UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster Universitario en Tecnologías de Telecomunicación

TRABAJO FIN DE MÁSTER

Integración de un sistema de registro y transmisión vía BLE de la actividad bioeléctrica cerebral basado en dispositivos IoT

Autor: Dña. Laura Burgos Muñiz

Tutor(es): D. Valentín de Armas Sosa

D. Félix Tobajas Guerrero

Fecha: Julio de 2017

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster Universitario en Tecnologías de Telecomunicación

TRABAJO FIN DE MÁSTER

Integración de un sistema de registro y transmisión vía BLE de la actividad bioeléctrica cerebral basado en dispositivos IoT

HOJA DE FIRMAS

Alumna

Fdo.: Burgos Muñiz, Laura

Tutor 1 **Tutor 2**

Fdo.: De Armas Sosa, Valentín Fdo.: Tobajas Guerrero, Félix

Fecha: Julio, 2017

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster Universitario en Tecnologías de Telecomunicación

TRABAJO FIN DE MÁSTER

Integración de un sistema de registro y transmisión vía BLE de la actividad bioeléctrica cerebral basado en dispositivos IoT

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario/a**

Fdo.: Fdo.:

Fecha: Julio, 2017

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Tabla de Contenidos

Índice de Figuras

Lista de Acrónimos

ADC	Analogue to Digital Converter
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
CAN	Controller Area Network
CCCD	Client Characteristic Configuration Descriptor
CD-ROM	Compact Disc Read-Only Memory
COIT	Colegio Oficial de Ingenieros de Telecomunicación
CSV	Comma Separated Values
DAC	Digital to Analogue Converter
DSI	Diseño de Sistemas Integrados
EEG	Electroencefalograma
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GPIO	General Purpose Input/Output
I2C	Inter-Integrated Circuit
I2S	Inter-IC Sound
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
ISM	Industrial Scientific and Medical
IUMA	Instituto Universitario de Microelectrónica Aplicada
LED	Light-Emitting Diode
PC	Personal Computer
PPCP	Peripheral Preferred Connection Parameters
PWM	Pulse-Width Modulation
RGB	Red, Green, Blue

RTOS	Real-Time Operating System
SIG	Special Interest Group
SMD	Surface Mount Device
SPI	Serial Peripheral Interface
TGAM	ThinkGear ASIC Module
TIC	Tecnologías de la Información y las Comunicaciones
TFM	Trabajo Fin de Máster
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver-Transmitter
ULPGC	Universidad de Las Palmas de Gran Canaria
USB	Universal Serial Bus
UUID	Universally Unique Identifier
WiFi	Wireless Fidelity

Memoria

Capítulo 1. Introducción

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

1.1 Introducción

En los últimos años, la continua transformación de las Tecnologías de la Información y las Comunicaciones (TIC) ha cambiado drásticamente la forma en la que vivimos. Uno de los aspectos clave en los que este hecho se ve reflejado es en las interacciones humanas a través de Internet. Comúnmente, los usuarios utilizan esta herramienta como medio de comunicación virtual, abarcando todo tipo de intercambio de información, desde sus relaciones sociales hasta su vida profesional.

Esta tendencia ha hecho posible la interconexión entre personas a un ritmo y a una escala sin precedentes. Sin embargo, el aumento del número de dispositivos conectados a la red, y la posibilidad de intercambio de información que puede darse entre ellos, ha llevado a plantear una evolución del uso de Internet.

Es en este punto donde surge el concepto de Internet de las Cosas (*Internet of Things, IoT*). La base de esta idea consiste en la utilización de Internet para interconectar objetos entre sí, creando un entorno inteligente y fácilmente accesible por los usuarios **[1]**. El modelo de IoT contempla una interacción basada en la computación ubicua, entendida como la integración de dispositivos presentes en la vida cotidiana de las personas, con los cuales puedan interactuar para realizar actividades cotidianas **[2]**.

Cada día mayor número de objetos son capaces de sensorizar, monitorizar y gestionar datos de su entorno de forma automática, disponibles a través de la red. El impacto es exponencial en la salud, la ayuda a la discapacidad, los sistemas de transportes o el desarrollo de las ciudades inteligentes. El número de dispositivos conectados crecerá un 30% respecto al año 2015, hasta alcanzar los 6.400 millones de objetos en el año 2016; 5,5 millones de objetos se conectarán cada día. Manteniendo la tendencia, en el año 2020 existirán 20.800 millones de dispositivos aplicados a Internet de las Cosas **[3]**.

Así, entre otros ámbitos, gracias a las aplicaciones IoT es posible ofrecer a las personas con discapacidad la ayuda y el apoyo que necesitan para lograr una buena calidad de vida y permitirles integrarse en la vida social y económica. En la actualidad, alrededor

del 15% de la población mundial vive con algún tipo de discapacidad. En la mayoría de los casos dicha discapacidad hace que las personas sean excesivamente dependientes y tengan problemas de integración y accesibilidad **[4]**.

Se prevé que, gracias a las tecnologías de IoT, será factible el desarrollo de dispositivos de coste muy reducido que garanticen una vida más autónoma e independiente a este tipo de usuarios. En este ámbito, varios fabricantes han desarrollado sensores de electroencefalograma (EEG) que pueden integrarse con diferentes plataformas, en la mayoría de los casos mediante soluciones propietarias, basadas en el uso de tecnologías inalámbricas, lo que permite transmitir, especialmente en el caso de las personas con discapacidad física, las señales neurofisiológicas procedentes del cerebro **[4]**.

Este Trabajo Fin de Máster (TFM) se centrará en la integración de dispositivos IoT con un sensor de EEG y pretende contribuir a la propuesta fundamentada de un próximo proyecto de investigación en la línea de investigación de Tecnologías de la Telecomunicación, en concreto en la División de Diseño de Sistemas Integrados.

1.2 Objetivos

El objetivo general de este TFM consiste en el diseño e implementación de una plataforma HW/SW que, basándose en el uso de dispositivos IoT, permita a personas con discapacidad interactuar con el medio que les rodea a partir del registro de la actividad bioeléctrica cerebral para su posterior procesamiento.

Para la consecución de dicho objetivo se propone el desarrollo de una plataforma inalámbrica basada en el estándar BLE (*Bluetooth Low Energy*) o *Bluetooth* 4.0 cuyas principales aportaciones se centran en constituir una solución de muy bajo coste y consumo que permita visualizar, transmitir y/o procesar comandos a partir del registro y/o procesamiento de la información proporcionada por un sensor de electroencefalograma y actuar posteriormente en base a ellos.

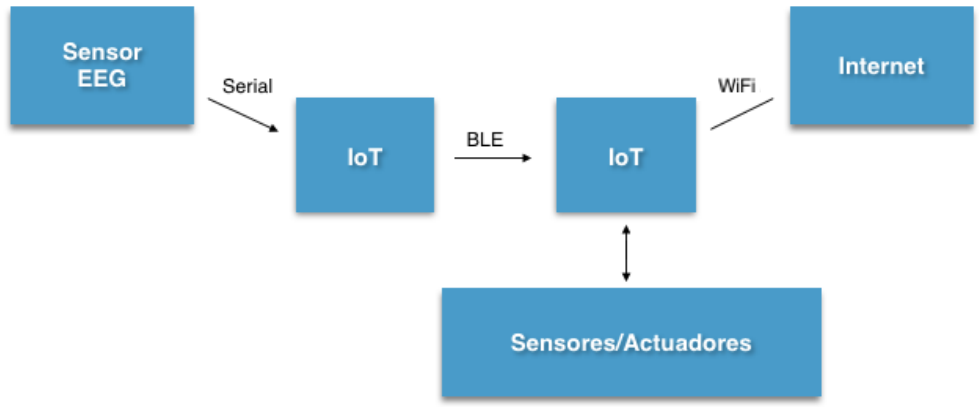


Figura 1.1 - Elementos principales de la plataforma HW/SW propuesta

En la Figura 1.1 se muestran los elementos principales de la plataforma HW/SW propuesta. El sensor de electroencefalograma empleado será el dispositivo *Mindflex* que integra el módulo *ThinkGear ASIC* (TGAM) **[5],** no compatible con BLE, y como dispositivos de IoT se emplearán las placas de desarrollo *Bluz DK* **[6]** y *RedBear Duo* **[7]**.

La placa de desarrollo *Bluz DK,* de la empresa *Bluz*, mostrada en la Figura 1.2, está basada en el procesador *ARM Cortex-M0* y ofrece una solución de bajo coste compatible con el estándar BLE. Este dispositivo actuará como dispositivo *Peripheral*, y recibirá a través de una interfaz serie la información registrada por el sensor de electroencefalograma para enviarla a través de una conexión BLE al dispositivo *RedBear Duo*.

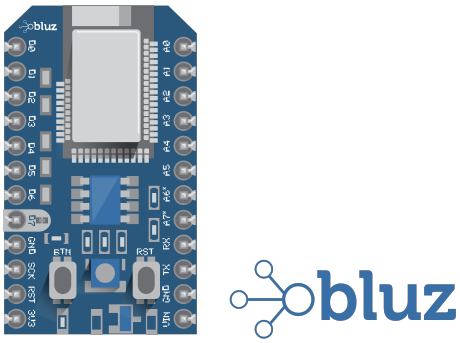


Figura 1.2 - Dispositivo Bluz DK

Por su parte, el dispositivivo *RedBear Duo,* de la empresa *RedBear*, mostrado en la Figura 1.3, es una placa de desarrollo de bajo coste enfocada a productos de IoT que cuenta con soporte WiFi y BLE entre sus características. Este dispositivo actuará como dispositivo *Central,* y además de proporcionar conectividad con Internet a través de

una conexión WiFi. De este modo, gracias al uso de BLE, se conseguirá abstraer una comunicación serie de datos.

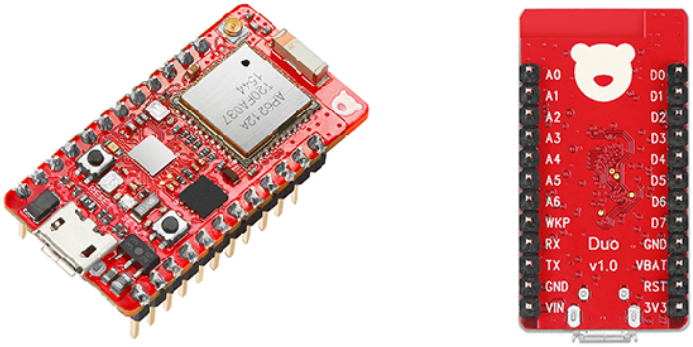


Figura 1.3 - Dispositivo RedBear Duo

Además, la integración de los dispositivos *Bluz DK* y *RedBear Duo* supone un valor añadido que podrá ser empleado en futuros proyectos. Es por ello, que se realizará un análisis de las prestaciones para el uso de estos dispositivos en una comunicación BLE con máximo *Throughput*.

1.3 Peticionario

Actúa como peticionario de este TFM la División de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA) de la Universidad de Las Palmas de Gran Canaria (ULPGC), en el contexto de las líneas de investigación promovidas por la citada división. Por otro lado, la realización de un Trabajo Fin de Máster es requisito indispensable para la obtención del título de Máster en Tecnologías de Telecomunicación por el Instituto Universitario de Microelectrónica Aplicada, perteneciente a la Universidad de Las Palmas de Gran Canaria.

1.4 Organización de la memoria

El presente documento está dividido en dos partes claramente diferenciadas: Memoria y Anexo. A su vez, la Memoria está estructurada en 7 capítulos, además de la bibliografía empleada. El contenido de estos capítulos es el resumido a continuación:

- *Capítulo 1. Introducción.* En este capítulo se presentan los motivos que han dado lugar al planteamiento de este Trabajo Fin de Máster (TFM), así como los

objetivos que se pretenden satisfacer durante su desarrollo. Asimismo, se incluye el peticionario y la estructura del documento con el fin de proporcionar una idea global del trabajo realizado.

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

- *Capítulo 3. Análisis del módulo TGAM del dispositivo Mindflex.* En este capítulo se realiza un análisis del dispositivo empleado para el registro de las ondas EEG. Además, se recogen una serie de modificaciones necesarias para adaptarlo a las necesidades de este TFM, y algunas caracterizaciones iniciales de su funcionamiento.

- *Capítulo 4. Análisis de la información de los paquetes ThinkGear del módulo TGAM.* En este capítulo se realiza una primera integración del dispositivo *Mindflex*, en este caso con el dispositivo *RedBear Duo,* con el fin de comprobar la correcta recepción de datos en los diferentes modos de funcionamiento del módulo TGAM. Posteriormente, se comprueba la correcta integración de los dispositivos *RedBear Duo* y *Bluz DK*, para conseguir, en último lugar, la integración de ambos dispositivos con el módulo TGAM.

- *Capítulo 5. Integración de los dispositivos RedBear Duo y Bluz DK mediante BLE.* En este capítulo se recogen los pasos seguidos para lograr la integración final del módulo TGAM con el dispositivo *Bluz DK,* actuando como dispositivo *Peripheral*, que se comunicará mediante el protocolo BLE con el dispositivo *RedBear Duo,* que actuará como dispositivo *Central*.

- *Capítulo 6. Procesamiento de la información EEG para la detección del parpadeo de ojos en el dispositivo RedBear Duo.* En este capítulo se expone el algoritmo propuesto para la detección del parpadeo de los ojos mediante las

señales de EEG registradas por el módulo TGAM en la plataforma HW/SW desarrollada.

- *Capítulo 7. Conclusiones.* Tras haber completado los objetivos propuestos para este Trabajo Fin de Máster, en este capítulo se recogen las conclusiones obtenidas.

La segunda parte del documento se corresponde con el Anexo, que incluye la estructura y el contenido del CD-ROM adjunto.

Capítulo 2. Introducción al estándar BLE

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

2.1 Conceptos básicos de Bluetooth

Bluetooth Low Energy (BLE) se corresponde con la especificación 4.0 del estándar *Bluetooth* desarrollado por *Bluetooth Special Interest Group* (SIG) en el año 2010. Planteado para ser considerado un estándar con objetivos y aplicaciones diferentes a las de su antecesor, BLE es una tecnología enfocada a ofrecer mínimo consumo de potencia, bajo coste, mínimo ancho de banda y reducida complejidad **[8]**.

En BLE, los dispositivos se dividen en dos tipos básicos, denominados dispositivo *Central* y dispositivo *Peripheral*, como se muestra en la Figura 2.1, y entre los cuales se puede establecer una conexión:

- Dispositivo *Central* (*Master*). Escanea repetidamente el medio en busca de paquetes de *Advertising,* e inicia la conexión. Una vez establecida la conexión, es el encargado de controlar el *timing* e iniciar los intercambios periódicos de datos.
- Dispositivo *Peripheral* (*Slave*). Envía periódicamente paquetes de *Advertising* y acepta conexiones. Una vez conectado, este dispositivo sigue el *timing* establecido por el dispositivo *Central,* e intercambia datos regularmente con él en ambas direcciones.

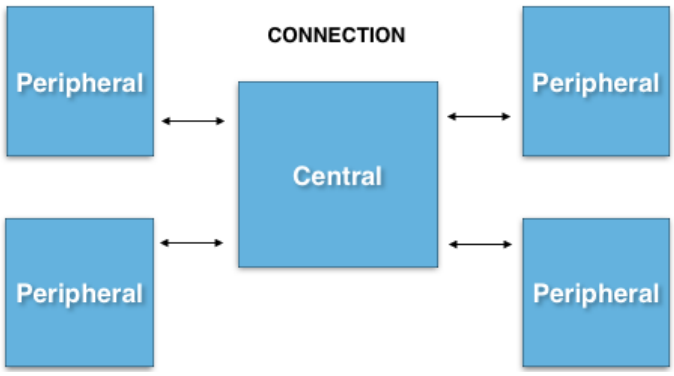


Figura 2.1 - Dispositivos Central y Peripheral

2.1.1 Direcciones de los dispositivos Bluetooth.

La dirección es el identificador esencial de los dispositivos *Bluetooth* (*Bluetooth Device Address*). Se trata de un valor de 6 bytes que permite identificar inequívocamente los dispositivos BLE y que se puede clasificar en dos tipos:

- *Random device address*: Dirección que puede ser programada dinámicamente o preprogramada en el dispositivo.

- *Public device address*: Dirección del dispositivo fija y establecida por el fabricante. Debe ser registrada por *IEEE Registration Authority* y no cambia durante la vida útil del dispositivo.

2.1.2Procedimiento de *Advertising*

En BLE existe un único formato de paquetes que a su vez se divide en dos tipos (de *Advertising* y de datos), que simplifican enormemente el protocolo. Los paquetes de *Advertising* sirven para descubrir dispositivos esclavos y conectarse a ellos, o bien para enviar paquetes de datos en aplicaciones en las que no se requiere de una conexión completa.

Estos paquetes pueden contener hasta 31 bytes de datos de *Advertising*, además de la información básica de cabecera, y son enviados sin el previo conocimiento por parte del dispositivo *Advertiser* de la presencia de un dispositivo *Scanner*. Son enviados a una velocidad fija que puede variar entre 20 milisegundos y 10.4 segundos. Cuanto menor sea este intervalo, mayor es la frecuencia a la que los paquetes de *Advertising* son difundidos, y por tanto, mayor es la probabilidad de que éstos sean recibidos por un dispositivo *Scanner*, aunque mayor es también el consumo de potencia.

BLE usa para sus comunicaciones la banda de 2,4 GHz ISM (*Industrial, Scientific and Medical)*. Esta banda se divide en 40 canales desde 2,4000 GHz hasta 2,4835 GHz. De estos canales, treinta y siente son empleados para la transmisión de datos y los últimos tres de ellos para los mensajes de *Advertising* (canales 37, 38 y 39), como se muestra en la Figura 2.2 **[8]**.

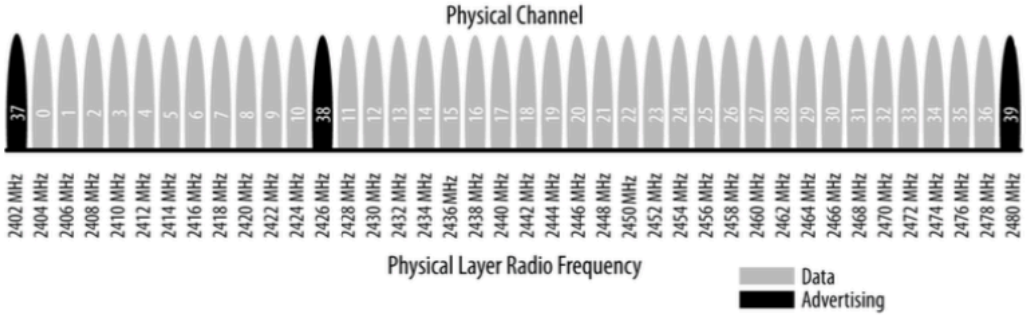


Figura 2.2 - Canales BLE

Los paquetes de *Advertising* se pueden clasificar, según sus propiedades, en los tipos que se muestran en la Figura 2.3, y que se describen a continuación:

Advertising Packet Type	Connectable	Scannable	Directed	GAP Name
ADV_IND	Yes	Yes	No	Connectable Undirected Advertising
ADV_DIRECT_IND	Yes	No	Yes	Connectable Directed Advertising
ADV_NONCONN_IND	No	No	No	Non-connectable Undirected Advertising
ADV_SCAN_IND	No	Yes	No	Scannable Undirected Advertising

Figura 2.3 - Tipos de paquetes de Advertising

- *Connectable Undirected Advertising* (ADV_IND): Es el más común y el más genérico. El dispositivo *Advertiser* puede ser encontrado por cualquier dispositivo *Scanner,* y este último puede iniciar la conexión.
- *Connectable Directed Advertising* (ADV_DIRECT_IND): Se emplea cuando un dispositivo necesita conectarse de manera rápida con un dispositivo *Scanner* concreto. Estos paquetes de *Advertising* deben ser enviados, como máximo, cada 3.75 milisegundos.
- *Nonconnectable Undirected Advertising* (ADV_NONCONN_IND): Es usado por dispositivos que desean emitir datos en modo *Broadcast* sin conectarse a ningún otro dispositivo, ni tan siquiera recibir ninguna información.
- *Scannable Undirected Advertising* (ADV_SCAN_IND): Es similar al anterior, pero en este caso el dispositivo *Scanner* puede obtener datos del dispositivo *Advertiser*, ya que éste responde a cada paquete de tipo *Scan Request* detectado, con un paquete de tipo *Scan Response*.

Por último, los parámetros de *Advertising* de un dispositivo *Peripheral* son los siguientes, representados como referencia en el ejemplo de la Figura 2.4.

```
// BLE peripheral advertising parameters
static advParams_t adv_params = {
  .adv_int_min   = 0x0030,
  .adv_int_max   = 0x0030,
  .adv_type      = BLE_GAP_ADV_TYPE_ADV_IND,
  .dir_addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
  .dir_addr      = {0,0,0,0,0,0},
  .channel_map   = BLE_GAP_ADV_CHANNEL_MAP_ALL,
  .filter_policy = BLE_GAP_ADV_FP_ANY
};
```

Figura 2.4 - Parámetros de la funicón Advertising

- *Advertising_Interval_Min, Advertising_Interval_Max*: Son campos de dos bytes que permitan establecer el valor del intervalo de *Advertising* a un valor fijo. Según la especificación del estándar, el intervalo de *Advertising* para los casos *Nonconnectable Advertising* y *Discoverable Advertising* tiene que ser igual o superior a 100 milisegundos. El valor mínimo y el máximo no tienen por qué coincidir, de esta forma se permite dejar un margen de elección al dispositivo *Scanner* dependiendo de sus otras actividades, como se muestra en la Figura 2.5.

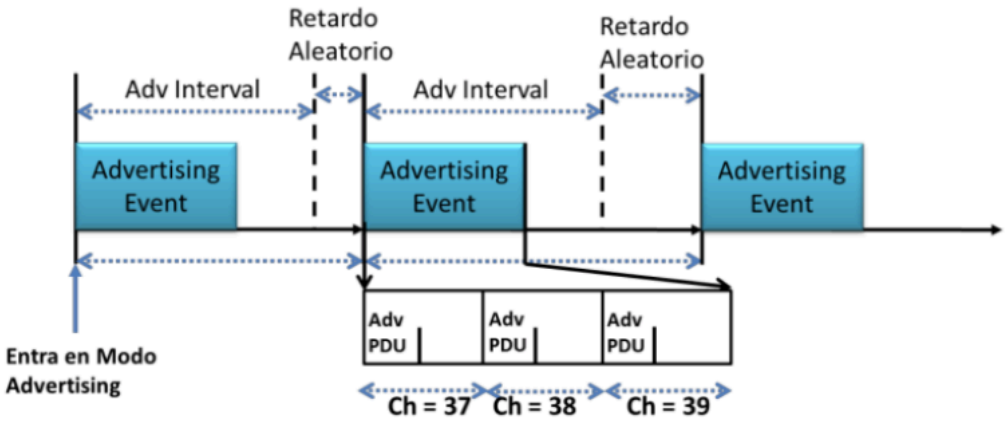


Figura 2.5 - Intervalo de Advertising

- *Advertising_Type*: Es un campo de un byte que define el tipo de *Advertising* para los tipos *Nonconnectable Advertising* y *Discoverable Advertising,* tomando el valor 0x03 y 0x02, en hexadecimal, respectivamente.
- *Own_Address_Type*, *Direct_Address_Type*: Son campos de un byte, que sirven para definir las direcciones como públicas, tomando por defecto el valor 0x00.
- *Direct_Address*: Es un campo de 6 bytes, que se usa para especificar la dirección de destino en los tipos de *Advertising* dirigidos, ignorándose en los casos *Nonconnectable Advertising* y *Discoverable Advertising,* para los cuales su valor es cero.
- *Advertising_Channel_Map*: Es un campo de un byte, que se usa para la elección de los canales de *Advertising*. En el caso de querer enviar los paquetes de *Advertising* en los tres canales de *Advertising,* se especifica el valor 0x07.

- *Advertising_Filter_Policy*: Es un campo de un byte, que indica la política de filtrado de paquetes. Para permitir *Scan Request* y *Connect Request* de cualquier dispositivo, a este campo se le ha de asignar el valor 0x00.

2.1.3 Procedimiento de *Scanning*

En BLE, el estado de *Scanning* se da cuando el dispositivo *Scanner* está pendiente de la recepción de paquetes de *Advertising*. Las especificaciones del estándar definen dos tipos de procedimientos básicos de S*canning*, representados en la Figura 2.6 **[8]**.

- *Passive Scanning*: El dispositivo *Scanner* simplemente está a la espera de recibir paquetes de tipo *Advertising*, mientras que, por su parte, el dispositivo *Advertiser* nunca es consciente del hecho de la llegada, o no, de los paquetes de tipo *Advertising* al dispositivo *Scanner*.
- *Active Scanning*: El dispositivo *Scanner* envía paquetes de tipo *Scan Request* para solicitar información adicional al dispositivo *Advertiser,* que responde con el envío de paquetes de tipo *Scan Response*.

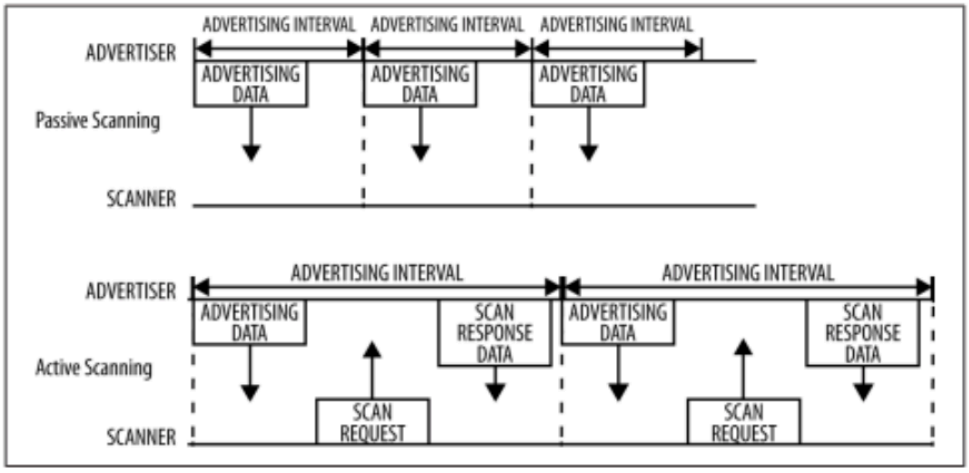


Figura 2.6 - Tipos de procedimientos de Scanning

Además del tipo de *Scanning*, la función de *Scan* establece un conjunto de parámetros, representados en el ejemplo de la Figura 2.7.

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

Figura 2.7 - Parámetros de la función Scanning

- *Scan_Interval*: Intervalo de tiempo que transcurre desde que el dispositivo *Scanner* inicia un *Scan Request,* hasta que inicia el siguiente.

  Rango: 0x0004 to 0x4000.

  Por defecto: 0x0010 (10 ms).

  Tiempo = N * 0.625 milisegundos.

  Rango de tiempos: 2.5 milisegundos a 10.24 segundos.
- *Scan_Window*: Duración del procedimiento de *Scanning*. Debe ser menor o igual al valor del parámetro Scan-Interval.

  Rango: 0x0004 to 0x4000.

  Por defecto: 0x0010 (10 ms).

  Tiempo = N * 0.625 milisegundos.

  Rango de tiempos: 2.5 milisegundos a 10.240 milisegundos.

2.2 UUIDs

BLE emplea identificadores UUID (*Universally Unique Identifier*) para diferentes propósitos, entre ellos la identificación de Servicios y las Características. Un UUID es un valor de 128 bits (16 bytes) que actúa como un identificador único. Las especificaciones de BLE añaden dos formatos de UUID adicionales de 16 y 32 bits que pueden ser usados para UUIDs que estén definidos en las especificaciones de BLE.

2.3 Perfiles genéricos BLE

Los perfiles BLE constituyen las especificaciones de la funcionalidad que definen cómo usar los protocolos con el fin de alcanzar un determinado objetivo, ya sea genérico o específico. Los protocolos, por su parte, son las especificaciones seguidas por todos los dispositivos BLE que representan los niveles que implementan los diferentes formatos de paquetes, *routing, multiplexing, encoding, decoding,* que permiten que los datos sean transferidos de manera efectiva entre dispositivos **[8].**

Así, de entre los perfiles asociados a los modos básicos de operación destacan por su importancia los Perfiles Genéricos (*Generic Profiles*). De entre ellos, las especificaciones definen dos perfiles que son necesarios en todos los dispositivos BLE,

y cuyo conocimiento resulta determinante para asegurar la compatibilidad e interoperabilidad entre dispositivos BLE. Estos dos perfiles son los siguientes:

- GAP (*Generic Access Profile*): El perfil GAP constituye la capa de control de mayor nivel jerárquico de BLE, permitiendo a los dispositivos interoperar entre sí. Permite difundir datos, descubrir dispositivos, establecer y gestionar conexiones y negociar niveles de seguridad. Una vez establecida la conexión, se emplearán los Servicios y Características definidos en el perfil GATT.
- GATT (*Generic Attribute Profile*): El perfil GATT constituye la capa de datos de BLE, estableciendo en detalle cómo se enviarán los datos de usuario y de perfil en una conexión BLE. Especifica modelos de datos y procedimientos para permitir a los dispositivos descubrir, leer, escribir y enviar elementos de datos entre ellos. Estos datos se organizan de forma jerárquica en secciones llamadas Servicios, cada uno de ellos con sus propias Características.

2.4 Perfil GAP

El perfil GAP (*Generic Access Profile*) proporciona un entorno que todo dispositivo BLE debe seguir para posibilitar que los dispositivos se descubran entre sí, puedan difundir datos, establecer conexiones seguras, y realizar otras muchas operaciones fundamentales de manera conocida y estándar. En la especificación del estándar BLE se definen los siguientes aspectos para la interacción entre dispositivos **[8]**:

- *Roles*: Cada dispositivo puede operar en uno o más roles al mismo tiempo. Los roles que se contemplan en el perfil GAP son los siguientes:
    - *Broadcaster*: Optimizado para operaciones de sólo transmisión que difunden datos regularmente. En este rol se envían periódicamente paquetes de *Advertising* con datos.
    - *Observer*. Optimizado para operaciones únicamente de recepción de datos de dispositivos *Broadcaster*. En este rol se reciben datos encapsulados en paquetes de *Advertising* enviados por dispositivos *Broadcaster*.

- *Central*. Los dispositivos *Central* son los encargados de iniciar una conexión. Se trata, por lo general, de dispositivos móviles, como *smartphones* que pueden mantener conexiones con múltiples dispositivos.
- *Peripheral*. Los dispositivos *Peripheral* usan paquetes de *Advertising* para facilitar que los dispositivos *Central* los detecten, y establezcan conexiones con ellos.

- *Modes*: Un modo representa el estado al que un dispositivo puede pasar por un periodo de tiempo para permitir a un *peer* realizar un determinado procedimiento. El paso entre modos de funcionamiento puede activarse mediante acciones del usuario, o de manera automática cuando sea necesario. En la Figura 2.8 se muestran los modos y los procedimientos aplicables a cada uno de ellos.

Mode	Applicable Role(s)	Applicable Peer Procedure(s)
Broadcast	Broadcaster	Observation
Non-discoverable	Peripheral	N/A
Limited discoverable	Peripheral	Limited and General discovery
General discoverable	Peripheral	General discovery
Non-connectable	Peripheral, broadcaster, observer	N/A
Any connectable	Peripheral	Any connection establishment

Figura 2.8 - Modos y procedimientos aplicables

- *Procedures*: Un procedimiento es una secuencia de acciones que permiten a un dispositivo alcanzar un determinado objetivo. Cada procedimiento, por lo general, está asociado a un modo en el otro *peer*. En la Figura 2.9 se muestran los procedimientos y los modos en los que se pueden aplicar.

Procedure	Applicable Role(s)	Applicable Peer Mode(s)
Observation	Observer	Broadcast
Limited discovery	Central	Limited discoverable
General discovery	Central	Limited and General discoverable
Name discovery	Peripheral, central	N/A
Any connection establishment	Central	Any connectable
Connection parameter update	Peripheral, central	N/A
Terminate connection	Peripheral, central	N/A

Figura 2.9 - Procedimientos y modos requeridos

- *Security*: En perfil GAP define una serie de modos y procedimientos que especifican cómo establecen los *peer* el nivel de seguridad necesario para un determinado intercambio de datos, y cómo se garantiza su nivel de seguridad.
- *Additional GAP Data Formats*: El perfil GAP introduce elementos adicionales relevantes para los desarrolladores:
    - *Advertising Data Format*: En las secciones anteriores se han indicado los datos de usuario que pueden incluir los paquetes de *Advertising*, pero no se ha mencionado el formato en el que deben ser difundidos los datos. Este formato consiste en una secuencia de estructuras de datos, cada una de las cuales está formada por *LENGHT* (1 byte), *ADType* (*Advertising Data Type*, 1 byte) y *Data* (*Variable lenght*), como se muestra en la Figura 2.10. Cada estructura de datos contiene un elemento independiente de los datos de usuario.

Name	Actual data length in bytes	Description
Flags	1 (extendable)	Used to set limited or general discovery mode, as described in "Discovery" on page 39
Local Name	variable	Partial or complete user-readable local name in UTF-8
Appearance	2	A 16-bit value describing the type of device sending the advertising packet
TX Power Level	1	The power level in dBm used to transmit the advertising packet, useful to calculate path loss at the observer or central end
Service UUID	variable	A complete or partial list of GATT services offered by the device sending the packet (as a GATT server)
Slave Connection Interval Range	4	A suggestion to the central about the connection interval range that best fits this peripheral
Service Solicitation	variable	A list of GATT services supported by the device sending the packet (as a GATT client)
Service Data	variable	A UUID representing a GATT service and its associated data
Manufacturer Specific Data	variable	Freely formattable data, to be used at the discretion of the implementation

Figura 2.10 – Formato de datos de Advertising

- GAP *Service*: El servicio GAP (*GAP Service*) es un Servicio que obligatoriamente todos los dispositivos BLE deben incluir entre sus atributos. El servicio GAP es accesible para todos los dispositivos conectados, sin requisito alguno de seguridad, y contiene las tres Características siguientes:

- *Device Name*: Esta Característica contiene el mismo identificador que puede incluirse en el parámetro *ADType Local Name* del formato *Additional GAP Data Format*.
- *Appearance*: Este valor de 16 bits asocia el dispositivo con una determinada categoría (*smartphone*, ordenador, …) y típicamente es utilizado por el cliente GATT para representar un icono de la categoría correspondiente. Esta Característica también puede hacerse disponible en el paquete de *Advertising* con el parámetro *ADType Appearance*.
- *Peripheral Preferred Connection Parameters* (PPCP): Una vez que el dispositivo *Central* ha establecido una conexión con un dispositivo *Peripheral*, puede leer el valor de esta Característica y realizar un procedimiento de actualización de parámetros de conexión con el fin de modificar los parámetros de la conexión a los parámetros especificados por el dispositivo *Peripheral*.

2.5 Perfil GATT

El perfil GATT establece en detalle el modo en el que se intercambian los datos de perfil y usuario de las conexiones BLE, tratando con los formatos y los métodos de transmisión de datos. El perfil GATT emplea *Attribute Protocol* (ATT) como su protocolo de transporte para el intercambio de datos entre dispositivos. Establece una jerarquía estricta para organizar los atributos de una manera práctica y reusable, permitiendo que el acceso y la recepción de datos entre dispositivos *Client* y *Server* siga un conjunto de reglas que contribuyan al entorno empleado por todos los perfiles basados en GATT. Así, en el perfil GATT los atributos se organizan jerárquicamente en secciones llamadas Servicios (*Services*), que agrupan datos de usuarios que tienen relación entre sí, llamados Características (*Characteristics*), que a su vez puede tener o no Descriptores (*Descriptors*), como se muestra en la Figura 2.11.

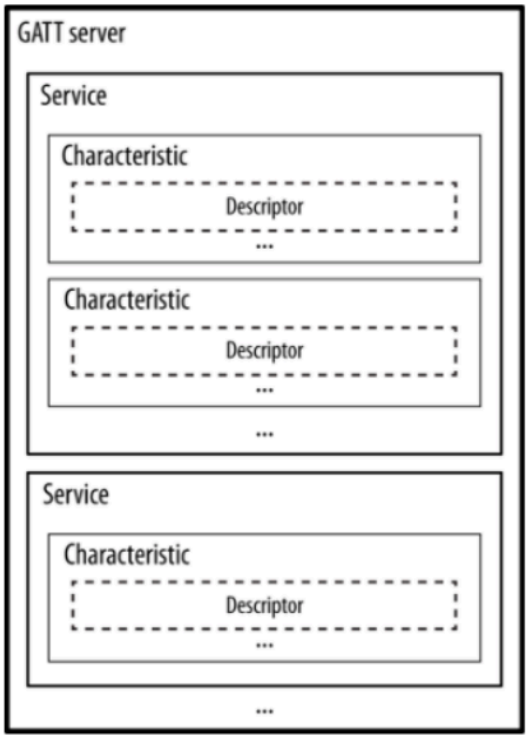


Figura 2.11 - GATT Server (Peripheral)

Por su parte, las Características representan los contenedores que mantienen los datos asociados a un Servicio. Tanto los Servicios como las Características se identifican mediante un identificador único, denominado UUID (*Universally Unique Identifier*).

2.5.1 Características

Las Características del perfil GATT incluyen siempre al menos dos atributos: la Declaración de la Característica (*Characteristic Declaration*), que proporciona metadatos relativos a los datos, y el Valor de la Característica (*Characteristic Value*), que contiene los datos de usuario, como se muestra en la Figura 2.12. Las Características pueden contener también Descriptores adicionales para ampliar los metadatos. Así, la declaración, junto con el valor y cualquier otro descriptor opcional, constituye una Característica.

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

Figura 2.12 - Estructura de una Característica (Declaración y Valor)

El tipo de UUID del atributo de la declaración de una Característica está estandarizado y se corresponde con el valor 0x2803, siendo usado exclusivamente para indicar el comienzo de una Característica. Los diferentes campos concatenados en el atributo *Value* de la Declaración de una Característica incluyen: C*haracteristic Properties*, *Characteristic Value Handle* y *Characteristic* UUID, como se muestra en la Figura 2.13.

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

Figura 2.13 - Atributo Valor de la Declaración de la Característica

El campo *Characteristic Properties,* de 1 byte, indica fundamentalmente, junto con dos bits adicionales en el Descriptor *Extended Properties,* las operaciones y procedimientos que pueden utilizarse con la Característica. Estas propiedades están definidas en la Figura 2.14, codificadas cada una de ellas con 1 bit.

Property	Location	Description
Broadcast	Properties	If set, allows this characteristic value to be placed in advertising packets, using the Service Data AD Type (see "GATT Attribute Data in Advertising Packets")
Read	Properties	If set, allows clients to read this characteristic using any of the ATT read operations listed in "ATT operations"
Write without response	Properties	If set, allows clients to use the Write Command ATT operation on this characteristic (see "ATT operations")
Write	Properties	If set, allows clients to use the Write Request/Response ATT operation on this characteristic (see "ATT operations")
Notify	Properties	If set, allows the server to use the Handle Value Notification ATT operation on this characteristic (see "ATT operations")
Indicate	Properties	If set, allows the server to use the Handle Value Indication/Confirmation ATT operation on this characteristic (see "ATT operations")
Signed Write Command	Properties	If set, allows clients to use the Signed Write Command ATT operation on this characteristic (see "ATT operations")
Queued Write	Extended Properties	If set, allows clients to use the Queued Writes ATT operations on this characteristic (see "ATT operations")
Writable Auxiliaries	Extended Properties	If set, a client can write to the descriptor described in "Characteristic User Description Descriptor"

Figura 2.14 - Propiedades del cliente GATT

El dispositivo *Client* puede leer estas propiedades para determinar qué operaciones pueden realizarse sobre una Característica, lo cual es especialmente importante para las propiedades *Notify* e *Indicate*, ya que estas operaciones son iniciadas por el dispositivo *Server* (*Peripheral Device*) pero requieren su habilitación por parte del

dispositivo *Client* (*Central Device*), en primer lugar mediante el descriptor CCCD (*Client Characteristic Configuration Descriptor*).

Como referencia, las Características definidas como *Read* son leídas por el dispositivo *Client* usando una petición de lectura, siendo el valor recibido como respuesta, el valor de la Característica. Los valores de las Características pueden escribirse mediante peticiones de escritura. El dispositivo *Server* devuelve una confirmación después de que el valor es escrito. Existe una propiedad adicional en la que, cuando el valor de la Característica es escrito mediante un *Write Command*, el dispositivo *Server* no envía ninguna respuesta de vuelta al dispositivo *Client,* por lo que a esta propiedad también se la denomina *Write Without Response*.

Por otro lado, las propiedades adicionales *Notify* e *Indicate* son iniciadas por el dispositivo *Server*. Un dispositivo *Client* se subscribe para ser notificado cuando cambie el valor de una Característica, de manera que cuando se produce un cambio, el dispositivo *Server* se lo notifica al dispositivo *Client* mediante el envío del nuevo valor. La diferencia entre una indicación y una notificación es que en la primera de ellas el dispositivo *Client* debe confirmar la recepción.

Además, cabe mencionar el campo *Characteristic Value Handle*, de 2 bytes, que contiene el valor actual de la Característica. Por otra parte, el campo *Characteristic* UUID indica el UUID de la Característica.

Por último, el atributo *Characteristic Value* de la Característica contiene los datos que el dispositivo *Client* puede leer o escribir para el intercambio de información. El tipo de este atributo es siempre el mismo UUID especificado en el campo *Characteristic* UUID de la declaración de la Característica.

2.5.2 Servicios

Los Servicios GATT agrupan atributos relacionados lógicamente entre sí en una sección común definida en el dispositivo *Server*. Todos los atributos de un Servicio se conocen como *Service Definition* (Definición de Servicio), de manera que los atributos de un

GATT *Server* son una sucesión de definiciones de Servicios, cada uno de ellos comenzando con un atributo que indica el comienzo de un Servicio, denominado *Service Declaration* (Declaración del Servicio), como se representa en la Figura 2.15.

Handle	Type	Permissions	Value	Value length
0xNNNN	$UUID_{primary\ service}$ or $UUID_{secondary\ service}$	Read Only	Service UUID	2, 4, or 16 bytes

Figura 2.15 - Declaración del Servicio

El tipo de UUID del atributo de la Declaración del Servicio está estandarizado y se corresponde con 0x2801, siendo empleado únicamente para indicar el comienzo de un Servicio. El campo *Value* del atributo de la Declaración de un Servicio contiene el UUID del Servicio que introduce la declaración.

En la definición de un Servicio se pueden añadir una o más referencias a otros Servicios mediante *Include definitions*, que consisten en un único atributo (*Include Declaration*) que contiene los detalles necesarios para el dispositivo *Client* con el fin de poder hacer referencia al Servicio incluido, como se muestra en la Figura 2.16.

Handle	Type	Permissions	Value	Value length
0xNNNN	$UUID_{include}$	Read only	Included service handle, end group handle, Included Service UUID	6, 8, or 20 bytes

Figura 2.16 - Include Declaration

De nuevo, el tipo de UUID del atributo de la Declaración del *Include* está estandarizado y se corresponde con el valor 0x2802, siendo usado exclusivamente para indicar el comienzo de un *Include Service*.

2.6 Conexiones

En el contexto del presente TFM, una conexión básica BLE consiste en dos dispositivos (denominados *Central* y *Peripheral*) que se comunican entre sí mediante eventos de comunicación. Para establecer una conexión, el dispositivo Scanner o *Central* comienza realizando un proceso de *Scanning* para localizar dispositivos *Advertiser* o *Peripheral* que acepten una petición de conexión. Para ello, una vez elegido el dispositivo *Advertiser*, el dispositivo *Scanner* le envía una petición de conexión a la que el dispositivo *Advertiser* debe responder para establecer la conexión. El paquete de petición de conexión incluye el incremento de *frequency hop*, que determina la

frecuencia de *hopping* que ambos dispositivos seguirán a lo largo del tiempo de vida de la conexión.

Durante el establecimiento de una conexión, el dispositivo *Central* comunica al dispositivo *Peripheral* un campo adicional de parámetros clave, que son los siguientes:

- *Connection Interval*: Tiempo que transcurre entre el inicio de dos eventos de conexión consecutivos. Este valor puede estar entre 7.5 milisegundos (*high throughput*) y 4 segundos (*lowest throughput*) y se incrementa en saltos de 1.25 milisegundos. En la Figura 2.17 se muestra el ejemplo del envío de múltiples paquetes por intervalo de conexión entre un dispositivo Master (M) y un dispositivo Slave (S) **[9].**

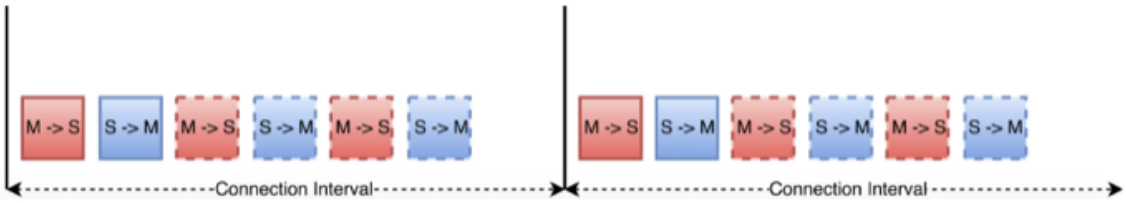


Figura 2.17 – Connection Interval

- *Slave Latency*: Número de eventos de conexión que el dispositivo *Slave* puede ignorar sin arriesgarse a una desconexión.
- *Connection supervisión timeout*: Tiempo máximo entre la recepción de dos paquetes de datos válidos antes de que se considere que la conexión se ha perdido.

En cada intervalo de conexión se pueden intercambiar hasta un máximo de 6 paquetes con una longitud máxima de 20 bytes de datos. Por defecto, en cada intervalo de conexión el dispositivo *Central* y el dispositivo *Peripheral* transmiten un paquete, aunque no tengan datos que enviarse.

La máxima velocidad real de transporte de datos viene definida por la siguiente expresión **[9]**:

$$\text{Throughput} = \frac{1000\text{mSecs} * \text{Number of Packets in a Connection Interval} * \text{Data Per Packet}}{\text{Connection Interval (mSecs)}}$$

2.6.1 Funciones de *Callback*

Las funciones de *Callback* permiten a los dispositivos ser eficientes en lo referente al consumo de energía y se asocian a eventos concretos. De esta forma, los dispositivos pueden encontrarse en modo *Sleep* hasta que uno de estos eventos se produzca, o pueden ser interrumpidos mientras ejecutan el *loop* del programa. BLE soporta diferentes conjuntos de funciones de *Callback,* como pueden ser las siguientes:

- De conexión o desconexión, respondiendo a eventos de conexión o desconexión. Dicha función toma como parámetros el estado de la conexión y un entero sin signo de 16 bits que se asigna a cada conexión (*Connection Handle*). En caso de que la conexión no sea válida, toma el valor 0xFFFF, como se muestra en los ejemplos de la Figura 2.18 y la Figura 2.19 para los procesos de conexión y desconexión, respectivamente.

```
static void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
  switch (status) {
    case BLE_STATUS_OK:
      Serial.print("BLE device connection established! Connection handle: ");
      Serial.println(handle, HEX);
      conn_handle = handle;
      break;
    default:
      Serial.println("Failed to establish connection with peer device!");
      break;
  }
}
```

Figura 2.18 - Función de Callback de conexión

```
static void deviceDisconnectedCallback(uint16_t handle) {
  Serial.print("Disconnected from peer BLE device. Connection handle: ");
  Serial.println(handle, HEX);
  conn_handle = 0xFFFF;
}
```

Figura 2.19 - Función de Callback de desconexión

- De escritura o lectura de datos, respondiendo a eventos de escritura y lectura. Contienen los datos a enviar o leer, así como un *buffer* y el tamaño del mismo.
- Propias del cliente en el perfil GATT. Entre ellas se encuentran funciones que son llamadas cuando se encuentra un nuevo Servicio, una nueva Característica o un nuevo Descriptor de una Característica; funciones de respuesta a lecturas y escrituras del perfil GATT o funciones que notifican eventos del dispositivo

remoto. En general estas funciones tienen como parámetros de entrada el estado y un entero sin signo de 16 bits que funciona como *handle* de la conexión, como se muestra en el ejemplo de la Figura 2.20. Además, pueden obtener también parámetros como el Servicio, la Característica, el Descriptor o *buffers* para lecturas y escrituras.

```
static void gattReadCallback(BLEStatus_t status, uint16_t conn_handle, uint16_t value_handle, uint8_t *value, uint16_
  uint8_t index;
  if (status == BLE_STATUS_OK) {
    Serial.println(" ");
    Serial.println("Reads characteristic value successfully");

    Serial.print("Characteristic value attribute handle: ");
    Serial.println(value_handle, HEX);

    Serial.print("Characteristic value : ");
    for (index = 0; index < length; index++) {
      Serial.print(value[index], HEX);
      Serial.print(" ");
    }
    Serial.println(" ");
  }
  else if (status != BLE_STATUS_DONE) {
    Serial.println("Reads characteristic value failed.");
  }
}
```

Figura 2.20 - Ejemplo de función propia del cliente GATT

- De respuesta a otro tipo de eventos propios de cada aplicación

Capítulo 3. Análisis del módulo TGAM del dispositivo *Mindflex*

En este capítulo se realiza un análisis del dispositivo empleado para el registro de las ondas EEG. Además, se recogen algunas modificaciones necesarias para adaptarlo a las necesidades de este TFM, y algunas caracterizaciones iniciales de su funcionamiento.

3.1 Introducción

El dispositivo disponible para registrar las ondas EEG es el que incluye el producto *Mindflex*, de la empresa *Mattel*, mostrado en la Figura 3.1, con el que es posible controlar la altura que alcanza una bola impulsada por un ventilador, en función de la concentración mental del jugador. El dispositivo *Mindflex* consta de tres electrodos, dos de referencia que han de colocarse en los lóbulos de las orejas, y otro que debe ser situado en la parte superior de la ceja izquierda **[10]**.

Figura 3.1 - Producto Mindflex

El dispositivo de registro de las ondas EEG está basado en el chip *ThinkGear* ASIC (*ThinkGear* ASIC *Module*, TGAM) de la empresa *Neurosky*, representado en la Figura 3.2, que se comunica a través de un enlace TTL con una radio, que a su vez interacciona con otros dispositivos, como por ejemplo la base que se incluye con el producto *Mindflex*. Sin embargo, esta radio, para el caso del producto *Mindflex*, no es compatible con *Bluetooth*, ni por supuesto con BLE. Por tanto, el primer paso consiste en modificar el *hardware* del módulo TGAM en el dispositivo *Mindflex* para poder acceder a la información registrada por éste, y enviarla mediante un enlace serie TTL a la radio, a través de una conexión a los pines *RX/TX* (además de *GND* y *VCC*) del módulo TGAM **[11]**.

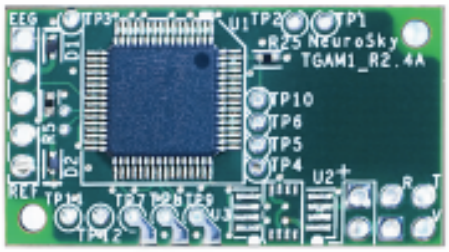
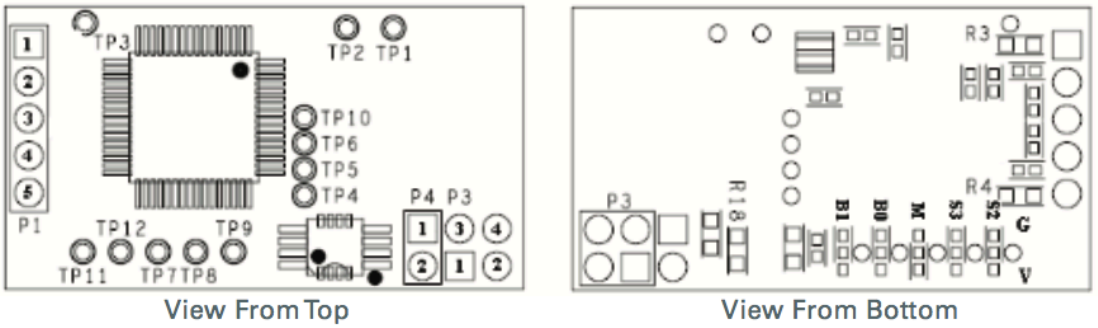


Figura 3.2 - Módulo TGAM

El módulo TGAM constituye el cerebro del sistema *ThinkGear* y es el encargado de generar y procesar los datos relativos a las señales de EEG. Este módulo es ampliamente utilizado, ya que gracias a su bajo consumo de energía resulta adecuado para aplicaciones portátiles. El módulo TGAM contiene el chip TGAT, que se conecta a los electrodos secos y, gracias a su avanzada tecnología de filtrado, ofrece una alta inmunidad frente al ruido. En la Figura 3.3 se muestra el *layout* del módulo TGAM.



Header p1 (Electrode)
Pin 1 - EEG Electrode "EEG"
Pin 2 - EEG Shield
Pin 3 - Ground Electrode
Pin 4 - Reference Shield
Pin 5 - Reference Electrode "REF"

Header p4 (Power)
Pin 1 - VCC "+"
Pin 2 -GND "-"

Header p3 (UART/Serial)
Pin 1 - GND "-"
Pin 2 - VCC "+"
Pin 3 - RXD "R"
Pin 4 - TXD "T"

Figura 3.3 – Layout del módulo TGAM

3.2 Modificaciones HW del dispositivo *Mindflex*

Con el fin de acceder a los datos proporcionados por el módulo TGAM integrado en el dispositivo *Mindflex* se deben realizar una serie de modificaciones *hardware*. Estas modificaciones, proporcionadas por la empresa *Neurosky* **[12]**, tienen la finalidad de permitir su conexión serie con otros dispositivos a través de los pines *RX/TX*. Para ello, en primer lugar, es necesario desatornillar los cuatro tornillos de la tapa de la unidad *Mindflex* que incluye el interruptor de ON/OFF, y no de la unidad que contiene las pilas, como se muestra en la Figura 3.4.

Figura 3.4 - Unidades del dispositivo Mindflex

Una vez desatornillados, se podrá acceder a las diferentes placas de circuito impreso integradas en el dispositivo *Mindflex,* mostradas en la Figura 3.5. Dicho dispositivo incluye una placa principal y dos placas adicionales, de las cuales se debe acceder a la mayor de ellas (correspondiente al módulo TGAM) para realizar las conexiones necesarias. Una vez desatornillada la placa principal de la carcasa, quitando los dos tornillos que incluye, en la placa de mayor tamaño se encuentra el módulo ASIC *ThinkGear* EEG. En este punto, se suelda un cable al pin de transmisión (*T*) y al pin de recepción (*R*) de la placa que contiene el módulo *ThinkGear* EEG. A continuación, será necesario soldar otro cable en la conexión del pin *GND* dispuesto en la placa principal.

Figura 3.5 - Placas del dispositivo Mindflex

Finalmente, y de manera opcional, se suelda un último cable al pin *VCC* localizado en la parte trasera del interruptor de encendido, debiéndose conectar al pin central de este interruptor, no sin antes comprobar que se obtienen aproximadamente 4.5V cuando el

interruptor está en la posición *ON,* y 0V cuando se desconecta en la posición *OFF,* como se muestra en la Figura 3.6.

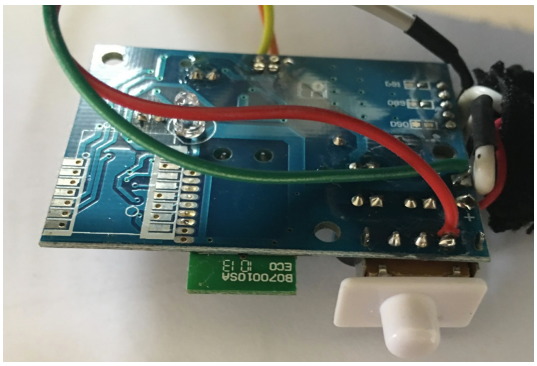


Figura 3.6 - Parte trasera de las placas del dispositivo Mindflex

Se ha de tener en cuenta en el presente documento que el cable soldado al pin *T* del módulo TGAM es de color amarillo, el soldado al pin *R* de color naranja, el soldado al pin *VCC* de color rojo, y el soldado a *GND* de color verde, como se muestra en la Figura 3.7.

Figura 3.7 - Mindflex con modificaciones HW

3.3 Formato de los paquetes *ThinkGear*

Los datos recogidos por el sensor de EEG son enviados como un flujo de datos en serie asíncronos. Dicho flujo de datos es interpretado como paquetes *ThinkGear*, que comienzan con un campo *Header*, seguidos por el campo *Payload* y, por último, finalizan con el byte *Checksum*, como se muestra en la Figura 3.8 **[13]**.

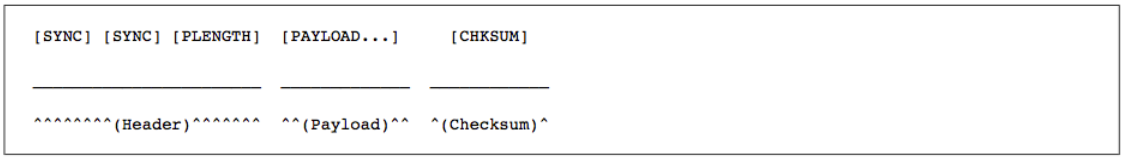


Figura 3.8 - Formato de los paquetes ThinkGear

El tamaño máximo de la sección de *Payload* es 169 bytes, mientras que las secciones *Sync*, *Plength* y *Chksum* del campo *Header* son de tamaño byte, lo que implica que el tamaño mínimo de un paquete *ThinkGear* es de 4 bytes, y el máximo de 173 bytes.

Así, el campo *Header* consta de tres 3 bytes: dos de sincronismo [*SYNC*], con los valores hexadecimales 0xAA 0xAA (170 en decimal) empleados para notificar la llegada de un nuevo paquete de datos, seguidos del byte [*PLENGTH*] que indica la longitud del campo *Payload* y, por tanto, debe tomar valores comprendidos entre 0 y 169.

Por su parte, el campo *Payload* contiene una serie de bytes especificados en [*PLENGTH*], y por último, el campo *Checksum* sirve para verificar la integridad de los datos contenidos en el campo *Payload*. En esta sección de tamaño byte se encuentra el dato correspondiente a los 8 bits menos significativos de la suma de todos los bytes del campo *Payload* tras realizar su complemento a uno.

El campo *Payload* consiste en una serie de valores de datos contenidos en series denominadas *DataRow*. Cada *DataRow* contiene información sobre el tipo de valor que representa, la longitud del mismo, y los bytes de datos, como se muestra en la Figura 3.9 **[13]**.

Figura 3.9 - Contenido del paquete DataRow

Los bytes representados entre paréntesis son opcionales. Por ello, el paquete *DataRow* puede empezar con uno o varios bytes [*EXCODE*], o en su defecto con ninguno. Este valor se emplea, junto con el valor [*CODE*], para determinar el tipo de datos que contiene dicho *DataRow.*

Los valores del campo [*CODE*] comprendidos entre 0x00 y 0x7F implican una longitud del campo [*VALUE*] de un byte, y en este caso no se emplea el campo [*VLENGTH*], como se muestra en la Figura 3.10. Sin embargo, si el campo [*CODE*] tiene un valor superior a 0x7F, el campo [*VLENGTH*] indica el número de bytes que contiene el campo [VALUE], como se muestra en la Figura 3.11 **[13]**.

```
Extended             (Byte)
Code Level  [CODE]  [LENGTH]  Data Value Meaning
----------  ------  --------  ------------------
```

```
        0    0x02         -  POOR_SIGNAL Quality (0-255)
        0    0x03         -  HEART_RATE (0-255)
                             Once/s on EGO.
        0    0x04         -  ATTENTION eSense (0 to 100)
        0    0x05         -  MEDITATION eSense (0 to 100)
        0    0x06         -  8BIT_RAW Wave Value (0-255)
        0    0x07         -  RAW_MARKER Section Start (0)
```

Figura 3.10 - Valores y descripción del campo CODE

```
Extended             (Byte)
Code Level  [CODE]  [LENGTH]  Data Value Meaning
----------  ------  --------  ------------------
        0    0x80         2  RAW Wave Value: a single big-endian
                             16-bit two's-compliment signed value
                             (high-order byte followed by
                             low-order byte) (-32768 to 32767)
        0    0x81        32  EEG_POWER: eight big-endian 4-byte
                             IEEE 754 floating point values
                             representing delta, theta, low-alpha
                             high-alpha, low-beta, high-beta,
                             low-gamma, and mid-gamma EEG band
                             power values
```

```
0     0x83        24  ASIC_EEG_POWER: eight big-endian
                      3-byte unsigned integer values
                      representing delta, theta, low-alpha
                      high-alpha, low-beta, high-beta,
                      low-gamma, and mid-gamma EEG band
                      power values
0     0x86         2  RRINTERVAL: two byte big-endian unsigned
                      integer representing the milliseconds
                      between two R-peaks
```

```
Any   0x55         -  NEVER USED (reserved for [EXCODE])
Any   0xAA         -  NEVER USED (reserved for [SYNC])
```

Figura 3.11 Valores y descripción del campo CODE (valor superior a 0x07)

3.4 Modos de funcionamiento del módulo TGAM

El módulo TGAM es capaz de enviar las señales recogidas por el sensor EEG en dos modos, *NORMAL* y *RAW*. Por defecto, al encender el dispositivo, los datos se transmiten en modo *NORMAL* **[13]**.

3.4.1 Modo *NORMAL*

En el modo NORMAL se obtiene un paquete de datos cada segundo a una tasa de 9600 baudios. En este modo, los datos contenidos en un paquete *ThinkGear* son: la Intensidad de la Señal (*Signal Quality Value*), el nivel de Atención (*Attention*), el nivel de Meditación (*Meditation*), y los valores de las 8 bandas de potencia de EEG, como se muestra en la Figura 3.12. Este paquete *ThinkGear* se identifica con el valor 0x83 del campo [*CODE*].

NORMAL
(8 bits)

0xAA	Sync header
0xAA	Sync header
0x20	Packect lenght (32)
0x02	Signal Quality EXCODE
0x..	Signal Quality Value

0x83	EEG Power EXCODE
0x18	EEG Power length (24)
0x..	EEG Power Value (x8)
0x..	
0x..	
0x04	Attention EXCODE
0x..	Attention Value
0x05	Meditation EXCODE
0x..	Meditation Value
0x..	Checksum

Figura 3.12 - Contenido del paquete de datos en modo NORMAL

Para comprobar que el contenido de los paquetes recibidos desde el módulo TGAM se corresponde con lo esperado según el formato definido, se ha empleado el dispositivo *USB to TTL Serial Cable* de la empresa *Adafruit* **[14]**, mostrado en la Figura 3.13, para conectar en módulo TGAM integrado en el dispositivo *Mindflex*, al puerto USB de un PC.

Figura 3.13 - Cable USB TTL de Adafruit [13]

Para ello se deben realizar las conexiones indicadas en la Tabla 3.1.

TGAM	**Cable USB TTL**
T (Amarillo)	RX (Blanco)
R (Naranja)	TX (Verde)
GND (Verde)	GND (Negro)

VCC (Rojo)	POWER (Rojo)

Tabla 3. 1 - Conexiones Mindflex - USB to TTL

En la Figura 3.14, se resalta en rojo uno de los paquetes proporcionados por el módulo TGAM en modo NORMAL, y se comprueba que el contenido se corresponde con lo esperado según el formato definido por el fabricante. Para ello, es necesario establecer la tasa de recepción a 9600 baudios y mostrar los valores recibidos en formato hexadecimal.

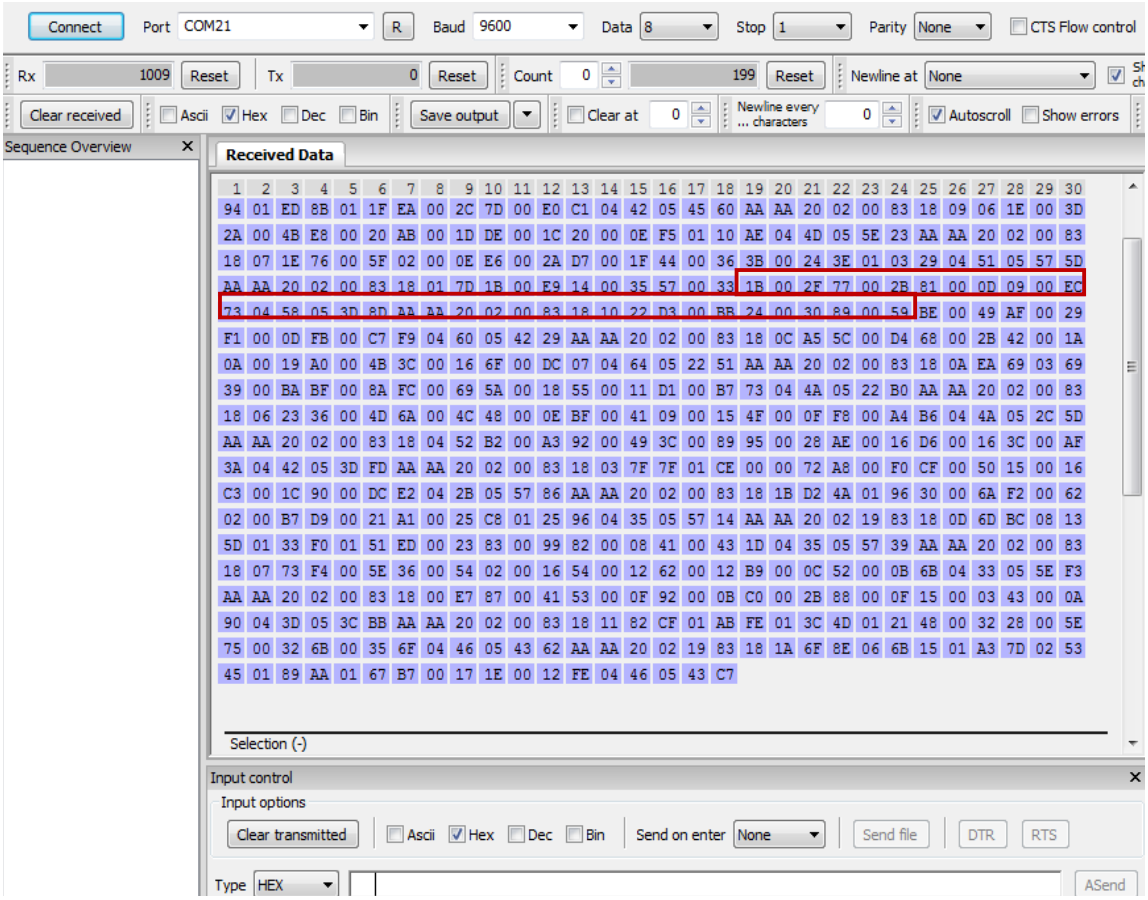


Figura 3.14 - Paquetes recibidos desde el módulo TGAM en modo NORMAL

El parámetro *Signal Quality Value* describe la calidad de la señal medida por el módulo TGAM, es de tipo byte sin signo y puede tomar valores entre 0 y 200. El valor 0 indica la correcta conexión sin ruido, y el valor 200 la falta de señal, debido generalmente a que los electrodos no están en contacto con la piel de la persona.

Los valores *Attention* y *Meditation* varían en el rango de 0 a 100. En dicha escala, los valores comprendidos entre 0 y 20 indican niveles muy bajos de concentración o

meditación, los valores entre 20 y 40 indican niveles reducidos, los valores comprendidos entre 40 y 60 se consideran valores neutros o normales, los niveles entre 60 y 80 indican una mayor concentración o meditación, y en el rango de 80 a 100 se considera una concentración o meditación elevada.

Por último, las bandas de EEG son una representación de la actividad cerebral relativa en diferentes longitudes de onda del cerebro. Se representa con 4 bytes en el siguiente orden: *delta* (0,5 - 2,75 Hz), *theta* (3,5 - 6,75 Hz), *low alpha* (7,5 - 9,25 Hz), *high alpha* (10 - 11,75 Hz), *low beta* (13 - 16,75 Hz), *high beta* (18 - 29,75 Hz), *low gamma* (31 - 39,75 Hz) y *high gamma* (41 - 49,75 Hz).

3.4.2 Modo RAW

El módulo TGAM también es capaz de trabajar en modo RAW. En este caso, el módulo TGAM proporciona un valor de 16 bits (2 bytes) que representan el valor de cada muestra en el rango decimal de -2048 a 2047, y que son transmitidos en una comunicación serie a una tasa de 57600 baudios. Este paquete *ThinkGear*, representado en la Figura 3.15, se identifica con el valor 0x80 del campo [*CODE*].

RAW
(8 bits)

0xAA	Sync header
0xAA	Sync header
0x04	packect lenght
0x80	RAW EXCODE
0x02	RAW value lenght
0x..	RAW value
0x..	RAW value
0x..	CHECKSUM

Figura 3.15 - Contenido del paquete de datos en modo RAW

Para habilitar este modo de funcionamiento en el módulo TGAM existen dos alternativas. La primera de ellas consiste en modificar la disposición de una serie de resistencias SMD localizadas en la placa del módulo TGAM del dispositivo *Mindflex*, de

forma que, en lugar de establecer por defecto el modo NORMAL se establezca por defecto el modo RAW. Sin embargo, con el fin de no modificar el hardware del dispositivo, y mantener su compatibilidad con el juego *Mindflex,* está opción se descartó.

La segunda alternativa consiste en enviar mediante el mismo enlace serie a través del cual se reciben los datos del módulo TGAM, un comando, equivalente a 1 byte. Este comando debe ser transmitido cada vez que se enciende el dispositivo *Mindflex,* ya que, al apagarlo, éste vuelve a su modo de funcionamiento por defecto. Para conocer cuál es el comando a enviar, en primer lugar, se debe conocer la versión del *firmware* correspondiente a la versión del módulo TGAM incluido en el dispositivo *Mindflex*. Para ello se verificó que la versión de este módulo es la versión TGAM1_R2.7, como se observa en la Figura 3.16.

Figura 3.16 - TGAM1_R2.7

Sin embargo, no se pudo encontrar ninguna referencia explícita que indicara la versión correspondiente al *firmware* para este módulo. Ante esta situación, se envió una consulta al servicio de soporte de *Neurosky*, sin obtener respuesta, encontrando de manera experimental que la versión del *firmware* se corresponde con la versión R2.8 del módulo TGAM es la versión 1.7, por lo que se decidió utilizar como referencia inicial la especificación correspondiente a esta versión del *firmware* para el envío del comando que configura el módulo TGAM en modo RAW para transmitir a una tasa de 57600 baudios **[13]**.

Así, como se muestra en la Figura 3.17, para configurar el módulo TGAM en modo RAW, es necesario enviar a través del protocolo de comunicación, el comando 0x02 a la línea *RX* del enlace serie. Sin embargo, en este punto aparece un problema, y es que

este comando debe ser enviado a una tasa de 9600 baudios, mientras que, a partir de su envío, los datos se recibirán desde el módulo TGAM a una tasa de 57600 baudios.

Firmware 1.7 Command Byte Table

```
Page  0 (0000____) (0x0_):  STANDARD/ASIC CONFIG COMMANDS* **

    00000000 (0x00): 9600 baud, normal output mode

    00000001 (0x01): 1200 baud, normal output mode

    00000010 (0x02): 57.6k baud, normal+raw output mode

    00000011 (0x03): 57.6k baud, FFT output mode
```

```
Page  1 (0001____) (0x1_):  RAW WAVE OUTPUT

    bit[0] (____0001): Set/unset to enable/disable raw wave output

    bit[1] (____0010): Set/unset to use 10-bit/8-bit raw wave output

    bit[2] (____0100): Set/unset to enable/disable raw marker output

    bit[3] (____1000): Ignored
```

```
Page  2 (0010____) (0x2_):  MEASUREMENTS OUTPUTS

    bit[0] (____0001): Set/unset to enable/disable poor quality output

    bit[1] (____0010): Set/unset to enable/disable EEG powers (int) output

    bit[2] (____0100): Set/unset to enable/disable EEG powers (legacy/floats) output

    bit[3] (____1000): Set/unset to enable/disable battery output***
```

```
Page  3 (0011____) (0x3_):  ESENSE OUTPUTS

    bit[0] (____0001): Set/unset to enable/disable attention output

    bit[1] (____0010): Set/unset to enable/disable meditation output

    bit[2] (____0100): Ignored

    bit[3] (____1000): Ignored
```

```
Page  6 (0110____) (0x6_):  BAUD RATE SELECTION* **

    01100000 (0x60): No change

    01100001 (0x61): 1200 baud

    01100010 (0x62): 9600 baud

    01100011 (0x63): 57.6k baud
```

Figura 3.17 Command Byte Table

En el proyecto *Mindflex EEG with raw data over Bluetooth* **[15]** se especifica que el envío del comando 0x00 0xF8 0x00 0x00 0x00 0xE0 a una tasa de 57600 baudios, y a continuación realizar una pausa de la longitud del envío de 1 bit a una tasa de 9600

baudios (aproximadamente 1 milisegundo), es equivalente al envío del comando 0x02 a una tasa de 9600 baudios, incluyendo 4 *glitches* y dos bits adicionales iniciales a nivel bajo, teniendo en cuenta que durante la pausa, a través del enlace serie se enviarán bits a nivel alto, de acuerdo con el protocolo RS-232.

Para comprobar la correcta recepción de datos, en primer lugar, se decidió verificar que los datos correspondientes al modo RAW eran enviados correctamente desde el módulo TGAM tras ser configurado éste enviando el comando 0x00 0xF8 0x00 0x00 0x00 0xE0 a una tasa de 57600 baudios. Para ello, se utilizó el cable Serial-USB, a través del cual se pudo conectar el módulo TGAM a uno de los puertos USB del PC con el fin de recibir a través de éste los datos enviados desde el dispositivo *Mindflex*, y por otro lado enviar el comando para configurarlo en modo RAW. Para ello se hizo uso de la aplicación terminal *HTerm*, comprobándose, como se observa en la Figura 3.18, que el dispositivo *Mindflex* envía los datos asociados al modo RAW, con el formato indicado, una vez configurado éste correctamente.

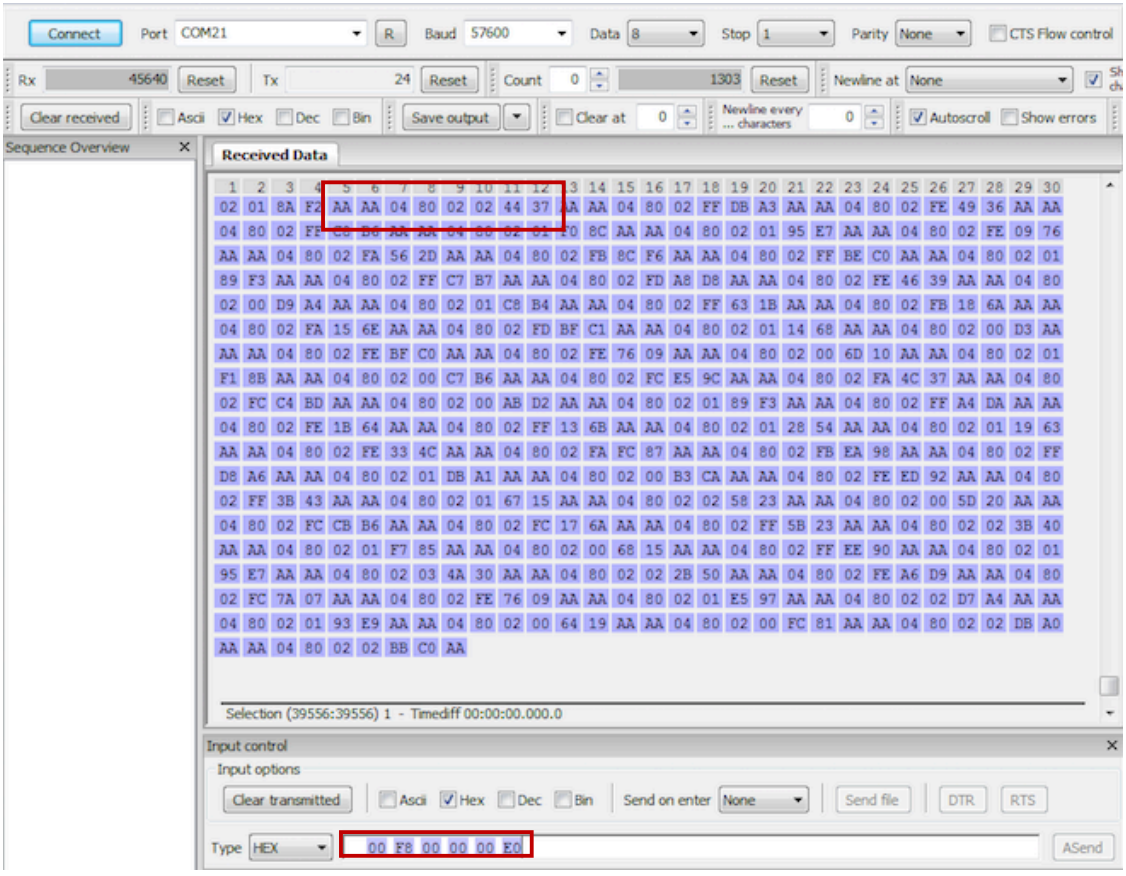


Figura 3.18 - Paquetes recibidos desde el módulo TGAM en modo RAW

Capítulo 4. Análisis de la información de los paquetes *ThinkGear* del módulo TGAM

En este capítulo se realiza una primera integración del módulo TGAM, en este caso con el dispositivo *RedBear Duo,* con el fin de comprobar la correcta recepción de datos, tanto en modo NORMAL, como en modo RAW. Posteriormente, se comprueba la correcta integración de los dispositivos *RedBear Duo* y *Bluz DK* para lograr, en último lugar, la integración de ambos dispositivos con el dispositivo *Mindlfex*.

En este punto se pretende integrar inicialmente el módulo TGAM con el dispositivo *RedBear Duo* para registrar y procesar las ondas EEG en una plataforma de bajo coste basada en un dispositivo *IoT*. Para ello, ambos dispositivos se conectarán mediante una conexión serie y los datos obtenidos se mostrarán en un PC al que estará conectado el dispositivo *RedBear Duo* mediante conexión USB, como se muestra esquemáticamente en la Figura 4.1.

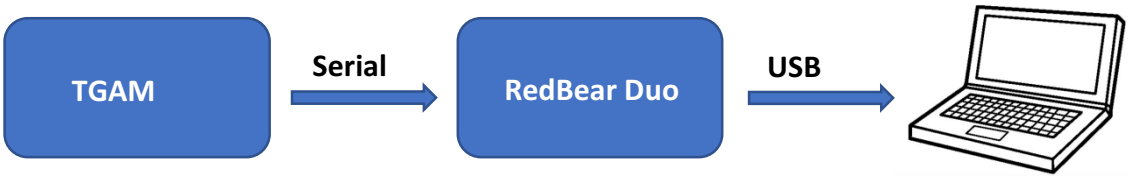


Figura 4.1 - Elementos de la conexión

Con los cuatro cables soldados previamente en el módulo TGAM, ya es posible realizar las conexiones correspondientes a los pines del dispositivo *Redbear Duo*, de manera que, dependiendo de si la alimentación del dispositivo *Redbear Duo* se obtiene del dispositivo *Mindflex,* o a través de un puerto USB, se harán las conexiones indicadas en la Tabla 4.1(a) o en la Tabla 4.1(b), respectivamente.

Mindflex	**Redbear Duo**
T	RX
R (opcional)	TX
GND	GND
VCC	VIN

(a)

Mindflex	**Redbear Duo**
T	RX
R (opcional)	TX
GND	GND

(b)

Tabla 4.1 – Conexiones con alimentación del Redbear Duo mediante el dispositivo Mindflex (a) o mediante USB (b).

4.1 Dispositivo de desarrollo *hardware RedBear Duo*

Antes de comenzar con las pruebas de recepción de paquetes desde el módulo TGAM, se expondrán las especificaciones técnicas básicas del dispositivo *RedBear Duo*, que será empleado para registrar inicialmente la información de EEG. El kit de desarrollo *hardware* de IoT *RedBear Duo* de la empresa *RedBear Lab* **[7]** se ha escogido para el desarrollo de este TFM debido a su bajo coste, su programabilidad, su utilización de

código abierto y por integrar un módulo WiFi y BLE que lo dotan de la conectividad necesaria para llevar a cabo el objetivo propuesto en este TFM. Además, este dispositivo es soportado por la plataforma *Particle IDE* para dispositivos *IoT* **[16]**.

Figura 4.2 - Dispositivo RedBear Duo

El dispositivo *RedBear Duo*, mostrado en la Figura 4.2, combina un potente microcontrolador *ARM Cortex-M3* con el *chip Broadcom* BCM43438 que combina WiFi + BLE. Entre otras características, el dispositivo *RedBear Duo* dispone de 1MB de memoria *Flash* y de 128KB de RAM, además de varios LEDs (*Light-Emitting Diode*) integrados, 18 entradas/salidas de propósito general (GPIO), periféricos avanzados y un sistema operativo en tiempo real (*FreeRTOS*), como se muestra en la Figura 4.3. Además, este dispositivo cuenta con una gran cantidad de interfaces analógicas, digitales y de comunicación como SPI, UART, I2S, I2C, CAN o USB entre otras **[17].**

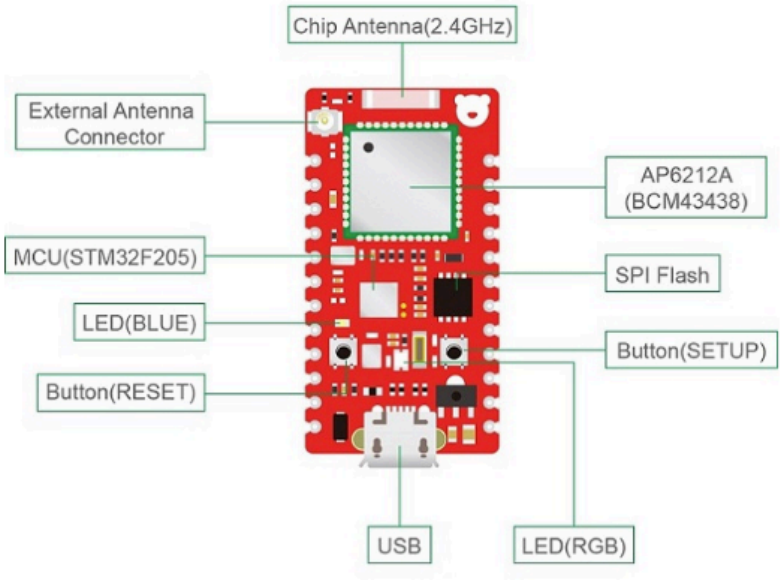


Figura 4.3 - Diagrama de bloques del dispositivo RedBear Duo

La plataforma cuenta con dos botones (*Setup* y *Reset*) que permiten configurar nuevas credenciales WiFi y reiniciar el dispositivo. También se pueden utilizar en conjunto

para provocar el restablecimiento de los valores de fábrica. Entre ambos botones se encuentra un LED de tipo RGB (Red, Green, Blue) que proporciona información acerca del estado del dispositivo. Por ejemplo, si se encuentra conectado a una red WiFi estará parpadeando en color celeste, mientras que si se está cargando un programa en *Flash*, este LED parpadeará rápidamente en color rosa.

En la parte superior del dispositivo *RedBear Duo* se encuentra el puerto micro-USB, cuyo objetivo principal es proporcionar alimentación al dispositivo, aunque también se puede emplear para la programación del dispositivo vía USB y para realizar comunicaciones serie USB con un ordenador. A la derecha del conector micro-USB están situados los pines de *reset* y alimentación (*3V3, RST, VBAT* y *GND*).

El dispositivo *RedBear Duo* convierte la energía de entrada proporcionada a través de la alimentación del puerto micro-USB o del pin *VIN*, en un suministro de 3.3 Voltios, ya que toda la lógica del dispositivo funciona con 3.3V. El pin *RST* se puede utilizar, al igual que el botón *Reset*, para reiniciar el sistema.

En la Figura 4.4 se muestran los pines del dispositivo *RedBear Duo*. El pin *VBAT* permite que una pequeña batería de reserva se conecte a la del dispositivo *RedBear Duo*, con el fin de proporcionarle alimentación mientras éste se encuentra en modo *deep sleep*, conservando así el contenido de su memoria, de forma que cuando vuelva a funcionar en modo normal, pueda continuar en el estado en el que se encontraba previamente.

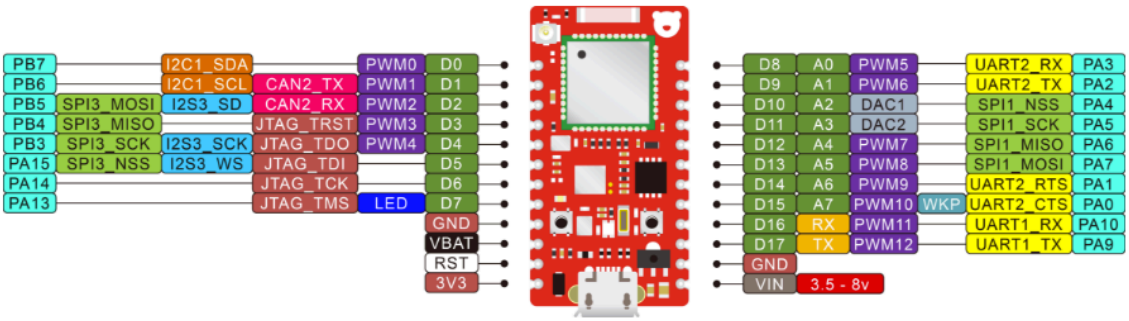


Figura 4.4 - Pinout del dispositivo RedBear Duo

Los pines *D0* a *D7* son pines de propósito general que pueden actuar como entradas o salidas digitales. Además, los pines *D0* a *D4* también pueden actuar como salidas analógicas utilizando técnicas PWM (*Pulse-Width Modulation*). Tal y como se aprecia

en la Figura 4.4, existe también un LED azul situado junto al pin D7 que se encuentra conectado directamente a este pin.

Los pines *A0* a *A7* constituyen entradas analógicas que trabajan con tensiones de entre 0 y 3.3V. Los pines analógicos también se pueden utilizar como entradas o salidas digitales, como los pines *D0* a *D7* y, al igual que los pines digitales, algunos pines analógicos (todos excepto *A2* y *A3*) también se pueden utilizar como salidas analógicas PWM. Además, los pines *A2* y *A3* se corresponden con los pines del conversor analógico-digital (*Digital Analog Converter, DAC*), los cuales son pines de salida analógica especial, capaz de proporcionar voltajes comprendidos entre 0 y 3.3V. A continuación, en el pin de salida analógica *A7* se dispone el pin *WKP*, el cual se utiliza para despertar al dispositivo *RedBear Duo* después de que se haya puesto en modo *deep sleep*.

Los pines *TX* y *RX* (Transmisión y Recepción, respectivamente) se utilizan para la comunicación serie. Por último, situados por debajo de estos pines se encuentran, un segundo pin *GND,* y el pin *VIN*. Tal y como se comentó anteriormente, se puede alimentar al dispositivo *RedBear Duo* suministrando entre 3.6V y 5.5V al pin *VIN*, como alternativa al uso del puerto USB.

4.2 Recepción de datos EEG en el dispositivo *RedBear Duo*

En primer lugar, en este apartado se analizaron los ficheros que conforman la librería *Brain Library* **[18]**, a través de la cual se realiza la transferencia de los datos capturados por el módulo TGAM a una plataforma *Arduino*, aunque únicamente en modo *NORMAL.* Tomando como referencia inicial esta librería, y una vez realizadas las modificaciones HW sobre el módulo TGAM integrado en el dispositivo *Mindflex*, el siguiente paso consiste en desarrollar el *firmware* para el dispositivo *RedBear Duo* que permitirá analizar, extraer y registrar la información proporcionada por el módulo TGAM a través del enlace serie. Los ficheros de la librería desarrollada en este TFM se han denominado *Parser.cpp* y *Parser.h*, y su implementación se basa en la información recogida en el documento *ThinkGear Communication Protocol* en el que se especifica, como se ha mencionado con anterioridad en este TFM, el formato de los paquetes

ThinkGear transferidos por el módulo TGAM, tanto en modo NORMAL, como en modo RAW.

En el fichero *Parser.cpp* desarrollado en este TFM, en primer lugar, se encuentra la función *Parser(),* mostrada en la Figura 4.5, que recibe por parámetros el flujo de datos, la variable denominada *modeR*, que a nivel bajo deja el módulo TGAM con la configuración por defecto (9600 baudios + modo NORMAL), y a nivel alto lo configura en modo RAW (57600 baudios + modo RAW), y la variable *modeD*, que a nivel alto proporciona por el terminal de pantalla información completa de los datos recibidos desde el módulo TGAM, y a nivel bajo genera únicamente los valores correspondientes a las muestras en modo RAW. Por otro lado, la función *init()* inicializa el valor de las variables utilizadas, como se muestra en la Figura 4.5.

```
#include "application.h"
#include "Parser.h"

Parser::Parser(Stream &_ParserStream, boolean modeR, boolean modeD) {
    ParserStream = &_ParserStream;
    init(modeR, modeD);
}

void Parser::init(boolean modeR, boolean modeD) {
    freshPacket = false;
    inPacket = false;
    packetIndex = 0;
    packetLength = 0;
    eegPowerLength = 0;
    hasPower = false;
    checksum = 0;
    checksumAccumulator = 0;
    signalQuality = 200;
    attention = 0;
    meditation = 0;
    rawValue = 0;
    modeRaw = modeR;
    modeDebug = modeD;
    npacketError = 0;
    npacketOK = 0;
    clearEegPower();
    lastmills = 0;
}
```

Figura 4.5 - Funciones Parser() e init()

Posteriormente, en el código del fichero *Parser.cpp,* la función *update()* lee *byte* a *byte* los datos recibidos vía serie en busca de los *bytes* de sincronismo (0xAA, 0xAA) equivalentes al valor decimal 170, que marcan el inicio de la recepción de un paquete *ThinkGear,* comprueba que la longitud del paquete se ajusta a la longitud máxima permitida, y si es así, realiza el cálculo del *Checksum* y se procesa el paquete, en caso de que éste sea correcto. Este procesamiento se realiza mediante la función

parsePacket(), que será descrita posteriormente, en función del tipo de paquete recibido.

En la función *update()*, como se puede observar en la Figura 4.6, inicialmente se implementó la sentencia *if(Serial1.available())* que fue sustituida por la sentencia *while(Serial1.available(), línea 31).* Gracias a esta modificación se corrigió un problema encontrado al habilitar el modo RAW, y que residía en que el *buffer* de la interfaz serie *Serial1* es de 64 *bytes*, y tal como estaba implementada la comprobación de que hubiera datos disponibles en *Serial1*, se leían los *bytes* cada *loop*, lo cual implica una vez cada milisegundo, por lo que no se leía lo suficientemente rápido, lo que provoca una situación de *overflow*. Además, entre las líneas 95 y 105, se implementa el envío del comando 0x00 0xF8 0x00 0x00 0x00 0xE0 a una tasa de 57600 baudios con el fin de configurar el módulo TGAM en modo RAW.

```
29
30  boolean Parser::update() {
31      while (ParserStream->available()) {
32          latestByte = ParserStream->read();
33          if (inPacket) {
34              if (packetIndex == 0) {
35                  packetLength = latestByte;
36                  if (packetLength > MAX_PACKET_LENGTH) {
37                      npacketError++;
38                      if (modeDebug) {
39                          Serial.print("ERR Packet too long: ");
40                          Serial.print(packetLength);
41                          Serial.print(" (");
42                          Serial.print(npacketError);
43                          Serial.println(")");
44                      }
45                      inPacket = false;
46                  }
47                  packetIndex++;
48              }
49              else if (packetIndex <= packetLength) {
50                  packetData[packetIndex - 1] = latestByte;
51                  checksumAccumulator += latestByte;
52                  packetIndex++;
53              }
54              else if (packetIndex > packetLength) {
55                  checksum = latestByte;
56                  checksumAccumulator = 255 - checksumAccumulator;
57
58                  if (checksum == checksumAccumulator) {
59                      boolean parseSuccess = parsePacket();
60
61                      if (parseSuccess) {
62                          freshPacket = true;
63                          FirstPacketReceived++;
64                          npacketOK++;
65                          if (modeDebug)
66                              printDebug();
67                          else
68                              Serial.println(readCSV());
69                      }
70                      else {
71                          npacketError++;
```

```
                        if (modeDebug) {
                            Serial.println("ERR Could not parse");
                            Serial.print(" (");
                            Serial.print(npacketError);
                            Serial.println(")");
                        }
                    }
                }
                else {
                    npacketError++;
                    if (modeDebug) {
                        Serial.print("ERR Checksum: ");
                        Serial.print(checksum);
                        Serial.print(" ");
                        Serial.println(checksumAccumulator);
                        Serial.print(" (");
                        Serial.print(npacketError);
                        Serial.println(")");
                    }
                }
                inPacket = false;
            }
        }
        if ((latestByte == 224) && (lastByte == 224) && !inPacket && modeRaw &&
         !CommandSent) {
                const char command[] = {0x00, 0xF8, 0x00, 0x00, 0x00, 0xE0};
                for (int j = 0; j < sizeof(command); j++)
                    ParserStream->write(command[j]);
                CommandSent = true;
                if (modeDebug)
                    Serial.println(">>> Command Byte for 57.6kbaud, normal + 16bits
                    raw mode SENT");
                delay(2000);
            }
        if ((latestByte == 170) && (lastByte == 170) && !inPacket) {
            inPacket = true;
            packetIndex = 0;
            checksumAccumulator = 0;
        }
        lastByte = latestByte;
    }

    if (freshPacket) {
        freshPacket = false;
        return true;
    }
    else {
        return false;
    }

}
```

Figura 4.6 - Función update()

La función *parsePacket()*, mostrada en la Figura 4.7, separa los datos en sus correspondientes componentes según se trate de modo NORMAL o de modo RAW, tomando para ello el valor del *EXCODE* asociado.

```
boolean Parser::parsePacket() {
    hasPower = false;
    hasRaw = false;
    boolean parseSuccess = true;

    clearEegPower();

    for (uint8_t i = 0; i < packetLength; i++) {
        switch (packetData[i]) {
            case 0x2:
                signalQuality = packetData[++i];
                break;
            case 0x4:
                attention = packetData[++i];
                break;
```

```
            case 0x5:
                meditation = packetData[++i];
                break;
            case 0x6:
                rawValue = ((int)packetData[++i]);
                break;
            case 0x83:
                i++;
                for (int j = 0; j < EEG_POWER_BANDS; j++) {
                    eegPower[j] = ((uint32_t)packetData[++i] << 16) |
                    ((uint32_t)packetData[++i] << 8) | (uint32_t)packetData[++i];
                }
                hasPower = true;
                break;
            case 0x80:
                i++;
                rawValue = ((int)packetData[++i] << 8) | packetData[++i];
                if (rawValue >= 32768)
                    rawValue = rawValue - 65536;
                hasRaw = true;
                break;
            default:
                parseSuccess = false;
                break;
        }
    }
    return parseSuccess;
}
```

Figura 4.7 - Función parsePacket()

A continuación, la función *readCSV(),* mostrada en el código de la Figura 4.8*,* crea un *buffer* con los valores separados por coma de los datos EEG y los muestra por pantalla, haciendo distinción de los datos en modo RAW y modo NORMAL, y de si estos últimos contienen o no el valor de las bandas EEG.

```
char* Parser::readCSV() {
    if (modeRaw && hasRaw) {
        sprintf(csvBuffer,"%d",
            rawValue
        );
        return csvBuffer;
    }
    else if (!modeRaw) {
        if (hasPower) {
            sprintf(csvBuffer,"%d,%d,%d,%lu,%lu,%lu,%lu,%lu,%lu,%lu,%lu",
                signalQuality,
                attention,
                meditation,
                eegPower[0],
                eegPower[1],
                eegPower[2],
                eegPower[3],
                eegPower[4],
                eegPower[5],
                eegPower[6],
                eegPower[7]
            );
            return csvBuffer;
        }
        else {
            sprintf(csvBuffer,"%d,%d,%d",
                signalQuality,
                attention,
                meditation
            );
        return csvBuffer;
        }
    }
}
```

Figura 4.8 - Función readCSV()

Por su parte, la función *printPacket()*, se encarga de mostrar por pantalla el contenido del paquete de datos, como se muestra en el código de la Figura 4.9.

```
204 void Parser::printPacket() {
205     Serial.print("[");
206     for (uint8_t i = 0; i < MAX_PACKET_LENGTH; i++) {
207         Serial.print(packetData[i], DEC);
208 
209             if (i < MAX_PACKET_LENGTH - 1) {
210                 Serial.print(", ");
211             }
212     }
213     Serial.println("]");
214 }
215 
```

Figura 4.9 - Función printPacket()

Por último, la función *printDebug()* es llamada cuando el valor de la variable *modeD* se encuentra a *true*, y se utiliza para imprimir la información EEG relativa a cada paquete de datos, indicando el inicio y el tiempo transcurrido entre la recepción de los paquetes de datos, como se muestra en el código de la Figura 4.10.

```
216 void Parser::printDebug() {
217     Serial.println("");
218     Serial.print("--- Start Packet (");
219     Serial.print(npacketOK);
220     Serial.println(")");
221     Serial.print("    Lapsed Time (ms): ");
222     Serial.println(millis()-lastmills);
223     lastmills = millis();
224 
225     if (!hasRaw) {
226         Serial.print("    Signal Quality: ");
227         Serial.println(signalQuality, DEC);
228         Serial.print("    Attention: ");
229         Serial.println(attention, DEC);
230         Serial.print("    Meditation: ");
231         Serial.println(meditation, DEC);
232     }
233 
234     if (!hasRaw && hasPower) {
235         Serial.println("    EEG POWER:");
236         Serial.print("        Delta: ");
237         Serial.println(eegPower[0], DEC);
238         Serial.print("        Theta: ");
239         Serial.println(eegPower[1], DEC);
240         Serial.print("        Low Alpha: ");
241         Serial.println(eegPower[2], DEC);
242         Serial.print("        High Alpha: ");
243         Serial.println(eegPower[3], DEC);
244         Serial.print("        Low Beta: ");
245         Serial.println(eegPower[4], DEC);
246         Serial.print("        High Beta: ");
247         Serial.println(eegPower[5], DEC);
248         Serial.print("        Low Gamma: ");
249         Serial.println(eegPower[6], DEC);
250         Serial.print("        Mid Gamma: ");
251         Serial.println(eegPower[7], DEC);
252     }
253     if (hasRaw) {
254         Serial.print("    Raw Value: ");
255         Serial.println(rawValue, DEC);
256     }
257 
258     Serial.print("--- End   Packet (");
259     Serial.print(npacketOK);
260     Serial.println(")");
261 }
```

Figura 4.10 - Función printDebug()

Por su parte, el fichero *Parser.h*, mostrado en el código de la Figura 4.11, contiene las declaraciones de constantes, variables y funciones.

```
#ifndef Parser_h
#define Parser_h

#include "application.h"

#define MAX_PACKET_LENGTH 169
#define EEG_POWER_BANDS 8

class Parser {
    public:
        Parser(Stream &_ParserStream, boolean modeR, boolean modeD);
        boolean update();

        char* readCSV();

        uint8_t readSignalQuality();
        uint8_t readAttention();
        uint8_t readMeditation();
        uint32_t* readPowerArray();
        uint32_t readDelta();
        uint32_t readTheta();
        uint32_t readLowAlpha();
        uint32_t readHighAlpha();
        uint32_t readLowBeta();
        uint32_t readHighBeta();
        uint32_t readLowGamma();
        uint32_t readMidGamma();

    private:
        Stream* ParserStream;
        uint8_t packetData[MAX_PACKET_LENGTH];
        boolean inPacket;
        uint8_t latestByte;
        uint8_t lastByte = 'c';
        uint8_t packetIndex;
        uint8_t packetLength;
        uint8_t checksum;
        uint8_t checksumAccumulator;
        uint8_t eegPowerLength;
        boolean hasPower;
        boolean hasRaw;
        unsigned long lastmills;

        int npacketError;
        int npacketOK;

        boolean parsePacket();

        void printPacket();
        void init(boolean modeR, boolean modeD);
        void printCSV();
        void printDebug();

        char csvBuffer[256];

        char latestError[256];

        uint8_t signalQuality;
        uint8_t attention;
        uint8_t meditation;
        int16_t rawValue;
        boolean modeRaw;
        boolean modeDebug;
        uint8_t FirstPacketReceived = 0;
        boolean CommandSent = false;
        boolean freshPacket;

        uint32_t eegPower[EEG_POWER_BANDS];
};

#endif
```

Figura 4.11 - Fichero Parser.h

Una vez implementado el código necesario para realizar este análisis, se incluyeron los ficheros *Parser.cpp* y *Parser.h* en la creación de una *app* en el entorno *Particle* IDE. Además, también se generó el fichero *BrainSerialTest-DuoV05.ino*, mostrado en la Figura 4.12, que permite recibir los paquetes de datos. En este fichero, cabe destacar que el objeto que se le pasa a la función *Parser* debe ser *Serial1*, ya que es el correspondiente a la interfaz serie asociada a los pines *TX/RX* del dispositivo *RedBear Duo*.

```
1  #include "Parser.h"
2  #include "application.h"
3
4  boolean modeDebug = false;
5  boolean modeRaw = false;
6
7  Parser parser(Serial1, modeRaw, modeDebug);
8
9  void setup() {
10     if (modeRaw) {
11         Serial1.begin(57600);
12         Serial.begin(57600);
13     }
14     else {
15         Serial1.begin(9600);
16         Serial.begin(9600);
17     }
18 }
19
20 void loop() {
21     parser.update();
22 }
```

Figura 4.12 - Fichero BrainSerialTest-DuoV05.ino

Por último, tras completar este código fue posible compilar el *firmware* y verificar su funcionamiento inicial básico en el dispositivo *RedBear Duo*, tanto en modo NORMAL como en modo RAW. Para ello unicamente fue necesario configurar la variable *modeRaw* a *false* para activar el modo NORMAL, como se muestra en la línea 5 de la Figura 4.12, o en caso contrario, establecer el valor de dicha variable a *true* para obtener los valores en modo RAW.

De este modo, se mostrarán por el terminal filas con los valores obtenidos para cada muestra, separados por comas, en el caso de los datos en modo NORMAL, como se puede observar en la Figura 4.13.

```
Script started on Mon Jul 10 22:16:22 2017
200,0,0,772933,1425224,51191,320881,324173,823503,47263,23514
200,0,0,127308,1161109,29237,343457,127572,280977,113332,736611
200,0,0,1454361,866382,389011,491518,309408,231564,77612,325987
200,0,0,425795,1042429,11303,390631,388071,376856,158423,1067928
200,0,0,514152,706072,334733,348608,61854,358317,158604,432986
51,0,0,847981,2402597,104629,255687,125806,149454,95393,88655
51,0,0,566184,67347,18876,15011,21466,28857,62762,18395
0,0,0,50898,17838,112,734,352,872,350,869
0,0,0,515124,22695,3573,3103,3248,2416,2772,2993
```

```
0,0,0,193214,62638,29588,11990,6664,13191,9335,10930
0,53,43,88689,14426,6350,6947,11116,40967,16903,55403
0,56,56,7158,32756,8659,25395,15379,23821,20880,42239
0,56,66,625888,176642,56277,26149,8344,42829,31734,37008
0,53,63,177653,14098,919,552,1245,1828,1507,2400
0,44,69,4530,34814,8306,21171,14530,27820,17248,38639
0,43,69,23182,48842,17249,14672,4054,17347,15364,32097
0,48,60,1181014,100333,12493,11621,15225,50560,13230,36323
0,56,66,155551,11792,492,3229,4620,5880,3187,9296
0,48,61,313652,163454,12911,25226,24206,24162,11471,27796
0,53,56,79812,39338,3219,18718,15795,28846,15271,39923
0,54,54,530924,36968,4235,6879,21240,24702,11233,37838
0,53,63,257559,20871,14583,3487,6955,10582,5286,8022
0,57,77,120508,8859,20760,6508,12160,9450,5039,5319
0,60,74,318075,57873,3527,12738,9498,45689,34980,31055
0,54,83,85500,23878,5765,11791,7556,12130,8925,9574
0,53,74,90253,113291,16118,5706,18286,24694,28470,32611
0,47,60,440592,134704,18684,15844,10236,17727,24889,19961
0,44,69,2064341,52289,32246,18687,36376,34171,27843,15758
0,43,64,65857,13027,3386,1472,831,3825,1132,1336
0,38,75,34767,18951,6231,3939,1515,3027,530,954
27,38,75,1084188,27307,2659,2346,2413,1422,1067,1693
0,37,80,375066,308643,143899,14110,26843,28896,22077,29659
51,37,80,1471089,382111,95398,44171,41574,68714,77091,33107
```

Figura 4.13 - Datos obtenidos en modo normal

Del mismo modo, en caso de configurar la variable *modeRaw* a *true*, se mostrarán por el terminal cada uno de los valores obtenidos en modo RAW, como se observa en la Figura 4.14.

```
Script started on Mon Jul 10 22:15:13 2017
-2048
-2048
-291
-37
-50
-193
-470
-610
-452
-317
-361
-659
-903
-946
-978
-942
-954
-948
-861
-742
-826
-967
-957
-904
-881
```

Figura 4.14 - Datos obtenidos en modo RAW

Capítulo 5. Integración de los dispositivos *RedBear Duo* y *Bluz DK* mediante BLE

En este capítulo se recogen los pasos seguidos para lograr la integración final del módulo TGAM con el dispositivo *Bluz DK* actuando como dispositivo *Peripheral*, que se comunicará mediante el protocolo BLE con el dispositivo *RedBear Duo,* que actuará como dispositivo *Central*.

5.1 *RedBear Duo* como dispositivo *Central*

En primer lugar, se presentan los resultados obtenidos a partir del análisis inicial realizado sobre el dispositivo *RedBear Duo* con el objetivo de validar una posible conexión BLE, actuando éste como dispositivo *Central* en la recepción de datos.

Para ello, inicialmente se implementó la conexión entre el dispositivo *RedBear Duo* actuando como dispositivo *Central* y un *Virtual Peripheral* actuando como dispositivo *Peripheral*, definido en la *app Lightblue* para iOS, con el fin de validar el correcto funcionamiento del código correspondiente al dispositivo *RedBear Duo* actuando como dispositivo *Central*, y con ello, los procedimientos de *Scanning*, conexión y descubrimiento de los Servicios y Características definidos, así como la realización de diferentes acciones de *READ/WRITE* sobre el dispositivo *RedBear Duo*.

En este caso se aplicarán los conceptos básicos del estándar BLE para la implementación de las funciones básicas de un dispositivo *Central*. Con esta finalidad se creó el fichero *simpleblecentral_duo.ino*, que logra la compatibilidad del dispositivo *RedBear Duo* con las especificaciones del dispositivo *Virtual Peripheral* que crea la *app Lightblue* **[20]**, y el fichero *bstack_hal_define.h* de la librería BTStack **[21]**, en el que se definen las constantes asociadas a los diferentes parámetros del estándar BLE en el *firmware* del dispositivo *RedBear Duo*.

Así, en la función *reportCallback()* asociada al proceso de *Scanning* del dispositivo *Central* se establece que solo se conectará a un dispositivo *Peripheral* cuyo parámetro *Short Local Name* (asociado al *AD_TYPE*: BLE_GAP_AD_TYPE_SHORT_LOCAL_NAME = 0x08 en los datos de los paquetes *Advertising* enviados por el dispositivo *Peripheral*, tal como se indica en el fichero *bstack_hal_define.h*) sea “Biscuit”. Sin embargo, en el dispositivo *Virtual Peripheral* de la app *Lightblue*, el nombre que puede especificarse en un dispositivo de tipo *Blank*, se corresponde con el parámetro *Complete Local Name* (asociado al *AD_TYPE*: BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME = 0x09), por lo que en caso de especificarse el *Complete Local Name*, se indica que el dispositivo *Central* solo se conectará a un dispositivo *Peripheral* cuyo *Complete Local Name* sea “*Biscuit*”, como se muestra en el código de la Figura 5.1.

```
75  void reportCallback(advertisementReport_t *report) {
76    uint8_t index;
77
78    Serial.println("reportCallback: ");
79    Serial.print("The advEventType: ");
80    Serial.println(report->advEventType, HEX);
81    Serial.print("The peerAddrType: ");
82    Serial.println(report->peerAddrType, HEX);
83    Serial.print("The peerAddr: ");
84    for (index = 0; index < 6; index++) {
85      Serial.print(report->peerAddr[index], HEX);
86      Serial.print(" ");
87    }
88    Serial.println(" ");
89
90    Serial.print("The rssi: ");
91    Serial.println(report->rssi, DEC);
92
93    Serial.print("The ADV data: ");
94    for (index = 0; index < report->advDataLen; index++) {
95      Serial.print(report->advData[index], HEX);
96      Serial.print(" ");
97    }
98    Serial.println(" ");
99    Serial.println(" ");
100
101   uint8_t len;
102   uint8_t adv_name[31];
103   if (0x00 == ble_advdata_decode(0x08, report->advDataLen, report->advData, &len,
104       adv_name)) {
105     Serial.print("  The length of Short Local Name : ");
106     Serial.println(len, HEX);
107     Serial.print("  The Short Local Name is        : ");
108     Serial.println((const char *)adv_name);
109     if (0x00 == memcmp(adv_name, "Biscuit", min(7, len))) {
110       ble.stopScanning();
111       device.addr_type = report->peerAddrType;
112       memcpy(device.addr, report->peerAddr, 6);
113       ble.connect(report->peerAddr, BD_ADDR_TYPE_LE_RANDOM);
114     }
115   }
116   else if (0x00 == ble_advdata_decode(0x09, report->advDataLen, report->advData, &len,
117             adv_name)) {
118     Serial.print("  The length of Complete Local Name : ");
119     Serial.println(len, HEX);
120     Serial.print("  The Complete Local Name is        : ");
121     Serial.println((const char *)adv_name);
122     if (0x00 == memcmp(adv_name, "Biscuit", min(7, len))) {
123       ble.stopScanning();
124       device.addr_type = report->peerAddrType;
125       memcpy(device.addr, report->peerAddr, 6);
126       ble.connect(report->peerAddr, {BD_ADDR_TYPE_LE_RANDOM});
127     }
128   }
129 }
```

Figura 5.1 – Función reportCallback()

Por tanto, se crea en la *app LightBlue* un dispositivo *Virtual Peripheral* con el nombre "*Biscuit*", inicialmente con un único Servicio, como se muestra en la Figura 5.2.

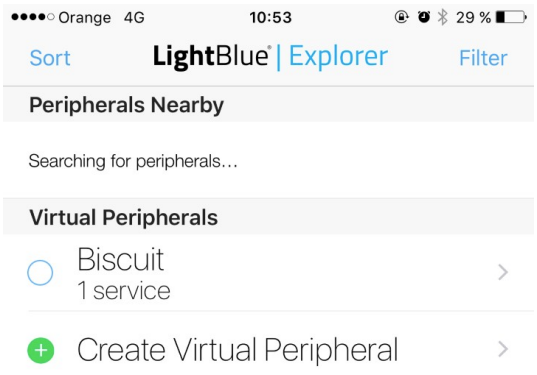


Figura 5.2 - Virtual Peripheral

De esta manera, el dispositivo *Central* ya es capaz de descubrir el dispositivo *Virtual Peripheral*, y con ello conectarse y descubrir los Servicios definidos en éste, mediante llamadas a la función *discoveredServiceCallback()*, mostrada en la Figura 5.3.

```
177 static void discoveredServiceCallback(BLEStatus_t status, uint16_t con_handle,
178             gatt_client_service_t *service) {
179   uint8_t index;
180   if (status == BLE_STATUS_OK) {
181     Serial.println(" ");
182     Serial.print("Service start handle: ");
183     Serial.println(service->start_group_handle, HEX);
184     Serial.print("Service end handle: ");
185     Serial.println(service->end_group_handle, HEX);
186     Serial.print("Service uuid16: ");
187     Serial.println(service->uuid16, HEX);
188     Serial.print("The uuid128 : ");
189     for (index = 0; index < 16; index++) {
190       Serial.print(service->uuid128[index], HEX);
191       Serial.print(" ");
192     }
193     Serial.println(" ");
194     if (0x00 == memcmp(service->uuid128, service1_uuid, 16)) {
195       Serial.println("Target uuid128");
196       device.service.service = *service;
197     }
198   }
199   else if (status == BLE_STATUS_DONE) {
200     Serial.println("Discovered service done");
201     ble.discoverCharacteristics(device.connected_handle, &device.service.service);
202   }
203 }
```

Figura 5.3 - Función discoveredServiceCallback()

En el código de la función *deviceConnectedCallback()* se especifica que el único Servicio *Target* de usuario a descubrir es aquel cuyo UUID se corresponde con el de la variable *service1_uuid[],* como se muestra en la Figura 5.4, definida al siguiente valor:

```
32 static uint8_t service1_uuid[16] = {
33 0x71,0x3d,0x00,0x00,0x50,0x3e,0x4c,0x75,0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
```

Figura 5.4 - UUID de Service1

Por tanto, en el dispositivo *Virtual Peripheral* fue necesario definir un Servicio con el UUID establecido en la variable *service1_uuid[].* Una vez descubiertos todos los Servicios, tanto el Servicio de Usuario como los necesarios para establecer una conexión, relativos al GAP *Service* y el GATT *Service*, el dispositivo *Central* pasa a descubrir las Características asociadas a los Servicios de Usuario, mediante llamadas a la función *discoveredCharsCallback()*, como se muestra en la Figura 5.5.

```
215 static void discoveredCharsCallback(BLEStatus_t status, uint16_t con_handle,
216             gatt_client_characteristic_t *characteristic) {
217   uint8_t index;
218   if (status == BLE_STATUS_OK) {
219     Serial.println(" ");
220     Serial.print("characteristic start handle: ");
221     Serial.println(characteristic->start_handle, HEX);
222     Serial.print("characteristic value handle: ");
223     Serial.println(characteristic->value_handle, HEX);
224     Serial.print("characteristic end_handle: ");
225     Serial.println(characteristic->end_handle, HEX);
226     Serial.print("characteristic properties: ");
227     Serial.println(characteristic->properties, HEX);
228     Serial.print("characteristic uuid16: ");
229     Serial.println(characteristic->uuid16, HEX);
230     Serial.print("characteristic uuid128 : ");
231     for (index = 0; index < 16; index++) {
232       Serial.print(characteristic->uuid128[index], HEX);
233       Serial.print(" ");
234     }
235     Serial.println(" ");
236     if (chars_index < 2) {
237       device.service.chars[chars_index].chars= *characteristic;
238       chars_index++;
239     }
240   }
241   else if (status == BLE_STATUS_DONE) {
242     Serial.println("Discovered characteristic done");
243     chars_index = 0;
244     ble.discoverCharacteristicDescriptors(device.connected_handle,
245         &device.service.chars[chars_index].chars);
246   }
247 }
```

Figura 5.5 - Función discoveredCharsCallback()

En el código de la función *discoveredCharsCallback()* se especifica que, del Servicio de Usuario que se descubre en el proceso de *Scanning*, se descubrirán un máximo de dos Características, y como éstas no se especifican, para definir las Características asociadas al Servicio definido en el dispositivo *Virtual Peripheral* de la *app Lightblue* se asignan los siguientes UUIDs y propiedades, quedando como se muestra en la Figura 5.6.

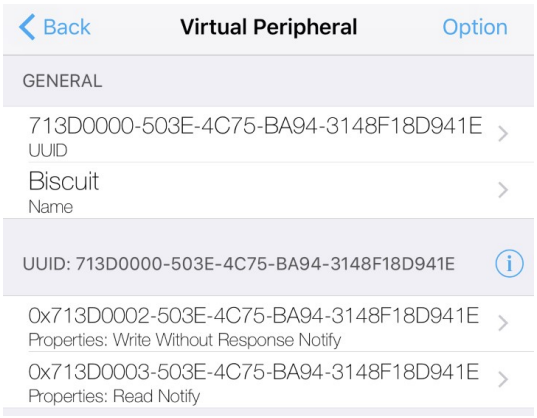


Figura 5.6 - Características del Virtual Peripheral

En este punto, tras cargar el *firmware* correspondiente a la aplicación *simpleblecentral_duo.ino* en el dispositivo *RedBear Duo* desde el entorno *Particle IDE*, y activando el *Virtual Peripheral* en la *app LightBlue*, fue posible completar con éxito los procesos de Conexión/Descubrimiento de Servicios/Características/Descriptores, obteniéndose, los siguientes mensajes a través del terminal serie conectado al dispositivo *Central* implementado en el dispositivo *RedBear Duo*.

En primer lugar, comienza el proceso de *Scanning*, y cuando se recibe el *Complete Local Name* del dispositivo *Peripheral*, el dispositivo *Central* verifica que coincide con el especificado en el código y se lleva a cabo la conexión, tal y como se puede observar en la Figura 5.7.

```
BLE central demo!
Start scanning
reportCallback:
The advEventType: 0
The peerAddrType: 1
The peerAddr: 44 A2 90 44 6F 43
The rssi: -28
The ADV data: 2 1 1A 11 7 1E 94 8D F1 48 31 94 BA 75 4C 3E 50 0 0 3D 71 8
9 42 69 73 63 75 69 74

  The length of Complete Local Name : 7
  The Complete Local Name is        : Biscuit

Device connected!
```

Figura 5.7 - Proceso de Scanning y conexión por el dispositivo RedBear Duo

Una vez conectados los dispositivos *Central* y *Peripheral*, comienza el descubrimiento de Servicios. En este caso se descubren seis Servicios, el primero de ellos con *Service*

UUID de valor 0x1800, se corresponde con el Servicio BLE_UUID_GAP, el segundo, con *Service* UUID de valor 0x1801, se corresponde con el Servicio BLE_UUID_GATT, el tercero, con *Service Start Handle* de valor 1C y *Service* UUID de valor 0x180F, se corresponde con BLE_UUID_BATTERY_SERVICE, el cuarto con *Service* UUID de valor 0x1805, se corresponde con el Servicio BLE_UUID_CURRENT_TIME_SERVICE, el quinto con *Service* UUID de valor 0x180A, se corresponde con BLE_UUID_DEVICE_INFORMATION_SERVICE, y, por último, se descubre el *Target uuid128* que se corresponde con el SERVICE1_UUID que es el Servicio de Usuario, como se muestra en la Figura 5.8.

```
Service start handle: 1
Service end handle: 5
Service uuid16: 1800
The uuid128 : 0 0 18 0 0 0 10 0 80 0 0 80 5F 9B 34 FB

Service start handle: 6
Service end handle: 9
Service uuid16: 1801
The uuid128 : 0 0 18 1 0 0 10 0 80 0 0 80 5F 9B 34 FB

Service start handle: A
Service end handle: E
Service uuid16: 0
The uuid128 : D0 61 1E 78 BB B4 45 91 A5 F8 48 79 10 AE 43 66

Service start handle: F
Service end handle: 13
Service uuid16: 0
The uuid128 : 9F A4 80 E0 49 67 45 42 93 90 D3 43 DC 5D 4 AE

Service start handle: 1C
Service end handle: 1F
Service uuid16: 180F
The uuid128 : 0 0 18 F 0 0 10 0 80 0 0 80 5F 9B 34 FB

Service start handle: 20
Service end handle: 25
Service uuid16: 1805
The uuid128 : 0 0 18 5 0 0 10 0 80 0 0 80 5F 9B 34 FB

Service start handle: 26
Service end handle: 2A
Service uuid16: 180A
The uuid128 : 0 0 18 A 0 0 10 0 80 0 0 80 5F 9B 34 FB


Service start handle: 2B
Service end handle: 34
Service uuid16: 0
The uuid128 : 79 5 F4 31 B5 CE 4E 99 A4 F 4B 1E 12 2D 0 D0

Service start handle: 35
Service end handle: 40
Service uuid16: 0
The uuid128 : 89 D3 50 2B F 36 43 3A 8E F4 C5 2 AD 55 F8 DC
```

```
Service start handle: 41
Service end handle: 48
Service uuid16: 0
The uuid128 : 71 3D 0 0 50 3E 4C 75 BA 94 31 48 F1 8D 94 1E
Target uuid128
Discovered service done
```

Figura 5.8 - Servicios descubiertos por el dispositivo RedBear Duo

Una vez descubiertos todos los Servicios se descubren dos Características, la primera de ellas se corresponde con la Característica 1 y la segunda con la Característica 2 definidas en el dispositivo *Virtual Peripheral* con sus UUIDs correspondientes, mostradas en la Figura 5.9.

```
characteristic start handle: 42
characteristic value handle: 43
characteristic end_handle: 45
characteristic properties: 94
characteristic uuid16: 0
characteristic uuid128 : 71 3D 0 2 50 3E 4C 75 BA 94 31 48 F1 8D 94 1E

characteristic start handle: 46
characteristic value handle: 47
characteristic end_handle: 48
characteristic properties: 12
characteristic uuid16: 0
characteristic uuid128 : 71 3D 0 3 50 3E 4C 75 BA 94 31 48 F1 8D 94 1E
Discovered characteristic done
```

Figura 5.9 - Características descubiertas por el dispositivo RedBear Duo

Por último, se descubren los Descriptores, como se muestra en la Figura 5.10.

```
descriptor handle: 44
descriptor uuid16: 2900
descriptor uuid128 : 0 0 29 0 0 0 10 0 80 0 0 80 5F 9B 34 FB

descriptor handle: 45
descriptor uuid16: 2902
descriptor uuid128 : 0 0 29 2 0 0 10 0 80 0 0 80 5F 9B 34 FB
Discovered descriptor done

descriptor handle: 48
descriptor uuid16: 2902
descriptor uuid128 : 0 0 29 2 0 0 10 0 80 0 0 80 5F 9B 34 FB
Discovered descriptor done
```

Figura 5.10 - Descriptores descubiertos por el dispositivo RedBear Duo

A continuación, en el dispositivo *Virtual Peripheral* de la *app LightBlue* se procede a modificar el valor asociado a la Característica 2, definida con las propiedades READ, NOTIFY, a 0x0C.

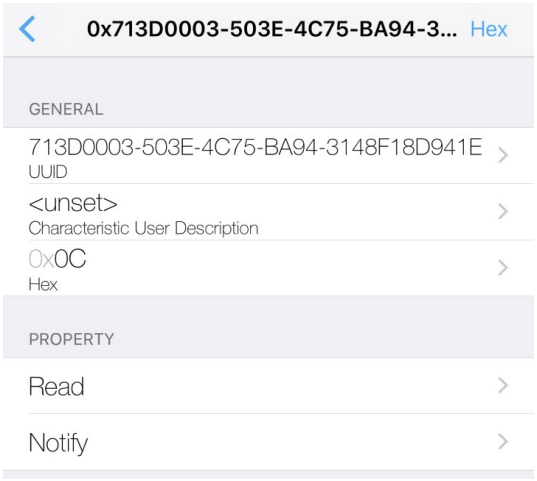


Figura 5.11 - Modificación valor característica 2

Esto llevó en el dispositivo *Central* a llamar a la función *gattReadCallback(),* representada en la Figura 5.12, que mostraba a través del terminal serie el nuevo valor establecido para esta Característica.

```
302 void gattReadCallback(BLEStatus_t status, uint16_t con_handle, uint16_t value_handle,
303 uint8_t *value, uint16_t length) {
304   uint8_t index;
305   if (status == BLE_STATUS_OK) {
306     Serial.println(" ");
307     Serial.println("Read characteristic ok");
308     Serial.print("conn handle: ");
309     Serial.println(con_handle, HEX);
310     Serial.print("value handle: ");
311     Serial.println(value_handle, HEX);
312
313     Serial.print("The value : ");
314     for (index = 0; index < length; index++) {
315       Serial.print(value[index], HEX);
316       Serial.print(" ");
317     }
318     Serial.println(" ");
319   }
320   else if (status == BLE_STATUS_DONE) {
321     uint8_t data[]= {0x01,0x02,0x03,0x04,0x05,1,2,3,4,5};
322     ble.writeValue(device.connected_handle, device.service.chars[0].chars.value_handle,
323     sizeof(data), data);
324   }
325 }
```

Figura 5.12 - Función gattReadCallback()

En este caso, se comprueba cómo la Característica 2, con *Value Handle de valor 47,* contiene el valor 0xC especificado en el dispositivo *Virtual Peripheral*, como se muestra en la Figura 5.13.

```
Read characteristic ok
conn handle: 40
value handle: 47
The value : C
Disconnected handle:40
Restart scanning.
```

Figura 5.13 - Característica 2

5.2 Dispositivo de desarrollo *hardware Bluz DK*

En este apartado se incluyen las especificaciones técnicas del dispositivo *Bluz DK*, que será empleado posteriormente como dispositivo *Peripheral* en la conexión con el dispositivo *Central* implementado en el dispositivo *RedBear Duo* **[7]**. El kit de desarrollo hardware de IoT *Bluz DK* de la empresa *Bluz* **[19]**, se ha escogido para el desarrollo de este TFM debido a su bajo coste, su programabilidad, su utilización de código abierto y por integrar un módulo BLE que lo dota de la conectividad necesaria para llevar a cabo el objetivo propuesto en este TFM. Además, este dispositivo, al igual que el dispositivo *RedBear Duo*, es soportado por la plataforma *Particle IDE* para dispositivos IoT.

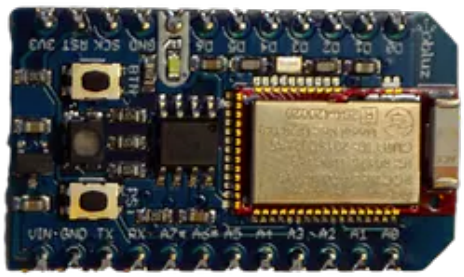


Figura 5.14 - Dispositivo Bluz DK

El dispositivo *Bluz DK*, mostrado en la Figura 5.14, combina un potente microcontrolador *ARM Cortex-M0* con el módulo *nrf51822* que ofrece conectividad BLE. Entre otras características, el dispositivo *Bluz DK* dispone de 256KB de memoria *Flash* y 32KB de RAM, además de varios LEDs (*Light-Emitting Diode*) integrados, 18 entradas/salidas de propósito general (GPIO), periféricos avanzados y un sistema operativo en tiempo real (*FreeRTOS*), como se muestra en la Figura 5.15. Además, este

dispositivo cuenta con una gran cantidad de interfaces analógicas, digitales y de comunicación, como SPI, UART o I2C, entre otras.

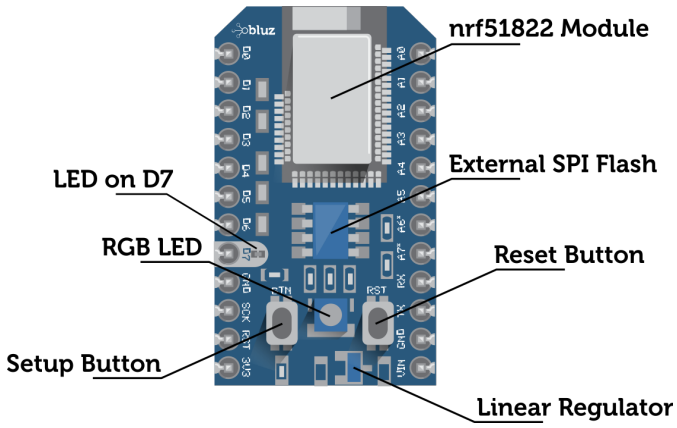


Figura 5.15 - Diagrama de bloques del dispositivo Bluz DK

La plataforma cuenta con dos botones (*Setup* y *Reset*) que permiten configurar y reiniciar el dispositivo. Entre ambos botones se encuentra un LED de tipo RGB (Red, Green, Blue) que proporciona información acerca del estado del dispositivo. Además, en la Figura 5.16 se muestra la configuración de pines del dispositivo *Bluz D*K, que serán explicados a continuación.

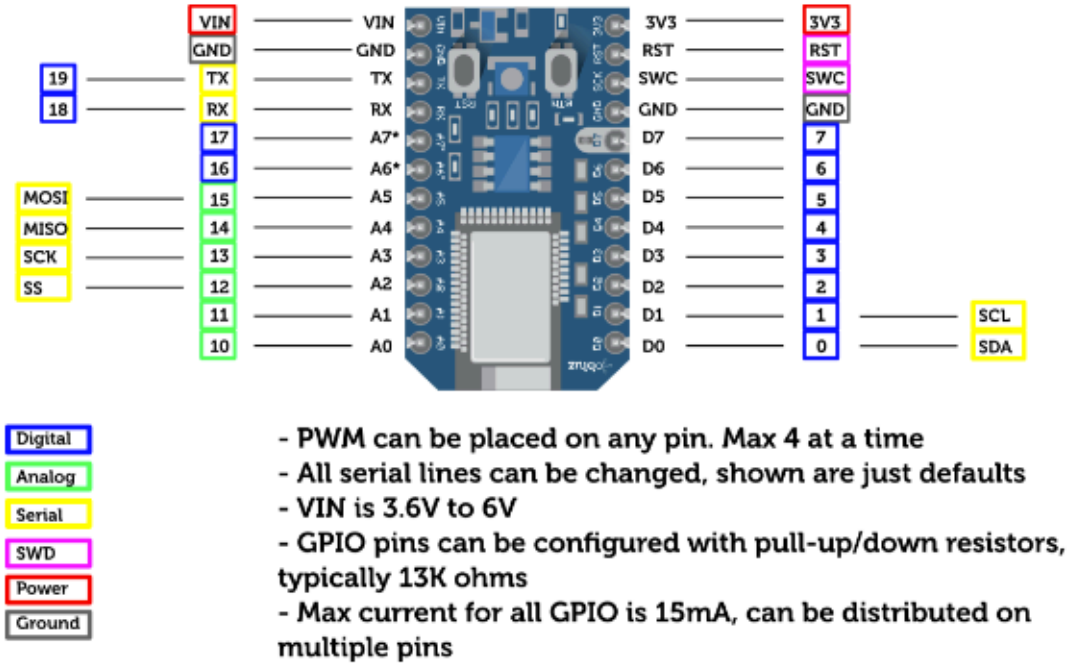


Figura 5.16 - Pinout del dispositivo Bluz DK

Para la alimentación del dispositivo se deben emplear los pines de alimentación (*VIN* y *GND*), y es necesario un suministro de 3.3 Voltios al pin *VIN*, ya que toda la lógica del

dispositivo funciona con 3.3V. El pin *RST* se puede utilizar, al igual que el botón *Reset*, para reiniciar el sistema.

Los pines *D0* a *D7* son pines de propósito general que pueden actuar como entradas o salidas digitales. Tal y como se aprecia en la Figura 5.16, existe también un LED azul situado junto al pin *D7,* que se encuentra conectado directamente a este pin.

Los pines *A0* a *A5* constituyen entradas analógicas que trabajan con tensiones de entre 0 y 3.3V. Los pines analógicos también se pueden utilizar como entradas o salidas digitales, como los pines *D0* a *D7* y, al igual que los pines digitales, algunos pines analógicos también se pueden utilizar como salidas analógicas PWM.

Los pines *TX* y *RX* (Transmisión y Recepción, respectivamente) se utilizan para la comunicación serie. Por último, situados por debajo de estos pines se encuentran un segundo pin *GND* y el pin *VIN*.

5.3 *Bluz DK* como dispositivo *Peripheral*

A continuación, se presentan los resultados obtenidos a partir del análisis inicial realizado sobre el dispositivo *Bluz DK* con el objetivo de validar una posible conexión BLE, actuando éste como dispositivo *Peripheral*, con un dispositivo *RedBear Duo* que actuará como dispositivo *Central* para la transferencia de datos.

En este sentido, los primeros pasos para analizar el funcionamiento del dispositivo *Bluz DK* estaban encaminados a intentar conectar un dispositivo *Bluz DK* actuando como dispositivo *Peripheral*, a una *app* que actuara como dispositivo *Central*. Para ello, se intentó utilizar en primer lugar la *app Lightblue* para iOS, ya utilizada para analizar el funcionamiento del dispositivo *RedBear Duo*. Sin embargo, utilizando esta *app* fue imposible completar con éxito una conexión con el dispositivo *Bluz DK,* debido a un problema que presentaba inicialmente el *firmware* del dispositivo *Bluz* DK y que se explicará posteriormente en este TFM, por lo que se decidió usar como alternativa en este caso la *app BLEscanner* para *Android* **[22]**, a pesar de que se comprobó que también podía usarse la *app nrfConnect* para iOS.

En este punto hay que tener en cuenta que para lograr la conexión con el entorno *Particle IDE* y poder programar el dispositivo *Bluz DK*, la variable SYSTEM_MODE, que por defecto se encuentra en modo MANUAL, debe encontrarse en modo AUTOMATIC, lo cual se implementa en el *firmware* del dispositivo *Bluz DK* haciendo que, cuando el pin *D6* se encuentre a nivel alto, el valor de la variable *SYSTEM_MODE* sea AUTOMATIC, para lo cual será necesario conectar externamente el pin *D6* al pin *3V3* del dispositivo *Bluz DK*. Una vez programado, se debe deshacer dicha conexión y realizar un *reset* del dispositivo *Bluz DK* para asegurar la correcta ejecución del *firmware* programado.

Así, con el fin de analizar el funcionamiento del dispositivo *Bluz DK* como dispositivo *Peripheral,* se desarrolló el fichero *localcomm_bluzDKv01.ino,* mostrado en la Figura 5.17. Con este *firmware* se pretendía, en primer lugar, determinar el procedimiento y extraer la información contenida en los paquetes de *Advertising* enviados por el dispositivo *Bluz DK* con el fin de poder establecer una conexión, y con ello, descubrir los diferentes Servicios y Características definidas en él para enviar/recibir datos.

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

bool sendData = false;

void dataCallbackHandler(uint8_t *data, uint16_t length) {
    sendData = true;
    digitalWrite(D7, HIGH);
}

void setup() {
    pinMode(D7, OUTPUT);
    digitalWrite(D7, HIGH);
    delay(5000);
    digitalWrite(D7, LOW);

    BLE.registerDataCallback(dataCallbackHandler);

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

void loop() {
    System.sleep(SLEEP_MODE_CPU);
    if (sendData)
    {
        uint8_t rsp[2] = {'H', 'i'};
        BLE.sendData(rsp, 2);
        sendData = false;
    }
}
```

Figura 5.17 - Fichero localcomm_bluzDKv01.ino

Así, en el código *localcomm_bluzDK_v01.ino* desarrollado se usan dos funciones del *firmware* del dispositivo *Bluz DK*, que son *BLE.registerDataCallback()* en la línea 17, que registra en este caso la función *dataCallbackHandler(),* que será llamada cuando se envíen datos al dispositivo *Bluz DK* a través de una comunicación local con otro dispositivo, y la función *BLE.SendData()* en la línea 30, que usará el Servicio de Usuario definido en el dispositivo *Bluz DK* para enviar datos directamente al dispositivo *Central*, que inicialmente será la *app BLEScanner*. Es importante indicar que, según está definido en el Servicio de Usuario del dispositivo *Bluz DK*, se añadirá una cabecera de 1 byte con el valor 0x04 a los datos enviados desde el dispositivo *Bluz DK* mediante la función *BLE.SendData()*, así como un paquete final de cola incluyendo un *payload* de 2 bytes con el valor 0x0304*.* Del mismo modo, también es importante indicar que los datos recibidos por la función *dataCallbackHandler(),* que será llamada cuando el dispositivo *Bluz DK* reciba datos mediante una comunicación local, no incluirá la cabecera de 1 byte con el valor 0x04, que será eliminada, así como la cola de 2 bytes con el valor 0x0304.

Así, a partir de este *firmware*, se espera que cuando se envían datos correctamente desde la *app BLEScanner*, estos sean recibidos satisfactoriamente por el dispositivo *Bluz DK*, y se encienda el LED *D7*, permaneciendo apagado en caso contrario. Por otro lado, con el fin de validar que la programación del dispositivo *Bluz DK* se ha realizado correctamente, o bien que se ha iniciado la ejecución del *firmware*, en el código se especifica que en el procedimiento de *setup()* se encienda el LED *D7* durante un periodo de 5 segundos. Por otro lado, además de encenderse el LED *D7*, cuando se reciban datos correctamente en el dispositivo *Bluz DK*, éste enviará hacia el dispositivo *Central* un paquete de respuesta que contiene el dato 0x4869, correspondiente a los caracteres ASCII ‘H’ e ‘i’.

Por otro lado, en cuanto al proceso de envío/recepción de datos en una comunicación BLE local con el dispositivo *Bluz DK,* se especifica el formato de datos del Servicio de Usuario definido en el dispositivo *Bluz DK*, indicándose que debe incluirse con los datos una cabecera de 1 byte de valor 0x04, así como una cola de 2 bytes con el valor

0x0304, siendo este el formato seguido por el dispositivo *Bluz DK* para enviar datos hacia el dispositivo *Central*.

Con todo lo anterior, ya se está en disposición de validar el proceso de envío/recepción de datos entre la *app BLEScanner* actuando como dispositivo *Central,* y un dispositivo *Bluz DK* actuando como dispositivo *Peripheral*.

Así, tras programar el dispositivo *Bluz DK*, y encontrarse éste en modo *Advertising* (LED de estadp parpadeando en verde), se inicia la *app BLEScanner*, detectándose el dispositivo *Bluz DK* al realizar el proceso de *Scanning*, y pudiendo conectarse a él, obteniéndose la información relativa al *GAP Service* (UUID 0x1800) definido en el dispositivo *Bluz DK*, en el que se incluyen las Características indicadas en la Figura 5.18.

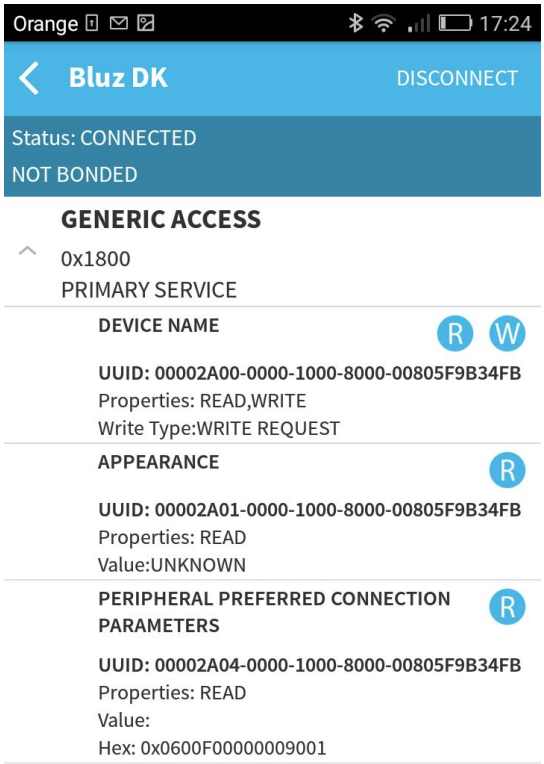


Figura 5.18 - Información relativa al GAP Service

Por otro lado, además del GAP *Service*, se puede obtener la información relativa al Servicio de Usuario definido en el dispositivo *Bluz DK (Custom Service)*, y de las dos Características que incluye, junto con sus propiedades, como se muestra en la Figura 5.19.

- *Custom Service UUID* (BLUZ_UUID): 871E0223 – 38FF – 77B1 – ED41 – 9FB3AA142DB2
- *Custom Characteristic1* (BLUZ_CHAR_TX_UUID): 871E0224 – 38FF – 77B1 – ED41 – 9FB3AA142DB2, READ, WRITE, WRITE_WITHOUT_RESPONSE
- *Custom Characteristic2* (BLUZ_CHAR_RX_UUID): 871E0225 – 38FF – 77B1 – ED41 – 9FB3AA142DB2, READ, NOTIFY

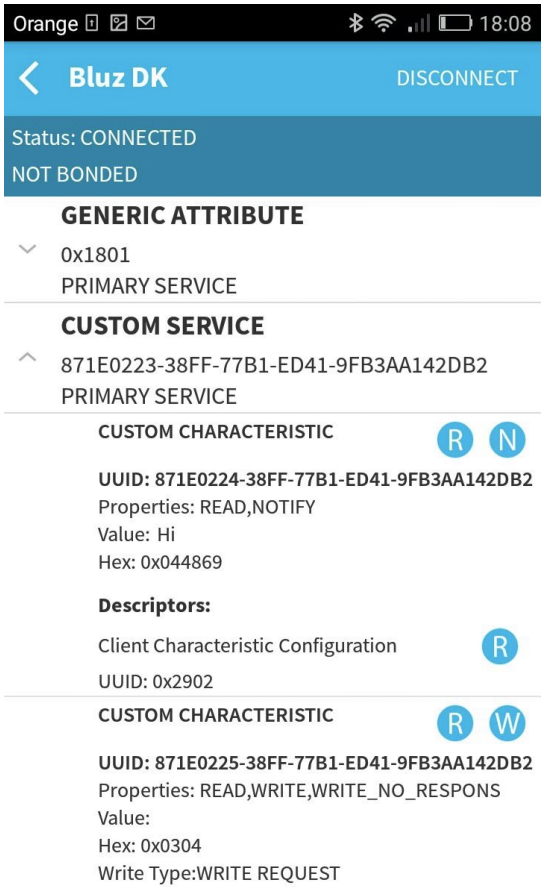


Figura 5.19 - Custom Service y características del dispositivo Bluz DK

A continuación, en la app *BLEScanner* se hizo uso de la Característica *CustomCharacteristic1* para enviar al dispositivo *Bluz DK* (W) un primer paquete de datos de valor 0x0403, correspondiente a la cabecera 0x04 y al dato 0x03, y posteriormente un segundo paquete de datos de valor 0x0304, correspondiente a la cola de 2 bytes, que indica el final de los datos enviados. Como resultado se obtuvo que el LED *D7* se encendió, como se muestra en la Figura 5.20, indicando la correcta recepción de los datos en el dispositivo *Bluz DK*, además de recibirse en la *app BLEScanner* el paquete de respuesta, enviado por el dispositivo *Bluz DK* con los datos

0x044864, correspondientes a los caracteres ASCII “Hi”, precedido de la cabecera de 1 byte de valor 0x04, al seleccionar el botón (R) en la Característica *CustomCharacteristic2*, mostrada en la figura 5.19.

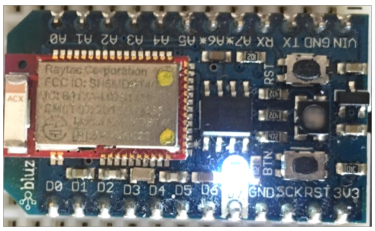


Figura 5.20 - LED D7 del dispositivo Bluz DK

5.4 Conexión mediante BLE de los dispositivos *RedBear Duo* y *Bluz DK*

A partir de los análisis realizados hasta ahora sobre el dispositivo *RedBear Duo* actuando como dispositivo *Central*, y el dispositivo *Bluz DK* actuando como dispositivo *Peripheral*, el siguiente paso planteado fue la integración de ambos dispositivos en una conexión BLE. Por ello, se desarrollaron los códigos de los ficheros *simpleblecentral-duo-bluzdk-v03.ino* y *localcomm-bluzdk-v03.ino*, correspondientes al *firmware* de los dispositivos *RedBear Duo* y *Bluz DK*, respectivamente.

En el caso del fichero *localcomm_bluzdk_v03.ino,* mostrado en la Figura 5.21, correspondiente al dispositivo *Bluz DK* actuando como dispositivo *Peripheral*, se pretende controlar el apagado/encendido del LED *D7* en función de que el valor de los datos recibidos desde el dispositivo *Central* sean 0x6E/0x6F (líneas 38 y 41), respectivamente, además de incluir en la respuesta enviada por el dispositivo *Bluz DK* todos los datos almacenados en el *buffer sav_data[],* en función de su longitud *sav_length* (líneas 5 y 6), además de los valores hexadecimales correspondientes a los caracteres ASCII “Hi”, de valor, 0x4869.

```
1   #include "application.h"
2   SYSTEM_MODE(MANUAL);
3
4   bool sendData = false;
5   uint16_t sav_length;
6   uint8_t sav_data[20];
7   uint8_t rsp_data[20];
8
9   void dataCallbackHandler(uint8_t *data, uint16_t length) {
10      sav_length = length;
11      for (int i=0; i<length; i++) sav_data[i] = *data++;
12      sendData = true;
13  }
```

```
15  void setup() {
16      pinMode(D7, OUTPUT);
17      digitalWrite(D7, HIGH);
18      delay(5000);
19      digitalWrite(D7, LOW);
20
21      BLE.registerDataCallback(dataCallbackHandler);
22
23      pinMode(D6, INPUT_PULLDOWN);
24      if (digitalRead(D6) == HIGH) {
25          SYSTEM_MODE(AUTOMATIC);
26      }
27  }
28
29  void loop() {
30      System.sleep(SLEEP_MODE_CPU);
31      if (sendData) {
32          rsp_data[0] = 'H';
33          rsp_data[1] = 'i';
34          for (int i=0; i<sav_length; i++) rsp_data[i+2] = sav_data[i];
35
36          BLE.sendData(rsp_data, (sav_length+2));
37          switch (sav_data[0]) {
38              case 0x6e:
39                  digitalWrite(D7, HIGH);
40                   break;
41              case 0x6f:
42                  digitalWrite(D7, LOW);
43                  break;
44          }
45          sendData = false;
46      }
47  }
```

Figura 5.21 - localcomm_bluzdk_v03.ino

Por otra parte, en el caso del código correspondiente al dispositivo *RedBear Duo* actuando como dispositivo *Central,* fue necesario realizar una serie de modificaciones que dieron lugar a la creación del fichero *simpleblecentral-duo-bluzdk-v03.ino* en el entorno *Particle IDE*. El objetivo final de este código era reproducir, desde el dispositivo *RedBear Duo,* la interacción lograda con el dispositivo *Bluz DK* a partir de la *app BLEScanner*, detallada anteriormente en este TFM, de manera que al enviar un primer paquete de valor 0x046E seguido de un segundo paquete de valor 0x0304, desde el dispositivo *RedBear Duo* hacia el dispositivo *Bluz DK*, una vez establecida correctamente la conexión entre ambos se encenderá el LED *D7* del dispositivo *Bluz DK*, como se especifica en el código *localcomm-bluzdk-v03.ino,* recibiéndose como respuesta en el dispositivo *RedBear Duo*, ante la realización de la lectura de la Característica correspondiente, un paquete que además del byte de cabecera de valor 0x04, incluyera el código hexadecimal correspondiente a los caracteres ASCII “Hi”, de valor 0x4869, y el byte de datos recibidos por el dispositivo *Bluz DK*, correspondiente en este caso al valor 0x6E, y el paquete de cola con el valor 0x0304 en hexadecimal. Del mismo modo, al enviar desde el dispositivo *RedBear Duo* hacia el dispositivo *Bluz*

DK un primer paquete de valor 0x046F, seguido de un segundo paquete de valor 0x0304, se apagará el LED *D7* del dispositivo *Bluz DK*, recibiéndose como respuesta en el dispositivo *RedBear Duo* un paquete que, además del byte de cabecera 0x04, incluirá el código hexadecimal correspondiente a los caracteres ASCII "Hi", de valor 0x4869, y el byte de datos recibidos por el dispositivo *Bluz DK*, correspondiente en este caso al valor 0x06, y el paquete de cola con el valor 0x0304.

A continuación, se muestran las modificaciones que dieron lugar al código correspondiente al *firmware* del dispositivo *RedBear Duo* actuando como dispositivo Central (*simpleblecentralduo-bluzdk-v04.ino*), con los que se obtuvo el funcionamiento esperado, así como la información obtenida a través de la conexión serie del dispositivo *RedBear Duo*, en un terminal, y que refleja todos los procesos involucrados en la conexión y posterior transferencia de datos entre ambos dispositivos.

En primer lugar se modificó la variable *service1_uuid[]* con el UUID del dispositivo *Bluz DK,* como se puede observar en la Figura 5.22..

```
31  static uint8_t service1_uuid[16] = {
32  0x87,0x1e,0x02,0x23,0x38,0xff,0x77,0xb1,0xed,0x41,0x9f,0xb3,0xaa,0x14,0x2d,0xb2 };
```

Figura 5.22 - UUID del dispositivo Bluz DK

Además, se obtiene el valor del parámetro *Complete Local Name* y se comprueba que se corresponde con el que define el dispositivo *Bluz DK,* como se muestra en la Figura 5.23.

```
92      if (0x00 == memcmp(adv_name, "Bluz DK", min(7, len))) {
93        Serial.println("_____ Found BluzDK");
94        ble.stopScanning();
95        device.addr_type = report->peerAddrType;
96        memcpy(device.addr, report->peerAddr, 6);
97
98        ble.connect(report->peerAddr, {BD_ADDR_TYPE_LE_RANDOM});
99      }
```

Figura 5.23 - Complete Local Name

Por último, también se incluyó la función *characteristic1_bluzdk_write(),* que realiza el envío y recepción de los paquetes transferidos entre los dispositivos *RedBear Duo* y *Bluz DK*, mostrada en el código de la Figura 5.24.

```
348 static void characteristic1_bluzdk_write(btstack_timer_source_t *ts) {
349   Serial.println(" ");
350   uint8_t data_bluzdk_ON_1[]= {0x04,0x6e};
351   uint8_t data_bluzdk_ON_2[]= {0x03,0x04};
352   uint8_t data_bluzdk_OFF_1[]= {0x04,0x6f};
353   uint8_t data_bluzdk_OFF_2[]= {0x03,0x04};
354   uint8_t index;
355
356   if (bluzdk_write_index == 0) {
357     Serial.println("");
358     Serial.println("> Characteristic1_bluzdk_write: ");
359     Serial.println("   - data_bluzdk_ON_1");
360     Serial.print("   - Connection handle: ");
361     Serial.println(device.connected_handle);
362     Serial.print("   - Characteristic value attribute handle: ");
363     Serial.println(device.service.chars[1].chars.value_handle);
364     Serial.print("   - Characteristic value : ");
365     for (index = 0; index < sizeof(data_bluzdk_ON_1); index++) {
366       Serial.print(data_bluzdk_ON_1[index], HEX);
367       Serial.print(" ");
368       }
369     ble.writeValue(device.connected_handle, device.service.chars[1].chars.value_handle,
370     sizeof(data_bluzdk_ON_1), data_bluzdk_ON_1);
371     bluzdk_write_index++;
372
373     ble.setTimer(ts, 10000);
374     ble.addTimer(ts);
375   }
376   else if (bluzdk_write_index == 1) {
377     Serial.println("");
378     Serial.println("> Characteristic1_bluzdk_write: ");
379     Serial.println("   - data_bluzdk_ON_2");
380     Serial.print("   - Connection handle: ");
381     Serial.println(device.connected_handle);
382     Serial.print("   - Characteristic value attribute handle: ");
383     Serial.println(device.service.chars[1].chars.value_handle);
384     Serial.print("   - Characteristic value : ");
385     for (index = 0; index < sizeof(data_bluzdk_ON_2); index++) {
386       Serial.print(data_bluzdk_ON_2[index], HEX);
387       Serial.print(" ");
388       }
389     ble.writeValue(device.connected_handle, device.service.chars[1].chars.value_handle,
390     sizeof(data_bluzdk_ON_2), data_bluzdk_ON_2);
391     bluzdk_write_index++;
392
393     ble.setTimer(ts, 10000);
394     ble.addTimer(ts);
395   }
396   else if (bluzdk_write_index == 2) {
397     Serial.println("");
398     Serial.println("> Characteristic0_bluzdk_read: ");
399     Serial.print("   - Connection handle: ");
400     Serial.println(device.connected_handle);
401     Serial.print("   - Characteristic value attribute handle: ");
402     Serial.print(device.service.chars[0].chars.value_handle);
403     ble.readValue(device.connected_handle,&device.service.chars[0].chars);
404     bluzdk_write_index++;
405
406     ble.setTimer(ts, 10000);
407     ble.addTimer(ts);
408   }
409   else if (bluzdk_write_index == 3) {
410     Serial.println("");
411     Serial.println("> Characteristic1_bluzdk_write: ");
412     Serial.println("   - data_bluzdk_OFF_1");
413     Serial.print("   - Connection handle: ");
414     Serial.println(device.connected_handle);
415     Serial.print("   - Characteristic value attribute handle: ");
416     Serial.println(device.service.chars[1].chars.value_handle);
417     Serial.print("   - Characteristic value : ");
```

```
418     for (index = 0; index < sizeof(data_bluzdk_OFF_1); index++) {
419       Serial.print(data_bluzdk_OFF_1[index], HEX);
420       Serial.print(" ");
421       }
422     ble.writeValue(device.connected_handle, device.service.chars[1].chars.value_handle,
423     sizeof(data_bluzdk_OFF_1), data_bluzdk_OFF_1);
424     bluzdk_write_index++;
425
426     ble.setTimer(ts, 10000);
427     ble.addTimer(ts);
428   }
429   else if (bluzdk_write_index == 4) {
430     Serial.println("");
431     Serial.println("> Characteristic1_bluzdk_write: ");
432     Serial.println("    - data_bluzdk_OFF_2");
433     Serial.print("    - Connection handle: ");
434     Serial.println(device.connected_handle);
435     Serial.print("    - Characteristic value attribute handle: ");
436     Serial.println(device.service.chars[1].chars.value_handle);
437     Serial.print("    - Characteristic value : ");
438     for (index = 0; index < sizeof(data_bluzdk_OFF_2); index++) {
439       Serial.print(data_bluzdk_OFF_2[index], HEX);
440       Serial.print(" ");
441       }
442     ble.writeValue(device.connected_handle, device.service.chars[1].chars.value_handle,
443     sizeof(data_bluzdk_OFF_2), data_bluzdk_OFF_2);
444     bluzdk_write_index++;
445
446     ble.setTimer(ts, 10000);
447     ble.addTimer(ts);
448   }
449   else if (bluzdk_write_index == 5) {
450     Serial.println("");
451     Serial.println("> Characteristic0_bluzdk_read: ");
452     Serial.print("    - Connection handle: ");
453     Serial.println(device.connected_handle);
454     Serial.print("    - Characteristic value attribute handle: ");
455     Serial.print(device.service.chars[0].chars.value_handle);
456     ble.readValue(device.connected_handle,&device.service.chars[0].chars);
457     bluzdk_write_index++;
458
459     ble.setTimer(ts, 10000);
460     ble.addTimer(ts);
461   }
462 }
463
464
```

Figura 5.24 - Función characteristic1_bluzdk_write()

En la función *setup()*, mostrada en la Figura 5.25, se muestra todo el proceso de conexión, descubrimiento de Servicios, Características, etc., así como la llamada a la función *characteristic1_bluzdk_write()*, como se muestra en el código de la Figura 5.25.

```
465 void setup() {
466   Serial.begin(115200);
467   delay(5000);
468
469   Serial.println(" ");
470   Serial.println("_____ BLE Central <-> BluzDK _____");
471
472   ble.init();
473
474   ble.onConnectedCallback(deviceConnectedCallback);
475   ble.onDisconnectedCallback(deviceDisconnectedCallback);
476   ble.onScanReportCallback(reportCallback);
477   ble.onServiceDiscoveredCallback(discoveredServiceCallback);
478   ble.onCharacteristicDiscoveredCallback(discoveredCharsCallback);
479   ble.onDescriptorDiscoveredCallback(discoveredCharsDescriptorsCallback);
480   ble.onGattCharacteristicReadCallback(gattReadCallback);
```

```
481     ble.onGattCharacteristicWrittenCallback(gattWrittenCallback);
482     ble.onGattDescriptorReadCallback(gattReadDescriptorCallback);
483     ble.onGattWriteClientCharacteristicConfigCallback(gattWriteCCCDCallback);
484     ble.onGattNotifyUpdateCallback(gattNotifyUpdateCallback);
485
486     ble.setScanParams(BLE_SCAN_TYPE, BLE_SCAN_INTERVAL, BLE_SCAN_WINDOW);
487
488     ble.startScanning();
489     Serial.println("_____ BLE Central start scanning");
490
491     characteristic1_bluzdk.process = &characteristic1_bluzdk_write;
492     ble.setTimer(&characteristic1_bluzdk, 10000);
493     ble.addTimer(&characteristic1_bluzdk);
494 }
```

Figura 5.25 - Función setup()

Una vez volcados los ficheros analizados con anterioridad en los dispositivos *RedBear Duo* y *Bluz DK*, se obtiene por el terminal los mensajes de la conexión que se muestran a continuación. En primer lugar, como se muestra en la Figura 5.26, el dispositivo *Central* comienza el proceso de *Scanning* y encuentra un dispositivo cuyo AD_TYPE = 0x09 (BLE_GAP_ADTYPE_COMPLETE_DEVICE_NAME) coincide con el parámetro *Complete Local Name* del dispositivo *Peripheral Bluz DK,* iniciándose la conexión. A continuación, se descubren los Servicios *GAP* y *GATT*, así como el *User Defined Service*.

```
_____ BLE Central <-> BluzDK _____
_____ BLE Central start scanning

* BLE scan callback:
   - Advertising event type: 0
   - Peer device address type: 1
   - Peer device address: E7 A8 6D 31 7D A
   - RSSI: -16
   - Advertising/Scan response data packet: 8 9 42 6C 75 7A 20 44 4B 3 19 0 0 2 1 6
      - AVD/SR data decoding -> ad_type: 9, length: 8

   - Complete Local Name: Bluz DK
_____ Found BluzDK

_____ Device connected
        - Device connected handle: 64

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

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

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

* Discover all services completed
```

Figura 5.26 - Proceso de Scanning y descubrimiento de Servicios

Posteriormente, como se muestra en la Figura 5.27, se descubren las dos Características del dispositivo *Bluz DK*, con las siguientes propiedades:

- *Custom Characteristic0* (BLUZ_CHAR_RX_UUID): 871E0225 – 38FF – 77B1 – ED41 – 9FB3AA142DB2, READ, NOTIFY
- *Custom Characteristic1* (BLUZ_CHAR_TX_UUID): 871E0224 – 38FF – 77B1 – ED41 – 9FB3AA142DB2, READ, WRITE, WRITE_WITHOUT_RESPONSE

```
* Characteristic found successfully 0 :
   - Characteristic start handle: A
   - Characteristic end handle: C
   - Characteristic value handle: B
   - Characteristic properties: 12
   - Characteristic uuid16: 0
   - Characteristic uuid128 : 87 1E 2 24 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2
* Characteristic found successfully 1 :
   - Characteristic start handle: D
   - Characteristic end handle: FFFF
   - Characteristic value handle: E
   - Characteristic properties: E
   - Characteristic uuid16: 0
   - Characteristic uuid128 : 87 1E 2 25 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2

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

Figura 5.27 - Descubrimiento de Características

Por último, se realizan una serie de procesos de escritura y lectura cada 10 segundos, como se muestra en la Figura 5.28. En primer lugar, se realiza una escritura en el dispositivo *Bluz DK* del valor 0x046E en hexadecimal y del valor 0x0304, empleando para ello Characteristic1. A continuación, se realiza una lectura del valor 0x0448696E a través de Characteristic0. Por último, se vuelve a repetir este proceso escribiendo los valores 0x046F y 0x0304, y realizando una lectura del valor 0x0448696F.

```
> Characteristic1_bluzdk_write:
   - data_bluzdk_ON_1
   - Connection handle: 64
   - Characteristic value attribute handle: 14
   - Characteristic value : 4 6E
* Write characteristic value done:
   - Connection handle: 40


> Characteristic1_bluzdk_write:
   - data_bluzdk_ON_2
   - Connection handle: 64
   - Characteristic value attribute handle: 14
   - Characteristic value : 3 4
* Write characteristic value done:
   - Connection handle: 40
```

```
> Characteristic0_bluzdk_read:
   - Connection handle: 64
   - Characteristic value attribute handle: 11
* Read characteristic value successfully:
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Characteristic value : 3


> Characteristic1_bluzdk_write:
   - data_bluzdk_OFF_1
   - Connection handle: 64
   - Characteristic value attribute handle: 14
   - Characteristic value : 4 6F
* Write characteristic value done:
   - Connection handle: 40


> Characteristic1_bluzdk_write:
   - data_bluzdk_OFF_2
   - Connection handle: 64
   - Characteristic value attribute handle: 14
   - Characteristic value : 3 4
* Write characteristic value done:
   - Connection handle: 40


> Characteristic0_bluzdk_read:
   - Connection handle: 64
   - Characteristic value attribute handle: 11
* Read characteristic value successfully:
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Characteristic value : 4 48 69 6F


_____ Device disconnected
        - Device disconnected handle: 40

_____ BLE Central restart scanning!
```

Figura 5.28 - Procesos de lectura y escritura

A partir de este punto, se comenzaron a experimentar sucesivos problemas relacionados con la habilitación de las notificaciones en el dispositivo *Bluz DK*, desde el dispositivo *RedBear Duo*, haciéndose uso de la función *ble.writeClientCharsConfigDescriptor().* El problema detectado impedía la posible transferencia de datos entre ambos dispositivos a través de notificaciones. Con el fin de determinar la causa de este problema, y su posible solución, se realizaron consultas en los foros de desarrolladores de los dispositivos *Bluz DK* y *RedBear Duo* **[20]**.

A partir de los análisis y los comentarios recogidos en estas consultas, se determinó finalmente que el problema residía en el dispositivo *Bluz DK,* y en concreto en la respuesta a una petición de lectura del valor CCCD actual del Descriptor, paso previo a su escritura, tal cual está implementada la función *ble.writeClientCharsConfigDescritor()* en el *firmware* del dispositivo *RedBear Duo*, para la posterior escritura del valor 0x1 con el fin de habilitar el envío de notificaciones desde el dispositivo *Central*. De forma más específica, el problema residía en el

proceso de lectura del valor CCCD cuando éste aún no había sido establecido. Básicamente, el valor CCCD no estaba establecido al iniciar una conexión (a no ser que el dispositivo *Central* realizase la escritura del mismo), y cuando se intentaba realizar una lectura se daba un caso de error especifico. Este caso de error estaba contemplado en el *firmware* del dispositivo *Bluz DK*, sin embargo, se encontraba comentado por alguna razón según lo explicado en las consultas al desarrollador del dispositivo. Esto dio lugar al reconocimiento por parte del equipo de soporte técnico de *Bluz DK* de un error en su *firmware*, realizando dicho equipo la *release* 2.1.50, mostrada en la Figura 5.29 en la que se soluciona este error y se indica que la mejora realizada se refiere a "*Fix to an issue where CCCD values couldn´t be read from the Central*" **[23]**. Una vez actualizada dicha versión del *firmware*, previamente a su *release* pública, se pudo comprobar la correcta habilitación de la notificación en el dispositivo *Bluz DK* por parte del dispositivo *RedBear Duo actuando* como dispositivo *Central*, mediante la función *ble.writeClientCharsConfigDescriptor()*.

Figura 5.29 – Release 2.1.50 del firmware del dispositivo Bluz DK

En concreto, la primera prueba se realizó ejecutando el código contenido en el fichero *localcomm_bluzdk_v01.ino* en el dispositivo *Bluz DK*, y el código *simpleblecentral-duo-bluzdk-v05.ino en el dispositivo RedBear Duo,* obteniéndose por el terminal el resultado mostrado en la Figura 5.30.

```
_____ BLE Central start scanning

* BLE scan callback:
   - Advertising event type: 0
   - Peer device address type: 1
   - Peer device address: E7 A8 6D 31 7D A
   - RSSI: -17
   - Advertising/Scan response data packet: 8 9 42 6C 75 7A 20 44 4B 3 19 0 0 2 1 6
      - AVD/SR data decoding -> ad_type: 9, length: 8
```

```
   - Complete Local Name: Bluz DK
_____ Found BluzDK

_____ Device connected
        - Device connected handle: 64

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

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

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

* Discover all services completed

_____ BLE Central start scanning

* BLE scan callback:
   - Advertising event type: 0
   - Peer device address type: 1
   - Peer device address: E7 A8 6D 31 7D A
   - RSSI: -17
   - Advertising/Scan response data packet: 8 9 42 6C 75 7A 20 44 4B 3 19 0 0 2 1 6
      - AVD/SR data decoding -> ad_type: 9, length: 8

   - Complete Local Name: Bluz DK
_____ Found BluzDK

_____ Device connected
        - Device connected handle: 64

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

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

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

* Discover all services completed

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

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

* Discover all characteristics completed

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

* Discover all descriptors completed

* Discover all descriptors completed

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

RETURN CODE: 0

D --- gattWriteCCCDCallback (1)

* Write CCCD value successfully
   - CCCD value: 1

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

_____ Notifications enabled

* Received new notification (26789 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 1 10 0 1 2 1 6 16 33 2 0 16 33 0 0 0 0 0 0 0

* Received new notification (28 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 0 0 0 0 0

* Received new notification (60 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4

_____ Device disconnected
        - Device disconnected handle: 40
```

Figura 5.30 – Correcto lectura y escritura del valor CCCD

5.5 Integración del módulo TGAM con la plataforma BLE formada por los dispositivos *RedBear Duo* y *Bluz DK*.

El siguiente paso a realizar con el objetivo final de integrar un dispositivo que permita transferir la información de las ondas EEG a un dispositivo *Bluz DK* con el fin de poder enviar los paquetes en formato RAW capturados, hasta un dispositivo *Redbear Duo* mediante el protocolo BLE para su registro y posterior procesamiento, consistió en la conexión serie entre el dispositivo *Bluz DK* y el módulo TGAM integrado en el

dispositivo *Mindflex* a través de la interfaz serie *Serial1*, y el envío, por parte del dispositivo *Bluz DK*, de los valores recibidos, mediante una conexión BLE, como se muestra en la Figura 5.31.

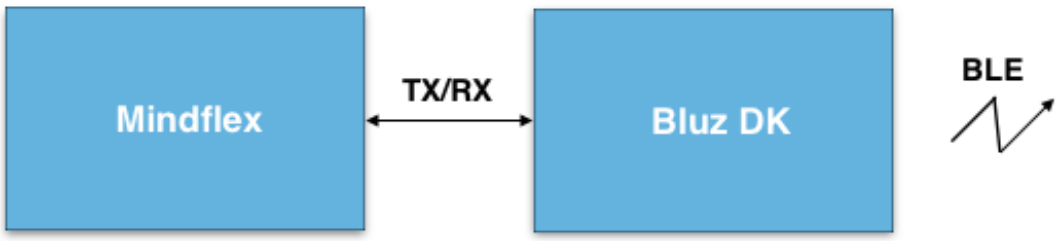


Figura 5.31 - Conexión Mindflex - Bluz DK

El primer paso para lograr este objetivo consiste en verificar que, de acuerdo a la tasa de transferencia a la que el módulo TGAM envía el valor de muestras en formato RAW, y a la tasa máxima de transferencia soportada por el estándar Bluetooth 4.0, y más en concreto, por los dispositivos *Bluz DK* y *RedBear Duo*, esta comunicación es posible.

Así, a partir de las especificaciones proporcionadas por *Neurosky* en el documento *ThinkGear Serial Stream Guide*, en modo RAW (16 bits), el módulo TGAM proporciona un valor (de 2 bytes), a una tasa de transferencia de 512 muestras por segundo, o una vez cada 1,95 milisegundos (aproximadamente cada 2 milisegundos), lo que equivale a una tasa de 1024 bytes/segundo. Por otra parte, en modo NORMAL, los datos proporcionados por el dispositivo TGAM, correspondientes a *Signal Quality*, *EEG Power*, *Attention* y *Meditation*, que representan un mínimo de 27 bytes, se envían a una tasa de transferencia de una vez por segundo, lo que equivale a una tasa de 27bytes/segundo. Estos datos teóricos se han validado en la práctica modificando el código correspondiente al *firmware BrainSerialTest-DuoV05.ino* incorporando en *modeDebug = True* información relativa al intervalo de tiempo entre la recepción de dos paquetes consecutivos desde el dispositivo *Mindflex*, tanto en modo NORMAL, como en modo RAW, como se muestra en la Figura 32 y en la Figura 33.

```
--- Start Packet (483)
    Lapsed Time (ms): 2
    Raw Value: -913
--- End Packet (483)

--- Start Packet (484)
    Lapsed Time (ms): 2
    Raw Value: -1239
--- End Packet (484)

--- Start Packet (485)
    Lapsed Time (ms): 2
    Raw Value: -1331
--- End Packet (485)

--- Start Packet (486)
    Lapsed Time (ms): 2
    Raw Value: -1604
--- End Packet (486)

--- Start Packet (487)
    Lapsed Time (ms): 2
    Raw Value: -1875
--- End Packet (487)
```

Figura 5.32 – modeRaw = true

```
Start packet (2)
Lapsed Time (ms): 995
Signal Quality: 29
Attention: 0
Meditation: 0
EEG POWER:
    Delta: 220142
    Theta: 19785
    Low Alpha: 3546
    High Alpha: 4432
    Low Beta: 3604
    High Beta: 2610
    Low Gamma: 507
    Mid Gamma: 22613
End   Packet (2)

Start packet (3)
Lapsed Time (ms): 995
Signal Quality: 29
Attention: 0
Meditation: 0
EEG POWER:
    Delta: 374315
    Theta: 15054
    Low Alpha: 7519
    High Alpha: 1736
    Low Beta: 2334
    High Beta: 1717
    Low Gamma: 505
    Mid Gamma: 19956
End   Packet (3)
```

Figura 5.33 - modeRaw = false

Por otro lado, en lo que respecta a los módulos *Bluz DK* y *RedBear Duo*, el estándar establece que entre un dispositivo *Central* y un dispositivo *Peripheral* conectados entre sí, el intervalo de tiempo entre dos eventos consecutivos (un intercambio de datos antes de pasar al modo IDLE para ahorrar consumo) está determinado por el valor del parámetro *Connection Interval*, especificado en el dispositivo *Peripheral* entre los parámetros del campo PPCP, y cuyo valor debe estar comprendido entre un mínimo de 7.5 milisegundos (*high throughput*) y un máximo de 4 segundos (*low throughput*).

En este intervalo de conexión se puede transferir más de un paquete, cada uno de ellos con un *Payload* máximo de 20 bytes. Así, siendo *P* el número máximo de paquetes que es posible transferir por intervalo de conexión, la tasa de transferencia máxima soportada por un dispositivo BLE será:

*(1 evento / 7,5 ms) * (P*20 bytes) = 2660 * P * bytes/s*

De manera que, aunque en cada intervalo de conexión se envíe un único paquete de 20 bytes, (P=1), la tasa de transferencia que podrían soportar los dispositivos BLE sería

suficiente teóricamente para transferir los datos en modo RAW proporcionados por el módulo TGAM.

En este punto, un aspecto a tener en cuenta en el dispositivo *Bluz DK,* es que en los 20 bytes que como máximo se pretenden enviar en cada paquete, este dispositivo introducirá el *byte* de cabecera con el valor 0x04, además de un paquete final con los *bytes* de cola, de valor 0x0304 en hexadecimal.

Por otra parte, en cuanto al dispositivo *RedBear Duo,* el valor del intervalo de conexión, que se desea establecer al valor mínimo de 7,5 milisegundos, está fijado en el *firmware* del dispositivo a un intervalo comprendido entre 10 milisegundos y 30 milisegundos, aunque es posible modificarlo en el código mediante una llamada a la función *gap_update_connection_parameters()* de la librería BTStack tras establecer la conexión con el dispositivo *Peripheral*. Sin embargo, dicha función no puede utilizarse desde una aplicación de usuario en el dispositivo *RedBear Duo*.

Por tanto, la única posibilidad que existiría en el dispositivo *RedBear Duo* para establecer los parámetros asociados a la conexión con un dispositivo *Peripheral*, entre ellos el valor del intervalo de conexión, sería en principio utilizar la función *setConnParams(),* como se muestra en la Figura 5.34.

```
#define MIN_CONN_INTERVAL          0x0008
#define MAX_CONN_INTERVAL          0x0009
#define SLAVE_LATENCY              0x0000
#define CONN_SUPERVISION_TIMEOUT   0x03E8
.
.
le_connection_parameter_range_t connection_parameter_range;
.
.
void setup() {
  connection_parameter_range.le_conn_interval_min = MIN_CONN_INTERVAL;
  connection_parameter_range.le_conn_interval_max = MAX_CONN_INTERVAL;
  connection_parameter_range.le_conn_latency_min = SLAVE_LATENCY;
  connection_parameter_range.le_conn_latency_max = SLAVE_LATENCY;
  connection_parameter_range.le_supervision_timeout_min = CONN_SUPERVISION_TIMEOUT;
  connection_parameter_range.le_supervision_timeout_max = CONN_SUPERVISION_TIMEOUT;
.
.
  ble.setConnParams(connection_parameter_range);
```

Figura 5.34 - Parámetros asociados a la conexión

En este punto, como primer paso a implementar, se decidió abordar la transferencia de datos entre el módulo TGAM y el dispositivo *Bluz DK* mediante la conexión serie entre ambos. Para ello, se desarrolló un código a ejecutar en el dispositivo *Bluz DK*, contenido en el fichero *brainserialtest-bluzdk-v01.ino*, en el que se realiza la transferencia, vía BLE, de la información recibida en el dispositivo *Bluz DK* a través de la interfaz serie *Serial1*, desde el dispositivo *Mindflex*, usando para ello la función definida en la librería *Parser.cpp*.

```
#include "Parser.h"
#include "application.h"

boolean modeDebug = false;
boolean modeRaw = true;
uint8_t *buffer;
uint8_t resp_buffer[19];

Parser parser(Serial1, modeRaw, modeDebug);


void setup() {
    pinMode(D7, OUTPUT);
    digitalWrite(D7, LOW);
    for (int i=0; i<19; i++)
        resp_buffer[i]=i;

    if (modeRaw) {
        Serial1.begin(57600);
    }
    else {
        Serial1.begin(9600);
    }
}

void loop() {
    System.sleep(SLEEP_MODE_CPU);
    if (brain.update())
    {
        buffer = parser.readCSV();
        BLE.sendData(buffer, 6);
        digitalWrite(D7, HIGH);
    }
    else
    {
        digitalWrite(D7, LOW);
    }
}
```

Figura 5.35 - Fichero brainserialtest-bluzdk-v01.ino

La principal modificación realizada en la librería *Parser* está relacionada con la función *parser.readCSV(),* con el fin de adaptar los datos recibidos desde el módulo TGAM, tanto en modo RAW como en modo NORMAL, a la información a incluir en los paquetes BLE, como se muestra en la sección de código de la Figura 5.36.

```
162 uint8_t * Parser::readCSV() {
163     if (modeRaw && hasRaw) {
164         csvBuffer[0] = rawValue & 0xff;
165         csvBuffer[1] = rawValue >> 8;
166         return csvBuffer;
167     }
168     else if (!modeRaw) {
169         csvBuffer[0] = signalQuality;
170         csvBuffer[1] = attention;
171         csvBuffer[2] = meditation;
172         for(int i = 0; i < EEG_POWER_BANDS; i++) {
173             csvBuffer[(i*3)+3] = eegPower[i] & 0x000000ff;
174             csvBuffer[(i*3)+4] = (eegPower[i] & 0x0000ff00) >> 8;
175             csvBuffer[(i*3)+5] = (eegPower[i] & 0x00ff0000) >> 16;
176         }
177         return csvBuffer;
178     }
179 }
```

Figura 5.36 - Función parser.readCSV()

Tras estas modificaciones, fue posible compilar el código correspondiente al fichero *brainserialtest-bluzdk-v01.ino,* así como validar su correcto funcionamiento, en este como mediante el uso de la *app nRF Connect (Android)* **[25]** , que actuando como dispositivo *Central* permitió la conexión con el dispositivo *Bluz DK* y la correcta recepción de los paquetes transmitidos por éste con una tasa de un paquete cada segundo, al estar inicialmente el código en el dispositivo *Bluz DK* configurado para la operación del dispositivo *Mindflex* en modo NORMAL. Para ello, fue necesario inicialmente habilitar la recepción de NOTIFICACIONES en las características del dispositivo *Bluz DK*, desde la *app nRF Connect*.

Por otro lado, aunque en modo NORMAL la cantidad de información enviada a través de la interfaz serie *Serial1* desde el módulo TGAM supera los 20 bytes que pueden incluirse como máximo en cada uno de los paquetes transmitidos entre los dispositivos *Central* y *Peripheral* (en concreto, esta información sería de 1 byte de *Signal Quality* + 8*3 bytes de EEG *Power* + 1 byte de *Attention* + 1 byte de *Meditation* que hacen un total de 27 bytes) el comando *ble.sendData()* del *firmware* del dispositivo *Bluz DK* tomará los 27 bytes y los separará en paquetes de 20 bytes, aunque, incluyendo el paquete de cola con el contenido 0x0304 los datos pueden ser reconstruidos en el otro extremo de la conexión.

En todo caso, con el fin de validar experimentalmente este hecho, se decidió modificar el fichero *serialtestbluzdk_v01.ino* de forma que la longitud del vector que se enviara mediante el comando *ble.sendData()* fuera de 30 bytes, con valores ascendentes y

descendentes en cada envío, comprendidos en los rangos de 0x00 a 0x1D y 0x1D a 0x00, respectivamente, generándose el código del fichero *serialtestbluzdk-v02.ino*, mostrado en la Figura 5.37.

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

boolean sendData = false;
boolean turn = false;
uint8_t idx = 0;
uint8_t sav_data[30];


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

    pinMode(D6, INPUT_PULLDOWN);
    if (digitalRead(D6) == HIGH) {
        SYSTEM_MODE(AUTOMATIC);
    }
    for (int i=0; i<30; i++)
        sav_data[i] = i;
}


void loop() {
    System.sleep(SLEEP_MODE_CPU);

    if (BLE.getState() == BLE_CONNECTED) {
        if (sendData)
        {
            digitalWrite(D7, HIGH);
                BLE.sendData(sav_data, 30);
                sendData = false;
        }
        else {
            if (turn) {
                turn = false;
                for (int i=0; i<30; i++)
                    sav_data[i] = 29-i;
            }
            else {
                turn = true;
                for (int i=0; i<30; i++)
                   sav_data[i] = i;
            }
            digitalWrite(D7, LOW);
            sendData = true;
        }
    delay(1000);
    }
}
```

Figura 5.37 - Fichero serialtestbluzdk_v02.ino

A partir de la ejecución de este fichero en el dispositivo *Bluz DK*, se pudo comprobar el correcto envío de estos vectores en 2 *chunks* de 20 bytes (contando la cabecera 0x04) y 11 bytes, más el paquete de cola con el valor 0x0304, como se muestra en la Figura 5.38.

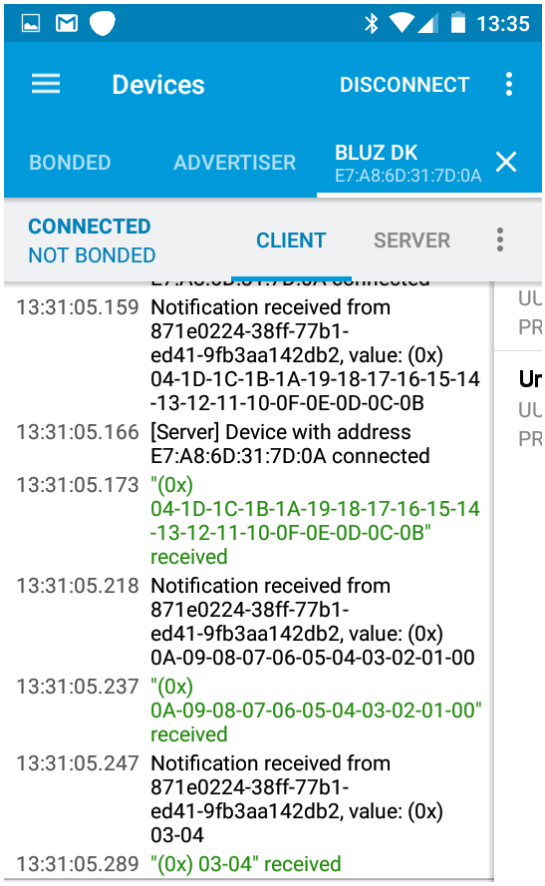


Figura 5.38 - Envío de los vectores en 2 chunks de 20 bytes

A partir de esta verificación, se introdujo una modificación en el fichero *brainserialtest-bluzdk-v01.ino* con el fin de que, en modo NORMAL, se incluyera en los datos a enviar, los 24 bytes correspondientes a la información de EEG-POWER. Lo que dió lugar al fichero *brainserialtest-bluzdk-v02.ino*, mostrado en la Figura 5.39.

```
#include "Parser.h"
#include "application.h"

boolean modeDebug = false;
boolean modeRaw = true;
uint8_t *buffer;
int buffer_size;

Parser parser(Serial1, modeRaw, modeDebug);

SYSTEM_MODE(MANUAL);

void setup() {
    pinMode(D6, INPUT_PULLDOWN);
    if (digitalRead(D6) == HIGH) {
        SYSTEM_MODE(AUTOMATIC);
    }
    pinMode(D7, OUTPUT);
    digitalWrite(D7, LOW);
```

```
    if (modeRaw) {
        Serial1.begin(57600);
        Serial.begin(57600);
        buffer_size = 2;
    }
    else {
        Serial1.begin(9600);
        Serial.begin(9600);
        buffer_size = 27;
    }
}


void loop() {
    System.sleep(SLEEP_MODE_CPU);
    if (BLE.getState() == BLE_CONNECTED) {
        if (parser.update())
        {
            buffer = parser.readCSV();
            BLE.sendData(buffer, buffer_size);
            digitalWrite(D7, HIGH);
        }
        else
        {
            digitalWrite(D7, LOW);
        }
    }
}
```

Figura 5.39 - Fichero brainserialtest-bluzdk-v02.ino

Además, se realizaron las modificaciones necesarias en los ficheros de la librería *Parser.cpp*, mostradas en la Figura 5.40.

```
uint8_t * Parser::readCSV() {
    if (modeRaw && hasRaw) {
        csvBuffer[0] = rawValue & 0xff;
        csvBuffer[1] = rawValue >> 8;
        return csvBuffer;
    }
    else if (!modeRaw) {
        Serial.print(signalQuality, HEX);
        Serial.print(",");
        Serial.print(attention, HEX);
        Serial.print(",");
        Serial.print(meditation, HEX);

        if (hasPower) {
            for(int i = 0; i < EEG_POWER_BANDS; i++) {
                Serial.print(",");
                Serial.print(eegPower[i], HEX);
            }
        }

        csvBuffer[0] = signalQuality;
        csvBuffer[1] = attention;
        csvBuffer[2] = meditation;
        for(int i = 0; i < EEG_POWER_BANDS; i++) {
            csvBuffer[(i*3)+3] = eegPower[i] & 0x000000ff;
            csvBuffer[(i*3)+4] = (eegPower[i] & 0x0000ff00) >> 8;
            csvBuffer[(i*3)+5] = (eegPower[i] & 0x00ff0000) >> 16;
        }
        return csvBuffer;
    }
}
```

Figura 5.40 - Modificaciones fichero Parser.cpp

Así a partir de la ejecución del código *brainserialtest-bluzdk-v02.ino* en el dispositivo *Bluz DK*, se pudo comprobar, haciendo de nuevo uso de la *app nRF Connect (Android)* la correcta recepción de los 27 bytes de datos asociados a cada muestra del módulo TGAM en modo NORMAL, en dos paquetes, el primero de ellos conteniendo, además de la cabecera con el valor 0x04, 19 bytes de datos, y el segundo con los restantes 8 bytes, además de un tercer paquete de cola con el valor 0x0304.

Finalmente, se modificó el código de fichero *brainserialtest-bluzdk-v02.ino* para incorporar el envío de datos recibidos desde el módulo TGAM en modo RAW, que consisten en una muestra de 2 bytes aproximadamente cada 2 milisegundos, o bien 512 muestras de 2 bytes, por segundo. Con el fin de acondicionar esta tasa de datos a las limitaciones relativas al número de paquetes de 20 bytes que pueden transferirse por intervalo de conexión entre un dispositivo *Peripheral* y un dispositivo *Central*, en el código del fichero *brainserialtest-bluzdk-v03.ino* se decidió incorporar para el modo RAW un buffer dimensionado con un tamaño inicial de 58 bytes, lo que se correspondería con la transferencia de 3 *chunks* de 20 bytes (en realidad de 19 bytes+20 bytes+19 bytes) más un último paquete de cola con el valor 0x0304, por cada llamada a la función *ble.sendData()* en el dispositivo *Bluz DK*, si bien este valor se ha definido a el parámetro RESP_BUFFER_SIZE_RAW que puede ser modificado en el futuro.

Una vez programado el dispositivo *Bluz DK* con este código, se comprobó con la *app nFR Connect*, tanto que seguía obteniéndose un correcto funcionamiento en la transferencia de datos desde el módulo TGAM en modo NORMAL, como un correcto funcionamiento en modo RAW, según se muestra en el *log* de la Figura 5.41.

```
I       11:18:16.149    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 04-
E5-07-FF-07-FF-07-FF-07-FF-07-56-00-00-F8-00-F8-00-F8-00
I       11:18:16.156    [Server] Device with address E7:A8:6D:31:7D:0A connected
A       11:18:16.198    "(0x) 04-E5-07-FF-07-FF-07-FF-07-FF-07-56-00-00-F8-00-F8-00-F8-00" received
V       11:18:16.205    [Server] Creating server connection...
I       11:18:16.213    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) F8-
B7-00-FF-07-FF-07-FF-07-FF-07-FF-07-9A-05-00-F8-00-F8-00-F8-00
D       11:18:16.220    server.connect(device, autoConnect = false)
A       11:18:16.309    "(0x) F8-B7-00-FF-07-FF-07-FF-07-FF-07-9A-05-00-F8-00-F8-00-F8-00" received
I       11:18:16.318    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) F8-5D-
FB-FF-07-FF-07-FF-07-FF-07-FF-07-3E-F8-00-F8-00-F8
A       11:18:16.353    "(0x) F8-5D-FB-FF-07-FF-07-FF-07-FF-07-FF-07-3E-F8-00-F8-00-F8" received
I       11:18:16.362    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 03-04
A       11:18:16.399    "(0x) 03-04" received
D       11:18:16.406    [Server callback] Connection state changed with status: 0 and new state: CONNECTED
(2)
I       11:18:16.414    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 04-05-
F8-FF-07-FF-07-FF-07-FF-07-FF-07-AA-FE-00-F8-00-F8-00
I       11:18:16.421    [Server] Device with address E7:A8:6D:31:7D:0A connected
A       11:18:16.459    "(0x) 04-05-F8-FF-07-FF-07-FF-07-FF-07-FF-07-AA-FE-00-F8-00-F8-00" received
V       11:18:16.469    [Server] Creating server connection...
I       11:18:16.476    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) F8-00-
F8-99-03-FF-07-FF-07-FF-07-FF-07-7A-03-00-F8-00-F8-00
D       11:18:16.483    server.connect(device, autoConnect = false)
A       11:18:16.520    "(0x) F8-00-F8-99-03-FF-07-FF-07-FF-07-FF-07-7A-03-00-F8-00-F8-00" received
I       11:18:16.533    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) F8-00-
F8-2A-FE-FF-07-FF-07-FF-07-FF-07-22-07-4B-F8-00-F8
A       11:18:16.571    "(0x) F8-00-F8-2A-FE-FF-07-FF-07-FF-07-FF-07-22-07-4B-F8-00-F8" received
I       11:18:16.584    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 03-04
A       11:18:16.628    "(0x) 03-04" received
I       11:18:16.639    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 04-00-
F8-83-FA-FF-07-FF-07-FF-07-FF-07-FF-07-FD-FD-00-F8-00
D       11:18:16.653    [Server callback] Connection state changed with status: 0 and new state: CONNECTED
A      11:18:17.704    "(0x) 04-96-02-9D-FD-67-FA-FA-F9-23-FA-7D-FD-0A-04-10-07-E6-05-12" received
I      11:18:17.739    [Server] Device with address E7:A8:6D:31:7D:0A connected
I      11:18:17.746    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 05-8A-03-48-FF-
F5-FA-63-FA-48-FA-5C-FC-0D-02-C9-06-30-06
V      11:18:17.753    [Server] Creating server connection...
A      11:18:17.759    "(0x) 05-8A-03-48-FF-F5-FA-63-FA-48-FA-5C-FC-0D-02-C9-06-30-06" received
D      11:18:17.794    server.connect(device, autoConnect = false)
I      11:18:17.800    Notification received from 871e0224-38ff-77b1-ed41-9fb3aa142db2, value: (0x) 03-04
A      11:18:17.809    "(0x) 03-04" received
```

Figura 5.41 - Fichero log generado por la app nRF Connect

5.6 Dispositivo *Bluz DK* como dispositivo Peripheral y *RedBear Duo* como dispositivo *Central*

Posteriormente, se procedió a analizar el *Throughput* en la transferencia de datos entre un dispositivo *Bluz DK* actuando como *Peripheral,* y un dispositivo *RedBear Duo* como *Central*, con el fin de validar la correcta transferencia de muestras capturadas por el módulo TGAM, en modo RAW. Así, por un lado, en este modo de funcionamiento, el dispositivo *Mindflex* genera una muestra de 2 bytes a una tasa de 512 muestras/segundo. A esta tasa, y teniendo en cuenta que cada paquete del protocolo BLE puede tener un tamaño de 20 bytes (19 en el caso de emplear el dispositivo *Bluz DK* como *Peripheral*, al introducir en el primer paquete de cada transmisión un byte de valor 0x04), debería ser posible enviar entre ambos dispositivos, 1 paquete (10 muestras) cada 19´53125 milisegundos para soportar la tasa de generación de muestras del dispositivo *Mindflex* en modo RAW. En este sentido, según la definición del estándar BLE, la transferencia de paquetes entre un dispositivo *Peripheral* y un dispositivo *Central* puede realizarse mientras la conexión

esté activa, hecho que está determinado por el parámetro correspondiente al intervalo de conexión (*connection interval*), cuyo valor puede definirse teóricamente entre 7.5 milisegundos y 4 segundos. Así el intervalo de conexión determina el número de eventos de conexión (periodos de intercambio de paquetes) que pueden darse por unidad de tiempo. En principio, en cada intervalo de conexión, el estándar BLE, permite la transferencia de más de un paquete entre el dispositivo *Central* y el dispositivo *Peripheral*, dependiendo de las características de cada dispositivo concreto, pudiendo variar entre un mínimo de 1 y un máximo de 6.

Así, para soportar la tasa de muestras del módulo TGAM en modo RAW, existen varias alternativas:

- En caso de establecer el número de paquetes que es posible trasferir entre un dispositivo *Central* y un dispositivo *Peripheral* a 1, el intervalo de conexión debería establecerse a un valor menor a 19.53125 milisegundos.
- En caso de poder establecer un número de paquetes transferidos por intervalo de conexión superior a 1, el valor del intervalo de conexión podría ser mayor que 19.53125 milisegundos. Así, considerando el envío de, por ejemplo 2 paquetes por intervalo de conexión, y una latencia despreciable entre el envío de ambos, el valor del intervalo de conexión debería ser en esta ocasión menor de 39.0625 milisegundos.

Con todo, con el fin de analizar el *Throughput* que es posible alcanzar en la transferencia de paquetes entre un dispositivo *Bluz DK* y un dispositivo *RedBear Duo*, y con ello la viabilidad de soportar la transferencia de muestras generadas por el módulo TGAM en modo RAW, se realizó un primer experimento a partir del código correspondiente al fichero *brainserialtest-bluzdk-v02.ino* en el que, para el caso del modo RAW en el módulo TGAM, se enviaba cada muestra de 2 bytes en un paquete, con el fin de analizar el peor caso.

Así, en el caso concreto de utilizar el dispositivo *Bluz DK* como *Pheripheral* para el envío de las muestras generadas por el módulo TGAM, en el fichero *log* de la Figura 5.42 se observa cómo, considerando el hecho de que este dispositivo envía un byte de

cabecera con el valor 0x04 en el primer paquete generado para el envío de los datos especificados en la llamada a la función *BLE.SendData()* (en este caso, un único paquete), y que envía un paquete adicional de cola con el valor 0x0304, en condiciones normales se transfiere un paquete cada aproximadamente 30 milisegundos al dispositivo *RedBear Duo*, latencia que se corresponde prácticamente con el valor del intervalo de conexión, que inicialmente se consideraba que se establecía a un valor por defecto de 30 milisegundos (posteriormente se justificará esta consideración), deduciéndose así en primera instancia que en este escenario, únicamente se enviaba un paquete por intervalo de conexión, a pesar de que cada llamada a la función *BLE.SendData()* conllevara el envío de 2 paquetes consecutivos.

```
_____ BLE Central <-> BluzDK _____
_____ BLE Central start scanning

* BLE scan callback:
   - Advertising event type: 0
   - Peer device address type: 1
   - Peer device address: 65 51 62 D5 C6 2A
   - RSSI: -89
   - Advertising/Scan response data packet: 2 1 6 13 FF 4C 0 C E 0 FC 88 D2 67 70 1E E8 D5
BF 68 4B 0 FE
      - AVD/SR data decoding -> ad_type: 1, length: 2
      - AVD/SR data decoding -> ad_type: FF, length: 13

* BLE scan callback:
   - Advertising event type: 4
   - Peer device address type: 1
   - Peer device address: 65 51 62 D5 C6 2A
   - RSSI: -89
   - Advertising/Scan response data packet:

* BLE scan callback:
   - Advertising event type: 0
   - Peer device address type: 1
   - Peer device address: E7 A8 6D 31 7D A
   - RSSI: -45
   - Advertising/Scan response data packet: 8 9 42 6C 75 7A 20 44 4B 3 19 0 0 2 1 6
      - AVD/SR data decoding -> ad_type: 9, length: 8

   - Complete Local Name: Bluz DK
_____ Found BluzDK

_____ Device connected
        - Device connected handle: 64
* Service found successfully
   - Service start handle: 1
   - Service end handle: 7
   - Service uuid16: 1800
   - Service uuid128 : 0 0 18 0 0 0 10 0 80 0 0 80 5F 9B 34 FB

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

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

```
* Discover all services completed

* Characteristic found successfully 0 :
   - Characteristic start handle: A
   - Characteristic end handle: C
   - Characteristic value handle: B
   - Characteristic properties: 12
   - Characteristic uuid16: 0
   - Characteristic uuid128 : 87 1E 2 24 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2
* Characteristic found successfully 1 :
   - Characteristic start handle: D
   - Characteristic end handle: FFFF
   - Characteristic value handle: E
   - Characteristic properties: E
   - Characteristic uuid16: 0
   - Characteristic uuid128 : 87 1E 2 25 38 FF 77 B1 ED 41 9F B3 AA 14 2D B2

* Discover all characteristics completed

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

* Discover all descriptors completed

* Discover all descriptors completed

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

D --- gattWriteCCCDCallback (1)

* Write CCCD value successfully
   - Connection handle: 40
   - CCCD value: 1
D --- gattReadDescriptorCallback (
* Read descriptor value successfully:
   - Connection handle: 40
   - Descriptor value attribute handle: C
   - Descriptor value : 1 0

_____ Notifications enabled

* Received new notification (77323 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 B9 0

* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4
* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 1D 1

* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4

* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 35 1

* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4

* Received new notification (390 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 1A 1
```

```
* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4
```

Figura 5.42 - Fichero log envío de más de un paquete por intervalo de conexión desde el dispositivo Bluz DK

Además, se observó que había ocasiones en las que esta latencia entre el envío de paquetes aumentaba hasta un valor de unos 390 milisegundos, hecho que en principio no tiene una explicación concreta, ni siquiera por parte del equipo de soporte del dispositivo *Bluz DK* (aunque como se comprobará posteriormente este hecho no se observa al capturar los datos con el dispositivo *Bluefruit LE Sniffer*).

Así, en la Figura 5.43, se muestra el código del fichero *brainserialtest-bluzdk-v04.ino* disponible en el entorno *Partide IDE*, en el que, en modo RAW, ahora es posible configurar el tamaño del *buffer* en el que se irán almacenando temporalmente las muestras proporcionadas por el módulo TGAM al dispositivo *Bluz DK*, hasta su envío, en este caso a la *app LightBlue* (iOS) actuando como dispositivo *Central*, en tantos paquetes consecutivos como sea necesario en cada caso.

```
#include "Brain.h"
#include "application.h"

boolean modeRaw = true;
boolean modeDebug = false;

#define RESP_BUFFER_SIZE_RAW 48

#define BUFFER_SIZE_NORMAL 27

uint8_t resp_buffer[RESP_BUFFER_SIZE_RAW];
int resp_buffer_idx;
uint8_t *buffer;

Brain brain(Serial1, modeRaw, modeDebug);

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

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

    if (modeRaw) {
        Serial1.begin(57600);
        resp_buffer_idx = 0;
    }
    else {
        Serial1.begin(9600);
    }
}
```

```
38  void loop() {
39      System.sleep(SLEEP_MODE_CPU);
40      if (BLE.getState() == BLE_CONNECTED) {
41          if (brain.update())
42          {
43              buffer = brain.readCSV();
44              if (modeRaw) {
45                  resp_buffer[resp_buffer_idx++] = buffer[0];
46                  resp_buffer[resp_buffer_idx++] = buffer[1];
47                  if (resp_buffer_idx > (RESP_BUFFER_SIZE_RAW-1)) {
48                      BLE.sendData(resp_buffer, RESP_BUFFER_SIZE_RAW);
49                      resp_buffer_idx = 0;
50                      digitalWrite(D7, HIGH);
51                  }
52              }
53              else {
54                  BLE.sendData(buffer, BUFFER_SIZE_NORMAL);
55                  digitalWrite(D7, HIGH);
56              }
57          }
58          else
59          {
60              digitalWrite(D7, LOW);
61          }
62      }
63  }
```

Figura 5.43 - Fichero brainserialtest-bluzdk-v04.ino

En este punto, en la Figura 5.44, la Figura 5.45, la Figura 5.46 y la Figura 5.47, se muestran los resultados obtenidos a partir del fichero *log* generado por la *app LightBlue*, para los casos en los que el tamaño de este *buffer*, determinado en el *firmware* de usuario por el parámetro RESP_BUFFER_SIZE_RAW, es de 48 bytes (3+1paquetes consecutivos), 78 (4+1 paquetes consecutivos), 118 (6+1 paquetes consecutivos) y 158 (8+1 paquetes consecutivos), respectivamente.

```
11:24:41.748 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <01ff07ff 07ff07ff 07930500
f800f800 f800f825>
11:24:41.749 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <fbff07ff 07ff07ff 07>
11:24:41.749 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
11:24:41.777 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <04ff071b fa00f800 f800f800
f8ff07ff 07ff07ff>
11:24:41.779 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff071c 0000f800 f800f800
f84403ff 07ff07ff>
11:24:41.779 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff0756 0400f800 f8>
11:24:41.780 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
```

Figura 5.44 - Fichero log de la app LightBlue con intervalo de conexión i igual a 4

```
11:34:24.089 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <04630000 f800f800 f800f851
01ff07ff 07ff07ff>
11:34:24.090 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07490500 f800f800 f800f8b5
faff07ff 07ff07ff>
11:34:24.090 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff0700 f800f800 f800f800
f8ff07ff 07ff07ff>
11:34:24.091 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff0777 fd00f800 f800f800
f88204ff 07ff07>
11:34:24.092 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
11:34:24.188 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <04ff07ff 07730300 f800f800
f800f8e8 fcff07ff>
11:34:24.193 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07c60700 f800f800
f800f848 738ff8ff>
11:34:24.194 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff07ff 0773fa00
f800f800 f800f8f1>
11:34:24.194 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff07ff 07470100
f800f800 f800f8>
11:34:24.195 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
```

Figura 5.45 - Fichero log de la app LightBlue con intervalo de conexión i igual a 5

```
10:28:39.564 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0400f800 f8e707ff 07ff07ff
07ff0706 0200f800>
10:28:39.566 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f800 f8d5fdff 07ff07ff
07ff07ff 0700f800>
10:28:39.566 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f800 f800f8ff 07ff07ff
07ff07ff 0700f800>
10:28:39.567 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f800 f800f8ff 07ff07ff
07ff07ff 07befd00>
10:28:39.567 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f800 f800f891 01ff07ff
07ff07ff 07d90600>
10:28:39.568 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f800 f800f884 f8ff07ff
07ff07ff 07ff07>
10:28:39.593 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
10:28:39.685 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0400f800 f800f800 f800f8ff
07ff07ff 07ff07ff>
10:28:39.686 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0735fa00 f800f800 f800f8a5
05ff07ff 07ff07ff>
10:28:39.686 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07b00300 f800f800 f800f822
fbff07ff 07ff07ff>
10:28:39.687 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff0700 f800f800 f800f800
f8ff07ff 07ff07ff>
10:28:39.687 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff0700 f800f800 f800f800
f8ff07ff 07ff07ff>
10:28:39.687 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff0759 0000f800 f800f800
f8cbfeff 07ff07>
10:28:39.713 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
```

Figura 5.46 - Fichero log de la app LightBlue con intervalo de conexión i igual a 6 (1)

```
11:11:31.610 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0400f8ff 0700f800 f800f800
f800f8ff 07ff07ff>
11:11:31.611 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07affa00 f800f800
f800f8e7 03ff07ff>
11:11:31.612 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07000400 f800f800
f800f8a6 faff07ff>
11:11:31.612 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff0700 f800f800
f800f800 f8ff07ff>
11:11:31.612 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff0700 f800f800
f800f800 f8ff07ff>
11:11:31.613 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff074b 0000f800
f800f800 f8fdfdff>
11:11:31.638 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff07fb 0700f800
f800f800 f800f8ff>
11:11:32.149 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <07ff07ff 07ff07ff 0700f800
f800f800 f800f8>
11:11:32.149 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
11:11:32.300 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <04ff07ff 07ff07ff 0769fa00
f800f800 f800f888>
11:11:32.301 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <04ff07ff 07ff07ff 074b0300
f800f800 f800f817>
11:11:32.302 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <fbff07ff 07ff07ff 07ff0700
f800f800 f800f800>
11:11:32.302 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f8ff07ff 07ff07ff 07ff0700
f800f800 f800f800>
11:11:32.302 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f8ff07ff 07ff07ff 07ff0726
0000f800 f800f800>
11:11:32.303 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f864feff 07ff07ff 07ff07ff
0700f800 f800f800>
11:11:32.329 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f8ff 07ff07ff 07ff07ff
0700f800 f800f800>
11:11:32.839 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <f800f8ff 07ff07ff 07ff07ff
076cfc00 f800f8>
11:11:32.839 ó Characteristic (871E0224-38FF-77B1-ED41-9FB3AA142DB2) notified: <0304>
```

Figura 5.47 - Fichero log de la app LightBlue con intervalo de conexión i igual a 6 (2)

A partir de estos resultados, en los que quedaba inicialmente verificado que la limitación en el número de paquetes enviados por intervalo de conexión a 1 estaba determinada por el dispositivo *RedBear Duo* actuando como *Central*, se decidió ir un paso más allá y utilizar un dispositivo *Sniffer* que permitiera capturar todos los paquetes transferidos entre un dispositivo *Peripheral* y el dispositivo *Central* compatibles con el estándar BLE, con el fin de validar los datos obtenidos hasta ahora y las conclusiones inicialmente establecidas.

Para ello se usó el dispositivo *Bluefruit LE Sniffer – nRF51822,* mostrado en la Figura 5.48, de la empresa *Adafruit*, siendo necesario instalar las aplicaciones *nrF Sniffer* de *Nordic* para *Windows* **[26]**, además de la versión 1.12.13 de la aplicación *Wireshark **[27***]**, para el análisis de los paquetes capturados.

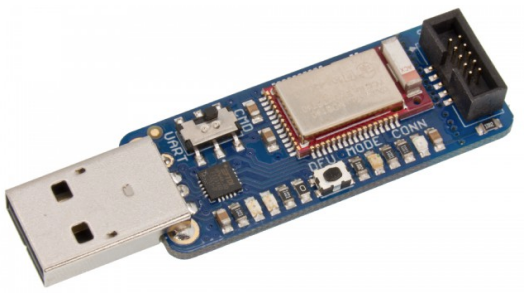


Figura 5.48 - Bluefruit LE Sniffer

Así, a partir de estos elementos se llevaron a cabo dos experimentos. En un primer experimento, se programó el dispositivo *Bluz DK* con el *firmware* de usuario *brainserialtest-bluzdk-v04.ino* con el parámetro RESP_BUFFER_SIZE_RAW = 58 (3+1 paquetes consecutivos) utilizándose como dispositivo *Central* la *app LightBlue*. En la Figura 5.49 se muestra un extracto de los paquetes de datos de usuario transferidos entre el dispositivo *Slave* (*Bluz DK*) y *Master* (*LightBlue*), así como el resto de paquetes transferidos periódicamente entre ambos dispositivos, bien como respuesta a la recepción de un paquete de una notificación, o bien al protocolo asociado a los intervalos de conexión (*Empty PDU*). En la información mostrada se incluye, además del *timestamp* asociado a cada evento (*TIME*), el campo *MOREDATA*, que con el valor TRUE indica que el paquete recibido es parte de un conjunto de paquetes consecutivos, observándose que, de los 4 paquetes consecutivos enviados desde el dispositivo *Bluz DK*, los 3 primeros tienen este atributo al valor TRUE, y el último (el paquete *tail* con el contenido 0x0304 en hexadecimal), a FALSE. Por otro lado, a partir del valor de los *timestamps* asociados a cada paquete de notificación, en cada caso, se deduce que en este caso se envían los 4 paquetes en un mismo intervalo de conexión desde el dispositivo *Bluz DK* a la *app LightBlue*.

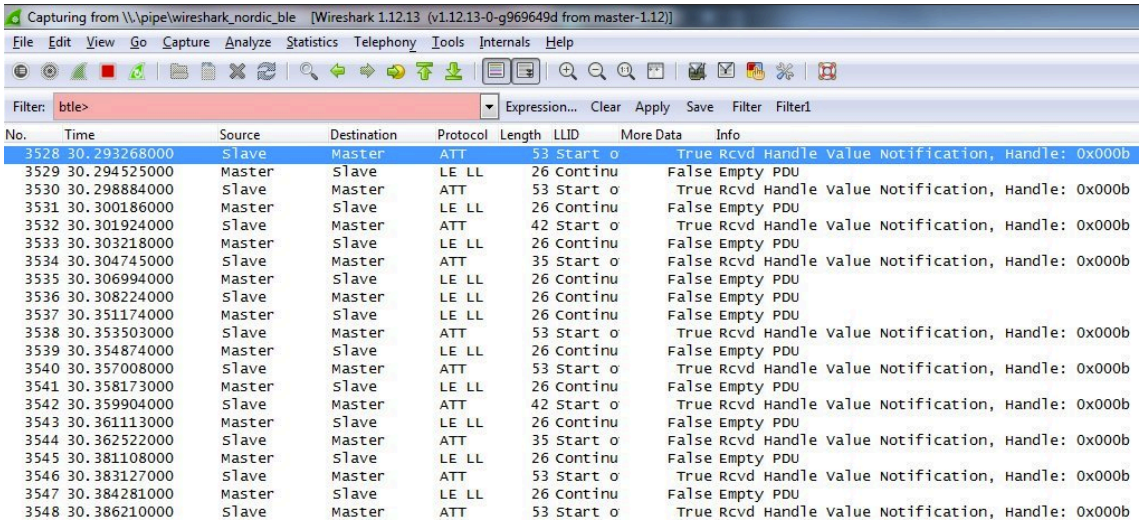


Figura 5.49 - Datos de usuario transferidos entre el dispositivo Slave (Bluz DK) y Master (LightBlue)

Con el fin de verificar de manera más rigurosa esta conclusión se generó un fichero *log* desde la aplicación *Wireshark* incluyendo toda la información relativa a los paquetes capturados en este experimento. Para determinar el número de paquetes transferidos por intervalo de conexión, además de la latencia, pueden analizarse otros factores, entre ellos los siguientes:

1.- El campo *EVENTCOUNTER* es el mismo para todos los paquetes en un mismo intervalo de conexión.

2.- En un mismo intervalo de conexión, los paquetes son transferidos a través de un mismo canal, por lo que un cambio de canal implica un cambio de intervalo de conexión.

Así, a partir del análisis de estos factores en los resultados obtenidos a partir de este primer experimento se confirma que con el dispositivo *Bluz DK* actuando como *Peripheral* y la *app Lightblue* actuando como dispositivo *Central*, en cada intervalo de conexión se transfieren los 4 paquetes asociados a cada llamada a la función *BLE.SendData()* en el dispositivo *Bluz DK*.

En un segundo experimento, mostrado en la Figura 5.50, se usó como dispositivo *Central* el dispositivo *RedBear Duo* ejecutando el *firmware* de usuario *simpleblecentral-duo-bluzdk-v07.ino,* manteniendo el resto de parámetros a los mismos valores que en

el experimento anterior. Sin embargo, en esta ocasión se observó, en primer lugar, que la correspondencia entre el valor del atributo *MOREDATA* con cada paquete de notificación resulta diferente a la del caso anterior, y en principio, errónea a no ser que se deba al hecho de que, como se puede comprobar en segundo lugar, en cada intervalo de conexión se transfiera un único paquete. Por otra parte, el campo *EVENTCOUNTER* en este caso es distinto para todos los paquetes en un mismo intervalo de conexión. Además, en este caso, los paquetes son transferidos a través de distintos canales.

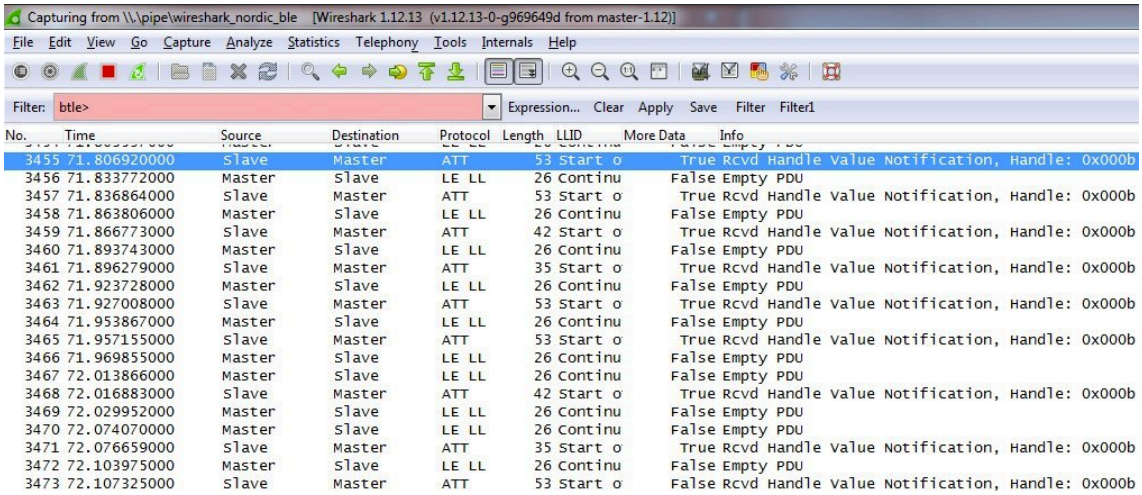


Figura 5.50- Datos de usuario transferidos entre el dispositivo Slave (Bluz DK) y Master (RedBear Duo)

Ante este hecho, sin aparente explicación, se decidió realizar una consulta en el foro de desarrolladores de *RedBear Duo*, indicando los resultados obtenidos a partir de la realización de ambos experimentos con el fin de obtener una explicación, y con ello, una posible solución a la limitación detectada en el dispositivo *RedBear Duo* actuando como *Central* para la transferencia de más de un paquete de datos de usuario por intervalo de conexión.

En este punto, tras las consultas realizadas se constató una limitación, reconocida por el desarrollador de la librería *BTStack* en la que se basa el *firmware* del dispositivo *RedBear Duo,* Matthias Ringwald, y mostrada en la Figura 5.51, relativa al dispositivo *Broadcom* BCM43438 integrado en el dispositivo *RedBear Duo*.

Figura 5.51 - Reconocimiento de la limitación detectada por parte del desarrollador de la librería BTStack

La detección de esta limitación dio lugar a una consulta a *Cypress* indicando el problema detectado en este TFM, mostrado en la Figura 5.52, y que aún se encuentra a la espera de respuesta.

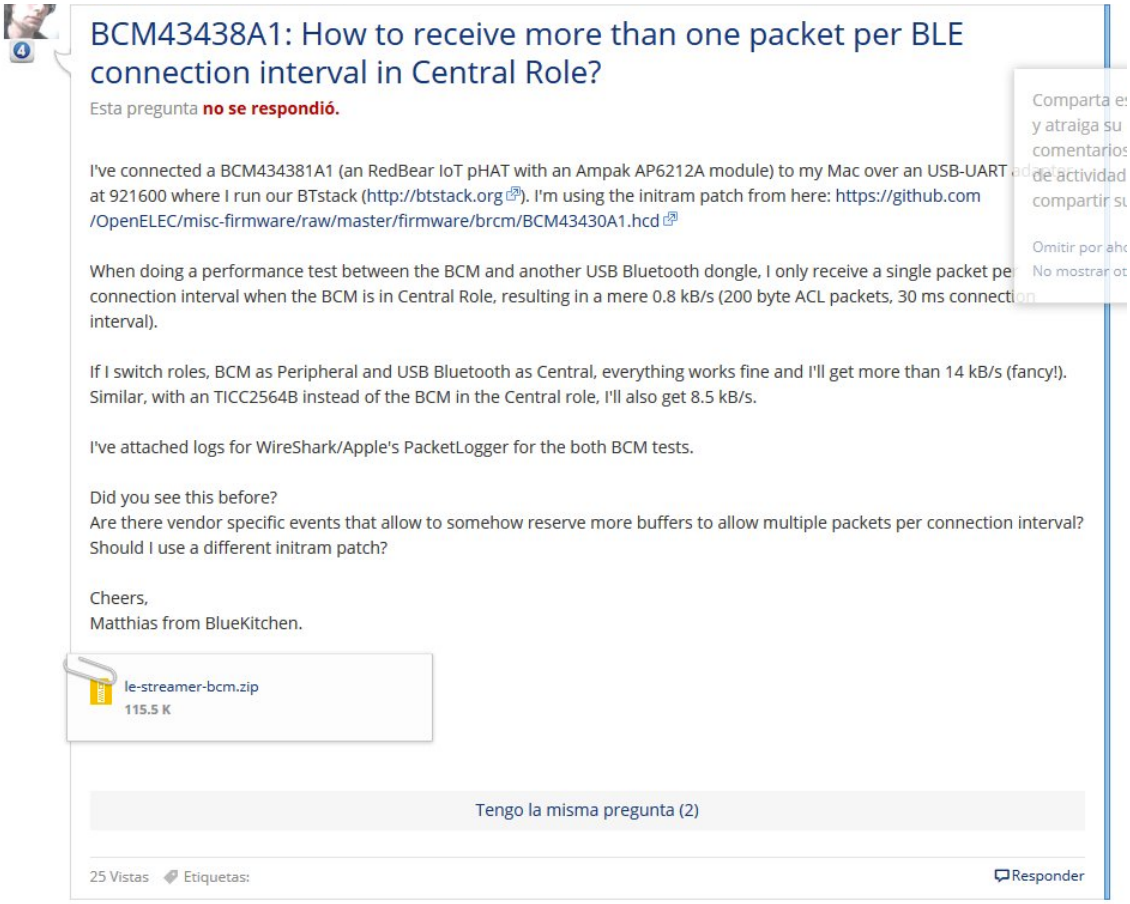


Figura 5.52 - Consulta a Cypress del problema detectado

Capítulo 6. Procesamiento de la información EEG para la detección del parpadeo de ojos en el dispositivo *RedBear Duo*

En este capítulo se expone el algoritmo propuesto para la detección del parpadeo de los ojos mediante las señales de EEG registrados por el módulo TGAM haciendo uso de la plataforma HW/SW realizada.

6.1 Características de la señal EEG asociada al parpadeo de ojos

En este punto del desarrollo del presente TFM, se pretende detectar el parpadeo de ojos en el dispositivo *RedBear Duo*, a partir de la información de EEG en modo RAW recibida desde el módulo TGAM integrado en el dispositivo *Mindflex* a través de la conexión BLE establecida con el dispositivo *Bluz DK*. Para completar esta tarea, se tomará como referencia las características de la señal EEG asociada al parpadeo de ojos, mostrada en la Figura 6.1 **[28]**.

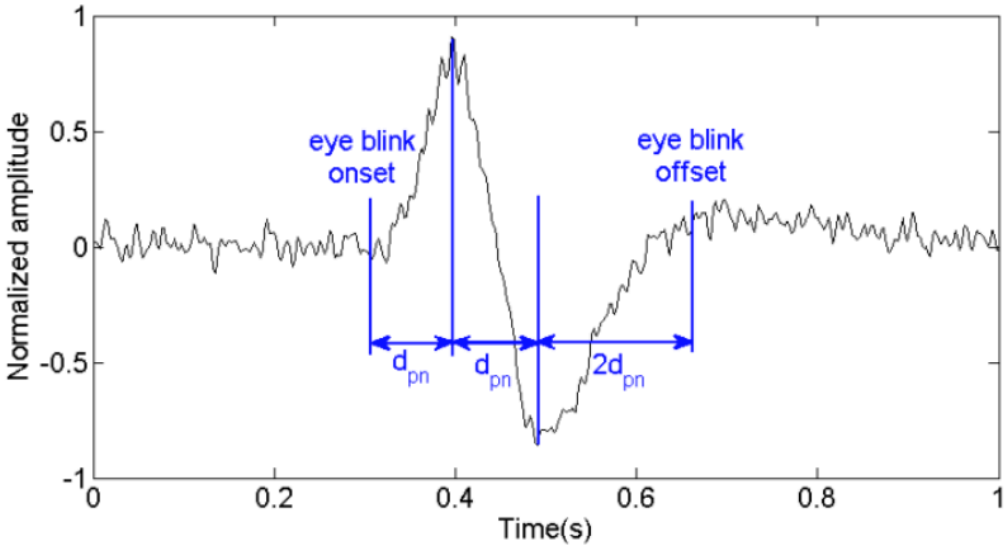


Figura 6.1 – Señal EEG asociada al parpadeo de ojos

Así, en lo que respecta a la duración de la señal observada en el EEG cuando se realiza un parpadeo de ojos, en diversas fuentes se recoge que su duración está comprendida en términos generales, entre 200 milisegundos y 400 milisegundos, presentando inicialmente un pico positivo en su amplitud, seguido de un pico negativo de menor amplitud.

Por otro lado, precisamente en relación con la interpretación de los valores asociados a las muestras proporcionadas por el dispositivo *Mindflex* en modo RAW, como se ha mencionado con anterioridad en este TFM se conoce que, a pesar de que son de 16 bits con signo (que comprenderían un rango de valores en el intervalo -32768 a +32767), en la práctica proporciona valores de muestra que para el caso del módulo *ThinkGear ASIC*, están comprendidos únicamente en el intervalo (-2048/2047), lo que representa un rango de 4096 valores **[12].**

Por otro lado, en la información de soporte proporcionada por *Neurosky* se indica que para el caso del módulo TGAM, el valor en voltios correspondiente a cada valor de las muestras proporcionadas por el dispositivo *Mindflex* en modo RAW puede obtenerse a través de la siguiente expresión:

$$volts = rawValue * \left(\frac{1{,}8}{4096}\right) / 2000$$

Así, para la detección del parpadeo de ojos **[29]**, se deduce que en caso de recibir el RAW EXCODE correspondiente al valor 0x80, de acuerdo a las especificaciones del dispositivo *ThinkGear TGAM ASIC*, se debe realizar en primer lugar una conversión del valor de cada muestra recibida (de 16 bits) a su correspondiente valor negativo, en caso necesario. A continuación, se espera hasta detectar un valor de muestra superior al umbral de 200 para considerar la posible detección de un parpadeo de ojos. De darse esta circunstancia se inicia un *timer*, denominado *blinkTimer*, a partir del valor inicial 0. A partir de este primer evento, en caso de detectarse en un periodo de tiempo comprendido entre 10 milisegundos y 350 milisegundos, un valor de muestra inferior a un umbral de -90, se considera que ha detectado un parpadeo de ojos, deteniéndose e inicializándose a 0 el *timer*. Finalmente, en caso de no detectar este segundo evento antes de un intervalo de tiempo de 500 milisegundos, después de haberse detectado el primero evento, se considera que éste se corresponde con un falso parpadeo de ojos.

Esta funcionalidad se incluirá en la plataforma desarrollada en este TFM en el dispositivo *RedBear Duo*, para lo cual será necesario modificar el fichero *brainserialtest-bluzdk-v05.ino*, que incluye además los ficheros *Parser.cpp* y *Parser.h*.

Sin embargo, antes de integrar la detección del parpadeo de ojos en el dispositivo *RedBear Duo* a partir del valor de las muestras en modo RAW recibidas desde el dispositivo *Bluz DK* a través de la conexión BLE, se decidió validar su funcionamiento inicial en el dispositivo *RedBear Duo*, conectándolo de forma cableada a través del puerto serie al módulo TGAM integrado en el dispositivo. Para ello se creó el fichero

brainserialtest-duo-eyeblink-v01.ino y se modificó parcialmente el código del fichero *Parser.cpp.* En primer lugar, la modificación realizada sobre el fichero *Parser.cpp* afectó a las líneas 148 y 149, comentando el código en el que se realizaba la conversión a valores negativos de las muestras obtenidas en modo RAW desde el módulo TGAM, definiendo la variable *rawValue* como *int*.

```
145             case 0x80:
146                 i++;
147                 rawValue = ((int)packetData[++i] << 8) | packetData[++i];
148               //  if (rawValue >= 32768)
149               //      rawValue = rawValue - 65535;
150                 hasRaw = true;
151                 break;
```

Figura 6.2 - Modificación fichero Parser.cpp

Por otro lado, el código *brainserialtest-duo-eyeblink-v01.ino*, mostrado en la Figura 6.3, permite procesar los valores de las muestras proporcionadas por el módulo TGAM en modo RAW a través del puerto serie *Serial1*, y detectar el parpadeo de los ojos.

```
1   #include "Parser.h"
2   #include "application.h"
3
4   boolean modeRaw = true;
5   boolean modeDebug = false;
6
7   uint8_t *buffer;
8   int16_t rawValueDEC;
9
10  unsigned long time_event1;
11  boolean event1 = false;
12
13
14  Parser parser(Serial1, modeRaw, modeDebug);
15
16  void setup() {
17      pinMode(D7, OUTPUT);
18      digitalWrite(D7, LOW);
19      Serial.begin(115200);
20
21      if (modeRaw) {
22          Serial1.begin(57600);
23      }
24      else {
25          Serial1.begin(9600);
26      }
27
28      Serial.println("welcome to parserserialtest-photon-eyeblink-v01");
29      Serial.println("-----------------------------------------------");
30  }
31
32
33  void loop() {
34      if (parser.update())
35      {
36          buffer = parser.readCSV();
37          if (modeRaw) {
38              rawValueDEC = ((int)buffer[1] << 8) | buffer[0];
39              Serial.print("  rawvalueDEC(DEC) = ");
40              Serial.println(rawValueDEC, DEC);
41
42
```

```
            if ((rawValueDEC > 200) && (!event1)) {
                time_event1 = millis();
                event1 = true;
                Serial.print("*** event1 enabled *** rawValueDEC = ");
                Serial.print(rawValueDEC, DEC);
                Serial.print(" time = ");
                Serial.print(time_event1);
               Serial.println("-----------------------------------------------");
            }
            else if ((rawValueDEC < -90) && (event1) && ((millis()-time_event1) > 10)
            && ((millis()-time_event1) < 350)) {
                event1 = false;
                Serial.print("***--- EYE BLINK DETECTED ---*** rawValueDEC = ");
                Serial.print(rawValueDEC, DEC);
                Serial.print(" lapsed time = ");
                Serial.print((millis()-time_event1));
                Serial.println("-----------------------------------------------");
                blinked();
            }
            else if ((event1) && ((millis()-time_event1) > 500)) {
                event1 = false;
                Serial.print("*** event1 disabled *** time = ");
                Serial.print((millis()-time_event1));
                Serial.println("-----------------------------------------------");
            }
        }
    }
}

void blinked(void) {
```

Figura 6.3 - Fichero brainserialtest-duo-eyeblink-v01.ino

Sin embargo, al realizar las primeras pruebas conectando directamente al dispositivo *Mindflex* al dispositivo *RedBear Duo* a través de la interfaz serie *Serial1*, y ejecutando en éste el *firmware* de usuario *brainserialtest-duo-eyeblink-v01.ino*, se observó cómo, aunque no se realizase ningún parpadeo, se detectaban multitud de falsos parpadeos de ojos, estando el LED *D7* del dispositivo *RedBear Duo* continuamente luciendo, además de obtenerse valores de muestras en modos RAW desde el dispositivo *Mindflex* con valores que presentaban diferencias significativas en intervalos de tiempo muy reducidos. Este efecto, totalmente indeseado, se relacionó con el posible ruido interferente que podía estar enmascarando el valor de las muestras, por lo que se decidió incorporar entre el dispositivo *Mindflex* y el dispositivo *RedBear Duo,* en este escenario, un optoacoplador en la conexión serie cableada entre ambos. En concreto, el optoacoplador seleccionado fue el dispositivo 6N138 de la compañía VISHAY **[30]**, mostrado en la Figura 6.4. Este optoacoplador, además de ser compatible con los niveles TTL asociados a la señal transmitida entre el módulo *TGAM* y el dispositivo *RedBear Duo* en modo RAW, soporta una tasa de transferencia de 57600 baudios.

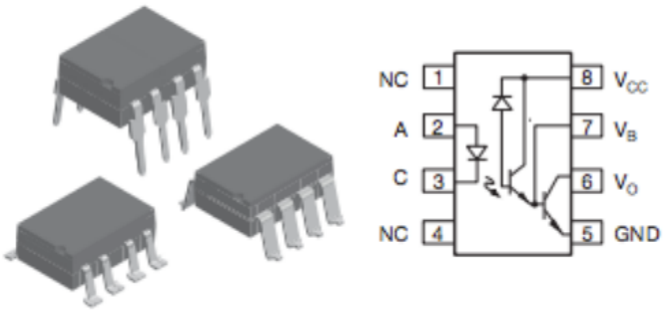


Figura 6.4 - Optoacoplador 6N138

Con el fin de determinar el correcto conexionado del optoacoplador 6N138 se analizaron principalmente las hojas de características proporcionadas por el fabricante, quedando el esquema de conexión realizado inicialmente como se muestra en la Figura 6.5.

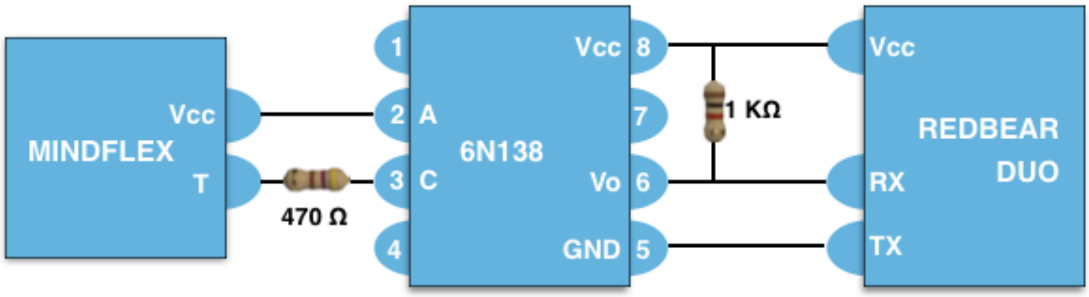


Figura 6.5 - Conexiones del optoacoplador 6N138

A partir de esta conexión se pudo demostrar finalmente el correcto funcionamiento del código *brainserialtest-duo-eyeblink-v01.ino* desarrollado para detectar el parpadeo de los ojos a partir del valor de las muestras proporcionadas por el módulo TGAM en modo RAW, una vez eliminado el ruido interferente en la conexión serie entre éste y el dispositivo *RedBear Duo* mediante el optoacoplador 6N138, como se puede comprobar en el vídeo *EyeBlink1.mov* grabado y disponible en el CD-ROM adjunto a este documento.

Así, el siguiente paso consistiría en reproducir este funcionamiento en el dispositivo *RedBear Duo*, pero a partir de las muestras transferidas en paquetes BLE desde el dispositivo *Mindflex* conectado directamente a través de la interfaz serie al dispositivo *Bluz DK* con el fin de emular un dispositivo capaz de registrar la información EEG y

transmitirla mediante una conexión BLE, como se muestra esquemáticamente en la Figura 6.6, que representa el escenario objetivo de este TFM.

Figura 6.6 - Escenario objetivo del TFM

Sin embargo, la adaptación del código desarrollado en el fichero *brainserialtest-duo-eyeblink-v01.ino* a este escenario requiere, en primer lugar, adaptar el código desarrollado en *simpleblecentral-duo-bluzdk-v08.ino* con el fin de añadir la correcta recepción e interpretación de los datos recibidos en modo RAW desde el módulo TGAM a través de la trasferencia de paquetes BLE desde el dispositivo *Bluz DK*.

Con este objetivo se desarrolló, por un lado, el código *serialtest-bluzdk-v01.ino* para ser ejecutado en el dispositivo *Bluz DK* con el fin de emular el envío de las muestras proporcionadas por el módulo TGAM a través de la conexión serie con el dispositivo *Bluz DK* en modo RAW.

```
#include "application.h"

boolean modeRaw = true;
boolean modeDebug = false;

unsigned long reftime = 0;
unsigned long acttime;
#define RESP_BUFFER_SIZE_RAW 58

#define BUFFER_SIZE_NORMAL 27

uint8_t resp_buffer[RESP_BUFFER_SIZE_RAW];
int resp_buffer_idx = 0;
int16_t buffer = 0;

SYSTEM_MODE(MANUAL);

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

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

```
    if (modeRaw) {
        Serial1.begin(57600);
        resp_buffer_idx = 0;
    }
    else {
        Serial1.begin(9600);
    }
}


void loop() {
    System.sleep(SLEEP_MODE_CPU);
    if (BLE.getState() == BLE_CONNECTED) {
        if (millis()-reftime >= 2) {
           ++buffer;
           if (modeRaw) {
               resp_buffer[resp_buffer_idx++] = buffer & 0xff;
               resp_buffer[resp_buffer_idx++] = buffer >> 8;
               if (resp_buffer_idx > (RESP_BUFFER_SIZE_RAW-1)) {
                   BLE.sendData(resp_buffer, RESP_BUFFER_SIZE_RAW);
                   resp_buffer_idx = 0;
                   digitalWrite(D7, HIGH);
               }
               else {
                  digitalWrite(D7, LOW);
               }
               reftime = millis();
            }
        }
        else {
         digitalWrite(D7, LOW);
        }
    }
    else {
         digitalWrite(D7, LOW);
        }
}
```

Figura 6.7 - Fichero serialtest-bluzdk-v01.ino

Por otro lado, se desarrolló el código *brainserialtest-duo-eyeblink-v09.ino*, en el que se implementa la recepción e interpretación de los valores recibidos desde el dispositivo *Bluz DK*, en el dispositivo *RedBear Duo* en modo RAW, como se muestra en la sección de código representada en la Figura 6.8.

```
boolean modeRaw = true;
uint8_t tail = 0;
uint8_t Signalquality;
uint8_t Attention;
uint8_t Meditation;
uint32_t EEG0, EEG1, EEG2, EEG3, EEG4, EEG5, EEG6, EEG7;
int16_t Rawsample[29];
  if (modeRaw) {
     if ((value[0] == 0x04) &&  (tail == 0)) {
        tail++;
        Rawsample[0] = (value[1] + (value[2] << 8));
        Rawsample[1] = (value[3] + (value[4] << 8));
        Rawsample[2] = (value[5] + (value[6] << 8));
        Rawsample[3] = (value[7] + (value[8] << 8));
        Rawsample[4] = (value[9] + (value[10] << 8));
        Rawsample[5] = (value[11] + (value[12] << 8));
        Rawsample[6] = (value[13] + (value[14] << 8));
        Rawsample[7] = (value[15] + (value[16] << 8));
        Rawsample[8] = (value[17] + (value[18] << 8));
        Rawsample[9] = (value[19]);

```

```
397           Serial.print("   - Sample values (DEC): ");
398           Serial.print(Rawsample[0], DEC);
399           Serial.print(", ");
400           Serial.print(Rawsample[1], DEC);
401           Serial.print(", ");
402           Serial.print(Rawsample[2], DEC);
403           Serial.print(", ");
404           Serial.print(Rawsample[3], DEC);
405           Serial.print(", ");
406           Serial.print(Rawsample[4], DEC);
407           Serial.print(", ");
408           Serial.print(Rawsample[5], DEC);
409           Serial.print(", ");
410           Serial.print(Rawsample[6], DEC);
411           Serial.print(", ");
412           Serial.print(Rawsample[7], DEC);
413           Serial.print(", ");
414           Serial.print(Rawsample[8], DEC);
415           Serial.println();
416       }
417       else if (tail == 1) {
418           tail++;
419           Rawsample[9] = (Rawsample[9] + (value[0] << 8));
420           Rawsample[10] = (value[1] + (value[2] << 8));
421           Rawsample[11] = (value[3] + (value[4] << 8));
422           Rawsample[12] = (value[5] + (value[6] << 8));
423           Rawsample[13] = (value[7] + (value[8] << 8));
424           Rawsample[14] = (value[9] + (value[10] << 8));
425           Rawsample[15] = (value[11] + (value[12] << 8));
426           Rawsample[16] = (value[13] + (value[14] << 8));
427           Rawsample[17] = (value[15] + (value[16] << 8));
428           Rawsample[18] = (value[17] + (value[18] << 8));
429           Rawsample[19] = (value[19]);
430
431           Serial.print("   - Sample values (DEC): ");
432           Serial.print(Rawsample[9], DEC);
433           Serial.print(", ");
434           Serial.print(Rawsample[10], DEC);
435           Serial.print(", ");
436           Serial.print(Rawsample[11], DEC);
437           Serial.print(", ");
438           Serial.print(Rawsample[12], DEC);
439           Serial.print(", ");
440           Serial.print(Rawsample[13], DEC);
441           Serial.print(", ");
442           Serial.print(Rawsample[14], DEC);
443           Serial.print(", ");
444           Serial.print(Rawsample[15], DEC);
445           Serial.print(", ");
446           Serial.print(Rawsample[16], DEC);
447           Serial.print(", ");
448           Serial.print(Rawsample[17], DEC);
449           Serial.print(", ");
450           Serial.print(Rawsample[18], DEC);
451           Serial.println();
452       }
453       else if (tail == 2) {
454           tail++;
455           Rawsample[19] = (Rawsample[19] + (value[0] << 8));
456           Rawsample[20] = (value[1] + (value[2] << 8));
457           Rawsample[21] = (value[3] + (value[4] << 8));
458           Rawsample[22] = (value[5] + (value[6] << 8));
459           Rawsample[23] = (value[7] + (value[8] << 8));
460           Rawsample[24] = (value[9] + (value[10] << 8));
461           Rawsample[25] = (value[11] + (value[12] << 8));
462           Rawsample[26] = (value[13] + (value[14] << 8));
463           Rawsample[27] = (value[15] + (value[16] << 8));
464           Rawsample[28] = (value[17] + (value[18] << 8));
465
466           Serial.print("   - Sample values (DEC): ");
467           Serial.print(Rawsample[19], DEC);
468           Serial.print(", ");
469           Serial.print(Rawsample[20], DEC);
470           Serial.print(", ");
471           Serial.print(Rawsample[21], DEC);
472           Serial.print(", ");
```

```
            Serial.print(Rawsample[22], DEC);
            Serial.print(", ");
            Serial.print(Rawsample[23], DEC);
            Serial.print(", ");
            Serial.print(Rawsample[24], DEC);
            Serial.print(", ");
            Serial.print(Rawsample[25], DEC);
            Serial.print(", ");
            Serial.print(Rawsample[26], DEC);
            Serial.print(", ");
            Serial.print(Rawsample[27], DEC);
            Serial.print(", ");
            Serial.print(Rawsample[28], DEC);
            Serial.println();
        }
        else if (tail == 3) {
            tail = 0;
            if (value[0] == 0x03 && value[1] == 0x04) {
                Serial.print("Successfully received 29 16-bits sample in RAW mode -Time(ms)=");
                Serial.println(millis());
            }
        }
    }
    else {
        if ((value[0] == 0x04) &&  (tail == 0)) {
            tail++;
            Signalquality = value[1];
            Serial.print("      > Signalquality = ");
            Serial.println(Signalquality);
            Attention = value[2];
            Serial.print("      > Attention = ");
            Serial.println(Attention);
            Meditation = value[3];
            Serial.print("      > Meditation = ");
            Serial.println(Meditation);

            EEG0 = (value[4] + (value[5] << 8) + (value[6] << 16));
            Serial.print("      > EEG0 = ");
            Serial.println(EEG0);

            EEG1 = (value[7] + (value[8] << 8) + (value[9] << 16));
            Serial.print("      > EEG1 = ");
            Serial.println(EEG1);

            EEG2 = (value[10] + (value[11] << 8) + (value[12] << 16));
            Serial.print("      > EEG2 = ");
            Serial.println(EEG2);

            EEG3 = (value[13] + (value[14] << 8) + (value[15] << 16));
            Serial.print("      > EEG3 = ");
            Serial.println(EEG3);

            EEG4 = (value[16] + (value[17] << 8) + (value[18] << 16));
            Serial.print("      > EEG4 = ");
            Serial.println(EEG4);

            EEG5 = (value[19]);
        }
        else if (tail == 1) {
            tail++;
            EEG5 = (EEG5 + (value[0] << 8) + (value[1] << 16));
            Serial.print("      > EEG5 = ");
            Serial.println(EEG5);

            EEG6 = (value[2] + (value[3] << 8) + (value[4] << 16));
            Serial.print("      > EEG6 = ");
            Serial.println(EEG6);

            EEG7 = (value[5] + (value[6] << 8) + (value[7] << 16));
            Serial.print("      > EEG7 = ");
            Serial.println(EEG7);
        }
```

```
545       else if (tail == 2) {
546          tail = 0;
547          if (value[0] == 0x03 && value[1] == 0x04) {
548             Serial.println("Successfully received new sample data in NORMAL mode");
549          }
550          else {
551             Serial.println("ERROR receiving new sample data from Mindflex in NORMAL mode");
552          }
553       Serial.println();
554       }
555    }
556 }
```

Figura 6.8 - Modificaciones para la creación del fichero brainserialtest-duo-eyeblink-v09.ino

Se debe tener en cuenta que el código *brainserialtest-duo-eyeblink-v09.ino* se ha realizado de manera específica para el caso en el que el valor del parámetro RESP_BUFFER_SIZE_RAW sea 58, enviándose desde el dispositivo *Bluz DK* al dispositivo *RedBear Duo* tres paquetes consecutivos de aproximadamente 20 bytes con el valor de 29 muestras RAW en total (más 1 paquete de cola con el contenido 0x0304 en hexadecimal) en cada llamada a la función *BLE.SendData()* en el dispositivo *Bluz DK*.

A partir de la ejecución de estos códigos en los dispositivos *Bluz DK* y *RedBear Duo*, en cada caso, se pudo comprobar la correcta recepción de todos los valores correspondientes a las muestras RAW enviadas desde el dispositivo *Bluz DK* al dispositivo *RedBear Duo*, como se deduce de la información obtenida por pantalla en la consola del terminal, mostrada en la Figura 6.9.

```
packetData1(BIN) = 0  packetData0(BIN) = 1101101
  rawvalueDEC(DEC) = 109
packetData1(BIN) = 0  packetData0(BIN) = 1110001
  rawvalueDEC(DEC) = 113
packetData1(BIN) = 0  packetData0(BIN) = 10000001
  rawvalueDEC(DEC) = 129
packetData1(BIN) = 0  packetData0(BIN) = 10000101
  rawvalueDEC(DEC) = 133
packetData1(BIN) = 0  packetData0(BIN) = 10001000
  rawvalueDEC(DEC) = 136
packetData1(BIN) = 0  packetData0(BIN) = 10010010
  rawvalueDEC(DEC) = 146
packetData1(BIN) = 0  packetData0(BIN) = 10010100
  rawvalueDEC(DEC) = 148
packetData1(BIN) = 0  packetData0(BIN) = 10010011
  rawvalueDEC(DEC) = 147
packetData1(BIN) = 0  packetData0(BIN) = 10011000
  rawvalueDEC(DEC) = 152
```

Figura 6.9 - Valores de las muestras en modo RAW

Así el siguiente paso sería comprobar el funcionamiento del código *brainserialtest-duo-eyeblink-v09.ino*, recibiendo muestras reales enviadas desde el dispositivo *Bluz DK* a

partir de los valores proporcionados por el dispositivo *Mindflex* en modo RAW. Para ello se utilizó el código *brainserialtest-bluzkdk-v05.ino*, desarrollado con anterioridad, junto con los ficheros *Parser.cpp* y *Parser.h*, para ser programados en el dispositivo *Bluz DK* conectado vía serie, a través del optoacoplador 6N138, al módulo TGAM programado en modo RAW.

6.2 Adaptación de la conexión entre el módulo TGAM y el dispositivo *Bluz DK* mediante el uso de un optoacoplador.

A la hora de adaptar el montaje del módulo TGAM empleando el optoacoplador, a la integración del dispositivo *Bluz DK*, se consideró inicialmente la posibilidad de mantener el montaje mostrado en la Figura 6.11, si bien en el caso del dispositivo *Bluz DK*, a diferencia del dispositivo *RedBear Duo*, el pin *VIN* no puede actuar como salida, con el fin de obtener una tensión de aproximadamente 5V para conectarlo al pin 8 del octoacoplador 6N138 (lo que si era posible en el dispositivo *RedBear Duo* alimentado mediante USB). Por este motivo se decidió usar el dispositivo *BooSTick*, mostrado en la Figura 6.10, que es capaz de proporcionar una tensión de 3.3V/5V a partir de una pila AA de 1.5 V.

Figura 6.10 - Dispositivo BooSTick

A partir de este dispositivo, se decidió inicialmente realizar el montaje mostrado en la Figura 6.11.

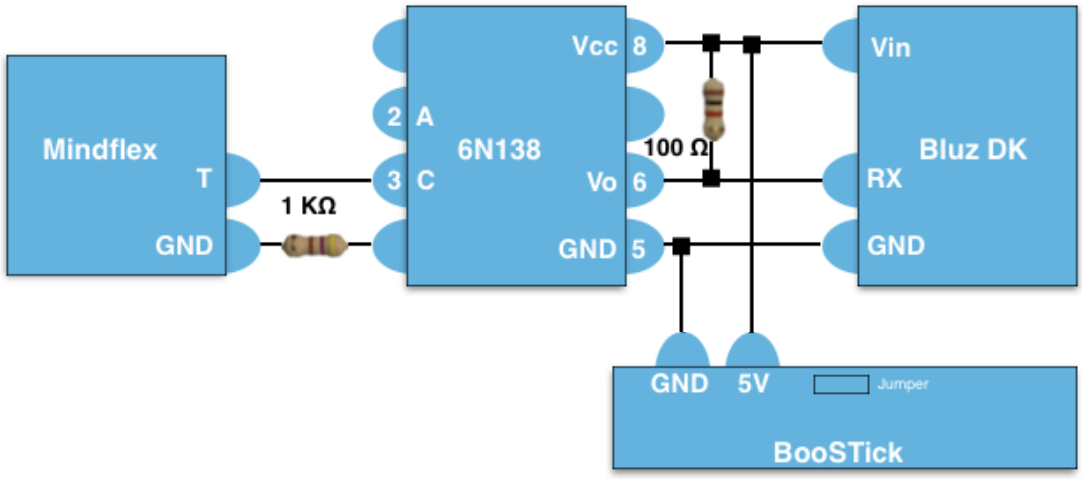


Figura 6.11 - Montaje inicial Minflex-Optoacoplador-BluzDK

Sin embargo, a partir de este montaje, no se podían recibir correctamente los datos a la salida del optoacoplador 6N138 en el dispositivo *Bluz DK*, recibiéndose, en lugar de los valores correspondientes a los bits de los valores de las muestras capturadas por el módulo TGAM en modo RAW, una señal continuamente a nivel bajo. El motivo de este hecho residía en la tensión aplicada en el terminal 2 (A) del optoacoplador, con respecto al valor correspondiente al nivel alto en la señal aplicada al termina 3 (C). Así la tensión aplicada en el terminal A, correspondiente al cable *VCC* del dispositivo *Mindflex* era aproximadamente de 4.5V (con las pilas nuevas) mientras que la tensión correspondiente al nivel alto en la señal aplicada al terminal 3 (TLL *Serial*) era de aproximadamente 3.3 V, lo que hacía que, como se muestra en la Figura 6.12, el fotodiodo emisor estuviera continuamente en estado ON, y debido al carácter inversor del montaje realizado, la salida del optoacoplador estuviera siempre a nivel bajo.

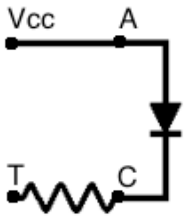


Figura 6.12 - Terminales del dispositivo optoacoplador

Por ello, se concluyó que este montaje era erróneo y que con el dispositivo *RedBear Duo* había funcionado, seguramente por el hecho de que circunstancialmente las pilas del dispositivo *Mindflex* estuvieran gastadas, y en consecuencia, el valor de la tensión aplicada en el terminal 2 del optoacoplador 6N138 fuese inferior a los 4.5V nominales

(se comprobó que en la práctica era próximo a 3.3 V). Con todo, se estudió el correcto montaje del dispositivo *Mindflex* junto con el optoacoplador y el dispositivo *Bluz Dk*, realizándose finalmente el montaje mostrado en la Figura 6.13, que llevado a la *protoboard* quedó de la manera mostrada en la Figura 6.14.

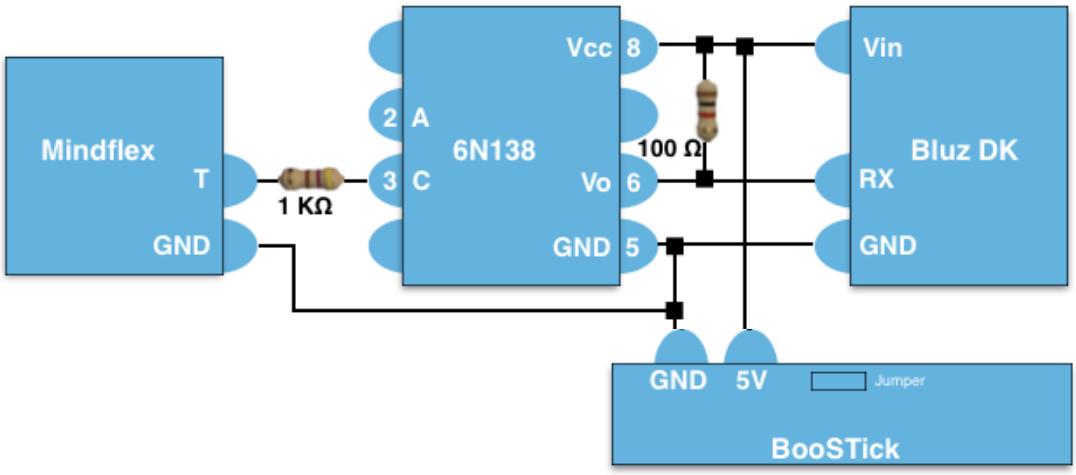


Figura 6.13 - Montaje final Minflex-Optoacoplador-BluzDK

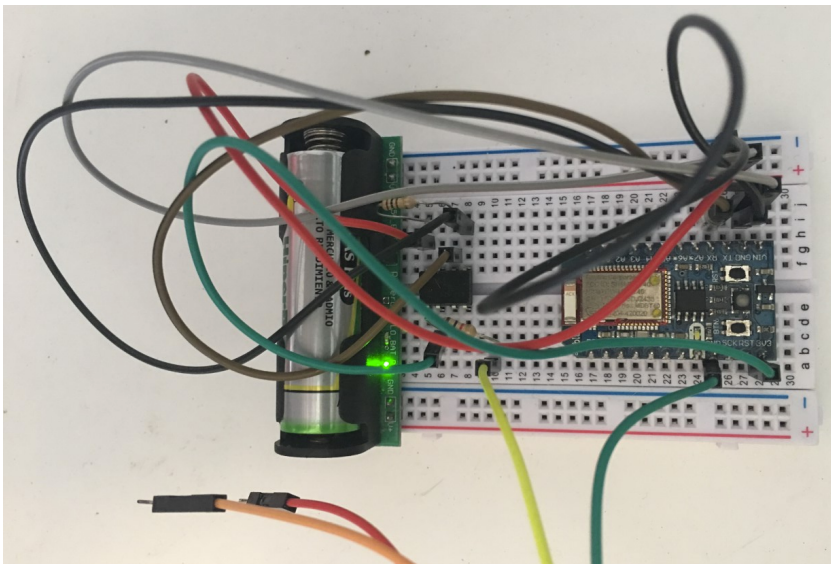


Figura 6.14 - Montaje implementado

A partir de este montaje se pudo comprobar la correcta recepción de los valores de las muestras EEG por parte del dispositivo *Bluz DK*, y a partir de la programación del código *serialtest-bluzdk-v05.ino* en el dispositivo *Bluz DK*, y del código *brainserialtest-duo-eyeblink-v09.ino* en el dispositivo *RedBear Duo*, su correcta transmisión, vía BLE a este último.

En segundo lugar, con el fin de adaptar al código desarrollado en *brainserialtest-duo-eyeblink-v01.ino* al escenario en el que se desea detectar el parpadeo de ojos en el dispositivo *RedBear Duo* a partir de los valores de las muestras del módulo TGAM en modo RAW, recibidos desde el dispositivo *Bluz DK* en paquetes BLE, en concreto en el caso específico en el que el parámetro RESP_BUFFER_SIZE_RAW sea 58 (3 paquetes de 20 bytes más un paquete *tail*, conteniendo un total de 29 valores correspondientes a muestras en modo RAW), se desarrolló el código *brainserialtest-duo-eyeblink-v021.ino* en el que el valor de las muestras RAW obtenidas desde el módulo TGAM a través del puerto serie, son almacenados temporalmente en el *buffer RawSample[],* de tamaño 29, de forma que cuando se dispone de 29 muestras (equivalente a la recepción de las muestras en el dispositivo *RedBear Duo* desde el dispositivo *Bluz DK*) se pasa a analizar sus valores con el fin de detectar el parpadeo de ojos.

En este sentido, la detección de las condiciones en las que se identifica un parpadeo de ojos en este escenario, no puede seguir basándose en la referencia temporal usada en el código, *brainserialtest-duo-v01.ino,* sino que se realizará a partir del número de muestras recibido, teniendo en cuenta que, de acuerdo al funcionamiento del módulo TGAM en modo RAW, se genera un nuevo valor de muestra aproximadamente cada 2 milisegundos (exactamente 0.001953125 segundos), considerándose esta referencia temporal, correspondiente a 10 milisegundos, equivalente a la recepción de 5 muestras (exactamente 5.12 muestras), mientras que la referencia temporal correspondiente a 350 milisegundos se considerará equivalente a la recepción de 179 muestras (el valor exacto sería 179.2 muestras), y la correspondiente a la referencia de 500 milisegundos, se considerará equivalente a la recepción de 250 muestras (el valor exacto sería 256 muestras), obteniéndose así el código de prueba *brainserialtest-duo-eyeblink-v021.ino*, mostrado en la Figura 6.15. Este código fue verificado correctamente, detectándose, al igual que en el caso de los códigos anteriores, aunque con cierto retardo, las situaciones de parpadeo de los ojos, como se muestra en el video *EyeBlink2.mov* grabado y disponible en el CD-ROM adjunto a este documento.

```
#include "Brain.h"
#include "application.h"

boolean modeRaw = true;
boolean modeDebug = false;

uint8_t *buffer;
int16_t rawValueDEC;
int16_t Rawsample[29];
int sampleindex = 0;

uint16_t counter_event1 = 0;
boolean event1 = false;

Brain brain(Serial1, modeRaw, modeDebug);

void setup() {
    pinMode(D7, OUTPUT);
    digitalWrite(D7, LOW);
    Serial.begin(115200);
    if (modeRaw) {
        Serial1.begin(57600);
    }
    else {
        Serial1.begin(9600);
    }
    Serial.println("welcome to brainserialtest-photon-eyeblink-v021");
    Serial.println("-----------------------------------------------");
}

void loop() {
    if (brain.update())
    {
        buffer = brain.readCSV();
        if (modeRaw) {
            rawValueDEC = ((int)buffer[1] << 8) | buffer[0];
            Serial.print("  rawvalueDEC(DEC) = ");
            Serial.println(rawValueDEC, DEC);
            Rawsample[sampleindex] = rawValueDEC;
            if (sampleindex == 28) {
               // process samples to detect eye blink
               sampleindex = 0;
               for(int i=0;i<29;i++) {
                  counter_event1++;
                  if ((Rawsample[i] > 200) && (!event1)) {
                     counter_event1 = 0;
                     event1 = true;
                     Serial.print("*** event1 enabled *** rawValueDEC = ");
                     Serial.print(Rawsample[i], DEC);
                     Serial.print(" counter_event1 = ");
                     Serial.print(counter_event1);
                     Serial.println("-----------------------------------------------");
                  }
                  else if ((Rawsample[i] < -90) && (event1) && (counter_event1 > 5) &&
                  (counter_event1 < 175)) {
                     event1 = false;
                     Serial.print("***--- EYE BLINK DETECTED ---*** rawValueDEC = ");
                     Serial.print(Rawsample[i], DEC);
                     Serial.print(" lapsed counter_event1 = ");
                     Serial.print(counter_event1);
                     Serial.println("-----------------------------------------------");
                     blinked();
                  }
                  else if ((event1) && (counter_event1 > 250)) {
                     event1 = false;
                     Serial.print("*** event1 disabled *** counter_event1 = ");
                     Serial.print(counter_event1);
                     Serial.println("-----------------------------------------------");
                  }
               }
            }
```

```
            else {
              sampleindex++;
            }
        }
    }
}

void blinked(void) {
  digitalWrite(D7, HIGH);
  delay(1);
  digitalWrite(D7, LOW);
}
```

Figura 6.15 - Fichero brainserialtest-duo-eyeblink-v021.ino

Para finalizar, en la Figura 6.16 se muestran los mensajes obtenidos por pantalla ante la correcta detección de un parpadeo de ojos empleando los ficheros finales descritos, y empleando el optoacoplador entre el módulo TGAM y el dispositivo *Bluz DK*.

```
* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 0 D6 0 D1 0 D1 0 C9 0 AD 0 8B 0 85 0 A5 0 C4 0

* Received new notification (28 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4
* Successfully received 29 16-bits sample values from Mindflex in RAW mode - Time(ms) = 57490
   - Sample values (DEC): *** event1 enabled *** rawValueDEC = 209 counter_event1 =
0--------------------------------------------------

* Received new notification (62 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 FD FF E6 FF C3 FF AC FF A5 FF 8F FF 78 FF 7D FF 89 FF 7F

* Received new notification (390 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: FF 85 FF AF FF E9 FF 3 0 F7 FF CA FF A6 FF A7 FF C4 FF CF

* Received new notification (30 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: FF BB FF 9D FF 8E FF 9B FF AB FF AF FF B5 FF BF FF DA FF

* Received new notification (28 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 3 4
* Successfully received 29 16-bits sample values from Mindflex in RAW mode - Time(ms) = 58000
   - Sample values (DEC): ***--- EYE BLINK DETECTED ---*** rawValueDEC = -91 lapsed counter_event1 =
26--------------------------------------------------

* Received new notification (32 ms):
   - Connection handle: 40
   - Characteristic value attribute handle: B
   - Notified value: 4 55 0 63 0 5A 0 41 0 28 0 24 0 20 0 20 0 1B 0 17
```

Figura 6.16 - Detección de un parpadeo de ojos

Capítulo 7. Conclusiones

Tras haber completado los objetivos propuestos para este Trabajo Fin de Máster, en este capítulo se recogen las conclusiones obtenidas.

7.1 Conclusiones

En primer lugar, en cuanto a los objetivos planteados inicialmente en este Trabajo Fin de Máster, se puede afirmar que han sido cumplidos durante su realización. Así, ha sido posible, diseñar e implementar una plataforma HW/SW inalámbrica basada en el estándar BLE cuyas principales aportaciones se centran en constituir una solución de muy bajo coste y consumo que permite visualizar, transmitir y procesar comandos a partir del registro de la información proporcionada por un sensor de electroencefalograma, y actuar posteriormente en base a ellos.

Para ello, en primer lugar, se ha llevado a cabo un estudio teórico del estándar BLE. A continuación, se ha verificado la correcta recepción de paquetes con la información de EEG del módulo TGAM integrado en el dispositivo *Mindflex*, tanto en modo NORMAL como en modo RAW. Una vez realizada dicha verificación, se ha implementado la comunicación e integración de los dispositivos IoT *Bluz DK* y *RedBear Duo,* que además supone un valor añadido y que podrá ser empleado en futuros proyectos.

Por último, una vez comprobada la correcta comunicación entre el dispositivo *RedBear Duo* actuando como dispositivo *Central,* y el dispositivo *Bluz DK* actuando como dispositivo *Peripheral,* se ha procedido a registrar en el dispositivo *Bluz DK* los datos de EEG en modo RAW provenientes del módulo TGAM del dispositivo *Mindflex,* y enviarlos vía BLE al dispositivo *Central,* que es capaz de realizar la acción asociada a la detección del parpadeo de ojos, que en el caso del presente TFM ha consistido en el encendido de un LED.

Por otra parte, la realización de este TFM ha llevado a la identificación de un problema en el *firmware* del dispositivo *Bluz DK,* relacionado con la lectura del CCCD, que fue comunicado al foro de desarrolladores, reconocido y corregido en la *release* 2.1.50 del *firmware* del dispositivo *Bluz DK*.

Asimismo, se detectó una limitación relacionada con el número de paquetes enviados por intervalo de conexión por parte del dispositivo *RedBear Duo* como dispositivo *Central*, y que en principio afecta al *chip Broadcom* BCM43438, integrado en el

dispositivo *RedBear Duo*. Esta limitación ha sido confirmada por el desarrollador de la librería *BTStack* en la que se basa la implementación del *firmware* del dispositivo *RedBear Duo*, y aún se encuentra a la espera de ser solucionada.

7.2 Líneas futuras

Cabe destacar que la plataforma desarrollada en este Trabajo Fin de Máster, basada en el uso de dispositivos IoT, permite controlar el estado de un LED mediante la información EEG obtenida a partir del módulo TGAM. Sin embargo, este hecho pretende ser el punto de inicio para lograr permitir a personas con discapacidad interactuar con el medio que les rodea a partir del registro de la actividad bioeléctrica cerebral. En este sentido se podrían desarrollar funcionalidades relacionadas con el ámbito de la domótica, como puede ser el control de persianas, iluminación, étc.

Por otra parte, otra posible línea futura del presente TFM puede ser la integración directa de un sensor de EEG que presente conectividad BLE con un dispositivo IoT que soporte dicho estándar. De esta forma se simplificaría la plataforma HW/SW propuesta ya que se reduciría el número de dispositivos IoT necesarios.

Finalmente, resultaría conveniente programar la funcionalidad del dispositivo *RedBear Duo* como *Central*, directamente a partir de la librería *BTStack*, en lugar de hacerlo utilizando las funciones propias del *firmware* del dispositivo *RedBear Duo*, con el fin de confirmar que la limitación detectada en este TFM, relativa al número de paquetes que se pueden transferir por intervalo de conexión, está relacionada con el *chip* BCM43438, y no con el *firmware* del dispositivo *RedBear Duo*.

Bibliografía

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

[2] Almeida, Edwing A.; Buitrón, Marcela. "*The Internet of Things and the future design*". México: Universidad Autónoma Metropolitana-Azcapotzalco. *Academia*, 2015.

[3] Informe Gartner, Noviembre 2015. Disponible en: http://www.gartner.com/newsroom/id/3165317 [Último acceso: Julio de 2017]

[4] Domingo, Mari Carmen; "*An overview of the Internet of Things for people with disabilities*". Barcelona: Universidad Politécnica de Barcelona. *Journal of Network and Computer Application, 2012.*

[5] *ThinkGear ASIC*. Disponible en: http://developer.neurosky.com/docs/doku.php?id=what_is_thinkgear [Último acceso: Julio de 2017]

[6] *Bluz DK*. Disponilble en: http://bluz.io/ [Último acceso: Julio de 2017]

[7] *RedBear Duo*. Disponible en: https://redbear.cc/product/wifi-ble/redbear-duo.html [Último acceso: Julio de 2017]

[8] Kevin Townsend, Carles Cufí, Akiba & Robert Davidson; "*Getting Started with Bluetooth Low Energy*". O'reilly, 2014.

[9] Maximizing BLE Throughput on iOS and Android. Disponible en: https://punchthrough.com/blog/posts/maximizing-ble-throughput-on-ios-and-android [Último acceso: Julio de 2017]

[10] Mindflex. Disponible en: https://store.neurosky.com/products/mindflex [Último acceso: Julio de 2017]

[11] TGAM Features + Technical Specifications. Disponible en: https://cdn.hackaday.io/files/11146476870464/TGAM%20Datasheet.pdf [Último acceso: Julio de 2017]

[12] Hacks. Disponible en: http://developer.neurosky.com/docs/doku.php?id=projects [Último acceso: Julio de 2017]

[13] ThinkGear Serial Stream Guide. Disponible en: http://developer.neurosky.com/docs/doku.php?id=thinkgear_communications_protocol [Último acceso: Julio de 2017]

[14] USB to TTL Serial Cable. Disponible en: https://www.adafruit.com/product/954 [Último acceso: Julio de 2017]

[15] Mindflex EEG with raw data over Bluetooth. Disponible en: http://developer.neurosky.com/features/mindflex-eeg-with-raw-data-over-bluetooth/ [Último acceso: Julio de 2017]

[16] Particle IDE. Disponible en: https://build.particle.io/ [Último acceso: Julio de 2017]

[17] *RedBear Duo.* Disponible en: https://github.com/redbear/Duo [Último acceso: Julio de 2017]

[18] Arduino Brain Library. Disponible en: https://github.com/kitschpatrol/Brain [Último acceso: Julio de 2017]

[19] Bluz DK. Disponible en: http://docs.bluz.io/hardware/dk/ [Último acceso: Julio de 2017]

[20] App LightBlue. Disponible en: https://punchthrough.com/ [Último acceso: Julio de 2017]

[21] App BLEScanner. Disponible en: http://www.bluepixeltech.com/ [Último acceso: Julio de 2017]

[22] RedBear Discussion Forums: Data transfer between RedBear Duo and Bluz DK. Disponible en: http://discuss.redbear.cc/t/solved-data-transfer-between-red-bear-duo-and-bluzdk/1355 [Último acceso: Julio de 2017]

[23] BTSTack. Disponible en: https://github.com/bluekitchen/btstack [Último acceso: Julio de 2017]

[24] BluzDK-Firmware: 2.1.50 Release for DK and Gateway. Disponible en: https://github.com/bluzDK/bluzDK-firmware/releases/tag/v2.1.50 [Último acceso: Julio de 2017]

[25] nRF Connect. Disponible en: http://www.nordicsemi.com/ [Último acceso: Julio de 2017]

[26] Bluefruit LE Sniffer https://www.adafruit.com/product/2269 [Último acceso: Julio de 2017]

[27] Wireshark. Disponible en: https://www.wireshark.org/ [Último acceso: Julio de 2017]

[28] "A New EEG Acquisition Protocol for Biometric Identification Using Eye Blinking Signals" Disponible en: http://www.mecs-press.org/ijisa/ijisa-v7-n6/IJISA-V7-N6-5.pdf [Último acceso: Julio de 2017]

[29] BlinkTalk. Disponible en: https://developer.mbed.org/users/RorschachUK/code/BlinkTalk/ [Último acceso: Julio de 2017]

[30] 6N138/ 6N139 Vishay Semiconductors. Disponible en: http://i2c2p.twibright.com/datasheet/6n139.pdf [Último acceso: Julio de 2017]

Anexo. Contenido del CD-ROM

Adjunto a la memoria de este Trabajo Fin de Máster se encuentra disponible un CD-ROM. En este anexo se describe la estructura del mismo.

A.1 Estructura del CR-ROM

El contenido y la estructura del CD-ROM que se adjunta es la siguiente:

- Memoria.pdf: Versión electrónica del Trabajo Fin de Máster, en el formato del IUMA.
- Resumen.pdf: Artículo resumen de la memoria justificativa en inglés.
- Póster.pdf: Póster del trabajo realizado en DIN A0 y redactado en español, en formato .pdf.
- Póster.ppt: Póster del trabajo realizado en DIN A0 y redactado en español, en formato .ppt.
- EyeBlink1.mov: Vídeo demostrativo de la correcta detección del parpadeo de ojos en el dispositivo *RedBear Duo* a partir de las muestras EEG recibidas de su conexión con el puerto serie del módulo TGAM integrado en el dispositivo *Mindflex*.
- EyeBlink2.mov: Vídeo demostrativo de la correcta detección del parpadeo de ojos en el dispositivo *RedBear Duo* a partir de las muestras EEG recibidas de su conexión BLE con el dispositivo *Bluz DK*, conectado con el módulo TGAM integrado en el dispositivo *Mindflex* a través del puerto serie.