UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de un sistema de representación de textos en código Braille

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: Dª. Araceli Marrero Mendoza

Tutores: D. Valentín De Armas Sosa

D. Félix B. Tobajas Guerrero

Fecha: Enero de 2019

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de un sistema de representación de textos en código Braille

HOJA DE EVALUACIÓN

Calificación: _____

PRESIDENTE

Fdo._____

VOCAL **SECRETARIO**

Fdo._____ **Fdo.** _____

Fecha: **Enero de 2019**

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Lista de acrónimos

ADC	*Analog Digital Converter*
API	*Aplication Programming Interface*
ARM	*Advanced RISC Machine*
ASCII	*American Standard Code for Information Interchange*
ATT	*Attribute protocol*
BD-LLC	*Bi-Directional Logic Level Converter*
BLE	*Bluetooth Low Energy*
CLK	*Clock*
CA	Corriente Alterna
CAN	*Controller Area Network*
CC	Corriente Continua
CCCD	*Client Characteristic Configuration Descriptor*
COITT	Colegio Oficial de Ingenieros Técnicos de Telecomunicación
CPU	*Central Processing Unit*
DAC	*Digital Analog Converter*
DC	*Direct Current*
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
FHSS	*Frequency Hopping Spread Spectrum*
GAP	*Generic Access Profile*
GATT	*Generic Attribute profile*
GFSK	*Gaussian Frequency Shift Keying*
GITT	Grado en Ingeniería en Tecnologías de la Telecomunicación
GND	*Ground*
HCI	*Host Controller Interface*

HV	*High Voltage*
HW	*Hardware*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronics Engineers*
IGIC	Impuesto General Indirecto Canario
INE	Instituto Nacional de Estadística
IoT	*Internet of Things*
ISM	*Industrial Scientific Medical*
L2CAP	*Logical Link Control and Adaption Protocol*
LED	*Light-Emitting Diode*
LL	*Link Layer*
LV	*Low Voltage*
MCU	*Microcontrollller Unit*
MTU	*Maximum Transmission Unit*
OMS	Organización Mundial de la Sadud
PC	*Personal Computer*
PCB	*Printed Circuit Board*
PDU	*Protocol Data Unit*
PHY	*Physical layer*
PWK	*Pulse-Width Modulation*
RGB	*Red, Green, Blue*
RTC	*Real-Time Clock*
RTOS	*Real-Time Operating System*
SMP	*Security Manager Protocol*
SPI	*Serial Peripheral Interface*
SRAM	*Static Random Access Memory*

STBR	*Strobe*
SW	*Software*
TFG	Trabajo Fin de Grado
TIC	Tecnologías de la Información y Comunicación
UUID	*Universally Unique Identifier*
ULPGC	Universidad de Las Palmas de Gran Canaria

Parte I: Memoria

Capítulo 1. Introducción

1.1 Antecedentes

La Organización mundial de la Salud (OMS) [1] estima que en el mundo existe un índice del 15% de población que sufre algún tipo de minusvalía, es decir, aproximadamente 1000 millones de personas. Este número está al alza debido al envejecimiento poblacional que se está produciendo. En el mundo existen aproximadamente 36 millones de personas con ceguera y alrededor de 217 millones con diferentes grados de discapacidad visual [2].

En el caso de España, el Instituto Nacional de Estadística (INE) [3] realizó en 2008 el último censo oficial de discapacitados, en el cual se estimó que existen 3,84 millones de personas con algún tipo de discapacidad. En la Figura 1, se puede ver la pirámide de población con discapacidad en España, separada por sexo y edad.

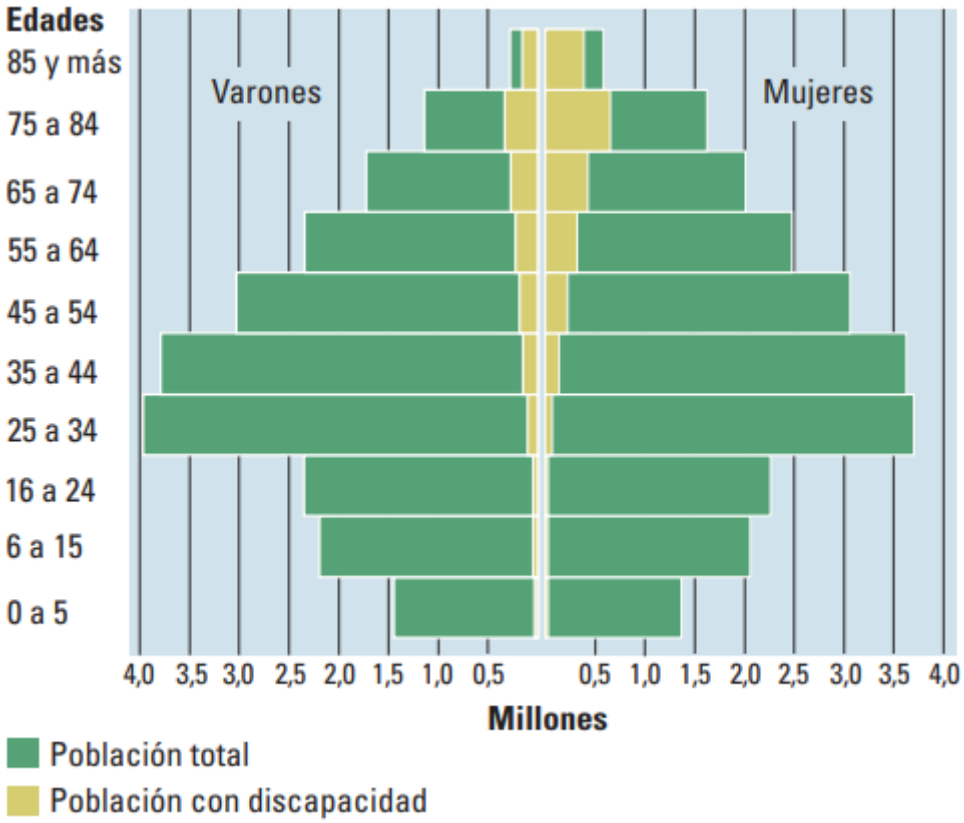

Figura 1. Pirámide de población en España con discapacidad.

En la Figura 2 se representan los problemas más frecuentes que causan la discapacidad en España. En esta gráfica, los porcentajes marcados indican la cantidad de personas que padecen algún grado de las minusvalías indicadas, dentro de la totalidad de discapacitados existentes en España.

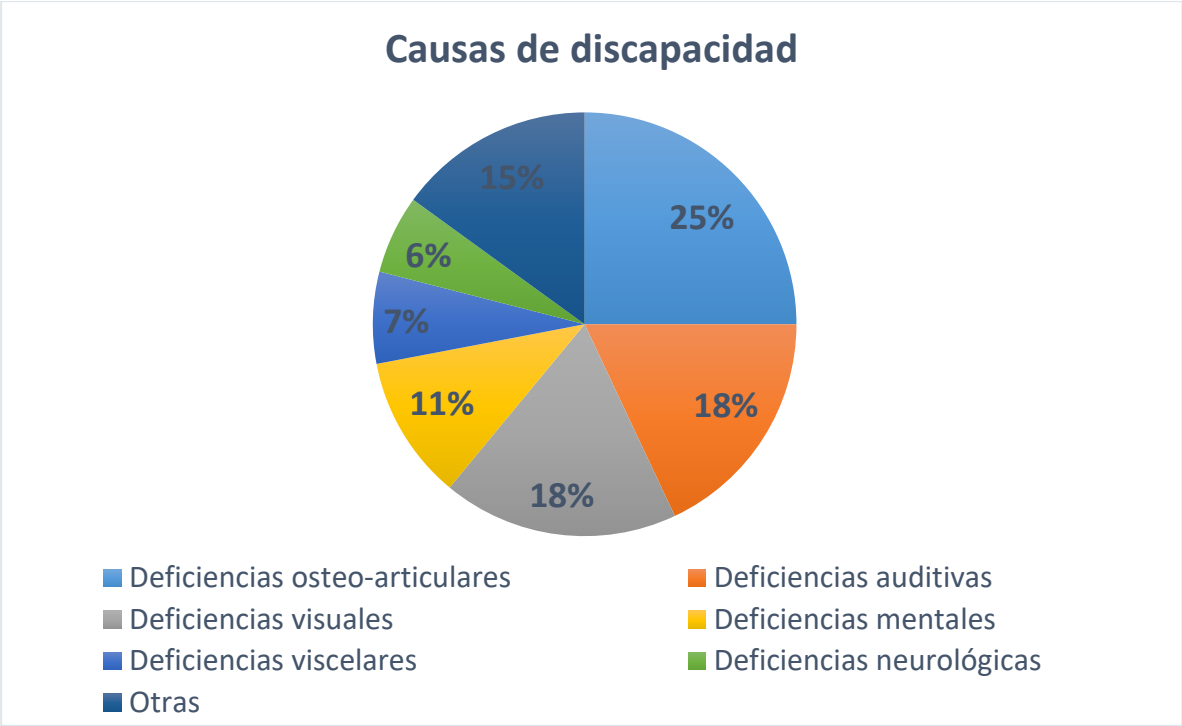

Figura 2. Causas de discapacidad en España.

El 18% de la población discapacitada posee deficiencias visuales. Se calcula que más del 80% de los casos se podrían evitar o curar, pero el resto necesita ayuda para sobrellevar esta discapacidad.

Al hablar de ceguera o deficiencia visual, se hace referencia a condiciones caracterizadas por una limitación total o muy seria de la función visual. Las personas con ceguera son aquellas que no ven nada en absoluto, o solamente tienen una ligera percepción de luz, mientras que las personas con deficiencia visual son aquellas que con la mejor corrección posible podrían ver o distinguir.

De la información utilizada para vivir, el 80% pertenece o usa el órgano de la visión. Esto supone que casi toda la información adquirida a diario es obtenida a través de la vista. Debido a ello, tener una deficiencia de visión supone un hándicap tanto físico, como psicológico. Los individuos que poseen esta discapacidad tendrán problemas para sus desplazamientos, en las actividades realizadas en la vida diaria, y a la hora de obtener información, entre otros.

En el siglo XVIII Diderot fue pionero en exponer que había que educar a las personas ciegas, para mejorar sus facultades. En 1784, Valentin Haüy fue el precursor en la fundación escuelas para ciegos, llegando a enseñar a leer con caracteres de imprenta de alto relieve.

En 1829 Louis Braille, un francés ciego, se basó en un sistema militar de comunicación con un código táctil para inventar un sistema de lectoescritura. Tardó trece años en simplificar dicho código, de tal manera que lo adaptó a las capacidades y necesidades de las personas

con discapacidad visual [4]. Este sistema fue adquirido con mucha facilidad, debido a su versatilidad a la hora de utilizarlo en diferentes lenguas y ser fácilmente adaptable a las matemáticas, la música, etc. En la Figura 3 se puede ver el alfabeto representado en código Braille.

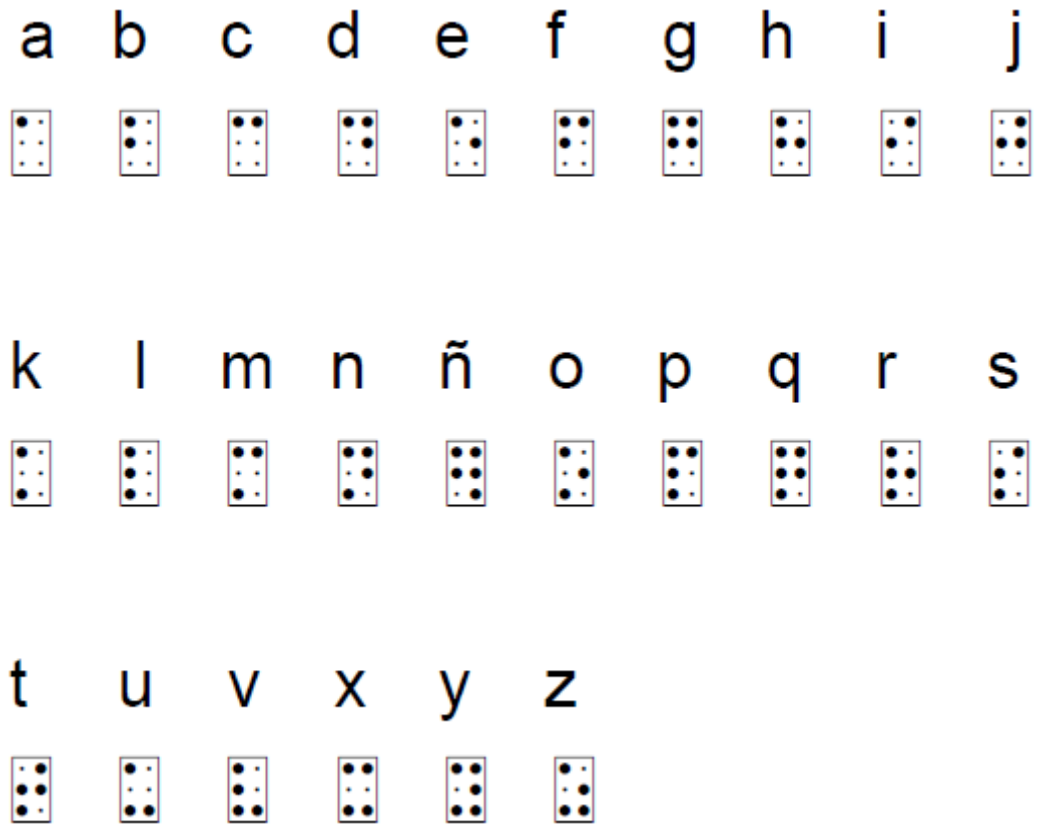

Figura 3. Alfabeto representado en código Braille.

El elemento más utilizado por este grupo de personas es el código Braille, el alfabeto de lectoescritura internacional para personas con discapacidad visual. Este sistema consta normalmente de 6 puntos con posiciones binarias. Existen 256 caracteres en esta alfabetización, incluyendo números, notas musicales o signos de puntuación.

En la actualidad, donde las Tecnologías de la Información y Comunicación (TIC) son la piedra angular de la sociedad, siguen existiendo restricciones para las personas con distintos tipos de discapacidad. Pueden encontrarse en casi cualquier lugar los símbolos del código Braille, pero siguen existiendo problemas con la reproducción o impresión de textos debidos al alto coste que representan por lo general los dispositivos comercialmente disponibles. Aun así, estos dispositivos se encuentran en un nivel de desarrollo muy inferior al de la tecnología no inclusiva.

Es por esto que ya se está trabajando en la generación de libros electrónicos en Braille, o en la implementación de sistemas que faciliten dicha lectura o escritura en distintos dispositivos, como el que se muestra en la Figura 4 [5], que alcanzan precios

considerablemente elevados, entorno a los 2500€. Este dispositivo es el *Brailliant BI* 40 de la empresa *HumanWare,* y se compone de una pantalla con la que se puede realizar una lectura y la navegación en distintos dispositivos de manera intuitiva. Los inconvenientes más evidentes de este dispositivo son el alto coste que representa, las grandes dimensiones (31 x 8.7 x 1.8 cm) y el peso, que ronda los 650 gramos.

Figura 4. Brailliant BI 40 de la empresa *HumanWare*.

Así, se plantea la búsqueda de alternativas de bajo coste, como podría ser *blitab* [6], una tableta con la cual se implementa, no sólo la lectura y la escritura por puntos, sino que también permite la representación de gráficos, entre otras funciones. Esta tableta se pretende que salga por unos 430€. Parece que puede llegar a ser el mejor producto en el mercado para las personas invidentes, pero por ahora no se puede adquirir, dado que la empresa está teniendo problemas para finalizar el producto antes de lanzarlo. En la Figura 5 se muestra el prototipo con el que se han estado promocionando la empresa desarrolladora de *blitab*.

Figura 5. La primera *tablet* para ciegos, *blitab*.

Otra alternativa, que si está actualmente disponible en el mercado, es *BraiBook*. Este dispositivo fue creado por el español Carlos Madolell, y realiza una lectura automática basada en una célula de 8 puntos que se actualiza en función de los símbolos a representar en cada momento. De esta manera las personas invidentes pueden leer cualquier documento que deseen [7]. Este dispositivo, representado en la Figura 6, es un *"eReader*" para ciegos, pues es capaz de transformar todos los textos a código Braille, ya sean transmitidos vía USB, vía *Bluethooth* o con una microSD. *BraiBook* tiene un precio de mercado de 445€, tiene dimensiones parecidas a un móvil, pesa 350 gramos y tiene una autonomía de 5 horas.

Figura 6. *BraiBook*, el "*eReader*" para invidentes.

Este Trabajo Fin de Grado (TFG) pretende contribuir con el desarrollo de un prototipo de sistema de representación del código Braille con conectividad *Bluetooth Low Energy* (BLE) de bajo coste y reducido consumo de potencia. En este caso no se pretende inicialmente conseguir dimensiones en el prototipo como para que la lectura en Braille sea directa o cómoda.

1.2 Objetivos

El objetivo principal del presente Trabajo Fin de Grado (TFG) consiste en la implementación de una plataforma Hardware/Software (HW/SW) que permita la representación de caracteres del código Braille a partir del desarrollo del prototipo de una célula de 6 puntos y su integración con un sistema basado en MCU *(MicroControlller Unit)*. Para la consecución de este objetivo se plantean como objetivos parciales, en primer lugar, la implementación de una plataforma HW/SW funcional a partir de la integración la célula *modul P16* de la empresa *metec* [8] con un sistema integrado basado en microcontrolador, que proporcione conectividad BLE para la transmisión de los textos a reproducir en código Braille. En segundo lugar, se realizará el desarrollo de un prototipo de célula de 6 puntos con la misma funcionalidad, basada en el uso de motores *push-pull*, para su integración en la plataforma HW/SW final.

Así, a partir del análisis de los requisitos funcionales de los componentes necesarios para la operación de la célula Braille *modul P16*, se realizará un *firmware* para el sistema basado en MCU integrado en la plataforma HW/SW, que permitirá al usuario la lectura de la

representación en código Braille correspondiente a los caracteres de un texto transmitido desde dispositivos móviles, mediante BLE. Posteriormente, se llevará a cabo la implementación del prototipo de célula a partir de componentes discretos, y finalmente, se integrará esta célula con el sistema basado en MCU con el fin de obtener la plataforma HW/SW final.

1.3 Peticionario

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

1.4 Organización de la memoria

El presente documento está distribuido en cuatro partes bien diferenciadas: Memoria, Pliego de condiciones, Presupuesto y Anexo.

La memoria comprende 7 capítulos, además de la bibliografía empleada. El contenido de estos capítulos es el que se resume a continuación:

- Capítulo 1. Introducción. Este capítulo recoge los antecedentes que han dado lugar a la realización de este TFG, los documentos del mismo, el peticionario y la estructura del documento.
- Capítulo 2. Integración de la célula de lectura Braille *modul P16* de *metec*. Este capítulo comprenderá el análisis de los materiales especificados utilizando las hojas de características, y la integración de dichos componentes para el correcto funcionamiento de la célula. Se estudiará el funcionamiento de la célula *modul P16* de *metec*.
- Capítulo 3. *Bluetooth Low Energy*. Este capítulo supone una introducción a los sistemas de comunicaciones basados en esta tecnología de comunicación inalámbrica.
- Capítulo 4. Código de integración para el funcionamiento de la célula *modul P16* de *metec*. Este capítulo se basa en el diseño *software* de un código que cumpla con todas las especificaciones establecidas como objetivos, en este TFG sobre una célula Braille integrada en la plataforma hardware desarrollada en el capítulo 2.
- Capítulo 5. Diseño e implementación de una célula de lectura Braille basada en motores *push-pull*. Este capítulo trata del diseño de una célula Braille de lectura utilizando motores *push-pull*. En éste, se analizarán los componentes adecuados,

estudiando las hojas de características de los fabricantes, para conseguir implementar una célula de lectura Braille que cumpla los objetivos de bajo coste, baja potencia consumida, y lectura de datos recibidos vía *BLE*. Una vez encontrados dichos materiales, se procede a la implementación sobre una placa de prototipado.

- Capítulo 6. Desarrollo e integración de un código para la célula de lectura Braille diseñada con motores *push-pull*. Con intención de adaptar el código generado en el Capítulo 4, este capítulo se centrará en realizar un código que se ajuste a las especificaciones de funcionamiento de la célula de lectura Braille diseñada, e intentar cumplir con todos los requisitos establecidos como objetivos en este TFG.
- Capítulo 7. Conclusiones. Tras haber completado los objetivos propuestos para este TFG, en este capítulo se analizarán las conclusiones y los resultados obtenidos con el desarrollo del proyecto realizado.

El pliego de condiciones está compuesto de unas tablas en las que se detallan los sistemas *hardware* y las plataformas *software* utilizadas. En el presupuesto se detalla el coste asociado a la realización de este TFG. Por último en el aparatado Anexo se incluirán diferente documentación referente a la memoria de TFG.

Capítulo 2. Integración de la célula de lectura Braille *modul P16* de *metec*

2.1 Introducción

Se dispone de varios dispositivos de la empresa *metec*, para la lectura en código Braille, con los cuales se pretende descubrir y aprender su modo de funcionamiento. Para que esto sea posible, es necesario preparar un circuito, siguiendo las indicaciones recogidas en las hojas de características facilitadas por el fabricante, consiguiendo que dichos componentes funcionen correctamente.

El problema principalmente planteado en este capítulo es la descripción de todos los componentes que puedan hacer que la célula *modul* P16 de la empresa *metec* trabaje de forma correcta. Para que esto se cumpla, es imprescindible desglosar cada componente finalmente usado, de tal manera que se demuestre que es el componente idóneo para la tarea encomendada.

2.2 Descripción de los materiales utilizados

Como se ha mencionado con anterioridad, este capítulo ira mostrando los diferentes componentes que se utilizaran en un montaje posterior para la lectura Braille utilizando los dispositivos de la empresa *metec*. Puesto que inicialmente se desconoce qué será necesario para ello, en la Figura 7 se muestran los pasos que inicialmente se han seguido en este TFG.

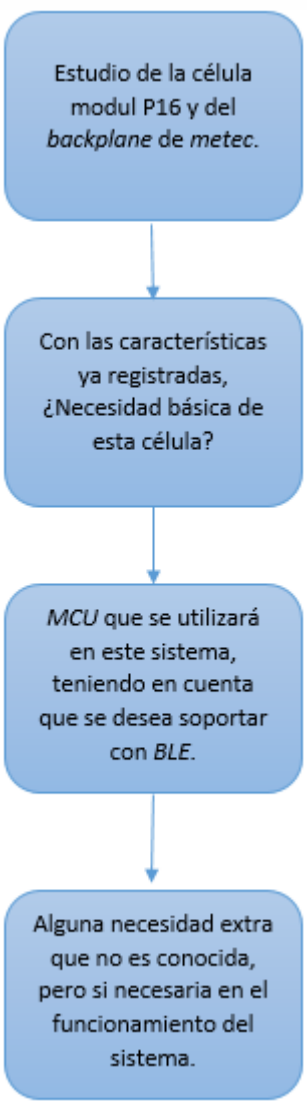

Figura 7. Diagrama de flujo de la metodología a seguir.

2.2.1 Descripción de la célula Braille *modul P16* y su *backplane*

En este apartado se analizarán en profundidad todos los aspectos relacionados con la célula *modul P16* de *metec* y el *backplane* que la acompaña, finalizando con todos los conceptos claros de cómo funciona a nivel mecánico, y las señales que necesita en sus entradas para su correcto funcionamiento.

Se dividirá el estudio en dos etapas, pues primero se analizará la célula, y posteriormente se estudiará en profundidad el *backplane*.

- Célula *modul P16*.

Esta célula de lectura en código Braille es un sistema final utilizado en este caso para el estudio de su funcionamiento. Se puede ver el aspecto general de esta célula en la Figura 8.

Figura 8: Célula Braille *modul P16* de *metec*.

Esta célula Braille con actuadores piezoeléctricos es una unidad que muestra o refresca un carácter del código Braille en función de una entrada de control [9]. Está compuesta de una base estructural, una placa de circuito impreso (PCB), bimorfos piezoeléctricos (versátiles transductores electromecánicos capaces de convertir energía mecánica en energía eléctrica), una tapa y los puntos para la representación del código Braille. Cada célula está compuesta por ocho puntos, formando una matriz rectangular de tamaño 4x2, y a su vez estos puntos son controlados por bimorfos piezoeléctricos. Éstos se encuentran dentro de la célula Braille y unidos a cada punto, y su funcionalidad es doblarse hacia arriba o abajo dependiendo del valor de tensión aplicado. El bimorfo en reposo es una placa recta. Si existe tensión suficiente, el bimorfo se doblará hacia arriba, produciendo que los puntos blancos se levanten. Si por el contrario no se aplica la tensión suficiente, entonces se doblará hacia abajo, produciendo que los puntos bajen, como se muestra en la Figura 9.

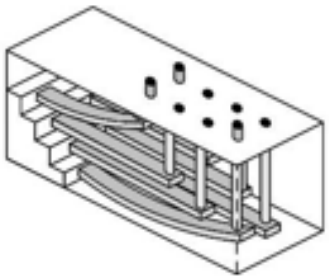

Figura 9. Muestra de los bimorfos piezoeléctricos en funcionamiento.

Los puntos están en reposo debajo de una superficie táctil plana, lo que implica mayor comodidad para la lectura en código Braille. Estos puntos serán tomados a partir de ahora como se puede ver en la Figura 10, y entendiendo que en este caso no se desea trabajar con los ocho puntos, sino con seis, como establece el código Braille internacional. Es por ello que los puntos 7 y 8 se establecerán más adelante siempre a nivel bajo, o 0V.

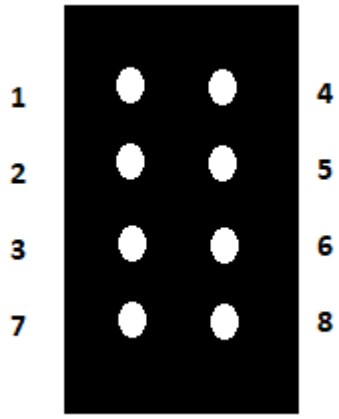

Figura 10. Posicionamiento elegido de la célula Braille *modul P16.*

Las dimensiones de la célula *modul P16* de *metec* son 6.42 x 16. 8 x 84 milímetros (mm), y se muestran en la Figura 11. Hay que tener en cuenta que todas las dimensiones mostradas se encuentran en milímetros.

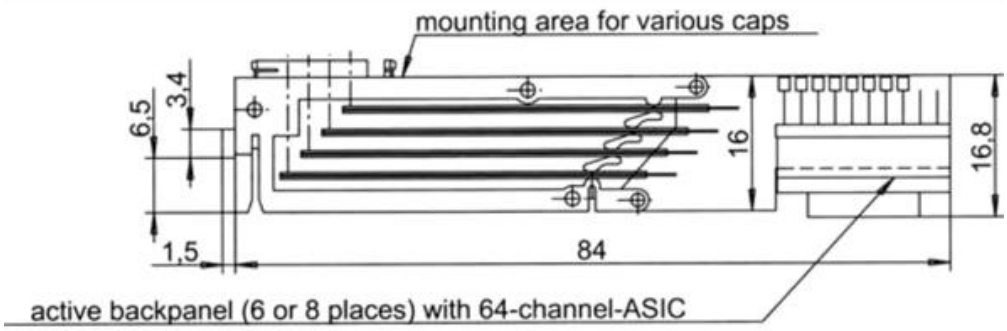

Figura 11. Dimensiones de la célula *modul* P16 de metec.

Posee 10 pines en su placa *PCB*, como se presenta en la Figura 12, en el cual se puede ver la función de cada uno de ellos. Así, el primero es la entrada de tensión de alimentación de 200 voltios (V), la segunda es la tierra del sistema (GND), y el resto representa la señal de control de cada punto de la célula. Cuando se aplica la tensión de 200V, los bimorfos piezoeléctricos se doblan hacia arriba, produciendo que el punto se eleve, mientras que en caso de aplicarse 0V, se invierte la situación. Estos pines van conectados al Backplane. Un dato relevante a tener en cuenta es que los puntos se mantienen elevados 24 milisegundos (ms), antes de refrescarse.

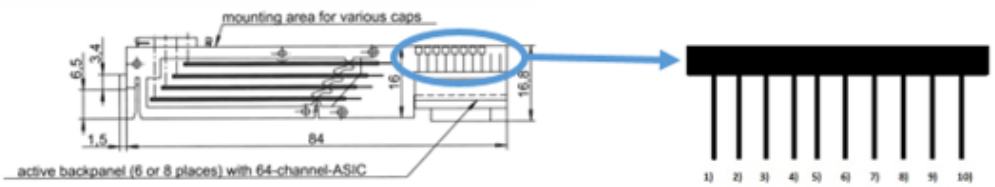

Figura 12. Pines de la célula *modul* P16.

Las principales características a tener en cuenta para el funcionamiento de esta célula, se pueden muestra en la Tabla 2. 1.

Tabla 2. 1. Requerimientos técnicos para el funcionamiento de la célula *modul P16*, de *metec*.

Requerimientos de Potencia	200V ±10% con un nivel máximo de 240V. 1.2 microamperios (µA) por punto	
Especificaciones Ambientales	Operacionales	Soporta un temperatura de 10º C a 40º C Soporta una humedad del 10% - 70% sin condensación.
	Almacenamiento	Soporta un temperatura de -15º C a 60º C Soporta una humedad del 5% - 90% sin condensación.
	Accesorios	-Bandejas de metal - *Backplane* pasivo - Tapas
	Promedio de actuaciones piezoeléctricas	$>10^9$

- *Backplane* de *metec*.
  Este dispositivo no es más que una placa *PCB* con 10 pines de entrada y 6 u 8 ristras con hueco para interconectar varias células Braille en paralelo, que implementa un registro de desplazamiento de 8 bits [ref. página de metec para la célula]. En la Figura 13 se puede visualizar el aspecto general de este componente.

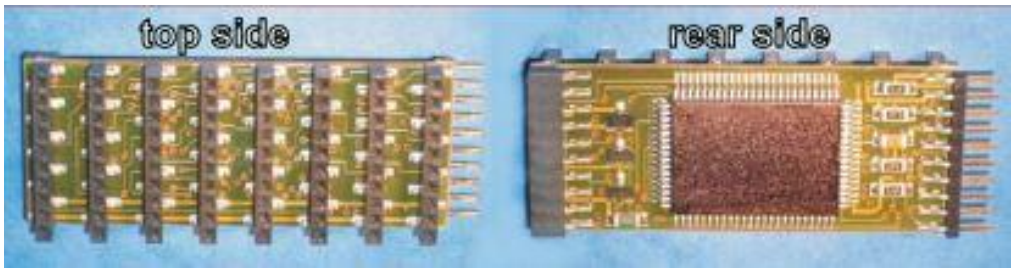

Figura 13. *Backplane* de metec.

Un registro de desplazamiento es un circuito secuencial lógico, usado normalmente para almacenar datos digitales. Se compone de una serie de *flip-flop* conectados en cascada, de tal manera que la salida de uno es la entrada del siguiente. Todos los *flip-flop* están conectados a un reloj común y se actualizan simultáneamente[9].

En este TFG, se utilizará un registro de desplazamiento para transformar un bus de datos de serie a paralelo, ya que es necesario que los datos que vienen en serie sean transmitidos en paralelo a cada punto de la célula Braille *modul P16*. Como ya se explicó cuando se desglosó toda la información referente a la célula *modul P16* de *metec*, cada célula Braille está compuesta por 8 pines asociados a cada punto de la célula Braille. El *chip* integrado en el *backplane* para llevar a cabo la función de registro de desplazamiento es el HV507PG [10]. Se trata de un *chip* de baja tensión a alta tensión convirtiendo de serie a paralelo con 64 salidas *push-pull*. También es utilizado en aplicaciones de múltiples salidas, alto voltaje y baja corriente. Este chip, integrado en el *backplane*, se puede ver en la Figura 14 y su diagrama de bloques funcional en la Figura **15**.

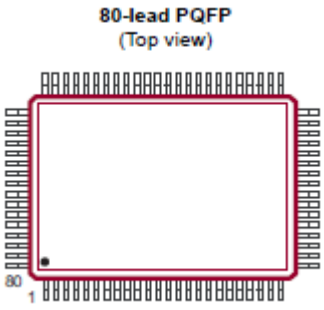

Figura 14. *Footprint* del chip HV507PG.

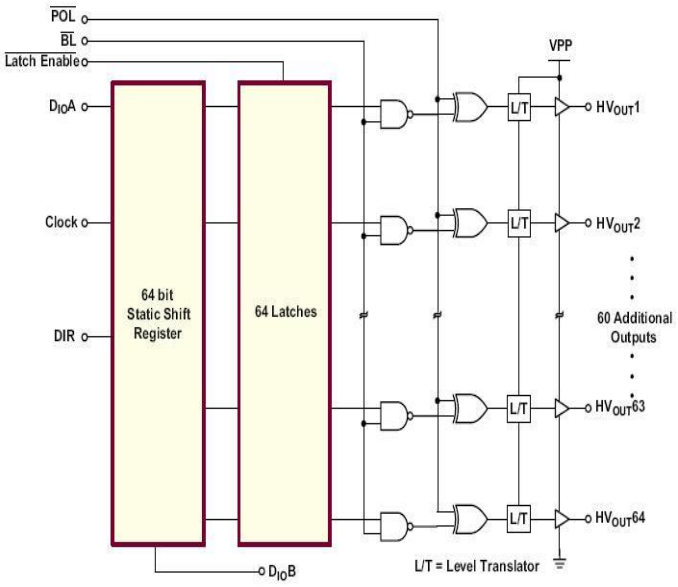

Figura 15. Diagrama de bloques funcional del chip HV507PG.

Las condiciones de operatividad de este chip se pueden obtener en las tablas de su *datasheet* (Tabla 2. 2 y Tabla 2. 3), y que se representan en la Figura 15.

Tabla 2. 2. Condiciones recomendadas.

Parameter	Sym.	Min.	Typ.	Max.	Unit	Conditions
Logic Supply Voltage	V_{DD}	4.5	5	5.5	V	
High-voltage Supply Voltage	V_{PP}	60	—	300	V	
High-level Input Voltage	V_{IH}	V_{DD}–0.9V	—	V_{DD}	V	
Low-level Input Voltage	V_{IL}	0	—	0.9	V	

Tabla 2. 3. Especificaciones eléctricas en corriente continua (CC).

Electrical Specifications: For V_{DD} = 5V, V_{PP} = 300V and T_A = 25°C.							
Parameter		**Sym.**	**Min.**	**Typ.**	**Max.**	**Unit**	**Conditions**
V_{DD} Supply Current		I_{DD}	—	—	15	mA	f_{CLK} = 8 MHz, f_{DATA} = 4 MHz, $\overline{LE}$ = low
Quiescent V_{DD} Supply Current		I_{DDQ}	—	—	200	µA	All V_{IN} = 0V or V_{DD}
High-voltage Supply Current		I_{PP}	—	—	0.5	mA	V_{PP} = 300V, all outputs high
			—	—	0.5	mA	V_{PP} = 300V, all outputs low
High-level Logic Input Current		I_{IH}	—	—	10	µA	V_{IN} = V_{DD}
Low-level Logic Input Current		I_{IL}	—	—	–10	µA	V_{IN} = 0V
High-level Output	HV_{OUT}	V_{OH}	265	—	—	V	V_{PP} = 300V, IHV_{OUT} = –1 mA, ID_{OUT} = –100 µA
	Data Out		V_{DD} –1	—	—	V	
Low-level Output	HV_{OUT}	V_{OL}	—	—	35	V	V_{DD} = 5V, IHV_{OUT} = 1 mA, ID_{OUT} = 100 µA
	Data Out		—	—	1	V	
HV_{OUT} Clamp Voltage		V_{OC}	—	—	V_{PP} + 1.5	V	I_{OL} = 1 mA
			—	—	–30	V	I_{OL} = –1 mA

Se ha suprimido la tabla de especificaciones eléctricas en corriente alterna (CA), debido a que en este Trabajo Fin de Grado no se requiere dicha descripción. En la Tabla 2. 4 se muestran las condiciones o especificaciones de temperatura.

Tabla 2. 4. Especificaciones de temperatura.

Parameter	Sym.	Min.	Typ.	Max.	Unit	Conditions
TEMPERATURE RANGE						
Operating Ambient Temperature	T_A	0	—	+70	°C	
Storage Temperature	T_S	–65	—	+150	°C	
PACKAGE THERMAL RESISTANCE						
80-lead PQFP	θ_{JA}	—	37	—	°C/W	

Y por último, es necesario conocer su tabla de verdad mostrada en la Tabla 2. 5.

Tabla 2. 5. Tabla de verdad de las funciones.

Function	Inputs						Outputs				
							Shift Register		High-voltage Output		Data Out
	Data	CLK	$\overline{LE}$	$\overline{BL}$	$\overline{POL}$	DIR	1	2...64	1	2...64	*
All On	X	X	X	L	L	X	*	*...*	H	H...H	*
All Off	X	X	X	L	H	X	*	*...*	L	L...L	*
Invert Mode	X	X	L	H	L	X	*	*...*	$\overline{*}$	$\overline{*}$...$\overline{*}$	*
Load S/R	H or L	↑	L	H	H	X	H or L	*...*	*	*...*	*
Store Data in Latches	X	X	↓	H	H	X	*	*...*	*	*...*	*
	X	X	↓	H	L	X	*	*...*	$\overline{*}$	$\overline{*}$...$\overline{*}$	*
Transparent Latch Mode	L	↑	H	H	H	X	L	*...*	L	*...*	*
	H	↑	H	H	H	X	H	*...*	H	*...*	*
I/O Relation	$D_{IO}A$	↑	X	X	X	L	$Q_N \rightarrow$	Q_{N+1}	—		$D_{IO}B$
	$D_{IO}B$	↑	X	X	X	H	$Q_N \rightarrow$	Q_{N+1}	—		$D_{IO}A$

Note: H = High-logic level
L = Low-logic level
X = Irrelevant
↑ = Low-to-high transition
↓ = High-to-low transition
* = Dependent on the previous stage's state before the last CLK or last $\overline{LE}$ high

Aparte de todas las especificaciones mecánicas para su funcionamiento, es necesario conocer su *pinout*, es por ello que en la Tabla 2. 6 se recogen todos sus pines, especificando su funcionalidad.

Tabla 2. 6. Tabla de *pinout* del chip HV507PG.

Número de Pin	Nombre del Pin	Descripción
1	HVOUT41	Alta tensión de salida
2	HVOUT42	Alta tensión de salida
3	HVOUT43	Alta tensión de salida
4	HVOUT44	Alta tensión de salida
5	HVOUT45	Alta tensión de salida
6	HVOUT46	Alta tensión de salida
7	HVOUT47	Alta tensión de salida
8	HVOUT48	Alta tensión de salida
9	HVOUT49	Alta tensión de salida

10	HVOUT50	Alta tensión de salida
11	HVOUT51	Alta tensión de salida
12	HVOUT52	Alta tensión de salida
13	HVOUT53	Alta tensión de salida
14	HVOUT54	Alta tensión de salida
15	HVOUT55	Alta tensión de salida
16	HVOUT56	Alta tensión de salida
17	HVOUT57	Alta tensión de salida
18	HVOUT58	Alta tensión de salida
19	HVOUT59	Alta tensión de salida
20	HVOUT60	Alta tensión de salida
21	HVOUT61	Alta tensión de salida
22	HVOUT62	Alta tensión de salida
23	HVOUT63	Alta tensión de salida
24	HVOUT64	Alta tensión de salida
25	VPP	Fuente de alimentación de alto voltaje
26	DIOA	Entrada/Salida del dato serie A
27	NC	No conectado
28	NC	No conectado
29	$\overline{BL}$	Blanking
30	$\overline{POL}$	Polaridad
31	VDD	Baja tensión de salida
32	DIR	Dirección
33	GND	Tensión de tierra lógica
34	HVGND	Tensión de tierra lógica para la alta tensión de salida
35	NC	No conectado
36	NC	No conectado
37	CLK	Reloj de registro de cambio de dato. Las entradas cambian el registro cuando hay un flanco de subida del reloj.
38	$\overline{LE}$	Habilita el *Latch*
39	DIOB	Entrada/Salida del dato serie B
40	VPP	Fuente de alimentación de alto voltaje
41	HVOUT1	Alta tensión de salida
42	HVOUT2	Alta tensión de salida
43	HVOUT3	Alta tensión de salida
44	HVOUT4	Alta tensión de salida
45	HVOUT5	Alta tensión de salida
46	HVOUT6	Alta tensión de salida
47	HVOUT7	Alta tensión de salida
48	HVOUT8	Alta tensión de salida
49	HVOUT9	Alta tensión de salida

50	HVOUT10	Alta tensión de salida
51	HVOUT11	Alta tensión de salida
52	HVOUT12	Alta tensión de salida
53	HVOUT13	Alta tensión de salida
54	HVOUT14	Alta tensión de salida
55	HVOUT15	Alta tensión de salida
56	HVOUT16	Alta tensión de salida
57	HVOUT17	Alta tensión de salida
58	HVOUT18	Alta tensión de salida
59	HVOUT19	Alta tensión de salida
60	HVOUT20	Alta tensión de salida
61	HVOUT21	Alta tensión de salida
62	HVOUT22	Alta tensión de salida
63	HVOUT23	Alta tensión de salida
64	HVOUT24	Alta tensión de salida
65	HVOUT25	Alta tensión de salida
66	HVOUT26	Alta tensión de salida
67	HVOUT27	Alta tensión de salida
68	HVOUT28	Alta tensión de salida
69	HVOUT29	Alta tensión de salida
70	HVOUT30	Alta tensión de salida
71	HVOUT31	Alta tensión de salida
72	HVOUT32	Alta tensión de salida
73	HVOUT33	Alta tensión de salida
74	HVOUT34	Alta tensión de salida
75	HVOUT35	Alta tensión de salida
76	HVOUT36	Alta tensión de salida
77	HVOUT37	Alta tensión de salida
78	HVOUT38	Alta tensión de salida
79	HVOUT39	Alta tensión de salida
80	HVOUT40	Alta tensión de salida

El *backplane,* al igual que la célula *modul P16* de *metec*, tiene 10 pines de entrada. En la Figura 16 se muestra el *footprint* de esta placa PCB, en la que se pueden distinguir 8 ristras de salida en las que irán conectadas las células (en el dispositivo adquirido son 6 ristras, pero su funcionalidad es la misma) y los 10 pines de entrada.

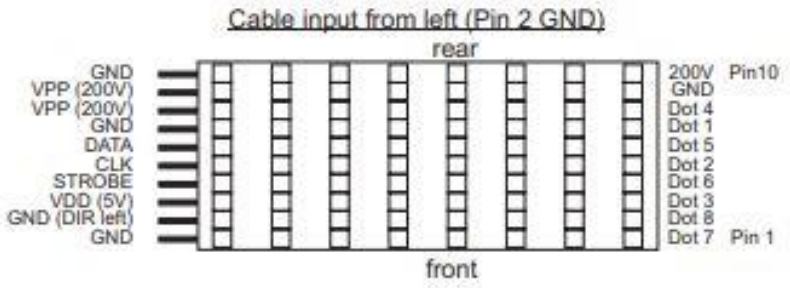

Figura 16. *Footprint* del *Backplane*.

A continuación, en la Tabla 2. 7 se puede visualizar el *pinout* de este componente y el cual es importante cumplir, ya que la desconexión de algún pin puede dar lugar a problemas de funcionamiento. Los pines de los que se está hablando no son estándares y es un dato a tener en cuenta.

Tabla 2. 7. *Pinout* del *backplane* de *metec*.

Número de Pin	Nombre del Pin	Descripción
1	GND	Tensión de tierra lógica
2	GND (DIR izquierda)	En el caso que ocupa este TFG, el DIR será llevado tensión de tierra lógica
3	VDD (5V)	Fuente de alimentación de bajo voltaje
4	*STROBE*	Refresco de los valores de los 8 puntos de la célula Braille a la vez
5	*CLK*	En cada flanco de subida de la señal de reloj, actualiza un dato de uno de los puntos de la célula
6	*DATA*	Bus de datos serie
7	GND	Tensión de tierra lógica
8	VPP (200V)	Fuente de alimentación de alto voltaje
9	VPP (200V)	Fuente de alimentación de alto voltaje
10	GND	Tensión de tierra lógica

En la Tabla 2. 7 se incluyen los pines *STROBE, CLK* y *DATA*, los cuales son pines que reciben datos de control para que sean ejecutados en la célula Braille. Como se puede ver en su definición, el pin *DATA* es un bus serie de datos que transporta la información que se desea representar en la célula de lectura Braille *modul P16* de *metec*. El pin *CLK* utiliza los flancos de subida de la señal de reloj del sistema para actualizar uno a uno los datos que se mostrará en la célula. El pin *STROBE* será el encargado de actualizar todos los valores que se han ido registrando en cada ciclo de reloj de una sola vez para que se pueda leer en la célula el nuevo valor representado en código Braille.

Para entender todo lo explicado en el párrafo anterior, se propone un ejemplo de representación en código Braille el carácter 'a' en la célula *modul P16* de *metec*, el cuál será transferido en serie a través del pin *DATA*. Este carácter tiene un valor de conversión, en código Braille para su lectura, de 0x01 en hexadecimal y 0000 0001 en binario. Debido a que la célula *modul P16* de *metec* tiene ocho puntos para la representación de código Braille, cada bit corresponde a uno de los pines. Entendiendo esto, se puede decir que en cada ciclo de señal de reloj *CLK* se van registrando los valores de cada punto, que en un flanco de subida establece que el punto 1 tendrá un valor de 1, en el siguiente flanco el punto 2 guardará un valor 0 y así consecutivamente. Tras un número de ciclos de la señal de reloj *CLK* igual al número de puntos de la célula, entonces el *STROBE* los actualiza y muestra en la célula para su lectura. Tras esto, la célula representará el valor de 'a' convertido al código Braille. En la Figura 17 se puede ver la conversión descrita, donde el color naranja muestra los puntos que están en relieve, y el color blanco los botones que están escondidos.

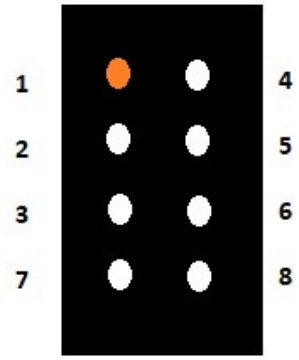

Figura 17. Representación en código Braille del carácter 'a'.

Queda establecer las especificaciones mecánicas necesarias que aparecen en sus hojas de características, y que inevitablemente hay que cumplir si se desea que este registro de desplazamiento o conversor serie-paralelo opere adecuadamente. Todos estos datos aparecen en la Tabla 2. 8.

Tabla 2. 8. Especificaciones mecánicas del *backplane*.

Requerimientos de Potencia	5V ±10% : 25 µA por célula 200V ±10%: nivel absoluto máximo de 240V 5 µA por punto
Máx. tiempo de transición entre CLK y STROBE	125 nanosegundos (ns)
Máx. velocidad de CLK	8 Megahercios (MHz)
Secuencia de datos	7 3 2 1 8 6 5 4 (secuencia de los puntos Braille)

Especificaciones Ambientales	Operacional	Soporta un temperatura de 10º C a 40º C Soporta una humedad del 10% - 70% sin condensación.
	Almacenamiento	Soporta un temperatura de -15º C a 60º C Soporta una humedad del 5% - 90% sin condensación.

En la Tabla 2. 8 se muestra la secuencia de datos necesaria para que la reproducción de datos en una célula Braille de *metec* sea correcta. Esto supone que al cargar los datos en la célula siempre han de comenzar con el sexto bit (recordar que los vectores comienzan en la celda 0), que se corresponderá con el séptimo punto de la célula. Corrigiendo lo explicado en el ejemplo expuesto, en cada ciclo de señal de reloj *CLK* se irán guardando los valores siguiendo la secuencia de puntos 7, 3, 2, 1, 8, 6, 5 y 4 o lo que es lo mismo, guardará los valores que tiene el bus DATA recolocando los bits de datos de manera que se almacenen en el orden 6, 2, 1, 0, 7, 5, 4 y 3. Si se modifica este orden, supondrá obtener a la salida una representación en código Braille que no es la esperada y en consecuencia no podrá ser interpretado correctamente el mensaje que se está transmitiendo.

Las señales *DATA*, *CLK* y *STROBE* han de entrar al *backplane* con una tensión de 5V, dado que si no es así, el *backplane* no las reconoce.

2.2.2 Análisis del Conversor DC-DC de 5V a 185V de metec

El conversor DC-DC (*Direct Current-Direct Current*) es un dispositivo capaz de conmutar de una tensión a otra en corriente continua (CC) [9]. La forma en que realiza esta transformación es almacenando la energía a la entrada, y liberándola a la salida con otra tensión. Existen dos maneras de realizar este almacenamiento, una es con componentes que generan campo magnético, como son los transformadores e inductores, y la otra es con componentes de generación de campo eléctrico, normalmente condensadores. Este conversor puede trabajar a plena capacidad con 40 células Braille conectadas en paralelo y reduciendo su velocidad, hasta 80 células (siempre y cuando se añada una tensión de entrada de 9V). Este dispositivo tendrá una tensión de entrada nominal de 5V, y ofrecerá una tensión salida de aproximadamente 185V, la cual alimentará cada pin responsable de algún punto en la célula Braille. En la Figura 18 se puede ver este dispositivo.

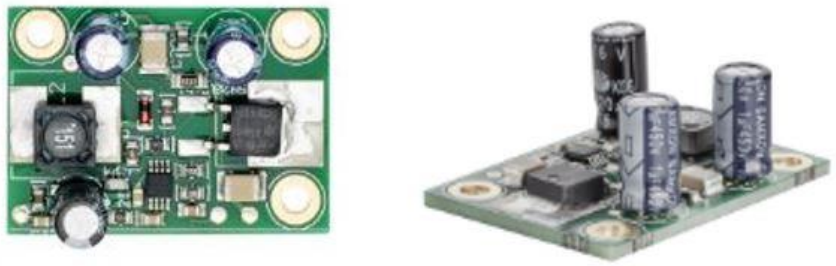

Figura 18. Conversor DC-DC de 5 a 185V de la empresa metec.

Las dimensiones de este conversor son de 26x38 mm, y como en el caso de la célula *modul* P16, todas las dimensiones mostradas en la Figura 19 vienen expresadas en milímetros.

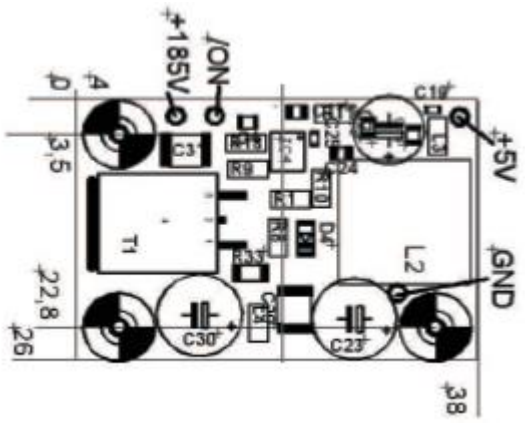

Figura 19. Dimensiones y *footprint* del conversor DC-DC de 5V a 185V

Las consideraciones mecánicas de este conmutador son las que se presentan en la Tabla 2. 9, y que han sido obtenidas de las hojas de características de este dispositivo, proporcionados por el fabricante.

Tabla 2. 9. Especificaciones eléctricas del conversor DC-DC de 5V a 185V.

Tensión de entrada (V_{in})	5V ±5%
Corriente de entrada (I_{in})	Máximo 500 miliamperios (mA)
Tensión de salida (V_{out})	185V ±5%
Corriente de salida (I_{out})	4 mA

Lo único que queda por conocer de este conversor son sus pines de entrada y salida, mostrados en la Tabla 2. 10.

Tabla 2. 10. *Pinout* del conversor DC-DC de 5V a 185V.

Número de Pin	Nombre del Pin	Descripción
1	V_{out}	Tensión de salida del dispositivo a 185V
2	*ON*	Señal de control del conmutador. Si está a 0V el dispositivo está en pleno rendimiento, si por el contrario está a 5V el dispositivo está desconectado.
3	V_{in}	Fuente de alimentación de 5V
4	GND	Tensión de tierra lógica

2.2.3 *RedBear Duo*

RedBear Duo [11] es una placa de desarrollo diseñada para la fabricación de productos IoT (*Internet of Things*). Este dispositivo de desarrollo hardware integra un potente microcontrolador ARM (*Advanced RISC Machine*) Cortex M3 con el chip BCM43438 de la empresa *Cypress* (anteriormente pertenecía a la empresa *Broadcom*, pero ésta fue adquirida por *Cypress*). Es capaz de trabajar con conectividad WiFi 802.11b/g/n y con *BLE* 4.1 (*Bluetooth Low Energy*), siendo esta última la característica principal por la cual se eligió este dispositivo, para cumplir las especificaciones del presente Trabajo Fin de Grado (TFG). Ambos comparten una antena de 2.4GHz, pudiendo operar en modo dual. En la Figura 20 se muestra el aspecto general de este dispositivo.

Figura 20. Dispositivo *RedBear Duo*.

Entre las principales características que posee el dispositivo *RedBear Duo* destacan la disponibilidad de 1MB de memoria *Flash SPI* externa y 128KB de SRAM, varios *Light-Emitting Diode* (LED) integrados, 18 entradas/salidas de propósito general, un sistema operativo en tiempo real (FreeRTOS), etc. En la Tabla 2. 11 se muestra la cantidad de interfaces de comunicación, tanto analógicas como digitales, y de ellas, cuales son entrada y/o salidas.

Tabla 2. 11. Periféricos del dispositivo *RedBear Duo*.

Tipo de periférico	Cantidad de pines	Entrada / Salida
Digital	18	Entrada / Salida
Analógico (ADC)	8	Entrada
Analógico (DAC)	2	Salida
SPI	2	Entrada / Salida
I2S	1	Entrada / Salida
I2C	1	Entrada / Salida
CAN	1	Entrada / Salida
High Speed USB	1	Entrada / Salida
PWM	13	Salida

En la Figura 21 se muestran los principales componentes del dispositivo *RedBear Duo* como pueden ser el microcontrolador y el conector micro-USB. En esta placa existen dos botones denominados RESET y SETUP, que se encargan de la configuración de este dispositivo de manera manual, es decir, permiten reiniciar el sistema, cambiar la configuración WiFi u otras configuraciones. También funcionan de manera combinada, haciendo que el dispositivo se restablezca a los valores predeterminados de fábrica. Se puede ver que hay un LED de tipo RGB (*Red, Green, Blue*) que se encarga de proporcionar información sobre el estado del dispositivo, de forma que adquiere un color parpadeante rosa cuando carga un programa al sistema, o un color celeste oscilando para indicar que se encuentra conectado a una red WiFi. El puerto micro-USB tiene dos funciones, la primera es servir de fuente de alimentación al dispositivo, y la segunda es realizas la transferencia de datos entre un *Personal Computer* (PC) y el *RedBear Duo*.

El dispositivo integra un chip de antena (AMPAK A6212A), integrado cerca del microcontrolador, que soporta WiFi 802.11b/g/n y BLE 4.1, dando la posibilidad a comunicaciones vía WiFi, BLE o la conjunción entre ambas. Permite la conexión de una antena externa vía UFL, consiguiendo ampliar el rango de cobertura para el WiFi. Este dispositivo tiene la capacidad de elegir la mejor antena de las que está conectado, excepto si se controla por *firmware*.

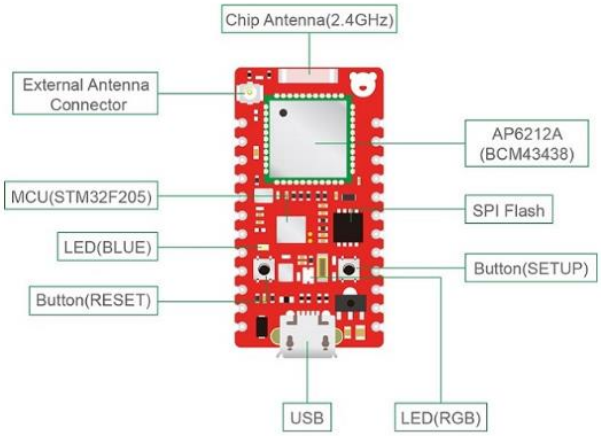

Figura 21. Diagrama de bloques de los principales componentes del *RedBear Duo*.

En la Tabla 2. 12 se recogen los pines de este dispositivo, indicando su principal operatividad, puesto que los diferentes pines pueden ejercer distintas funciones.

Tabla 2. 12. *Pinout* básico del *RedBear Duo*.

Número de Pin	Nombre del Pin	Descripción
1	D0	Entrada / Salida Digital
2	D1	Entrada / Salida Digital
3	D2	Entrada / Salida Digital
4	D3	Entrada / Salida Digital
5	D4	Entrada / Salida Digital
6	D5	Entrada / Salida Digital
7	D6	Entrada / Salida Digital
8	D7	Entrada / Salida Digital
9	GND	Tensión de tierra lógica
10	VBAT	Tensión de entrada de batería externa
11	RST	*Reset* del sistema, reinicia el dispositivo
12	3.3V	Salida de 3.3V dado que todas las señales del dispositivo trabajan a esa tensión
13	VIN	Entrada de alimentación del circuito
14	GND	Tensión de tierra lógica
15	TX	Transmisión en comunicación serie
16	RX	Recepción en comunicación serie
17	WKP	Despertar el dispositivo del modo *25 deep sleep*
18	A6	Entrada / Salida Analógica
19	A5	Entrada / Salida Analógica
20	A4	Entrada / Salida Analógica
21	A3	Entrada / Salida Analógica
22	A2	Entrada / Salida Analógica
23	A1	Entrada / Salida Analógica
24	A0	Entrada / Salida Analógica

En la Figura 22 se muestran gráficamente todas las posibles funciones del *pinout* de este dispositivo.

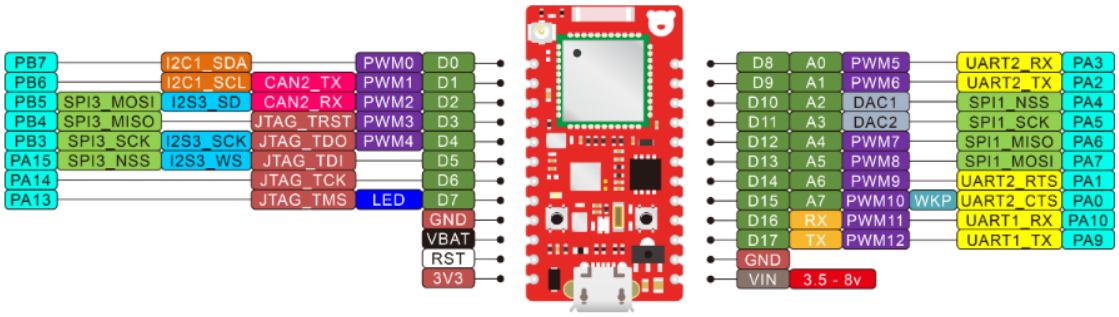

Figura 22. *Pinout* completo del dispositivo RedBear Duo.

A la izquierda del conector micro-USB se encuentra el pin *3.3V*, el cual se encarga de transformar todas las entradas lógicas a ese valor de tensión, dado que esta placa opera internamente con 3.3V. A continuación se encuentra el pin *RST*, utilizado para reiniciar el sistema. El pin *VBAT* ofrece la posibilidad de conectar una batería externa, con el principal objetivo de retener, en caso de pérdida de tensión, a la entrada los datos obtenidos por los registros *Real-Time Clock* (*RTC*) o hacer una copia de seguridad de la memoria *SRAM*, teniendo en cuenta que esta batería no alimenta este dispositivo. Tras este pin, se encuentra un pin de tierra lógica.

Los pines *D0* a *D7* (tener en cuenta que los pines que comiencen con *D* y un número se refiere a *Digital* y el número de entrada y/o salida) son pines de propósito general que pueden actuar como entradas / salidas digitales. Pegado al pin *D7* existe un *LED* azul que está unido a este pin, como se muestra en la Figura 22. Aparte, los pines *D0 a D4* pueden funcionar también como salidas analógicas usando técnicas *Pulse-Width Modulation* (PWM).

Los pines *A0* a *A6* (tener en cuenta que los pines que comiencen con *A* y un número se refiere a *Analog* y el número de entrada y/o salida) representan entradas analógicas que operan con tensiones entre 0 y 3.3V. Estos pines también se pueden utilizar como entradas o salidas digitales. Algunos de estos pines (*A0, A1, A4, A5 y A6*) se corresponden también con salidas analógicas utilizando las técnicas PWM. Los pines *A2* y *A3* se pueden usar como salidas de *Digital Analog Converter* (DAC), o salidas analógicas capaces de soportar entre 0 y 3.3V. Cuando el dispositivo *RedBear Duo* se encuentre configurado en el modo *deep sleep, el pin A6* puede despertar al dispositivo, aparte de poder funcionar como entrada analógica *(A7*), entrada / salida digital (*D15*) y como salida analógica PWM.

Los siguientes pines son *TX* (transmisión) y *RX* (recepción), que se utilizan para la comunicación serie del dispositivo *RedBear Duo*. Por último, situados por debajo de estos pines se encuentran, un pin *GND*, y el pin *VIN*. Se puede alimentar el dispositivo

RedBear Duo suministrando entre 3.5V y 8V al pin *VIN*, como alternativa al uso del puerto USB. Esta no es la única función que puede aceptar este pin, pues si no se usa para suministrar tensión al dispositivo, se puede usar como un pin de salida de tensión con un valor de 5V.

Las dimensiones del dispositivo *RedBear Duo* vienen expresadas en milímetros y son las que se muestran en la Figura 23.

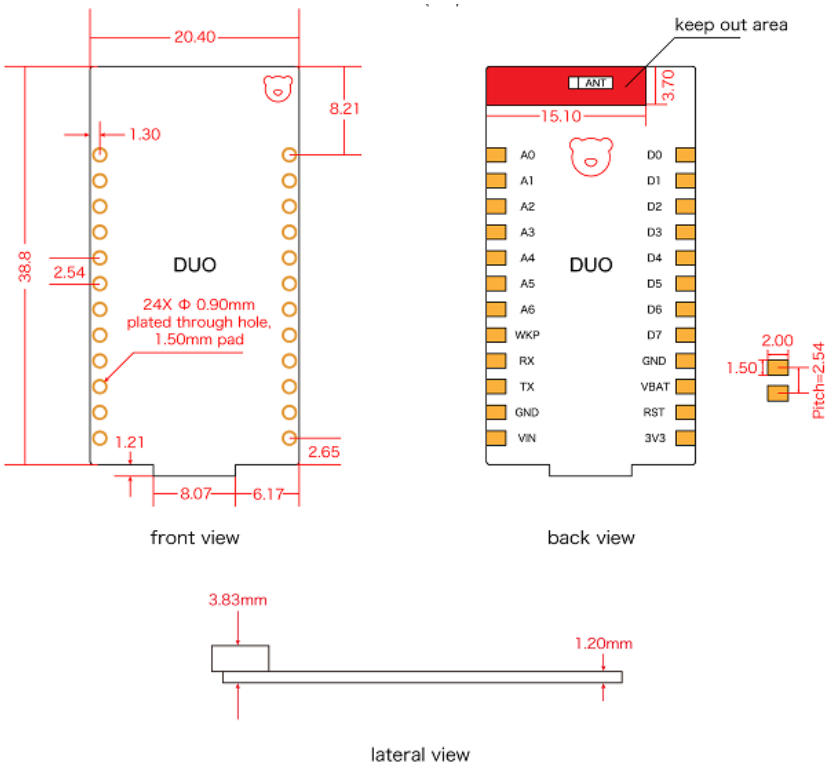

Figura 23.Dimensiones del RedBear Duo.

2.2.4 Estudio del conversor DC-DC de 3.3V a 5V

La utilización de diferentes dispositivos que operan con diferentes tensiones hace que se tenga que estudiar y añadir al circuito un conversor de la empresa *SparkFun* [12]. Este dispositivo implementa una lógica de desplazamiento de nivel bidireccional entre 3.3V y 5V. En las siguientes Figura 24 y Figura 25 se puede ver este conversor.

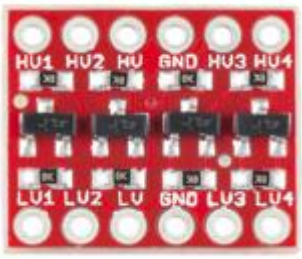

Figura 24. *Top layer* del conversor DC-DC de *SparkFun*.

Figura 25. *Bottom layer* del conversor DC-DC de *SparkFun*.

Si se revisa su esquemático, este conversor *Bi-Directional Logic Level Converter* (a partir de ahora *BD-LLC)* es muy sencillo, como se muestra en la Figura 26. Se puede ver que cada canal está compuesto por un *MOSFET* y dos resistencias que utilizan la tecnología *push-pull* para realizar la conversión de tensión. Es necesario tener en cuenta que si la tensión proporcionada en uno de los canales es 0V, a la salida del canal se obtendrán 0V.

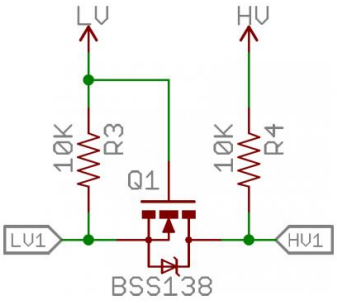

Figura 26. Esquemático de uno de los canales del conversor de *Sparkfun*.

EL *BD-LLC* está compuesto por 12 pines, divididos en 2 cabeceras de 6 pines. Por un lado se encuentran los 6 pines de alto voltaje, y por el otro los 6 pines de bajo voltaje. Estos pines comprenden 4 canales de datos bidireccionales, 2 entradas de tensión (una a *LV* (*Low Voltage*) y otra a *HV* (*High Voltage*)) y 2 pines de *GND*, como se muestra en la Figura 27 y en la Tabla 2. 13 . Los canales están etiquetados como *HV1*, *LV1*, *HV2*, *LV2*, *HV3*, *LV3*, *HV4* y *LV4*, siendo el número final de la etiqueta el canal utilizado. La entrada de tensión etiquetada como *LV* asumirá 3.3V, al igual que en la *HV* se proporcionarán 5V. Es importante que se tenga en cuenta que estas entradas/salidas son puramente digitales y que este dispositivo es incapaz de soportar tensiones analógicas.

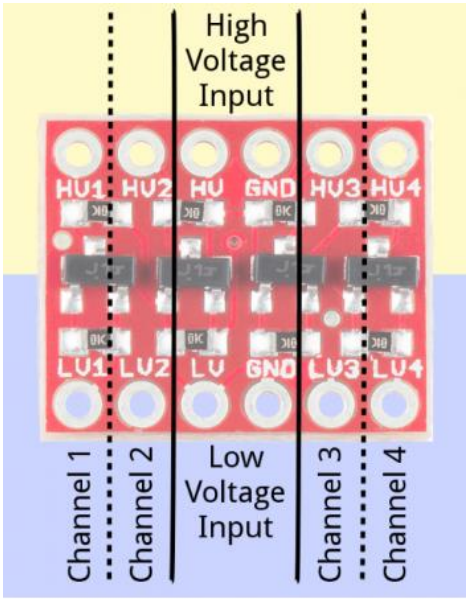

Figura 27. Pines y canales de datos del conversor de Sparkfun.

Tabla 2. 13. Pinout de *BD-LLC*.

Nombre del Pin	Descripción
HV1	Entrada/Salida de datos a 5V en el canal 1
LV1	Entrada/Salida de datos a 3.3V en el canal 1
HV2	Entrada/Salida de datos a 5V en el canal 2
LV2	Entrada/Salida de datos a 3.3V en el canal 2
HV	Entrada a nivel alto de 5V
LV	Entrada a nivel bajo de 3.3V
GND	Tensión de tierra lógica
GND	Tensión de tierra lógica
HV3	Entrada/Salida de datos a 5V en el canal 3
LV3	Entrada/Salida de datos a 3.3V en el canal 3
HV4	Entrada/Salida de datos a 5V en el canal 4
LV4	Entrada/Salida de datos a 3.3V en el canal 4

2.2.5 Cable FFC de 10 pines y 0.5mm de pin

El cable plano flexible [13] mostrado en la Figura 29 y en la Figura 29 es un circuito basado en puentes de conexión punto a punto. Se suelen encontrar en conexiones a pantallas LCD, pantallas táctiles, placas base, etc.

Figura 28. FFC de 10 pines y 0.5mm de longitud de pin.

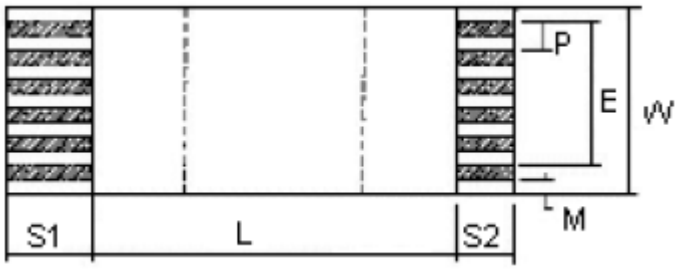

Figura 29. *Footprint* del cable FFC de 10 pines y 0.5mm de pin.

Este cable es capaz de soportar hasta 80ºC de temperatura y 60V de tensión trabajando a máximo rendimiento. Aun así, se podría utilizar con valores superiores, pero mermando su velocidad y con posibilidades de afectar a dicha conexión. El resto de sus características generales se pueden encontrar en la Tabla 2. 14, teniendo en cuenta que el cable analizado es el perteneciente a la primera columna de datos, y que todas las dimensiones recogidas en esta tabla están especificadas en milímetros.

Tabla 2. 14. Características generales del FFC.

Pitch	**: P**	0.5	1	1.25
Type		A*		
Number of Conductor (N) **Type A (min. / max.)**		10 to 50	10 to 30	4 to 40
Cable width	**: W**	(N+1) 0.5	(N+1) 1.0	(N+1) 1.25
Margin	**: M**	0.35	0.65	0.85
Conductor width	**: C**	0.30	0.70	0.80
Strip length	**: S**	4		
Insulated length	**: L**	202		
Conductor thickness **Flexible**	**: F (Type A)**	0.05		

*Los extremos del cable se encuentran sin de protección y con una cinta de refuerzo.

Se trata de un cable de cobre estañado plano, protegido con poliéster. Lleva incorporado una cinta de refuerzo de poliéster estándar para la terminación de los conectores, como se muestra en la Figura 30, asegurando una inserción fácil y precisa en el conector.

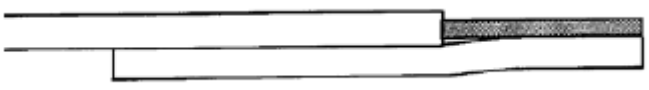

Figura 30. Enfoque ampliado de un extremo del cable.

Al analizar este tipo de cable es importante la resistencia del conductor medida en ohmios/metros (Ω/m), utilizado para caracterizar el aislamiento de los cables planos. Las características ya mencionadas van en correlación con una adecuada protección electromagnética, buena flexibilidad, peso ligero y coste razonable. En la Figura 31 se puede ver que esto último se ha conseguido con láminas de 10µm de aluminio envuelto

en el cable plano consiguiendo una cobertura de 360º. La curva descrita por la trasferencia de impedancia se representa en la Figura 32.

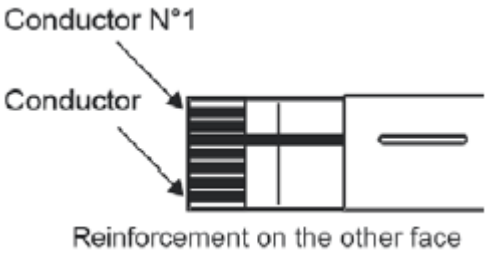

Figura 31. Muestra del recubrimiento de láminas aluminio.

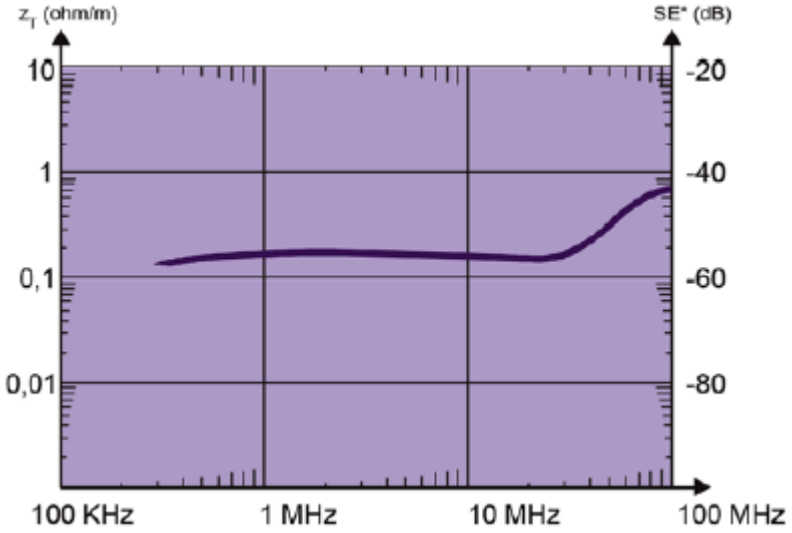

Figura 32. Curva de transferencia de impedancia.

2.3 Implementación de la plataforma *Hardware* (*HW*) basada en la célula *modul P16* de *metec*

Una vez analizados en el apartado anterior todos los componentes utilizados en el presente TFG, en la Figura 33 se muestra un esquemático inicial para el montaje del circuito basado en la célula *modul P16* de *metec* para la representación de código Braille.

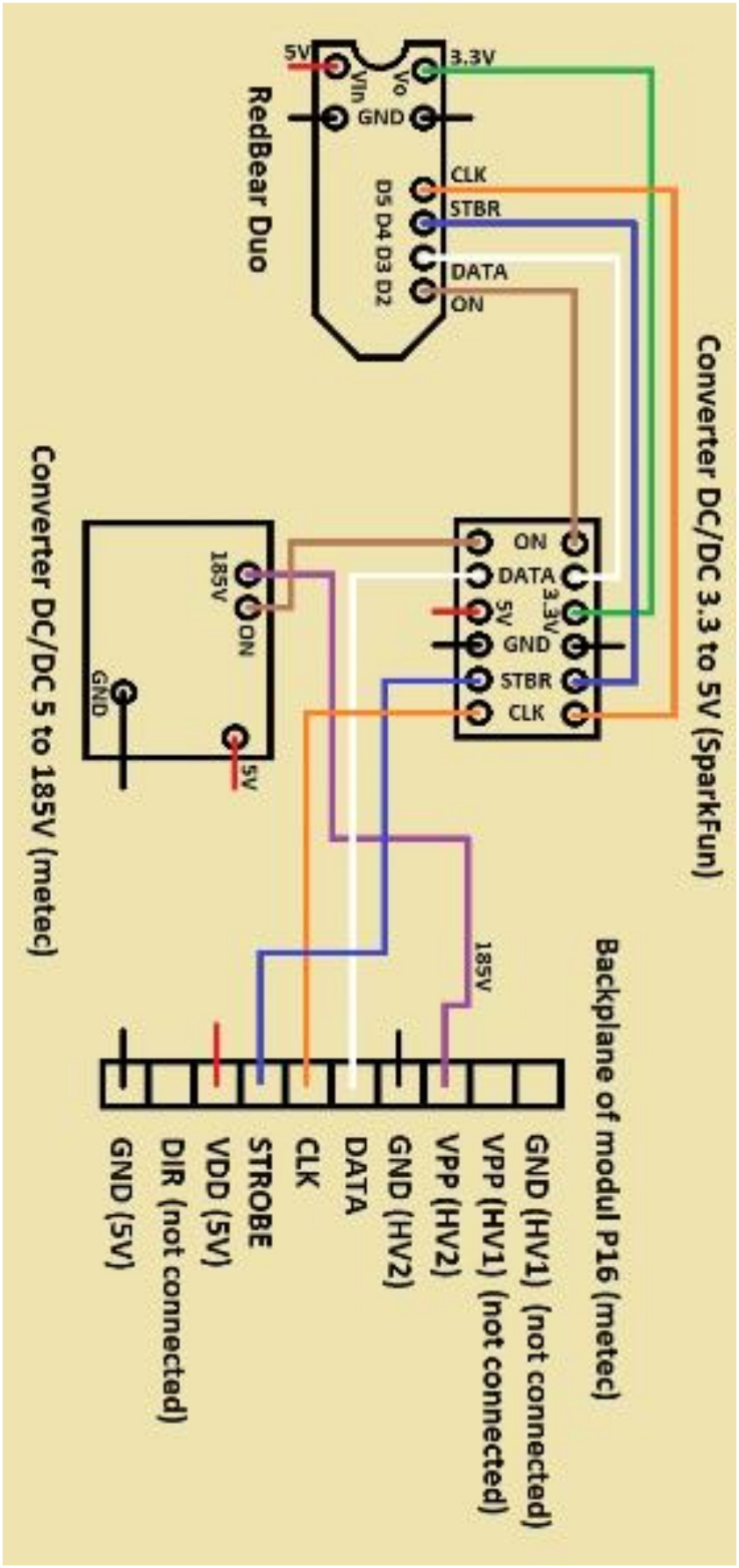

Figura 33.Esquemático inicial del montaje y conexionado de la célula Braille *modul P16.*

Antes de realizar el montaje de la plataforma *HW/SW* basada en la célula *modul P16,* hay que tener en cuenta varios aspectos, como que el *pitch* de los pines del *backplane* no son estándares a 1mm, sino que son de 0.5mm, con lo cual se necesitó utilizar un cable plano *FFC* con el que conectar al *backplane* con un adaptador de los pines de 0.5mm a 1mm. En el circuito, la tensión de alimentación de la célula *modul P16* es de 185V, y el cable utilizado no tolera esa tensión, pero como la corriente es muy pequeña, se consigue el funcionamiento deseado sin llegar a un deterioro irreversible. Por otro lado, las señales de salida del dispositivo *RedBear Duo* se corresponden con un nivel lógico de 3.3V, pero la célula *modul P16* soporta señales de 5V, por lo que es necesario utilizar un conversor CC-CC de 3.3V a 5V. También fue necesario integrar el conversor CC-CC de 5V a 185V, porque la célula necesita una tensión de alimentación de 200V ±10% de tensión para funcionar correctamente.

Revisando uno a uno los componentes y sus hojas de características, se descubre que el *backplane* de esta célula precisa que se conecten todas las entradas. Así, analizando detenidamente los datasheet de los componentes de la empresa metec, más concretamente las hojas de especificaciones del chip que contiene el *backplane*, se determina que en el pin *DIR*, cuando el *backplane* es a izquierda como el que se utiliza en este TFG, ha de ser conectado a *GND*. Por ello, se modifica el esquemático inicial, dando como resultado la Figura 34.

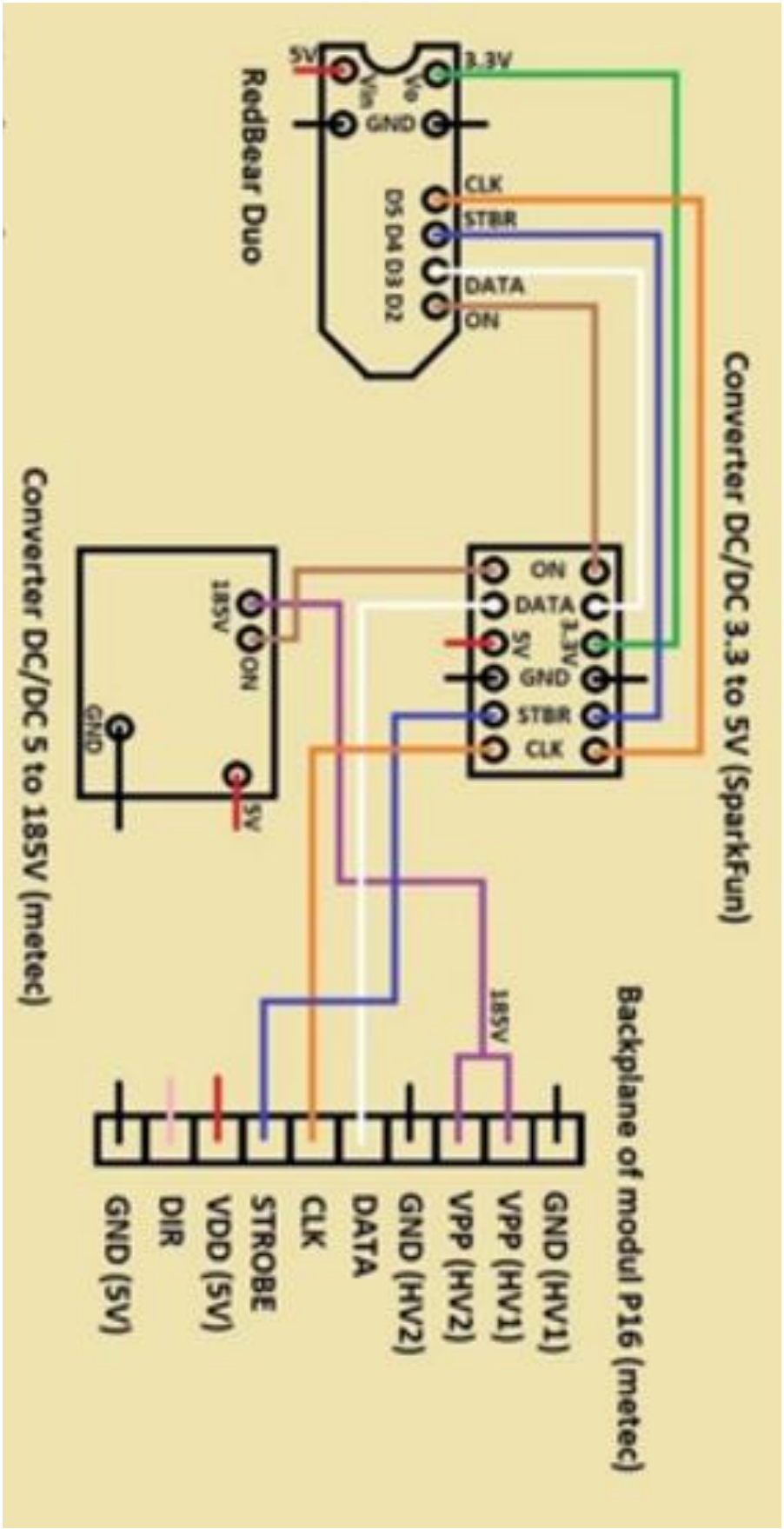

Figura 34. Esquemático del montaje y conexionado realizado para la célula Braille *modul P16*.

En el esquemático de la plataforma basada en la célula de lectura en código Braille *modul P16* de *metec,* mostrada en la Figura 35, se puede ver el código de colores utilizado para las diferentes señales. Una manera de entender completamente todo el funcionamiento de este circuito es conocer todas las señales implicadas.

- *GND*: se trata de tierra lógica. En el circuito hay varios nodos que se conectan este punto para proteger distintas partes del mismo.
- La tensión de alimentación de 3.3V es necesaria para alimentar el conversor DC/DC de 3.3V a 5V.
- La tensión de alimentación de 5V, proveniente del pin *Vin* del *RedBear Duo*, es necesaria para alimentar los dos conversores *DC/DC* y el *blackplane* de la célula.
- La señal de reloj *Clock* (*CLK*). Esta señal es necesaria para que con cada flanco de subida de la señal de reloj introduzca el valor de un bit, coincidiendo con un punto de la célula.
- La señal *Strobe* (*STBR*). Esta señal será la encargada de actualizar la célula al completo, tras el número de ciclos de la señal de reloj *CLK* necesarios para cargar los nuevos valores que adquirirán los puntos de ésta.
- La señal *Data* (*DATA*). Esta señal contendrá cada uno de los símbolos del texto traducido a código Braille para su reproducción en la célula.
- La señal *On* (*ON*). Esta señal tiene que estar siempre a nivel bajo (0) si se desea que el conversor *DC/DC* de 5v a 185V esté activo, y como consecuencia funcionen el *backplane* y la célula *modul P16*. Podría haberse conectado directamente a *GND*, pero se decidió hacer por software.
- La alimentación de 185V que ofrece el conversor *DC/DC* de 5V a 185V es utilizada por dos entradas en el *backplane* para alimentar la célula de lectura Braille de *metec*.

En la Figura 35 se muestra el circuito final de representación de código Braille, basado en una implementación *HW* con la célula *modul P16* de *metec*, montado sobre *protoboard*.

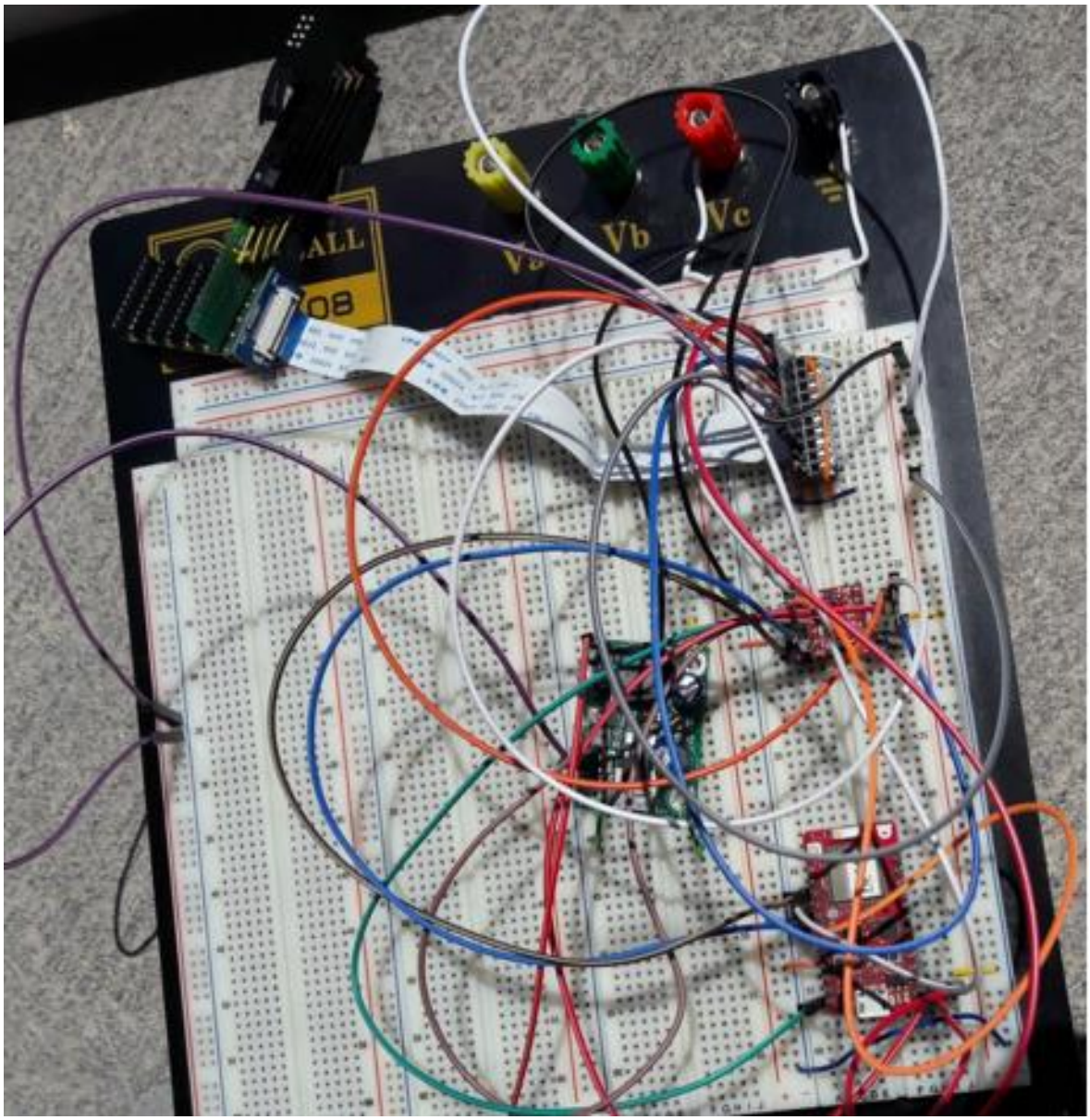

Figura 35. Montaje sobre *protoboard* de la célula Braille *modul P16* de *metec.*

En la Figura 36 se muestra el mismo circuito mostrado en la imagen anterior, pero identificando cada componente para mayor comprensión del mismo.

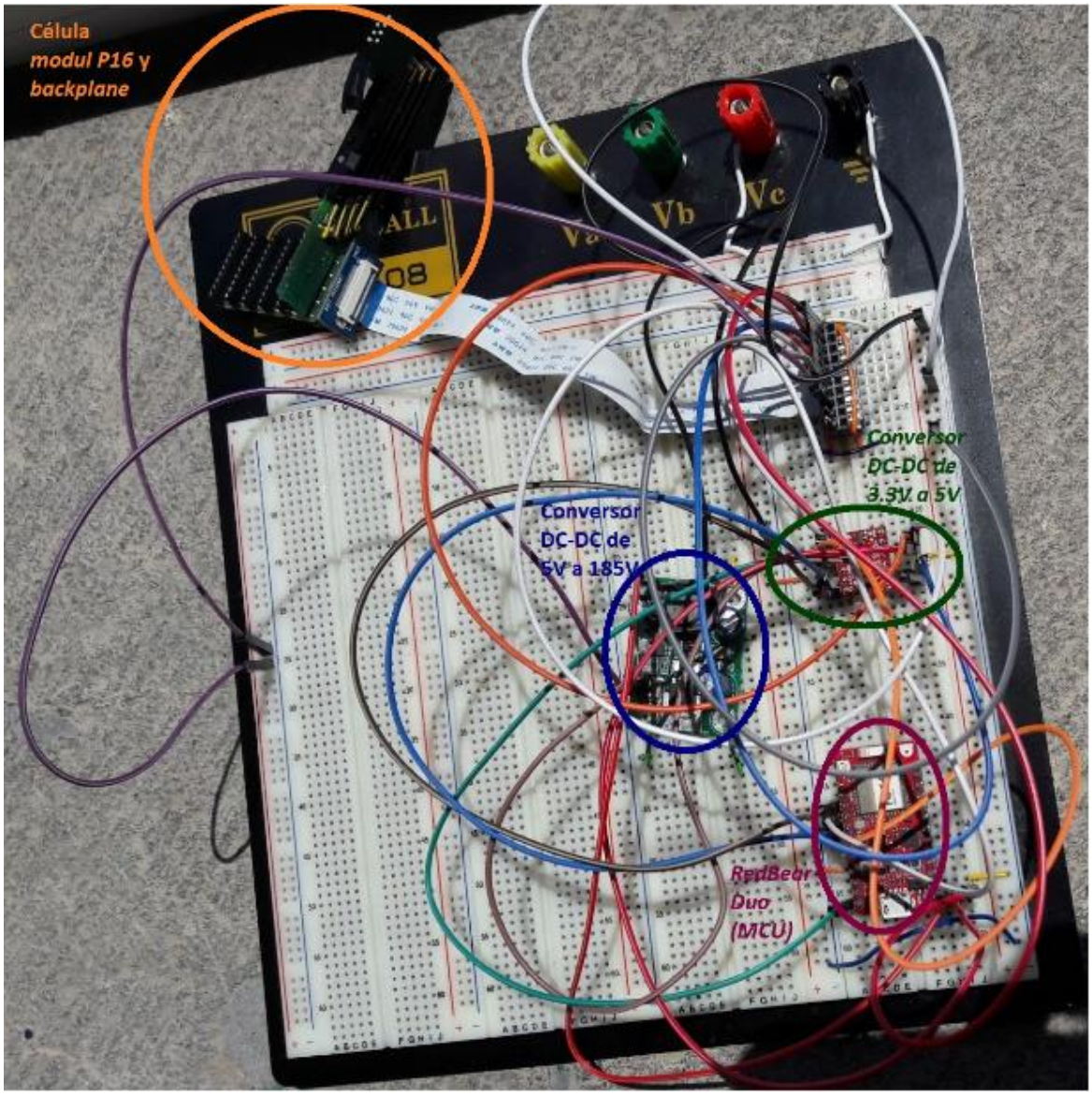

Figura 36. Montaje sobre *protoboard* de la célula Braille *modul P16* de *metec*, con indicación de los componentes utilizados.

Capítulo 3. Introducción a *Bluetooth Low Energy* (*BLE*)

3.1 Introducción

Bluetooth Low Energy (*BLE*) es la especificación 4.0 [14], 4.1 y 4.2 de la tecnología Bluetooth desarrollado por *Bluetooth Special Interest Group* (*SIG*), anteriormente denominada *Bluetooth Smart*. Se ha diseñado como una tecnología complementaria a *Bluetooth* clásico para garantizar un consumo de energía bajo, y menor tiempo de establecimiento de conexión. A pesar del uso de la misma banda de frecuencia y las similitudes compartidas, *BLE* debe considerarse un nuevo estándar con objetivos y aplicaciones diferentes.

BLE está diseñado para la transmisión de pequeñas cantidades de datos (tiempos de transmisión muy pequeños) y por lo tanto de ultra-bajo consumo de energía. No está diseñado para mantener una conexión de larga duración entre dispositivos transmitiendo grandes cantidades de datos a alta velocidad. Esto permite que los dispositivos estén activos solo cuando se les pide la transmisión de datos. Con esta idea surgen muchas aplicaciones de *IoT* (*Internet of Things*).

3.2 Arquitectura

La topología de red es de tipo estrella en *BLE*. Los dispositivos *Master* pueden tener varias conexiones con dispositivos periféricos o *Slaves* y simultáneamente realizar búsquedas de otros dispositivos. Un dispositivo con el rol de *Slave* solo puede tener una conexión de capa de enlace con un único *Master*. Además, un dispositivo puede enviar datos en modo *Broadcast*, eventos de *Advertising*, sin esperar ninguna conexión; esto permite enviar datos a los dispositivos en estado *Scanning* sin necesidad de establecer la conexión *Master-Slave*. En consecuencia, un dispositivo *BLE* puede comunicarse de acuerdo a dos modos de operación:

- *Broadcasting*: Se envían datos en el rango de escucha a cualquier dispositivo receptor. Este mecanismo permite transmitir datos en un solo sentido a cualquier dispositivo capaz de recibir los datos emitidos. En esta topología se definen dos roles distintos:
    - *Broadcaster.* Envía paquetes de *Advertising* no conectables periódicamente a cualquier dispositivo *BLE*.

- *Observer*. Escanea constantemente las frecuencias pre-configuradas para la recepción de algún paquete *Advertising* no conectable que se esté emitiendo.

En la Figura 37 se muestra lo explicado en este párrafo.

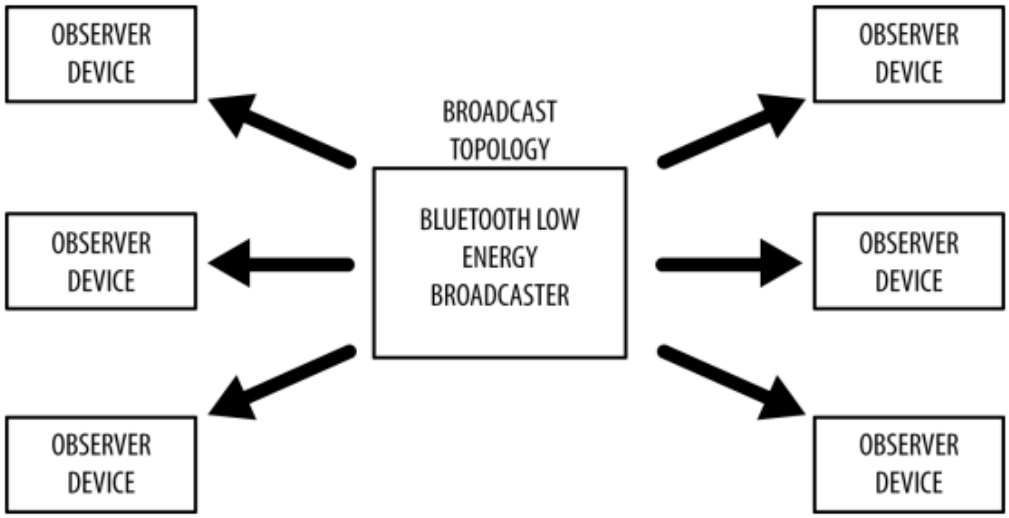

Figura 37. Topología en *Broadcast.*

- *Connections*: Una conexión es un intercambio permanente y periódico de paquetes de datos entre dos dispositivos. Se trata de una conexión privada entre los dispositivos implicados. Existen dos roles:
  - *Central*: Explora repetidamente las frecuencias pre-establecidas para recibir paquetes de *Advertising* conectables y, cuando corresponde, inicia una conexión. Una vez que se establece la conexión, el dispositivo *Central* administra el *timing* e inicia los intercambios periódicos de datos.
  - *Peripheral*: Envía paquetes de *Advertising* conectables periódicamente y acepta conexiones entrantes. Una vez establecida una conexión, el dispositivo *Peripheral* sigue la sincronización del dispositivo *Central* e intercambia datos regularmente con él.

  El modo de inicio de una conexión *BLE* entre dispositivos comienza con un dispositivo *Central* recibiendo, en primer lugar, paquetes *Advertising* conectables, y a continuación, enviando una solicitud de conexión al dispositivo *Peripheral*. Una vez que se establece la conexión, y conociendo la exclusividad que presenta esta comunicación, el dispositivo *Peripheral* deja de enviar paquetes de *Advertising* y los dos dispositivos pueden comenzar a intercambiar paquetes de datos en ambas direcciones, como se muestra esquemáticamente en la Figura 38.

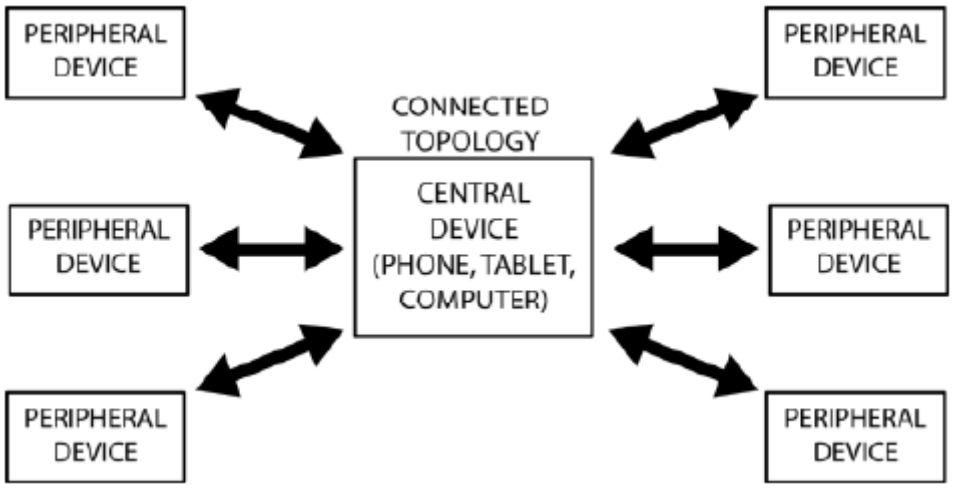

Figura 38. Topología en *Connection*.

3.3 Perfiles

Los Perfiles definen cómo deben usarse los Protocolos para alcanzar un objetivo particular, ya sea genérico o específico. Mientras, los protocolos son las capas que implementan los diferentes formatos de paquete, encaminamiento, multiplexación, codificación y decodificación, que permiten que los datos se envíen de manera efectiva entre dispositivos.

De acuerdo con las especificaciones de SIG, los Perfiles Genéricos definidos en BLE son los siguientes:

- Perfil de Acceso Genérico (*GAP*, *Generic Access Profile*): Define los roles, procedimientos y modos que permiten a los dispositivos transmitir datos, descubrir dispositivos, establecer conexiones, administrar conexiones, y negociar niveles de seguridad. *GAP* es, en esencia, la capa de control superior de *BLE*. Este perfil es obligatorio para todos los dispositivos *BLE*, y todos deben cumplirlo.
- Perfil de Atributo Genérico (*GATT*, *Generic Attribute Profile*): Al tratar con el intercambio de datos en *BLE*, *GATT* define un modelo básico de datos y procedimientos para permitir a los dispositivos descubrir, leer, escribir y enviar elementos de datos entre ellos. *GATT* es, en esencia, la capa de datos superior de *BLE*.

En cuanto a los Perfiles Específicos, *SIG* proporciona un conjunto predefinido de Perfiles de Casos de Uso, basados en *GATT*, que cubren por completo todos los procedimientos y formatos de datos necesarios para implementar una amplia gama de casos específicos, como por ejemplo: *Find Me Profile* (permite que los dispositivos localicen físicamente otros dispositivos), *Proximity Profile* (detecta la presencia o ausencia de dispositivos cercanos),

Glucose Profile (transfiere de forma segura los niveles de glucosa sobre *BLE*), o *Health Thermometer Profile* (transfiere lecturas de temperatura corporal sobre *BLE*), entre otros.

3.4 Protocolos

Para la gestión de los dispositivos, la conexión y la interfaz de las aplicaciones, el *SIG* de *Bluetooth* define una pila de protocolos. La pila del protocolo *BLE* se divide en tres partes básicas: *Controller, Host* y *Applications*, como se identifican en la Figura 39. Cada uno de estos bloques básicos de la pila de protocolos se divide en varias capas que proporcionan la funcionalidad necesaria para operar:

- *Controller*. El *Controller* es el dispositivo físico que permite transmitir y recibir señales radio e interpretarlas como paquetes con información. Contiene:
    - Interfaz de Control de *Host* (*HCI, Host Controller Interface*), lado del Controlador.
    - Capa de Enlace (*LL, Link Layer*).
    - Capa Física (*PHY, PHYsical Layer*).
- *Host*. El *Host* es la pila de software que gestiona cómo dos o más dispositivos se comunican entre ellos. No está definida ninguna interfaz superior para el *Host*, cada sistema operativo o entorno tiene su propia manera de exponer *API's* (*Application Programming Interface*) de *Host* para los desarrolladores. Esta parte de la pila de Protocolos está formada por:
    - Perfil de Acceso Genérico (*GAP, Generic Access Profile*).
    - Perfil de Atributo Genérico (*GATT, Generic Attribute Profile*).
    - Protocolo de Control y Adaptación de Enlace Lógico (*L2CAP, Logical Link Control and Adaptation Protocol*).
    - Protocolo de Atributos (*ATT, ATTribute protocol*).
    - Administrador de Seguridad (*SMP, Security Manager Protocol*).
    - Interfaz de Controlador de *Host (HCI, Host Controller Interface*), lado del *Host.*
- *Application*. La capa *Application* es la más alta de la jerarquía y es responsable de contener la lógica, la interfaz de usuario, y la gestión de datos relacionados con el caso de uso actual que implementa la aplicación. La arquitectura de una aplicación depende en gran medida de cada implementación particular.

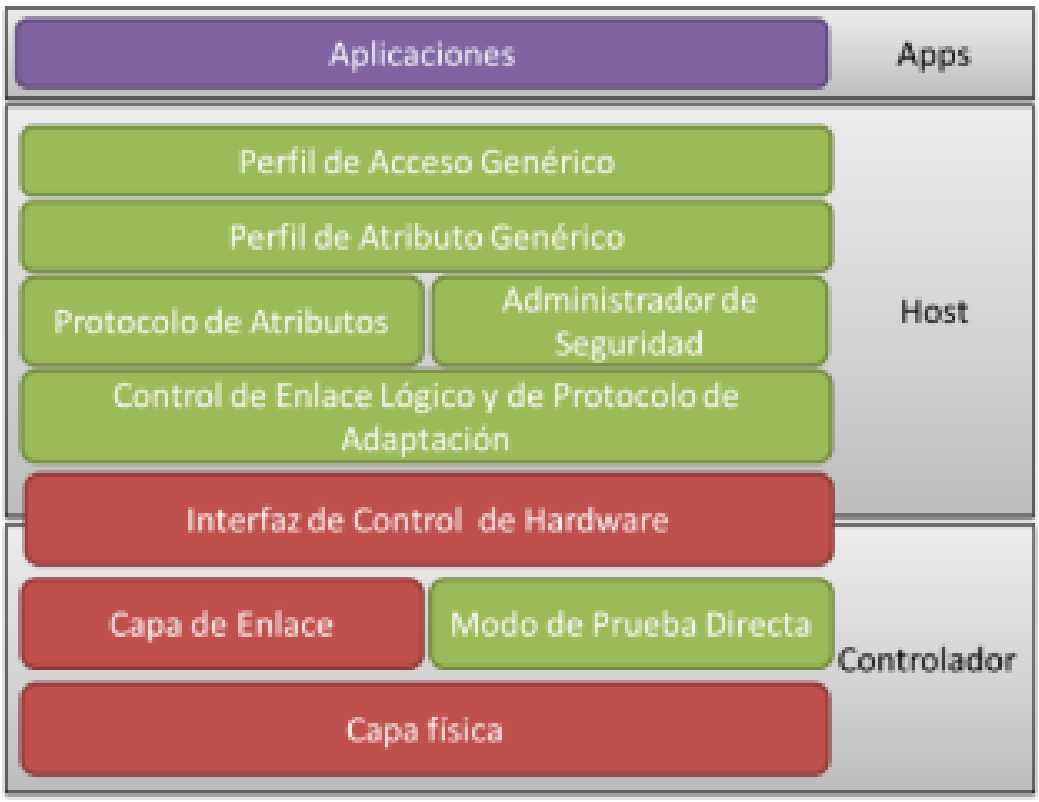

Figura 39. Pila de protocolos de *BLE*.

3.4.1 Capa Física (*PHY*)

Bluetooth Low Energy comparte algunas similitudes con *Bluetooth* clásico. Los dos usan la banda de 2.4 GHz. *Bluetooth* clásico y *BLE* usan la modulación *GFSK* (*Gaussian Frequency Shift Keying)* a 1Mbps, pero con índices de modulación diferentes. El estándar Bluetooth clásico tiene 79 canales mientras *BLE* tiene 40 tal como indica la Tabla 3. 1. La separación entre canales también es diferente. Debido a estas dos diferencias entre *BLE* y *Bluetooth* clásico, estos son incompatibles entre sí, por lo tanto no se pueden comunicar. Sin embargo, existen dispositivos *Dual Mode* que soportan las dos tecnologías conmutando los parámetros de modulación y los canales donde se está radiando.

Tabla 3. 1. Comparativa entre *BLE* y *Bluetooth Classic*.

	BLE	*Bluetooth Classic*
Modulación	GFSK 0.45 to 0.55	GFSK 0.28 to 0.35
Tasa de transferencia (Mbps)	1 Mbps	1 Mbps
Nº de canales	40	79
Separación (MHz)	2 MHz	1 MHz

La capa física (*PHY*, *PHYsical Layer*) es la parte que contiene el circuito de comunicaciones analógicas, capaz de modular y demodular señales analógicas y transformarlas en símbolos digitales. También se encarga de transmitir las señales al aire, usando ondas radio en la banda de frecuencia *Industrial Scientific Medical* (*ISM*) 2.4 Ghz que se extiende desde 2402 MHz hasta 2480 MHz. La separación entre los 40 canales utilizados es de 2 MHz (numerados de 0 a 39 y de 1 MHz de anchura cada uno). Como se muestra en la Figura 40, existen 3 canales dedicados para el *Advertising* (37, 38 y 39) y los restantes 37 son para el intercambio de datos durante la conexión. Estos tres canales de *Advertising* se encuentran situados estratégicamente para evitar interferencias causadas por otras tecnologías que coexisten en el mismo espectro (IEEE 802, *Zigbee*,…).

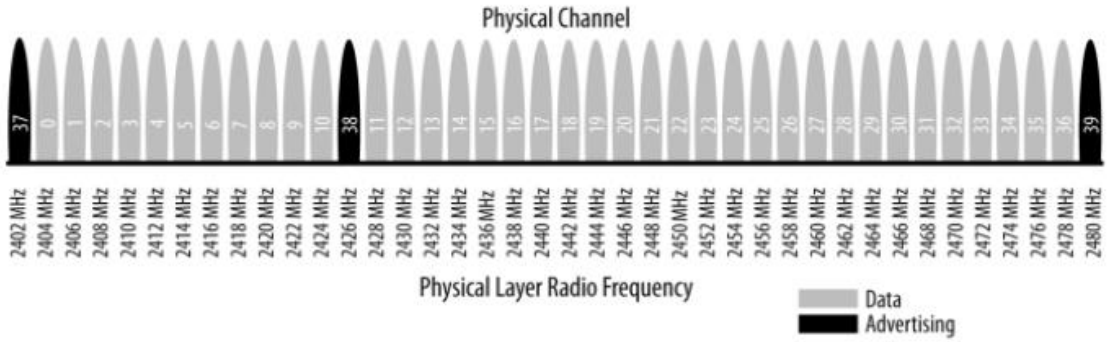

Figura 40. Distribución de canales de BLE.

El estándar *BLE* emplea la técnica denominada *Frequency Hopping Spread Spectrum* (FHSS), o espectro ensanchado por salto de frecuencia, donde se salta entre canales en cada evento de conexión usando la fórmula:

$$canal = (canal\ actual + salto)\ mod\ 37$$

El valor del salto se comunica cuando se establece la conexión y, por lo tanto, es diferente para cada nueva conexión establecida. Esta técnica contribuye a minimizar el efecto de cualquier interferencia de radio presente en la banda de 2.4 GHz, en cualquier canal.

3.4.2 Capa de Enlace (*LL*)

La capa de Enlace (*LL, Link Layer*) es la parte que interactúa directamente con la capa *PHY*, y es responsable de controlar, negociar y establecer los enlaces, seleccionar las frecuencias para la transmisión de datos, admitir diferentes topologías, y dar soporte a diversas formas de intercambio de datos. En esencia, esta capa es la responsable de los procesos de *Advertising* y *Scanning*, así como de la creación y mantenimiento de las conexiones y de la estructura de los paquetes de datos transmitidos. En la Figura

41 se muestra el funcionamiento de la capa de Enlace, el cual puede describirse como una máquina de estados. Sus diferentes estados son:

- *Standby*. Se trata de un estado predeterminado de la capa de Enlace. En este estado, el dispositivo ni transmite ni recibe paquetes. Un dispositivo puede entrar en este estado desde cualquier otro estado. Por lo general, este estado está asociado con un sistema durmiente, para conservación de energía.
- *Advertising*. La capa de Enlace de los dispositivos de tipo *Peripheral* entra en este estado en el que envía paquetes de *Advertising* en los canales de *Advertising*. En este estado también escucha cualquier respuesta (solicitud) de los paquetes desde los dispositivos *Central*. A este estado se puede ingresar desde el estado *Standby* cuando la capa de Enlace decide iniciar el proceso de *Advertising*. Este modo es crítico si se analiza desde el punto de vista de la potencia ya que el tiempo de transmisión afecta al consumo de energía, por tanto el intervalo de *Advertising* afecta directamente al consumo de potencia y la vida de las baterías.
- *Scanning*. La capa de Enlace del dispositivo *Central* escucha los paquetes de *Advertising* enviados por el dispositivo *Advertiser* (se denomina así al dispositivo *BLE* que utiliza los canales de *Advertising* para anunciar que es conectable y detectable, o que puede ser descubierto) a través de los canales asignados, pudiendo solicitarle que proporcione información adicional con el fin de explorar los dispositivos *BLE* existentes. A este estado se puede entrar desde el estado *Standby* cuando la capa de Enlace decide comenzar el proceso de *Scanning*. A un dispositivo en estado de *Scanning* se le denomina *Scanner*.
- *Initiating*. El dispositivo *Central* escucha los *Advertising* de periféricos y responde a ellos iniciando una conexión. En general, este es el estado en el que entra el dispositivo *Central* antes de pasar al estado de conexión. A este estado se puede pasar desde el estado *Standby* cuando el dispositivo *Scanner* decide iniciar una conexión con el dispositivo *Advertiser*, actuando como dispositivo *Initiator*.
- *Connection*: El dispositivo *Central* está conectado a otro dispositivo *Peripheral*, definiéndose los roles de *Master* y *Slave*. A este estado se puede pasar desde el estado *Initiating*, o desde el estado *Advertising*. Cuando se ingresa desde el estado *Initiating*, el dispositivo actúa como *Master*, mientras que cuando se entra desde el estado *Advertising*, el dispositivo actúa como *Slave* en el intercambio periódico de datos a través de eventos de conexión. Los canales de datos se utilizan una vez establecida la conexión entre ambos dispositivos.

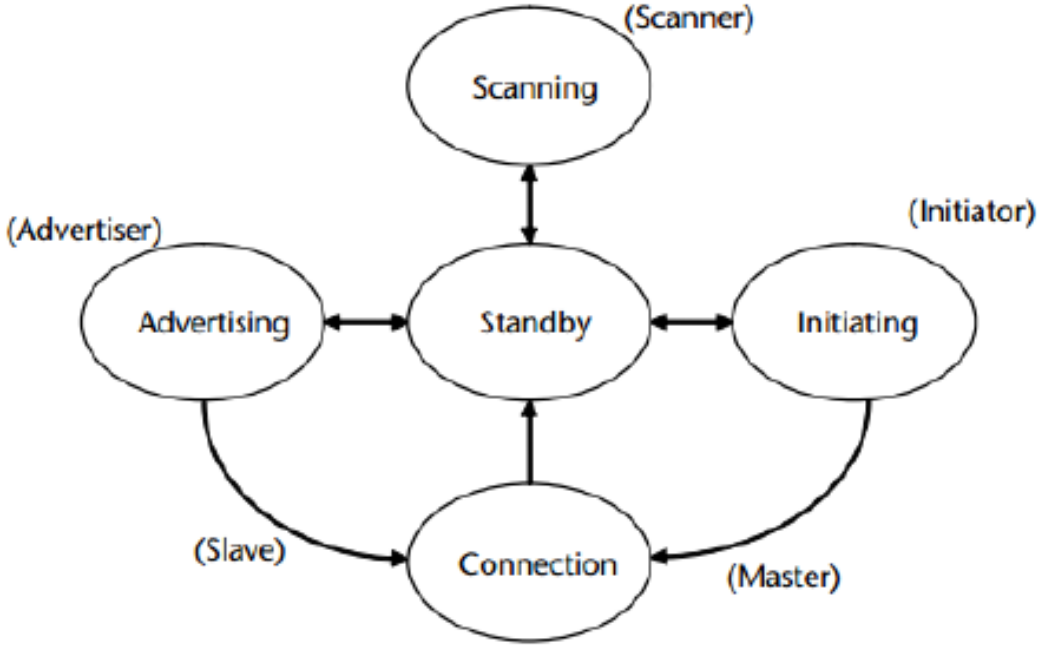

Figura 41. Máquina de estados de la capa de Enlace en *BLE*.

Para el *Slave*, el estado de *Advertising* también se considera como estado inicial antes del estado de conexión. El estado de conexión es el último estado en el que el *Slave* (*Peripheral*) y el *Maste*r (*Central*) pueden intercambiar datos. Éstos se intercambian datos periódicamente a través de eventos de conexión.

El dispositivo *Scanner* busca en los canales de *Advertising* paquetes de *Advertising* de otros dispositivos. Existen cuatro tipos de *Advertising*:

- *Connectable Undirected Advertising*: es el más común. El dispositivo *Scanner* puede recibir los anuncios o ir a hacía una conexión.
- *Connectable Directed Advertising*: se usa cuando un dispositivo necesita conectarse de manera rápida. Este *Advertising* debe repetirse cada 3.75 milisegundos. Se permite estar en este estado un tiempo máximo de 1.28 segundos. En la especificación 4.2 se introducen dos tipos de *Advertising* derivados del *Directed Advertising*: *High Duty Cycle Directed Advertising* y *Low Duty Cycle Directed Advertising*.
- *Nonconnectable Advertising*: es usado por dispositivos que quieren difundir datos. El dispositivo no puede ser detectable (no puede recibir *Scan Request*) ni en conexión.
- *Discoverable Advertising*: no se puede usar para iniciar una conexión, pero si para ser escaneado por otros dispositivos y detectar *Scan Request*. El dispositivo escáner puede obtener datos del *Advertiser* ya que éste responde a cada *Scan Request* detectado con un *Scan Response*. También se puede utilizar para emitir datos.

3.4.3 Interfaz de control de *Host* (*HCI*)

La Interfaz de Control de *Host* (*HCI*, *Host Controller Interface*) proporciona un método estándar de comunicación entre las capas superior e inferior de la pila de protocolos. En muchas implementaciones, las capas superiores generalmente residen en un *Host* y las capas inferiores residen en un chip Controlador de *Bluetooth* separado. La interfaz *HCI* proporciona un mecanismo de comunicación entre el *Host* y el Controlador *Bluetooth* vía comandos *HCI.* Algunos comandos *HCI* están relacionados con la configuración y la actualización de los parámetros de conexión.

3.4.4 Protocolo de Control y Adaptación de Enlace Lógico (*L2CAP)*

El Protocolo de Control y Adaptación de Enlace Lógico (*L2CAP*, *Logical Link Control and Adaptation Protocol*) sirve como un multiplexor de protocolos, tomando múltiples protocolos de las capas superiores y encapsulándolos en el formato de paquete *BLE* estándar (y viceversa). También realiza el proceso de fragmentación y recombinación, mediante el cual toma paquetes de elevado tamaño de las capas superiores y los divide en fragmentos en el lado de transmisión. En el lado de recepción, recibe múltiples paquetes que se han fragmentado, y los recombina en un solo paquete que luego se enviará en sentido ascendente a la entidad correspondiente en las capas superiores del *Host*. Desde el punto de vista del desarrollador de la aplicación, es importante tener en cuenta que el encabezado del paquete *L2CAP* ocupa cuatro bytes, lo que significa que la longitud efectiva de los datos de usuario es 27 - 4 = 23 bytes (siendo 27 bytes el tamaño de carga útil de la Capa de Enlace).

3.4.5 Protocolo de Atributos (*ATT*)

En *BLE*, cada dispositivo es un cliente, un Servidor o ambos, independientemente de si es un dispositivo *Master* o *Slave*. Un Cliente solicita datos de un Servidor, y un Servidor envía datos a los Clientes. Cada Servidor contiene datos organizados en forma de Atributos, a cada uno de los cuales se le asigna un Identificador de Atributo de 16 bits, un Identificador Único Universal (*UUID*, *Universally Unique IDentifier*), un conjunto de permisos, y un valor. El Identificador de Atributo es simplemente un identificador utilizado para acceder a un valor del Atributo, mientras que el Identificador *UUID* especifica el tipo y la naturaleza de los datos asociados al valor.

Cuando un Cliente desea leer o escribir valores de/en los Atributos, desde o hacia un Servidor, emite una solicitud de lectura o escritura al Servidor a través del Controlador. El Servidor responderá con el valor del Atributo, o un acuse de recibo. En el caso de

una operación de lectura, le corresponde al Cliente analizar el valor e interpretar el tipo de datos en función del *UUID* del Atributo. Por otro lado, durante una operación de escritura, se espera a que el Cliente proporcione datos que sean consistentes con el tipo de Atributo, pudiendo rechazar la operación el Servidor si ese no es el caso.

3.4.6 Administrador de Seguridad (*SMP*)

El Administrador de Seguridad (*SMP*, *Security Manager Protocol*) es a la vez un protocolo y una serie de algoritmos de seguridad diseñados para proporcionar la capacidad de generar e intercambiar claves de seguridad a la pila de protocolos *Bluetooth*, de manera que permitan a los dispositivos comunicarse de forma segura a través de un enlace cifrado para confiar en la identidad del dispositivo remoto, y finalmente, para ocultar la dirección pública de *Bluetooth* si es necesario, con el fin de evitar que dispositivos malintencionados rastreen un dispositivo específico.

3.4.7 Perfil de Atributo Genérico (*GATT*)

El Perfil de Atributo Genérico (*GATT*, *Generic Attribute Profile*) puede considerarse el sistema principal de la transferencia de datos *BLE*, ya que en él se define cómo se organizan y se intercambian los datos entre las aplicaciones. Se basa en el protocolo *ATT* (*ATTribute protocol*) para intercambiar datos entre dispositivos e incorpora una jerarquía y un modelo de abstracción de datos en la parte superior. Define objetos de datos genéricos que pueden ser utilizados y reutilizados por una variedad de perfiles de aplicación (perfiles basados en *GATT*). Mantiene la misma arquitectura Cliente/Servidor presente en *ATT*, pero ahora los datos están encapsulados en servicios, que constan de una o más Características. Se puede considerar que cada Característica es la unión de una parte de datos del usuario junto con otra parte de metadatos (información descriptiva sobre ese valor).

Roles GATT

El Perfil de Atributo Genérico (*GATT*) define dos tipos de roles que pueden adoptar los dispositivos *BLE*:

- Cliente: Envía solicitudes a un Servidor y recibe respuestas (y actualizaciones iniciadas por el Servidor). El Cliente *GATT* inicialmente no tiene conocimiento sobre los Atributos del Servidor, por lo que primero debe consultar acerca de la presencia y la naturaleza de estos, realizando para ello el proceso de descubrimiento de servicios. Después de completar el descubrimiento de

servicios, puede comenzar a leer y escribir los Atributos encontrados en el Servidor, así como a recibir actualizaciones iniciadas por el Servidor.

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

UUIDs

Un Identificador Único Universal (*UUID*, *Universally Unique Identifier*) es un número de 128 bits (16 bytes) globalmente único. Los *UUIDs* se utilizan en muchos protocolos y aplicaciones distintas a *Bluetooth*, y su formato, uso y generación se especifican en la *ISO/IEC 9834-8:2005 [15]*.

BLE emplea estos *UUIDs* para identificar Servicios y Características. Para una mayor eficiencia, y dado que 16 bytes tomarían una gran parte de la longitud de carga de datos de 27 bytes de la capa de Enlace, la especificación *BLE* agrega dos formatos de *UUID* adicionales: *UUID* de 16 y 32 bits. Estos formatos abreviados sólo se pueden usar con los *UUIDs* definidos en la especificación *Bluetooth*. *SIG* proporciona *UUIDs* abreviados para todos los tipos, servicios y perfiles que define y especifica.

Atributos

Los Atributos son la entidad de datos más pequeña definida por *GATT* (y *ATT*). Son piezas direccionables de información que pueden contener datos de usuario o metadatos. Tanto *GATT* como *ATT* únicamente pueden operar con Atributos, por lo que para que los Clientes y Servidores interactúen, toda la información debe organizarse de esta manera.

Conceptualmente, los Atributos siempre se ubican en el Servidor, mientras que el Cliente puede acceder a ellos y modificarlos. La especificación *Bluetooth* define Atributos sólo conceptualmente, y no obliga a las implementaciones *ATT* y *GATT* a usar un formato o mecanismo de almacenamiento interno particular. Cada Atributo contiene información sobre el atributo en sí y sobre los datos actuales, en los campos que se describen a continuación:

- Identificador: Representa un identificador único de 16 bits para cada Atributo en un Servidor *GATT* en particular. Es la parte de cada Atributo que hace que sea direccionable, y se garantiza que no cambiará entre transacciones o a

través de las conexiones. El valor 0x0000 denota un identificador no válido, y la cantidad de identificadores disponibles para cada Servidor *GATT* es 0xFFFE (65535), aunque en la práctica, el número de Atributos en un Servidor suele ser más cercano a unas pocas docenas. El Cliente debe usar la función de descubrimiento para obtener los identificadores de los Atributos que le interesan.

- Tipo: Se corresponde con el tipo del Atributo, que no es más que un *UUID*. Puede ser un *UUID* de 16, 32 o 128 bits, ocupando 2, 4 o 16 bytes, respectivamente. Determina el tipo de datos presentes en el valor del Atributo, y hay mecanismos disponibles para descubrir Atributos basados exclusivamente en su tipo.
- Permisos: Los Permisos son metadatos que especifican qué operaciones de ATT se pueden ejecutar en cada Atributo particular, y con qué requisitos de seguridad específicos. *ATT* y *GATT* definen los siguientes permisos:
    - Permisos de acceso: Determinan si el Cliente puede leer o escribir (o ambos) un valor de Atributo. Cada Atributo puede tener uno de los siguientes permisos de acceso:
        - *None*: El Atributo no puede ser leído ni escrito por un Cliente.
        - *Readable*: El Atributo puede ser leído por un Cliente.
        - *Writable*: El Atributo puede ser escrito por un Cliente.
        - *Readable and Writable*: El Atributo puede ser leído y escrito por el Cliente.
    - Cifrado: Determina si se requiere de un cierto nivel de cifrado para que el Cliente pueda acceder al Atributo. Estos son los permisos de cifrado permitidos, según lo define *GATT*:
        - *No encryption required (Security Mode 1, Level 1)*: El Atributo es accesible en una conexión de texto sin cifrar.
        - *Unauthenticated encryption required (Security Mode 1, Level 2*): La conexión debe cifrarse para acceder al Atributo, pero las claves de cifrado no necesitan ser autenticadas (aunque pueden serlo).
        - *Authenticated encryption required (Security Mode 1, Level 3)*: La conexión debe cifrarse con una clave autenticada para acceder al Atributo.
    - Autorización: Determina si se requiere permiso del usuario para acceder al Atributo. Un Atributo puede elegir entre requerir o no requerir de autorización:
        - *No authorization required*: El acceso al Atributo no requiere de autorización.
        - *Authorization required*: El acceso al Atributo requiere de autorización.

Todos los permisos son independientes entre sí y pueden combinarse libremente por parte del Servidor, que los almacena por Atributo.

- Valor: El *Valor* del Atributo contiene los datos actuales asociados al Atributo. No hay restricciones sobre el tipo de datos que puede contener, aunque su longitud máxima está limitada a 512 bytes de acuerdo a la especificación *Bluetooth*. Esta es la parte de un Atributo a la que un Cliente puede acceder libremente (con los permisos adecuados que lo permitan) para leer y escribir. Todas las demás entidades conforman la estructura del Atributo, y el Cliente no puede modificarlas ni acceder a ellas directamente.

Jerarquía de Datos y Atributos

ATT funciona en términos de Atributos y se basa en todos los conceptos expuestos anteriormente para proporcionar una serie de Unidades de Datos de Protocolo (*PDU*, *Protocol Data Unit*) precisas, comúnmente conocidas como paquetes, que permiten a un Cliente acceder a los Atributos en un Servidor. *GATT* va más allá al establecer una jerarquía estricta para organizar los Atributos de forma reutilizable y práctica, permitiendo el acceso y la recuperación de información entre el Cliente y el Servidor siguiendo un conjunto conciso de reglas que constituyen el marco utilizado por todos los perfiles basados en *GATT*. La Figura 42 muestra la Jerarquía de Datos y Atributos introducida por *GATT*.

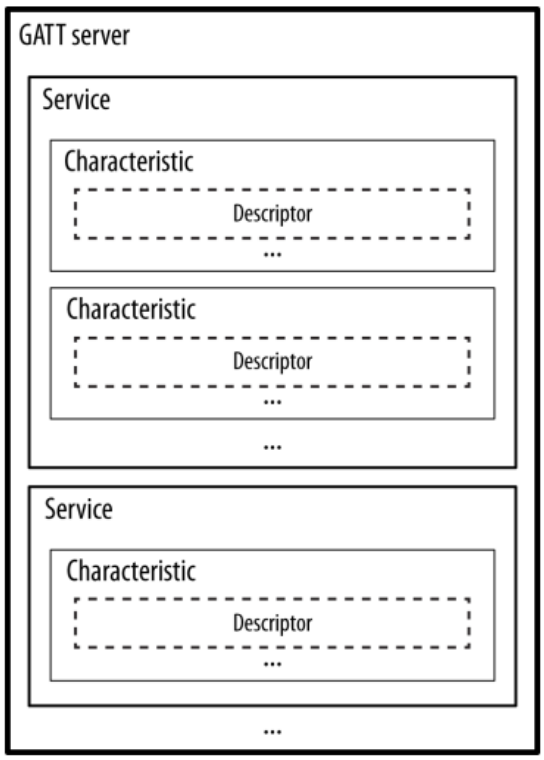

Figura 42. Jerarquía de Datos y Atributos de *GATT*.

En un Servidor *GATT*, los Atributos se agrupan en servicios, cada uno de los cuales puede contener, o no, Características. Estas Características, a su vez, pueden incluir Descriptores. Esta jerarquía se aplica estrictamente a cualquier dispositivo *BLE*, lo que significa que todos los Atributos en un Servidor *GATT* están incluidos en una de estas tres categorías, sin excepción. No puede existir ningún Atributo fuera de esta jerarquía, ya que el intercambio de datos entre dispositivos *BLE* depende de ello.

Servicios

Todos los Atributos dentro de un solo Servicio se conocen como la Definición del Servicio. Por lo tanto, los Atributos de un Servidor *GATT* se corresponden con una sucesión de Definiciones de Servicios, cada una comenzando con un único Atributo que marca el comienzo de un servicio, denominado *Declaración del Servicio*, como se muestra en la Figura 43.

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

Figura 43. Declaración del Servicio.

El *UUID* del Servicio Primario, así como el del Servicio Secundario, son *UUIDs* estándar asignados por *SIG* que se usan como un tipo exclusivo para introducir un servicio, siendo su valor 0x2800 y 0x2801, respectivamente. Un Servicio Primario es el tipo estándar de Servicio de *GATT*, que incluye la funcionalidad estándar relevante expuesta para el Servidor *GATT*. Un Servicio Secundario, por contra, está incluido en los Servicios Primarios y tiene sentido sólo como su modificador, sin tener ningún significado real por sí mismo. En la práctica, los Servicios Secundarios rara vez se utilizan.

Características

Las Características se pueden considerar como contenedores para los datos de usuario. Siempre incluyen al menos dos Atributos: la *Declaración de Característica* (que proporciona metadatos sobre los datos de usuario actuales) y el *Valor de Característica* (que es un atributo completo que contiene los datos de usuario en su campo de valor). Además, el *Valor de Característica* puede ir seguido de Descriptores, que amplían aún más los metadatos contenidos en la *Declaración de Característica*. La Declaración, el Valor y los Descriptores forman la *Definición de Característica*, que es el conjunto de Atributos que conforman una Característica específica. La Figura 44 muestra la estructura de los primeros dos Atributos de cada Característica individual.

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 44. Declaración y Valor de una Característica.

El tipo del Atributo de Declaración de Característica es un *UUID* único y estandarizado (cuyo valor es 0x2803) que se utiliza exclusivamente para indicar el comienzo de las Características. Este Atributo tiene permisos de solo lectura, ya que los Clientes solo pueden recuperar su valor, pero en ningún caso modificarlo. La Figura 45 enumera los diferentes elementos concatenados dentro del valor del Atributo de la Declaración de Característica.

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 45. Valor de la Declaración de Característica.

El campo *Characteristic Value Handle* representa el identificador del Atributo que contiene el valor actual de la Característica; el campo *Characteristic UUID* representa el *UUID* de la Característica; y el campo *Characteristic Properties* indica las operaciones y procedimientos que pueden realizarse con la Característica. En la Figura 46 se muestran las propiedades de las Características definidas por *SIG*.

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 46. Propiedades de una Característica.

El dispositivo Cliente 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 Servidor, pero requieren de su habilitación por parte del dispositivo Cliente, mediante el descriptor *CCCD* (*Client Characteristic Configuration Descriptor*).

Un Cliente puede subscribe para ser notificado cuando se modifique el valor de una Característica, de manera que cuando se produce un cambio, el Servidor se lo notifica al Cliente mediante el envío del nuevo valor. La diferencia entre una Indicación y una Notificación es que en la primera de ellas el Cliente debe confirmar la recepción.

Por último, el Atributo *Characteristic Value* de una Característica contiene los datos que el Cliente 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 Característica.

Descriptores

Los Descriptores se utilizan principalmente para proporcionar al Cliente metadatos (información adicional sobre una Característica y su Valor). Siempre se especifican dentro de la Definición de Característica y después del Atributo *Characteristic Value*. Los Descriptores siempre están compuestos por un solo Atributo, la Declaración de Descriptor de Característica, cuyo *UUID* es siempre el tipo de descriptor y cuyo valor contiene todo lo que define ese tipo de Descriptor particular.

Uno de los descriptores más importantes y de uso común definidos por el GATT es el *CCCD* (*Client Characteristic Configuration Descriptor*). Este descriptor es esencial para el funcionamiento de la mayoría de los Perfiles y casos de uso. Su función es simple: actúa como un interruptor, habilitando o deshabilitando las actualizaciones iniciadas por el Servidor, pero sólo para la Característica a la que se encuentra asociado.

El valor de un CCCD no es más que un campo de dos bits, con un bit correspondiente a las notificaciones, y el otro a las indicaciones. Un Cliente puede establecer y borrar esos bits en cualquier momento, y el Servidor los examinará siempre que la Característica asociada cambie su valor y pueda ser susceptible de una actualización. Cada vez que un Cliente desea habilitar notificaciones o indicaciones para una Característica particular, simplemente utiliza un paquete *ATT* de solicitud de escritura para establecer el bit correspondiente al valor 1.

3.4.8 Perfil de Acceso Genérico (*GAP*)

En el Perfil de Acceso Genérico (*GAP*, *Generic Access Profile*), existen cuatro roles que un dispositivo BLE puede adoptar para unirse a una red:

- *Broadcaster*: Optimizado para aplicaciones de solo transmisión, que distribuyen datos regularmente. Se envían periódicamente paquetes de *Advertising* con datos, y los datos están accesibles para cualquier dispositivo que esté escuchando. El rol *Broadcaster* utiliza el rol de *Advertiser* de la capa de Enlace.
- *Observer*: Optimizado para aplicaciones de solo recepción, que desean recopilar datos de dispositivos de *Braodcasting.* Escucha los datos incluidos en los paquetes de *Advertising* de los dispositivos de *Braodcasting*. El rol de *Observer* utiliza el rol *Scanner* de la capa de Enlace.
- *Central*: El rol *Central* corresponde al *Master* de la capa de Enlace. Un dispositivo capaz de establecer múltiples conexiones con otros dispositivos. El rol *Central* es siempre el iniciador de las conexiones, y esencialmente permite que los dispositivos entren en la red. El protocolo *BLE* es asimétrico, lo que significa que los requisitos del *Master* de la capa de Enlace son mayores que los de un *Slave,* por lo que el rol *Central* generalmente lo desempeña un *smartphone* o tableta, ya que tiene acceso a potentes *CPU* (*Central Processing Unit*) y recursos de almacenamiento. Esto le permite mantener conexiones con múltiples dispositivos. El dispositivo *Central* comienza por escuchar los paquetes de *Advertising* de otros dispositivos y luego inicia una conexión con el dispositivo seleccionado. Este proceso se puede repetir para incluir múltiples dispositivos en una sola red.
- *Peripheral*: El rol *Peripheral* corresponde al *Slave* de la capa de Enlace. Emplea paquetes de *Advertising* para permitir que los dispositivos *Central* lo encuentren y, posteriormente, establecer una conexión con él. El protocolo *BLE* está optimizado para requerir pocos recursos para la implementación del dispositivo *Peripheral*, al menos en términos de potencia de procesamiento y memoria.

Cada dispositivo particular puede operar en uno o más roles a la vez, y la especificación *BLE* no impone restricciones a este respecto. Muchos desarrolladores intentan asociar erróneamente los roles de Cliente y Servidor *GATT* con los roles de *GAP*. No existe conexión alguna entre ellos, y cualquier dispositivo puede ser un Cliente *GATT*, un servidor o ambos, dependiendo de la aplicación y su situación. Los roles Cliente/Servidor *GATT* dependen exclusivamente de la dirección en la que fluyen las solicitudes de datos y las transacciones de respuestas, mientras que los roles *GAP* se mantienen constantes.

3.5 Introducción a los procesos de conexión y transferencia de datos del protocolo *BLE*

3.5.1 Dirección de un dispositivo *BLE*

La dirección de un dispositivo *BLE* es un valor de 48 bits (6 bytes) que identifica de forma única al dispositivo. Existen dos tipos de direcciones, pudiendo configurarse una o ambas en cada dispositivo particular:

- *Public Device Address*: Equivalente a una dirección fija, programada de fábrica en el dispositivo BLE. Debe estar registrada por *IEEE Registration Authority*, y nunca cambiará durante la vida útil del dispositivo.
- *Random Device Address*: Esta dirección puede pre-programarse en el dispositivo o generarse dinámicamente en tiempo de ejecución.

3.5.2 Procedimientos de *Advertising* y *Scanning*

Existe un único formato de paquete *BLE* que se divide en dos tipos: Paquetes de *Advertising* y Paquetes de Datos. Los Paquetes de *Advertising* se emplean para transmitir datos en aplicaciones que no requieran de un establecimiento de conexión completo, o para descubrir dispositivos *Slave* y conectarse a ellos.

Cada Paquete de *Advertising* puede contener hasta 31 bytes de carga útil, junto con la información básica de encabezamiento (incluida la dirección del dispositivo *Bluetooth*). Tales paquetes son simplemente transmitidos a ciegas por el dispositivo *Advertiser* sin el conocimiento previo de la presencia de cualquier dispositivo *Scanner*. Se envían a una tasa fija definida por el parámetro *Advertising Interval*, cuyo valor está comprendido entre 20 ms y 10.24 s, en pasos de 0.625 ms, excepto para los tipos *Non-connectable* y *Scannable* que como mínimo tiene que ser de unos 100 ms. Cuanto menor es el valor de este intervalo, mayor es la frecuencia con la que se emiten los paquetes de *Advertising*, lo que aumenta la probabilidad de que esos paquetes sean recibidos por un dispositivo *Scanner*, pero también se incurre en un mayor consumo de energía. El retardo es un valor aleatorio entre 0 ms y 10 ms que se añade automáticamente. Este último valor ayuda a reducir colisiones entre *Advertising* de dispositivos diferentes. De esta manera, *BLE* mejora la robustez del protocolo haciendo más fácil la búsqueda de los *Advertising* por parte del *Scanner*.

Debido a que el proceso de *Advertising* emplea un máximo de tres canales de frecuencia, y el dispositivo *Advertiser* y el dispositivo *Scanner* no están sincronizados

de ninguna manera, el dispositivo *Scanner* solo recibirá un paquete de *Advertising* cuando se solapen aleatoriamente, como se muestra en la Figura 47.

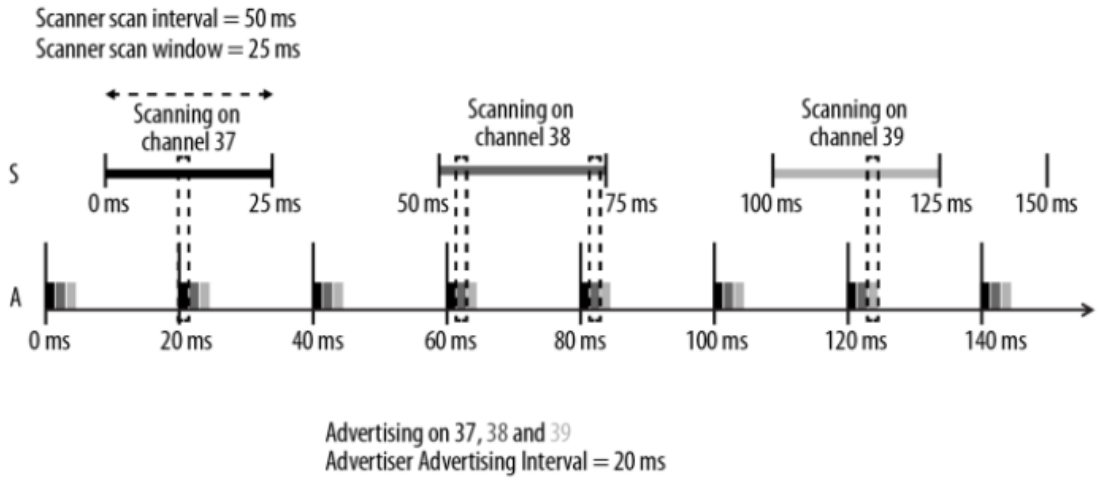

Figura 47. Procedimiento de *Advertising* y de *Scanning*.

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

La especificación *BLE* define dos tipos básicos de procedimientos de *Scanning*:

- *Passive Scanning*: El dispositivo *Scanner* simplemente escucha paquetes de *Advertising*, y el dispositivo *Advertiser* nunca sabe si alguno de estos paquetes ha sido recibido por un *Scanner*.
- *Active Scanning*: El dispositivo *Scanner* emite un paquete de solicitud de *Scanning* (*Scan Request*) después de recibir un paquete de *Advertising*. El dispositivo *Advertiser* lo recibe y responde con un paquete de respuesta de *Scanning* (*Scan Response Data*).

La Figura 48 ilustra la diferencia entre ambos tipos de procedimiento de *Scanning*.

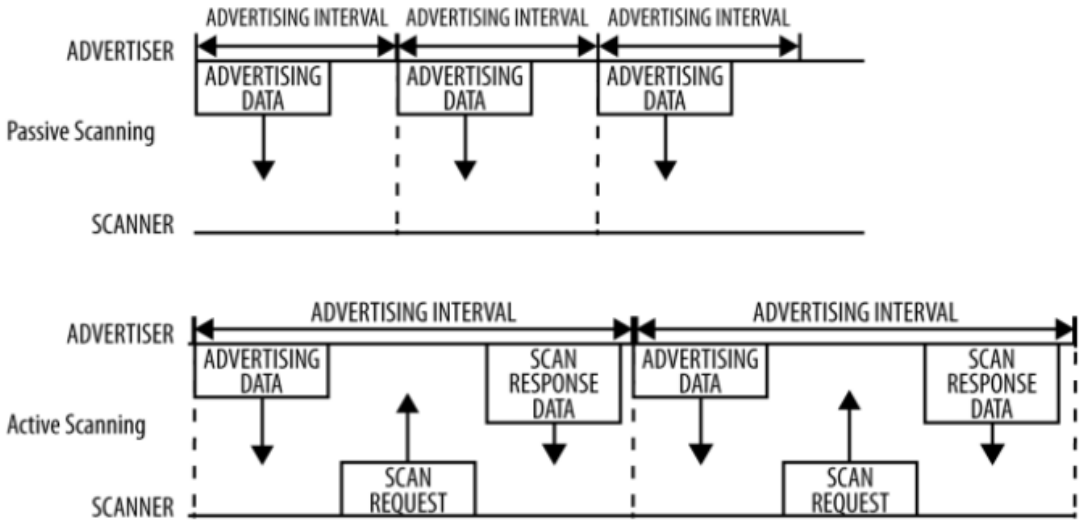

Figura 48. *Passive Scanning* y *Active Scanning.*

En cuanto a los tipos de paquetes de *Advertising*, estos se clasifican según las siguientes propiedades:

- *Connectable*: Indica si un dispositivo *Scanner* puede iniciar o no una conexión al recibir un paquete de *Advertising*.
- *Scannable*: Indica si un dispositivo *Scanner* puede emitir o no una solicitud de *Scanning* al recibir un paquete de *Advertising*.
- *Directed*: Si es dirigido, el paquete de *Advertising* contiene las direcciones *BLE* del dispositivo *Scanner* de destino y del dispositivo *Advertiser* en su carga útil, y no se permiten datos de usuario. En cambio, si no es dirigido, el paquete de *Advertising* no contiene la dirección *BLE* de ningún dispositivo *Scanner* particular, y puede contener datos de usuario en su carga útil.

Las diferentes combinaciones de estas propiedades se muestran en la Tabla 3. 2.

Tabla 3. 2. Tipos de Paquetes *Advertising* y sus características.

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 Undirect

- *ADV_IND* (*Connectable Undirected Advertising*): Este tipo de eventos es el más común y el más genérico de todos, debido a que no es dirigido y es conectable. Es decir, un dispositivo *Central* puede conectarse a otro *Peripheral* que está en estado de *Advertising* y no está dirigido hacía ningún dispositivo *Central* concreto.
  Cuando un periférico envía paquetes de *Advertising* (*ADV_IND*), está ayudando a dispositivos *Central* a encontrarle, una vez encontrado, el dispositivo *Central* puede empezar el proceso de conexión. En la Figura 49 se muestra este modo de conexión.

Figura 49. *Connectable Undirected Advertising*.

Para informar que ha sido encontrado por el *Scanner*, el *Advertiser* recibe un *Scan Request* al cual contesta con un *Scan Response* en el que completa la información necesaria para que el *Scanner* cambie de estado a Initiating. Estando en el estado *Initiating*, el siguiente *Advertising* que recibe se le contesta con un *Connect Request*, momento en el que empieza el proceso de conexión.

En la Figura 50 se puede visualizar cómo es el flujo de mensajes entre un dispositivo *Central* y uno *Pheripheral*.

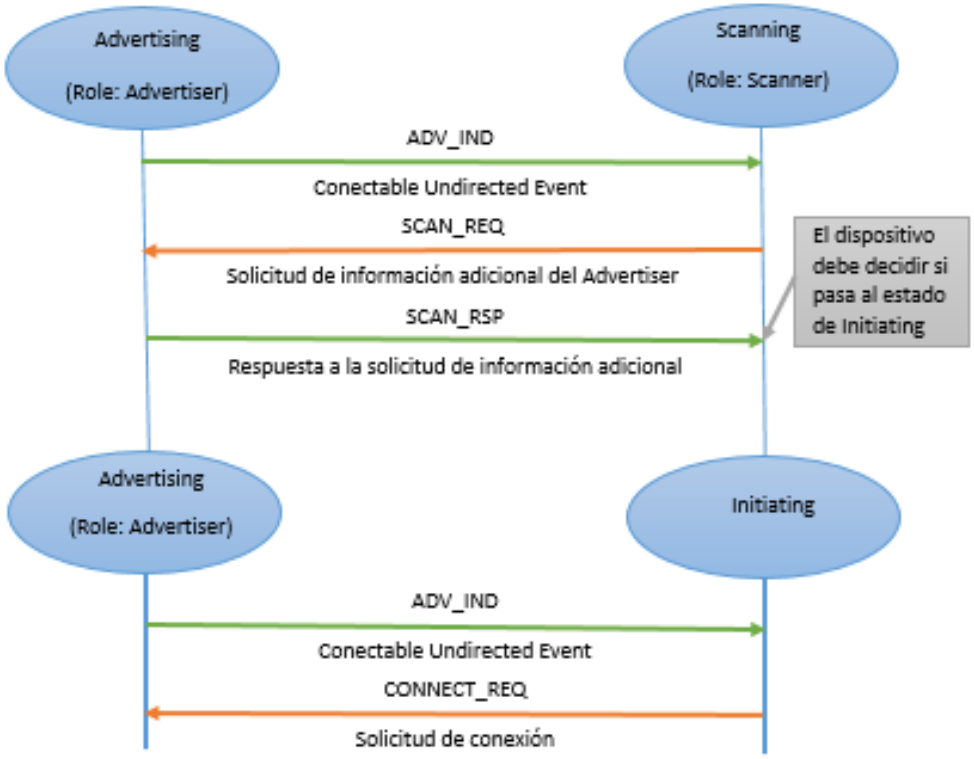

Figura 50. Flujo de mensajes del modo *Connectable Undirected Advertising*.

- *ADV_DIRECT_IND* (*Connectable Directed Advertising*): Este evento se usa cuando un *Advertiser* quiere que se conecte con un dispositivo *BLE* concreto. Después de recibir este tipo de *PDU* (*Protocol Data Unit*) por el dispositivo solicitado, este último puede contestar con un *Connect Request* para solicitar una conexión con el *Advertiser*. Estos paquetes contienen dos direcciones: la dirección del *Advertiser* y la del *Initiator*. Además tienen que cumplir unos requisitos de tiempo especiales, dos paquetes de este tipo de *Advertising* enviados por el mismo canal (dos eventos consecutivos) tienen que estar separados como máximo unos 3.75 ms. Este requisito permite un escaneo de solamente 3.75 ms para detectar los *Advertising.* Todo esto es mostrado en la Figura 51.

Figura 51. *Connectable Directed Advertising*.

Por lo tanto, de esta forma puede conectarse de rápidamente, pero esta estrategia hace que se congestionen los canales rápidamente. Por esta razón, no se permiten utilizar los *Advertising Directed* por un tiempo superior a 1.28 s. Si no se consigue establecer una conexión antes de los 1.28 s el *Controller* detiene los *Advertising*

de forma automática. Hay que destacar unas características importantes: cuando se usa este tipo de *Advertising*, el dispositivo ignora cualquier *Scan Request* recibido. Además, sus paquetes no permiten tener ningún tipo de datos adicionales, solamente las dos direcciones afectadas. En la Figura 52 se muestra el flujo de paquetes de este tipo de *Advertising*.

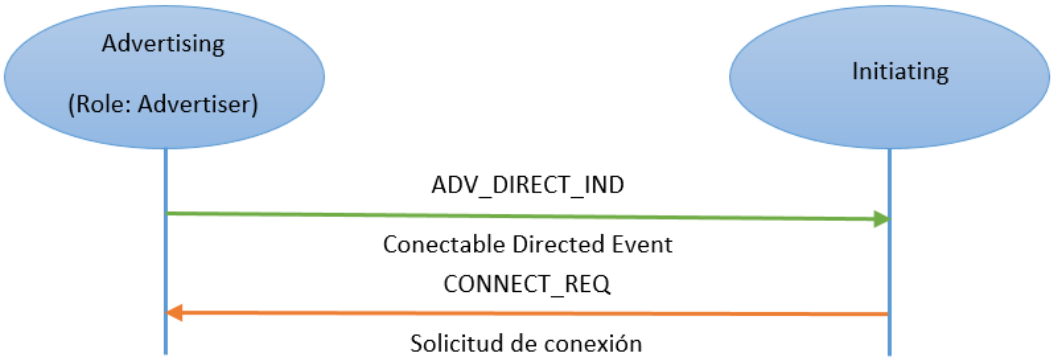

Figura 52. Flujo de mensajes del modo *Connectable Directed Advertising*.

- *ADV_NONCONN_IND* (*Non-connectable Undirected Advertising*): Este tipo de eventos de *Advertising* es el utilizado por dispositivos que no tienen intención de conectarse a ningún otro dispositivo, incluso no quieren ni saber si están siendo escaneados por otros dispositivos centrales. Estos dispositivos se utilizan para emitir datos en modo *Broadcast*. Los dispositivos *Scanner* solo pueden escuchar la información emitida, no pueden ni conectarse ni solicitar más información. Como se puede ver en la Figura 53, el tiempo entre dos paquetes ADV_NONCONN enviados en canales consecutivos tiene que ser como máximo 10 ms.

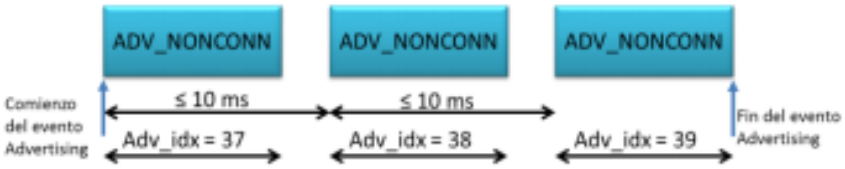

Figura 53. *Non-Connectable Undirected Advertising*.

Este tipo de *Advertising* puede ser útil para aplicaciones que simplemente quieren detectar la existencia de un objeto o un estado.

- *ADV_SCAN_IND* (*Scannable Undirected Advertising*): Este último tipo es parecido al anterior, con la diferencia de que es escaneable. No puede entrar en conexión con el otro dispositivo, pero además de enviar Advertising, puede escuchar *Scan Request* de un *Scanner* activo y responder con un *Scan Response*. En este caso se puede utilizar para enviar información adicional requerida por el dispositivo *Scanner*, como se muestra en la Figura 54.

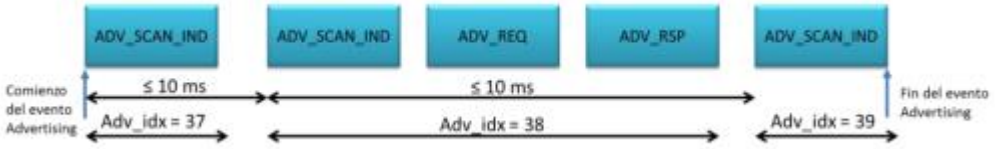

Figura 54. *Scannable Undirected Advertising*.

3.5.3 Procesos de conexión

Para establecer una conexión, un dispositivo *Scanner* comienza en primer lugar a buscar dispositivos *Advertiser* que acepten solicitudes de conexión. Cuando detecta un dispositivo *Advertiser* adecuado, el dispositivo *Scanner* envía un paquete de solicitud de conexión (*Connect Request*) al dispositivo *Advertiser* y, siempre que éste responda, establece una conexión. El paquete de solicitud de conexión incluye el incremento de salto de frecuencia, que determina la secuencia de salto que seguirán los dispositivos *Master* y *Slave* durante la vida útil de la conexión. Una conexión es simplemente una secuencia de intercambio de paquetes de datos entre el *Slave* y el *Master* en tiempos predefinidos, como se muestra en la Figura 55, denominándose cada intercambio Evento de Conexión (*Connection Event*). Además, por defecto, en cada Evento de Conexión ambos dispositivos transmiten un paquete, aunque no tengan datos que enviarse, denominado *Empty Link Layer PDU*, con el fin de garantizar que la conexión aún está activa.

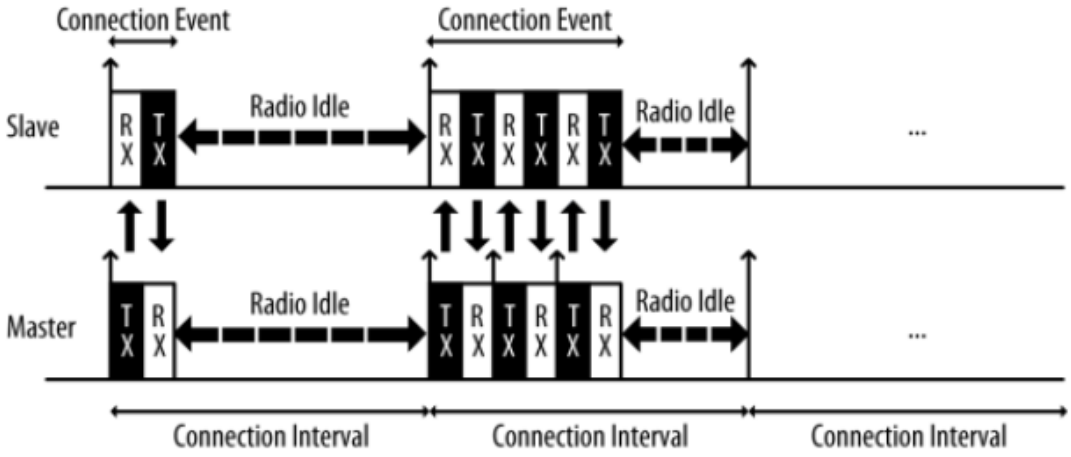

Figura 55. Eventos de Conexión.

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

- Intervalo de Conexión (*Connection Interval*): Tiempo entre el inicio de dos eventos de conexión consecutivos. Este valor varía entre 7.5 ms (alto

rendimiento) y 4 s (el rendimiento más bajo posible, pero también el que menos energía consume), incrementándose en pasos de 1.25 ms.

- Latencia del Esclavo (*Slave Latency*): Cantidad de eventos de conexión que un dispositivo *Slave* puede elegir ignorar sin poner en riesgo la conexión.
- Tiempo de Espera de Supervisión de Conexión (*Connection Supervision Timeout*): Tiempo máximo entre dos paquetes de datos válidos recibidos, antes de que una conexión se considere perdida.

En lo referente a los paquetes de datos intercambiados bidireccionalmente entre un dispositivo *Master* y un dispositivo *Slave* durante los eventos de conexión, estos tienen una carga útil de datos de 27 bytes, pero los protocolos superiores de la pila de protocolos de BLE normalmente limitan la cantidad real de datos de usuario a 20 bytes por paquete.

Todos los paquetes recibidos se comparan con un CRC (*Cyclic Redundancy Check*) de 24 bits, y se solicitan retransmisiones cuando la comprobación de errores detecta un error en la transmisión, sin haber un límite superior para el número de retransmisiones. La Capa de Enlace reenviará el paquete hasta que el receptor lo reconozca finalmente.

Además de los procesos de *Advertising, Scanning,* establecimiento (y destrucción) de conexiones, y la transmisión y recepción de datos, la capa de Enlace también es responsable de varios procedimientos de control, incluidos los siguientes procesos críticos:

- Actualización de los parámetros de conexión: Cada conexión se establece con un conjunto determinado de parámetros de conexión establecidos por el dispositivo *Master*, si bien las condiciones y los requisitos pueden cambiar durante la vida útil de la conexión. Un dispositivo *Slave* puede requerir en un determinado instante de mayor rendimiento para procesar una ráfaga breve de datos, o por el contrario, puede detectar que en un futuro próximo un Intervalo de Conexión más largo será suficiente para mantener la conexión con vida. La capa de Enlace permite a los dispositivos *Master* y *Slave* solicitar la actualización del valor de los parámetros de conexión y, en el caso del *Master*, configurarlos unilateralmente en cualquier momento. De esta forma, cada conexión se puede ajustar para proporcionar el mejor equilibrio entre rendimiento y consumo de energía.
- Cifrado: La capa de Enlace proporciona los medios para intercambiar datos de forma segura a través de un enlace cifrado. Las claves son generadas y administradas por el *Host*, pero la capa de Enlace realiza el cifrado y descifrado de datos de manera transparente a las capas superiores.

3.5.4 Formato de los paquetes *BLE*

Puede observarse en la Figura 56 el formato de un paquete *BLE*. Los paquetes *BLE* tienen una carga útil de datos de 27 bytes para las especificaciones *Bluetooth* 4.0 y 4.1, mientras que para la especificación *Bluetooth* 4.2, la carga útil de datos es de 255 bytes.

Figura 56. Formato de un paquete *BLE.*

Si se analiza el campo de datos, éste se ocupa con paquetes de la capa *L2CAP*, tal y como se muestra en la Figura 57. El encabezado *L2CAP* presenta un tamaño de 4 bytes, y se coloca para cumplir los requisitos de re-ensamblaje y fragmentación de paquetes que tienen una longitud mayor que la permitida en el campo de datos. Por tanto, esto hace que el tamaño máximo del campo de datos *ATT* sea de 23 bytes.

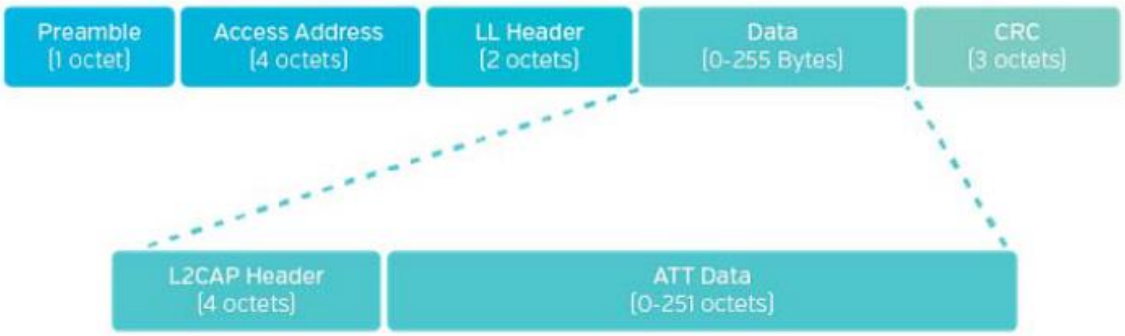

Figura 57. Campo de datos de un paquete *BLE*.

En cuanto al formato de los paquetes de datos *ATT*, estos presentan la estructura mostrada en la Figura 58. El campo *Op-Code*, de 1 byte, indica la operación ATT correspondiente: comando de escritura, notificación, respuesta de lectura, etc. Además, al enviar paquetes de escritura, lectura y notificación o indicación, también se deberá incluir el Identificador de Atributo asociado (2 bytes) para la identificación de los datos. Por tanto, esto resulta en una longitud efectiva de datos de usuario de 20 bytes. También aparece el concepto *ATT MTU* (*Maximum Transmission Unit*). Esta Unidad Máxima de Transmisión indica la longitud máxima de un paquete *ATT*. El parámetro *MTU* está definido por *L2CAP* y puede ser de 23 bytes o mayor. Su tamaño depende de la implementación de la pila *Bluetooth*.

Figura 58. Formato de los paquetes de datos ATT.

3.5.5 Descubrimiento de Servicios y Características

El dispositivo Cliente no tiene conocimiento sobre los Atributos que podrían estar presentes en un servidor *GATT* cuando se conecta por primera vez. Por lo tanto, es imprescindible que el cliente comience realizando una serie de intercambios de paquetes con el propósito de determinar la cantidad, ubicación y naturaleza de todos los Atributos que podrían ser de interés, antes de comenzar el intercambio de paquetes de datos. Para el descubrimiento de servicios primarios, el perfil *GATT* ofrece las siguientes opciones:

- Descubrimiento de todos los servicios primarios: Los dispositivos Cliente pueden obtener una lista completa de todos los Servicios principales (independientemente de los *UUID* de servicio) del servidor *GATT*. Esto se utiliza cuando el cliente admite más de un servicio.
- Descubrimiento del Servicio Primario*, por UUID de Servicio*: Siempre que el cliente sepa qué servicio está buscando (solo admite ese servicio único), simplemente puede buscar todas las instancias de un servicio particular haciendo uso de su *UUID*.

Cada uno de estos procedimientos genera rangos de identificadores, pertenecientes a los Atributos de cada uno de los servicios descubiertos. Si se emplea la opción de "descubrimiento de todos los servicios primarios", también se obtienen los *UUID* de cada servicio.

En términos de descubrimiento de Características, el perfil *GATT* ofrece las siguientes opciones:

- Descubrimiento de todas las Características de un servicio: Una vez que el Cliente ha obtenido el rango de identificadores para un Servicio de su interés, puede proceder a obtener una lista completa de sus Características. La única entrada es el rango de identificadores, y a cambio, el Servidor devuelve, tanto el identificador como el valor de todos los Atributos de las Características incluidas dentro de ese Servicio.
- Descubrimiento de Características por *UUID*: Este procedimiento es idéntico al anterior, excepto que el Cliente descarta todas las respuestas que no coinciden con el *UUID* de la Característica especificada.

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

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

Capítulo 4. Código de integración para el funcionamiento de la célula *modul P16* de *metec*

4.1 Introducción

En este capítulo se presenta el desarrollo correspondiente al *firmware* del sistema basado en *MCU,* con el fin de reproducir el código de representación Braille en la célula *modul P16* de *metec* integrada en la plataforma *HW*. Como ya se ha mencionado, el dispositivo basado en *MCU* que ejecutará el código diseñado específicamente para esta célula de lectura Braille, es el *RedBear Duo*. La plataforma utilizada para el desarrollo de este *firmware* es *build.particle.io,* basada en un lenguaje de programación *Wiring*.

Antes de presentar el código implementado y validar el funcionamiento de la plataforma *HW* planteada para la integración de la célula de lectura Braille, en el siguiente apartado se comentarán las aplicaciones utilizadas para la realización y verificación funcional del *firmware* diseñado para la célula de *metec*.

4.2 Aplicaciones

A continuación se realizará una descripción detallada de cada una de las aplicaciones que se han utilizado en el proceso de realización de este TFG.

4.2.1 Particle IDE

Se trata de un entorno de desarrollo integrado (*IDE*), es decir, que puede realizar desarrollo de software de aplicaciones, ejecutándose desde un navegador web [16].

La interfaz de usuario es muy intuitiva, y en la parte superior izquierda presenta tres botones:

- *Flash*: realiza una actualización del *firmware*, y carga el código en el dispositivo seleccionado.

- *Verify*: compila el código, y si aparece algún error los muestra por pantalla en la consola de depuración.
- *Save*: Almacena los cambios realizados sobre el código.

Aparte de los botones presentados, existen otro cinco botones en la parte inferior izquierda de la pantalla:

- *Code*: muestra la lista de las aplicaciones desarrolladas y permite cambiar de una a otra para editarlas o cargarlas en el dispositivo conectado.
- *Library*: Recoge todas las librerías, permitiendo generar otras.
- *Docs*: apartado de ayuda de *Particle*, sobre sus dispositivos y ejemplos de *firmware* desarrollado.
- *Devices*: muestra todos los dispositivos asociados a la cuenta de usuario que se desea usar.
- *Settings*: configuración de la cuenta de usuario.

Todo esto se muestra en la Figura 59.

Figura 59. Pantalla de desarrollo *SW* de la *IDE* de *Particle*.

En la esquina inferior derecha se puede ver el último dispositivo conectado.

Como ya se ha descrito, este *IDE* se ha utilizado para generar el código *firmware* que permita el control del *MCU* en el sistema desarrollado en el presente TFG.

4.2.2 nRF Connect

Nordic Semiconductor ha diseñado la aplicación *nRF Connect* con el objetivo de evaluar, probar y verificar diseños de Bluetooth de baja energía (*BLE*) para dispositivos iOS y Android [17]. Es una herramienta genérica que permite escanear, anunciar y explorar dispositivos *BLE* y comunicarse con ellos, como dispositivo *Central*. En la Figura 60 se muestra el logo de esta aplicación.

Figura 60. Logo de *nRF Connect*, aplicación de *Nordic Semiconductor*.

Presenta una interfaz de usuario sencilla, al igual que su configuración. En la Figura 61 se puede ver la pantalla de inicio de esta *app*.

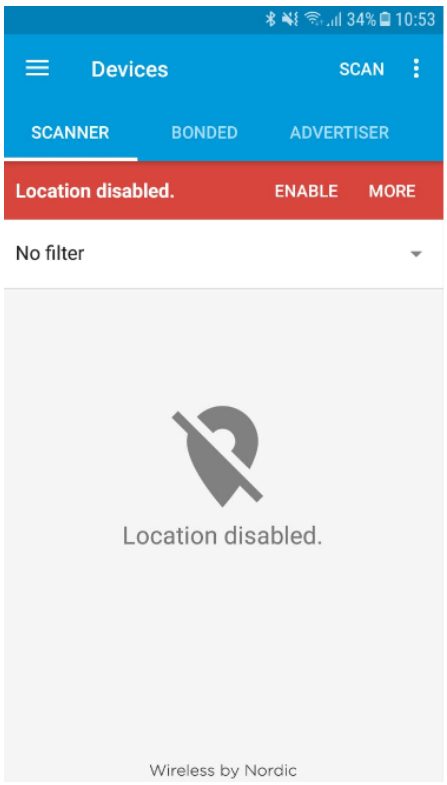

Figura 61.Interfaz de usuario de *nRF Connect*.

El modo de empleo de *nRF Connect* es intuitivo, pues para iniciar la búsqueda de dispositivos *Peripheral* que se encuentren en el proceso de *Advertising*, sólo es necesario pulsar en la esquina superior derecha sobre *Scan*. En la Figura 62 se puede ver que tras localizarlos, muestra por pantalla todos los dispositivos disponibles. Para establecer la comunicación con uno de estos dispositivos *Peripheral*, hay que seleccionarlo.

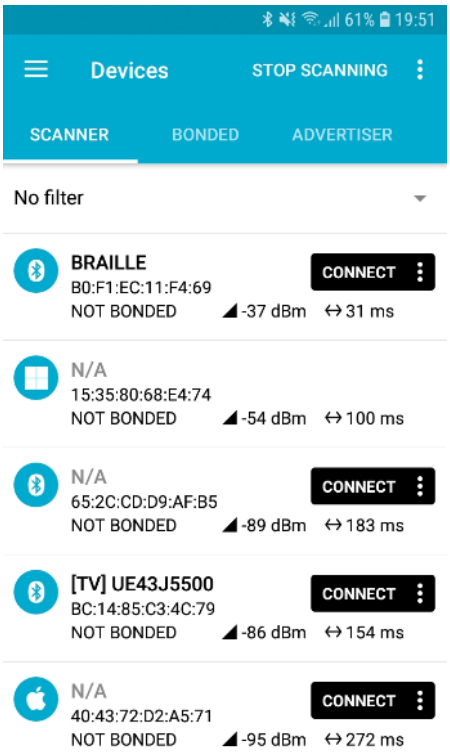

Figura 62. Aplicación *nRF Connect* - Dispositivos *Peripheral* descubiertos.

Tras la conexión con el dispositivo *Peripheral* se pueden ver los Servicios y las Características del dispositivo *Peripheral* y si éstas incluyen la propiedad de escritura, se le puede transmitir diferentes tipos de datos o información. Si el proceso se ha ejecutado correctamente, presentará por pantalla la información transmitida, como se muestra en la Figura 63.

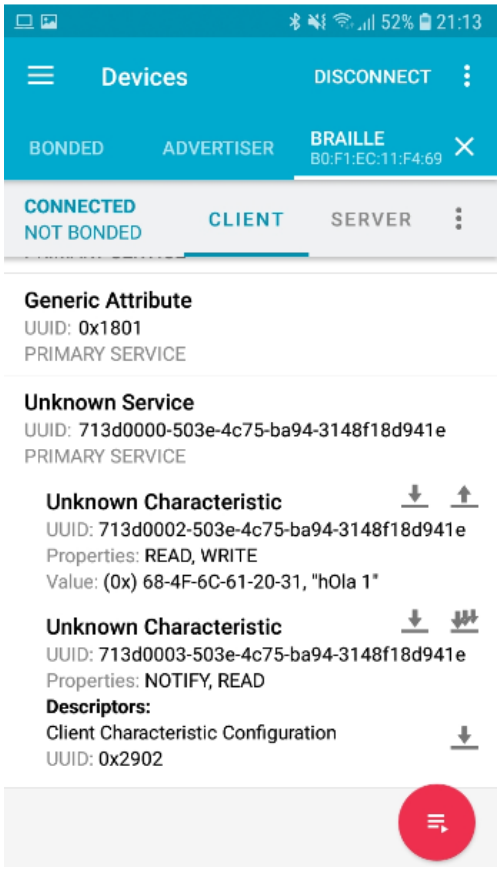

Figura 63. Envío del texto “hOla 1” desde el dispositivo *Central*.

En este TFG se utiliza esta aplicación para conectar un dispositivo móvil al sistema de representación de textos desarrollado en base a la célula modul P16 para la representación de textos en código Braille. La intención principal es enviar con el *smartphone* actuando como dispositivo *Central*, al *RedBear Duo* que actuará como dispositivo *Peripheral*, texto plano legible que pueda transformar a código Braille y que luego se represente en la célula de lectura de *metec*.

4.2.3 PuTTY

PuTTy es un emulador de terminal que se puede adquirir de manera gratuita. Este cliente remoto es capaz de trabajar con protocolos como SSH, Telnet y RLogin; y sobre sistemas Windows, Linux o Unix entre otros. Su logo se muestra en la Figura 64.

Figura 64. Logo de PuTTY.

Posee una interfaz de usuario intuitiva y con capacidad de configuración sencilla, como se muestra en la Figura 65. En este caso se opta por la opción de usar el terminal serie.

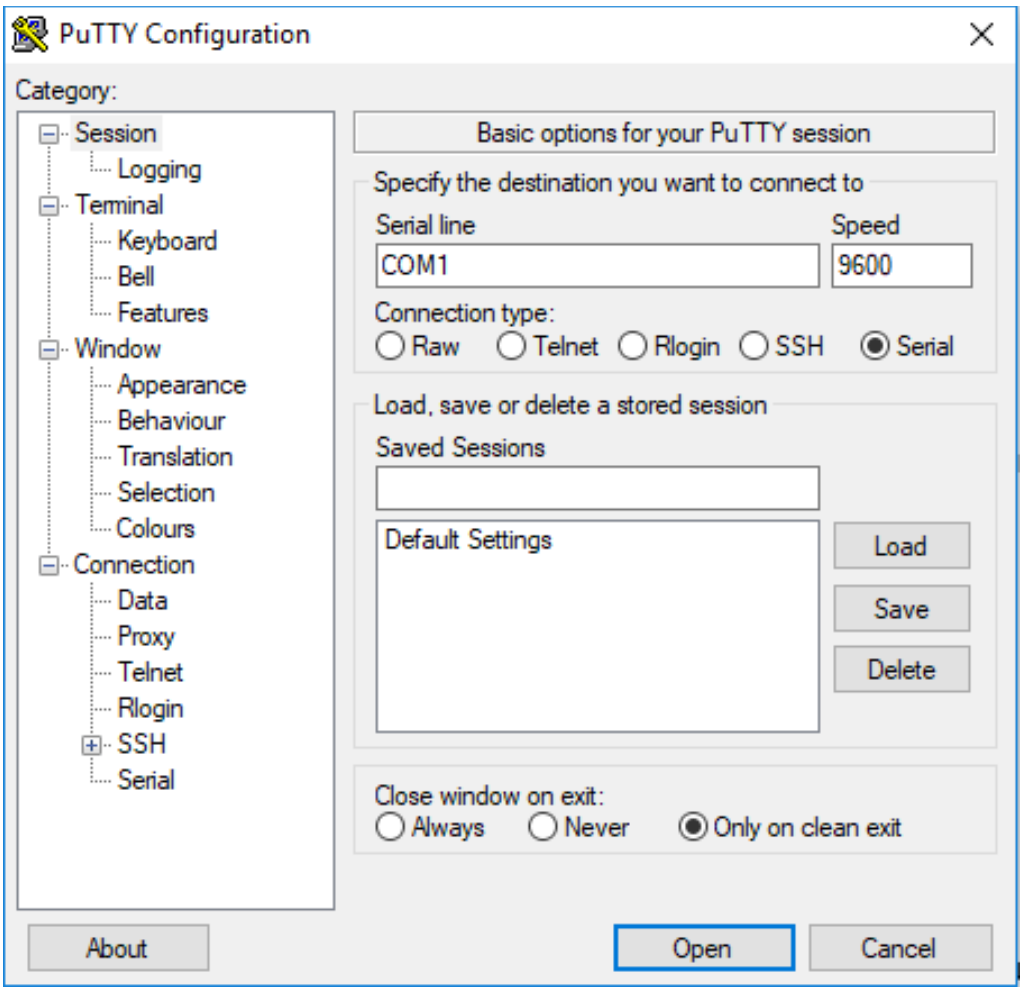

Figura 65. Ventana de configuración de *PuTTY*.

Una vez se haya elegido la configuración deseada, se abre la pantalla del terminal esperando órdenes, o haciendo una escucha de lo que ocurre. Este programa tiene un menú más elaborado y mayores opciones y prestaciones a las mencionadas, pero como en este TFG no se han utilizado, se ha decidido no describirlas. En la Figura 66 se puede ver el terminal que ofrece *PuTTY*.

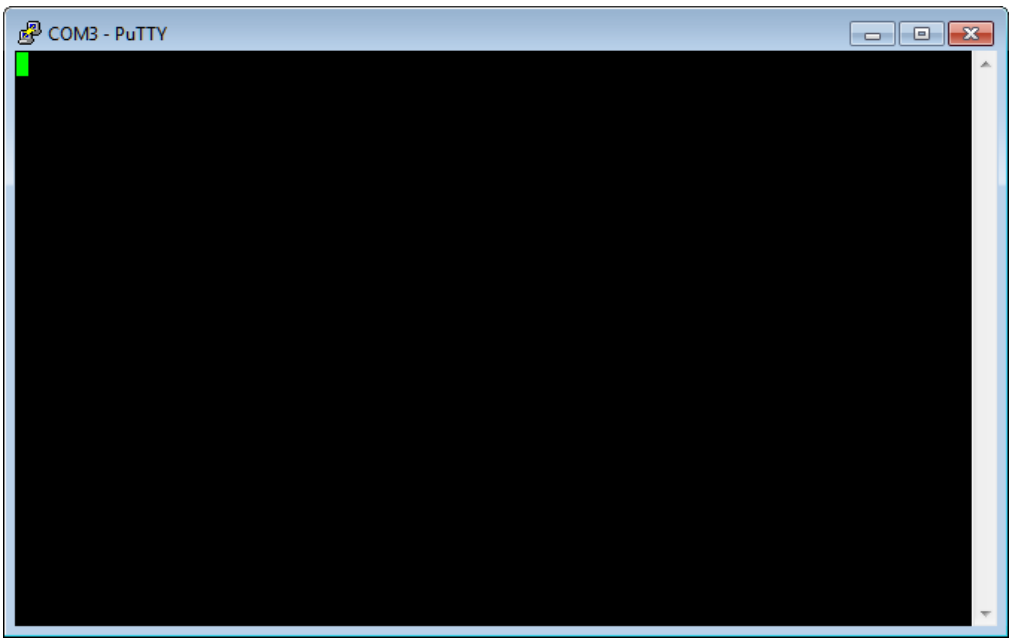

Figura 66. Terminal de *PuTTY* en espera.

En este TFG, este programa es necesario para comprobar que toda la información enviada desde el dispositivo móvil llega al dispositivo *RedBear Duo* y que el programa cargado sobre éste, se ejecuta según lo establecido. Se ha utilizado el hiperterminal a modo de escucha para verificar el funcionamiento del sistema y encontrar los errores de *software* de una manera más eficiente.

4.3 Funciones del dispositivo RedBear Duo como *BLE Pheripheral*

Como parte importante de este TFG, y antes de entrar en detalle en el código desarrollado, se va hacer un breve inciso para explicar las funciones que permiten programar la funcionalidad del dispositivo *RedBear Duo* como *BLE Peripheral [11]*.

Las funciones del *firmware* del dispositivo *RedBear Duo* encargadas de la funcionalidad de este dispositivo para que opere como *BLE Peripheral* son:

- *setAdvertisementParams()*: Establece los parámetros de *Advertising* del dispositivo *Peripheral*.
- *setAdvertisementData()*: Establece los datos de *Advertising* del dispositivo *Peripheral*. Los datos de *Advertising* son esenciales, y están limitados a 31 bytes. Por lo general, una unidad en los datos de *Advertising* está compuesta por "Longitud + Tipo AD + Datos AD". Con respecto a los dispositivos BLE, el Tipo AD de la primera unidad debe ser *BLE_GAP_AD_TYPE_FLAGS* (0x01) y los Datos AD indicados a continuación deben ser *BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE*

(BLE_GAP_ADV_FLAG_LE_GENERAL_DISC_MODE (0X02) | BLE_GAP_ADV_FLAG_BR_EDR_NOT_SUPPORTED (0X04)).

- *setScanResponseData()*: Establece los datos de respuesta al proceso de *scan* de dispositivos *Peripheral*. Estos datos son opcionales, y están limitados a 31 bytes. Por lo general, una unidad en los datos de respuesta al proceso de *scan* está compuesta por "Longitud + Tipo AD + Datos AD".
- *startAdvertising()*: Inicia el proceso de *Advertising* para hacer que el dispositivo *Peripheral* sea reconocible por los dispositivos *Central*. Antes de llamar a esta función, es necesario establecer los datos de *Advertising* mediante una llamada a la función *setAdvertisementData()*.
- *stopAdvertising()*: Detiene el proceso de *Advertising*. Cuando el dispositivo *Peripheral* está en el proceso de *Advertising*, se puede llamar a esta función para detenerlo, de modo que los dispositivos *Central* ya no puedan detectarlo.
- *requestConnParamsUpdate()*: Solicita la actualización del valor de los parámetros de conexión a partir de la especificación de cinco parámetros:
  - *conn_handle*: El *handle* (o gestor) de conexión.
  - *conn_interval_min*: Valor de tipo *uint16_t* que indica el valor mínimo del Intervalo de Conexión.
  - *conn_interval_max*: Valor de tipo *uint16_t* que indica el valor máximo del Intervalo de Conexión.
  - *latency*: Valor de tipo *uint16_t* que indica la latencia del dispositivo.
  - *Slave*. Es decir, la cantidad de eventos de conexión que un dispositivo *Slave* puede optar por omitir sin que se produzca una desconexión. Al establecer una latencia con un valor distinto de cero, el dispositivo *Peripheral* puede optar por no responder cuando el dispositivo *Central* solicita datos, hasta el número máximo de veces establecido. Sin embargo, si el dispositivo *Peripheral* tiene datos para enviar, puede optar por enviar datos en cualquier momento.
  - *supervision_timeout*: Valor de tipo *uint16_t* que indica el tiempo de espera de supervisión de la conexión. Es el tiempo de espera desde el último intercambio de datos, hasta que un enlace se considera perdido.

Las funciones relacionadas con el rol *GATT Serve*r de un dispositivo *BLE Peripheral* son:

- *addService()*: Añade un servicio *BLE* al Servidor *GATT*, cuyo *UUID* puede ser de 16 bits o de 128 bits. Esta función requiere de un único parámetro (el *UUID* del Servicio) y no devuelve nada. Las siguientes Características que se añadan estarán asociadas a este Servicio hasta que se agregue un nuevo Servicio.
- *addCharacteristicb()*: Añade una Característica al Servidor *GATT*, cuyo valor NO debe ser modificado por el Cliente *GATT*. Esta función requiere de cuatro parámetros:
  - *uuid*: Un *UUID* de 16 bits o un *UUID* de 128 bits.
  - *flags*: Propiedad o propiedades de la Característica añadida. Puede ser una combinación de los siguiente valores:
    - *ATT_PROPERTY_BROADCAST* (0x01).
    - *ATT_PROPERTY_READ* (0x02).
    - *ATT_PROPERTY_WRITE_WITHOUT_RESPONSE* (0x04).

        - *ATT_PROPERTY_WRITE* (0x08).
        - *ATT_PROPERTY_NOTIFY* (0x10).
        - *ATT_PROPERTY_INDICATE* (0x20).
        - *ATT_PROPERTY_AUTHENTICATED_SIGNED_WRITE* (0x40).
        - *ATT_PROPERTY_EXTENDED_PROPERTIES* (0x80).
    - *data*: Un puntero de tipo *uint8_t* al valor de la Característica.
    - *data_len*: Longitud máxima del valor de la Característica.

  Esta función devuelve un valor de tipo *uint16_t* que se corresponde con el identificador del atributo del valor de la Característica. La Característica añadida pertenecerá al último Servicio incluido.

- *addCharacteristicDynamic()*: Añade una Característica al Servidor *GATT*, cuyo valor puede ser modificado por el Cliente *GATT*. Los parámetros y el valor que proporciona, son los mismos que para la función *addCharacteristic()*. La Característica añadida pertenecerá al último Servicio incluido.
- *attServerCanSendPacket()*: Comprueba si el dispositivo puede enviar notificaciones o indicaciones al dispositivo al que está conectado, ya que únicamente podrán enviarse en caso de que el Cliente *GATT* haya suscrito la notificación o indicación mediante la llamada a la función *writeClientCharsConfigDescriptor()*. Esta función devuelve un valor de tipo *int* que indica si se pueden enviar notificaciones o indicaciones desde el dispositivo *Peripheral*.
- *sendNotify()*: Notifica al dispositivo al que está conectado que el valor de Característica ha sido modificado. El dispositivo *Peripheral* sólo podrá enviar una notificación en caso de que el Cliente *GATT* haya suscrito la notificación mediante la llamada a la función *ble.writeClientCharsConfigDescriptor()*. Esta función requiere de tres parámetros: el *handle* correspondiente al atributo del valor de la Característica, los datos que se enviarán, y la longitud de los datos. En este caso la longitud máxima de los datos está limitada a 20 bytes.
- *sendIndicate()*: Indica al dispositivo al que está conectado la modificación en el valor de una Característica. El dispositivo *Peripheral* sólo podrá enviar una notificación en caso de que el Cliente *GATT* haya suscrito la notificación mediante la llamada a la función *writeClientCharsConfigDescriptor()*. Esta función requiere de tres parámetros: el *handle* correspondiente al atributo del valor de la Característica, los datos que se enviarán, y la longitud de los datos. La longitud máxima de los datos está limitada en este caso a 20 bytes. Se necesita una respuesta por parte del Cliente *GATT*.
- *onDataReadCallback()*: Registra la función a la que se llamará (función *callback)* cuando el dispositivo al que está conectado lea el valor de la Característica. La función registrada requiere de tres parámetros y devuelve un valor de tipo *uint16_t*:
    - *value_handle*: Identificador de la Característica que se va a leer.
    - *buffer*: Puntero de tipo *uint8_t* al *buffer* que leerá el dispositivo al que está conectado.
    - *buffer_size*: Valor de tipo *uint16_t* que indica la longitud del *buffer*.
- *onDataWriteCallback()*: Registra la función a la que se llamará (función *callback)* cuando el dispositivo al que está conectado escriba el valor de la Característica. La función *callback* toma tres parámetros y devuelve un valor de tipo *int*:

- *value_handle*: Identificador de la Característica que se va a escribir.
- *buffer*: Puntero de tipo *uint8_t* al *buffer* que contiene los datos del dispositivo al que está conectado.
- *buffer_size*: Valor de tipo *uint16_t* que indica la longitud de los datos.

4.4 Desarrollo del código de implementación para la célula de lectura Braille *modul P16* de *metec*

La célula de lectura de código Braille que se va a utilizar necesita que desde el *MCU* se le indique la codificación de qué símbolo va a representar. Es necesario comprobar que lo que le ordena el *RedBear Duo* es lo que finalmente aparece en la célula.

4.4.1 Código de verificación de funcionamiento del *HW* implementado para la célula *modul P16*

Se inicia este apartado teniendo en cuenta que el contenido de éste es meramente una comprobación de que la plataforma *HW* implementado en el Capítulo 2 sea capaz de hacer que la célula reaccione a distintos impulsos de tensión. Para concretar más, este código forzará a que la célula eleve cada uno de sus puntos con la recepción en su *backplane*, por parte del *MCU*, de diferentes caracteres y símbolos.

A continuación, en la Figura 67, se muestra el diagrama de flujo de este proceso de verificación de la plataforma *HW*.

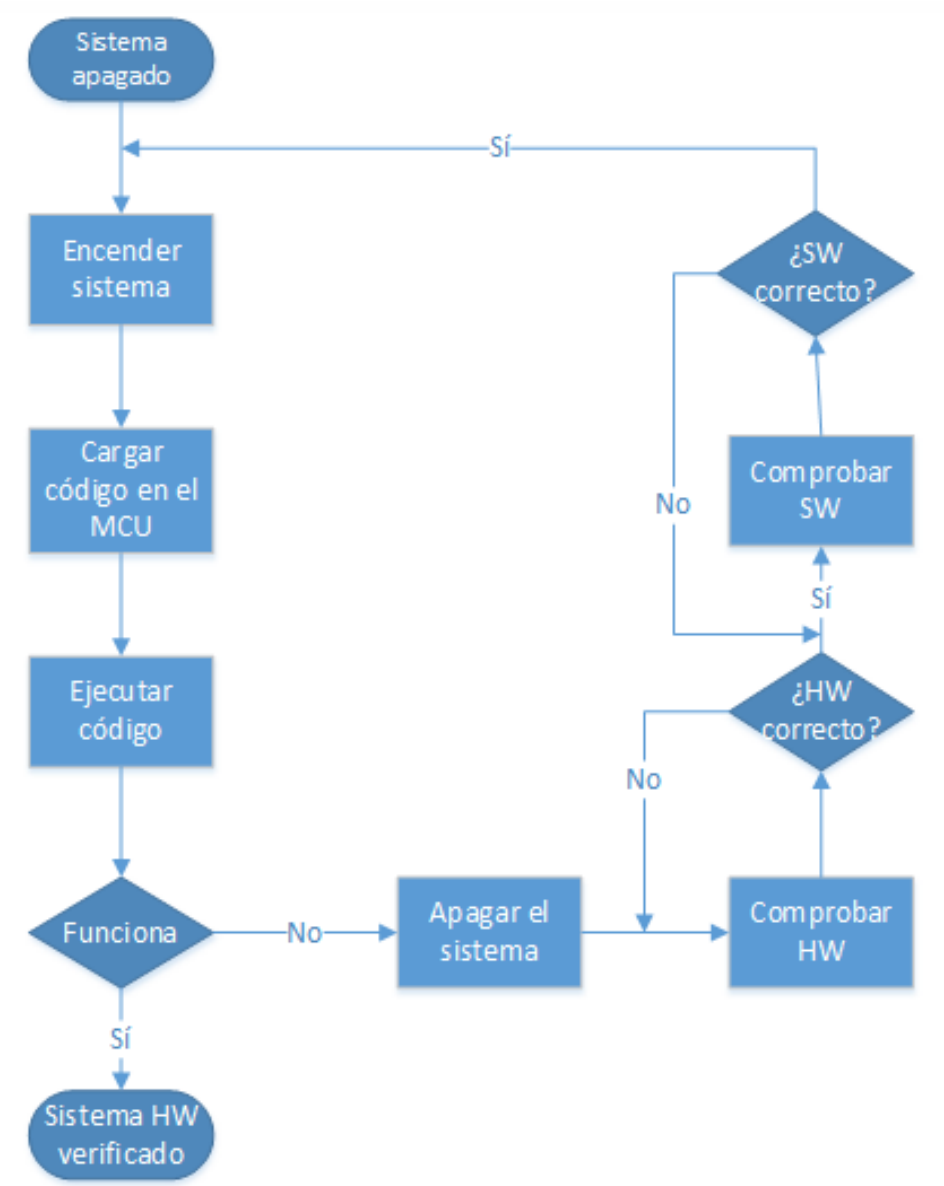

Figura 67. Diagrama de flujo de verificación del *HW* implementado para la célula *modul P16*.

Aclarar que en el diagrama anterior aparece el sistema apagado y es importante que mientras no se utilice, se mantenga apagado. Esto es necesario debido a que se trabajan con tensiones de aproximadamente 200V y que este sistema consume una potencia significativa.

El código que se redacta en este apartado pertenece al fichero *braille-p16-V00.ino.* En el segmento de Código 1 se muestran las definiciones de las diferentes señales utilizadas, así como la declaración de una función. A continuación, se añade la definición de una constante entera que equivaldrá al número de células conectadas al *MCU,* y un vector con el que se accederá a las distintas células de lectura Braille.

Código 1. Definiciones de las entradas y salidas del dispositivo.

```
#define PIN_ON D2      // DC-DC Converter ON Pin. You can use it to switch off
                       //the cell power in a sleeping mode
#define PIN_DATA D3    // Backpanel Pin 2 (Din)
#define PIN_STROBE D4  // Backpanel Pin 3 (STRB)
#define PIN_CLOCK D5   // Backpanel Pin 4 (CLK)

#define bitRead(value, bit) (((value) >> (bit)) & 0x01)

const int cellCount = 1;
byte cells[cellCount];
```

Tras esto aparece la función *setup()*, en la que se incluye la declaración de si los pines del dispositivo *RedBear Duo* definidos anteriormente actúan como entradas o salidas (Código 2). En este caso todos los pines son de salida. Aparte se le asigna al pin correspondiente al *ON* una salida fija a 0, pues es preciso recordar que es necesario que siempre tenga este valor si se desea que el conversor *DC-DC* de 5V a 185V de *metec* entregue dicha tensión a la célula, aunque se podía hacer directamente conectando este pin a *GND*.

Código 2. Función *setup()*.

```
void setup()
{
 pinMode(PIN_ON,     OUTPUT);
 pinMode(PIN_DATA,   OUTPUT);
 pinMode(PIN_STROBE, OUTPUT);
 pinMode(PIN_CLOCK,  OUTPUT);

 digitalWrite(PIN_ON, LOW);  // 0=ON, 1=OFF
}
```

En el segmento de Código 3 se muestra la función *loop()*, correspondiente a un bucle infinito de ejecución de la funcionalidad del *firmware*.

Código 3. Función *loop()*.

```
void loop()
{
 // left cell run all pins, next cell blinks and the others are running horizonal
 // Set the pins, send it and wait...
 cells[0] = 1;   Flush(); Wait();
 cells[0] = 2;   Flush(); Wait();
 cells[0] = 4;   Flush(); Wait();
 cells[0] = 8;   Flush(); Wait();
 cells[0] = 16;  Flush(); Wait();
 cells[0] = 32;  Flush(); Wait();
 cells[0] = 64;  Flush(); Wait();
 cells[0] = 128; Flush(); Wait();
}
```

Sobre estas líneas, en el Código 3 se puede ver cómo se hacen asignaciones de diferntes a la primera celda del vector *cells[]*.Tras realizar cada asignación, se realiza una llamada a las funciones *Flush()* y *Wait()*.

La función *Wait()* mostrada en el Código 4 se corresponde con un *delay* o retardo utilizado para que la reproducción en código Braille sea perceptible y no se modifique

con excesiva velocidad, pudiendo así ser leídos los caracteres que aparecen en la célula de lectura Braille *modul P16*.

Código 4. Función *Wait()*.

```
49  void Wait()
50  {
51   delay(500);
52  }
```

Una vez la función *Flush()* sea llamada desde la función *loop()*, entra en un bucle *for* que, separa cada una de las células del sistema especificado en la variable cellcount, y con cada flanco de reloj, escriben por el pin DATA el valor correspondiente (Código 5).

Código 5. Primera parte de la función *Flush()*.

```
54  // Send the data
55  void Flush ()
56  {
57   // This example is for one P16 backpanel. If you are using others you have
58   // to change the bit order!
59   // P20: 6,7,2,1,0,5,4,3
60   // P16: 6,2,1,0,7,5,4,3
61   // B11: 0,1,2,3,4,5,6,7
62   // Rotate the bytes from the right side. Cell 0 comes first and is on the
63   // left position.
64
65   for (int i = 0; i < cellCount; i++)
66   {
67     if ( bitRead(cells[i], 6) )  {
68       digitalWrite(PIN_DATA, LOW);
69       Serial.println("PIN_DATA 6 LOW");
70       }
71     else {
72       digitalWrite(PIN_DATA, HIGH);
73       Serial.println("PIN_DATA 6 HIGH");
74       }
75     digitalWrite(PIN_CLOCK, HIGH);
76     digitalWrite(PIN_CLOCK, LOW);
```

Las siguientes líneas de código son la continuación del código de la función *Flush()*. Como se muestra en el segmento de Código 6, el pin *STROBE* se activa una vez que se ha escrito el valor de cada uno de los puntos de la célula de *metec* y a continuación se vuelve a desactivar. Estas líneas suponen el refresco o actualización de la célula con los valores que se habían establecidos en el bucle *for* para cada uno de los puntos de la célula. Es decir, esta actualización se efectúa de manera instantánea en todos los puntos de la célula de lectura Braille *modul P16* de *metec*.

Código 6. Última parte de la función *Flush()*.

```
162   digitalWrite(PIN_STROBE, HIGH);  // Strobe on
163   Serial.println("Strobe HIGH");
164   digitalWrite(PIN_STROBE, LOW);  // Strobe off
165   Serial.println("Strobe LOW");
```

Una vez concluida la implementación *SW*, se comprueba que la implementación *HW* realizada para la célula de *metec,* y expuesta en el Capítulo 3, funciona según lo

previsto. Este *SW* fue diseñado sólo para la comprobación del sistema, aun así esta aplicación contiene líneas de código que se reutilizarán para la realización del diseño *software* que responda a los objetivos de este TFG.

4.4.2 *Firmware* de representación de textos en código Braille

Se comienza el diseño software relativo a completar las especificaciones de este TFG. Este código se basará en traducir texto a código Braille, y luego reproducirlo en la célula de lectura Braille *modul P16* de *metec*. Se intentará reutilizar la mayor cantidad de líneas de código posible de la aplicación expuesta en el apartado anterior.

En la Figura 68 se muestra un diagrama de flujo que corresponde a los pasos seguidos para completar los objetivos. Recordar que el sistema debe estar apagado siempre que no esté en uso debido al consumo de potencia y a que trabaja con 185V. Este código se encuentra en el fichero *braille-p16-sin-ble.ino.*

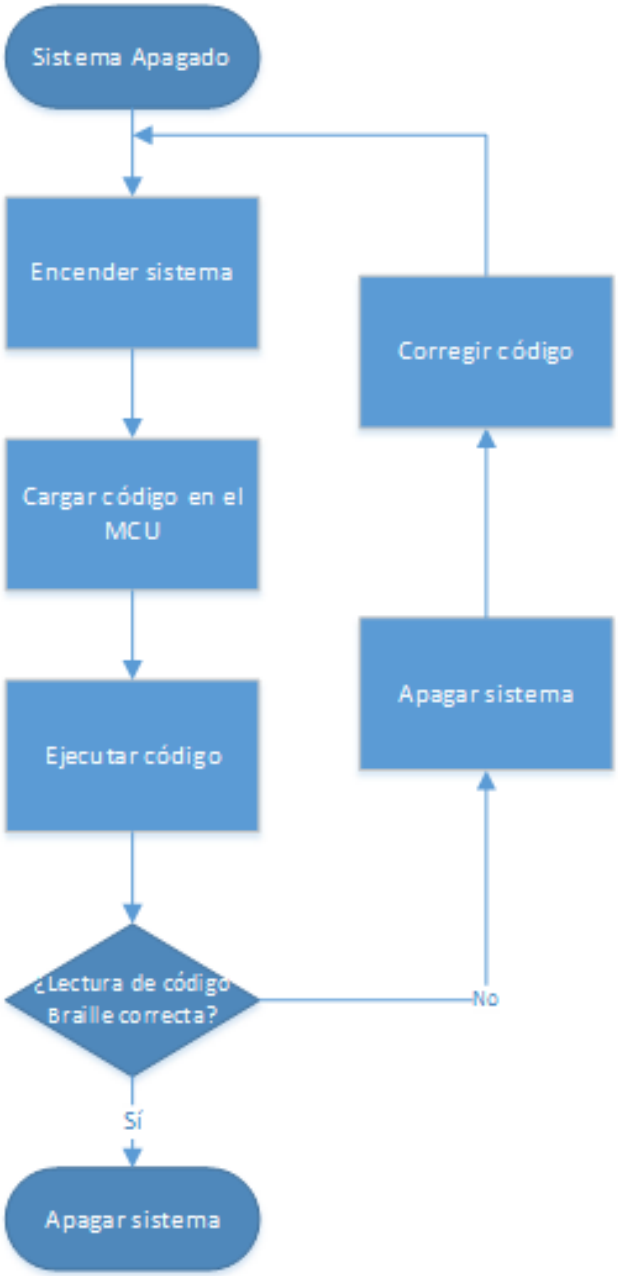

Figura 68.Flujograma de la aplicación sin conexión *BLE*.

En las primeras líneas del código, como se muestra en el Código 7, se realizan las definiciones de las diferentes señales necesarias en este sistema y de una función.

Código 7. Declaraciones de las entradas y salidas del dispositivo.

```
#define PIN_ON D2       // DC-DC Converter ON Pin. You can use it to switch off
                        // the cell power in a sleeping mode
#define PIN_DATA D3     // Backpanel Pin 2 (Din)
#define PIN_STROBE D4   // Backpanel Pin 3 (/STRB)
#define PIN_CLOCK D5    // Backpanel Pin 4 (CLK)
#define BUTTONPIN D6    // External Button. You can use it to update the Braille cells.
#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
```

Como recordatorio del funcionamiento de las distintas señales, en las siguientes líneas se vuelven a redefinir:

- *ON*: El pin que activa el conversor *DC-DC* de 5V a 185V. Este conversor se encarga de dar alimentación al *backplane* de la célula *modul P16*.
- *DATA*: El pin por el que se envía en serie, el valor correspondiente a cada uno de los puntos de la célula de *metec*.
- *STROBE*: Refresco de los valores de los puntos de la célula Braille a la vez.
- *CLOCK*: En cada flanco de subida de reloj, actualiza un dato de uno de los puntos de la célula.
- *BUTTONPIN*: Este pin corresponde a un dato de entrada del sistema. Se trata de un pulsador que cuando es accionado el sistema refresca el valor de las células.

Las siguientes líneas se corresponden con la generación de las distintas variables, vectores y constantes (Código 8).

Código 8: Declaraciones de variables.

```
const int cellCount = 3;
byte cells[cellCount];

char test[] = {"holloh"};
int test_idx = 0;
int cells_offset = 0;
const int charCount = 128;
byte charcells[charCount];
byte charcells_MAY;
byte charcells_NUM;

bool buttonAlreadyPressed;
bool enable_char;
int cells_index;
bool cells_prefix;
char char_tmp;
```

Sobre estas líneas y por orden, aparecen las distintas variables de este código y que es necesario definir para mayor compresión del código desarrollado:

- *cellCount:* Valor constante que modifica el programador de acuerdo con la cantidad de células de lectura Braille disponibles.
- *cells[]:* Vector que asigna a cada célula una posición en sus celdas.
- *test[]:* Vector de datos del cual se va a realizar la traducción en código Braille para su posterior reproducción en las células de *metec*.
- *test_indx:* Índice utilizado para recorrer el vector anterior.
- *cells_offset:* Esta variable se encarga de añadir el *offset* necesario para que el índice que marque el vector *charcells[]* coincida con el código ASCII del símbolo que se desea representar en código Braille.
- *charCount:* Constante que tiene un valor entero de 128.

- *charcells[]:* Vector que se encarga de la traducción de los caracteres del texto, a código Braille. Este vector posee un valor constante de celdas dispuesto por el valor de *charCount*.
- *charcells_MAY:* Variable de tipo byte, que será la primera parte de los caracteres en mayúscula.
- *charcells_NUM:* Variable de tipo byte, que incluye la primera parte de los caracteres numéricos.
- *buttonAlreadyPressed:* Variable que comprueba si el botón asociado al pin 6 del *MCU* ha sido pulsado, y en consecuencia fuerza una actualización de la célula de lectura Braille *modul P16*.
- *enable_char:* Variable booleana que indica si existe un carácter a representar.
- *cells_index:* Índice que recorre el vector *cells[]*.
- *cells_prefix:* Antes de explicar esta variable es importante saber que cuando se quiere reproducir en la célula Braille caracteres en mayúscula o numéricos, es necesario completarlo con un primer carácter que indica si se trata de una mayúscula o un número, y el segundo reproduciendo la conversión del código que representa ese valor. Esta variable es la encargada de forzar la reproducción del código que representa a las mayúsculas o los números, antes que el código que representa el valor del carácter que se quiere mostrar en la célula Braille *modul P16*.
- *char_tmp:* Variable que asume el valor del carácter a reproducir en cada momento.

La función *setup()*, mostrada en el Código 9, es la encargada de inicializar las variables y los pines, y declarar si los pines son de entrada o salida. Dentro de este código también se hace la llamada a la función *init_charcells()*.

Código 9. Función *setup()*.

```
27  void setup()
28  {
29   pinMode(PIN_ON,     OUTPUT);
30   pinMode(PIN_DATA,   OUTPUT);
31   pinMode(PIN_STROBE, OUTPUT);
32   pinMode(PIN_CLOCK,  OUTPUT);
33
34   digitalWrite(PIN_ON, LOW);  // 0=ON, 1=OFF
35   digitalWrite(PIN_DATA, LOW);
36   digitalWrite(PIN_STROBE, LOW); // 0=ON, 1=OFF
37   digitalWrite(PIN_CLOCK, LOW);
38
39   test_idx = 0;
40   cells_offset = 0;
41   init_charcells();
42
43   buttonAlreadyPressed = false;
44   enable_char = false;
45   cells_index = 0;
46   cells_prefix = false;
47
48   Serial.print("sizeof(test) = ");
49   Serial.println(sizeof(test));
50  }
```

En el Código 10 se puede ver la primera parte de la función *loop()*. En su primera línea se crea una variable con el valor que contiene el pin *BUTTONPIN*, asociado al valor del pulsador, y con el cual se actualizaría la célula de lectura Braille. Lo siguiente es una condición enmarcada en otra condición que si se cumplen ambas se activa la variable que establece que hay un carácter preparado para actualizar la célula, estableciéndose que el índice que recorrerá el vector *cells[]* esté a cero. Seguido se le asigna a la variable *buttonAlreadyPressed* el valor de la variable local antes inicializada.

Código 10. Primera parte de la función *loop()*.

```
53  void loop()
54  {
55
56     bool buttonPressed = digitalRead(BUTTONPIN);
57     if (buttonPressed && !buttonAlreadyPressed) {
58       if (test_idx < sizeof(test)) {
59         enable_char = true;
60       }
61       cells_index = 0;
62     }
63
64     buttonAlreadyPressed = buttonPressed;
```

La segunda parte del código se corresponde con una jerarquía de condicionales (Código 11). La condición de mayor categoría comprueba si existe algún carácter preparado para actualizar la célula. La siguiente condición comprara si el índice de *cells[]* es menor que la cantidad de células conectadas al *backplane*. Dentro de esta condición existen dos sentencias nuevas:

- La primera de ellas comprueba si el índice del vector *test[]* coincide con el tamaño del vector. Si es así, añade un carácter vacío, mientras que si por el contrario no se cumple dicha restricción, la variable *char_tmp* adquiere el valor contenido en el vector *test[]* en la posición marcada por el índice actual.
- El siguiente condicional tiene tres ventanas de análisis, dado que en la primera comprueba si el carácter está en mayúscula, el segundo si se trata de un número, y en el tercer caso si responde a cualquier otro carácter. Los códigos asociados a las mayúsculas y a los números son casi iguales, pues primero se asigna el valor de las mayúsculas o números, a continuación se establece el valor de *offset,* necesario para realizar correctamente la conversión posterior en el *array charcells[]*, y se finaliza estableciendo el valor de la variable encargada de forzar la reproducción de estos valores antes de la traducción del resto del valor. Como se explicó con anterioridad cuando existen mayúsculas o números, se necesita que la célula se actualice dos veces para representar el valor correcto de la traducción en código Braille. El último caso ofrece una condición que devuelve el *offset* al valor cero. Aparte, asigna el valor de *charcells[]* contenido en la celda resultante de la suma de *char_tmp* y

cells_offset, al vector *cells[]*. Luego establece a *false* el valor de *cell_prefix*. Por último, comprueba que el valor del índice del vector *test[]* es inferior que el tamaño del mismo, y si se cumple aumenta en uno el valor de dicho índice.

Código 11. Segunda parte de la función *loop()*.

```
66    if (enable_char) {
67       if (cells_index < cellCount) {
68          if (test_idx == sizeof(test)) {
69             char_tmp = ' ';
70          }
71          else {
72             char_tmp = test[test_idx];
73          }
74
75          if ((!cells_prefix) && (char_tmp >= 'A') && (char_tmp <= 'Z')) {
76             Serial.println(charcells_MAY, HEX);
77             cells[cells_index++] = charcells_MAY;
78             cells_offset = 32;
79             cells_prefix = true;
80          }
81          else if ((!cells_prefix) && (char_tmp >= '0') && (char_tmp <= '9') ) {
82             Serial.println(charcells_NUM, HEX);
83             cells[cells_index++] = charcells_NUM;
84             cells_offset = 0;
85             cells_prefix = true;
86          }
87          else {
88             if (!cells_prefix)
89                cells_offset = 0;
90
91             Serial.println(charcells[char_tmp+cells_offset], HEX);
92             cells[cells_index++] = charcells[char_tmp+cells_offset];
93             cells_prefix = false;
94             if (test_idx < sizeof(test))
95                test_idx++;
96          }
97       }
```

La última parte de la función *loop()*, la cual se puede ver en el Código 12, muestra el *else* correspondiente a la segunda condición en jerarquía. Este código asigna el valor *false* a la variable que determina si existe un carácter preparado para actualizar la célula de lectura en código Braille de *metec* y posteriormente realiza una llamada a la función *Flush()*. Tras esto, se recorre el vector *cells[]* asignándole el valor 0, y por último, se realizan dos llamadas a la función *Flush()* con una entrada de 1, es decir, dos desplazamientos de registro. Este código es necesario debido a que el *backplane* trabaja con un vector de 6 celdas y el *MCU* lo hace con uno de 8 celdas.

Código 12. Última parte de la función *loop()*.

```
98       else {
99         enable_char = false;
100        Flush(cellCount);
101
102        for (int i = 0; i < cellCount; i++)
103           cells[i] = 0x00;
104        Flush(1);
105        Flush(1);
106      }
```

Las funciones *Wait()* y *Flush()* coinciden con las del código descrito en el apartado 4.4.1, y en consecuencia no se explican en este apartado.

Para concluir con este diseño *SW*, queda analizar la función *init_charcells()*(Código 13). Esta función únicamente preestablece valores de traducción de diferentes caracteres y símbolos en código Braille a las variables *charcells_MAY* y *charcells_NUM*, y al vector *charcells[]*. El índice de este *array* es el código *ASCII* (*American Standard Code for Information Interchange*) del símbolo a representar.

Código 13. Función *init_charcells()*.

```
231  void init_charcells()
232  {
233   charcells_MAY = 0x28;
234   charcells_NUM = 0x3C;
235
236   charcells['a'] = 0x01;
237   charcells['b'] = 0x03;
238   charcells['c'] = 0x09;
239   charcells['d'] = 0x19;
240   charcells['e'] = 0x11;
241   charcells['f'] = 0x0B;
242   charcells['g'] = 0x1B;
243   charcells['h'] = 0x13;
244   charcells['i'] = 0x0A;
245   charcells['j'] = 0x1A;
246   charcells['k'] = 0x05;
247   charcells['l'] = 0x07;
248   charcells['m'] = 0x0D;
249   charcells['n'] = 0x1D;
250   charcells['ñ'] = 0x3B;
251   charcells['o'] = 0x15;
252   charcells['p'] = 0x0F;
253   charcells['q'] = 0x1F;
254   charcells['r'] = 0x17;
255   charcells['s'] = 0x0E;
256   charcells['t'] = 0x1E;
257   charcells['u'] = 0x25;
258   charcells['v'] = 0x27;
259   charcells['w'] = 0x3A;
260   charcells['x'] = 0x2D;
261   charcells['y'] = 0x3D;
262   charcells['z'] = 0x35;
263   charcells['0'] = 0x1A;
264   charcells['1'] = 0x01;
265   charcells['2'] = 0x03;
266   charcells['3'] = 0x09;
```

Para entender las líneas del segmento de código anterior, se va a exponer un ejemplo con el carácter ‘a’. El array charcells[‘a’] tiene un valor de 97 en decimal, dado a que trabaja con el código *ASCII*. A esta celda del array se le asigna el valor de conversión en código Braille (0x01 en hexadecimal), de tal manera que en la célula de lectura Braille modul P16 quede reflejado el valor de ‘a’. En la Figura 69 se puede ver la conversión descrita, donde el color naranja muestra los puntos que están en relieve, y el color blanco los botones que están escondidos.

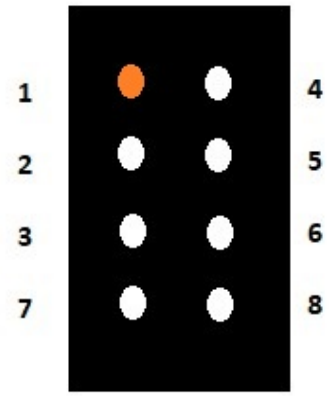

Figura 69. Representación en código Braille del carácter 'a' en la célula *modul P16* de *metec*.

4.4.3 Implementación de la funcionalidad del dispositivo Peripheral para la plataforma RedBear Duo a partir del código de usuario

En este TFG se ha utilizado el *firmware* proporcionado por *RedBear Duo* como base para la implementación del funcionamiento de un dispositivo *Peripheral*. Éste a su vez realiza el rol de *GATT Server* definido en *BLE*. Se enviaran paquetes de datos de un tamaño máximo de 20 bytes. Los datos serán números, caracteres en mayúscula y/o minúscula o espacios en blanco, enviados como texto desde el dispositivo *Central*. El archivo que contiene el código desarrollado se denomina *BLEPeripheral_Duo.ino*. En la Figura 70 se puede ver el diagrama de flujo la funcionalidad del firmware desarrollado en este TFG.

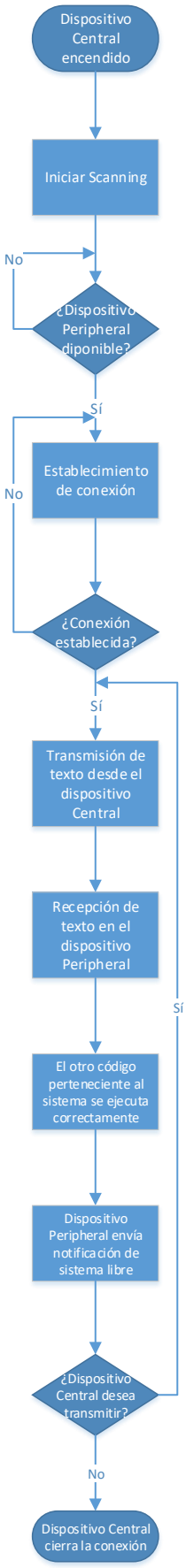

Figura 70. Diagrama de flujo de la conexión *BLE*.

Se comienza el código estableciendo el tiempo mínimo y máximo del parámetro correspondiente al Intervalo de Conexión en el dispositivo *Peripheral*, a 50 ms y 500 ms respectivamente. En todo caso, el dispositivo *Central* es quien impone el Intervalo de Conexión en el establecimiento de la conexión. Así mismo, se define el tiempo de latencia del dispositivo *Slave* a 0 ms, el tiempo de espera de supervisión de la conexión a 10 s, y el valor del parámetro *appearance*. También se define el nombre que se ha decidido establecer para el dispositivo *BLE Peripheral*, en este caso "BLE_Peripheral". Todo esto se muestra en el segmento de Código 14.

Código 14. Definiciones de diferentes parámetros de conexión y del nombre del dispositivo.

```
29  #define MIN_CONN_INTERVAL           0x0028 // 40 * 1.25ms = 50ms.
30  #define MAX_CONN_INTERVAL           0x0190 // 400 * 1.25ms = 500ms.
31  #define SLAVE_LATENCY               0x0000 // No slave latency.
32  #define CONN_SUPERVISION_TIMEOUT    0x03E8 // 1000 * 10ms = 10s.
33
34  #define BLE_PERIPHERAL_APPEARANCE   BLE_APPEARANCE_UNKNOWN
35
36  #define BLE_DEVICE_NAME             "BLE_Peripheral"
```

Seguido es definido el *UIDD* perteneciente a un Servicio y a dos Características del dispositivo *Peripheral*, mostradas en el Código 15. La Característica char1_uuid será utilizada principalmente para escribir desde el dispositivo Central, texto plano que luego se reproducirá en la célula de lectura Braille de metec, aunque también estará programada para hacer lectura, mientras que la Característica char2_uuid será la que notificará al dispositivo Central cuando el dispositivo Peripheral queda listo para recibir más texto. Como en la otra Característica, char2_uuid también puede realizar lectura.

Código 15. Definición del Servicio y Características propias del dispositivo *Peripheral*.

```
54  // Primary service 128-bits UUID
55  static uint8_t servicel_uuid[16] = { 0x71,0x3d,0x00,0x00,0x50,0x3e,0x4c,0x75,
56                                       0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
57  // Characteristics 128-bits UUID
58  static uint8_t charl_uuid[16]    = { 0x71,0x3d,0x00,0x02,0x50,0x3e,0x4c,0x75,
59                                       0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
60  static uint8_t char2_uuid[16]    = { 0x71,0x3d,0x00,0x03,0x50,0x3e,0x4c,0x75,
61                                       0xba,0x94,0x31,0x48,0xf1,0x8d,0x94,0x1e };
```

En el Código 16 se detallan los distintos parámetros asociados al proceso de *Advertising*. Los intervalos mínimo y máximo son de 30 ms en ambos casos, del tipo *AVD_IND*, con una dirección de tipo pública (*PUBLIC*), la dirección es la 0, el parámetro *channel map* estará en *ALL*, y el parámetro *filter policy* en *ANY*.

Código 16. Parámetros del proceso *Advertising*.

```
107  static advParams_t adv_params = {
108    .adv_int_min   = 0x0030, //48 * 0.625ms = 30ms
109    .adv_int_max   = 0x0030, //48 * 0.625ms = 30ms
110    .adv_type      = BLE_GAP_ADV_TYPE_ADV_IND,
111    .dir_addr_type = BLE_GAP_ADDR_TYPE_PUBLIC,
112    .dir_addr      = {0,0,0,0,0,0},
113    .channel_map   = BLE_GAP_ADV_CHANNEL_MAP_ALL,
114    .filter_policy = BLE_GAP_ADV_FP_ANY
115  };
```

A continuación, se especifican los datos relacionados con el proceso Advertising, mostrados en el Código 17. El primer parámetro es requerido para dispositivos *BLE* y el segundo parámetro coincide con el *UUID* del Servicio definido en el dispositivo *Peripheral*.

Código 17. Datos de *Advertising* en el dispositivo Peripheral.

```
118  static uint8_t adv_data[] = {
119    0x02,
120    BLE_GAP_AD_TYPE_FLAGS,
121    BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE,
122
123    0x11,
124    BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE,
125    0x1e, 0x94, 0x8d, 0xf1, 0x48, 0x31, 0x94, 0xba,
126    0x75, 0x4c, 0x3e, 0x50, 0x00, 0x00, 0x3d, 0x71
127  };
```

Posteriormente se añade el parámetro *Complete Local Name,* el cual aparecerá como BRAILLE (Código 18). Éste establece los datos de respuesta al proceso *Scan*.

Código 18. Datos del *Scan Response*.

```
130  static uint8_t scan_response[] = {
131    0x08,
132    BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME,
133    'B', 'R', 'A', 'I', 'L', 'L', 'E'
134  };
```

En el código implementado, en relación a las funciones *callback*, se concreta la función *deviceConnectedCallback()*, que admite como parámetros de entrada el estado y el identificador de la conexión. Este identificador es asignado en caso de que se establezca la conexión, como se puede ver en el Código 19. Esta función será invocada por *onConnectedCallback()* cuando se establezca su conexión con el dispositivo *BLE Peripheral*.

Código 19. Función *deviceConnectedCallback()*.

```
192  void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
193    switch (status) {
194      case BLE_STATUS_OK:
195        Serial.println("Device connected!");
196        connected_id = handle;
197        Serial.print("  - Device connected handle: ");
198        Serial.println(connected_id);
199        break;
200      default: break;
201    }
202  }
```

A continuación se define la función *deviceDisconnectedCallback()*, la cual acepta como parámetro de entrada el identificador de la conexión, tal y como se muestra en el Código 20. Esta función será llamada por *onDisconnectedCallback()* en el momento en el que se finalice la conexión *BLE*.

Código 20. Función *deviceDisconnectedCallback()*.

```
211  void deviceDisconnectedCallback(uint16_t handle) {
212    Serial.println("Device disconnected!");
213    connected_id = 0xFFFF;
214  }
```

A continuación, en el Código 21, se declara la función *gattReadCallback(),* que es la encargada de leer el valor de las Características definidas, de llenar la variable buffer con el texto recibido, y devolver la longitud del valor contenido en la Característica. Esta función es invocada por *onDataReadCallback()*.

Código 21. Función *gattReadCallback().*

```
/**
 * @brief Callback for reading event.
 *
 * @note  If characteristic contains client characteristic configuration,then client
 *        characteristic configration handle is value_handle+1.Now can't add user_descriptor.
 * @param[in]  value_handle
 * @param[in]  buffer
 * @param[in]  buffer_size    Ignore it.
 *
 * @retval  Length of current attribute value.
 */
uint16_t gattReadCallback(uint16_t value_handle, uint8_t * buffer, uint16_t buffer_size) {
  uint8_t characteristic_len = 0;
  Serial.print("Read value handler: ");
  Serial.println(value_handle, HEX);
  if (character1_handle == value_handle) {
    Serial.print("  - characteristic1 read: ");
    memcpy(buffer, characteristic1_data, characteristic1_data_len);
    characteristic_len = CHARACTERISTIC1_MAX_LEN;
    for (uint8_t index = 0; index < CHARACTERISTIC1_MAX_LEN; index++) {
      Serial.print(buffer[index], HEX);
    }
    Serial.println();
  }
  else if (character2_handle == value_handle) {
    Serial.print("  - characteristic2 read: ");
    memcpy(buffer, characteristic2_data, CHARACTERISTIC2_MAX_LEN);
    characteristic_len = CHARACTERISTIC2_MAX_LEN;
    for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN; index++) {
      Serial.print(buffer[index], HEX);
    }
    Serial.println();
  }
  return characteristic_len;
}
```

Finalizando con las funciones de *callback*, se define *gattWriteCallback()*, la cual es invocada por la función *onDataWriteCallback()*. Esta función es llamada cuando el dispositivo escribe el valor de una de las Características definidas. Haciendo un recorrido por las líneas de código que se muestran en el Código 22, se observa una condición que evalúa si el valor del identificador de la Característica *character1* coincide con el identificador del valor de la característica escrita, además de que el valor de *new_data* y de *data_on* estén a *false*. Si esto ocurre, el valor de *new_data* pasa a ser *true* debido a que se habrá recibido un nuevo valor de texto a representar, mostrando en el terminal serie el texto recibido. Si por el contrario no entra en esa condición, intentará entrar en la que comprueba si el identificador de la Característica *character2* más 1 coincide con el valor del identificador de *CCCD*. Si consigue acceder a esta condición, mostrará por el puerto serie el valor del parámetro *CCCD,* y posteriormente el contenido almacenado en el buffer. Esta función siempre devuelve el valor entero 0.

Código 22. Función *gattWriteCallback()*.

```
/**
 * @brief Callback for writting event.
 *
 * @param[in]  value_handle
 * @param[in]  *buffer       The buffer pointer of writting data.
 * @param[in]  size          The length of writting data.
 *
 * @retval
 */
int gattWriteCallback(uint16_t value_handle, uint8_t *buffer, uint16_t size) {
  Serial.print("Write value handler: ");
  Serial.println(value_handle, HEX);

  if ((characterl_handle == value_handle) && ((new_data == false) && (data_on == false))) {
    new_data = true;
    characteristicl_data_len = size;
    memcpy(characteristicl_data, buffer, size);
    Serial.print("  - characteristicl write value: ");
    for (uint8_t index = 0; index < size; index++) {
      Serial.print((char) characteristicl_data[index]);
    }
    Serial.println(" ");
  }
  else if (character2_handle+1 == value_handle) {   // Client Characteristic Configuration Descriptor Handle.
    Serial.print("  - characteristic2 CCCD write value: ");
    for (uint8_t index = 0; index < size; index++) {
      Serial.print(buffer[index], HEX);
    }
    Serial.println(" ");
  }
  return 0;
}
```

Si se analiza la función *setup()* del código desarrollado para un dispositivo BLE *Peripheral*, en el Código 23 se puede ver la inicialización de la interfaz *HCI* entre el *host* y el controlador; al igual que en el Código 24 se muestra el registro de las funciones de *callback*; la incorporación de los Servicios y Características del *GAP* (Código 25); así como la incorporación de los Servicios y Características del *GATT Server*, mostrados en el Código 26.

Código 23. Inicialización de la interfaz *HCI*.

```
  // Initialize ble_stack.
  ble.init();
```

Código 24. Registro de las funciones de *callback*.

```
  // Register BLE callback functions.
  ble.onConnectedCallback(deviceConnectedCallback);
  ble.onDisconnectedCallback(deviceDisconnectedCallback);
  ble.onDataReadCallback(gattReadCallback);
  ble.onDataWriteCallback(gattWriteCallback);
```

Código 25. Servicios y Características del *GAP*.

```
  // Add GAP service and characteristics
  ble.addService(BLE_UUID_GAP);
  ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_DEVICE_NAME, ATT_PROPERTY_READ|ATT_PROPERTY_WRITE,
                        (uint8_t*)BLE_DEVICE_NAME, sizeof(BLE_DEVICE_NAME));
  ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_APPEARANCE, ATT_PROPERTY_READ, appearance, sizeof(appearance));
  ble.addCharacteristic(BLE_UUID_GAP_CHARACTERISTIC_PPCP, ATT_PROPERTY_READ, conn_param, sizeof(conn_param));
```

Código 26. Servicios y Características del *GATT Server*.

```
311  // Add GATT service and characteristics
312  ble.addService(BLE_UUID_GATT);
313  ble.addCharacteristic(BLE_UUID_GATT_CHARACTERISTIC_SERVICE_CHANGED, ATT_PROPERTY_INDICATE,
314                        change, sizeof(change));
315
316  // Add primary service1.
317  ble.addService(service1_uuid);
318
319  // Add characteristic to service1, return value handle of characteristic.
320  character1_handle = ble.addCharacteristicDynamic(char1_uuid, ATT_PROPERTY_READ|ATT_PROPERTY_WRITE
321                                                   characteristic1_data, CHARACTERISTIC1_MAX_LEN);
322  character2_handle = ble.addCharacteristicDynamic(char2_uuid, ATT_PROPERTY_READ|ATT_PROPERTY_NOTIFY,
323                                                   characteristic2_data, CHARACTERISTIC2_MAX_LEN);
```

A tener en cuenta que la Característica *char1_uuid* posee las propiedades de lectura (*ATT_PROPERTY_READ*) y de escritura (*ATT_PROPERTY_WRITE*) y la Característica *char2_uuid* tiene propiedades de lectura (*ATT_PROPERTY_READ*) y de notificación (*ATT_PROPERTY_NOTIFY*). La primera Característica es la que se utilizará para el envío de texto desde el dispositivo *BLE Central* y la recepción de dichos datos en el dispositivo *Peripheral*; mientras que la segunda Característica será utilizada para el envío de notificaciones desde el dispositivo *BLE Peripheral*.

En la función *setup()* también se establecen los parámetros y los datos del proceso de *Advertising* (Código 27); y en el Código 28, se realiza la llamada a la función *ble.startAdvertising()*, que es la encargada de hacer que se inicie el proceso de *Advertising*.

Código 27. Parámetros y datos del proceso de *Advertising*.

```
326  // Set BLE advertising parameters
327  ble.setAdvertisementParams(&adv_params);
328
329  // Set BLE advertising and scan respond data
330  ble.setAdvertisementData(sizeof(adv_data), adv_data);
331  ble.setScanResponseData(sizeof(scan_response), scan_response);
```

Código 28. Inicialización del proceso *Advertising*.

```
333  // Start advertising.
334  ble.startAdvertising();
335  Serial.println("BLE peripheral start advertising");
```

Por último, en el código *loop()*, el cual ha sido explicado en el apartado 4.4.2, al finalizar de traducir el texto y reproducirlo en código Braille en la célula de lectura *modul P16*, se enviará una notificación al dispositivo *BLE Central*, siempre que se cumpla la condición de que el índice del vector recorrido coincida con el valor de su tamaño. En el Código 29 se presenta únicamente las líneas de esta petición, ya que el resto del código ha sido especificado en el apartado anterior.

Código 29. Envío de notificación desde el dispositivo *Peripheral* dentro del *loop()*.

```
106         if (test_idx == test_len){
107             Serial.print("Send characteristic2_notify: ");
108             ble.sendNotify(character2_handle, characteristic2_data, CHARACTERISTIC2_MAX_LEN);
109             for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN; index++) {
110                 Serial.print(characteristic2_data[index], HEX);
111             }
112             characteristic2_data[CHARACTERISTIC2_MAX_LEN-1]++;
113             Serial.println(" ");
114             test_idx = 0;
115             data_on = false;
116         }
```

4.5 Resultados obtenidos

Antes de presentar los resultados obtenidos, es necesario comentar que la unión de los dos códigos anteriormente mencionados en los apartados 4.4.2 y 4.4.3, sólo genera un cambio importante que mencionar, y es que el vector que en el apartado 4.4.2 se establece como predeterminado, para comprobar que el *MCU* traduce a código Braille texto plano y lo reproduce en la célula *modul P16* de *metec*, y al unir los códigos se crea sin valores debido a que luego se llena con el contenido del *buffer* de la transmisión en *BLE* descrito en el apartado 4.4.3.

Como resolución se ha obtenido una implementación *HW/SW* que es capaz de transferir texto desde un dispositivo móvil, actuando como dispositivo *BLE Central*, al *RedBear* Duo, que fue programado como dispositivo *BLE Peripheral*. Aparte, este último dispositivo también actuaba de *MCU* del resto del sistema, consiguiendo reproducir con exactitud el texto en código Braille en una plataforma *hardware* diseñada para la célula de lectura Braille *modul P16* de *metec*.

El código desarrollado está preparado para que una vez se establezca la cantidad de células, que se conectarán al sistema y modificando la variable que se asocia a estas, se reproduzca en cada una de estas células un carácter. También se ha conseguido que por medio de un pulsador las células se actualicen automáticamente, reproduciendo los siguientes caracteres del texto recibido vía *BLE*.

La implementación HW/SW basada en la célula de metec ha cubierto todas las necesidades que se puedan requerir, ya sea la posibilidad de recibir texto vía BLE, como conseguir que dicho texto sea representado correctamente en código Braille, y sin recurrir a excesiva potencia ni elevar demasiado el coste con estos dispositivos. Esto demuestra que existe un sobrecoste en los dispositivos que actualmente se encuentran a disposición de los discapacitados visuales.

Capítulo 5. Diseño e implementación de una célula de lectura Braille basada en motores *push-pull*

5.1 Introducción

Se pretende realizar la fabricación de un sistema de lectura Braille de bajo coste. En este capítulo se realiza un diseño con motores *push-pull*. Con este diseño se pretende demostrar la posibilidad de que existan dispositivos competentes de bajo coste y que además consuman poca energía.

Para plantear este problema hay que buscar las características de trabajo de los motores *push-pull*, pues es a partir de ellos de donde se intentará conseguir diseñar una plataforma hardware que sea capaz de reproducir textos en código Braille.

El problema principalmente planteado en este capítulo es la descripción de todos los componentes que puedan conseguir realizar este diseño. Para que esto se cumpla, es imprescindible desglosar cada componente finalmente usado, de tal manera que se demuestre que es el componente idóneo para la tarea encomendada. En el apartado 2.2 se encuentra el desglose del dispositivo *RedBear Duo*, que al ser el MCU utilizado en ambos sistemas, se decide no volver a desarrollar sus características.

5.2 Diseño y montaje de una célula Braille de lectura con motores *push-pull*

Como se ha mencionado con anterioridad, este capítulo ira mostrando los diferentes componentes que se utilizaran en un montaje posterior para la lectura Braille utilizando motores *push-pull*. Puesto que inicialmente se desconoce qué será necesario para ello, en la Figura 71 se muestran los pasos que inicialmente se han seguido en este TFG.

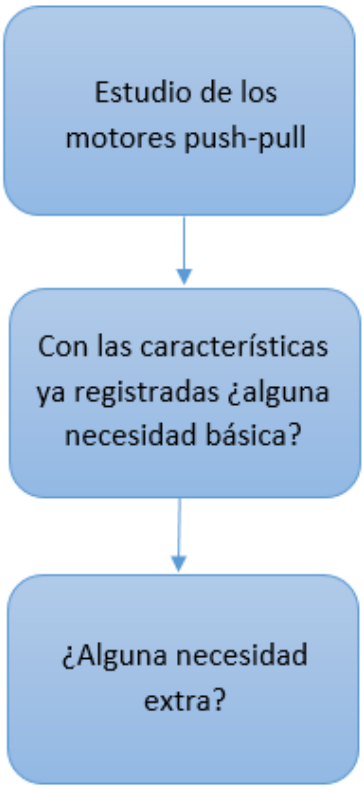

Figura 71. Diagrama de flujo de la metodología a seguir

5.2.1 Motores *push-pull*

Los motores *push-pull* son electroimanes hechos de una bobina de alambre de cobre con una pieza metálica central (Figura 72). Cuando se energiza la bobina, se genera un campo electromagnético capaz de tirar o empujar dicha pieza metálica.

Figura 72. Motor *push-pull*.

Los motores *push-pull* son componentes pasivos. Sus especificaciones mecánicas son las mostradas en la Tabla 5. 1, la cuáles fueron obtenidas en su *datasheet [18]*.

Tabla 5. 1. Especificaciones mecánicas de los motores *push-pull*.

Tensión de continua	5V
Corriente	0.42A
Potencia	2W
Resistencia en continua	12Ω ±5%
Clase de aislamiento	B

A continuación, en la Figura 73, se puede ver las diferentes dimensiones de este motor. La segunda imagen se corresponde con el motor excitado, y la tercera con el motor en reposo.

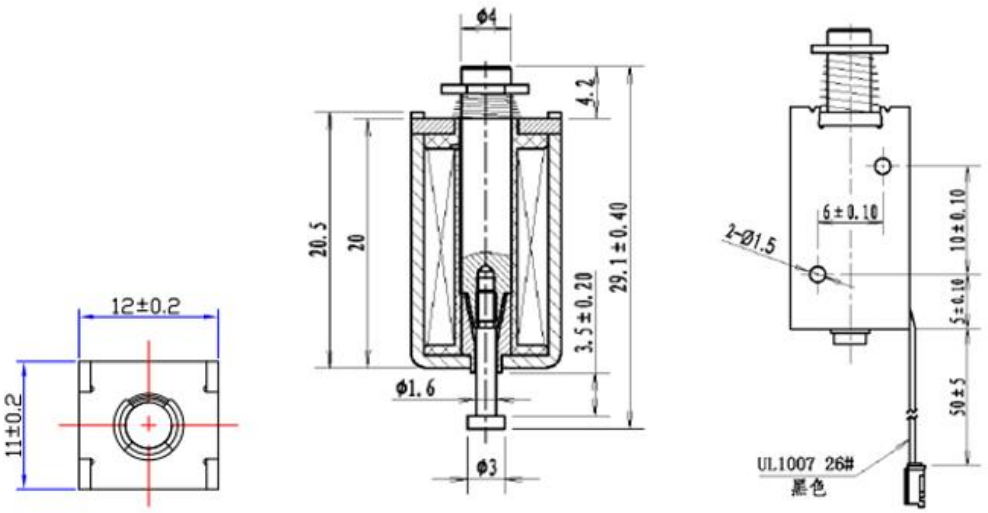

Figura 73. Dimensiones de los motores push-pull.

5.2.2 74HC595

El 74HC595 [12] es un registro de desplazamiento de salida paralelo y de entrada en serie de 8 bits que alimenta un registro de almacenamiento de tipo D de 8 bits. Este dispositivo es necesario debido a que los datos que transmite el *RedBear Duo* están en serie y para la reproducción de los motores es necesario que estén en paralelo. En la Figura 74 y en la Figura 75 se muestra este conversor.

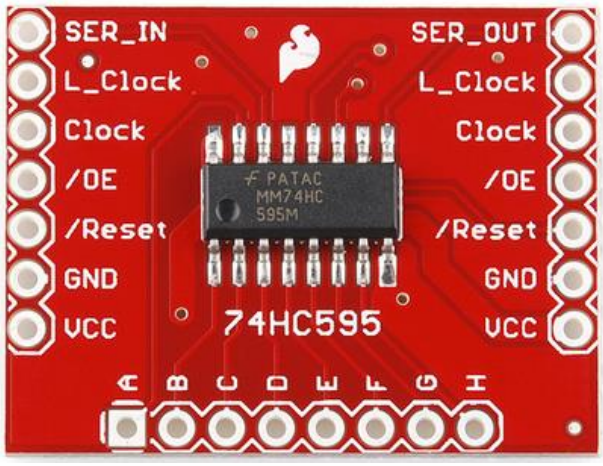

Figura 74. *Top layer* del conversor 74HC595 de *SparkFun.*

Figura 75. *Bottom layer* del conversor 74hc595 de *SparkFun*.

En la Tabla 5. 2 siguiente se muestra el pinout de este dispositivo *hardware*. Como se puede ver en la Figura 74, hay pines que se duplican. Esto es posible porque este conversor es capaz de enchufar en serie tantos dispositivos como se desee, obteniendo de cada uno de ellos 8 salidas de datos en paralelo.

Tabla 5. 2. *Pinout* del conversor 74hc595 de *SparkFun*.

Nombre del Pin	Descripción
SER_IN	Entrada de datos en serie
L_CLOCK	Registro de almacenamiento de entrada de reloj
CLOCK	Registro de desplazamiento
/OE	Habilita la salida
/RESET	Reinicia el sistema
GND	Tensión de tierra lógica
VCC	Entrada de tensión de 3.3V

A-H	Salida de las señales de datos en paralelo

Para el sistema a implementar, hay que conocer bien tres de los pines mostrados en la tabla anterior, ya que la forma de comprender el sistema es entender bien su funcionamiento. El pin *SER_IN* es la entrada de datos en serie enviada desde el *MCU* del sistema. Los datos que pasan por ese pin serán posteriormente reproducidos en paralelo, siendo este chip el encargado de dicha transformación. El pin *L_Clock* almacena con cada flanco de subida de reloj el valor de uno de los motores *push-pull*. Por último, el pin *Clock* se corresponde con el refresco de los valores de los 6 motores *push-pull* conectados a la vez.

5.2.3 Baterías

En el presente TFG hace falta una fuente de alimentación que alimente a los motores *push-pull*. Se decide utilizar dos bases para baterías correspondientes a pilas AA conectadas en paralelo. Cada una de estas bases tiene 3 pilas de 1.5V y contiene un interruptor de encendido y apagado, como se muestran en la Figura 76 y en la Figura 77.

Figura 76. Base para baterías (3xAA).

Figura 77. Base para baterías (3xAA) abierta.

5.3 Implementación de la plataforma *Hardware* (*HW*) basada en motores *push-pull* para la reproducción de código Braille

En este Una vez analizados en el apartado anterior todos los componentes diferentes utilizados en esta implementación del presente TFG, en la Figura 78 se muestra un esquemático inicial para el montaje del circuito basado en el diseño con motores *push-pull* para la representación de código Braille.

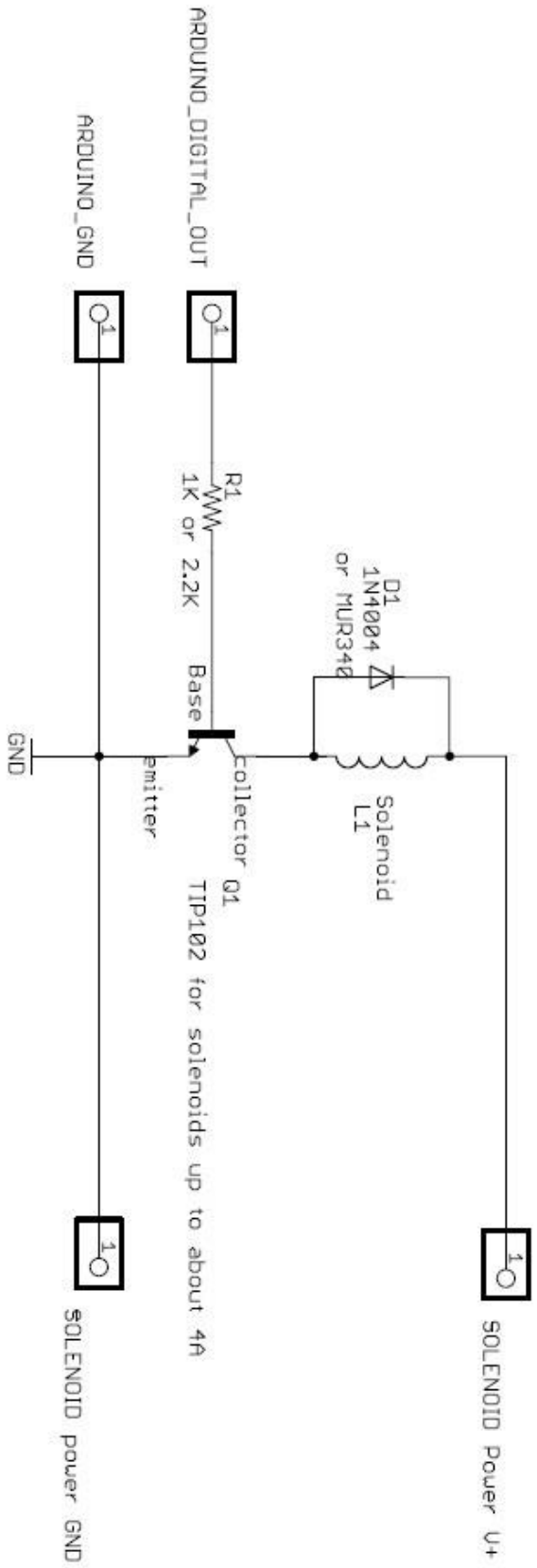

Figura 78. Esquemático de conexión de un motor *push-pull* a un *MCU*.

Siguiendo este esquemático y recordando lo importante que es tener un dispositivo conversor serie – paralelo para transformar los datos que llegan desde el *RedBear Duo*, se reproduce este sistema de tal manera que el esquemático representa los drivers de un motor, y por consiguiente hay que hacer 6 estructuras idénticas, una por cada motor que contendrá la célula de lectura Braille diseñada.

En la Figura 79 se muestra el circuito final de representación de código Braille, basado en una implementación *HW* con la utilización de 6 motores *push-pull*, montado sobre *protoboard*. Los motores aparecen pegados, pues es necesario que estén así para entender la reproducción en código Braille.

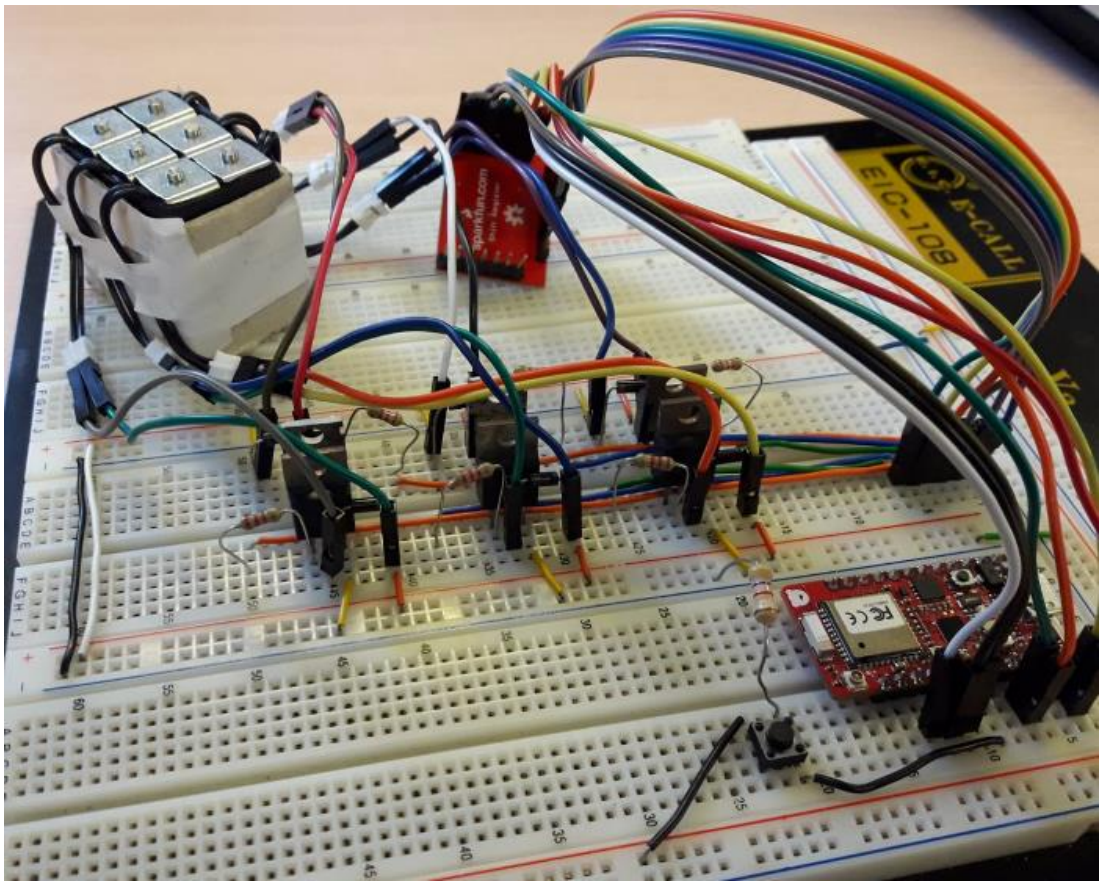

Figura 79. Montaje sobre *protoboard* de la célula diseñada con motores *push-pull*.

En la Figura 80 se muestra el mismo circuito mostrado en la imagen anterior, pero identificando cada componente para mayor comprensión del mismo.

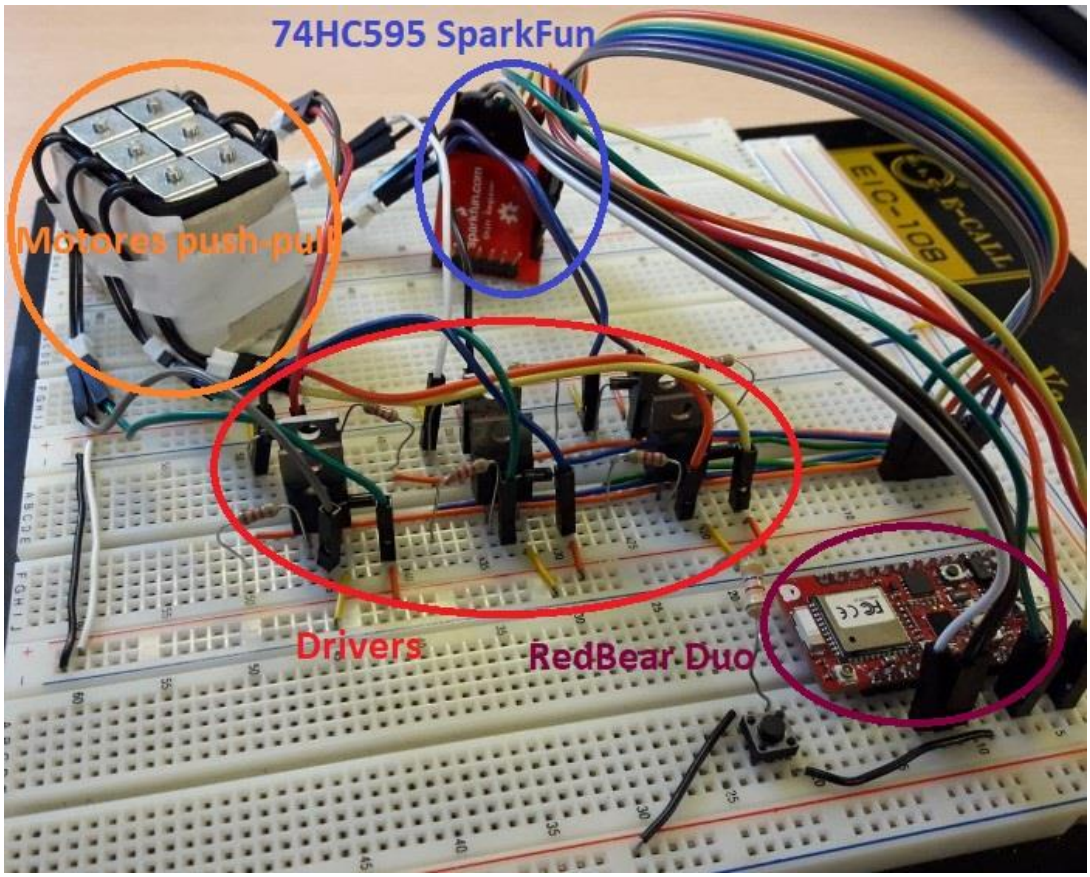

Figura 80. Montaje sobre *protoboard* de la célula diseñada con motores *push-pull*, con indicación de los componentes utilizados.

Capítulo 6. Desarrollo e integración de un código para la célula de lectura Braille diseñada con motores *push-pull*

6.1 Introducción

Como en el capítulo 4, se presenta el diseño correspondiente al *firmware* de la plataforma HW diseñada e implementada en el capítulo anterior. Este sistema estará basado en un *MCU* que reproducirá texto en código Braille, transmitido desde un dispositivo móvil vía *BLE*, en las células fabricadas con motores *push-pull*. La plataforma utilizada para el desarrollo de este *firmware* es *build.particle.io* basada en un lenguaje de programación *Wiring*.

6.2 Desarrollo software

Se pretende reutilizar todo el desarrollo *SW* posible de la implementación presentada en el capítulo 4, por lo que sólo se mencionarán las secciones de código que difieran. En este sentido, el código descrito en el apartado 4.4.3 no se modifica, debido a que la conexión entre el dispositivo *BLE Central* y el dispositivo *BLE Peripheral* se mantiene con respecto a la descripción funcional realizada.

La célula implementada con motores *push-pull* de lectura de código Braille necesita que desde el *MCU* se indique la codificación de cada uno de los símbolos a representar. Es necesario comprobar que lo que le ordena en este caso el dispositivo *RedBear Duo*, es lo que finalmente aparece representado en la célula.

Para el cumplimiento del objetivo 2 de este TFG y validar con ello la implementación *HW* planteada en el capítulo anterior de esta célula de lectura Braille, en el siguiente apartado se presenta el desarrollo de un *firmware* que permita verificar el correcto funcionamiento de los *drivers* y los motores *push-pull* utilizados.

6.2.1 Código de verificación de funcionamiento del *HW* implementado con motores *push-pull*

Se pretende que el código desarrollado sea capaz de hacer que la célula diseñada con motores *push-pull* reaccione a distintos impulsos de tensión. Para concretar más, este código forzará a que el conjunto de motores adopte primero en valor 0, a continuación el valor alto de tensión, y por último se vayan elevando los motores de dos en dos.

En la Figura 81 se muestra el diagrama de flujo seguido para esta verificación de la plataforma *HW* implementada.

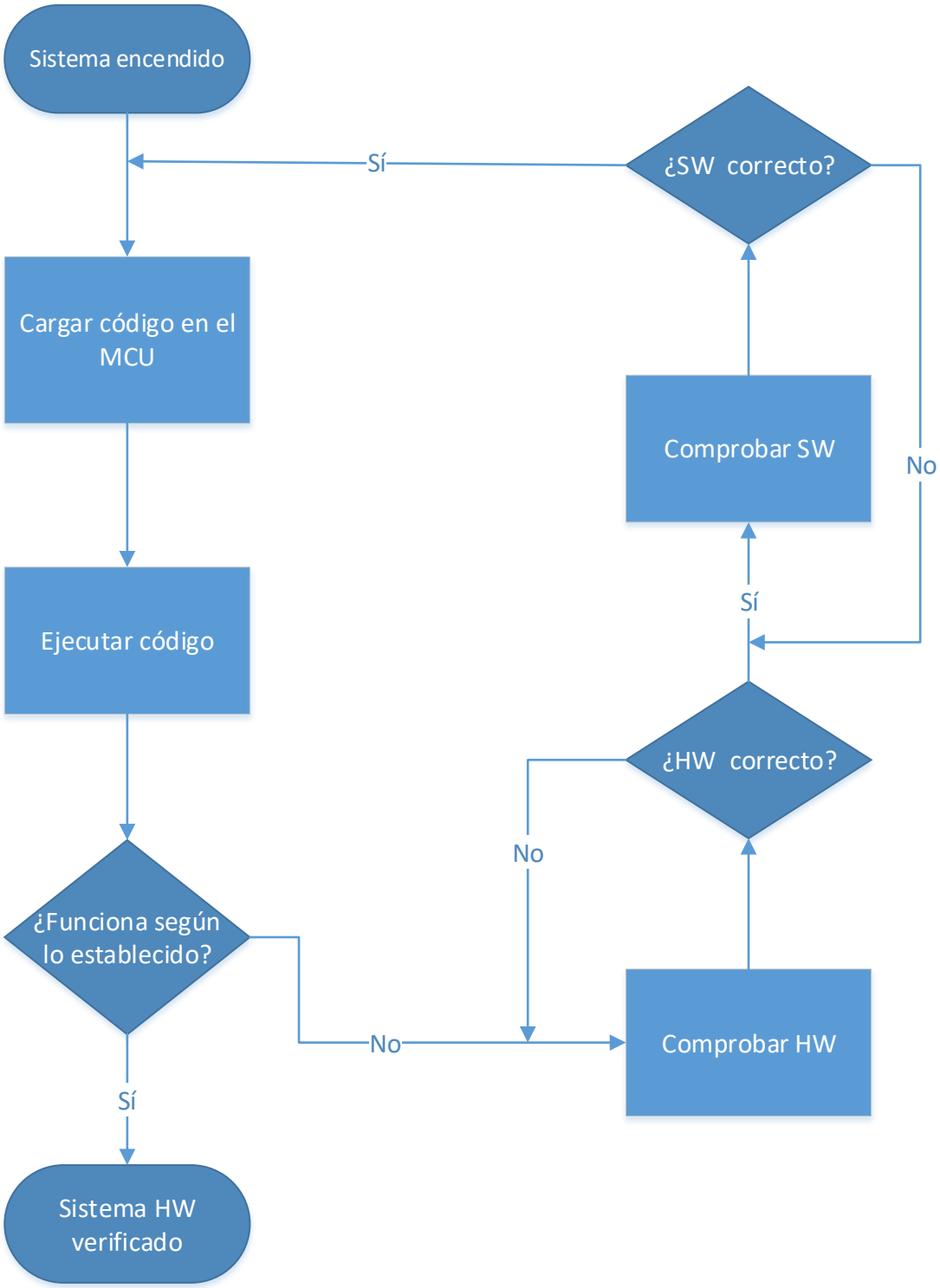

Figura 81. Diagrama de flujo de verificación del *HW* implementado con motores *push-pull*.

El código que se expone en este apartado se incluye en el fichero *sparkfun-74hc595.ino.* En el segmento de Código 30 se comienza mostrando un *include* del fichero *shifter.h* correspondiente a la librería *Shifter*, disponible en Particle IDE. A

continuación, se añaden las definiciones de las diferentes señales utilizadas, así como la variable que controla cuantos motores tiene conectados.

Código 30. Definiciones de las entradas y salidas del dispositivo.

```
#include "Shifter.h"

#define SER_Pin D4 //SER_IN
#define RCLK_Pin D3 //L_CLOCK - RCK
#define SRCLK_Pin D2 //CLOCK - SCK

#define NUM_REGISTERS 1 //how many registers are in the chain

// IMPORTANTE!. cuando NUM_REGISTERS es > 1, el índice del pin determina la
// célula en la que se representa. de manera que para 0<= i <= 7 será en la
// célula 0, para 8<= i <= 15 será en la célula 2, y así sucesivamente
```

Tras las últimas líneas de código, aparece el segmento de Código 31, que inicializa las variables creadas a partir de la librería *Shifter*.

Código 31. Inicialización de las librerías propias del *Shifter*.

```
//initaize shifter using the Shifter library
Shifter shifter(SER_Pin, RCLK_Pin, SRCLK_Pin, NUM_REGISTERS);
```

En la función *setup()* mostrada en el segmento de Código 32, se activa el puerto serie a 9600 bits/segundo.

Código 32. Función *setup()*.

```
void setup() {
  Serial.begin(9600);
}
```

Por último, en este segmento de Código 33 se muestra la función *loop()*. Las primeras líneas de esta función se corresponden con un *reset* de los motores, posicionándolos todos a nivel bajo; un *write* que actualiza con los valores almacenados; y un *delay* de 5 segundos. A continuación se presenta un repaso de todos los pines, en este caso concreto los pone todos a nivel alto menos el 6 y el 7 que los mantiene al valor 0, y vuelve a actualizar todos los valores almacenados a la vez. Tras esto aparece otro *delay* de 1 segundo, que se encarga de que la actualización de los motores vaya lenta para dar tiempo a reconocer el código Braille. El código de la función *loop()* va comprobando los valores que se han establecido para cada uno de los pines, y al finalizar los actualiza.

Código 33. Función *loop()*.

```
21  void loop() {
22     shifter.clear(); //set all pins on the shift register chain to LOW
23     shifter.write(); //send changes to the chain and display them
24     delay(5000);
25
26     shifter.setPin(0, HIGH); //set pin 1 in the chain(second pin) HIGH
27     shifter.setPin(1, HIGH);
28     shifter.setPin(2, HIGH);
29     shifter.setPin(3, HIGH);
30     shifter.setPin(4, HIGH);
31     shifter.setPin(5, HIGH);
32
33     shifter.setPin(6, LOW);
34     shifter.setPin(7, LOW);
35     shifter.write(); //send changes to the chain and display them
36     delay(1000);
```

6.2.2 *Firmware* de representación de textos en código Braille

Este código se basará en traducir texto a código Braille, y luego reproducirlo en la célula de lectura Braille diseñada con motores *push-pull*. Se intentará reutilizar la mayor cantidad de líneas de código posible de la aplicación expuesta en el apartado anterior.

En la Figura 82 se muestra un diagrama de flujo que corresponde a los pasos seguidos para completar los objetivos establecidos en este caso. El código que acompaña a este diagrama se encuentra en el fichero *braille-desing-sin-ble.ino.*

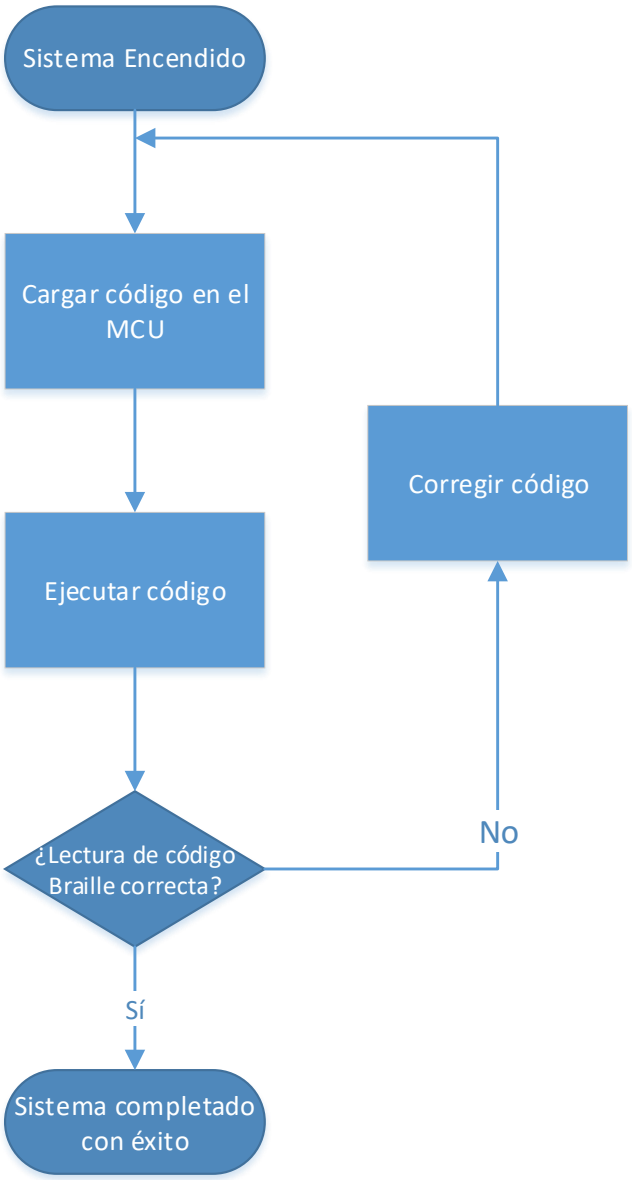

Figura 82. Flujograma del sistema de representación de código Braille basado en motores push-pull.

Como en el apartado anterior, en el Código 34 se inicia el desarrollo de este *firmware* con la inclusión del fichero *Shifter.h*, con las definiciones de los pines de entrada y salida, y la inicialización de la librería *Shifter*. Aparte de esto, se declara el vector *cells[]* , y por último, se define también la función *bitRead()*.

Código 34. Declaraciones de las entradas y salidas del dispositivo.

```
// This #include statement was automatically added by the Particle IDE.
#include "Shifter.h"

#define SER_Pin D4 //SER_IN
#define RCLK_Pin D3 //L_CLOCK - RCK
#define SRCLK_Pin D2 //CLOCK - SCK

#define cellCount 1 //how many registers are in the chain
//initaize 74hc595
Shifter shifter(SER_Pin, RCLK_Pin, SRCLK_Pin, cellCount);
byte cells[cellCount];

#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
```

Las variables coinciden al 100% con las del desarrollo planteado en el apartado 4.4.2, pero se considera importante recordar sus definiciones para comprender el código, ya que éste difiere en algunas líneas con el desarrollado para la célula *modul P16* de *metec*. Así que en el segmento de Código 35 aparece:

- *test[]:* Vector de datos del cual se va a realizar la traducción en código Braille para su posterior reproducción en las células de *metec*.
- *test_indx:* Índice utilizado para recorrer el vector anterior.
- *cells_offset:* Esta variable se encarga de añadir el *offset* necesario para que el índice que marque el vector *charcells[]* coincida con el código ASCII del símbolo que se desea representar en código Braille.
- *charCount:* Constante que tiene un valor entero de 128.
- *charcells[]:* Vector que se encarga de la traducción de los caracteres del texto, a código Braille. Este vector posee un valor constante de celdas dispuesto por el valor de *charCount*.
- *charcells_MAY:* Variable de tipo byte, que será la primera parte de los caracteres en mayúscula.
- *charcells_NUM:* Variable de tipo byte, que incluye la primera parte de los caracteres numéricos.
- *buttonAlreadyPressed:* Variable que comprueba si el botón asociado al pin 6 del *MCU* ha sido pulsado, y en consecuencia fuerza una actualización de la célula de lectura Braille *modul P16*.
- *enable_char:* Variable booleana que indica si existe un carácter a representar.
- *cells_index:* Índice que recorre el vector *cells[]*.
- *cells_prefix:* Antes de explicar esta variable es importante saber que cuando se quiere reproducir en la célula Braille caracteres en mayúscula o numéricos, es necesario completarlo con un primer carácter que indica si se trata de una mayúscula o un número, y el segundo reproduciendo la conversión del código que representa ese valor. Esta variable es la encargada de forzar la reproducción del código que representa a las

mayúsculas o los números, antes que el código que representa el valor del carácter que se quiere mostrar en la célula Braille *modul P16*.

- *char_tmp:* Variable que asume el valor del carácter a reproducir en cada momento.

Código 35. Declaraciones de variables.

```
char test[] = {"hola"};
int test_idx = 0;
int cells_offset = 0;
const int charCount = 128;
byte charcells[charCount];
byte charcells_MAY;
byte charcells_NUM;

#define BUTTONPIN D6
bool buttonAlreadyPressed;
bool enable_char;
int cells_index;
bool cells_prefix;
char char_tmp;
```

La función *setup()*, mostrada en el Código 36 , es la encargada de inicializar el puerto serie, hace un *reset* de los motores, y establecer valores iniciales para las variables. Dentro de este código también se hace la llamada a la función *init_charcells()*.

Código 36. Función *setup()*.

```
void setup()
{
 Serial.begin(9600);

 shifter.clear();
 shifter.write();

 test_idx = 0;
 cells_offset = 0;
 init_charcells();

 buttonAlreadyPressed = false;
 enable_char = false;
 cells_index = 0;
 cells_prefix = false;

 Serial.print("sizeof(test) = ");
 Serial.println(sizeof(test));
}
```

Debajo de estas líneas, en el segmento de Código 37, se presenta la primera parte de la función *loop()*. Este código coincide en un 95% con el que fue redactado en el apartado 4.4.2. En su primera línea se crea una variable con el valor que contiene el pin *BUTTONPIN*, asociado al valor del pulsador, y con el cual se actualizaría la célula de lectura Braille. Lo siguiente es una condición enmarcada en otra condición que si se cumplen ambas se activa la variable que establece que hay un carácter preparado para actualizar la célula, estableciéndose que el índice que recorrerá el vector *cells[]* esté a cero. Seguido se le asigna a la variable *buttonAlreadyPressed* el valor de la variable local antes inicializada.

Código 37. Primera parte de las función *loop()*.

```
void loop()
{
   bool buttonPressed = digitalRead(BUTTONPIN);
   if (buttonPressed && !buttonAlreadyPressed) {
      if (test_idx < sizeof(test)) {
         enable_char = true;
      }
      cells_index = 0;
   }

   buttonAlreadyPressed = buttonPressed;
```

La segunda y última parte de la función *loop()* se corresponde con una jerarquía de condicionales (Código 38). La condición de mayor categoría comprueba si existe algún carácter preparado para actualizar la célula. La siguiente condición comprara si el índice de *cells[]* es menor que la cantidad de motores que soporta, en caso negativo la variable *enable_char* adquiere el valor *false*, se realiza una llamada a la función *Flush()* con el valor de *cellCount*, y finaliza con una llamada a la función *Wait()*. Dentro de esta condición existen dos sentencias nuevas:

- La primera de ellas comprueba si el índice del vector *test[]* coincide con el tamaño del vector. Si es así, añade un carácter vacío, mientras que si por el contrario no se cumple dicha condición, la variable *char_tmp* adquiere el valor contenido en el vector *test[]* en la posición marcada por el índice actual.
- El siguiente condicional tiene tres ventanas de análisis, dado que en la primera comprueba si el carácter está en mayúscula, en el segundo si se trata de un número, y en el tercer caso si responde a cualquier otro carácter. Los códigos asociados a las mayúsculas y a los números son casi iguales, pues primero se asigna el valor de las mayúsculas o números, a continuación se establece el valor de *offset,* necesario para realizar correctamente la conversión posterior en el *array charcells[]*, y se finaliza estableciendo el valor de la variable encargada de forzar la reproducción de estos valores antes de la traducción del resto del valor. Como se explicó con anterioridad cuando existen mayúsculas o números, se necesita que la célula se actualice dos veces para representar el valor correcto de la traducción en código Braille. El último caso ofrece una condición que devuelve el *offset* al valor cero. Aparte, asigna el valor de *charcells[]* contenido en la celda resultante de la suma de *char_tmp* y *cells_offset,* al vector *cells[]*. Luego establece a *false* el valor de *cell_prefix*. Por último, comprueba que el valor del índice del vector *test[]* es inferior que el tamaño del mismo, y si se cumple aumenta en uno el valor de dicho índice.

Código 38. Última parte de la función *loop()*.

```
64    if (enable_char) {
65      if (cells_index < cellCount) {
66        if (test_idx == sizeof(test)) {
67          char_tmp = ' ';
68        }
69        else {
70          char_tmp = test[test_idx];
71        }
72
73        if ((!cells_prefix) && (char_tmp >= 'A') && (char_tmp <= 'Z')) {
74          Serial.println(charcells_MAY, HEX);
75          cells[cells_index++] = charcells_MAY;
76          cells_offset = 32;
77          cells_prefix = true;
78        }
79        else if ((!cells_prefix) && (char_tmp >= '0') && (char_tmp <= '9') ) {
80          Serial.println(charcells_NUM, HEX);
81          cells[cells_index++] = charcells_NUM;
82          cells_offset = 0;
83          cells_prefix = true;
84        }
85        else {
86          if (!cells_prefix)
87            cells_offset = 0;
88
89          Serial.println(charcells[char_tmp+cells_offset], HEX);
90          cells[cells_index++] = charcells[char_tmp+cells_offset];
91          cells_prefix = false;
92
93          if (test_idx < sizeof(test))
94            test_idx++;
95        }
96      }
97      else {
98        enable_char = false;
99        Flush(cellCount);
100       Wait();
101     }
```

En el Código 39 se muestra la función *Wait()*, encargada de que los movimientos ejercidos por los motores *push-pull* no adquieran una velocidad ilegible para poder leer las representaciones en código Braille.

Código 39. Función *Wait()*.

```
105 void Wait()
106 {
107  delay(1000);
108 }
```

La función *Flush()* se basa en el análisis de los pines que afectan a los motores, y al finalizar esa comprobación, los actualiza instantáneamente. Esto se puede ver en los segmentos de Código 40 y Código 41.

Código 40. Primera parte de la función *Flush()*.

```
void Flush (int count)
{
 for (int i = 0; i < count; i++)
 {
   Serial.print("* cell index = ");
   Serial.println(i);
   if ( bitRead(cells[i], 0) )  {
      shifter.setPin(0, i, HIGH);
      Serial.println("   - PIN_DATA 0 HIGH");
      }
   else {
      shifter.setPin(0, i, LOW);
      Serial.println("   - PIN_DATA 0 LOW");
      }

   if ( bitRead(cells[i], 1) )  {
      shifter.setPin(1, i, HIGH);
      Serial.println("   - PIN_DATA 1 HIGH");
      }
   else {
      shifter.setPin(1, i, LOW);
      Serial.println("   - PIN_DATA 1 LOW");
      }

   if ( bitRead(cells[i], 2) )  {
      shifter.setPin(2, i, HIGH);
      Serial.println("   - PIN_DATA 2 HIGH");
      }
   else {
      shifter.setPin(2, i, LOW);
      Serial.println("   - PIN_DATA 2 LOW");
      }

   if ( bitRead(cells[i], 3) )  {
      shifter.setPin(3, i, HIGH);
      Serial.println("   - PIN_DATA 3 HIGH");
      }
```

Código 41. Segunda parte de la función *Flush()*.

```
   else {
      shifter.setPin(3, i, LOW);
      Serial.println("   - PIN_DATA 3 LOW");
      }

   if ( bitRead(cells[i], 4) )  {
      shifter.setPin(4, i, HIGH);
      Serial.println("   - PIN_DATA 4 HIGH");
      }
   else {
      shifter.setPin(4, i, LOW);
      Serial.println("   - PIN_DATA 4 LOW");
      }

   if ( bitRead(cells[i], 5) )  {
      shifter.setPin(5, i, HIGH);
      Serial.println("   - PIN_DATA 5 HIGH");
      }
   else {
      shifter.setPin(5, i, LOW);
      Serial.println("   - PIN_DATA 5 LOW");
      }

   shifter.setPin(6, i, LOW);
   shifter.setPin(7, i, LOW);
 }

 shifter.write();
}
```

Para concluir con este desarrollo *SW*, queda analizar la función *init_charcells()* (Código 42). Esta función únicamente preestablece valores de traducción de diferentes caracteres y símbolos en código Braille a las variables *charcells_MAY* y *charcells_NUM*, y al vector *charcells[]*. El índice de este *array* es el código *ASCII* del símbolo a representar.

Código 42. Función *init_charcells()*.

```
void init_charcells()
{
 charcells_MAY = 0x28;
 charcells_NUM = 0x3C;

 charcells['a'] = 0x01;
 charcells['b'] = 0x03;
 charcells['c'] = 0x09;
 charcells['d'] = 0x19;
 charcells['e'] = 0x11;
 charcells['f'] = 0x0B;
 charcells['g'] = 0x1B;
 charcells['h'] = 0x13;
 charcells['i'] = 0x0A;
 charcells['j'] = 0x1A;
 charcells['k'] = 0x05;
 charcells['l'] = 0x07;
 charcells['m'] = 0x0D;
 charcells['n'] = 0x1D;
 charcells['ñ'] = 0x3B;
 charcells['o'] = 0x15;
 charcells['p'] = 0x0F;
 charcells['q'] = 0x1F;
 charcells['r'] = 0x17;
 charcells['s'] = 0x0E;
 charcells['t'] = 0x1E;
 charcells['u'] = 0x25;
 charcells['v'] = 0x27;
 charcells['w'] = 0x3A;
 charcells['x'] = 0x2D;
 charcells['y'] = 0x3D;
 charcells['z'] = 0x35;
 charcells['0'] = 0x1A;
 charcells['1'] = 0x01;
 charcells['2'] = 0x03;
 charcells['3'] = 0x09;
 charcells['4'] = 0x19;
```

6.3 Resultados obtenidos

Al igual que en el Capítulo 4, se ha obtenido una implementación *HW/SW* que es capaz de transferir texto desde un dispositivo móvil, actuando como dispositivo *BLE Central*, al *RedBear* Duo, que fue programado como dispositivo *BLE Peripheral*. Siendo este último dispositivo el *MCU* del resto del sistema, y consiguiendo reproducir texto en código Braille en una plataforma *HW/SW* diseñada con motores *push-pull*.

La implementación HW/SW basada en la célula diseñada con motores *push-pull* también ha cumplido con todas las especificaciones que se esperaban, pues se puede recibir texto vía BLE, y conseguir que dicho texto sea representado correctamente en código Braille, Este sistema es de bajo consumo de potencia y su diseño ha requerido un coste mínimo.

Capítulo 7. Conclusiones

Este TFG ha sido dividido en dos partes bien diferenciadas, pero que a la vez tienen los mismos objetivos, pues ambas han de cumplir con un *firmware* capaz de trabajar como dispositivo *BLE Peripheral*, a la vez conseguir traducir a código Braille, y hacer su representación.

En la parte que se trabajó con los dispositivos de *metec*, se partió de la célula *modul P16*, pero se tenía que buscar una implementación HW capaz de soportar altas tensiones, con un *backplane* con pines que no tienen el tamaño estándar, o con la necesidad de adquirir conversores para conseguir que la implementación cumpliera los requisitos de funcionalidad. Posteriormente hubo que realizar un *firmware* que se adaptara a todas las especificaciones que se planteaban. Este sistema, pese a tener un tamaño idóneo, no es aceptable a nivel de potencia consumida o económicamente hablando, que son los dos factores prioritarios a cumplir tras el desarrollo de este sistema *HW/SW*.

La plataforma *HW/SW* implementada con motores *push-pull* ha conseguido realizar representación en código Braille con texto recibido vía *BLE*. Este diseño ha supuesto una mejora económica bastante considerable con respecto a los productos que se encuentran en el mercado, con un consumo de potencia aceptable para un número reducido de células. Con esto se cumplen todos los objetivos planteados para este TFG. Aunque todo parezca correcto, este prototipo en concreto necesita algunas mejoras, como son la posibilidad de encontrar otros motores que se adapten mejor en cuanto a consumo de corriente, u obtener una batería que sea capaz de proporcionar corrientes elevadas.

Bibliografía

[1] *Organización Mundial de la Salud*. Disponible: https://www.who.int/es?page=1/////. Último acceso: 16 de Febrero, 2018.

[2] *OMS | Ceguera y discapacidad visual*. Disponible: http://www.who.int/mediacentre/factsheets/fs282/es/. Último acceso: 20 de Enero, 2018.

[3] *Instituto Nacional de Estadística.* Disponible: https://www.ine.es/. Último acceso: 17 de Febrero, 2018.

[4] *¿Cómo funciona el sistema Braille?*. Disponible: https://www.vix.com/es/btg/curiosidades/4659/como-funciona-el-sistema-braille. Último acceso: 15 de Enero, 2018.

[5] *Humanware - Brailliant BI 40 (NEW generation) braille display *. Disponible: http://store.humanware.com/hus/brailliant-bi-40-new-generation.html. Último acceso: 15 de Enero, 2018.

[6] *BLITAB® – Feelings get visible*. Disponible: https://blitab.com/. Último acceso: 20 de enero, 2018.

[7] *BraiBook*. Disponible: https://braibook.com/. Último acceso: 15 de Enero, 2018.

[8] *Braille cell P16*. Disponible: http://web.metec-ag.de/braille%20cell%20p16.html. Último acceso: enero,2018.

[9] *Kundenspezifische Braille-Module – metec AG*. Disponible: https://www.metec-ag.de/produkte-braille-module.php. Último acceso: Enero 15,2018.

[10] *HV507 - 64 Channel Serial-to-Parallel Converter with High-Voltage Push-Pull Outputs (datasheet)*.

[11] *RedBear Duo*. Disponible: https://redbear.cc/product/wifi-ble/redbear-duo.html. Último acceso: 15 de Enero, 2018.

[12] *SparkFun Electronics*. Disponible: https://www.sparkfun.com/. Último acceso: 18 de Enero, 2018.

[13] *Flat Flexible Cable 0.5mm, 1mm and 1.25mm Pitch (datasheet)*.

[14] G. Recio Rodríguez, "Estudio De La Tasa De Transferencia Del Dispositivo IoT RedBear Duo En Comunicaciones BLE.", Universidad de Las Palmas de Gran Canaria, 2018.

[15] *ISO/IEC 9834-8:2008*. Disponible: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/34/53416.html. Último acceso: 20 de Abril, 2018.

[16] *Particle Web IDE*. Disponible: https://build.particle.io. Último acceso: 23 de Febrero, 2018.

[17] *Nordic Semiconductor*. Disponible: https://www.nordicsemi.com/en. Último acceso: 20 de Marzo, 2018.

[18] *Pimoroni COM2700 Mini Push-Pull Solenoid 5V 0.42A*. Disponible: https://www.rapidonline.com/pimoroni-com2700-mini-push-pull-solenoid-5v-0-42a-73-5318. Último acceso: 15 de Enero, 2018.

Parte II: Pliego de Condiciones

Pliego de condiciones

El Pliego de Condiciones expone las condiciones bajo las que se ha desarrollado el presente trabajo. A continuación, se muestran el conjunto de herramientas *hardware y software* empleadas durante su realización.

PC.1 Condiciones *Hardware*

En la Tabla PC. 1 se presentan los equipos *hardware* utilizados.

Tabla PC. 1. Equipos *hardware.*

Dispositivo/Herramienta	Modelo	Fabricante/Comerciante
RedBear Duo	*RedBear DUO* con pines soldados	*RedBear*
Conversor *DC-DC* de 3.3V a 5V	*SparkFun Logic Level Converter - Bi-Directional*	*SparkFun*
Conversor *DC-DC* de 5V a 185V	*DC-DC Converter*	*metec*
Célula *modul P16*	*modul P16*	*metec*
Backplane	*Backpanel*	*metec*
Motores *push-pull*	DS - 0420S - 05A12	Shenzhen DingShengFeiYang Technology co.
Resistencias	-	-
Diodos	-	-
Transistores	TIP102	-
Smartphone	J7 2016	*Samsung*
Ordenador Portátil	*Aspire E15*	*Acer*

PC.2 Condiciones *Software*

En la Tabla PC. 2 se exponen las herramientas *software* utilizadas, especificando su versión.

Tabla PC. 2. Herramientas *software*.

Software	Versión	Desarrollador
Sistema operativo portátil	*Microsoft Windows 10 Home*	*Microsoft*
Sistema operativo *smartphone*	*Android 8.1.0*	*Samsung*
Microsoft Office	*Microsoft Office 365 ProPlus*	*Microsoft*

Microsoft Visio	2016	*Microsoft*
Particle IDE	-	*Particle*
PuTTY	V0.70	*PuTTY project*
Google Chrome	V 71.0.3578.98/ 64 bits	*Google*
Adobe Reader	V11.0.21.18	*Adobe Systems Software Ireland Ltd.*
nRF Connect	V4.19.0	*Nordic Semiconductor*

PC.3 Condiciones *Firmware*

Por último, en la se expone el *firmware* utilizado, especificando su versión.

Tabla PC. 3. *Firmware* usado.

Firmware	Versión
RedBear Duo	v.0.3.1

Parte III: Presupuesto

PRESUPUESTO

Este capítulo es en el que se tendrán en cuenta los gastos generados en la realización de presente TFG. Dicho presupuesto está compuesto por:

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

Una vez analizados cada uno de los criterios establecidos, se aplicarán los impuestos vigentes y se procederá a la obtención del coste total del Trabajo Fin de Grado.

P.1 Trabajo tarifado por tiempo empleado

Este concepto contabiliza los gastos que corresponden a la mano de obra, según el salario correspondiente a la hora de trabajo de un Ingeniero Técnico de Telecomunicación. Se propone utilizar la siguiente fórmula:

$$H=Ct \times 74{,}88 \times Hn + Ct \times 96{,}72 \times He \quad (1)$$

Donde:

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

Para la realización del presente TFG se han invertido un total de 300 horas. Todas ellas se han realizado dentro del horario normal, por lo que el número de horas especiales es cero. Además, de acuerdo a lo establecido por el COIT, el factor de corrección Ct a aplicar para 300 horas trabajadas es de 0,60, tal y como se puede comprobar en la Tabla P.1:

Tabla P. 1. Coeficientes reductores para trabajo tarifado (COIT)

Horas	Factor de corrección
Hasta 36	1,00
Exceso de 36 hasta 72	0,90
Exceso de 72 hasta 108	0,80
Exceso de 108 hasta 144	0,70
Exceso de 144 hasta 180	0,65
Exceso de 180 hasta 360	0,60
Exceso de 360 hasta 510	0,55
Exceso de 510 hasta 720	0,50
Exceso de 720 hasta 1080	0,45
Exceso de 1080	0,40

Por tanto, utilizando la fórmula (1) ofrecida por el COITT:

H=0,6 x 74,88 x 300+ 0,6 x 96,72 x 0=13478,40€ (2)

Por lo tanto, el trabajo tarifado por tiempo empleado asciende a la cantidad de trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos.

P.2 Amortización del inmovilizado material

Para la realización de este Trabajo Fin de Grado han sido necesarios tanto recursos hardware como recursos software. La amortización de estos recursos se calcula sobre el tiempo útil de los mismos. El sistema de amortización se toma como lineal siguiendo la siguiente fórmula (3):

$$\text{Cuota} = \frac{\text{Valor de la adquisición-Valor residual}}{\text{Tiempo de vida útil}} \quad (3)$$

P.2.1 Recursos *Hardware*

Debido a que la duración de este Trabajo Fin de Grado es de tan solo 4 meses, siendo este periodo muy inferior al de 3 años estipulado para el coste de amortización, los costes se calcularán en base a los derivados de los primeros 4 meses.

En la Tabla P. 2 se especifica el *hardware* amortizable necesario para la realización del trabajo, indicando su valor de adquisición y su amortización, teniendo en cuenta un tiempo de uso de 4 meses, excepto con los materiales o dispositivos que por su condición de bajo coste y utilización, donde su amortización coincide con su coste de adquisición.

Tabla P. 2. Recursos *hardware*.

Dispositivo/Herramienta	Valor de adquisición	Amortización
RedBear Duo (X 2)	43,78€	43,78€
Conversor *DC-DC* de 3.3V a 5V	2,59€	2,59€
Conversor *DC-DC* de 5V a 185V	52,00€	52,00€
Célula *modul P16* (X 3)	108,30€	108,30€
Backplane	27,40€	27,40€
Motores *push-pull (X 6)*	39,67€	39,67€
Material electrónico vario	31,93€	31,93€
Smartphone	245,00€	245,00€
Ordenador portátil	599,00€	66,58€
Total *hardware*	1149,67€	617,25€

El coste total del material *hardware* asciende a seiscientos diecisiete euros con veinticinco céntimos.

P.2.2 Recursos *Software*

Para el cálculo de los costes de amortización del material *software* se considerarán, al igual que con el material hardware, los costes derivados de los primeros 4 meses.

La Tabla P. 3 muestra los elementos *software* necesarios para la realización del trabajo, así como su valor de adquisición y su amortización.

Tabla P. 3. Recursos *software*.

Software	Valor de adquisición	Amortización
Sistema operativo portátil	Licencia ULPGC	0,00€
Sistema operativo *smartphone*	Licencia ULPGC	0,00€
Microsoft Office	Licencia ULPGC	0,00€
Microsoft Visio	Licencia ULPGC	0,00€
Particle IDE	*Software* libre	0,00€
PuTTY	*Software* libre	0,00€
Google Chrome	*Software* libre	0,00€
Adobe Reader	*Software* libre	0,00€
nRF Connect	*Software* libre	0,00€
Total *software*	0,00€	0,00€

Así que el software total del material *software* asciende a cero euros.

P.3 Redacción del Trabajo Fin de Grado

Se ha utilizado la fórmula (4) para determinar el coste asociado a la redacción de la presente memoria.

$$R = 0,07 \times P \times Cn \quad (4)$$

Donde:

- R: son los honorarios por la redacción del trabajo.
- P: es el presupuesto.
- C_n: es el coeficiente de ponderación en función del presupuesto.

El valor del presupuesto se calcula sumando los costes del trabajo tarifado por tiempo empleado y de la amortización del inmovilizado material, tanto *hardware* como *software*. El resultado de los costes se muestra en la Tabla P. 4.

Tabla P. 4. Presupuesto.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13478,40€
Recursos *hardware*	617,25€
Recursos *software*	0,00€
Total	14095,65€

El presupuesto acumulado hasta el momento es de 14095,65€. Por su parte, el valor del coeficiente de ponderación C_n tiene valor unitario ya que se trata de un proyecto que no supera los 30.050,00€ de coste.

Por lo tanto:

$$R = 0,07 \times 14095,65 \times 1 = 986,70€ \quad (5)$$

Los costes de redacción del presente TFG libres de impuestos ascienden a novecientos ochenta y seis euros con setenta céntimos.

P.4 Derechos de visado del COITT

El COIT establece que para proyectos técnicos de carácter general, los derechos de visado para 2018 se calculan en base a (6).

$$V = 0,006 \times P1 \times C1 + 0,003 \times P2 \times C2 \quad (6)$$

Donde:

- V es el coste de visado del trabajo.
- P_1 es el presupuesto del proyecto.
- C_1 es el coeficiente reductor en función del presupuesto.
- P_2 es el presupuesto de ejecución material correspondiente a la obra civil.
- C_2 es el coeficiente reductor en función a P_2.

El valor del presupuesto P_1 se halla sumando los costes de las secciones correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material y a la redacción del documento. Esta suma se muestra en la Tabla P. 5. Al igual que en el caso anterior, el coeficiente C_1 para proyectos de presupuesto inferior a 30050,00€ es de 1,00€, asimismo el valor de P_2 es de 0,00€ ya que no se realiza ninguna obra.

Tabla P. 5. Presupuesto, incluyendo trabajo tarifado, amortización y redacción del trabajo.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13478,40€
Recursos *hardware*	617,25€
Recursos *software*	0,00€
Redacción del TFG	986,70€
Total	15082,35€

De esta forma, aplicando a la fórmula (6) los datos descritos sobre estas líneas y el coeficiente especificado se obtiene:

$$V = 0,006 \times 15082,35 \times 1 + 0,003 \times 0 \times C2 = 90,49€ \quad (7)$$

Los costes por derechos de visado del presupuesto ascienden a noventa euros con cuarenta y nueve céntimos.

P.5 Gastos de tramitación y envío

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

P.6 Material fungible

Además de los recursos *hardware* y *software*, en este trabajo se han empleado otros materiales, como los folios y el tóner de la impresora entre otros, que quedan englobados como material fungible.

En la Tabla P. 6 se muestran los costes derivados de estos recursos.

Tabla P. 6. Material fungible.

Concepto	Coste
Folios	10,00€
Tóner	45,00€
Encuadernado	5,00€
Total	60,00€

Los costes de materiales fungibles ascienden a sesenta euros.

P.7 Aplicación de impuestos

Para la actividad económica del presente TFG el valor del Impuesto General Indirecto Canario (IGIC) graba el presupuesto con un 7 %. El coste total del proyecto se desglosa en la Tabla P. 7.

Tabla P. 7. Presupuesto total del Trabajo Fin de Grado.

Concepto	Coste
Trabajo tarifado por tiempo empleado	13478,40€
Recursos *hardware*	617,25€
Recursos *software*	0,00€
Redacción del TFG	986,70€
Derechos de visado del COITT	90,49€
Gastos de tramitación y envío	6,00€
Materiales fungibles	60,00€
Total (sin IGIC)	15238,84€
IGIC (7%)	1066,72€
Total (con IGIC)	16305,56€

El importe final al que asciende el presupuesto del presente TFG (“Desarrollo de un sistema de representación en código Braille”) asciende a un total de dieciséis mil trescientos cinco euros con cincuenta y seis céntimos.

FDO: Dña. Araceli Marrero Mendoza

En Las Palmas de Gran Canaria a 17 de Enero de 2019

Parte IV: Anexos

ANEXO I

Junto a la memoria del presente TFG va a ir el archivo TFG_AraceliMarreroMendoza.zip. Este anexo enumerará el contenido de dicho archivo ZIP:

- Memoria del TFG "Desarrollo de un sistema de representación de textos en código Braille". Este documento se encuentra en español y en formato PDF.
- Formulario de "Petición de defensa del TFG". Este documento se encuentra en español y en formato PDF.
- Carpeta **Firmware**: Esta carpeta contiene todas las implementaciones de *software* realizadas en el presente TFG.
    - Archivo *BLEPeripheral_Duo.ino: **firmware*** con el código para el dispositivo ***BLE Peripheral.*** Este desarrollo se encuentra fuera de las carpetas porque había que duplicarlo, pues pertenece a ambas.
    - Carpeta **Célula modul P16**: Esta carpeta contiene todo el código que se desarrolló para la implementación *HW/SW* basada en la célula *modul P16* de *metec*.
        - Archivo *braille-p16-V00.ino*: este desarrollo *SW* es la versión de prueba para comprobar que la implementación *hardware* conseguía el funcionamiento de la célula.
        - Archivo *braille-p16-sin-ble.ino*: *firmware* desarrollado para la representación de código Braille en la célula.
        - Archivo *araceli-ble-p16-completo*: Implementación final del *firmware* basado en la célula *modul P16* de metec que cumple con los objetivos de este TFG.
        - Archivo *Circuito-modulP16.mp4*: vídeo que demuestra la implementación completa de del sistema basado en la célula de lectura Braille *modul P16* de *metec*.
    - Carpeta **Motores push-pull**: Esta carpeta contiene todo el código desarrollado para la plataforma *HW/SW* utilizando motores *push-pull*.
        - Archivo *sparkfun-74hc595.ino*: este desarrollo *SW* es la versión de prueba para comprobar que la implementación *hardware* diseñada funcionaba correctamente.
        - Archivo *braille-desing-sin-ble.ino*: ***firmware*** desarrollado para la representación de código Braille en la célula diseñada.
        - Archivo *araceli-ble-desing-completo*: implementación final del *firmware* basado en la célula implementada con motores *push-pull* que cumple con los objetivos de este TFG.
        - Archivos *Shifter.h* y *Shifter.cpp*: código suministrado por el fabricante para el buen funcionamiento del conversor serie-paralelo de *SparkFun.*

- Archivo *Circuito-diseñado.mp4*: vídeo que demuestra la implementación completa de del sistema basado en motores *push-pull*.