustración 267: HCICoreResetSequence.

Los valores retornados por la llamada a esta función se almacenan en los campos *HCICore.bufsize* y *HCICorCb.numBufs*, que representan respectivamente el tamaño y número de elementos de almacenamiento disponibles en el Controlador. Por otro lado, el valor del parámetro *HCIcorecb.availBufs*, que indica en cada momento el número de *buffers* disponibles en el Controlador, se inicializa al valor de *HCICoreCb.numBufs*. En caso particular del presente TFG, los valores obtenidos son de 4 buffers de 251 bytes (Ilustración 268).

```
WSF_OS: WsfSetEvent handlerId:0 event:1
WsfSetEvent handlerId:0 event:1
 HCI_VS - hciCoreResetSequence
HCI_VS - hciCoreResetSequence - hciCoreCb.bufSize = 251
HCI_VS - hciCoreResetSequence - hciCoreCb.numBufs = 4
### HCI_IR: hciIrSendCmd: len = 3
### HCI_DRV_Apollo3: hciDrvWrite: len = 3
### HCI_DRV_Apollo3: hciDrvWrite - psWriteBuffer->ui32Length = 4
### HCI_DRV_Apollo3: hciDrvWrite: WsfSetEvent(g_HciDrvHandleID, BLE_TRANSFER_NEEDED_EVENT)
WSF_OS: WsfSetEvent handlerId:7 event:1
WsfSetEvent handlerId:7 event:1
### HCI_DRV_Apollo3: HciDrvHandler
```

Ilustración 268: SWO viewer número y tamaño Buffer HCI.

La pila *Cordio BLE* implementa un mecanismo de control de flujo HCI (*HCI Flow Control*), con el fin de prevenir la saturación de los elementos de almacenamiento del Controlador con más datos de los que puede gestionar. Este mecanismo de control de flujo opera de la siguiente manera:

Una llamada a una función *ATT* (por ejemplo*, ATTSHandleValueNtf*) da lugar a la copia de los datos de la aplicación en un *buffer* WSF (si está disponible) y la programación del paquete para la entrega a *L2CAP*. El mensaje es entonces gestionado por el proceso *L2CAP*, y pasado a la función *HCISendAclData()*, donde es transmitido a través de *HCI* si los *buffers* están disponibles, propagándose a continuación un estado de confirmación de vuelta a la aplicación para que ésta sepa que el paquete ha salido (a través de *HCI* y no a través del aire).

Para gestionar el control de flujo en el caso de que la aplicación/pila intente saturar al *HCI* con demasiados paquetes, se configura un mecanismo de control de flujo por conexión. El *flag pConn->flowDisabled* gestiona si el flujo de paquetes hacia el *HCI* ha sido deshabilitado o no.

El mecanismo de control de flujo se activa cuando todos los *buffers* de *HCI* están llenos. Esto ocurre cuando el número de paquete *s*en cola para una conexión excede el umbral configurable (*HCI_ACL_QUEUE_HI*). Cuando esto se produce, se habilita el indicador *flowDisabled* de las conexiones *HCI* (*HCICoreConn_t*) para evitar que se sigan almacenando datos en el *buffer*. El control de flujo se vuelve a activar después de que el *Host* haya procesado un número suficiente de eventos de paquetes completados de *HCI* (*HCICoreNumCmplPkts,* mostrada en Ilustración 271), de manera que la lista de paquetes salientes (en cola) descienda hasta el umbral bajo configurado (*HCI_ACL_QUEUE_LO*) o por debajo de éste. Los umbrales alto y bajo para el control de flujo se definen en */ble-Host/sources/HCI/common/HCI-core.c.*

El mecanismo de control de flujo *HCI* está conectado a través de *L2CAP* a los módulos *ATT* y *SMP* para activar y desactivar simultáneamente el control de flujo en la parte superior de la pila. Para el *Servidor ATT*, las *callback* están asosiadas a *ATTSIndCtrlCback* (nota: las *callback* no están conectadas para el *Cliente ATT*) cuando el control de flujo es reactivado. El valor de las unidades configurables que determinan el valor de *HCI_ACL_QUEUE_HI* y *HCI_ACL_QUEUE_LO*, se definen en el fichero *HCI_core.c ().*

```
/**************************************************************************************************
  Macros
**************************************************************************************************/

/* Default ACL buffer flow control watermark levels */
#ifndef HCI_ACL_QUEUE_HI
#define HCI_ACL_QUEUE_HI          5             /* Disable flow when this many buffers queued */
#endif
#ifndef HCI_ACL_QUEUE_LO
#if defined(AM_PART_APOLLO3) || defined(AM_PART_APOLLO3P)
#define HCI_ACL_QUEUE_LO          3             /* Enable flow when this many buffers queued */
#else
#define HCI_ACL_QUEUE_LO          1             /* Enable flow when this many buffers queued */
#endif
#endif
```

Ilustración 269: Definición HCI_ACL_QUEUE.

La activación de la parada del flujo de datos hacia el Controlador se muestra en la Ilustración 270.

```
void HciSendAclData(uint8_t *pData)
{
  uint16_t        handle;
  uint16_t        len;
  hciCoreConn_t   *pConn;

      APP_TRACE_INFO0("### HCI_CORE: HciSendAclData");

  /* parse handle and length */
  BYTES_TO_UINT16(handle, pData);
  BYTES_TO_UINT16(len, &pData[2]);
      APP_TRACE_INFO1("########################################## HCI_CORE: HciSendAclData - len = %u", len);

  /* look up connection structure */
  if ((pConn = hciCoreConnByHandle(handle)) != NULL)
  {
    /* if queue empty and buffers available */
    if (WsfQueueEmpty(&hciCoreCb.aclQueue) && hciCoreCb.availBufs > 0)
    {
      /* send data */
        APP_TRACE_INFO1("########################################## HCI_CORE: HciSendAclData - SEND DATA (availBufs = %d)", hciCoreCb.availBufs);

      hciCoreTxAclStart(pConn, len, pData);

    }
    else
    {
      /* queue data - message handler ID 'handerId' not used */
        APP_TRACE_INFO0("### HCI_CORE: HciSendAclData - queue data");
        APP_TRACE_INFO1("########################################## HCI_CORE: HciSendAclData - QUEUE DATA (availBufs = %d)", hciCoreCb.availBufs);

      WsfMsgEnq(&hciCoreCb.aclQueue, 0, pData);
    }

    /* increment buffer queue count for this connection with consideration for HCI fragmentation */
    pConn->queuedBufs += ((len - 1) / HciGetBufSize()) + 1;

    /* manage flow control to stack */
    if (pConn->queuedBufs >= hciCoreCb.aclQueueHi && pConn->flowDisabled == FALSE)
    {
      pConn->flowDisabled = TRUE;
      (*hciCb.flowCback)(handle, TRUE);

        APP_TRACE_INFO0("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! HCI_CORE: HciSendAclData - FLOW DISABLED = TRUE");

    }
  }
```

Ilustración 270: HCISendAclData.

La desactivación de la parada del flujo de datos se describe en la función *hciCoreNumCmplPkts*, cuyo código se representa en la Ilustración 271.

```
void hciCoreNumCmplPkts(uint8_t *pMsg)
{
  uint8_t         numHandles;
  uint16_t        bufs;
  uint16_t        handle;
  uint8_t         availBufs = 0;
  hciCoreConn_t   *pConn;

      APP_TRACE_INFO0("HCI_CORE_PS: hciCoreNumCmplPkts");

  /* parse number of handles */
  BSTREAM_TO_UINT8(numHandles, pMsg);

  /* for each handle in event */
  while (numHandles-- > 0)
  {
    /* parse handle and number of buffers */
    BSTREAM_TO_UINT16(handle, pMsg);
    BSTREAM_TO_UINT16(bufs, pMsg);

      APP_TRACE_INFO1("HCI_CORE_PS: hciCoreNumCmplPkts - bufs = %d", bufs);

    if ((pConn = hciCoreConnByHandle(handle)) != NULL)
    {
      /* decrement outstanding buffer count to controller */
      pConn->outBufs -= (uint8_t) bufs;

      /* decrement queued buffer count for this connection */
      pConn->queuedBufs -= (uint8_t) bufs;

      /* increment available buffer count */
      availBufs += (uint8_t) bufs;

      /* call flow control callback */
      if (pConn->flowDisabled && pConn->queuedBufs <= hciCoreCb.aclQueueLo)
      {
        APP_TRACE_INFO0("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! HCI_CORE_PS: HciSendAclData - FLOW DISABLED = FALSE");

        pConn->flowDisabled = FALSE;
        (*hciCb.flowCback)(handle, FALSE);
      }
    }
  }
```

Ilustración 271: HCICoreNumCmplPkts() en HCI_core_ps.c.

Sin embargo, en el caso particular de la plataforma *Artemis Thing Plus*, el chip que integra en el módulo *Ambiq Apollo 3*, y para este caso particular, en el código del fichero *HCI_core.c* de la implementación de la capa *HCI,* se establecen los umbrales *HCICoreCb.AclQueueHi y HCICoreCb.aclQueueLo*, a los valores (*HCI_ACL_QUEUE_HI-1)* y (*HCI_ACL_QUEUE_LO-1*), respectivamente, como se puede comprobar en la función *HCICoreInit().*

En consecuencia, en el caso que nos ocupa, con paquetes con un *payload* de 244 bytes, los umbrales establecidos en *HCIcoreCb.aclQueueHi* y *HCIcoreCb.aclQueueLo*, así como el valor del número de elementos de almacenamiento disponibles en el Controlador, y su tamaño, correspondiente a *HCICoreCb.numBuf y HCI.CoreCb.bufSize,* respectivamente, se pueden representar como se observa en el fichero *log* mostrado para el caso del dispositivo *Peripheral* implementado. En la Ilustración 272 se representan gráficamente los diferentes valores de los parámetros y el flujo seguido en este mecanismo.

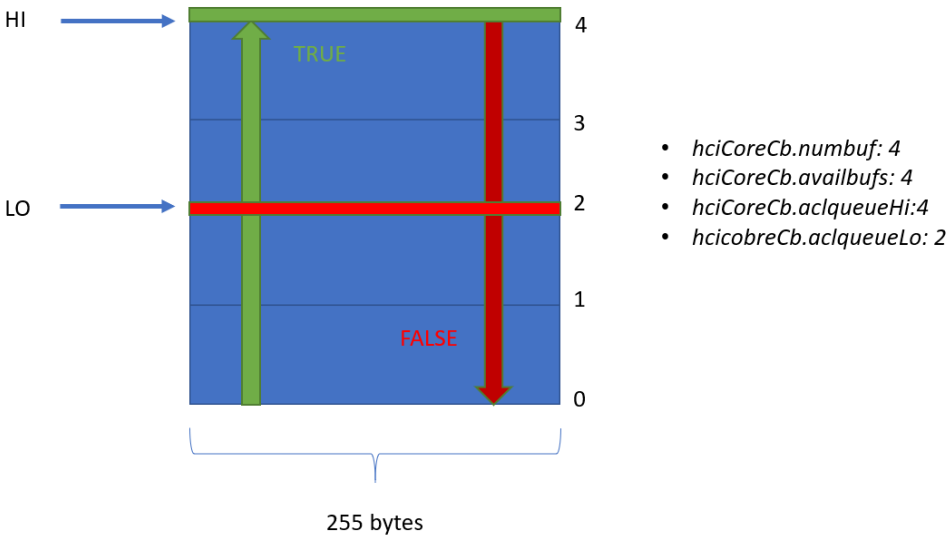


Ilustración 272: Representación Gráfica de HCICoreCb

Así, el flujo de envío de Notificaciones en un Evento de Conexión se verá interrumpido (*Flow Disabled = TRUE*) cuando el valor de *HCICoreCb.availBufs* sea 0, y será reestablecido, como se muestra en la Ilustración 273 (*flowDisabled=FALSE*), cuando el valor de *HCICoreCb.availbufs* sea mayor o igual a 2.

```
void HciCoreInit(void)
{
  uint8_t   i;

  APP_TRACE_INFO0("HCI_CORE: ESTOY EN AMBIQ/HciCoreInit");

  WSF_QUEUE_INIT(&hciCoreCb.aclQueue);

  for (i = 0; i < DM_CONN_MAX; i++)
  {
    hciCoreCb.conn[i].handle = HCI_HANDLE_NONE;
  }

  hciCoreCb.maxRxAclLen = HCI_MAX_RX_ACL_LEN;
  hciCoreCb.aclQueueHi = HCI_ACL_QUEUE_HI;
  hciCoreCb.aclQueueLo = HCI_ACL_QUEUE_LO;


  #if defined(AM_PART_APOLLO3) || defined(AM_PART_APOLLO3P)
    if (APOLLO3_GE_B0)
    {
      // B0 has only less internal ACL buffers
      hciCoreCb.aclQueueHi--;
      hciCoreCb.aclQueueLo--;
    }
  #endif

  APP_TRACE_INFO1("HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - HCI_ACL_QUEUE_HI = %d", HCI_ACL_QUEUE_HI);
  APP_TRACE_INFO1("HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - HCI_ACL_QUEUE_LO = %d", HCI_ACL_QUEUE_LO);
  APP_TRACE_INFO1("HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - hciCoreCb.aclQueueHi = %d", hciCoreCb.aclQueueHi);
  APP_TRACE_INFO1("HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - hciCoreCb.aclQueueLo = %d", hciCoreCb.aclQueueLo);


  hciCoreCb.extResetSeq = NULL;

  hciCoreInit();
}
```

Ilustración 273: HCICoreInit Apollo 3.

```
### HCI_DRV_Apollo3: HciDrvRadioBoot
Starting wicentric trace:

Creating pool len=16 num=8
              pStart=0x10003624
Creating pool len=32 num=4
              pStart=0x100036a4
Creating pool len=64 num=6
              pStart=0x10003724
Creating pool len=280 num=8
              pStart=0x100038a4
Creating pool len=8 num=0
              pStart=0x10004164
HCI_CORE: ESTOY EN AMBIQ/HciCoreInit
HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - HCI_ACL_QUEUE_HI = 5
HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - HCI_ACL_QUEUE_LO = 3
HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - hciCoreCb.aclQueueHi = 4
HCI_CORE: ESTOY EN AMBIQ/HciCoreInit - hciCoreCb.aclQueueLo = 2
```

Ilustración 274: SWO viewer AclQueue Peripheral.

En la Ilustración 274 y la Ilustración 275 se observa un extracto de los ficheros *logs* generados, tanto en el dispositivo *Central* como en el dispositivo *Peripheral,* durante una de las pruebas experimentales realizadas en este primer caso, observándose la recepción de 3 paquetes BLE en el dispositivo *Central* desde el dispositivo *Peripheral* en un determinado Evento de Conexión, así como el efecto del mecanismo de control de flujo implementado, con los umbrales establecidos, en la limitación del máximo número de paquetes BLE enviados desde el dispositivo *Peripheral* durante la mayor parte de los Eventos de Conexión.

```
*** datcWdxcFtdCallback - connId = 1, fileHdl = 1, len = 244
*** datcWdxcFtdCallback - pData (first byte) = 1 (1)
--- process ATT messages
*** datcProcMsg - ATTC_HANDLE_VALUE_NTF
WdxcProcMsg - ATTC_HANDLE_VALUE_NTF
WDXC: wdxcWdxsValueUpdate.
WDXC file transfer data.
WDXC: wdxcParseFtd.
*** datcWdxcFtdCallback - connId = 1, fileHdl = 1, len = 244
*** datcWdxcFtdCallback - pData (first byte) = 2 (2)
--- process ATT messages
*** datcProcMsg - ATTC_HANDLE_VALUE_NTF
WdxcProcMsg - ATTC_HANDLE_VALUE_NTF
WDXC: wdxcWdxsValueUpdate.
WDXC file transfer data.
WDXC: wdxcParseFtd.
*** datcWdxcFtdCallback - connId = 1, fileHdl = 1, len = 244
*** datcWdxcFtdCallback - pData (first byte) = 4 (4)
--- process ATT messages
*** datcProcMsg - ATTC_HANDLE_VALUE_NTF
WdxcProcMsg - ATTC_HANDLE_VALUE_NTF
WDXC: wdxcWdxsValueUpdate.
WDXC file transfer data.
WDXC: wdxcParseFtd.
*** datcWdxcFtdCallback - connId = 1, fileHdl = 1, len = 244
*** datcWdxcFtdCallback - pData (first byte) = 6 (6)
### HCI_DRV_Apollo3: HciDrvIntService
WSF_OS: WsfSetEvent handlerId:6 event:1
WsfSetEvent handlerId:6 event:1
```

Ilustración 275: SWO recepción de tres paquetes con el PDU deseado.

También se muestran las capturas (Ilustración 276, Ilustración 277 e Ilustración 278) de la herramienta *WireShark* en las que se puede observar la transferencia de estos 3 paquetes BLE a través del aire.

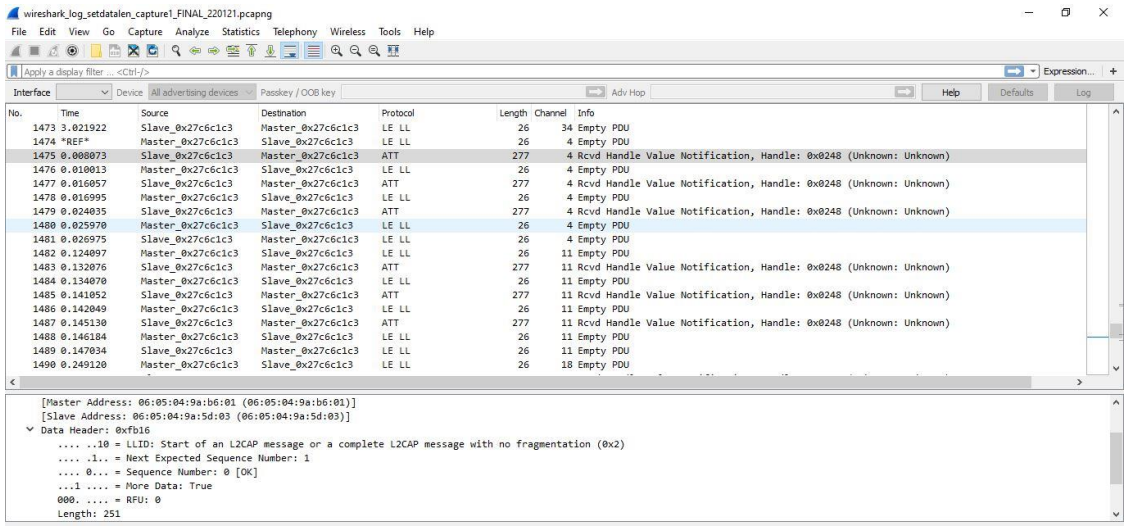


Ilustración 276: Wireshark paquetes caso 1.

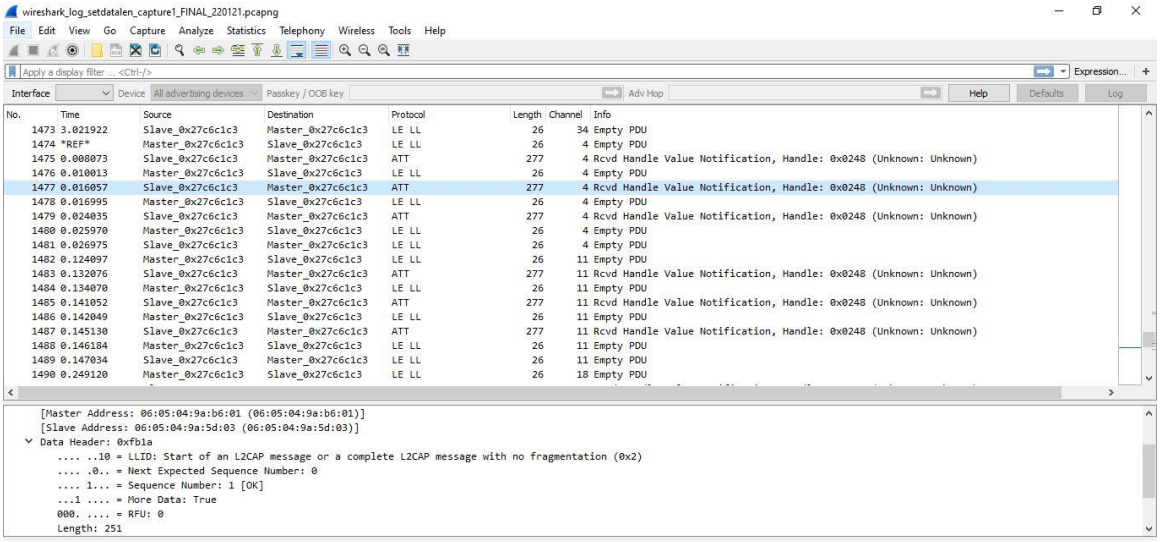


Ilustración 277: Wireshark 2º paquete.

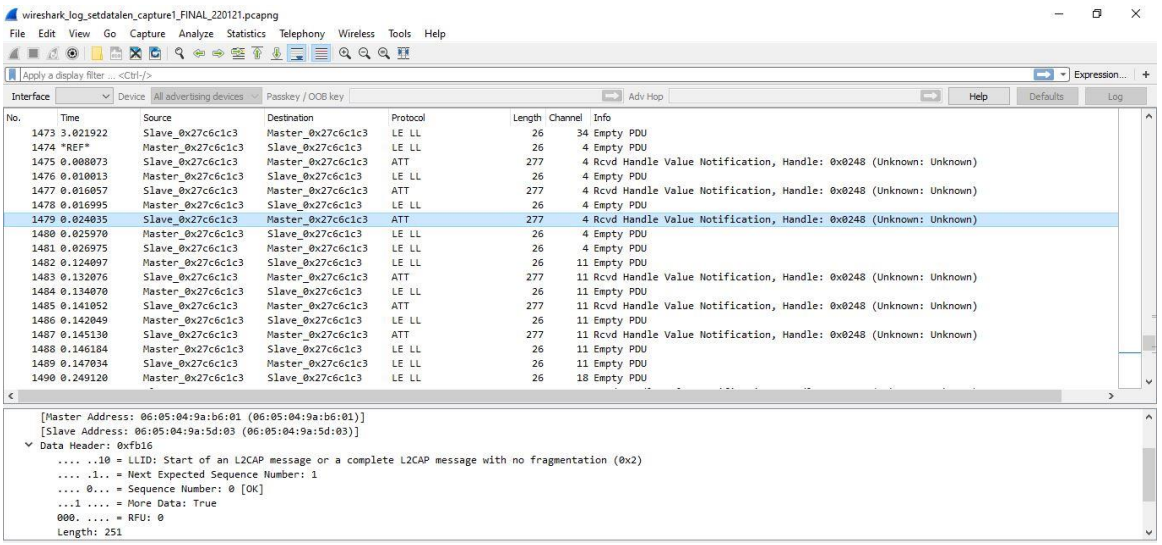


Ilustración 278: Wireshark 3º paquete.

Con el objetivo de determinar qué ocurre en el dispositivo *Peripheral,* y visualizar con mayor facilidad el efecto del control de flujo en este caso particular, se decidió analizar en detalle las situaciones que se generaban en el dispositivo *Peripheral*. Así, primer lugar, en el comienzo del *Stream*, se observa que aún existe un *buffer* disponible por lo que el envío seguirá un flujo "normal" (Ilustración 279).

```
WDXS_STREAM: --- wdxsStreamRead
WDXS_STREAM: --- wdxsSineRead - len = 244, dataval = 16
WDXS_STREAM: --- wdxsSineRead - pBuf (first byte) = 10 (16)
WDXS_FT: wdxsFtdSend - AttsHandleValueNtfZeroCpy: readLen = 244
ATTS_IND: attsHandleValueIndNtf - handle = 0x0248, valueLen = 244, zeroCpy = 1
ATTS_IND: attsHandleValueIndNtf - connection in use - mtu = 247
ATTS_IND: attsHandleValueIndNtf - MTU size known for connection
ATTS_IND: attsHandleValueIndNtf - !transTimedOut
ATTS_IND: attsHandleValueIndNtf - client is aware of any database changes
ATTS_IND: attsHandleValueIndNtf - (valueLen + ATT_VALUE_NTF_LEN) = 247
ATTS_IND: attsHandleValueIndNtf - pMsg->pPkt != NULL
ATTS_IND: attsHandleValueIndNtf - pMsg->pPkt->len = 247, pMsg->pPkt->handle = 0x0248
% WSF_MSG: WsfMsgSend handlerId:3 %
L2C_MAIN: L2cDataReq - len = 247
### HCI_CORE: HciSendAclData
########################################## HCI_CORE: HciSendAclData - len = 251
########################################## HCI_CORE: HciSendAclData - SEND DATA (availBufs = 1)
### HCI_CORE: hciCoreTxAclStart - len=251
```

Ilustración 279: SWO envío con buffers HCI desocupados.

Al ocupar el último *buffer* disponible, se activa el *log HCISendAclData-Flow Disabled = TRUE* (Ilustración 280), indicando que no se enviarán más paquetes hasta que se desocupen los *buffers* al umbral establecido (2).

```
### HCI_CORE: hciCoreTxAclStart - len=251
### HCI_CORE: hciCoreTxAclStart - hciLen=251
### HCI_CORE: hciCoreSendAclData - pConn->outBufs++ = 120
### HCI_TR: hciTrSendAclData: len = 255
### HCI_DRV_Apollo3: hciDrvWrite: len = 255
### HCI_DRV_Apollo3: hciDrvWrite - psWriteBuffer->ui32Length = 256
### HCI_DRV_Apollo3: hciDrvWrite: WsfSetEvent(g_HciDrvHandleID, BLE_TRANSFER_NEEDED_EVENT)
WSF_OS: WsfSetEvent handlerId:7 event:1
WsfSetEvent handlerId:7 event:1
### HCI_TR: hciTrSendAclData: llamada a hciCoreTxAclComplete
### HCI_CORE: hciCoreTxAclComplete
### HCI_CORE: hciCoreSendAclData - hciCoreCb.availBufs = 0
### HCI_CORE: hciCoreTxAclStart - len < hciLen - NO FRAGMENTATION, JUST SEND THE PACKET
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! HCI_CORE: HciSendAclData - FLOW DISABLED = TRUE
```

Ilustración 280: SWO desactiva flujo.

Así, una vez desocupados dos de ellos, se reanuda el envío de paquetes, como se muestra en la Ilustración 281 con *HCICoreTxAclContinue*.

```
HCI_CORE_PS: hciCoreNumCmplPkts - availBufs = 2
### HCI_CORE: hciCoreTxReady
### HCI_CORE: hciCoreTxReady - hciCoreCb.availBufs = 3
### HCI_CORE: hciCoreTxAclContinue
### HCI_CORE: hciCoreTxAclContinue - pConn == NULL
### HCI_CORE: hciCoreTxReady - hciCoreTxAclContinue(NULL) == FALSE
--- WDXS: WdxsHandler - Task Handler Evt=1
WDXS: WdxsHandler - WDXS_EVT_TX_PATH
```

Ilustración 281: SWO reanudar conexión.

A continuación, se realiza el envío de un nuevo paquete cuyo *payload* se corresponde con otro valor generado por la sinusoide configurada en el *Stream,* en este caso de valor 12, hasta llenar el *buffer* nuevamente y paralizar el envío de datos, como se observa en la Ilustración 282.

```
WDXS_STREAM: --- wdxsStreamRead
WDXS_STREAM: --- wdxsSineRead - len = 244, dataval = 12
WDXS_STREAM: --- wdxsSineRead - pBuf (first byte) = c (12)
WDXS_FT: wdxsFtdSend - AttsHandleValueNtfZeroCpy: readLen = 244
ATTS_IND: attsHandleValueIndNtf - handle = 0x0248, valueLen = 244, zeroCpy = 1
ATTS_IND: attsHandleValueIndNtf - connection in use - mtu = 247
ATTS_IND: attsHandleValueIndNtf - MTU size known for connection
ATTS_IND: attsHandleValueIndNtf - !transTimedOut
ATTS_IND: attsHandleValueIndNtf - client is aware of any database changes
ATTS_IND: attsHandleValueIndNtf - (valueLen + ATT_VALUE_NTF_LEN) = 247
ATTS_IND: attsHandleValueIndNtf - pMsg->pPkt != NULL
ATTS_IND: attsHandleValueIndNtf - pMsg->pPkt->len = 247, pMsg->pPkt->handle = 0x0248
% WSF_MSG: WsfMsgSend handlerId:3 %
L2C_MAIN: L2cDataReq - len = 247
### HCI_CORE: HciSendAclData
######################################## HCI_CORE: HciSendAclData - len = 251
######################################## HCI_CORE: HciSendAclData - SEND DATA (availBufs = 3)
```

Ilustración 282: SWO envío de otro valor de la sinusoide.

Con el fin de poder obtener más evidencias del efecto del número/tamaño de los *buffers* del Controlador, y el mecanismo del contro*l* de flujo HCI asociado en la limitación del número de paquetes BLE enviados desde el dispositivo *Peripheral* al dispositivo *Central,* en un Evento de Conexión que se obtuvo inicialmente, se llevó a cabo una prueba experimental adicional en la que se decrementaba el valor del umbral *HCI_ACL_QUEUE_LO*, del valor por defecto (3), al valor 2, lo que implicaría que el valor de *HCICoreCb.aclqueueLo* se decremente a 1. Con esta modificación se esperaría una reducción del *Throughput* obtenido para los paquetes de configuración establecidos en el CASO 1, puesto que la desactivación de la deshabilitación del control del flujo se produciría ahora cuando hubiera al menos 3 *buffers* libres, o lo que es lo mismo, cuando el valor de *HCICoreCb.availbufs* fuera 3, estando *FlowDisabled = TRUE*. Los resultados obtenidos son los que se muestran en la Ilustración 283, y confirman lo esperado, una reducción en el valor del *Throughput* obtenido experimentalmente.

```
--- WDXC_INCLUDED =  TRUE --- CS50_INCLUDED =  TRUE ---
-------------------------------------------------------
1. BLE_SCAN+CONN
2. BLE_SEND

hint: use 'h' to do main menu
-------------------------------------------------------
1. WDXC Discover files on periheral (WDXC/WDXS_INCLUDED = TRUE)
2. WDXC Start/Stop Streaming (WDXC/WDXS_INCLUDED =  TRUE)
3. Send BLE_DATA 'OK' (WDXC/WDXS_INCLUDED =  FALSE)

Choose a conn_id to perform WDXC Discover files (1):
 - WDXC File discovery completed from peripheral with conn_id = 1
Choose a conn_id to Start/Stop WDXC Streaming (1):
 - Start WDXC Streaming from peripheral with conn_id = 1
 -- Streaming data rate 33672 bps, overall 33672 bps
 -- Streaming data rate 32208 bps, overall 32940 bps
 -- Streaming data rate 33184 bps, overall 33021 bps
 -- Streaming data rate 34160 bps, overall 33306 bps
 -- Streaming data rate 32208 bps, overall 33086 bps
Choose a conn_id to Start/Stop WDXC Streaming (1):
 -- Streaming data rate 33184 bps, overall 33102 bps
 - Stop WDXC Streaming from peripheral with conn_id = 1
```

Ilustración 283: Putty Throghtput cambio de flujo en el caso 1.

6.5.2 Pruebas adicionales

Para corroborar que el *Throughput* en uso es el máximo, se plantean más casos, modificando los diferentes parámetros de configuración BLE.

CASO 2:

En este caso, se configura un dispositivo *Central* que establece una conexión con los parámetros característicos que se muestran en Ilustración 284. Antes de realizar las pruebas hay que tener en cuenta que al reducir considerablemente el valor del Intervalo de Conexión, se reduce el número de paquetes transmitidos a como máximo, 1 paquete por Evento de Conexión.

```
/*! Connection parameters */
static const hciConnSpec_t datcConnCfg =
{
  15,                                 /*! Minimum connection interval in 1.25ms units */ //- mínimo = 6
  15,                                 /*! Maximum connection interval in 1.25ms units */
  0,                                  /*! Connection latency */
  600,                                /*! Supervision timeout in 10ms units */
  0,                                  /*! Unused */ //FBTG - minCElen
//  0                                    /*! Unused */ //FBTG - maxCElen
  65535                                  /*! Unused */ //FBTG - maxCElen
};
```

Ilustración 284: Parámetros de conexión Caso 2.

Al acceder al menú BLE configurado, se obtiene el siguiente *Throughput*, que se corresponde aproximadamente con 63kbps (Ilustración 285).

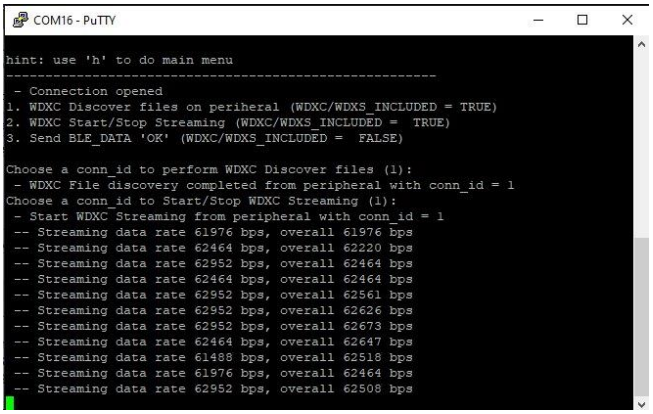


Ilustración 285: Putty Throughput caso 2.

Teóricamente, no siempre es posible enviar un paquete BLE en un Intervalo de Conexión debido a los parámetros establecidos, observándose tras realizar varias pruebas, que se envía un paquete cada 1-2 Intervalos de Conexión. Por ello, se evalúan los dos casos más típicos con esta configuración:

- Para 1 paquete BLE enviado por Intervalo de conexión:

$$Throughput = \frac{8\ (bits) x\ 1\ (paquetes\ por\ ic)\ x\ 244\ (bytes\ de\ payload)}{Ic\ (\ Intervalo\ de\ conexión)} \approx 104{,}1\ Kbps$$

- Para 1 paquete BLE enviado por cada 2 Intervalos de conexión:

$$Throughput = \frac{8\ (bits) x\ 1\ (paquetes\ por\ ic)\ x\ 244\ (bytes\ de\ payload)}{2xIc\ (\ Intervalo\ de\ conexión)} \approx 52{,}05\ Kbps$$

CASO 3:

Para este último caso, se busca configurar un Intervalo de Conexión elevado y observar qué ocurre con el *Throughput.* Así, se establece un Intervalo de Conexión de 675 ms (Ilustración 286).

```
/*! configurable parameters for connection parameter update */
static const appUpdateCfg_t datsUpdateCfg =
{
  500,                                  /*! Minimum connection interval in 1.25ms units */
  500,                                  /*! Maximum connection interval in 1.25ms units */
  0,                                    /*! Connection latency */
  600,                                  /*! Supervision timeout in 10ms units */
  5                                     /*! Number of update attempts before giving up */
};
```

Ilustración 286: Parámetros de conexión para Caso 3.

Cómo ya se ha demostrado, sin importar el incremento en el valor del Intervalo de Conexión, el máximo envío de paquetes será de 4 paquetes por Evento de Conexión. Por lo tanto, el *Througthput* medido desde la aplicación se muestra en la Ilustración 287. Como era de esperar, empeora significativamente, siendo el incremento del Intervalo de Conexión una opción totalmente inviable si se busca obtener un *Throughtput* máximo.

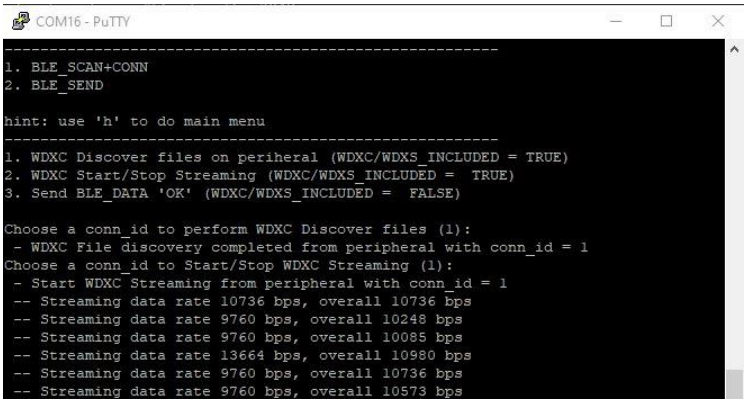


Ilustración 287: Putty Througthput caso 3.

- Para 4 paquetes en un intervalo:

$$Throughput = \frac{8\ (bits) x\ 4\ (paquetes\ por\ ic)\ x\ 244\ (bytes\ de\ payload)}{Ic\ (\ Intervalo\ de\ conexión)} \approx 12{,}49\ Kbps$$

- Para 3 paquetes en un intervalo:

$$Throughput = \frac{8\ (bits) x\ 3\ (paquetes\ por\ ic)\ x\ 244\ (bytes\ de\ payload)}{Ic\ (\ Intervalo\ de\ conexión)} \approx 9{,}36\ Kbps$$

Así, se llega a la conclusión que para la elección del Intervalo de Conexión se debe tener en cuenta que el incremento de este valor en la plataforma *Artemis Thing Plus* no asegura el envío de un mayor número de paquetes BLE por Evento de Conexión, es más, por limitaciones de los *buffers* en el Controlador, como mucho se obtienen 3 o 4 paquetes por Evento de Conexión, por lo que se debe establecer un valor de Intervalo de Conexión que permita enviar 4 paquetes, pero no superior.

Capítulo 7: Conclusiones

7.1 Conclusiones

En cuanto a los objetivos planteados inicialmente en este Trabajo Fin de Grado, se puede afirmar que han sido cumplidos durante su realización. Así, ha sido posible la implementación con éxito de un dispositivo *Central/Peripheral* en la placa *Artemis Thing Plus* a partir del SDK Ambiq 3, basado en la pila *Cordio BLE*, para finalmente realizar el un análisis experimental del *Throughput*.

Para ello, en primer lugar, se ha llevado a cabo un estudio teórico del estándar BLE 5. A continuación, se implementó la funcionalidad básica de un dispositivo *Peripheral* en la plataforma *Artemis Thing Plus* a partir del SDK proporcionado por *Sparkfun*. Tanto *Cordio BLE* (pila BLE utilizada) como la placa *Artemis Thing Plus* presentaban escasa documentación por parte del fabricante, y su uso. En consecuencia, se optó por documentarlo en el proyecto, ya que igualmente se había tenido que realizar ese estudio para esta plataforma, además de realizar un método propio de conversión de ficheros para esta librería. Una vez documentado y configurado el dispositivo *Peripheral* básico, se validó el funcionamiento inicialmente mediante la aplicación *Android nRFConnect* actuando como dispositivo *Central*.

Seguidamente, se implementó la funcionalidad de un dispositivo *Central* en la plataforma *Artemis Thing Plus*. Con ello, se creó un menú para validar su funcionamiento junto al dispositivo *Peripheral* desarrollado, desde donde se controla la conexión como dispositivo *Central.*

A partir de la implementación de ambos dispositivos, se estudió el método más efectivo para enviar datos aprovechando al máximo las funcionalidades de BLE 5. Finalmente, se optó por la funcionalidad WDXS/WDXC que contiene la pila *Cordio BLE*. Nuevamente, la documentación era escasa, lo que obligó a realizar un estudio de los ficheros que implementaba esta funcionalidad para realizar así una conexión BLE para la transferencia de datos en *Streaming,* que permitiera el envío continuo de paquetes con un *payload* determinado.

Llegados a este punto, se realizó el estudio del *Thougthput*. Para ello, se concretaron cuáles son los factores más importantes que influyen para optimizar este parámetro. Se consideraron los aspectos teóricos de la especificación, y cómo implementa la pila *Cordio* BLE dicho protocolo. Se pudieron establecer los siguientes apartados a tener en cuenta a la hora de maximizar el *Throughtput*:

- El PHY utilizado (*LE 1M, LE2M* o *LE Coded*).
- El valor del intervalo de conexión (CI).
- El valor de la unidad máxima de transmisión *ATT* (*ATT* MTU).
- La extensión de la longitud de datos (DLE).
- El tipo de operación: *Write with response/ Write without response, Indication/Notification*.

- IFS (*Inter Frame Space*), que corresponde al lapso de tiempo entre paquetes consecutivos (150 µs).
- La transmisión de paquetes vacíos (*Empty PDU*).
- El *overhead* de los paquetes, ya que no todos los bytes de un paquete se utilizan para el *payload* de la aplicación.

Tras el estudio de estos parámetros, se configuran en los dispositivos *Central/Peripheral* implementados, observando en su ejecución un comportamiento anómalo, ya que se enviaba un número de paquetes muy limitado comparado con el esperado, de acuerdo con los parámetros configurados. Tras observar mediante un *Sniffer* que la longitud de los paquetes no se correspondía con la establecida, se dio por hecho que el parámetro DLE no se configuraba correctamente. Se realizó nuevamente un análisis de las funciones proporcionadas con la librería *Cordio* BLE en el módulo DM, y se llegó a la conclusión de que era necesario añadir la función *DmConnsetDataLen* para que estos parámetros se configurasen correctamente.

Por último, tras realizar las modificaciones pertinentes, se realizó el cálculo del *Thoughtput* presentando una mejora significativa con respecto al valor inicialmente obtenido. Aun así, se seguía observando un comportamiento anómalo que, tras el estudio de éste, llevó a la conclusión de que existe una limitación por parte del Controlador que limita el envío de paquetes por Intervalo de Conexión a 4, ya que es el máximo número de paquetes que se pueden almacenar en los *buffers* del Controlador.

Bibliografía

[1] J. P. Alvaro Everlet, “Introducción al internet de las cosas.,” *Introducción al internet de las cosas.* https://cmapspublic2.ihmc.us/rid=1P6R5NKLS-SR8LMD-4NT1/Construyendo_un_proyecto_de_IOT.pdf (accessed Feb. 02, 2020).

[2] Statista, “IoT: number of connected devices worldwide 2012-2025 | Statista.” https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ (accessed Feb. 02, 2020).

[3] J. Carlos, R. Giraldo, and A. R. Castellano, “Estado del arte de la bioinformática,” vol. 61850, pp. 1–16, 2012.

[4] M. Afaneh, *Intro to Bluetooth Low Energy*, 1º. NovelBits, 2018.

[5] J. Macias, “Bluetooth BLE: el conocido desconocido,” 2018. https://solidgeargroup.com/bluetooth-ble-el-conocido-desconocido/?lang=es (accessed Feb. 03, 2020).

[6] “SparkFun Thing Plus - Artemis - WRL-15574 - SparkFun Electronics.” https://www.sparkfun.com/products/15574 (accessed Feb. 02, 2020).

[7] G. R. Rodr, T. G. D. Valent, and D. A. S. Septiembre, “TELECOMUNICACIÓN Y ELECTRÓNICA PROPUESTA DE TRABAJO FIN DE MÁSTER Estudio de la tasa de transferencia del dispositivo IoT RedBear Duo en comunicaciones BLE,” 2017.

[8] K. Sakamoto and T. Furumoto, *Getting Started with Blocks*. 2012.

[9] “Exploring Bluetooth 5 -Going the Distance | Bluetooth® Technology Website.” https://www.bluetooth.com/blog/exploring-bluetooth-5-going-the-distance/ (accessed Jun. 01, 2021).

[10] “4. G*ATT* (Services and Characteristics) - Getting Started with Bluetooth Low Energy [Book].” https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html (accessed Jun. 01, 2021).

[11] “Maximizing BLE Throughput Part 3: Data Length Extension (DLE) - Punch Through.” https://punchthrough.com/maximizing-ble-throughput-part-3-data-length-extension-dle-2/ (accessed Jun. 10, 2021).

[12] “Arm Mbed *Cordio* - Arm Mbed *Cordio* | Mbed *Cordio* Documentation.”

https://os.mbed.com/docs/mbed-*Cordio*/19.02/introduction/index.html (accessed Jun. 10, 2021).

[13] "Using SparkFun Edge Board with Ambiq Apollo3 SDK - learn.sparkfun.com." https://learn.sparkfun.com/tutorials/using-sparkfun-edge-board-with-ambiq-apollo3-sdk/bash-make-and-python----oh-my (accessed Feb. 02, 2020).

[14] "Hookup Guide for the SparkFun Artemis Thing Plus - learn.sparkfun.com." https://learn.sparkfun.com/tutorials/hookup-guide-for-the-sparkfun-artemis-thing-plus/all (accessed Jun. 11, 2021).

[15] J. Mayné, "Sistemas de Comunicaciones."

[16] "Apollo3 Blue - Ambiq." https://ambiq.com/apollo3-blue/ (accessed Jun. 11, 2021).

[17] N. Parody, N. Ca, and A. Fecha, "16F876 de Microchip .," 2006.

[18] "Using SparkFun Edge Board with Ambiq Apollo3 SDK - learn.sparkfun.com." https://learn.sparkfun.com/tutorials/using-sparkfun-edge-board-with-ambiq-apollo3-sdk/all#bash-make-and-python----oh-my (accessed Jun. 11, 2021).

[19] "GitHub - sparkfun/SparkFun_Apollo3_AmbiqSuite_BSPs at bb26f2761df92d3c31d9c8d0200e3385a136764c." https://github.com/sparkfun/SparkFun_Apollo3_AmbiqSuite_BSPs/tree/bb26f2761df92d3c31d9c8d0200e3385a136764c (accessed Jun. 11, 2021).

[20] "GitHub - sparkfun/AmbiqSuiteSDK: A copy of the AmbiqSuite SDK available on GitHub. Can be used to include AmbiqSuite as a submodule. May be used to track issues in SDK releases, however this repo is not maintained by AmbiqMicro." https://github.com/sparkfun/AmbiqSuiteSDK (accessed Jun. 11, 2021).

[21] "GitHub - adafruit/Adafruit-MLX90614-Library: Arduino library for the MLX90614 sensors in the Adafruit shop." https://github.com/adafruit/Adafruit-MLX90614-Library (accessed Jun. 14, 2021).

[22] "GitHub - 0xcart/artemis: Visual Studio Code development environment for SparkFun Artemis based boards." https://github.com/0xcart/artemis (accessed Jun. 14, 2021).

[23] "Wireless Data Exchange Profile." http://www.reisenweber.net/et_al/card10/firmware/lib/sdk/Libraries/BTLE/documentation/html/*Cordio*_Stack_*Cordio*_Profiles/group___w_i_r_e_l_e_s_s___d_a_t_a___e_x_c_h_a_n_g_e___p_r_o_f_i_l_e.html (accessed Jun. 17, 2021).

[24] "Maximizing BLE Throughput on iOS and Android - Punch Through." https://punchthrough.com/maximizing-ble-throughput-on-ios-and-android/ (accessed Jun. 18, 2021).

[25] “Bluetooth Packet Structure - MATLAB & Simulink - MathWorks España.” https://es.mathworks.com/help/comm/ug/bluetooth-packet-structure.html (accessed Jun. 18, 2021).

[26] “Nordic Semiconductor - Home - nordicsemi.com.” https://www.nordicsemi.com/ (accessed Jul. 06, 2021).

[27] “nRF Sniffer for 802.15.4 - nordicsemi.com.” https://www.nordicsemi.com/Products/Development-tools/nrf-sniffer-for-802154 (accessed Jul. 06, 2021).

Pliego de Condiciones

Este documento expone las condiciones bajo las que se ha desarrollado el presente Trabajo Fin de Grado. A continuación, se muestran el conjunto de herramientas hardware, software y firmware empleadas durante su realización.

PL. 1 Condiciones Hardware

En la Tabla PL.P - 1 se presentan los equipos hardware utilizados.

Tabla PL.P - 1: Relación de equipos Hardware

Equipo	Modelo	Fabricante
Ordenador	• CPU AMD Ryzen 5 3600 6-Core Processor a 3.59 GHz • 16 GB RAM • 1 Tb de disco duro	
Artemis Thing Plus	• Firmware 2.4.2	*Sparkfun*
J-Link EDU mini		Segger
Sensor de temperatura	• MLX90614	Adafruit
Sniffer	• nRF52840	Nordic Semiconductor

PL. 2 Condiciones Software

En la Tabla PL.P - 2 se exponen las herramientas *software* utilizadas, especificando su versión.

Tabla PL.P - 2: Relación de herramientas software.

Aplicación	Versión	Fabricante
Microsoft Office	2013	Microsoft
Notepad ++	8.1	Plataforma código abierto NOTEPAD ++
Wireshark	1.12.3	The Wireshark developer community
nRfConnect	4.14.0	Nordic Semiconductor
***SWO* viewer**		Segger
Python	3.8	Python
Bash	2.32	Git
Make	4.2.1	
PuTTy	0.75	

PL. 3 Condiciones firmware

Por último, en la Tabla PL.P - 3 se muestra el *firmware* utilizado y su versión.

Tabla PL.P - 3: Relación de firmware

Frimware	Versión
AmbiqSuiteSDK	2.4.2
***Cordio* BLE Stack**	19.02

Presupuesto

Este capítulo contiene el presupuesto que recoge los gastos generados en la realización del presente Trabajo Fin de Grado. Dicho presupuesto se divide en las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, dividido 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.

Una vez analizados cada uno de los criterios establecidos, se aplicarán los impuestos vigentes y se procederá a la obtención del coste total del TFG.

P. 1 Trabajo tarifado por tiempo empleado

Este concepto contabiliza los gastos que corresponden a la mano de obra, según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación.

Según la tabla retributiva de personal contratado en proyectos de investigación, elaborada por la ULPGC en el año 2019, para una dedicación de 20 horas semanales, el salario asciende a 711,90 €. Además, es importante saber que la duración aproximada del TFG es de 4 meses, por lo que el cálculo del coste total se realiza por tiempo empleado.

$$711{,}90 \times 4 = 2847{,}60\ €$$

Por lo tanto, el trabajo tarifado por tiempo empleado asciende a la cantidad de *dos mil ochocientos cuarenta y siete euros con sesenta céntimos*.

P. 2 Amortización del inmovilizado material

En el inmovilizado material se consideran tanto los recursos hardware como softwares empleados para la realización de este TFG.

Para estipular el coste de amortización en un periodo de 3 años se utiliza 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 de la siguiente forma:

$$Cuota\ anual = \frac{Valor\ de\ adquisición - Valor\ residual}{Número\ de\ años\ de\ vida\ útil} \quad \text{(P.3)}$$

Donde el valor residual se corresponde con el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil.

P.3 Amortización del material hardware

Debido a que la duración de este Trabajo Fin de Grado es de tan solo 4 meses, siendo este periodo muy inferior al de 3 años estipulado para el coste de amortización, los costes se calcularán en base a los derivados de los primeros 4 meses de uso.

En la Tabla P-2 se especifica el hardware amortizable necesario para la realización del trabajo, indicando su valor de adquisición y su amortización, teniendo en cuenta un tiempo de uso de 4 meses.

Tabla P- 1: Costes y amortización del hardware (I)

Elemento	Valor de adquisición	Amortización
Ordenador AMD custom	734,00 €	81,58 €
Total	**734,00 €**	**81,58 €**

Por otro lado, en la Tabla P-3 se muestra el resto de material hardware utilizado y por el que, debido a su bajo precio, su amortización coincide con el valor de adquisición.

Tabla P- 2: Costes y amortización del hardware (II)

Elemento	Valor de adquisición	Amortización
Artemis Thing Plus (x2)	**35.40 €**	35,40 €
J-Link EDU mini (x2)	**43,82 €**	43,82 €
Sensor MLX90624	**11.63 €**	11,63 €
nRF52840 (x2)	**16,12 €**	16,12 €
Total	**106,97 €**	**106,97 €**

Finalmente, tras realizar la suma de ambos se obtiene el coste total del material hardware, tal y como se muestra en la Tabla P-4.

Tabla P- 3: Costes y amortizaciones totales del hardware

Concepto	Coste
Costes y amortización del hardware (I)	81,58 €
Costes y amortización del hardware (II)	106,97 €
Total	**188,55 €**

El coste total del material hardware asciende a ciento *ochenta y ocho euros con cincuenta y cinco céntimos*.

P.4 Amortización del software

Para el cálculo de los costes de amortización del material software se considerará, al igual que con el material hardware, los costes derivados de los primeros 4 meses de uso.

La Tabla P-5 muestra los elementos software necesarios para la realización del trabajo, así como su valor de adquisición y su amortización.

Tabla P- 4: Costes y amortización del software

Elemento	Valor de adquisición	Amortización
Microsoft Office 2013	0,00 € (*)	0,00 €
Notepad ++	0,00 € (**)	0,00 €
Wireshark	0,00 € (**)	0,00 €
nRfConnect	0,00 € (**)	0,00 €
***SWO* viewer**	0,00 € (**)	0,00 €
PuTTy	0,00 € (**)	0,00 €
Make	0,00 € (**)	0,00 €

Phyton	0,00 € (**)	0,00 €
Bash	0,00 € (**)	0,00 €
Total	**0 €**	**0 €**

(*) *Licencia de uso proporcionada por la ULPGC.*

(**) *Software libre.*

Por tanto, el coste total del material software es de *cero* euros.

P. 5 Redacción del trabajo

Se ha utilizado (P.4) para determinar el coste asociado a la redacción de la memoria del presente Trabajo Fin de Grado.

$$R = 0,07 \times P \times C_n \,, \qquad \text{(P.4)}$$

donde:

- R son los honorarios por la redacción del trabajo.
- P es el presupuesto.
- C_n es el coeficiente de ponderación en función del presupuesto.

El valor del presupuesto P se calcula sumando los costes del trabajo tarifado por tiempo empleado y de la amortización del inmovilizado material, tanto hardware como software. El resultado de los costes se muestra en la Tabla P-5.

Tabla P- 5: Presupuesto, incluyendo trabajo tarifado y amortización del inmovilizado material.

Concepto	Coste
Trabajo tarifado por tiempo empleado	2847,60 €

Amortización del material hardware	188,55 €
Amortización del software	0,00 €
Total	**3.036,15 €**

Como el coeficiente de ponderación *Cn* para presupuestos menores de 30.050,00€ viene definido por el COITT con un valor de 1.00, el coste derivado de la redacción del Trabajo Fin de Grado es de:

$$R = 0,07 \times 3.036,15 \times 1 = 212,53 \text{ €} \quad \text{(P.5)}$$

Ascendiendo de esta forma el coste de la redacción del trabajo a *doscientos doce* euros *y cincuenta y tres* céntimos.

P. 6 Derechos de visado del COITT

El COITT establece que, para proyectos técnicos de carácter general, que los derechos de visado para 2021 se calculan en base a (P.6).

$$V = 0,006 \times P_1 \times C_1 + 0,003 \times P_2 \times C_2 \quad \text{(P.6)}$$

donde:

- V es el coste de visado del trabajo.
- P_1 es el presupuesto del proyecto.
- C_1 es el coeficiente reductor en función del presupuesto.
- P_2 es el presupuesto de ejecución material correspondiente a la obra civil.
- C_2 es el coeficiente reductor en función a P_2.

El valor del presupuesto P_1 se halla sumando los costes de las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del documento. Esta suma se muestra en la Tabla P-7. Al igual que en el caso anterior, el coeficiente C_1 para proyectos de presupuesto inferior a 30.050,00€ es de 1,00, asimismo el valor de P_2 es de 0,00€ ya que no se realiza ninguna obra.

Tabla P- 6: Presupuesto, incluyendo trabajo tarifado, amortización y redacción del trabajo

Concepto	Coste
Trabajo tarifado por tiempo empleado	2.847,60 €
Amortización del material hardware	188,55 €
Amortización del material software	0,00 €
Redacción del trabajo	212,53 €
Total	**3.248,68 €**

De esta forma, aplicando a (P.6) los datos de la tabla P-7 y el coeficiente especificado se obtiene:

$$V = 0,006 \times 3.248,68 \times 1 = 19,49 \text{ €} \qquad \text{(P.7)}$$

Los costes por derechos de visado del presupuesto ascienden a *diecinueve euros* con *cuarenta y nueve* céntimos.

P. 5 Gastos de tramitación y envío

Los gastos de tramitación y envió están estipulados en seis euros (6,00€) por cada documento visado de forma telemática.

P. 6 Aplicación de impuestos y coste total

La realización del presente TFG está gravada por el Impuesto General Indirecto Canario (IGIC) en un siete por ciento (7%). En la Tabla P-8 se muestra el presupuesto final con los impuestos aplicados.

Tabla P- 7:Presupuesto total del Trabajo Fin de Grado

Concepto	Coste
Trabajo tarifado por tiempo empleado	2.847,60 €
Amortización del material hardware	188,55 €
Amortización del material software	0,00 €
Redacción del trabajo	212,53 €
Derechos de visado del COITT	19,49 €
Gastos de tramitación y envío	6,00 €
Total (Sin IGIC)	**3.274,17 €**
IGIC (7%)	229,19 €
Total	**3.503,33 €**

El presupuesto total del presente TFG asciende a un total de *tres mil quinientos tres* euros con *treinta y seis céntimos*.

Así lo declaro, Jorge Fernández Heredero, en Las Palmas de Gran Canaria a

15 de Julio de 2021