UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

PLATAFORMA PARA LA INTERPRETACIÓN DEL ALFABETO DACTILOLÓGICO DE LA LENGUA DE SIGNOS ESPAÑOLA BASADA EN DISPOSITIVOS IOT

Titulación:	**Grado en Ingeniería en Tecnologías de la Telecomunicación**
Mención:	**Sistemas Electrónicos**
Autora:	**Claudia R. Rivero Santana**
Tutores:	**D. Valentín de Armas Sosa**
	D. Félix B. Tobajas Guerrero
Fecha:	**Junio 2018**

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

PLATAFORMA PARA LA INTERPRETACIÓN DEL ALFABETO DACTILOLÓGICO DE LA LENGUA DE SIGNOS ESPAÑOLA BASADA EN DISPOSITIVOS IOT

Calificación: _____

Presidente

Fdo.: _____

Vocal **Secretario/a**

Fdo.: _____ Fdo.: _____

Fecha: Junio de 2018

Índice

Índice de Figuras

Índice de Tablas

Acrónimos

ADC	*Analog to Digital Converter*
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
ASCII	*American Standard Code for Information Interchange*
ATT	*ATTribute protocol*
BLE	*Bluetooth Low Energy*
CAN	*Controller Area Network*
CCCD	*Client Characteristic Configuration Descriptor*
CD-ROM	*Compact Disc Read-Only Memory*
CLI	*Command Line Interface*
COITT	*Colegio Oficial de Ingenieros Técnicos de Telecomunicación*
DAC	*Digital to Analog Converter*
DCT	*Device Configuration Table*
DPS	*Degrees Per Second*
EITE	*Escuela de Ingeniería de Telecomunicación y Electrónica*
GAP	*Generic Access Profile*
GATT	*Generic Attribute Profile*
GPIO	*General Purpose Input/Output*
HCI	*Host Controller Interface*
HW	*Hardware*
I2C	*Inter-Integrated Circuit*
IDE	*Integrated Development Environment*
IGIC	*Impuesto General Indirecto Canario*
IoT	*Internet of Things*
L2CAP	*Logical Link Control and Adaptation Protocol*
LED	*Light Emitting Diode*
LL	*Link Layer*
LSE	*Lengua de Signos Española*
MB	*Mega Byte*
MCU	*Micro Controller Unit*
MTU	*Maximum Transmission Unit*

OMS	*Organización Mundial de la Salud*
OTA	*Over-The-Air*
PC	*Personal Computer*
PDF	*Portable Document Format*
PDU	*Protocol Data Unit*
PHY	*PHYsical Layer*
PWM	*Pulse-Width Modulation*
RAM	*Random Access Memory*
RBF	*Radial Basis Function*
RGB	*Red Green Blue*
RTOS	*Real Time Operating System*
SCL	*Serial Clock*
SDA	*Serial Data*
SIG	*Signal*
SMP	*Security Manager Protocol*
SPI	*Serial Peripheral Interface*
SVM	*Support Vector Machine*
SVR	*Support Vector Regression*
SW	*Software*
TFG	*Trabajo Fin de Grado*
TIC	*Tecnología de la Información y las Comunicaciones*
UART	*Universal Asynchronous Receiver/Transmitter*
ULPGC	*Universidad de Las Palmas de Gran Canaria*
USB	*Universal Serial Bus*
UUID	*Universally Unique IDentifier*

Capítulo 1. Introducción

1.1 Planteamiento del problema

La Organización Mundial de la Salud (OMS) estima que más de mil millones de personas viven con algún tipo de discapacidad [1] y, por lo general, dependen de otras personas para vivir. De estos mil millones de personas, un 5% (360 millones) padece pérdida de audición [2]. Más concretamente, la referencia sobre población con discapacidad auditiva en España la cifra en torno al millón de personas, de las cuales un 10% se comunica principalmente mediante lengua de signos [3].

Debido a la continua evolución de las Tecnologías de la Información y las Comunicaciones (TIC), existen muchos dispositivos y sistemas disponibles para ayudar a personas con problemas de audición. Es en este aspecto donde IoT (*Internet of Things*) toma relevancia. El término IoT se basa en la utilización de Internet para

interconectar objetos entre sí, creando un entorno inteligente y fácilmente accesible por los usuarios. Es por ello que los dispositivos de IoT pueden ofrecer a las personas con discapacidad asistencia y apoyo para lograr una buena calidad de vida, facilitándoles la participación de manera independiente en la vida social y económica.

IoT ha sido, y continúa siendo, una revolución tecnológica en informática y comunicaciones. Desde sus inicios, se han desarrollado diversas ideas para personas con discapacidades de todo tipo [4], entre las que se encuentran dispositivos orientados a personas con discapacidad auditiva [5]. Dentro de este sector, existen tecnologías que facilitan la comunicación de las personas mediante lengua de signos [6], que será el propósito fundamental de este Trabajo Fin de Grado (TFG).

En la actualidad, los principales métodos para reconocer las posiciones del cuerpo humano se basan en sistemas ópticos [7]. Sin embargo, para la realización de este TFG se opta por un sistema de detección del movimiento de la mano y la posición de los dedos, similar a los desarrollos propuestos en [8], [9]. En ambos casos se sigue el planteamiento general expuesto en la Figura 1, utilizando sensores de distinto tipo (resistencias flexibles, sensores de contacto, etc.) para capturar los datos relativos a la disposición de la mano en cada caso, que son transferidos a un elemento central (por lo general un PC) encargado de procesar los datos capturados y realizar una clasificación que permita identificar el símbolo realizado con la mano. Finalmente, los símbolos reconocidos son, o bien mostrados en una pantalla, o bien transmitidos a un dispositivo móvil utilizando *Bluetooth*.

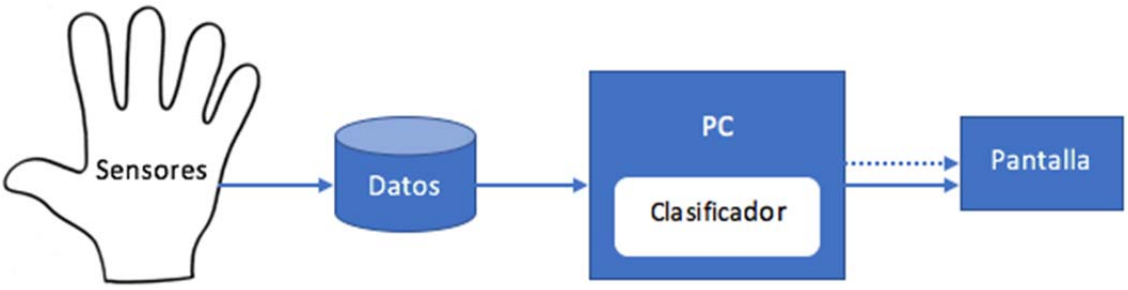


Figura 1. Planteamiento general de la idea

Para la idea propuesta en este TFG, se procurará realizar un diseño más compacto y con un grado de autonomía que permita al dispositivo principal, una vez entrenado, no

depender de un PC para la identificación de símbolos. Para su implementación, se utilizará inicialmente el dispositivo *Photon* [10], que proporciona conectividad inalámbrica WiFi, y el entorno de desarrollo proporcionado por su fabricante, Particle. Además, debido a que se pretende dar una solución IoT basada en bajo consumo para corto alcance, la tecnología escogida para la comunicación de los signos identificados a un dispositivo móvil es BLE (*Bluetooth Low Energy*), utilizándose para ello el dispositivo *RedBear Duo* [11], cuyo funcionamiento es similar al del *Photon*, con la diferencia de que, además de WiFi, integra BLE, siendo el código desarrollado para el dispositivo *Photon* compatible en gran medida con el dispositivo *RedBear Duo*.

1.2 Objetivos

El objetivo principal de este Trabajo Fin de Grado es la implementación de una plataforma Hardware/Software que permita la interpretación del alfabeto dactilológico de la Lengua de Signos Española (LSE) en tiempo real. Para ello, se desarrollará un guante sensor capaz de convertir los gestos realizados con la mano y enviar la traducción de los signos identificados a un dispositivo móvil de forma inalámbrica, mediante BLE. Asimismo, se pretende ofrecer la posibilidad de personalización de los símbolos del usuario. Con esto se consiguen dos propósitos fundamentales: permitir al usuario realizar una adaptación a diferentes lenguajes, y controlar otros dispositivos a través de símbolos concretos.

Para el desarrollo de la plataforma especificada se identifican tres funciones principales:

- Detección de símbolos, fundamentalmente a partir de resistencias flexibles o *flex sensors* [12], además de un acelerómetro y giroscopio integrados en el dispositivo LSM9DS1 de la empresa Sparkfun.
- Clasificación de los símbolos detectados basada en SVM (*Support Vector Machine).*
- Procesamiento/Transmisión de los símbolos detectados mediante conexión BLE.

1.3 Peticionario

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

1.4 Estructura del documento

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

Capítulo 1. Introducción. Se presenta el planteamiento del problema que ha motivado la realización de este TFG, se plantean las soluciones que se van a desarrollar, y se detallan los objetivos propuestos para ello. A continuación, se expone el peticionario y la estructura general del documento.

Capítulo 2. Componentes HW/SW. Se presentan los recursos que se utilizarán en el desarrollo del presente TFG, proporcionando una descripción detallada de los dispositivos hardware y los recursos software utilizados.

Capítulo 3. Proceso de clasificación. En este capítulo se describen los distintos métodos de clasificación posibles, destacando el basado en SVM, que será el utilizado para el desarrollo de la plataforma. Asimismo, se presentan el paquete software LIBSVM y la plataforma inicial desarrollada en este TFG, en la que se integran los diferentes recursos HW/SW.

Capítulo 4. Protocolo BLE. En este capítulo se presenta una introducción a la tecnología *Bluetooth Low Energy,* explicando con mayor detalle los aspectos relacionados con el presente TFG, además de introducir el dispositivo *RedBear Duo*.

Capítulo 5. Plataforma HW/SW final. En este capítulo se recogen los pasos seguidos para lograr la integración de todos los elementos en la plataforma HW/SW final, explicándose, tanto el código desarrollado, como los resultados obtenidos experimentalmente a partir de ella.

Capítulo 6. Conclusiones y líneas futuras. Se recogen las conclusiones obtenidas tras haber completado los objetivos propuestos para este TFG.

La segunda parte del documento consiste en el *Pliego de Condiciones*, mientras que la tercera parte se corresponde con el *Presupuesto*, y por último, un *Anexo*.

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

Presupuesto. En este apartado se recogen los gastos generados en la realización del presente TFG.

Anexo. Incluye la estructura y el contenido del CD-ROM adjunto.

Capítulo 2. Componentes HW/SW

2.1 Introducción

Para establecer los recursos Hardware/Software necesarios en el diseño de la plataforma establecida como objetivo principal del presente TFG, resulta fundamental realizar un análisis previo que permita conocer cómo son los movimientos asociados a los símbolos que se pretenden identificar. Para ello, se introducen dos conceptos: la lengua de signos y el alfabeto dactilológico.

Actualmente, las lenguas de signos cumplen con todas las características formales del lenguaje humano [13], poseen una gramática visual rica y propia, y su evolución y desarrollo depende de la comunidad de personas que la usan.

Sin embargo, no existe una única lengua de signos común, de forma que cada país posee una o varias lenguas de signos, y no existe una lengua de signos por cada lengua oral, ya que las lenguas de signos han evolucionado de forma natural en el contacto entre personas. De hecho, países que comparten el mismo idioma hablado utilizan diferentes lenguas de signos, y dentro de un mismo país, existen diferentes variantes según la región.

En cuanto al alfabeto dactilológico, es el alfabeto que usan las personas sordas [14]. Consiste en la representación manual de las letras que componen el alfabeto de la lengua oral. Esta representación se ejecuta en el aire en lugar de en un papel, concretamente en el espacio cercano a la cara de la persona que lo realiza, a la altura de la barbilla. Su realización se complementa con la articulación oral, por lo que es necesario que la cara y la boca sean visibles. Para realizar la dactilología, al igual que para producir cualquier signo de la lengua de signos, se utiliza la mano dominante (derecha para los diestros, e izquierda para los zurdos).

En sentido estricto, el alfabeto dactilológico no se considera lengua de signos, ya que en realidad es, como se ha comentado, una representación de las letras que forman las palabras de la lengua oral. Es decir, la dactilología se considera un complemento de la lengua de signos, un instrumento que sirve de puente entre la lengua de signos y la lengua oral.

En consecuencia, el alfabeto dactilológico puede resultar útil para:

- Hacer referencia expresa a nombres propios.
- Representar las palabras de una lengua oral cuando estas no tienen signo, o se desconoce.
- Identificar un signo cuando éste se presenta por primera vez.

La dactilología dejará de emplearse cuando exista un signo para una palabra o concepto. Sin embargo, su importancia no puede ser subestimada, pues resulta esencial para la persona que se inicie en la lengua de signos, además de poder utilizarse como mecanismo de control de diferentes acciones en el ámbito de IoT.

En la Figura 2 se recogen los 30 signos correspondientes al alfabeto dactilológico español, entre los cuales existen gestos de signos con diferencias muy sutiles entre sí. Para distinguir los diferentes signos, han de tenerse en cuenta varios aspectos: la flexión de cada uno de los dedos de la mano, el contacto entre los dedos, la posición de la mano para cada símbolo, y el movimiento, si lo hubiese.

La flexión de los dedos es el aspecto más importante a tener en cuenta para distinguir entre los diferentes signos. Extender y curvar uno o más dedos en varios ángulos puede indicar diferentes signos, como puños abiertos o cerrados. Por ejemplo, el signo gestual para la letra "I" es muy similar al signo de la letra "A", excepto que el dedo meñique está completamente extendido, en lugar de completamente flexionado. De la misma manera, es posible distinguir la mayor parte de las letras que componen el alfabeto.

Por otro lado, muchos signos requieren la misma posición exacta de la mano y se distinguen únicamente por el ángulo o el movimiento, como es el caso de las letras “N”, “Ñ”, “U”, y “V”. La letra “N” se distingue de la “Ñ” debido a que en ésta última existe movimiento, mientras que la letra “U” se distingue únicamente por el ángulo en el que se encuentra la muñeca cuando se realiza el movimiento. Lo mismo ocurre con las letras “L” y “LL”, “CH” y “H”, “R” y “RR”, y “M”, “P” y “W”. En todos estos casos, la forma de la mano es la misma con respecto a la flexión de los dedos, y las letras difieren entre sí solo porque existe movimiento, o por el movimiento en el ángulo en que se encuentra la muñeca cuando se realiza el signo.

En el caso particular de las letras “I”, “Y” y “Z”, tanto la flexión de los dedos como la posición de la mano son similares, por lo que será determinante tener en cuenta el movimiento que se realiza. De hecho, en este caso se contempló inicialmente la posibilidad de añadir sensores de contacto en la plataforma para distinguir entre las letras “I” e “Y”, pues en esta última, el movimiento del dedo meñique hace que exista un contacto con el lateral del dedo anular en un determinado momento del movimiento.

Figura 2. Alfabeto dactilológico español

2.2 Componentes Hardware de la plataforma inicial

A partir del planteamiento inicial, y tras un análisis visual de los gestos asociados al alfabeto dactilológico español, en este apartado se establecen los recursos HW necesarios para el diseño de la plataforma inicialmente propuesta.

Así, en principio se establece la necesidad de disponer de: sensores flexibles para detectar la posición de los dedos de la mano, un acelerómetro que permita identificar

el movimiento de la mano, y un dispositivo capaz de procesar localmente los datos recogidos por los sensores, que en el caso concreto de este TFG se corresponderá inicialmente con el dispositivo *Photon* de la empresa Particle.

En la Figura 3 se muestra un diagrama de bloques que permite tener una visión global de los componentes necesarios para el desarrollo de la plataforma inicial, donde los datos son recogidos y visualizados en un PC.

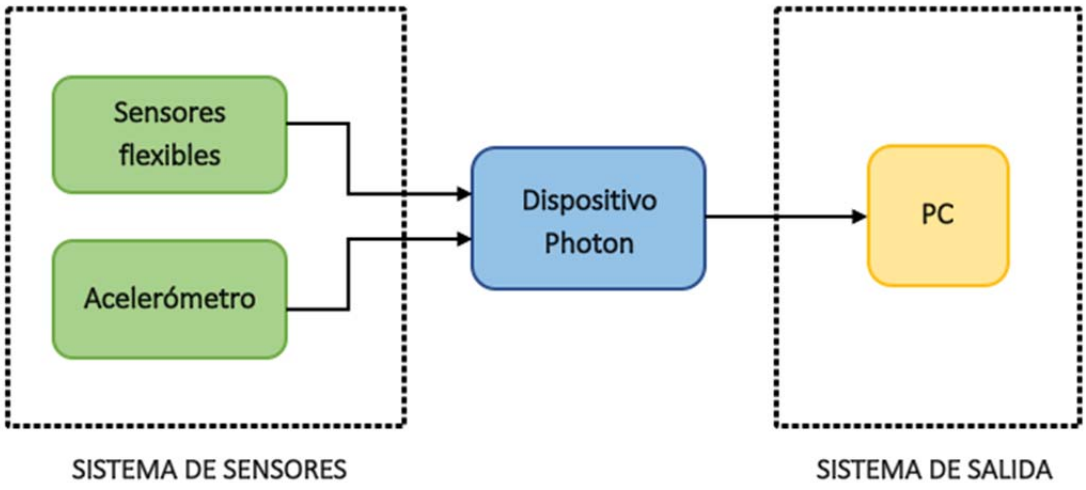


Figura 3. Diagrama de bloques con los componentes de la plataforma HW/SW inicial

2.2.1 Dispositivo *Photon*

Photon es un dispositivo desarrollado por la empresa Particle para la creación de proyectos y productos basados en IoT. Se ha escogido por ser fácil de usar, presentar un bajo coste, y por integrar un módulo WiFi.

Entre otras características, el dispositivo *Photon* dispone de 1MB de memoria *Flash* y 128KB de memoria RAM, además de varios LED (*Light-Emitting Diode*) integrados, 18 entradas/salidas de propósito general, periféricos avanzados y un sistema operativo en tiempo real (*FreeRTOS*). Este dispositivo cuenta con una gran cantidad de interfaces analógicas, digitales y de comunicación, tal y como se muestra en la Tabla 1 [15].

Tabla 1. Periféricos del dispositivo *Photon*

Tipo de Periférico	Cantidad	Entrada(E)/Salida(S)
Digital	18	E/S
Analógico (ADC)	8	E
Analógico (DAC)	2	S
SPI	2	E/S
I2S	1	E/S
I2C	1	E/S
CAN	1	E/S
USB	1	E/S
PWM	9	S

En la Figura 4 se detallan los principales componentes del dispositivo *Photon*. Los dos botones disponibles en la plataforma (*Setup* y *Reset*) permiten configurar nuevas credenciales WiFi y reiniciar el dispositivo, respectivamente. También se pueden utilizar en conjunto para configurar el modo del dispositivo. Entre ambos botones se encuentra un LED de tipo RGB (*Red*, *Green*, *Blue*) que proporciona información acerca del estado del dispositivo. Por ejemplo, si se encuentra conectado a una red WiFi, estará parpadeando muy suavemente en color celeste o, si se está cargando un programa en *Flash*, este LED parpadeará rápidamente en color magenta.

En la parte superior se encuentra el puerto micro-USB. Aunque su principal objetivo es proporcionar alimentación al dispositivo *Photon*, también se puede utilizar para la programación del dispositivo vía USB, así como para realizar comunicaciones serie USB con un ordenador.

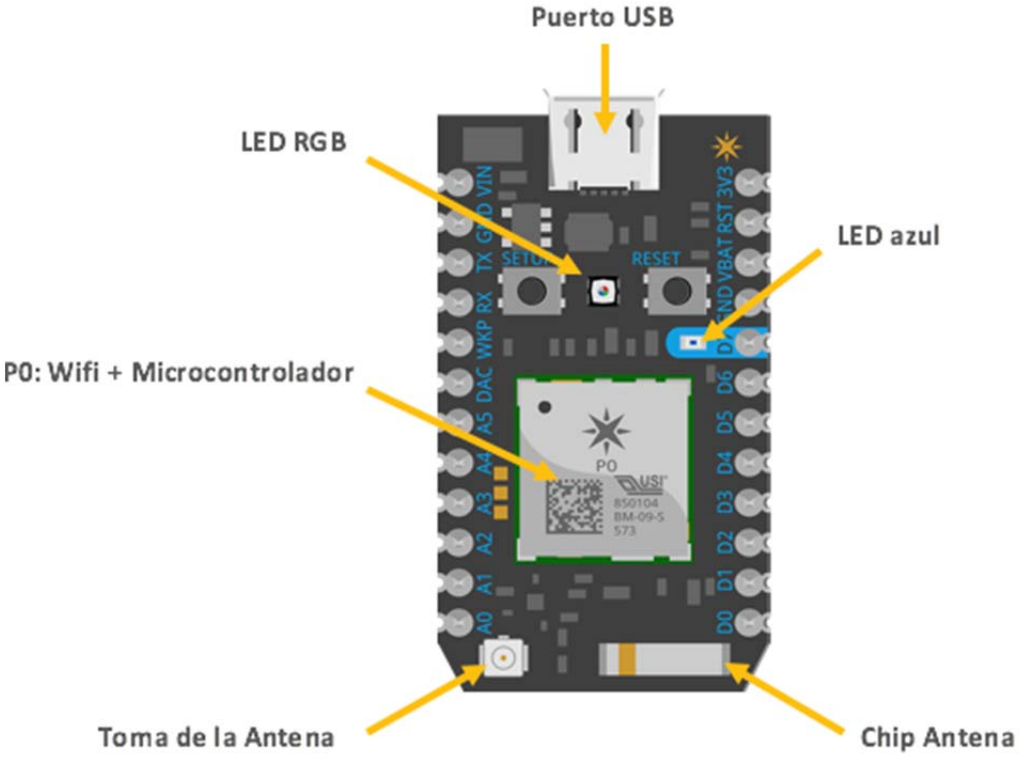


Figura 4. Componentes del dispositivo Photon

Este dispositivo combina un chip Cypress WiFi con un microcontrolador STM32 ARM Cortex M3 en un módulo llamado P0. El módulo WiFi permite la conexión a Internet, mientras que el microcontrolador es el encargado de ejecutar el software y, a diferencia de un ordenador, sólo puede ejecutar una aplicación.

Los pines son otro componente fundamental, pues son los que permiten las interacciones con el microcontrolador. Por un lado, los pines *GPIO* pueden conectarse a sensores o botones, funcionando como entradas, o a LED y timbres, funcionando como salidas. Por otro lado, hay pines de *reset* y alimentación (*3V3*, *RST*, *VBA*T, y *GND*), situados a la derecha del micro-USB, y pines para la comunicación serie (*TX,* y *RX*). Además, situados por encima de estos pines se encuentran un segundo pin *GND,* y el pin *VIN*, el cual permite alimentar al dispositivo *Photon* suministrando entre 3.6V y 5.5V, como una alternativa al uso del puerto USB.

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

Los pines *D0* a *D7* son pines de propósito general que pueden actuar como entradas o salidas digitales. Además, los pines *D0* a *D3* también pueden actuar como salidas analógicas utilizando técnicas PWM (*Pulse-Width Modulation*). Tal y como se aprecia en la Figura 4, existe también un LED azul situado junto al pin *D7* que se encenderá cuando este pin esté a nivel alto.

Por último, los pines *A0* a *A5* constituyen entradas analógicas que trabajan con tensiones de entre 0 y 3.3V. Los pines analógicos también se pueden utilizar como entradas o salidas digitales, como los pines *D0* a *D7* y, al igual que los pines digitales, algunos pines analógicos (*A4*, *A5*) también se pueden utilizar como salidas analógicas PWM. A continuación se encuentra el pin del conversor analógico-digital (*Digital to Analog Converter, DAC*). Se trata de un pin de salida analógica especial, capaz de proporcionar voltajes comprendidos entre 0V y 3.3V. A su lado está el pin *WKP*, el cual se utiliza para despertar al dispositivo *Photon* después de que se ha puesto en modo *deep sleep*, aunque también se puede emplear como salida analógica (*A7*).

2.2.2 Sensores flexibles

Los sensores flexibles de 2.2” del fabricante *Spectra Symbol* [16] son resistencias variables que aumentan su valor a medida que se doblan. Esto es posible porque un lado del sensor está impreso con una tinta de polímero con partículas conductoras incrustadas en él, de tal manera que cuando el sensor está recto, las partículas le dan a la tinta una resistencia de aproximadamente 30kΩ, mientras que cuando está doblado, las partículas conductoras se separan y la resistencia aumenta. Por todo ello, estos sensores son uno de los componentes fundamentales para el desarrollo

de la plataforma HW/SW objeto del presente TFG, ya que permitirán conocer la posición de cada dedo de la mano a partir del grado de doblez de las resistencias.

Es importante tener en cuenta que el sensor flexible está diseñado para flexionarse en una sola dirección, como se muestra en la Figura 5. Doblar el sensor en la dirección opuesta no produce datos fiables, y puede dañar el sensor. Además, es conveniente evitar doblar el sensor cerca de la base.

Figura 5. *Flex sensor*

En la Figura 6 se representa el circuito propuesto en [17] para conectar los sensores utilizando una plataforma basada en MCU, y el cual se ha tomado como referencia para el montaje del circuito utilizado en el presente TFG. Al combinar el sensor de flexión con una resistencia estática para crear un divisor de tensión (R2=47kΩ en el ejemplo tomado como referencia), es posible generar una tensión variable que pueda leerse mediante un convertidor analógico-digital integrado en el microcontrolador.

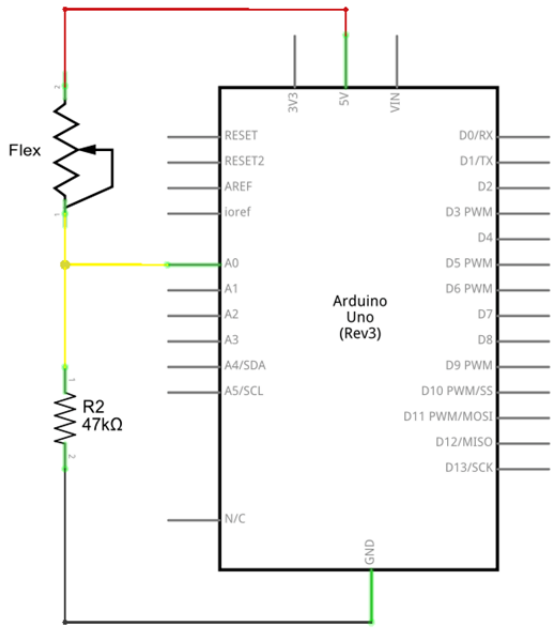


Figura 6. Circuito de referencia para la conexión de los sensores flexibles

Se recomienda que el valor de la resistencia usada para formar el divisor de tensión se encuentre entre 10kΩ y 100kΩ. Para calcular el valor ideal de dicha resistencia, es decir, el que permita cubrir el mayor rango de tensiones posible, se han calculado los valores mínimos y máximos de tensión teniendo en cuenta los valores mínimos y máximos de cada sensor flexible.

Tabla 2. Ejemplo de resultados para R óptima

R	Vmin	Vmax	Vmax-Vmin
48000	1,23211488	2,20758621	0,97547132
49000	1,21623711	2,19246575	0,97622864
50000	1,20076336	2,17755102	0,97678766
51000	1,18567839	2,16283784	0,97715945
52000	1,17096774	2,14832215	0,97735441
53000	1,15661765	2,134	0,97738235
54000	1,14261501	2,11986755	0,97725254
55000	1,12894737	2,10592105	0,97697368
56000	1,11560284	2,09215686	0,97655403
57000	1,10257009	2,07857143	0,97600134
58000	1,08983834	2,06516129	0,97532295
59000	1,07739726	2,05192308	0,97452582
60000	1,06523702	2,0388535	0,97361648

Así, realizando las siguientes operaciones, donde $V_{CC} = 3.3\ V$, R es la resistencia del divisor de tensión, y tanto $R_{flexmin}$ como $R_{flexmax}$ son valores tomados experimentalmente mediante el uso de un multímetro, se obtienen resultados semejantes a los reflejados en la Tabla 2. De esta manera, como puede verse en la Tabla 3, se obtiene que los valores óptimos se encuentran en torno a 51kΩ, motivo por el cual se escoge este valor para los cinco sensores requeridos.

$$V_{min} = V_{CC} \cdot \frac{R_{flexmin}}{R + R_{flexmin}} \qquad (1)$$

$$V_{max} = V_{CC} \cdot \frac{R_{flexmax}}{R + R_{flexmax}} \qquad (2)$$

Tabla 3. Valores de la resistencia para el divisor de tensión

	Rflexmin	Rflexmax	Rdiv ideal	Rdiv usada
46 - pulgar	32000	83000	51000	51000
51 - indice	25000	103400	51000	51000
2 - corazon	55000	120000	58000	51000
40 - anular	28000	126000	53000	51000
25 - meñique	26700	108400	54000	51000

2.2.3 Dispositivo LSM9DS1

El dispositivo LSM9DS1 de la empresa Sparkfun, es un circuito integrado capaz de medir tres propiedades clave del movimiento: velocidad angular, aceleración y dirección. Esto es posible porque cuenta con un acelerómetro, un giroscopio y un magnetómetro de tres ejes cada uno.

El giroscopio es el encargado de medir la velocidad angular, es decir, la velocidad de rotación alrededor de un eje. Las velocidades angulares se miden en grados por segundo, por lo general abreviado como DPS o °/s. El dispositivo LSM9DS1 puede medir hasta ± 2000 DPS, aunque esa escala también se puede configurar en 245 o 500 DPS para obtener una mayor resolución.

A su vez, el acelerómetro mide la aceleración, asociada al cambio de velocidad. La aceleración normalmente se mide en m/s^2 o en g (gravedades [alrededor de 9.8 m/s^2]). Si un objeto se encuentra inmóvil, experimenta aproximadamente 1 g de aceleración hacia el suelo (suponiendo que el suelo está en la tierra y el objeto está cerca del nivel del mar). El dispositivo LSM9DS1 mide su aceleración en g, y su escala se puede establecer en ± 2, 4, 8 o 16 g.

Finalmente, se encuentra el magnetómetro, que mide la potencia y la dirección de los campos magnéticos. El dispositivo LSM9DS1 mide los campos magnéticos en unidades de gauss (Gs), y puede establecer su escala de medición en ± 4, 8, 12 o incluso 16 Gs.

Se escoge este chip porque, a partir de estas tres propiedades, es capaz de proporcionar una gran cantidad de conocimiento sobre el movimiento y la orientación de un objeto.

Como se comentó anteriormente, el dispositivo LSM9DS1 cuenta con 3 ejes para cada propiedad, y por lo tanto mide cada una de estas propiedades de movimiento en tres dimensiones. Eso significa que produce nueve datos: aceleración en x/y/z, rotación angular en x/y/z, y fuerza del campo magnético en x/y/z.

En la Figura 7 puede verse que la placa tiene etiquetas que indican las orientaciones del eje del acelerómetro y el giroscopio.

Figura 7. *LSM9DS1 Breakout*

Además de poder medir una amplia variedad de vectores de movimiento, el dispositivo LSM9DS1 es compatible, tanto con la interfaz SPI (*Serial Peripheral*

Interface) como con I2C (*Inter-Integrated Circuit*). Por lo general, la interfaz SPI es más fácil de implementar, pero también requiere de más señales, cuatro en comparación con las dos señales del protocolo I2C.

En el presente TFG se utilizará el método de comunicación I2C. Se trata de un protocolo diseñado para permitir que múltiples circuitos integrados digitales esclavos se comuniquen con uno o más dispositivos maestros. Al igual que SPI, está diseñado para comunicaciones de corta distancia dentro de un solo dispositivo, y al igual que las interfaces seriales asíncronas (como RS-232 o UART), solo requiere de dos cables de señal para intercambiar información.

Así, en la Tabla 4 se recogen los pines del dispositivo LSM9DS1 necesarios en el diseño de la plataforma HW/SW del presente TFG.

Tabla 4. Pines para la comunicación I2C

Pin	Función	Descripción
GND	Conexión a tierra	0 V de tensión
VDD	Alimentación	Suministro de tensión al chip
SDA	I2C: *Serial Data*	Datos en serie I2C
SCL	*Serial Clock*	Reloj en serie I2C

Por último, se detalla a continuación el *pin-out* propuesto inicialmente para la interconexión del chip LSM9DS1 con el dispositivo *Photon* mediante la interfaz I2C:

LSM9DS1 **Photon**

SCL ---------- D1 (SCL)

SDA ---------- D0 (SDA)

VDD ---------- 3V3

GND ---------- GND

2.3 Componentes Software

En este apartado se recogen las herramientas proporcionadas por la empresa Particle, fundamentales para el desarrollo de la plataforma HW/SW objeto del presente TFG.

2.3.1 Plataforma de desarrollo de Particle

El dispositivo *Photon* se puede programar utilizando lenguaje C, C++ o, incluso, ensamblador. Particle utiliza *Wiring* [18], un *framework* de código abierto para microcontroladores, cuyas funciones son compatibles con la popular plataforma Arduino. Esto implica que la mayor parte del código desarrollado para Arduino podrá ejecutarse sin excesivas modificaciones. Además, Particle cuenta con una amplia colección de librerías creadas a partir de la contribución de terceros, lo cual puede simplificar las tareas de programación [19].

Particle Tinker

Particle cuenta con una aplicación de usuario denominada *Tinker* [20]. Se trata de una librería *firmware* capaz de ejecutar funciones básicas de *GPIO* a través de una API (*Application Programming Interface*). Funciona con una aplicación móvil para iOS y Android también llamada *Tinker*, que permite conmutar fácilmente pines digitales y realizar lecturas analógicas y digitales sin necesidad de escribir ni una línea de código [18], entre otras funciones básicas.

Como puede verse en la Figura 8, la aplicación consta de 16 pines: 8 analógicos situados a la izquierda, y 8 digitales situados a la derecha. Estos pines representan los 16 pines *GPIO* del dispositivo *Photon*.

Cada pin puede tener hasta cuatro funciones posibles que pueden seleccionarse pulsando en cualquiera de ellas. Esto permite interactuar directamente con el dispositivo *Photon* desde la aplicación. Las funciones posibles son:

- ***digitalWrite*.** Establece en un pin el nivel lógico *HIGH* (alto) o *LOW* (bajo), que lo conecta a 3.3V (la tensión máxima del sistema) o a *GND* (tierra), respectivamente. El pin *D7* está conectado a un *LED* integrado, de tal

manera que, si se establece el pin *D7* al nivel lógico *HIGH*, el *LED* se encenderá, y si se configura al nivel lógico *LOW*, se apagará.

- ***analogWrite.*** Establece en un pin un valor comprendido entre 0 y 255, donde el valor 0 está asociado al nivel lógico *LOW,* y el valor 255 al nivel lógico *HIGH*. Este proceso se corresponde con el establecimiento de una tensión a un valor comprendido entre 0 y 3.3V, pero dado que se trata de un sistema digital, usa un mecanismo basado en *PWM* (*Pulse-Width Modulation*). Esta función se podría usar, por ejemplo, para atenuar un LED.
- ***digitalRead***. Esta función permite leer el valor digital establecido en un pin, que puede leerse como *HIGH* o *LOW*, de tal manera que, conectando el pin a 3.3V, leería un nivel lógico *HIGH, y* conectándolo a *GND*, leería un nivel lógico *LOW*.
- ***analogRead.*** Esta función permite leer el valor analógico de un pin, que se corresponde con un valor comprendido entre 0 a 4095, donde el valor 0 se corresponde con *LOW (GND)* y el valor 4095 con *HIGH (3.3V).* Todos los pines analógicos (*A0* a *A7*) pueden gestionar esta función. *analogRead* es ideal para leer datos de sensores.

Figura 8. Aplicación móvil Tinker

En otras palabras, *Tinker* permite controlar y leer la tensión de los pines en un dispositivo *Photon*. Hace posible tanto encender un LED proporcionándole tensión como leer el voltaje de un sensor.

Particle Web IDE

A través de la web de Particle [21] se puede utilizar el *IDE (Integrated Development Environment) Particle Build* [22] para desarrollar código, compilarlo y cargarlo en la memoria *Flash* del dispositivo *Photon* de forma inalámbrica. Este método es el que se utilizará para el desarrollo de la parte f*irmware* en el presente TFG.

Atendiendo a la Figura 9, en la parte superior izquierda del entorno de desarrollo se encuentran tres botones importantes:

- **Flash:** inicia una actualización de *firmware* inalámbrica y carga el nuevo software en su dispositivo.
- **Verify:** compila el código sin cargarlo en el dispositivo para verificar que no hay errores.
- **Save:** guarda cualquier cambio que se realice en el código.

En la parte inferior, hay siete botones más, cuya funcionalidad es la siguiente:

- **Code:** muestra una lista de las aplicaciones *firmware* de usuario y permite seleccionar cuál editar o programar en el dispositivo.
- **Libraries:** contiene las librerías disponibles en la plataforma.
- **Help:** ofrece información sobre los diferentes estados del dispositivo, según el color y el parpadeo del led RGB que integra.
- **Docs:** accede a la documentación de Particle.
- **Devices:** muestra una lista con los dispositivos asociados a la cuenta de usuario y permite elegir cuál de ellos se va a utilizar, así como obtener información de los mismos.
- **Console:** permite acceder directamente a la herramienta *Particle Console*, que facilita la visualización de los eventos que se han publicado en la nube de Particle desde los diferentes dispositivos asociados a la cuenta de usuario, junto con las variables y funciones expuestas en ella desde la

aplicación programada. La nube de Particle, o *Particle Cloud*, es una plataforma que proporciona un medio seguro para interactuar con dispositivos de entrada/salida a través de APIs, y desarrollar servicios web inteligentes alrededor del dispositivo *Photon*.

- **Settings:** permite cambiar la contraseña, cerrar la sesión y obtener el *token* de acceso que contiene las credenciales de identificación del usuario que utiliza el entorno IDE de Particle.

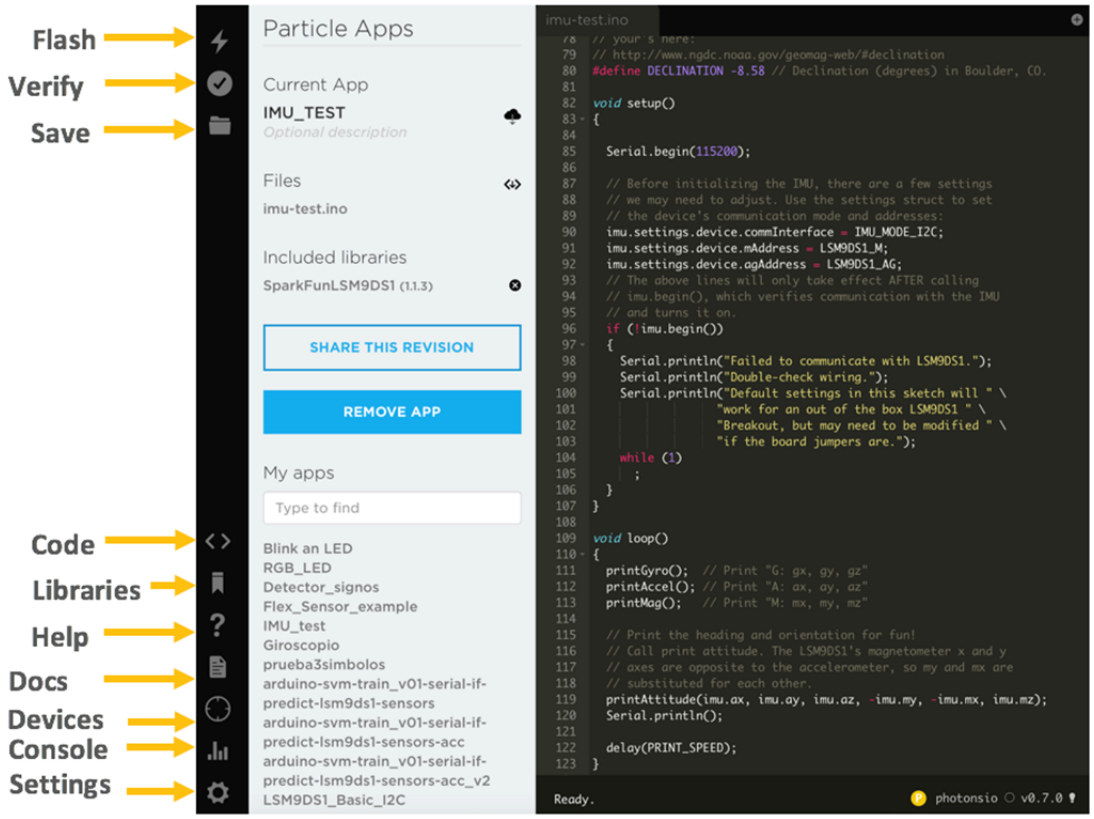


Figura 9. Entorno de desarrollo IDE Particle Build

Asimismo, en la sección **Particle Apps**, se muestra el nombre de la aplicación actual, así como una lista con el resto de aplicaciones. La aplicación abierta en el editor se muestra con el encabezado *Current App*, y debajo se detallan los archivos y las librerías incluidas en la aplicación. En este panel se encuentran los siguientes botones que permiten administrar la biblioteca de aplicaciones: crear, eliminar, cambiar el nombre, e incluso ejemplos de aplicaciones que sirven de referencia para crear aplicaciones propias, como se muestra en la Figura 10.

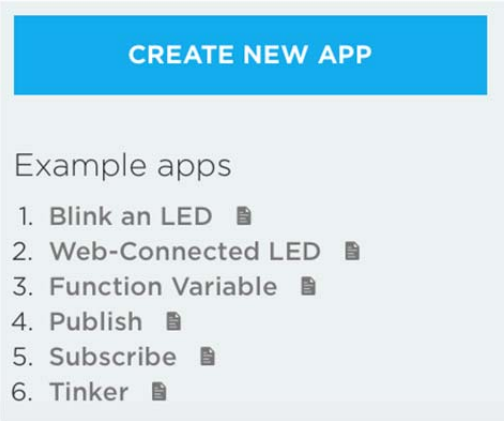


Figura 10. IDE Particle Build: ejemplos de aplicaciones

Por último, en la parte derecha del entorno de desarrollo se encuentra localizado el panel en el que se escribirá el código necesario para la aplicación de usuario que se desee implementar. En la parte inferior de dicho panel se halla una sección que determina el estado del código, mostrando los errores, el tamaño de memoria utilizada en el dispositivo, o si se ha realizado la programación del dispositivo de forma satisfactoria, entre otros.

Todos los procedimientos se realizan en la nube, desde el proceso de compilación del código hasta la programación del dispositivo. Además, se puede optar por editar desde el navegador, utilizando la página web de Particle, o descargar su IDE y editar el código de forma local.

Cabe destacar que la estructura que siguen todos los programas desarrollados para el dispositivo *Photon* (formato `.ino`) consta de dos funciones básicas: *setup()* y *loop()*. La función *setup()* es utilizada normalmente para inicializar pines y objetos, y sólo se ejecuta una vez al cargar el código en la memoria *Flash* del dispositivo, mientras que dentro de la función l*oop* se desarrolla el código que define el funcionamiento de la aplicación de usuario.

Además de *Particle Tinker* y *Particle Build*, la empresa Particle ofrece la herramienta *Particle CLI (Command Line Interface)* [23], que permite interactuar con la nube de Particle y sus dispositivos utilizando el lenguaje de programación *Node.JS* a través de una serie de comandos definidos que pueden ejecutarse desde la consola o terminal del sistema, tanto en Windows como en Mac OS X y Linux.

2.3.2 Validación funcional de los componentes

Una vez determinados los valores de las resistencias a utilizar con los sensores flexibles, se han calibrado de tal manera que el valor obtenido de la resistencia sin doblar sea próximo a cero. En la Figura 11 se muestra el código inicial desarrollado para la caracterización experimental de cada uno de los sensores flexibles, mediante el uso del dispositivo *Photon*, contenido en el fichero `flex-sensor-example.ino`.

flex-sensor-example.ino

```
const int FLEX_PIN = A0; // Pin connected to voltage divider output
// Measure the voltage at 5V and the actual resistance of your
// 47k resistor, and enter them below:
const float VCC = 3.3; // Measured voltage of Photon 3.3V line
const float R_DIV = 51000.0; // Measured resistance
// Upload the code, then try to adjust these values to more
// accurately calculate bend degree.
const float STRAIGHT_RESISTANCE = 55000.0;
const float BEND_RESISTANCE = 102000.0; // resistance at 90 deg
void setup() {
 Serial.begin(9600);
 pinMode(FLEX_PIN, INPUT);
}

void loop() {
 // Read the ADC, and calculate voltage and resistance from it
 int flexADC = analogRead(FLEX_PIN);
 float flexV = flexADC * VCC / 4095.0;
 float flexR = R_DIV*(VCC/flexV-1.0);
 Serial.println("Resistance: " + String(flexR) + " ohms");
 // Use the calculated resistance to estimate the sensor's bend angle:
float angle = map(flexR, STRAIGHT_RESISTANCE, BEND_RESISTANCE,0, 90.0);
 Serial.println("Bend: " + String(angle) + " degrees");
 Serial.println();
 delay(1000);
}
```

Figura 11. Código de prueba de los sensores flexibles adaptado al dispositivo Photon

Por un lado, han de modificarse los valores de las constantes VCC y R_DIV, en función del valor de tensión proporcionado por el dispositivo *Photon* a través del pin *3V3*, y de la resistencia utilizada para el divisor de tensión establecido con el sensor flexible, respectivamente. Además, los valores STRAIGHT_RESISTANCE y BEND_RESISTANCE se obtienen midiendo con un polímetro la resistencia que presenta cada sensor en las posiciones correspondientes a completamente estirado y flexionado, respectivamente.

Por otro lado, es importante tener en cuenta que el conversor Analógico-Digital *(Analog to Digital Converter, ADC)* asociado a la función *analoRead()* en el dispositivo *Photon* presenta una resolución de 12 bits, por lo que las tensiones de entrada comprendidas entre 0 y 3.3 V se mapean en valores enteros en el rango comprendido entre 0 y 4095. Por este motivo, para obtener la tensión en voltios (correspondiente a la variable `flexV`) es necesario multiplicar el valor `flexADC` (obtenido como salida de la función *analogRead())* por VCC y dividirlo entre 4095.

A partir de los parámetros definidos, en la línea 17 se almacena el valor leído del pin *A0,* correspondiente a la salida del divisor de tensión, para posteriormente, en la línea 18, obtener la tensión en voltios que permitirá calcular el valor de la resistencia en la línea 19. Seguidamente, en la línea 22 se utiliza la resistencia calculada para estimar el ángulo de doblez del sensor flexible.

De la misma manera, se ha utilizado el código `LSM9DS1_Basic_I2C.ino` [24] mostrado en la Figura 12 para verificar la funcionalidad del dispositivo LSM9DS1. En la Figura 13 pueden verse los datos obtenidos por pantalla a partir de una de las pruebas experimentales realizadas.

```
82  void setup()
83  {
84
85    Serial.begin(115200);
86
87    // Before initializing the IMU, there are a few settings
88    // we may need to adjust. Use the settings struct to set
89    // the device's communication mode and addresses:
90    imu.settings.device.commInterface = IMU_MODE_I2C;
91    imu.settings.device.mAddress = LSM9DS1_M;
92    imu.settings.device.agAddress = LSM9DS1_AG;
93    // The above lines will only take effect AFTER calling
94    // imu.begin(), which verifies communication with the IMU
95    // and turns it on.
96    if (!imu.begin())
97    {
98      Serial.println("Failed to communicate with LSM9DS1.");
99      Serial.println("Double-check wiring.");
100     Serial.println("Default settings in this sketch will " \
101                    "work for an out of the box LSM9DS1 " \
102                    "Breakout, but may need to be modified " \
103                    "if the board jumpers are.");
104     while (1)
105       ;
106   }
107 }
108
```

```
109 void loop()
110 {
111   printGyro();  // Print "G: gx, gy, gz"
112   printAccel(); // Print "A: ax, ay, az"
113   printMag();   // Print "M: mx, my, mz"
114
115   // Print the heading and orientation for fun!
116   // Call print attitude. The LSM9DS1's magnetometer x and y
117   // axes are opposite to the accelerometer, so my and mx are
118   // substituted for each other.
119   printAttitude(imu.ax, imu.ay, imu.az, -imu.my, -imu.mx, imu.mz);
120   Serial.println();
121
122   delay(PRINT_SPEED);
123 }
124
125 void printGyro()
126 {
127   // To read from the gyroscope, you must first call the
128   // readGyro() function. When this exits, it'll update the
129   // gx, gy, and gz variables with the most current data.
130   imu.readGyro();
131
132   // Now we can use the gx, gy, and gz variables as we please.
133   // Either print them as raw ADC values, or calculated in DPS.
134   Serial.print("G: ");
135 #ifdef PRINT_CALCULATED
136   // If you want to print calculated values, you can use the
137   // calcGyro helper function to convert a raw ADC value to
138   // DPS. Give the function the value that you want to convert.
139   Serial.print(imu.calcGyro(imu.gx), 2);
140   Serial.print(", ");
141   Serial.print(imu.calcGyro(imu.gy), 2);
142   Serial.print(", ");
143   Serial.print(imu.calcGyro(imu.gz), 2);
144   Serial.println(" deg/s");
145 #elif defined PRINT_RAW
146   Serial.print(imu.gx);
147   Serial.print(", ");
148   Serial.print(imu.gy);
149   Serial.print(", ");
150   Serial.println(imu.gz);
151 #endif
152 }
153
154 void printAccel()
155 {
156   // To read from the accelerometer, you must first call the
157   // readAccel() function. When this exits, it'll update the
158   // ax, ay, and az variables with the most current data.
159   imu.readAccel();
160
161   // Now we can use the ax, ay, and az variables as we please.
162   // Either print them as raw ADC values, or calculated in g's.
163   Serial.print("A: ");
164 #ifdef PRINT_CALCULATED
165   // If you want to print calculated values, you can use the
166   // calcAccel helper function to convert a raw ADC value to
167   // g's. Give the function the value that you want to convert.
168   Serial.print(imu.calcAccel(imu.ax), 2);
169   Serial.print(", ");
170   Serial.print(imu.calcAccel(imu.ay), 2);
171   Serial.print(", ");
172   Serial.print(imu.calcAccel(imu.az), 2);
173   Serial.println(" g");
174 #elif defined PRINT_RAW
175   Serial.print(imu.ax);
176   Serial.print(", ");
177   Serial.print(imu.ay);
178   Serial.print(", ");
```

```
  Serial.println(imu.az);
#endif
}
void printMag()
{
  // To read from the magnetometer, you must first call the
  // readMag() function. When this exits, it'll update the
  // mx, my, and mz variables with the most current data.
  imu.readMag();
  // Now we can use the mx, my, and mz variables as we please.
  // Either print them as raw ADC values, or calculated in Gauss.
  Serial.print("M: ");
#ifdef PRINT_CALCULATED
  // If you want to print calculated values, you can use the
  // calcMag helper function to convert a raw ADC value to
  // Gauss. Give the function the value that you want to convert.
  Serial.print(imu.calcMag(imu.mx), 2);
  Serial.print(", ");
  Serial.print(imu.calcMag(imu.my), 2);
  Serial.print(", ");
  Serial.print(imu.calcMag(imu.mz), 2);
  Serial.println(" gauss");
#elif defined PRINT_RAW
  Serial.print(imu.mx);
  Serial.print(", ");
  Serial.print(imu.my);
  Serial.print(", ");
  Serial.println(imu.mz);
#endif
}
// Calculate pitch, roll, and heading.
// Pitch/roll calculations take from this app note:
// Heading calculations taken from this app note:
void printAttitude(
float ax, float ay, float az, float mx, float my, float mz)
{
  float roll = atan2(ay, az);
  float pitch = atan2(-ax, sqrt(ay * ay + az * az));
  float heading;
  if (my == 0)
    heading = (mx < 0) ? 180.0 : 0;
  else
    heading = atan2(mx, my);

  heading -= DECLINATION * M_PI / 180;

  if (heading > M_PI) heading -= (2 * M_PI);
  else if (heading < -M_PI) heading += (2 * M_PI);
  else if (heading < 0) heading += 2 * M_PI;
  // Convert everything from radians to degrees:
  heading *= 180.0 / M_PI;
  pitch *= 180.0 / M_PI;
  roll  *= 180.0 / M_PI;
  Serial.print("Pitch, Roll: ");
  Serial.print(pitch, 2);
  Serial.print(", ");
  Serial.println(roll, 2);
  Serial.print("Heading: "); Serial.println(heading, 2);
}
```

Figura 12. Código de prueba del dispositivo LSM9DS1

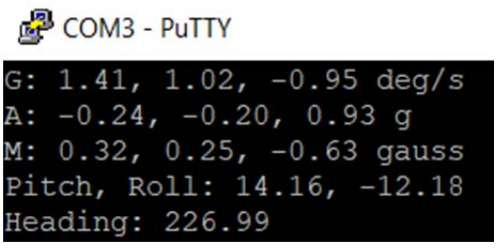


Figura 13. Datos obtenidos del dispositivo LSM9DS1

Capítulo 3. Proceso de clasificación

Una vez establecidos los recursos necesarios inicialmente para detectar los símbolos requeridos en la plataforma HW/SW a desarrollar en el presente TFG, se estudia la manera de realizar el proceso de clasificación en el dispositivo *Photon* a partir de diferentes datos obtenidos de las resistencias flexibles y del acelerómetro integrado en el dispositivo LSM9DS1.

3.1 Introducción

La clasificación es una técnica muy útil que permite realizar un reconocimiento de patrones a partir de un conjunto de datos considerados como muestras, de tal manera que se consigue realizar una predicción cuya eficiencia estará determinada por un porcentaje de acierto.

Las principales técnicas de clasificación para agrupar muestras según un criterio o método establecido son:

- **Clasificación supervisada.** Este tipo de clasificación cuenta con modelos ya clasificados. Se divide en dos fases: la primera es un entrenamiento que permite diseñar el clasificador, y la segunda es el test de validación que permite clasificar nuevas muestras. Esta técnica se utiliza por lo general para el diagnóstico de enfermedades, predicción de quiebra en empresas, minería de datos, etc.
- **Clasificación no supervisada.** En este caso no se cuenta con un conocimiento previo, es decir, se parte de muestras que no pertenecen a una clase concreta y, por lo tanto, la finalidad de esta técnica es identificar diferentes clases a partir de características comunes entre las muestras y algunos parámetros que limiten el número de clases.

Algunos de los métodos más utilizados en las diferentes técnicas de clasificación son los siguientes:

- **Árboles de decisión.** Se trata de un modelo de predicción que permite analizar las alternativas disponibles ante una toma de decisiones a partir de una representación esquemática de las mismas.
- **Redes bayesianas.** Es un método de clasificación probabilístico basado en el teorema de Bayes que representa un conjunto de variables aleatorias y sus dependencias condicionales a través de un grafo acíclico dirigido.
- **Redes neuronales.** Una red neuronal es un modelo matemático inspirado en el comportamiento biológico de las neuronas, y consiste en un método que combina los parámetros dados para predecir un resultado.
- **El k-vecino más cercano.** Es un método de clasificación supervisada que permite estimar una función de densidad para predecir un valor concreto para una clase concreta. Con este algoritmo, se tiene en cuenta el vecino más próximo a la hora de determinar la clase a la que pertenece una nueva muestra.

- **K medias.** Es un algoritmo de clasificación no supervisada que determina grupos seleccionando K puntos al azar y agrupa las muestras teniendo en cuenta la cercanía a esos grupos establecidos.
- **SVM (*Support Vector Machines*).** Las máquinas de vectores de soporte constituyen un método de clasificación supervisada que, a partir de un conjunto de muestras pertenecientes a clases diferentes, genera un modelo capaz de predecir la clase de una nueva muestra.

En el caso concreto del presente TFG, se utilizará la técnica basada en SVM para la clasificación de los datos proporcionados por los elementos que detectan la posición de los dedos y de la mano del usuario, de tal manera que se pueda interpretar el símbolo representado en cada momento. Para ello, es necesario realizar un proceso de entrenamiento, es decir, caracterizar la clasificación mediante un conjunto de datos de prueba. Tras este entrenamiento, se consigue un modelo a partir del cual se clasificará cualquier otro caso símbolo existente en el futuro.

3.2 Clasificación SVM

SVM representa un conjunto de algoritmos de aprendizaje supervisado desarrollados por Bladimir Vapnik y sus colaboradores para la resolución de problemas de clasificación binaria [25]. Dicha resolución está basada en una primera fase de entrenamiento donde, a partir de un conjunto de muestras, se pueden etiquetar distintas clases, y una segunda fase donde se construye un modelo capaz de predecir la clase de una nueva muestra.

Aunque originalmente las SVM fueron pensadas para resolver problemas de clasificación binaria, actualmente se utilizan para resolver otros tipos de problemas (regresión, agrupamiento, multiclasificación). Además, han sido utilizadas con éxito en diversos campos como visión artificial, reconocimiento de caracteres, categorización de texto, etc.

Dentro de la tarea de clasificación, las SVM pertenecen a la categoría de los clasificadores lineales, puesto que utilizan separadores lineales o hiperplanos, ya sea

en el espacio original de las muestras de entrada, si éstas son separables, o en un espacio transformado (espacio de características), si las muestras no son separables linealmente en el espacio original [26].

Mientras la mayoría de los métodos de aprendizaje se centran en minimizar los errores cometidos por el modelo generado a partir de los ejemplos de entrenamiento (error empírico), el sesgo inductivo asociado a las SVM se basa en la minimización del denominado riesgo estructural. La idea es seleccionar un hiperplano de separación que equidista de las muestras más cercanas de cada clase para, de esta forma, conseguir un margen de separación óptimo, como se representa en la Figura 14. Además, a la hora de definir el hiperplano, sólo se consideran las muestras de entrenamiento de cada clase que caen justo en la frontera de dichos márgenes. Estas muestras reciben el nombre de vectores soporte.

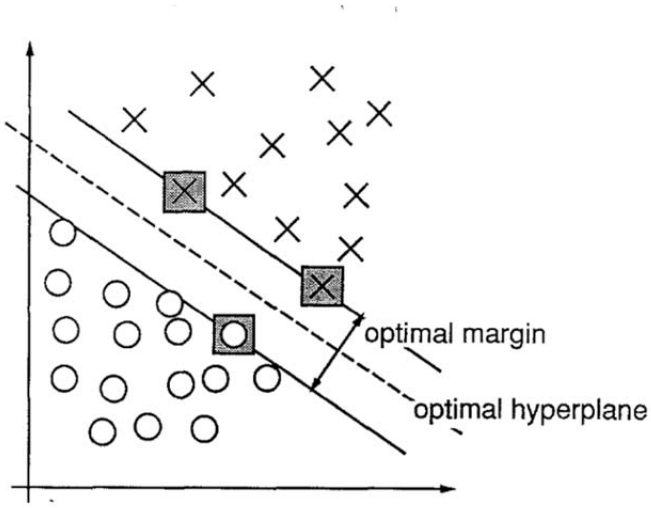


Figura 14. Ejemplo de problema binario separable

La manera más simple de realizar la separación entre clases es mediante una línea recta, un plano recto o un hiperplano N-dimensional. Sin embargo, en la mayoría de las aplicaciones el algoritmo SVM debe tratar con problemas no lineales, es decir, problemas donde las clases no son linealmente separables. En estos casos es necesario utilizar funciones *kernel* que permitan definir espacios transformados de alta dimensionalidad y buscar hiperplanos de separación óptimos en dichos espacios transformados. A continuación se recogen algunos ejemplos de funciones *kernel:*

- Polinomial-homogénea: $K(x_i, y_j) = (x_i \cdot y_j)^n$ (3)
- Perceptrón: $K(x_i, y_j) = ||x_i - y_j||$ (4)
- Función de base radial Gaussiana: $K(x_i, y_j) = e^{-\gamma||x_i - y_j||^2}$ (5)
- Sigmoid: $K(x_i, y_j) = \tanh(\gamma < x_i, y_j > + \tau)$ (6)

El problema de clasificación planteado en el presente TFG consta de 30 clases (una por cada signo del alfabeto dactilológico español) que no son linealmente separables, por lo que es necesario utilizar una función *kernel*. Concretamente, se ha decidido utilizar la función de base radial Gaussiana o RBF (5), que define una distancia entre dos muestras a partir de una función gaussiana. Se escoge esta función por ser un *kernel* de uso general que proporciona buenos resultados en la mayoría de las situaciones.

Al realizar clasificaciones usando este método existen dos tipos de SVM: *C-SVM* y *nu-SVM*, donde *C* y *nu* son parámetros de regularización que ayudan a implementar una penalización en las clasificaciones erróneas que se realizan al separar las clases. Por lo tanto, ayudan a mejorar la precisión de la salida. *C* varía de 0 a infinito y puede ser un poco difícil de estimar y usar. En cambio, *nu* opera entre 0-1 y representa el límite inferior y superior en el número de ejemplos que son vectores de soporte y que se encuentran en el lado equivocado del hiperplano. Ambos presentas capacidades de clasificación similares, siendo *C-SVM* el utilizado en el presente TFG.

Un clasificador *C-SVM* que utiliza un kernel RBF tiene dos parámetros: *gamma* y *C*. Por un lado, el parámetro *gamma* define hasta dónde llega la influencia de un único ejemplo de entrenamiento, de tal manera que, cuando tiene un valor bajo, la curva del límite de decisión es muy baja y, en consecuencia, la región de decisión es muy amplia, mientras que cuando tiene un valor alto, el algoritmo se esfuerza más en evitar la clasificación errónea de los datos de entrenamiento, lo que conduce a un ajuste excesivo. En la Figura 15[27] se representa un ejemplo para distintos valores de *gamma*.

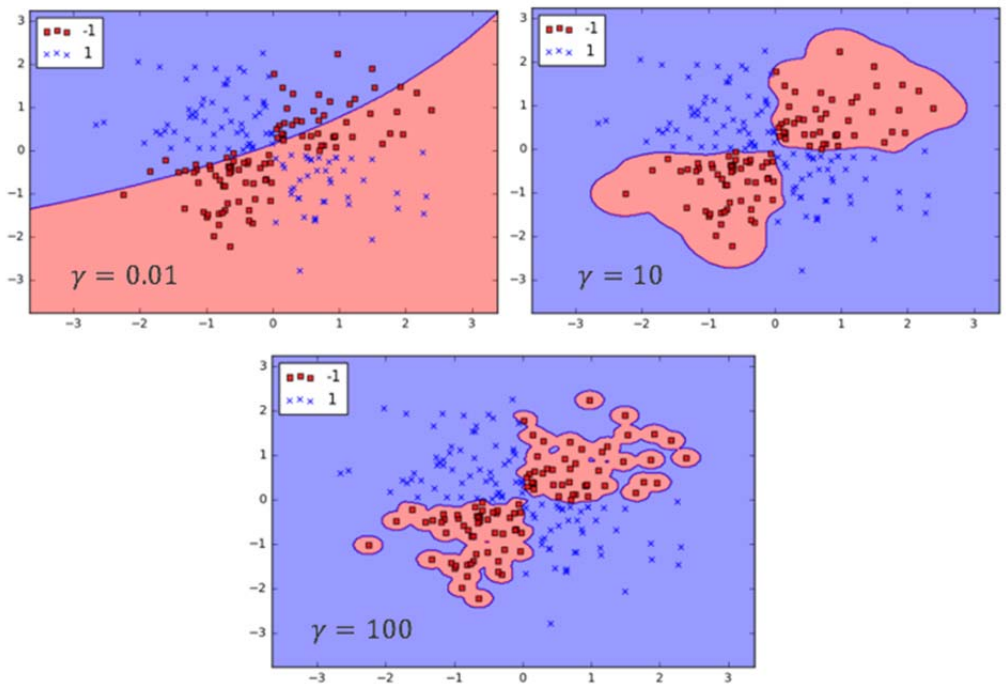


Figura 15. Ejemplo de clasificación binaria con kernel RBF para distintos valores de *gamma*

Por otro lado, el parámetro *C* es conocido como factor de regulación o parámetro de penalización del término de error. Cuando el valor del parámetro *C* es pequeño, el clasificador funciona correctamente con puntos de datos mal clasificados (alto sesgo, baja varianza). Cuando el valor del parámetro *C* es grande, el clasificador es muy penalizado por datos mal clasificados y, por lo tanto, trata de evitar cualquier punto de datos mal clasificado (bajo sesgo, alta varianza). En la Figura 16 se representa un ejemplo para distintos valores del parámetro *C*.

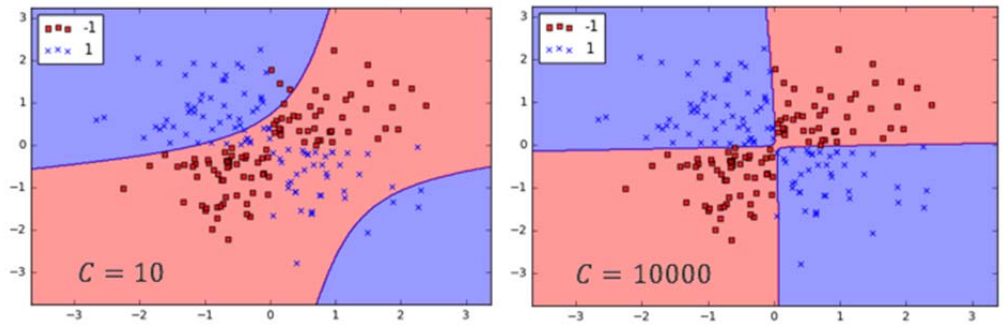


Figura 16. Ejemplo de clasificación binaria con kernel RBF para distintos valores de C

Mediante la técnica de validación cruzada [28] es posible encontrar los parámetros de *gamma* y *C* que aporten mayor precisión. Sin embargo, en el presente TFG se asignan

inicialmente los valores establecidos por defecto para estos parámetros, es decir, *C=100 y γ=0.0001.*

En resumen, el procedimiento propuesto para el uso de SVM es el siguiente:

- Transformar los datos capturados por los dispositivos sensores al formato de un paquete SVM.
- Realizar un escalado simple de los datos para mejorar la precisión.
- Utilizar el kernel RBF.
- Asignar valores a los parámetros C y γ.
- Utilizar estos parámetros para el entrenamiento.
- Realizar pruebas con el modelo generado a partir del entrenamiento.

3.3 LIBSVM

En la actualidad existen varios paquetes de software libre orientados a la implementación de SVM, siendo LIBSVM [29] el usado en este TFG. Se trata de una librería que implementa diferentes formulaciones de SVM para la clasificación, la regresión y la estimación de distribución, dando la posibilidad de usar diferentes tipos de kernels.

En concordancia con la descripción de SVM, el uso de LIBSVM implica dos pasos: en primer lugar, el entrenamiento de un conjunto de datos para obtener un modelo y, en segundo lugar, el uso del modelo para predecir información.

Además, la librería LIBSVM soporta las siguientes tareas de aprendizaje:

- SVC (*Support Vector Classification*), utilizada para dos clases y múltiples clases.
- SVR (*Support Vector Regression*).
- SVM de una sola clase.

Debido a que el problema a tratar en el presente TFG implica la diferenciación entre múltiples clases, se utilizará un clasificador SVC. Para la clasificación de clases múltiples, LIBSVM implementa el método "*one-against-one*". Esto implica que, si *k* es el número de clases, entonces se construyen *k*(k-1)/2* clasificadores y cada uno

entrena datos de dos clases. Para los datos de entrenamiento de las clases *i-ésima* y *-j-ésima*, se resuelve un problema de clasificación de dos clases.

En la clasificación se utiliza una estrategia de votación: cada clasificación binaria se considera una votación en la que se pueden emitir los votos para todas las muestras de datos *x*; al final, una muestra se designa para estar en una clase con el número máximo de votos. En caso de que dos clases tengan votos idénticos, se elige la clase que aparece primero en la matriz de almacenamiento de nombres de clase.

Organización del código

El paquete LIBSVM está estructurado de la siguiente manera:

- Directorio principal: programas implementados en lenguaje C/C++ y datos de muestra. En particular, el archivo `svm.cpp` implementa algoritmos de entrenamiento y prueba.
- El subdirectorio de la herramienta: este subdirectorio incluye herramientas para verificar el formato de los datos y para seleccionar los parámetros de SVM.
- Otros subdirectorios: contienen archivos binarios precompilados e interfaces a otros lenguajes / software.

En concreto, el archivo `svm.cpp` contiene todas las funciones y los algoritmos de entrenamiento y prueba, siendo *svm_train* y *svm_predict* las principales subrutinas.

Asimismo, dentro de LIBSVM se encuentra el fichero `svm-train.c`. Se trata del fichero que permite recoger los parámetros por línea de comandos (valores de *gamma* y *C*) y ajustarlos teniendo en cuenta los valores establecidos por defecto, de tal manera que, si no existe ningún error, se llama a la subrutina *svm_train* que se encuentra en el fichero `svm.cpp` y que se encarga de generar el modelo de clasificación.

Llegados a este punto, la explicación se centrará en la opción *C-SVC*, que como ya se ha comentado, es el método utilizado para realizar la clasificación multiclase, y mediante el cual se obtienen los vectores de soporte y otra información de importancia, como por ejemplo los parámetros del *kernel* en el modelo de predicción.

Además, se describen con mayor detalle los ficheros `svm-predict.c`. y `svm-scale.c`, ya que posteriormente se realizará una adaptación de los mismos al dispositivo *Photon*.

svm-predict.c

El fichero `svm-predict.c` contiene las funciones mostradas en la Figura 17. Para comprender mejor el método de clasificación, se prestará atención a la función *predict*, así como a las funciones más relevantes que derivan de ésta.

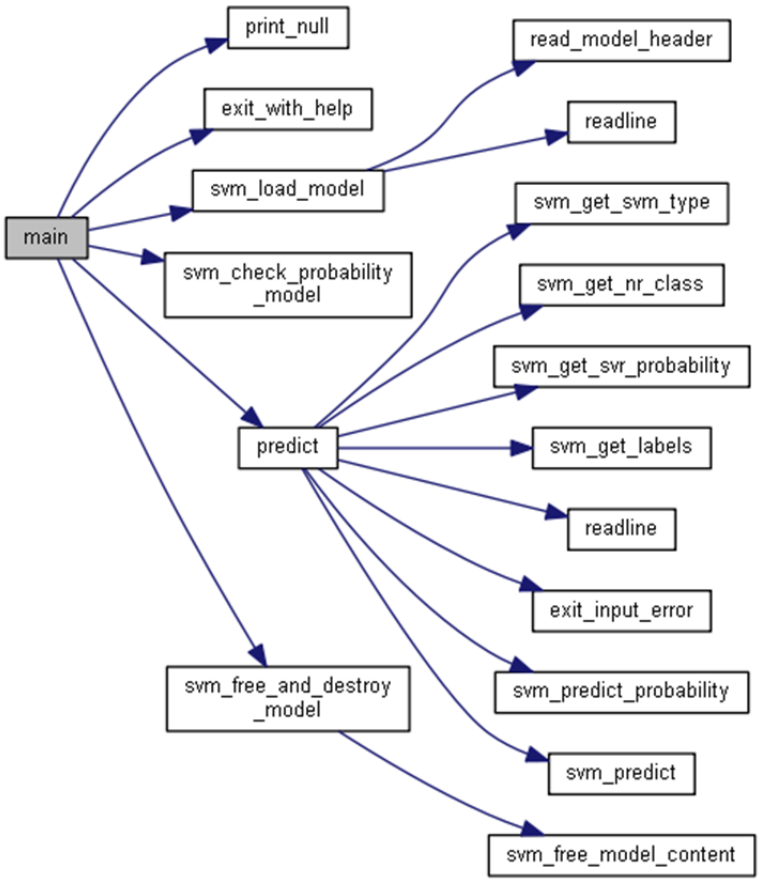


Figura 17. Dependencias de funciones de *svm-predict.c.*

- *svm_get_svm_type* es la función que determina el parámetro *svm_type* del modelo, cuyos valores posibles vienen definidos en el fichero `svm.h`.
- *svm_get_nr_class* proporciona el número de clases para un modelo de clasificación.

- *svm_get_labels* genera el nombre de etiquetas en una matriz denominada *label* para un modelo de clasificación.
- *svm_predict_probability* es la función encargada del proceso de clasificación en un vector de prueba *x* a partir de un modelo con información de probabilidad. En la Figura 18 puede verse que esta función proporciona las estimaciones de probabilidad *nr_class* en la matriz *prob_estimates* para devolver la clase con la probabilidad más alta.

```
120  if (predict_probability && (svm_type==C_SVC || svm_type==NU_SVC))
121  {
122      predict_label = svm_predict_probability(model,x,prob_estimates);
123      fprintf(output,"%g",predict_label);
124      for(j=0;j<nr_class;j++)
125          fprintf(output," %g",prob_estimates[j]);
126      fprintf(output,"\n");
127  }
```

Figura 18. Sección de código para *c_svc.*

En la Figura 19 se recogen las funciones que derivan de *svm_predict_probability*, de las cuales se destaca la función *svm_predict*. Esta función es la encargada de llevar a cabo el proceso de clasificación en un vector de prueba *x*, de tal manera que, para un modelo de clasificación, se devuelve la clase predicha para *x*.

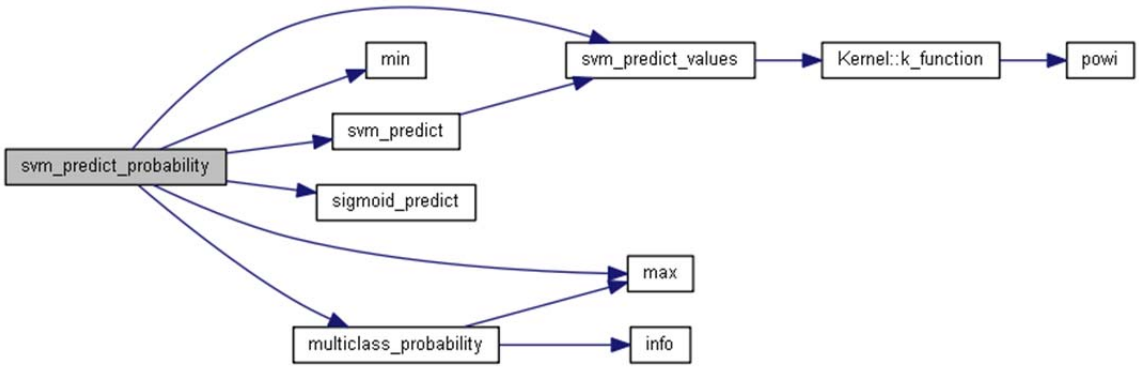


Figura 19. Dependencias de funciones con *svm_predict_probability.*

En la Figura 20 se muestra la sección de código correspondiente a la función *svm_predict* de la librería LIBSVM. En ella puede verse que se llama a la función

svm_predict_values, encargada de proporcionar valores de decisión en un vector de prueba *x* a partir de un modelo y de devolver la etiqueta o clase predicha.

```
2577  double svm_predict(const svm_model *model, const svm_node *x)
2578  {
2579      int nr_class = model->nr_class;
2580      double *dec_values;
2581      if(model->param.svm_type == ONE_CLASS ||
2582         model->param.svm_type == EPSILON_SVR ||
2583         model->param.svm_type == NU_SVR)
2584          dec_values = Malloc(double, 1);
2585      else
2586          dec_values = Malloc(double, nr_class*(nr_class-1)/2);
2587      double pred_result = svm_predict_values(model, x, dec_values);
2588      free(dec_values);
2589      return pred_result;
2590  }
```

Figura 20. Función *svm_predict* de la librería LIBSVM

El código de la función *svm_predict_values* se presenta en la Figura 21. En ella puede verse que, para un modelo de clasificación con *nr_class* clases, esta función proporciona *nr_class*(nr_class-1)/2* valores de decisión en la matriz *dec_values*. El valor devuelto es la clase predicha para la muestra *x*.

```
2501  double svm_predict_values(const svm_model *model, const svm_node *x, double* dec_values)
2502  {
2503      int i;
2504      if(model->param.svm_type == ONE_CLASS ||
2505         model->param.svm_type == EPSILON_SVR ||
2506         model->param.svm_type == NU_SVR)
2507      {
2508          double *sv_coef = model->sv_coef[0];
2509          double sum = 0;
2510          for(i=0;i<model->l;i++)
2511              sum += sv_coef[i] * Kernel::k_function(x,model->SV[i],model->param);
2512          sum -= model->rho[0];
2513          *dec_values = sum;
2514
2515          if(model->param.svm_type == ONE_CLASS)
2516              return (sum>0)?1:-1;
2517          else
2518              return sum;
2519      }
2520      else
2521      {
2522          int nr_class = model->nr_class;
2523          int l = model->l;
2524
2525          double *kvalue = Malloc(double,l);
2526          for(i=0;i<l;i++)
2527              kvalue[i] = Kernel::k_function(x,model->SV[i],model->param);
2528
2529          int *start = Malloc(int,nr_class);
2530          start[0] = 0;
2531          for(i=1;i<nr_class;i++)
2532              start[i] = start[i-1]+model->nSV[i-1];
2533
2534          int *vote = Malloc(int,nr_class);
2535          for(i=0;i<nr_class;i++)
2536              vote[i] = 0;
```

```
2537
2538            int p=0;
2539            for(i=0;i<nr_class;i++)
2540                for(int j=i+1;j<nr_class;j++)
2541                {
2542                    double sum = 0;
2543                    int si = start[i];
2544                    int sj = start[j];
2545                    int ci = model->nSV[i];
2546                    int cj = model->nSV[j];
2547
2548                    int k;
2549                    double *coef1 = model->sv_coef[j-1];
2550                    double *coef2 = model->sv_coef[i];
2551                    for(k=0;k<ci;k++)
2552                        sum += coef1[si+k] * kvalue[si+k];
2553                    for(k=0;k<cj;k++)
2554                        sum += coef2[sj+k] * kvalue[sj+k];
2555                    sum -= model->rho[p];
2556                    dec_values[p] = sum;
2557
2558                    if(dec_values[p] > 0)
2559                        ++vote[i];
2560                    else
2561                        ++vote[j];
2562                    p++;
2563                }
2564
2565            int vote_max_idx = 0;
2566            for(i=1;i<nr_class;i++)
2567                if(vote[i] > vote[vote_max_idx])
2568                    vote_max_idx = i;
2569
2570            free(kvalue);
2571            free(start);
2572            free(vote);
2573            return model->label[vote_max_idx];
2574        }
2575    }
```

Figura 21. Función *svm_predict_values* de la librería LIBSVM

Seguidamente, en la Figura 22 se muestra la función *kernel* de la que hace uso la función *svm_predict_values* en las líneas 2511 y 2527. En esta sección de código se recogen los distintos casos posibles según el *kernel* a utilizar que, como ya se ha comentado, en esta ocasión es RBF.

```
316 double Kernel::k_function(const svm_node *x, const svm_node *y,
317                          const svm_parameter& param)
318 {
319     switch(param.kernel_type)
320     {
321         case LINEAR:
322             return dot(x,y);
323         case POLY:
324             return powi(param.gamma*dot(x,y)+param.coef0,param.degree);
325         case RBF:
326         {
327             double sum = 0;
328             while(x->index != -1 && y->index !=-1)
329             {
330                 if(x->index == y->index)
331                 {
```

```
332                 double d = x->value - y->value;
333                 sum += d*d;
334                 ++x;
335                 ++y;
336             }
337             else
338             {
339                 if(x->index > y->index)
340                 {
341                     sum += y->value * y->value;
342                     ++y;
343                 }
344                 else
345                 {
346                     sum += x->value * x->value;
347                     ++x;
348                 }
349             }
350         }
351
352         while(x->index != -1)
353         {
354             sum += x->value * x->value;
355             ++x;
356         }
357
358         while(y->index != -1)
359         {
360             sum += y->value * y->value;
361             ++y;
362         }
363
364         return exp(-param.gamma*sum);
365     }
366     case SIGMOID:
367         return tanh(param.gamma*dot(x,y)+param.coef0);
368     case PRECOMPUTED:  //x: test (validation), y: SV
369         return x[(int)(y->value)].value;
370     default:
371         return 0;  // Unreachable
372 }
373 }
```

Figura 22. Función kernel de la librería LIBSVM

3.3.1 Adaptación del código para el dispositivo *Photon*

Para implementar la funcionalidad de clasificación en el dispositivo *Photon* se ha realizado una adaptación del código proporcionado por la librería LIBSVM. Cabe destacar que, a diferencia de LIBSVM, los datos de entrada (las medidas que se toman de los sensores en cada momento) están sin escalar. Es por ello que, en un principio, se implementa esta función aparte, es decir, primero se recogen los datos obtenidos de los diferentes sensores, se escalan sus valores, y posteriormente se introducen en el fichero `svm-train-predict-ble.ino`, que contiene el código *firmware* a ejecutar en el dispositivo *Photon*.

Como referencia, en la Tabla 5 se muestra la relación de parámetros entre el código de predicción tomado como referencia de LIBSVM, y el adaptado para el dispositivo *Photon*.

Tabla 5.Relación de parámetros entre el código LIBSVM y la adaptación para el dispositivo *Photon*

Predict en LIBSVM	**Predict en Photon**
p	rhoCounter
sum	accumulator
vote	result
vote_max_idx	recognizedClass
d	temp
x->value	u[j]
y->value	v[j]

El resultado de la adaptación para realizar una predicción de la clase a la que pertenecen los datos medidos en cada momento se recoge en la función *svm_predict()*, mostrada en la Figura 23, donde el parámetro *nr_sv* es un vector que contiene el número de muestras tomadas para cada símbolo, y tanto *supportVectors* como *valuesForSupport* contienen los datos obtenidos a partir del modelo generado tras el entrenamiento. Como se observa en la línea 866, esta función recibe el parámetro *sensor* con los datos obtenidos a partir de los diferentes sensores para, a partir de los vectores de soporte (líneas 877 y 878), comparar cada clase con el resto y decidir así si el problema es más probable para una clase o para la otra. Esto es posible gracias a la función *svm_evaluate* utilizada en las líneas 889 y 890. La sección de código de esta función se recoge en la Figura 24. Seguidamente, en la línea 894 se realiza el recuento de votos que decidirá la clase a la que pertenecen los datos medidos, la cual se asigna posteriormente a la variable *recognizedClass* para imprimirse por pantalla.

```
// prediction of the class to which the newly measured data belongs to.
inline void svm_predict(int sensor[]){
  int recognizedClass = 1;
  float scaledSensor[VEC_DIM];
  scale(sensor,scaledSensor);
  int rhoCounter = 0;

  for(int i=0; i<NR_CLASS; i++){
    for(int j=i+1; j<NR_CLASS; j++){
      float accumulator = 0;

      float* sv_class1 = (float*) supportVectors[i];
      float* sv_class2 = (float*) supportVectors[j];
      float coeffs1[nr_sv[i]];
      for (int k=0; k<nr_sv[i]; k++){
        coeffs1[k] = valuesForSupport[i][(nr_sv[i]*(j-1))+k];
      }

      float coeffs2[nr_sv[j]];
      for (int k=0; k<nr_sv[j]; k++){
        coeffs2[k] = valuesForSupport[j][(nr_sv[j]*(i))+k];
      }

      accumulator += svm_evaluate(nr_sv[i], coeffs1, sv_class1, scaledSensor);
      accumulator += svm_evaluate(nr_sv[j], coeffs2, sv_class2, scaledSensor);
      float rhoNr = rho[rhoCounter];
      accumulator -= rhoNr;

      if (accumulator > 0) {
        result[i]++;
        }
      else {
        result[j]++;
        }
      rhoCounter++;
      }
  }

   // find out which class has the most votes
  int temp = 0;
  for(int t = 0; t < NR_CLASS; t++){
    if(result[temp] <= result[t]){
      recognizedClass = t;
      temp = t;
    }
  }

  // output of the predicted class.
  Serial.print("> recognized letter/number = ");
  Serial.println(recognizedClass, DEC);
  delay(500);
  for(int q = 0; q < NR_CLASS; q++){
    result[q]=0;
  }
}
```

Figura 23. *svm_predict()*

```
699 inline float svm_evaluate(int n_sv, float* coeffs, float* sv_class, float* sensors){
700   float result= 0;
701   float sv_current[VEC_DIM];
702
703   for (int i=0; i<n_sv; i++){
704     float coeff = coeffs[i];
705
706     for (int j=0; j<VEC_DIM; j++){
707       sv_current[j] = sv_class[(i*VEC_DIM)+j];
708     }
709
710     result += coeff * rbf_kernel(sv_current, sensors);
711
712   }
713
714   return result;
715
716 }
```

Figura 24. *svm_evaluate()*

Además, al igual que en el código proporcionado por LIBSVM, se utiliza una función *kernel*. Sin embargo, como puede verse en la Figura 25, en la adaptación se simplifica el código, especificando la parte exclusiva para el *kernel RBF.*

```
666 inline float rbf_kernel(float* u, float* v){
667   float result=0;
668   // calculate squared norm
669   for (int j=0; j<VEC_DIM; j++){
670     float temp = u[j] - v[j];
671
672     result += temp * temp;
673   }
674   return exp(-GAMMA * result);
675 }
```

Figura 25. *rbf_kernel()*

3.4 Plataforma HW/SW inicial

Una vez establecido el método de clasificación, es posible implementar una primera versión de la plataforma HW/SW que permita realizar el proceso de entrenamiento y la identificación de gran parte de los signos del alfabeto dactilológico asociado a la Lengua de Signos Española (LSE).

Inicialmente se requiere la integración del dispositivo *Photon* con los sensores flexibles y el dispositivo LSM9DS1 con el fin de recoger información de movimiento que permita identificar aquellos signos del alfabeto que no son estáticos. Además, se añade un pulsador, tanto para iniciar la fase de entrenamiento, como para iniciar y detener la

fase de predicción de signos. En la Figura 26 se ilustra un esquema del conexionado de todos los dispositivos requeridos en la plataforma HW/SW inicial.

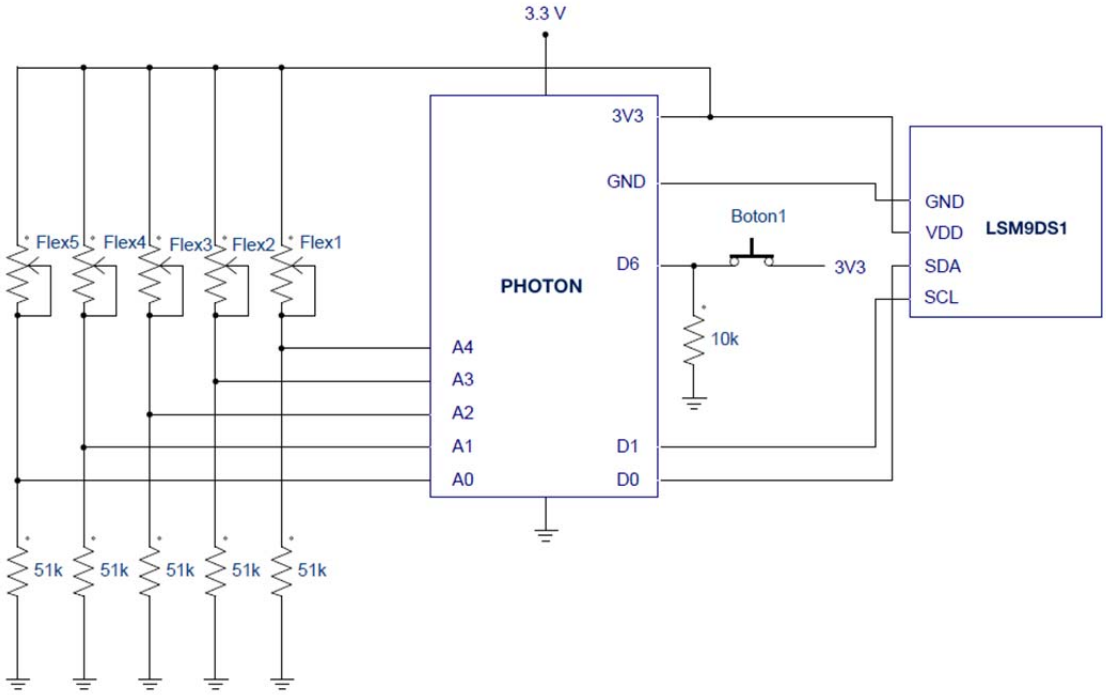


Figura 26. Esquemático de la plataforma inicial

En la Tabla 6 se recoge la asignación de dedos que se le ha dado a cada sensor flexible en el presente TFG.

Tabla 6. Relación de los *flex sensors*

Flex Sensor	**Dedo**
Flex5	Meñique
Flex4	Anular
Flex3	Corazón
Flex2	Índice
Flex1	Pulgar

Así, tras conectar todos los elementos sobre una *protoboard,* se obtiene la plataforma HW/SW inicial mostrada en la Figura 27.

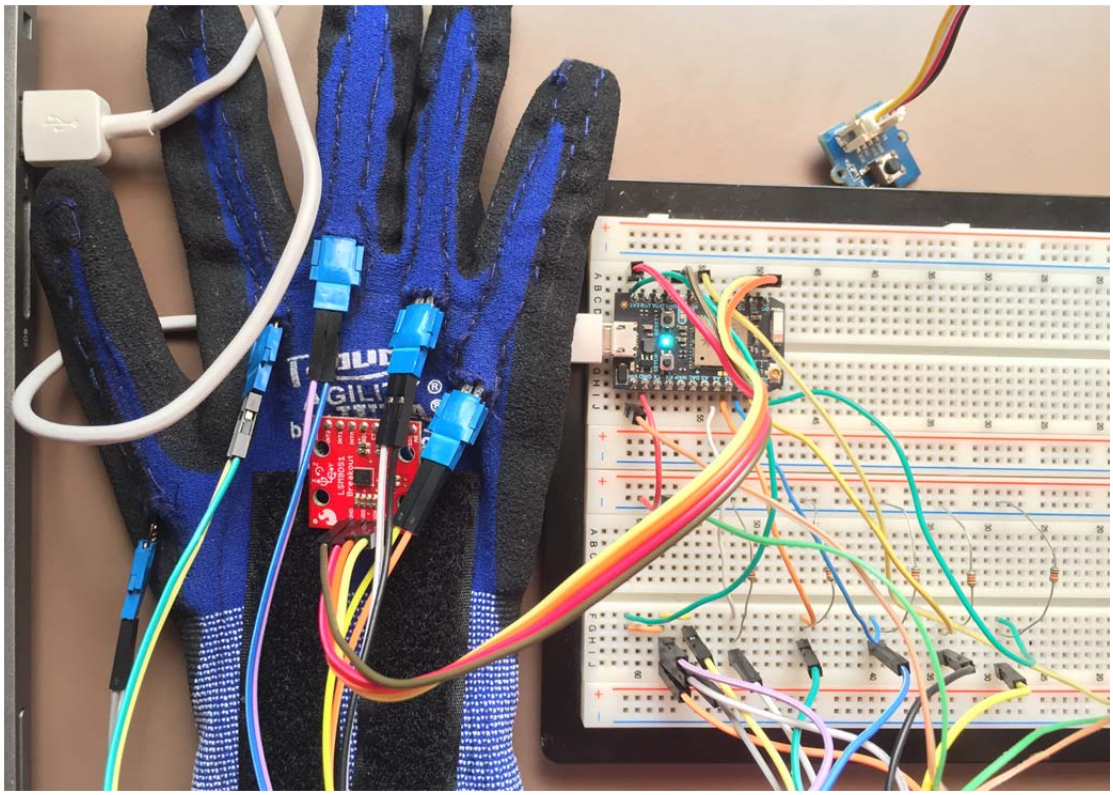


Figura 27. Plataforma HW/SW inicial

Asimismo, en la plataforma inicial se carga el fichero `svm-train-predict-v01.ino` como una primera versión del código desarrollado en el dispositivo *Photon* para que la clasificación sea posible. Para ello se establecen dos opciones en el menú de usuario dentro de la función *setup():* una para realizar el entrenamiento (opción 1) y otra para ejecutar las funciones de predicción (opción 2), tal como se muestra en la Figura 28.

```
110   Serial.println("MENU:");
111   Serial.println("-------------------------------------------------");
112   Serial.println("1. Capture Hand Gesture Data for training SVM");
113   Serial.println("2. Predict Hand Gestures");
114   Serial.println("-------------------------------------------------");
115   Serial.print ("Enter the number corresponding to the menu option to perform: ");
```

Figura 28. Sección de la función *setup()* del menú de la plataforma inicial

Así, dentro de la función *loop()* del *firmware* se lee qué opción del menú es seleccionada por el usuario, de tal manera que la al escoger la opción 1 se activa la variable *enableSampling*, y al seleccionar la opción 2 se habilita la variable *enablePrediction*, tal como se muestra en la sección de código de la Figura 29.

```
void loop()
{
//  while (!Serial.available()) Particle.process(); // wait for serial port to connect.
  if ((Serial.available() > 0) && (!enableSampling)) {
     char inOption = Serial.read();

     Serial.println(inOption);

     if (inOption == '1') {
        enableSampling = true;
        n_values = 0;
        Serial.print ("Enter the letter/number for the hand gesture data: ");
        while (!(Serial.available() > 0)) Particle.process(); // wait for serial port
        char inID = Serial.read();
        Serial.println(inID);
        symbol = inID; // the symbol associated to the sampled mean values
     }
     else if (inOption == '2') {
        enablePrediction = true;
     }
     else {
        Serial.println("Not a valid option");
        Serial.println("");
        Serial.print ("Enter the number corresponding to the menu option to perform: ");
    }
  }
```

Figura 29. Sección de código de la función *loop()* de la plataforma inicial

En la implementación del código ha de tenerse en cuenta que la orientación de la mano viene dada por los parámetros *Roll*, *Pitch* y *Yaw,* tal como se muestra en la Figura 30.

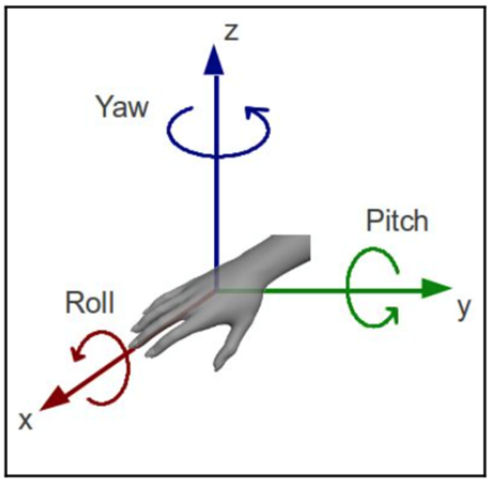


Figura 30. Orientación y movimiento de la mano

Para el problema tratado en el presente TFG, se utilizan únicamente los datos de *pitch* y *roll*, calculados a partir de los valores del acelerómetro integrado en el dispositivo LSM9DS1 para los ejes x/y/z con el fin de distinguir los signos con movimiento.

Además, debido a la necesidad de establecer un método para la lectura de los datos que requieren movimiento, se ha hecho uso de un pulsador asociado al pin digital *D6*,

y del LED azul situado junto al pin *D7* (Figura *31*), de tal manera que se leen los datos durante el tiempo que permanece el pin a nivel lógico *HIGH,* es decir, mientras el LED esté encendido.

```
41  #define LEDPIN D7
42  #define BUTTONPIN D6
```

Figura 31. Definición de las constantes asociadas a los pines digitales

Más concretamente, para cada letra a entrenar se realiza una medida de los datos proporcionados por los sensores a partir de la representación del gesto asociado a cada letra. Por cada gesto realizado se toman 50 muestras con un periodo de muestreo de 40 ms, es decir, durante un intervalo de dos segundos se tomarán muestras de los datos proporcionados por los sensores. Para controlar este intervalo de tiempo se utilizan *timers* implementados a partir de la librería externa *SparkIntervalTimer*, ya que aportan mayor precisión que la gestión de *timers* propia del dispositivo *Photon*. En la Figura 32 se muestra la sección de código donde se declara el *timer* denominado *sampleTimer,* así como las constantes: NUMBER_MEAN_VALUES para determinar el número de medidas que se toman por cada letra que se entrena, NUMBER_SAMPLES que establece el número de muestras que se recogen en cada medida, y SAMPLE_PERIOD para especificar el periodo de muestreo.

```
11  #include <SparkIntervalTimer.h>
12  IntervalTimer sampleTimer;
13
14  #define NUMBER_MEAN_VALUES 1 // number of mean values per button press
15  #define NUMBER_SAMPLES 50    // number of samples per mean value
16  #define SAMPLE_PERIOD  40    // sample period (ms)
```

Figura 32. Declaración de un *timer* y definición de las constantes para la toma de muestras

De igual forma, se declara un segundo *timer* denominado *HandMovDetectionTimer* para iniciar la detección del símbolo interpretado únicamente cuando la mano esté quieta. Para ello, se toman 25 muestras con un intervalo de muestreo de 50 ms, por lo que el *timer* está activo durante 1.25 segundos. En la Figura 33 se muestran los parámetros asociados a este segundo *timer.*

```
81  IntervalTimer HandMovDetectionTimer;
82  #define SAMPLE_PERIOD_HANDMOV   50
83  #define NUMBER_SAMPLES_HANDMOV 25
```

Figura 33. Declaración de un *timer* y definición de las constantes para detectar la mano quieta

En la Figura 34 se recoge el diagrama de flujo asociado a la primera opción de la plataforma inicial. Como se muestra en el diagrama, una vez seleccionada la letra a entrenar, el mecanismo para tomar las muestras durante el entrenamiento es el siguiente: se pulsa el pulsador identificado como Botón 1 para iniciar la recogida de muestras durante dos segundos a través de la función *sampleWrite().* Una vez pasado ese tiempo, se realizan los cálculos necesarios a través de la función *sampleValue()* para generar el modelo de predicción.

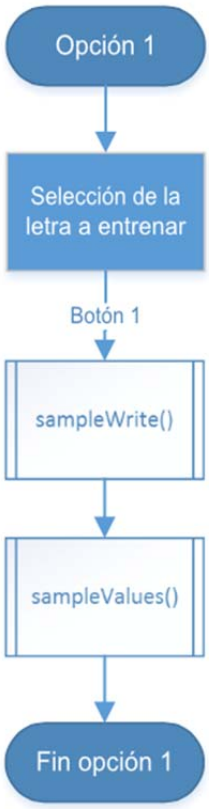


Figura 34. Opción 1 de la plataforma inicial

En la Figura 35 se recoge la sección de código de la función *loop()* una vez la variable *enableSampling* se ha puesto a *true*. Como puede verse en las líneas 171-176, al pulsar el botón 1 *(BUTTONPIN)* se activa la variable *enableCaptureSampling*, momento en el que se activará el *timer sampleTimer*, implementado a partir del uso de la librería

SparkIntervalTimer, que a su vez llamará a la función *sampleWrite()* cada vez que transcurra el intervalo correspondiente al periodo de muestreo de 40ms para, a partir de los sensores, tomar 50 muestras. Esto queda definido en la línea 182. A continuación, se llama a la función *sampleValues(),* y una vez que se ha realizado la medida, se desactiva el *timer sampleTimer* y se pide que se especifique una de las dos posibilidades del menú de usuario inicial (líneas 198-206).

```
166   if (enableSampling)
167   {
168     // Checks if BUTTONPIN is pressed to take NUMBER_SAMPLES samples
169     // (one sample each SAMPLE_PERIOD ms) in order to generate
170     // NUMBER_MEAN_VALUES mean values
171     bool buttonPressed = digitalRead(BUTTONPIN);
172     if(buttonPressed && !buttonAlreadyPressed)
173     {
174       enableCaptureSampling = true;
175     }
176     buttonAlreadyPressed = buttonPressed;
177
178     if (enableCaptureSampling)
179     {
180       if (!isSampling)
181       {
182         sampleTimer.begin(sampleWrite, SAMPLE_PERIOD, hmSec);
183         isSampling = true;
184       }
185       else
186       {
187         if (available == NUMBER_SAMPLES)
188         {
189           // print the symbol
190           Serial.print(symbol);
191           Serial.print(" ");
192
193           sampleValues();
194
195           available = 0;
196           n_values++;
197
198           if (n_values == NUMBER_MEAN_VALUES)
199           {
200             // stop recording
201             sampleTimer.end();
202             isSampling = false;
203             enableCaptureSampling = false;
204             enableSampling = false;
205             Serial.println("");
206             Serial.print ("Enter the number corresponding to the menu option to perform: ");
207           }
208         }
209       }
210     }
211   }
```

Figura 35. Función *loop()* – Sección de código correspondiente a la opción 1 de la plataforma inicial

En la Figura 36 se recoge el código asociado a la función *sampleWirte().* Por un lado (líneas 312-339), se recogen los valores leídos por los pines analógicos conectados a los sensores flexibles para, siguiendo el procedimiento visto en el apartado 2.3.2, estimar el ángulo de doblez de cada una de ellos. Por otro lado, se recogen los valores asociados al acelerómetro del dispositivo LSM9DS1 (líneas 344-347) para calcular el *pitch* en la línea 349 y el *roll* en la línea 350. Seguidamente en las líneas

352 y 353 se convierten de radianes a grados los valores de *pitch* y *roll*, respectivamente. Por último, en las líneas 355-390 se recogen en distintas matrices todos los datos registrados durante el proceso de muestreo, es decir, los ángulos de doblez de cada uno de los cinco sensores flexibles asociados a cada uno de los dedos de la mano, las medidas de *pitch* y *roll*, y los valores máximos y mínimos de los tres ejes del acelerómetro x/y/z.

```
311 void sampleWrite(void) {
312     // Read the ADC, and calculate voltage and resistance from it
313     int flexADC0 = analogRead(FLEX_PIN0);
314     float flexV0 = flexADC0 * VCC / 4095.0;
315     float flexR0 = R_DIV*(VCC/flexV0-1.0);
316 
317     int flexADC1 = analogRead(FLEX_PIN1);
318     float flexV1 = flexADC1 * VCC / 4095.0;
319     float flexR1 = R_DIV*(VCC/flexV1-1.0);
320 
321     int flexADC2 = analogRead(FLEX_PIN2);
322     float flexV2 = flexADC2 * VCC / 4095.0;
323     float flexR2 = R_DIV*(VCC/flexV2-1.0);
324 
325     int flexADC3 = analogRead(FLEX_PIN3);
326     float flexV3 = flexADC3 * VCC / 4095.0;
327     float flexR3 = R_DIV*(VCC/flexV3-1.0);
328 
329     int flexADC4 = analogRead(FLEX_PIN4);
330     float flexV4 = flexADC4 * VCC / 4095.0;
331     float flexR4 = R_DIV*(VCC/flexV4-1.0);
332 
333     // Use the calculated resistance to estimate the sensor's bend angle:
334     float value[NUMBER_SENSORS];
335     value[0] = map(flexR0, STRAIGHT_RESISTANCE0, BEND_RESISTANCE0,0.0, 90.0);
336     value[1] = map(flexR1, STRAIGHT_RESISTANCE1, BEND_RESISTANCE1,0.0, 90.0);
337     value[2] = map(flexR2, STRAIGHT_RESISTANCE2, BEND_RESISTANCE2,0.0, 90.0);
338     value[3] = map(flexR3, STRAIGHT_RESISTANCE3, BEND_RESISTANCE3,0.0, 90.0);
339     value[4] = map(flexR4, STRAIGHT_RESISTANCE4, BEND_RESISTANCE4,0.0, 90.0);
340 
341     float valueacc[NUMBER_ACC];
342     int16_t a[NUMBER_MMACC];
343 
344     imu.readAccel();
345     a[0] = imu.ax;
346     a[1] = imu.ay;
347     a[2] = imu.az;
348 
349     valueacc[0] = atan2(a[1], a[2]);
350     valueacc[1] = atan2(-a[0], sqrt(a[1] * a[1] + a[2] * a[2]));
351     // Convert from radians to degrees:
352     valueacc[0] *= 180.0 / M_PI;
353     valueacc[1]  *= 180.0 / M_PI;
354 
355     if (isSampling)
356     {
```

```
357     {
358       if (available == NUMBER_SAMPLES)
359         return;
360
361       for (int k = 0; k < NUMBER_SENSORS; k++)
362       {
363         mBuffer[k][mWritePos] = value[k];
364       }
365
366       for (int k = 0; k < NUMBER_ACC; k++)
367       {
368         maccBuffer[k][mWritePos] = valueacc[k];
369       }
370
371       for (int k = 0; k < NUMBER_MMACC; k++)
372         {
373             if (available == 0)
374             {
375               min_value_acc[k] = a[k];
376               max_value_acc[k] = a[k];
377             }
378             else
379             {
380               if (a[k] < min_value_acc[k])
381                 min_value_acc[k] = a[k];
382               if (a[k] > max_value_acc[k])
383                 max_value_acc[k] = a[k];
384             }
385          }
386
387       mWritePos++;
388
389       if (mWritePos == NUMBER_SAMPLES)
390         mWritePos = 0;
391
392       available++;
393     }
394 }
```

Figura 36. Función *sampleWrite()* de la plataforma inicial

El código asociado a la función *sampleValue()* se muestra en la Figura 37, y tiene tres partes diferenciadas: en la primera parte (líneas 265-279) se calcula el valor medio para cada sensor flexible, en la segunda parte (líneas 281-295) se calcula el valor medio para las medidas de *pitch* y *roll* del acelerómetro, y en la tercera parte (líneas 297-307) se calculan la diferencia entre los valores máximos y mínimos de cada uno de los tres ejes del acelerómetro durante el proceso de muestreo.

```
264 void sampleValues(void) {
265     // calculate the mean value for each sensor
266     float sum_value[NUMBER_SENSORS];
267     for (int j = 0; j < NUMBER_SENSORS; j++)
268     {
269        sum_value[j] = 0;
270        for (int i = 0; i < NUMBER_SAMPLES; i++)
271        {
272           sum_value[j] = (sum_value[j] + mBuffer[j][i]);
273        }
274        mean_value[j] = sum_value[j] / (float) NUMBER_SAMPLES;
275        Serial.print(j+1);
276        Serial.print(":");
277        Serial.print((int) mean_value[j]);
278        Serial.print(" ");
279     }
280
281     // calculate the mean value for each acc
282     float sum_value_acc[NUMBER_ACC];
283     for (int j = 0; j < NUMBER_ACC; j++)
284     {
285        sum_value_acc[j] = 0;
286        for (int i = 0; i < NUMBER_SAMPLES; i++)
287        {
288           sum_value_acc[j] = (sum_value_acc[j] + maccBuffer[j][i]);
289        }
290        mean_value_acc[j] = sum_value_acc[j] / (float) NUMBER_SAMPLES;
291        Serial.print(j+NUMBER_SENSORS+1);
292        Serial.print(":");
293        Serial.print((int) (mean_value_acc[j]));
294        Serial.print(" ");
295     }
296
297     // calculate the max-min value for each acc
298     for (int j = 0; j < NUMBER_MMACC; j++)
299     {
300         maxmin_value_acc[j] = (max_value_acc[j] - min_value_acc[j]);
301         Serial.print(j+NUMBER_SENSORS+NUMBER_ACC+1);
302         Serial.print(":");
303         Serial.print((int) (maxmin_value_acc[j] / 10));
304         Serial.print(" ");
305     }
306     Serial.println("");
307 }
```

Figura 37. Función *sampleValues()* de la plataforma inicial

En la Figura 38 pueden verse las constantes usadas en esta sección de código. Así, la constante NUMBER_SENSORS determina el número de sensores flexibles, mientras que NUMBER_ACC define el número de medidas tomadas directamente del acelerómetro y NUMBER_MACC el número de diferencias máximas calculadas a partir de las medidas del acelerómetro.

```
18 #define NUMBER_SENSORS 5  // number of sensors = VEC_DIM
19 #define NUMBER_ACC 2 // number of measures from accelerometer (pitch, roll)
20 #define NUMBER_MMACC 3 // number of measures from accelerometer (max-min x, max-min y, max-min z)
```

Figura 38. Definición de las constantes asociadas a los sensores

Por otro lado, el diagrama de flujo asociado a la segunda opción de uso de la plataforma inicial se muestra en la Figura *39*. El mecanismo para la predicción de signos es similar al seguido en la opción 1, aunque con algunas variaciones: en primer lugar, se presiona el Botón 1 para iniciar la recogida de muestras, aunque en este caso, antes de recoger las muestras es necesario asegurar que el usuario está preparado para realiza el signo, por lo que se añade la función *sampleHandMov()* para detectar cuándo está la mano quieta. Para ello se utiliza en este caso el *timer* denominado *HandMovDetectionTimer*. Seguidamente se encenderá el led azul *(LEDPIN)* durante 1,25 segundos con el fin de mostrar el tiempo disponible para representar el símbolo que se quiere identificar. Una vez transcurrido ese tiempo, con los datos obtenidos se realizan los cálculos necesarios a través de la función *sampleValue()* vista anteriormente, y se envían a la función de predicción, la cual, mediante un proceso de votación, es capaz de identificar el número asociado al signo identificado (siendo 0 = a, 1 = b, 2 = c, etc.).

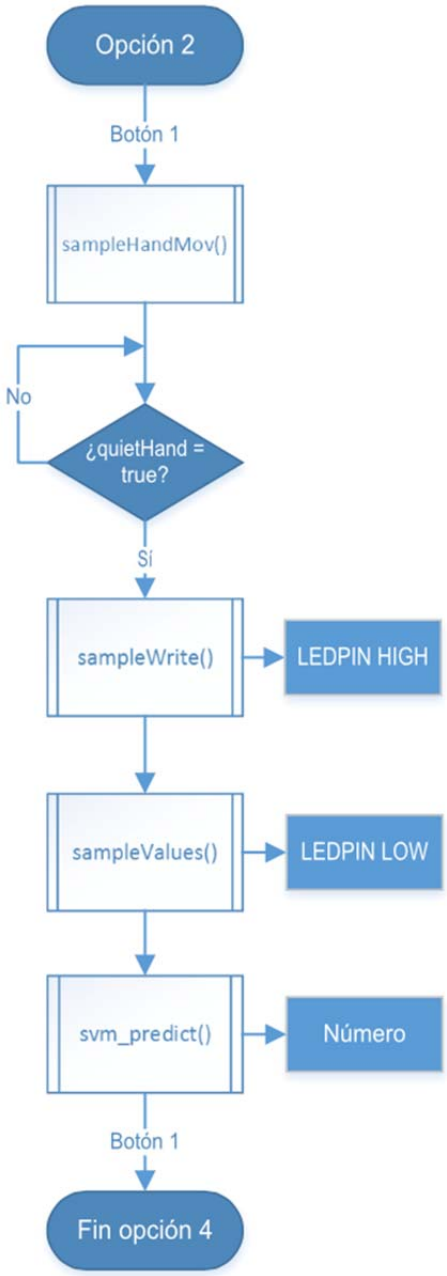


Figura 39. Opción 2 de la plataforma inicial

En la Figura 40 se recoge la sección de código de la función *loop()* una vez la variable *enablePrediction* se ha puesto a *true*. Como puede verse en las líneas 171-176, al pulsar el Botón 1 *(BUTTONPIN)* se activa la variable *enablePredictGesture,* momento en el que se activará el *timer HandMovDetectionTimer*, implementado a partir del uso de la librería *SparkIntervalTimer*, para, a partir de la función *sampleHandMov(),* detectar cuándo está la mano quieta. Esto queda definido en la línea 228. Así, una vez que se detecta la mano quieta, se vuelve a activar el *timer SampleTimer* para tomar 50 muestras, con un intervalo de muestreo de 40ms, de los datos proporcionados por los

sensores, mediante llamadas a la función *sampleWrite()*. Esto queda definido en la línea 237. A continuación, se llama a la función *sampleValues()* para calcular el valor medio de cada sensor flexible, las medidas de *pitch* y *roll* del acelerómetro, y la diferencia máxima de cada uno de los tres ejes del acelerómetro x/y/z. Todos estos datos se recogen en la variable *sensors* para a continuación ser pasados a la función *svm_predict(),* como puede verse en la línea 253. Atendiendo al orden asignado a los datos en la variable *sensors*, los valores de 1, 2, 3, 4 y 5 se corresponden con los datos medidos en cada una de las resistencias flexibles (siguiendo la relación vista anteriormente en la Tabla 6), 6 y 7 son las medidas de *pitch* y *roll*, y 8, 9 y 10 los valores de la máxima diferencia de cada uno de los tres ejes del acelerómetro x, y, z.

```
212   else if (enablePrediction)
213   {
214     bool buttonPressed = digitalRead(BUTTONPIN);
215     if(buttonPressed && !buttonAlreadyPressed)
216     {
217       if (enablePredictGesture)
218       {
219         enablePredictGesture = false;
220         enablePrediction = false;
221         HandMovDetectionTimer.end();
222         Serial.println("");
223         Serial.print ("Enter the number corresponding to the menu option to perform: ");
224       }
225       else
226       {
227         enablePredictGesture = true;
228         HandMovDetectionTimer.begin(sampleHandMov, SAMPLE_PERIOD_HANDMOV, hmSec);
229       }
230     }
231     buttonAlreadyPressed = buttonPressed;
232
233     if (enablePredictGesture)
234     {
235       if (quietHand)
236       {
237         sampleTimer.begin(sampleWrite, SAMPLE_PERIOD, hmSec);
238         digitalWrite(LEDPIN, HIGH);
239         isSampling = true;
240         quietHand = false;
241       }
242       else
243       {
244         if (available == NUMBER_SAMPLES)
245         {
246           digitalWrite(LEDPIN, LOW);
247           sampleValues();
248
249           int sensors[NUMBER_SENSORS+NUMBER_ACC+NUMBER_MMACC]= {(int) mean_value[0], (int) mean_value[1],
250           (int) mean_value[2], (int) mean_value[3],  (int) mean_value[4], (int) mean_value_acc[0], (int)
251           mean_value_acc[1], (int) (maxmin_value_acc[0] / 10), (int) (maxmin_value_acc[1] / 10), (int)
252           (maxmin_value_acc[2] / 10)};
253           svm_predict(sensors);
254
255           available = 0;
256           // stop recording
257           sampleTimer.end();
258           isSampling = false;
259         }
260       }
261     }
262   }
263 }
```

Figura 40. Función *loop()* – Sección de código correspondiente a la opción 2 de la plataforma inicial

La función *svm_predict(),* vista anteriormente en la Figura 23, requiere de los vectores de soporte generados durante el proceso de entrenamiento. Como ya se ha comentado, inicialmente los datos obtenidos de los sensores en el proceso de entrenamiento se escalan de manera externa al dispositivo *Photon*. Para ello, se sigue el procedimiento mostrado en la Figura 41, donde, partiendo de los datos obtenidos de los sensores a través de la ejecución de las funciones asociadas a la opción 1, se utiliza el comando *svm-scale* de la librería LIBSVM en la ventana de comandos para generar el fichero que contenga los datos de los sensores ya escalados, y otro fichero denominado *range* que contiene los valores mínimos y máximos de cada uno de los sensores. A partir del fichero con los datos escalados, se genera el modelo que permitirá predecir la clase de una nueva muestra mediante el comando *svm-train* de LIBSVM. Por último, se necesitan los ficheros *range* y *model* para, utilizando la herramienta *Arduino_SVM* de Java, generar un sketch `.ino` con los vectores de soporte.

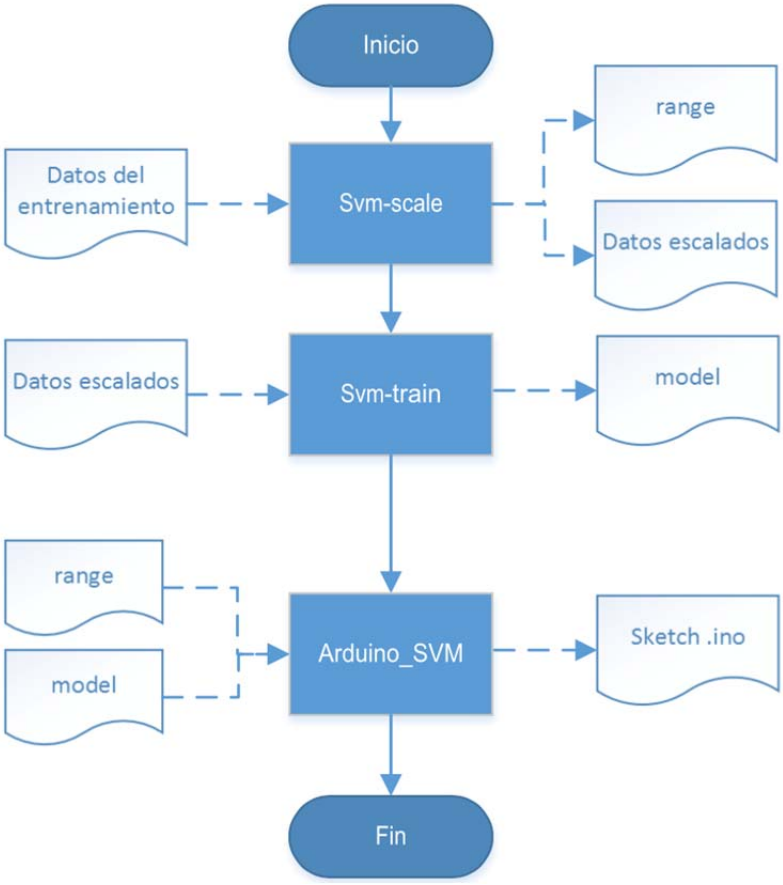


Figura 41. Diagrama de flujo escalado independiente

A continuación se expone un ejemplo de referencia con los pasos que es necesario seguir inicialmente para la generación del modelo de predicción:

1. Se carga el *firmware* desarrollado en el dispositivo *Photon*. Para ello, habrá que conectar el dispositivo a un PC, momento en el que el LED RGB empezará a parpadear en verde hasta conectarse a la red WiFi, que pasará a parpadear más lentamente en color celeste. A continuación, desde el *IDE Particle Build* en la web de Particle, se selecciona el dispositivo *Photon* asociado a la cuenta de usuario y se carga el fichero `svm-train-predict_v01.ino` desarrollado utilizando el botón *Flash*, tal como se detalla en el apartado 2.3.1. En ese momento el LED RGB comenzará a parpadear rápidamente en magenta.
2. Se abre el terminal para la comunicación serie USB con el PC. En el presente TFG se ha utilizado la aplicación *PuTTy* como terminal serie. Una vez abierta la aplicación, se selecciona el puerto COM correspondiente, y entre tanto, el LED RGB habrá pasado a parpadear en color verde. Al conectarse a la red WiFi, y una vez seleccionado el puerto COM correspondiente, aparecerá el menú diseñado, tal como se muestra en la Figura 42.

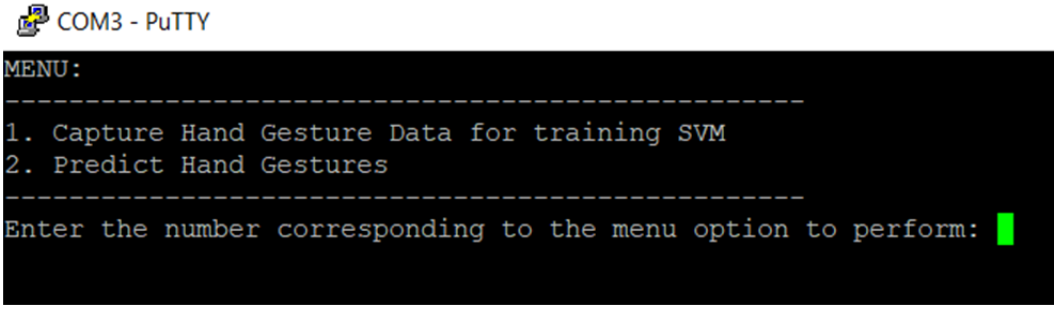


Figura 42. Menú plataforma inicial en el terminal

3. Se selecciona la opción 1 para realizar el entrenamiento. En la Figura 43 puede verse que se toma una medida para cada letra, donde los valores de 1, 2, 3, 4 y 5 se corresponden con los datos medidos en cada una de las resistencias flexibles, 6 y 7 son las medidas de *pitch* y *roll*, y 8, 9 y 10 los valores de la diferencia máxima en cada uno de los tres ejes x, y, z. Este proceso se repite seis veces para contar con mayor variedad de datos a la hora de generar los vectores de soporte.

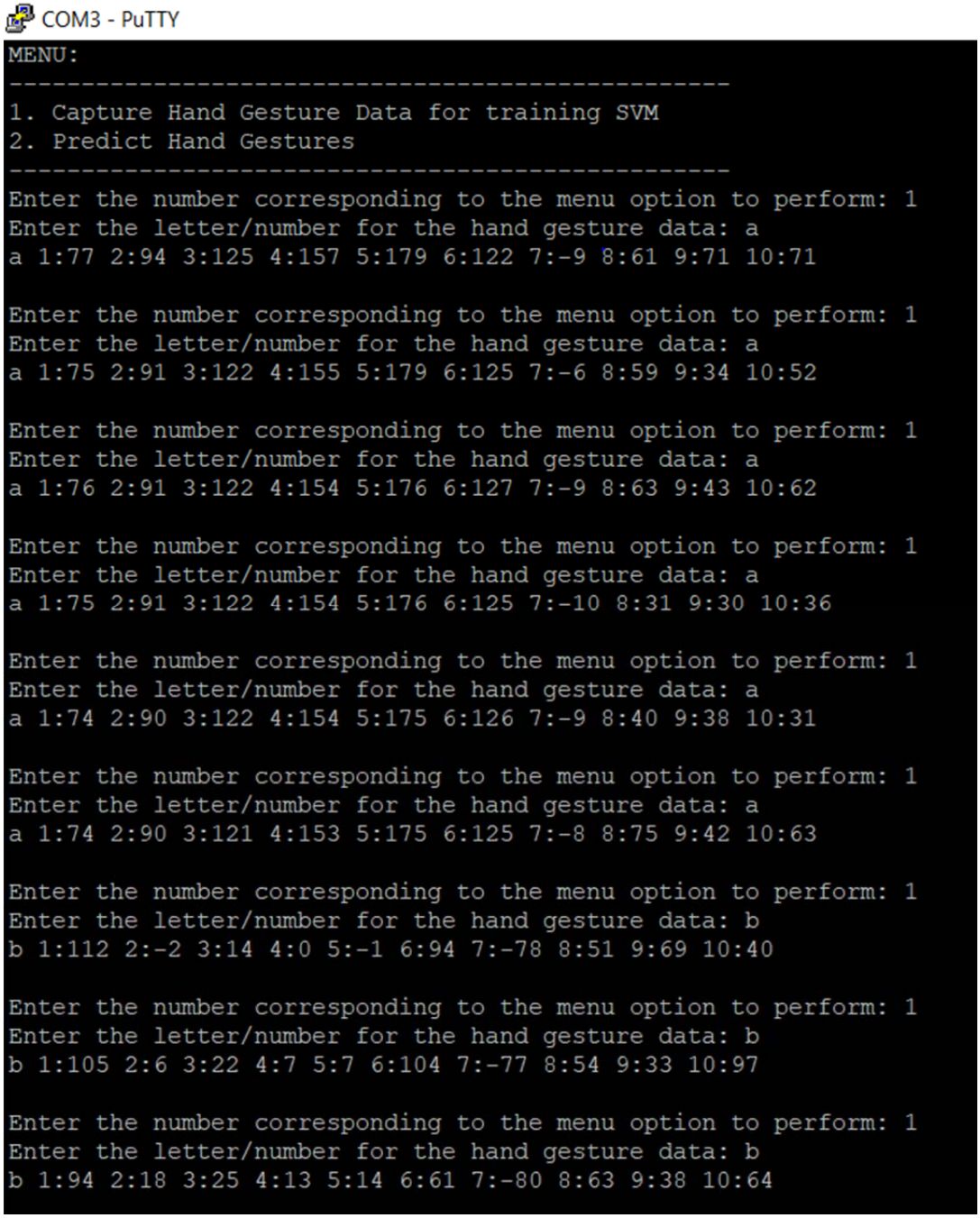
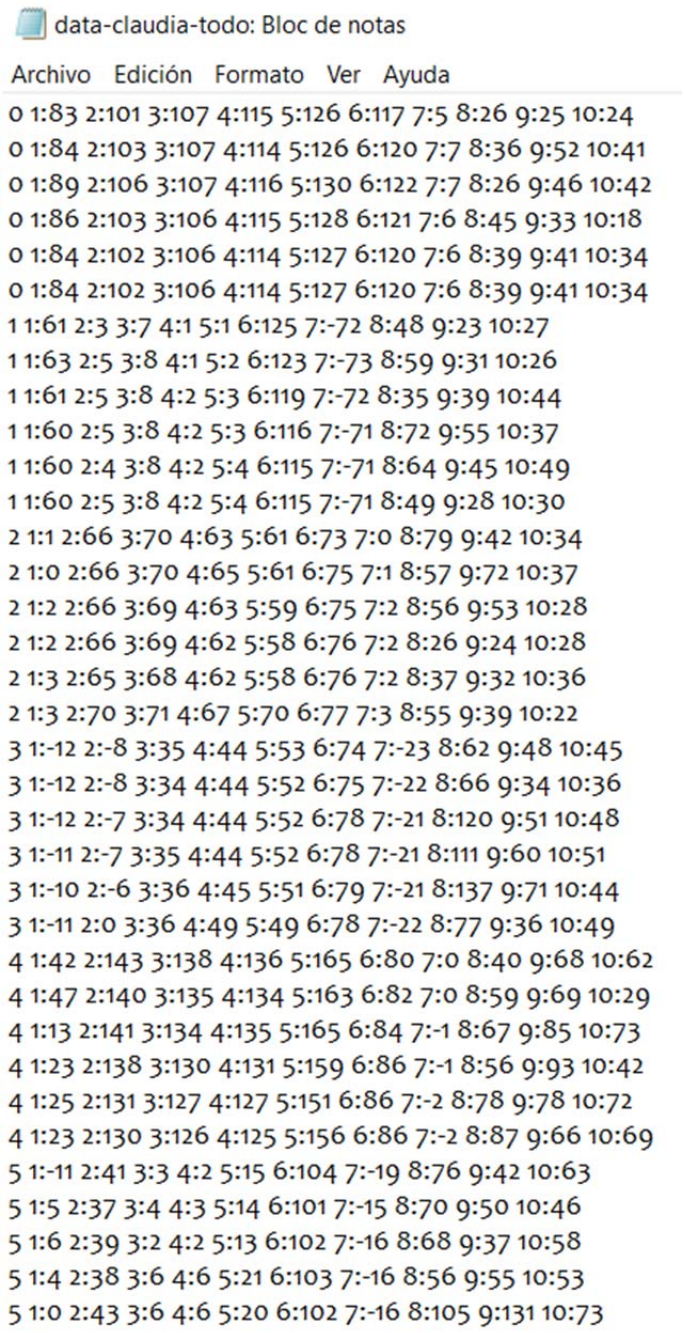


Figura 43.Datos obtenidos en el terminal con la opción “1” de la plataforma inicial

Siguiendo este proceso, se realiza el entrenamiento para todas las letras y a partir de los datos obtenidos, se genera manualmente un fichero como el mostrado en la Figura 44, incluyendo un identificador numérico asociado a cada símbolo.


data-claudia-todo: Bloc de notas

Archivo Edición Formato Ver Ayuda

0 1:83 2:101 3:107 4:115 5:126 6:117 7:5 8:26 9:25 10:24
0 1:84 2:103 3:107 4:114 5:126 6:120 7:7 8:36 9:52 10:41
0 1:89 2:106 3:107 4:116 5:130 6:122 7:7 8:26 9:46 10:42
0 1:86 2:103 3:106 4:115 5:128 6:121 7:6 8:45 9:33 10:18
0 1:84 2:102 3:106 4:114 5:127 6:120 7:6 8:39 9:41 10:34
0 1:84 2:102 3:106 4:114 5:127 6:120 7:6 8:39 9:41 10:34
1 1:61 2:3 3:7 4:1 5:1 6:125 7:-72 8:48 9:23 10:27
1 1:63 2:5 3:8 4:1 5:2 6:123 7:-73 8:59 9:31 10:26
1 1:61 2:5 3:8 4:2 5:3 6:119 7:-72 8:35 9:39 10:44
1 1:60 2:5 3:8 4:2 5:3 6:116 7:-71 8:72 9:55 10:37
1 1:60 2:4 3:8 4:2 5:4 6:115 7:-71 8:64 9:45 10:49
1 1:60 2:5 3:8 4:2 5:4 6:115 7:-71 8:49 9:28 10:30
2 1:1 2:66 3:70 4:63 5:61 6:73 7:0 8:79 9:42 10:34
2 1:0 2:66 3:70 4:65 5:61 6:75 7:1 8:57 9:72 10:37
2 1:2 2:66 3:69 4:63 5:59 6:75 7:2 8:56 9:53 10:28
2 1:2 2:66 3:69 4:62 5:58 6:76 7:2 8:26 9:24 10:28
2 1:3 2:65 3:68 4:62 5:58 6:76 7:2 8:37 9:32 10:36
2 1:3 2:70 3:71 4:67 5:70 6:77 7:3 8:55 9:39 10:22
3 1:-12 2:-8 3:35 4:44 5:53 6:74 7:-23 8:62 9:48 10:45
3 1:-12 2:-8 3:34 4:44 5:52 6:75 7:-22 8:66 9:34 10:36
3 1:-12 2:-7 3:34 4:44 5:52 6:78 7:-21 8:120 9:51 10:48
3 1:-11 2:-7 3:35 4:44 5:52 6:78 7:-21 8:111 9:60 10:51
3 1:-10 2:-6 3:36 4:45 5:51 6:79 7:-21 8:137 9:71 10:44
3 1:-11 2:0 3:36 4:49 5:49 6:78 7:-22 8:77 9:36 10:49
4 1:42 2:143 3:138 4:136 5:165 6:80 7:0 8:40 9:68 10:62
4 1:47 2:140 3:135 4:134 5:163 6:82 7:0 8:59 9:69 10:29
4 1:13 2:141 3:134 4:135 5:165 6:84 7:-1 8:67 9:85 10:73
4 1:23 2:138 3:130 4:131 5:159 6:86 7:-1 8:56 9:93 10:42
4 1:25 2:131 3:127 4:127 5:151 6:86 7:-2 8:78 9:78 10:72
4 1:23 2:130 3:126 4:125 5:156 6:86 7:-2 8:87 9:66 10:69
5 1:-11 2:41 3:3 4:2 5:15 6:104 7:-19 8:76 9:42 10:63
5 1:5 2:37 3:4 4:3 5:14 6:101 7:-15 8:70 9:50 10:46
5 1:6 2:39 3:2 4:2 5:13 6:102 7:-16 8:68 9:37 10:58
5 1:4 2:38 3:6 4:6 5:21 6:103 7:-16 8:56 9:55 10:53
5 1:0 2:43 3:6 4:6 5:20 6:102 7:-16 8:105 9:131 10:73

Figura 44. Ejemplo de fichero de datos entrenados en la plataforma inicial

En la Tabla 7 se recoge la relación entre los identificadores numéricos y las letras entrenadas inicialmente.

Tabla 7. Relación de identificadores numéricos y letras

Número	Letra	Número	Letra	Número	Letra	Número	Letra
0	A	8	I	16	O	24	V
1	B	9	J	17	P	25	W
2	C	10	K	18	Q	26	X
3	D	11	L	19	R	27	Y
4	E	12	LL	20	RR	28	Z
5	F	13	M	21	S		
6	G	14	N	22	T		
7	H	15	Ñ	23	U		

4. Se abre la ventana de comandos para utilizar los comandos de LIBSVM comentados. Por un lado, el comando *svm-scale*, indicando los parámetros de escalado y el nombre del nuevo fichero, tal como se muestra en la Figura 45.

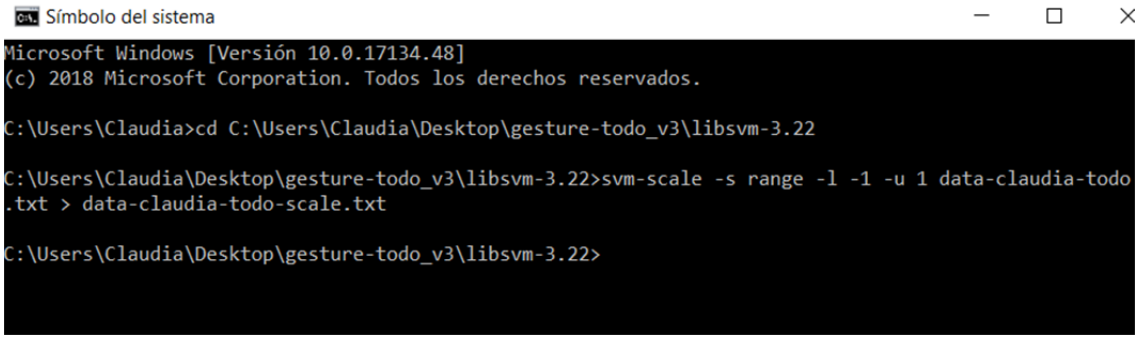
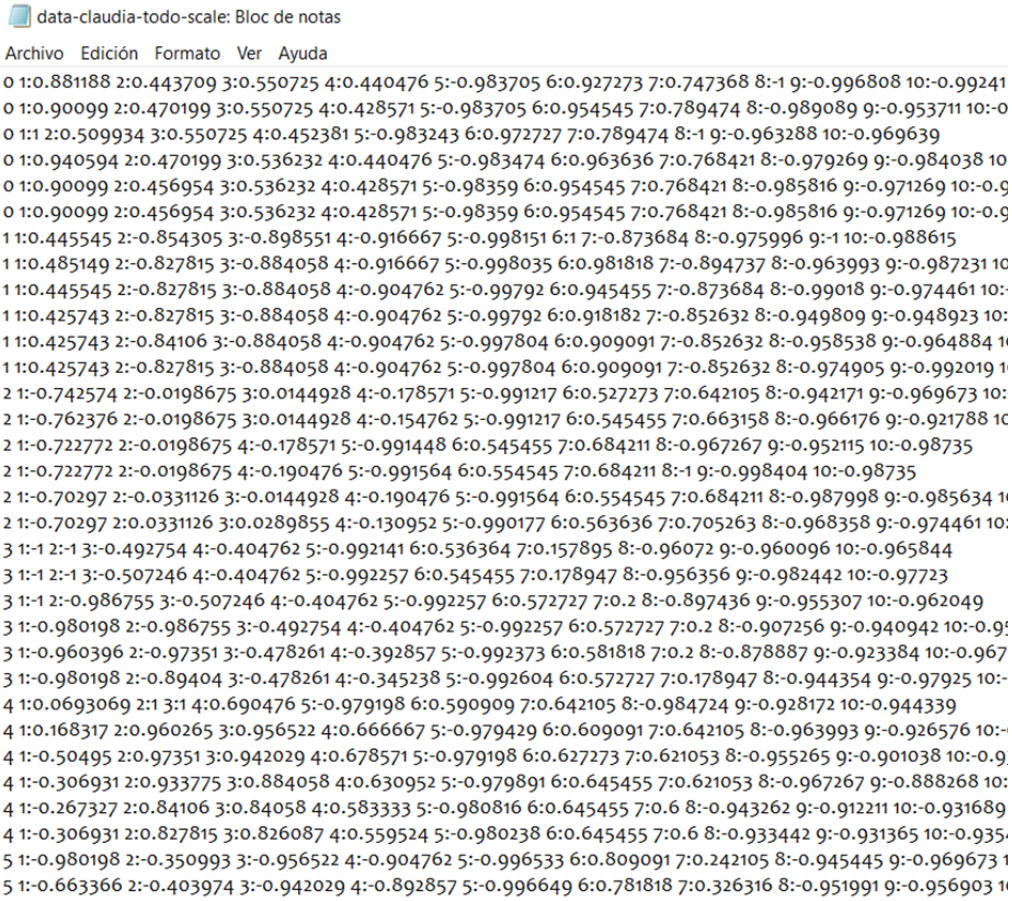


Figura 45. Utilización del comando svm-scale de LIBSVM

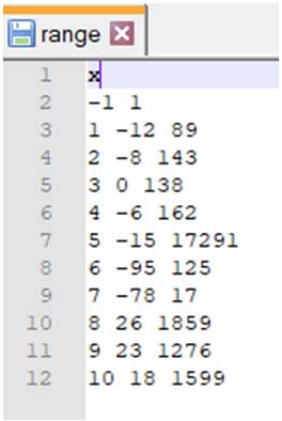
De esta manera se genera el fichero `data-claudia-todo-scale.txt` mostrado en la Figura 46 y el fichero *range* de la Figura 47.


data-claudia-todo-scale: Bloc de notas

Archivo Edición Formato Ver Ayuda

0 1:0.881188 2:0.443709 3:0.550725 4:0.440476 5:-0.983705 6:0.927273 7:0.747368 8:-1 9:-0.996808 10:-0.99241
0 1:0.90099 2:0.470199 3:0.550725 4:0.428571 5:-0.983705 6:0.954545 7:0.789474 8:-0.989089 9:-0.953711 10:-0
0 1:1 2:0.509934 3:0.550725 4:0.452381 5:-0.983243 6:0.972727 7:0.789474 8:-1 9:-0.963288 10:-0.969639
0 1:0.940594 2:0.470199 3:0.536232 4:0.440476 5:-0.983474 6:0.963636 7:0.768421 8:-0.979269 9:-0.984038 10
0 1:0.90099 2:0.456954 3:0.536232 4:0.428571 5:-0.98359 6:0.954545 7:0.768421 8:-0.985816 9:-0.971269 10:-0.9
0 1:0.90099 2:0.456954 3:0.536232 4:0.428571 5:-0.98359 6:0.954545 7:0.768421 8:-0.985816 9:-0.971269 10:-0.9
1 1:0.445545 2:-0.854305 3:-0.898551 4:-0.916667 5:-0.998151 6:1 7:-0.873684 8:-0.975996 9:-1 10:-0.988615
1 1:0.485149 2:-0.827815 3:-0.884058 4:-0.916667 5:-0.998035 6:0.981818 7:-0.894737 8:-0.963993 9:-0.987231 10
1 1:0.445545 2:-0.827815 3:-0.884058 4:-0.904762 5:-0.99792 6:0.945455 7:-0.873684 8:-0.99018 9:-0.974461 10:-
1 1:0.425743 2:-0.827815 3:-0.884058 4:-0.904762 5:-0.99792 6:0.918182 7:-0.852632 8:-0.949809 9:-0.948923 10:
1 1:0.425743 2:-0.84106 3:-0.884058 4:-0.904762 5:-0.997804 6:0.909091 7:-0.852632 8:-0.958538 9:-0.964884 1
1 1:0.425743 2:-0.827815 3:-0.884058 4:-0.904762 5:-0.997804 6:0.909091 7:-0.852632 8:-0.974905 9:-0.992019 1
2 1:-0.742574 2:-0.0198675 3:0.0144928 4:-0.178571 5:-0.991217 6:0.527273 7:0.642105 8:-0.942171 9:-0.969673 10:
2 1:-0.762376 2:-0.0198675 3:0.0144928 4:-0.154762 5:-0.991217 6:0.545455 7:0.663158 8:-0.966176 9:-0.921788 1
2 1:-0.722772 2:-0.0198675 4:-0.178571 5:-0.991448 6:0.545455 7:0.684211 8:-0.967267 9:-0.952115 10:-0.98735
2 1:-0.722772 2:-0.0198675 4:-0.190476 5:-0.991564 6:0.554545 7:0.684211 8:-1 9:-0.998404 10:-0.98735
2 1:-0.70297 2:-0.0331126 3:-0.0144928 4:-0.190476 5:-0.991564 6:0.554545 7:0.684211 8:-0.987998 9:-0.985634 1
2 1:-0.70297 2:0.0331126 3:0.0289855 4:-0.130952 5:-0.990177 6:0.563636 7:0.705263 8:-0.968358 9:-0.974461 10:
3 1:-1 2:-1 3:-0.492754 4:-0.404762 5:-0.992141 6:0.536364 7:0.157895 8:-0.96072 9:-0.960096 10:-0.965844
3 1:-1 2:-1 3:-0.507246 4:-0.404762 5:-0.992257 6:0.545455 7:0.178947 8:-0.956356 9:-0.982442 10:-0.97723
3 1:-1 2:-0.986755 3:-0.507246 4:-0.404762 5:-0.992257 6:0.572727 7:0.2 8:-0.897436 9:-0.955307 10:-0.962049
3 1:-0.980198 2:-0.986755 3:-0.492754 4:-0.404762 5:-0.992257 6:0.572727 7:0.2 8:-0.907256 9:-0.940942 10:-0.9
3 1:-0.960396 2:-0.97351 3:-0.478261 4:-0.392857 5:-0.992373 6:0.581818 7:0.2 8:-0.878887 9:-0.923384 10:-0.967
3 1:-0.980198 2:-0.89404 3:-0.478261 4:-0.345238 5:-0.992604 6:0.572727 7:0.178947 8:-0.944354 9:-0.97925 10:-
4 1:0.0693069 2:1 3:1 4:0.690476 5:-0.979198 6:0.590909 7:0.642105 8:-0.984724 9:-0.928172 10:-0.944339
4 1:0.168317 2:0.960265 3:0.956522 4:0.666667 5:-0.979429 6:0.609091 7:0.642105 8:-0.963993 9:-0.926576 10:-
4 1:-0.50495 2:0.97351 3:0.942029 4:0.678571 5:-0.979198 6:0.627273 7:0.621053 8:-0.955265 9:-0.901038 10:-0.9
4 1:-0.306931 2:0.933775 3:0.884058 4:0.630952 5:-0.979891 6:0.645455 7:0.621053 8:-0.967267 9:-0.888268 10:
4 1:-0.267327 2:0.84106 3:0.84058 4:0.583333 5:-0.980816 6:0.645455 7:0.6 8:-0.943262 9:-0.912211 10:-0.931689
4 1:-0.306931 2:0.827815 3:0.826087 4:0.559524 5:-0.980238 6:0.645455 7:0.6 8:-0.933442 9:-0.931365 10:-0.935
5 1:-0.980198 2:-0.350993 3:-0.956522 4:-0.904762 5:-0.996533 6:0.809091 7:0.242105 8:-0.945445 9:-0.969673 1
5 1:-0.663366 2:-0.403974 3:-0.942029 4:-0.892857 5:-0.996649 6:0.781818 7:0.326316 8:-0.951991 9:-0.956903 1

Figura 46. Ejemplo de fichero con los datos escalados


range

```
x
-1 1
1 -12 89
2 -8 143
3 0 138
4 -6 162
5 -15 17291
6 -95 125
7 -78 17
8 26 1859
9 23 1276
10 18 1599
```

Figura 47. Ejemplo de fichero *range*

Por otro lado, el comando *svm-train,* utilizando los datos del fichero `data-claudia-todo-scale.txt` generado a partir del comando *svm-scale, e* indicando los valores de los parámetros *C* y *gamma*, tal como se muestra en la Figura 48. De esta manera se obtiene el fichero con extensión `.model.`

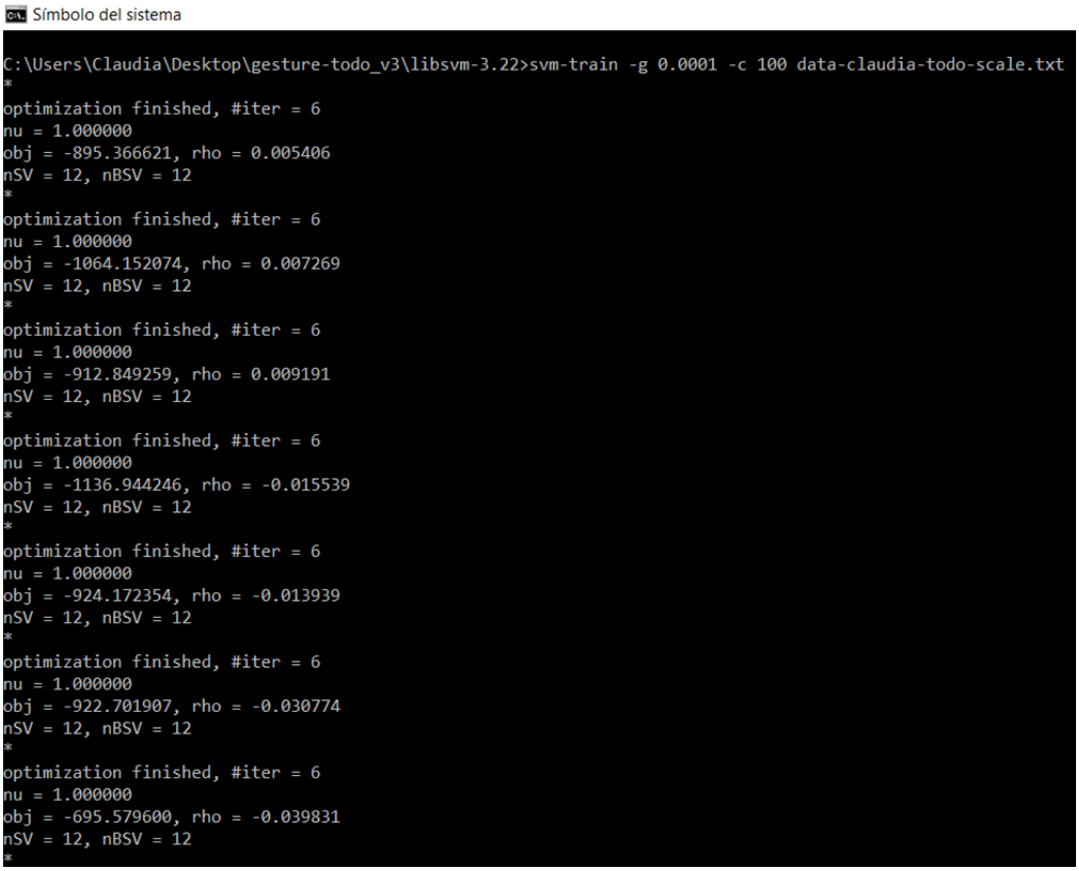


Figura 48. Utilización del comando *svm-train* de LIBSVM

5. Una vez hecho esto, se tienen los dos ficheros necesarios para, utilizando la herramienta *Arduino_SVM,* generar un *sketch* temporal `.ino` con los vectores soporte.

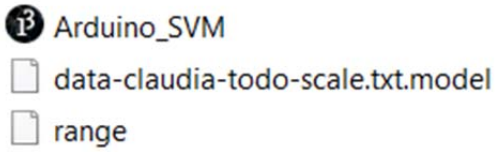


Figura 49. Ficheros necesarios para generar sketch.ino

Así, ejecutando la herramienta *Arduino_SVM*, mostrada en la Figura 49, aparece la pantalla de la Figura 50, y pulsando en el botón *play*, se abre una nueva pantalla en la que se deben especificar el nombre de los dos ficheros requeridos, y el nombre que se le quiera dar a la carpeta que se va a generar, como se muestra en la Figura 51.

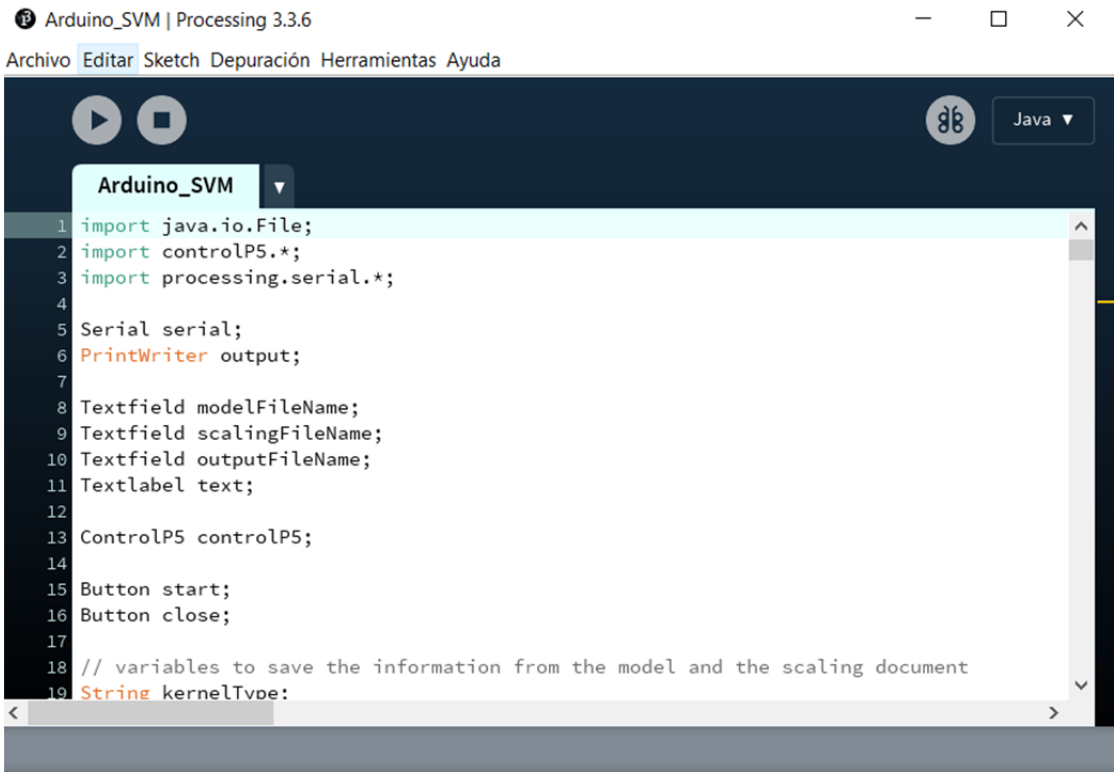


Figura 50. Herramienta *Arduino_SVM – Play*

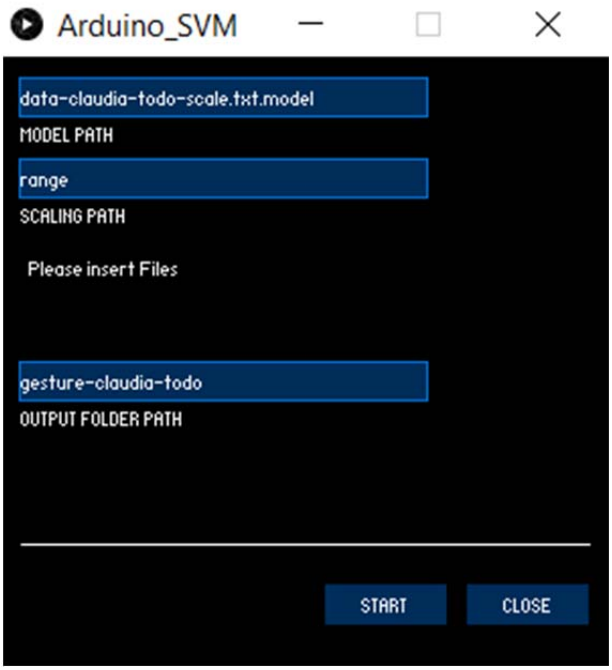


Figura 51. Herramienta Arduino_SVM

6. Se abre la nueva carpeta generada, que se representa en la Figura 52, y que contiene el *sketch* `.ino` requerido, como puede verse en la Figura 53.

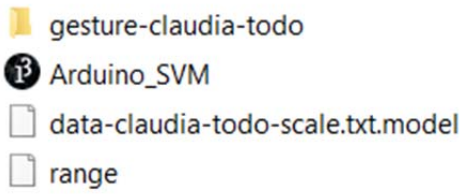


Figura 52. Carpeta generada a través de la herramienta *Arduino_SVM*

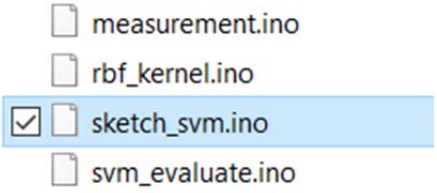


Figura 53. Contenido de la carpeta gesture-claudia-todo

El contenido del archivo `sketch_svm.ino` se ilustra en la Figura 54. En ella puede verse una sección del código con los vectores de soporte necesarios para sun integración en la función *svm_predict()* del código desarrollado para el dispositivo *Photon.*

sketch_svm.ino

```
#include <avr/pgmspace.h>
#define VEC_DIM 10

#define SVM_TYPE c_svc
#define KERNEL_TYPE rbf
#define GAMMA 0.0001
#define NR_CLASS 29
#define TOTAL_SV 174
const  PROGMEM float rho[] = {0.00540614, 0.00726879, 0.00919104, -0.0155389, -0.013
const int label[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
const int nr_sv[] = {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
 const PROGMEM float yalpha1[6 * (NR_CLASS-1)] = {100, 100, 100, 100, 100, 100, 100,

 const PROGMEM float sv1[6 * VEC_DIM] = {0.881188, 0.443709, 0.550725, 0.440476, -0.9

 const PROGMEM float yalpha2[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100,

 const PROGMEM float sv2[6 * VEC_DIM] = {0.445545, -0.854305, -0.898551, -0.916667, -

 const PROGMEM float yalpha3[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100,

 const PROGMEM float sv3[6 * VEC_DIM] = {-0.742574, -0.0198675, 0.0144928, -0.178571,

 const PROGMEM float yalpha4[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100,

 const PROGMEM float sv4[6 * VEC_DIM] = {-1, -1, -0.492754, -0.404762, -0.992141, 0.5

 const PROGMEM float yalpha5[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100,
```

Figura 54. Contenido del fichero *sketch_svm.ino*

7. El siguiente paso es copiar los vectores de soporte en el fichero `svm-train-predict-v01.ino`, que contiene el desarrollo *firmware* de la plataforma inicial sobre el dispositivo *Photon*. En la Figura 55 se muestra una sección del código asociado a esta parte, mientras que en la Figura 56 pueden verse las variables usadas en la función *svm_predict(),* con todos los vectores de soporte utilizados para este caso e referencia.

```
const  PROGMEM float rho[] = {0.00540614, 0.00726879, 0.00919104, -0.0155389, -0.0139385, -0.0307739,
const int label[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22
const int nr_sv[] = {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
 const PROGMEM float yalpha1[6 * (NR_CLASS-1)] = {100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 1

 const PROGMEM float sv1[6 * VEC_DIM] = {0.881188, 0.443709, 0.550725, 0.440476, -0.983705, 0.927273,

 const PROGMEM float yalpha2[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, 100, 100, 100,

 const PROGMEM float sv2[6 * VEC_DIM] = {0.445545, -0.854305, -0.898551, -0.916667, -0.998151, 1, -0.

 const PROGMEM float yalpha3[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, -100, -100, -100

 const PROGMEM float sv3[6 * VEC_DIM] = {-0.742574, -0.0198675, 0.0144928, -0.178571, -0.991217, 0.52

 const PROGMEM float yalpha4[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, -100, -100, -100

 const PROGMEM float sv4[6 * VEC_DIM] = {-1, -1, -0.492754, -0.404762, -0.992141, 0.536364, 0.157895,

 const PROGMEM float yalpha5[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, -100, -100, -100

 const PROGMEM float sv5[6 * VEC_DIM] = {0.0693069, 1, 1, 0.690476, -0.979198, 0.590909, 0.642105, -0

 const PROGMEM float yalpha6[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, -100, -100, -100

 const PROGMEM float sv6[6 * VEC_DIM] = {-0.980198, -0.350993, -0.956522, -0.904762, -0.996533, 0.809

 const PROGMEM float yalpha7[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, -100, -100, -100

 const PROGMEM float sv7[6 * VEC_DIM] = {0.0891089, -0.788079, 0.594203, 0.654762, -0.984745, 0.454543

 const PROGMEM float yalpha8[6 * (NR_CLASS-1)] = {-100, -100, -100, -100, -100, -100, -100, -100, -100
```

Figura 55. Vectores de soporte en el código de la plataforma inicial

```
const int scalePar[] = {-1,1};
const int low[] = {-12,-8,0,-6,-15,-95,-78,26,23,18};
const int high[] = {89,143,138,162,17291,125,17,1859,1276,1599};
int result[NR_CLASS]={0};

// all support vectors and alpha values
float const* const supportVectors[NR_CLASS] PROGMEM = {
sv1, sv2, sv3, sv4, sv5, sv6, sv7, sv8, sv9, sv10, sv11, sv12, sv13, sv14,
sv15, sv16, sv17, sv18, sv19, sv20, sv21, sv22, sv23, sv24, sv25, sv26, sv27,
sv28, sv29, };
 const float* const valuesForSupport[NR_CLASS] PROGMEM = {
yalpha1, yalpha2, yalpha3, yalpha4, yalpha5, yalpha6, yalpha7, yalpha8, yalpha9,
yalpha10, yalpha11, yalpha12, yalpha13, yalpha14, yalpha15, yalpha16, yalpha17,
yalpha18, yalpha19, yalpha20, yalpha21, yalpha22, yalpha23, yalpha24, yalpha25,
yalpha26, yalpha27, yalpha28, yalpha29, };
// scaling function: in this function the new sensor data which are measured are
scaled with help of the data from the scaling file.
void scale(const int* sensor, float* scaledSensor){
  for(int p=0; p<VEC_DIM;p++){
    scaledSensor[p] = (float)scalePar[0] *((float)scalePar[1] -((float)scalePar[0]))
    *((float)sensor[p]-(float)low[p])/((float)high[p]-(float)low[p]);
  }
}
```

Figura 56. Variables asociadas a los vectores de soporte

Así, una vez cargado el código en el dispositivo *Photon*, ha sido posible realizar el entrenamiento para un conjunto de letras, tanto estáticas como con movimiento, y generar un modelo que permita identificar las distintas clases existentes.

En la Figura 57 se recoge un ejemplo de los datos obtenidos en el proceso de predicción al realizar el gesto correspondiente a los signos del alfabeto dactilológico español asociados a las vocales.

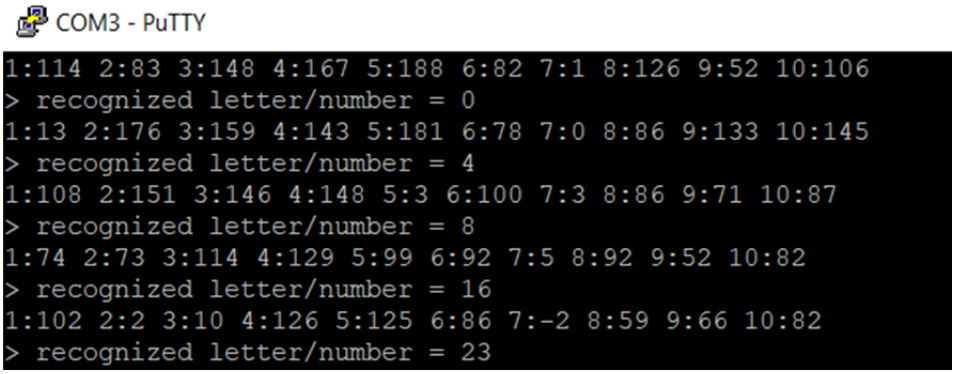


Figura 57. Identificación de las letras "A","E","I","O" y "U"

Finalmente, en la Figura 58 se muestran los resultados satisfactorios obtenidos al representar los pares de letras “L” y “LL”, y “U” y “V”, las cuales se distinguen entre sí únicamente por el movimiento de la mano.

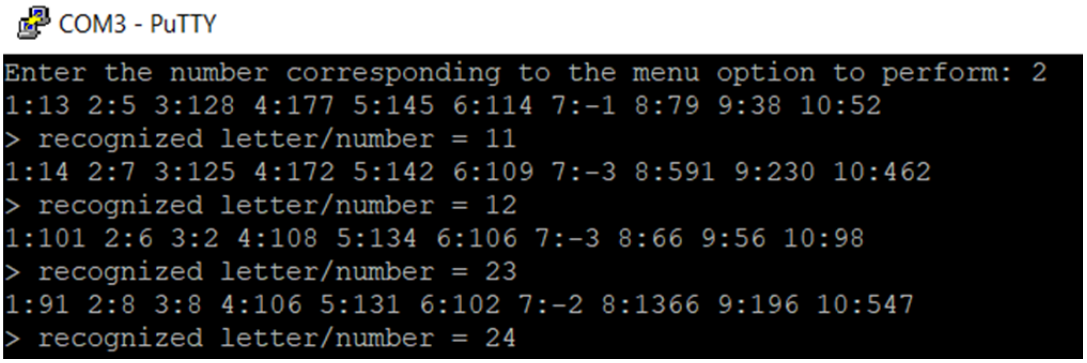


Figura 58. Identificación de letras con y sin movimiento

Sin embargo, a partir de la realización de diferentes pruebas experimentales se determinó que el código implementado en esta plataforma inicial no permitía distinguir en todos los casos entre algunas letras, como es el caso de las letras “I” e “Y”, pues no se detecta adecuadamente el movimiento del dedo meñique, requerido para interpretar y distinguir de manera unívoca la letra “Y”. En cualquier caso,

disponiendo en este punto de una versión inicial de la plataforma HW/SW prácticamente funcional, en la que es posible realizar la correcta identificación de la mayor parte de los 30 símbolos del alfabeto dactilológico de la LSE, se decidió continuar el desarrollo del presente TFG integrando la conectividad BLE para poder transmitir los símbolos reconocidos a un dispositivo móvil, antes de optimizar el proceso de reconocimiento con el fin de obtener una versión final de la plataforma HW/SW.

Capítulo 4. Protocolo BLE

En este capítulo se realiza una descripción del protocolo de comunicaciones BLE, con el fin de comprender su funcionamiento y principales características. Además, se recoge la parte de código implementada en el presente TFG relativa al uso de este protocolo, cuyo propósito es permitir la transmisión del signo reconocido en cada momento, a un dispositivo móvil. Por último, se detalla la validación funcional del *firmware* implementado, en este caso para el dispositivo *RedBear Duo*.

4.1 Aspectos básicos de BLE

Bluetooth Low Energy (BLE) [30] [31] se corresponde con la especificación 4.0, 4.1 y 4.2 de la tecnología Bluetooth, desarrollada por Bluetooth *Special Interest Group (SIG)* en el año 2010. Planteado para ser considerado un estándar con objetivos y aplicaciones complementarias a *Bluetooth Classic*, a pesar de utilizar la misma banda de frecuencia, entre otras similitudes, BLE es una tecnología orientada a garantizar un bajo consumo de energía, menor tiempo de establecimiento de conexión, y reducida complejidad.

BLE está diseñado para la transmisión de pequeñas cantidades de datos (con tiempos de transmisión muy pequeños) a muy corto alcance y, por lo tanto, con un reducido consumo de potencia. No está planteado para mantener una conexión entre dispositivos por un largo periodo de tiempo transmitiendo grandes cantidades de datos a alta velocidad, como en el caso de *Bluetooth Classic.* Esto permite que los dispositivos BLE estén activos sólo cuando se les solicita la transmisión de datos. La potencia de transmisión (generalmente medida en dBm) suele ser configurable en un cierto rango (generalmente entre -30 y 0 dBm), pero cuanto mayor es la potencia de transmisión (mejor rango de alcance), mayor es la demanda de alimentación, reduciendo la vida útil de las celdas de batería. Es posible crear y configurar un dispositivo BLE que pueda transmitir datos de manera confiable a 30 metros, o más, de línea de visión, si bien un rango operativo típico comprende de 2 a 5 metros.

La topología de red en BLE es de tipo estrella. Los dispositivos *Master* pueden establecer múltiples conexiones con dispositivos *Slave*, y generalmente realizan búsquedas de otros dispositivos. Por otro lado, un dispositivo *Slave*, sólo puede tener una conexión con un único dispositivo *Master*. Además, un dispositivo puede enviar datos en modo *Broadcasting*. En consecuencia, un dispositivo BLE puede comunicarse básicamente de acuerdo a dos modos de operación:

- ***Broadcasting*:** Se pueden enviar datos a cualquier dispositivo receptor en el rango de escucha. Como se ilustra en la Figura 59, este mecanismo permite enviar datos en un solo sentido a cualquier dispositivo capaz de recibir los datos transmitidos. En esta topología se definen dos roles distintos: Broadcaster

(envía paquetes de Advertising no conectables periódicamente a cualquier dispositivo BLE) y Observer (escanea repetidamente las frecuencias pre-configuradas para recibir cualquier paquete de Advertising no conectable que se esté emitiendo).

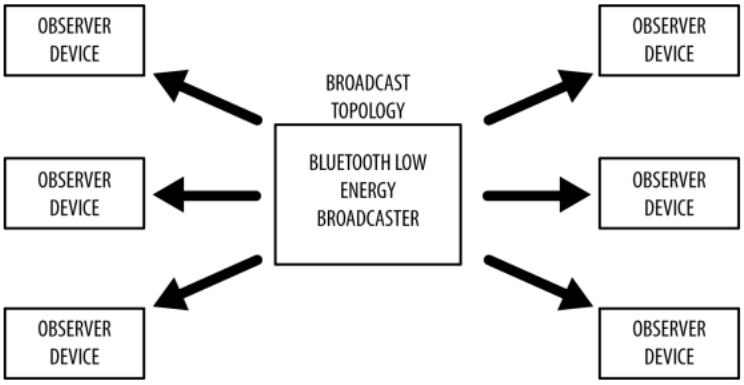


Figura 59. Topología Broadcasting

- ***Connections*:** Se transmiten datos en ambas direcciones. Una conexión es un intercambio permanente y periódico de paquetes de datos entre dos dispositivos. Por lo tanto, es intrínsecamente privado (los datos solo se envían y reciben entre los dos dispositivos implicados en una conexión, y ningún otro dispositivo). Las conexiones involucran dos roles diferentes:
    - *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 que se encuentra en una conexión activa, el dispositivo *Peripheral* sigue la sincronización del dispositivo *Central* e intercambia datos regularmente con él.

  Para iniciar una conexión, un dispositivo *Central* recibe en primer lugar los paquetes de *Advertising* conectables, y a continuación, envía una solicitud al dispositivo *Peripheral* para establecer una conexión exclusiva entre ambos. Una vez que se establece la conexión, el dispositivo *Peripheral* deja de enviar

paquetes de *Advertising* y los dos dispositivos pueden comenzar a intercambiar paquetes de datos en ambas direcciones, como se muestra esquemáticamente en la Figura 60.

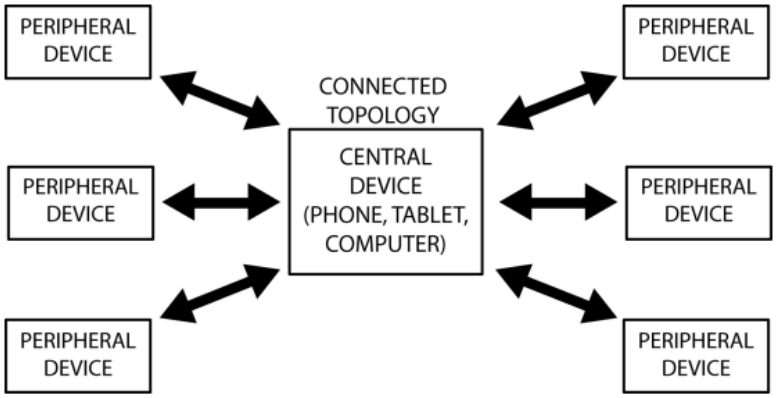


Figura 60. Topología Connections

4.2 Perfiles

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. En cambio, los Perfiles definen cómo deben usarse los Protocolos para lograr un objetivo particular, ya sea genérico o específico.

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), o *Health Thermometer Profile* (transfiere lecturas de temperatura corporal sobre BLE), entre otros.

4.3 Protocolos

Para la gestión de los dispositivos, la conexión y la interfaz de las aplicaciones, en el estándar BLE se define una pila de protocolos. Como se muestra en la Figura 61, la pila de protocolos BLE se divide en tres partes básicas: *Application*, *Host,* y *Controller*. Cada uno de estos bloques básicos de la pila de protocolos se divide en varias capas que proporcionan la funcionalidad necesaria para operar:

- *Application*: Es la capa más alta y la 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.
- *Host*: Es la pila de *software* que gestiona la comunicación entre dos o más dispositivos. Esta parte de la pila de protocolos contiene:
    - 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*.

- *Controller*: Se corresponde con el dispositivo físico que permite transmitir y recibir señales de radio e interpretarlas como paquetes de información. Esta parte de la pila de protocolos contiene:
  - Interfaz de Control de *Host* (HCI, *Host Controller Interface*), lado del Controlador.
  - Capa de Enlace (LL, *Link Layer*).
  - Capa Física (PHY, *PHYsical Layer*).

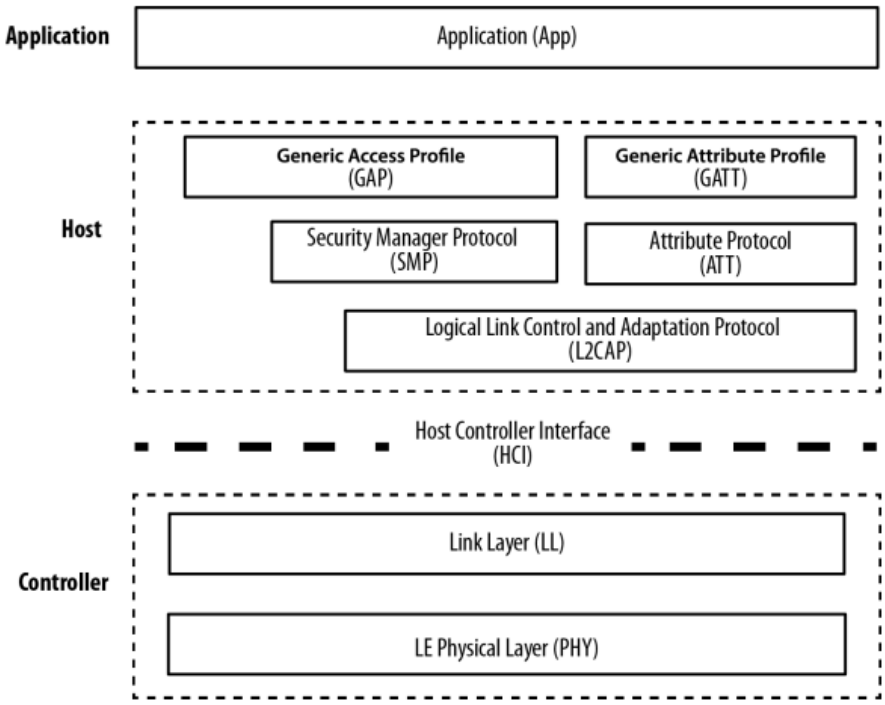


Figura 61. Pila de protocolos BLE

Capa Física (PHY)

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. Emplea la banda ISM (*Industrial, Scientific and Medical*) de 2.4 GHz para comunicarse, al igual que *Bluetooth Classic*, si bien BLE tiene 40 canales, y con una separación de canales diferente, desde 2.4000 GHz hasta

2.4835 GHz. Como se muestra en la Figura 62, 37 de estos canales se utilizan para datos de conexión, y los últimos tres canales (37, 38 y 39) se utilizan como canales de *Advertising* para establecer conexiones y enviar datos de *Broadcasting*. 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*…). *Bluetooth Classic* y BLE usan la modulación *GFSK* (*Gaussian Frequency Shift Keying*) a 1 Mbps, aunque con índices de modulación diferentes.

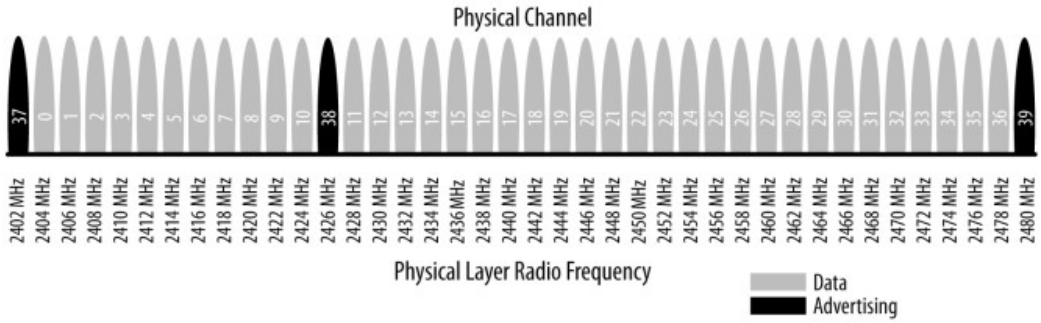


Figura 62. Canales de frecuencia BLE

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 siguiente fórmula:

$$\text{canal} = (\text{canal actual} + \text{salto}) \bmod 37$$

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.

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, la

Capa de Enlace 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. El funcionamiento de la Capa de Enlace puede describirse en términos de una máquina de estados (Figura 63) muy simple con los siguientes cinco estados:

- *Standby*: Este es el estado predeterminado de la Capa de Enlace. En este estado no se reciben ni se transmiten paquetes. Un dispositivo puede entrar en este estado desde cualquiera de los otros estados.

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

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

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

- *Connection:* El dispositivo Central está conectado a otro dispositivo *Peripheral*,

definiéndose los roles de *Master* y *Slave*. A este estado se puede pasar desde el estado *Initiating*, o desde el estado *Advertising*. Cuando se ingresa desde el estado *Initiating*, el dispositivo actúa como *Master*, mientras que cuando se entra desde el estado *Advertising*, el dispositivo actúa como *Slave* en el intercambio periódico de datos a través de eventos de conexión. Los canales de datos se utilizan una vez establecida la conexión entre ambos dispositivos.

Para el dispositivo *Slave*, el estado *Advertising* también se considera como estado inicial antes del estado *Connection*.

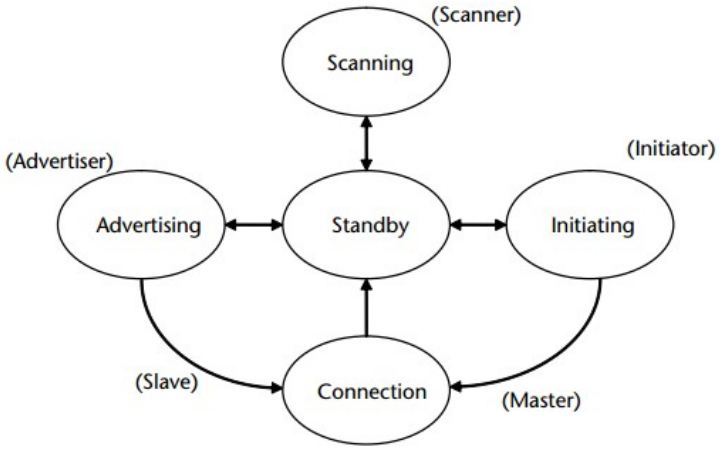


Figura 63. Máquina de estados de la Capa de Enlace de BLE

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.

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 apropiada 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).

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, el Cliente analiza el valor e interpreta 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.

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.

Perfil de Atributo Genérico (GATT)

El Perfil de Atributo Genérico (GATT, *Generic Attribute Profile*) puede considerarse la columna vertebral 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, si bien 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, como sus propiedades, el nombre visible para el usuario, y las unidades, entre otros).

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 *Rec. UIT-T X.216*|*X.667*, alternativamente conocida como *ISO/IEC 9834-8:2005* [].

BLE emplea estos UUIDs para, entre otras funciones, 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 los 27 bytes de la Capa de Enlace, la especificación BLE agrega dos formatos de UUID adicionales: UUID de 16 y de 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 entre sí, 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 (65534), 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 los 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 existen 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/o 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 opera 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 *64* muestra la jerarquía de Datos y Atributos introducida por GATT.

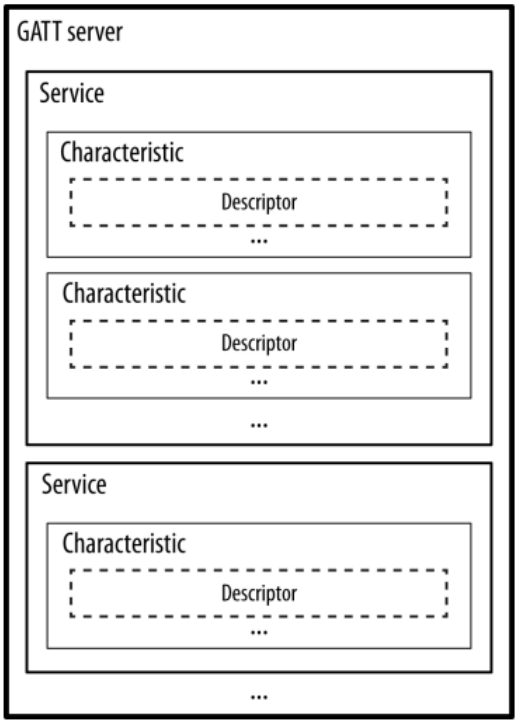


Figura 64. Jerarquía de Datos y Atributos de GATT

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 indica el comienzo de un Servicio, denominado *Declaración del Servicio*, como se muestra en la Figura *65*.

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

Figura 65. 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, 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 *66* 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 66. 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 *67* 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 67. 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 Carcaterística. En la Figura *68* 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 68. 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 determinada Característica con propiedad *Notify* o *Indicate*, 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 Descriptor 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.

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 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.

4.4 Conexión y transferencia de datos en el protocolo BLE

4.4.1 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.

4.4.2 Procedimiento de Advertising y procedimiento de 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. 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.

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 *69*.

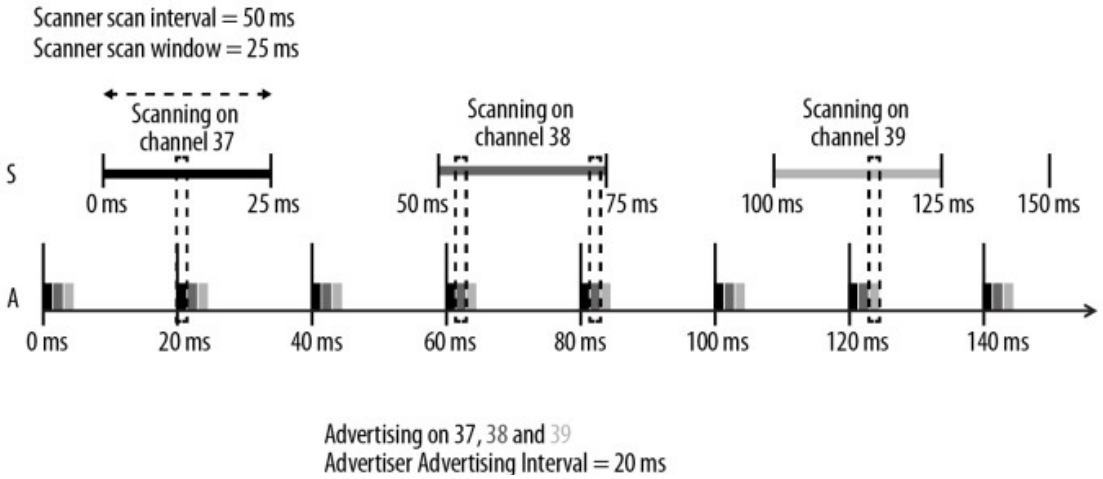


Figura 69. Procedimientos de Advertising y Scanning

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 *70*ilustra la diferencia entre ambos tipos de procedimiento de *Scanning.*

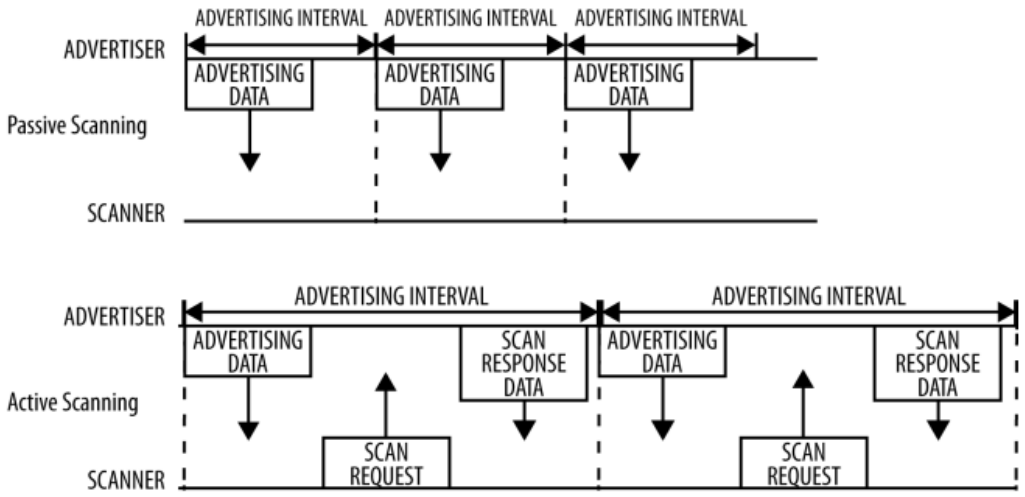


Figura 70. Passive Scanning y Active Scanning

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 Figura 71.

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

Figura 71. Tipos de paquetes de Advertising

- *ADV_IND* (*Connectable Undirected Advertising*): Es el más común y el más genérico. El dispositivo *Advertiser* puede ser encontrado por cualquier dispositivo *Scanner,* y este último puede iniciar la conexión.

- *ADV_DIRECT_IND* (*Connectable Directed Advertising*): Se emplea cuando un dispositivo *Advertiser* necesita conectarse de manera rápida con un dispositivo *Scanner* concreto. Estos paquetes de *Advertising* deben ser enviados, como máximo, cada 3.75 ms.

- *ADV_NONCONN_IND* (*Non-connectable Undirected Advertising*): Es empleado por dispositivos *Advertiser* que desean emitir datos en modo *Broadcast* sin conectarse a ningún dispositivo *Scanner*, ni tan siquiera recibir ninguna información.

- *ADV_SCAN_IND* (*Scannable Undirected Advertising*): Es similar al anterior, pero en este caso el dispositivo *Scanner* puede obtener datos del dispositivo *Advertiser*, ya que éste responde a cada paquete de tipo *Scan Request* detectado, con un paquete de tipo *Scan Response*.

4.4.3 Proceso de Conexión

Para establecer una conexión, un dispositivo *Scanner* comienza en primer lugar a buscar dispositivos *Advertiser* que acepten solicitudes de conexión. Cuando detecta un dispositivo *Advertiser* adecuado, el dispositivo *Scanner* envía un paquete de solicitud de conexión (*Connect Request*) al dispositivo *Advertiser* y, siempre que éste responda, establece una conexión. El paquete de solicitud de conexión incluye el incremento de salto de frecuencia, que determina la secuencia de salto que seguirán los dispositivos *Master* y *Slave* durante la vida útil de la conexión. Una conexión es simplemente una secuencia de intercambio de paquetes de datos entre los dispositivos *Slave* y *Master* en tiempos predefinidos, como se muestra en la Figura *72*, 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.

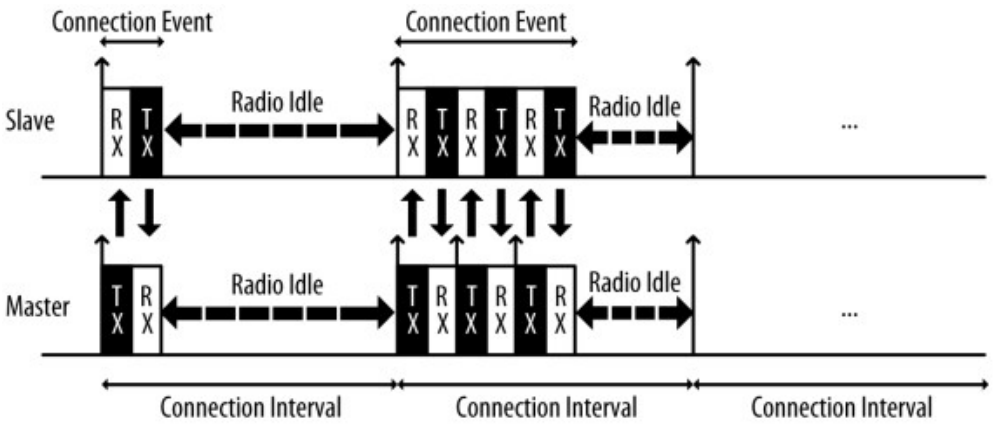


Figura 72. Eventos de conexión

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 en las versiones 4.0 y 4.1 del estándar BLE.

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.

4.4.4 Formato de los paquetes BLE

El formato de un paquete BLE se define como se muestra en la Figura *73*.

Figura 73. Formato de un paquete BLE

Como puede verse, la carga útil de datos es de 255 bytes, que representa la carga útil definida a partir de la especificación *Bluetooth* 4.2. Sin embargo, como se comentó anteriormente, los paquetes BLE tienen una carga útil de datos de 27 bytes para las especificaciones *Bluetooth* 4.0 y 4.1, que son las contempladas en el presente TFG.

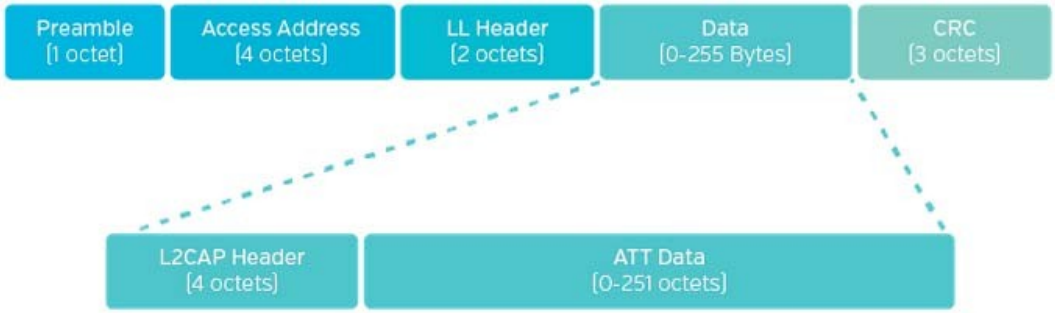


Figura 74. Campo de datos de un paquete BLE

Si se analiza el campo de datos, éste se ocupa con paquetes de la capa L2CAP, tal y como se muestra en la Figura *74*. 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.

En cuanto al formato de los paquetes de datos ATT, estos presentan la estructura mostrada en la Figura *75*.

Figura 75. Formato de los paquetes de datos ATT

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, tal y como se comentó con anterioridad.

Por otro lado, en la Figura *75* 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*. Para el desarrollo del presente TFG se contemplará un tamaño máximo de 23 bytes, que es el correspondiente a las especificaciones 4.0 y 4.1 del estándar BLE.

4.4.5 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 esencial que el Cliente empiece 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. Por lo general, esta opción se usa 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 (generalmente porque 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 UUIDs 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.

4.5 Dispositivo *RedBear Duo*

Con el fin de cubrir el objetivo de transmitir los símbolos detectados, se sustituye en la plataforma hardware/software el dispositivo *Photon* por el dispositivo *ReadBear Duo*, que además de WiFi, integra la tecnología de comunicación BLE.

RedBear Duo [11] es una placa de desarrollo diseñada para simplificar el proceso de creación de productos IoT. Integra un microcontrolador ARM Cortex M3 con el chip BCM43438 de la empresa Broadcom (actualmente *Cypress*), que combina conectividad WiFi y Bluetooth 4.0 (BLE). Comparten la misma antena de 2.4 GHz y pueden operar en modo dual.

Entre otras características, el dispositivo *RedBear Duo* dispone de 1MB de memoria Flash interna, 2MB de memoria *Flash* SPI externa y 128KB de SRAM, además de varios LED integrados, 18 entradas/salidas de propósito general, periféricos avanzados y un sistema operativo en tiempo real (*FreeRTOS*). Este dispositivo cuenta con una gran cantidad de interfaces analógicas, digitales y de comunicación, tal y como se detalla en la Tabla 8.

Tabla 8. Periféricos del dispositivo *RedBear Duo*

Tipo de Periférico	Cantidad	Entrada(E)/Salida(S)
Digital	18	E/S
Analógico (ADC)	8	E
Analógico (DAC)	2	S
SPI	2	E/S
I2S	1	E/S
I2C	1	E/S
CAN	1	E/S
USB	1	E/S
PWM	13	S

En la Figura 76 se detallan los principales componentes del dispositivo *RedBear Duo,* mientras que en la Figura 77 se especifica su *pin-out*.

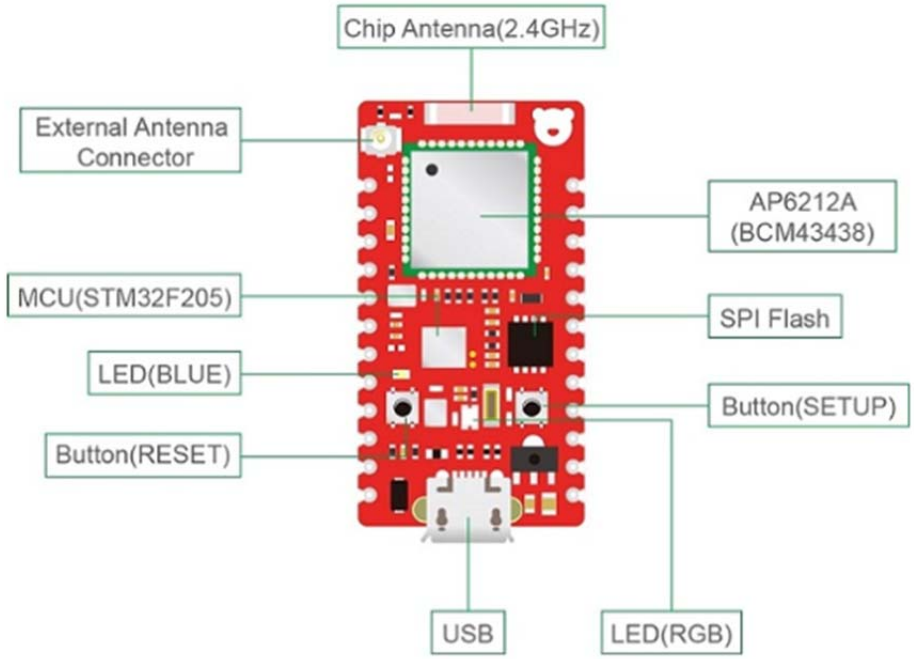


Figura 76. Componentes principales del dispositivo *RedBear Duo*

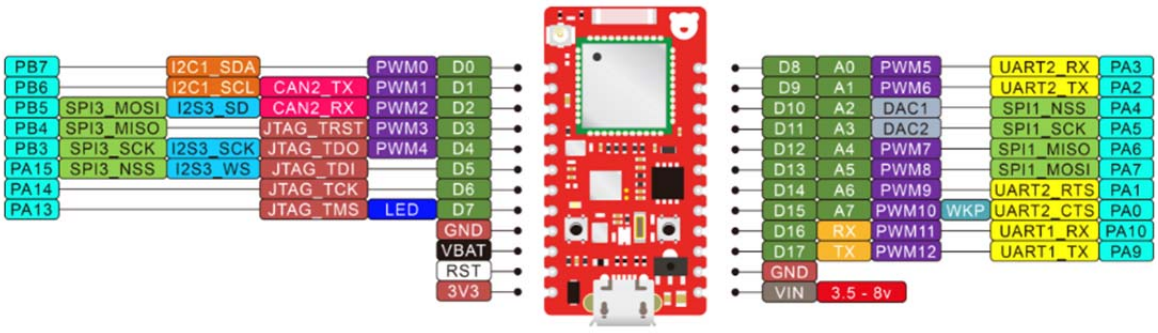


Figura 77. Pin-out del dispositivo *RedBear Duo*

Los dos botones disponibles en la plataforma (*Setup* y *Reset*) permiten configurar nuevas credenciales WiFi y reiniciar el dispositivo, entre otras funciones básicas. También se pueden utilizar en conjunto para provocar el restablecimiento de los valores de fábrica por defecto. Entre ambos botones se encuentra un LED de tipo RGB (*Red*, *Green*, *Blue*) que proporciona información acerca del estado del dispositivo. Por ejemplo, si se encuentra conectado a una red WiFi estará parpadeando suavemente en color celeste o, si se está cargando un programa en memoria *Fl*ash, este LED

parpadeará rápidamente en color magenta.

En la parte inferior se encuentra el puerto micro-USB, cuyo propósito principal es proporcionar alimentación al dispositivo *RedBear Duo*, aunque también se puede utilizar para la programación del dispositivo, así como para realizar comunicaciones serie USB con un ordenador. A la izquierda del puerto micro-USB se encuentran los pines de *reset* y alimentación (*3V3, RST, VBAT y GND*). El dispositivo *RedBear Duo* convierte la energía de entrada proporcionada a través de la alimentación del puerto micro-USB, o del pin *VIN*, en un suministro de 3.3V, ya que toda la lógica del dispositivo funciona con esta tensión. El pin *RST* se puede utilizar, al igual que el botón *Reset*, para reiniciar el sistema. El pin *VBAT* permite conectar una pequeña batería de reserva externa al dispositivo *RedBear Duo*, con el fin de retener el contenido de los registros RTC (*Real-Time Clock),* hacer una copia de seguridad de la memoria SRAM y suministrar el RTC cuando se deje de suministrar la tensión *VDD*. Es importante destacar que la batería establecida en el pin *VBAT* no puede alimentar al dispositivo *RedBear Duo*.

Los pines *D0* a *D7* son pines de propósito general que pueden actuar como entradas o salidas digitales. Además, los pines *D0* a *D4* también pueden actuar como salidas analógicas utilizando técnicas PWM (*Pulse-Width Modulation*). Por otro lado, tal y como se aprecia en la Figura 76, existe también un LED azul situado junto al pin *D7* que se encuentra conectado directamente a este pin.

Este dispositivo integra el módulo AMPAK A6212A, construido en torno al circuito integrado BCM43438 de Broadcom, que soporta WiFi 802.11b/g/n y BLE 4.0, lo que permite desarrollar aplicaciones que se comuniquen con otros dispositivos inalámbricos a través de BLE, WiFi, o BLE+WiFi al mismo tiempo. Además, dispone también de un pequeño conector UFL a través del cual se puede conectar una antena externa. Con esto se conseguiría ampliar el rango de cobertura WiFi, añadiendo una antena más sensible o direccional. Por defecto, el dispositivo *RedBear Duo* intentará elegir la mejor antena, aunque también se puede controlar qué antena utilizar mediante *firmware*.

Los pines *A0* a *A6* constituyen entradas analógicas que operan con tensiones comprendidas entre 0V y 3.3V. Los pines analógicos también se pueden utilizar como entradas o salidas digitales, al igual que los pines *D0* a *D7* y, del mismo modo que los pines digitales, algunos pines analógicos (*A0, A1, A4, A5* y *A6*) se pueden utilizar como salidas analógicas PWM. Asimismo, los pines *A2* y *A3* se pueden emplear como salidas de los conversores analógico-digitales (*Digital Analog Converter*, *DAC*) internos, los cuales constituyen dos pines analógicos de salida especiales capaces de proporcionar tensiones comprendidas entre 0V y 3.3V. A continuación del pin *A6* se encuentra el pin WKP, el cual se utiliza para despertar al dispositivo *RedBear Duo* después de que se haya configurado en modo *deep sleep*, aunque también se puede emplear como salida analógica (*A7*), como entrada o salida digital (*D15*), y como salida analógica PWM.

Los pines *TX* y *RX* (transmisión y recepción) se utilizan para la comunicación serie. Por último, situados por debajo de estos pines se encuentran, un segundo pin *GND*, y el pin *VIN*. Tal y como se comentó anteriormente, se puede alimentar el dispositivo *RedBear Duo* suministrando entre 3.5V y 8V al pin VIN, como alternativa al uso del puerto USB.

El dispositivo *RedBear Duo* soporta varios tipos de lenguaje de programación, como Arduino, C/C++, JavaScript y Python, y las aplicaciones se pueden desarrollar con GCC, el entorno de desarrollo WICED SDK, o con los IDE de Arduino, Particle y Espruino.

En cuanto al tipo de aplicaciones, el dispositivo *RedBear Duo* se puede utilizar para el desarrollo de aplicaciones en diferentes ámbitos, como: Automatización industrial, automatización de edificios, electrodomésticos inteligentes, juguetes inteligentes, sensores IoT, WiFi/BLE *Gateway* o *Beacon Management*.

Además, debido a que dispositivo *RedBear Duo* es compatible con la ejecución de aplicaciones de usuario basadas en el *firmware* de la empresa Particle, en el proceso de fabricación del dispositivo se instala el *firmware* personalizado de Particle de forma predeterminada. El la Figura 78 se muestra exactamente la asignación de memoria en el dispositivo *RedBear Duo*, a partir de la versión v0.3.0 del firmware de usuario.

La memoria Flash externa del dispositivo *RedBear Duo* almacena el firmware correspondiente a la conectividad WiFi, que se cargará durante el proceso de arranque

en el chip BCM43438 que integra, así como el firmware de recuperación. Por lo tanto, este diseño permite que todo el espacio de la memoria *Flash* interna de 256KB esté disponible para el *sketch* de usuario.

El gestor de arranque (*Bootloader*) está almacenado en el espacio de memoria *Flash* interna comprendido entre las direcciones 0x08000000 a 0x08007FFF, cubriendo un tamaño de 32KB. El gestor de arranque determina si ejecutar el *firmware* de Particle o las aplicaciones WICED. Si ninguno es válido, entra al modo DFU (*Device Firmware Update*) para cargar el firmware.

	Sector	Internal Flash Address	Size	Photon (for Particle)	Duo (for Particle/Arduino)	Duo (for WICED)
Internal Flash	0	0x08000000 - 0x08003FFF	16K	Bootloader	Bootloader (Modified)	Bootloader (Modified)
	1	0x08004000 - 0x08007FFF	16K	DCT1		
	2	0x08008000 - 0x0800BFFF	16K	DCT2	DCT1	DCT1
	3	0x0800C000 - 0x0800FFFF	16K	EEPROM 1	DCT2	DCT2
	4	0x08010000 - 0x0801FFFF	64K	EEPROM 2	Reserved	User-part
	5	0x08020000 - 0x0803FFFF	128K	System-part1	System-part1	
	6	0x08040000 - 0x0805FFFF	128K		System-part2	
	7	0x08060000 - 0x0807FFFF	128K	System-part2		
	8	0x08080000 - 0x0809FFFF	128K			
	9	0x080A0000 - 0x080BFFFF	128K	User-part		
	10	0x080C0000 - 0x080DFFFF	128K	OTA Image	User-part	
	11	0x080E0000 - 0x080FFFFF	128K	Factory Reset Image		
Exteral Serial Flash	0 - 183	0x00000000 - 0x000B7FFF	736K	No Ext. Flash	User Data	WICED
	184 - 187	0x000B8000 - 0x000BBFFF	16K		EEPROM 1	Reserved (Particle)
	188 - 191	0x000BC000 - 0x000BFFFF	16K		EEPROM 2	
	192 - 319	0x000C0000 - 0x0013FFFF	512K		OTA Images	
	320 - 383	0x00140000 - 0x0017FFFF	256K		Factory Reset Image	
	384 - 511	0x00180000 - 0x001FFFFF	512K		WiFi Firmware	

Figura 78. Mapa de memoria del dispositivo RedBear Duo

Además, si el dispositivo *RedBear Duo* ejecuta el *firmware* de Particle, el gestor de arranque es responsable del reinicio de la aplicación por defecto de fábrica, aplicando el firmware descargado OTA (*Over-The-Air*) o la aplicación de usuario, haciendo que el dispositivo *RedBear Duo* entre en modo *Safe*, borre las credenciales de la conexión WiFi, etc.

La DCT (*Device Configuration Table)* abarca desde la dirección 0x08008000 de la memoria *Flash* interna, hasta la 0x0800FFFF, comprendiendo un tamaño de 32KB.

Está separada en dos particiones (DCT1 y DCT2) de tamaño 16KB. Solo una de las DCT es válida a la vez, mientras que la otra está en espera para intercambiar datos. Se usan

alternativamente para mantener la integridad de los datos en caso de que se apague el dispositivo durante el cambio en la configuración.

La emulación EEPROM se usa para almacenar datos no volátiles del usuario, y comprende el rango de direcciones de la memoria Flash externa, desde la dirección 0x000B8000, hasta la 0x000BFFFF, abarcando un tamaño de 32KB. El principio de funcionamiento de la emulación EEPROM es el mismo que el de la DCT, estando separada en dos particiones de 16KB, y siendo una válida a la vez.

System-part 1, de tamaño 128KB, comprende desde la dirección de memoria *Flash* interna 0x08020000, hasta la 0x0803FFFF. Es parte del *firmware* del sistema de Particle, que implementa las funciones de comunicación y servicios en la nube. Estas funciones pueden invocarse mediante la parte del *firmware* del sistema System-part 2, y la aplicación del usuario, por lo que se denominan "dynalibs" (bibliotecas dinámicas).

System-part 2, de tamaño 512KB, abarca desde la dirección de memoria *Flash* interna 0x08040000, hasta la 0x080BFFFF. Es la parte principal del *firmware* del sistema, que implementa todas las funciones de la capa hardware, incluidos los periféricos internos, WiFi, BLE, etc. Pueden ser invocadas desde la aplicación del usuario en forma de dynalibs. Inicializa la plataforma y ejecuta el sistema operativo *FreeRTOS* en tiempo real. Es responsable de gestionar los eventos del sistema, actualizar el *firmware* y la programación OTA de la aplicación de usuario, configurar las credenciales de la conexión WiFi, y llamar a las funciones *setup()* y *loop()* en la aplicación de usuario. Dispone de una copia del gestor de arranque y, si encuentra que la versión del gestor de arranque es inferior a la de la copia, actualizará el gestor de arranque automáticamente.

La aplicación de usuario se denomina *User-part*. La aplicación de usuario basada en el firmware de Particle se extiende desde la dirección de memoria *Flash* interna 0x080C0000, hasta la 0x080FFFFF, abarcando un tamaño de 256KB. Se compone de las funciones *setup()* y *loop(),* al igual que en el caso del dispositivo *Photon*. En cuanto a la aplicación de usuario basada en el entorno de desarrollo WICED SDK, se extiende desde la dirección de memoria *Fl*ash interna 0x08010000, hasta la 0x080FFFFF, cuyo tamaño es de 960KB.

La imagen OTA se extiende desde la dirección de memoria *Flash* externa 0xC0000 hasta la 0x13FFFF, abarcando un tamaño de 512KB. Es el *firmware* del sistema o la aplicación de usuario descargada OTA. Las imágenes descargadas OTA se aplicarán a la memoria Flash interna durante el próximo arranque.

La imagen de reinicio de fábrica (*Factory Reset Application,* FAC), de tamaño 256KB, comprende desde la dirección de memoria *Flash* externa 0x140000, hasta la 0x17FFFF. Debe ser una aplicación de usuario validada de modo que, incluso si se ha cargado una aplicación de usuario incorrecta en memoria *Flash* interna, simplemente puede copiar la imagen de restablecimiento de fábrica a la ubicación donde se encuentra la aplicación de usuario al realizar una acción de restablecimiento de fábrica.

El *firmware* de la conexión WiFi se encuentra comprendido entre la dirección de memoria *Flash* externa 0x180000, y el final de la memoria *Flash* externa, abarcando un tamaño de 512KB. Se cargará en el chip BCM43438, en la parte del *firmware* del sistema *System-part 2* durante el proceso de inicialización.

La región de datos de usuario va desde la dirección de memoria *Flash* externa 0x00000, hasta la 0xB7FFF, representando un tamaño de 736KB. También se usa para almacenar datos de usuario no volátiles.

En cuanto a las funciones que proporciona el *firmware* del dispositivo *RedBear Duo* relativas a conexiones inalámbricas BLE, incluye funciones propias de un dispositivo *Peripheral* y funciones propias de un dispositivo *Central*. Se proporcionan además funciones relativas a los roles *GATT Server* y *GATT Client*, definidos en BLE, que en el caso concreto del presente TFG son roles que adoptan el dispositivo *Peripheral* y el dispositivo Central (dispositivo móvil), respectivamente.

4.5.1 Implementación de un dispositivo *Peripheral*

Haciendo uso de las funciones proporcionadas por el *firmware* de *RedBear*, en el presente TFG se ha implementado la funcionalidad del dispositivo *Peripheral*, que a su vez ejerce el rol de GATT Server definido en BLE. El planteamiento establecido es el siguiente: se definirá un servicio con dos características, usándose una de ellas para

enviar, mediante notificaciones, el carácter reconocido a partir del gesto realizado con la mano, y la otra se propone para su uso futuro, por ejemplo como mecanismo para configurar algún parámetro que se considere en la funcionalidad del dispositivo *Peripheral*, o bien para la implementación de un mecanismo de confirmación de recepción de notificaciones por parte del dispositivo *Central*. Una vez establecida la funcionalidad del dispositivo *Peripheral*, se pasa a la descripción del código desarrollado, disponible en el fichero `ble-peripheral-duo-signtranslator.ino`.

En primer lugar, se define el tiempo mínimo y máximo del parámetro que define el valor de preferencia del Intervalo de Conexión, en este caso a 50 ms y 500 ms, respectivamente. Sin embargo, estos valores especifican únicamente la preferencia del dispositivo *Peripheral*, ya que el dispositivo *Central* es el que decide y establece el tiempo correspondiente al Intervalo de Conexión en el establecimiento de la conexión. Asimismo, se define el tiempo correspondiente a la latencia del dispositivo *Slave* (0 ms) y el tiempo de espera de supervisión de la conexión (10 s), así como el valor del parámetro *appearance,* el nombre elegido para el dispositivo, y el tamaño máximo para el valor de cada una de las Características (*char1* y *char2*) que se definen en el dispositivo, como puede verse en la Figura 79.

A continuación, se define el UUID asociado al Servicio y a las Características del dispositivo *Peripheral*, tal y como se muestra en la Figura 80, correspondiente en este caso a un único Servicio (*service1_uuid*) con dos Características (*char1_uuid* y *char2_uuid*).

```
117  #define MIN_CONN_INTERVAL          0x0028 // 50ms.
118  #define MAX_CONN_INTERVAL          0x0190 // 500ms.
119  #define SLAVE_LATENCY              0x0000 // No slave latency.
120  #define CONN_SUPERVISION_TIMEOUT   0x03E8 // 10s.
121
122  #define BLE_PERIPHERAL_APPEARANCE  BLE_APPEARANCE_UNKNOWN
123
124  #define BLE_DEVICE_NAME            "BLE_Peripheral"
125
126  // Length of characteristic value.
127  #define CHARACTERISTIC1_MAX_LEN    2
128  #define CHARACTERISTIC2_MAX_LEN    2
```

Figura 79. Definiciones de los diferentes parámetros de conexión

```
/*****************************************************
 *               Variable Definitions
 *****************************************************/

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

Figura 80. Definición del Servicio y las Características asociadas al dispositivo *Peripheral*

Posteriormente se especifican los diferentes parámetros relacionados con el proceso de *Advertising*, siendo estos el intervalo mínimo y máximo de *Advertising* (30 ms), el tipo (*ADV_IND*), el tipo de dirección (*PUBLIC*), la dirección (0), y los parámetros *channel map* (ALL) y *filter policy* (ANY), como se muestra en la Figura 81.

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

Figura 81. Parámetros de Advertising en el dispositivo Peripheral

Seguidamente se definen los datos relativos al proceso de *Advertising*. La primera unidad es la requerida para dispositivos BLE, y la segunda unidad se corresponde con el UUID del Servicio definido en el dispositivo Peripheral, como se muestra en la Figura 82.

```
// BLE peripheral advertising data
static uint8_t adv_data[] = {
    0x02,
    BLE_GAP_AD_TYPE_FLAGS,
    BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE,

    0x11,
    BLE_GAP_AD_TYPE_128BIT_SERVICE_UUID_COMPLETE,
    0x1e, 0x94, 0x8d, 0xf1, 0x48, 0x31, 0x94, 0xba,
    0x75, 0x4c, 0x3e, 0x50, 0x00, 0x00, 0x3d, 0x71
};
```

Figura 82. Especificación de los datos de *Advertising* del dispositivo *Peripheral*

Asimismo, se definen el parámetro *Complete Local Name* (*SIGN-TRL*), propio del proceso de *Scanning,* como puede verse en la Figura 83.

```
209  // BLE peripheral scan respond data
210  static uint8_t scan_response[] = {
211    0x08,
212    BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME,
213    'S', 'I',  'G', 'N', 'T', 'R', 'L'
214  };
215
216  // Characteristic value handle
217  static uint16_t character1_handle = 0x0000;
218  static uint16_t character2_handle = 0x0000;
219  // Buffer of characteristic value.
220  uint8_t characteristic1_data[CHARACTERISTIC1_MAX_LEN] = {0x00};
221  uint8_t characteristic2_data[CHARACTERISTIC2_MAX_LEN] = {0x00};
222
```

Figura 83. Especificación de parámetros del dispositivo *Peripheral*

En cuanto a las funciones de *callback* incluidas en el código, en primer lugar se define la función *deviceConnectedCallback(),* que toma como parámetros de entrada el estado de la conexión y el identificador de la conexión, que es un identificador asignado para la conexión en caso de que se establezca correctamente, como se muestra en la Figura 84. La función *onConnectedCallback()* es la encargada de registrar esta función, que es a la que se debe invocar cuando se complete el procedimiento de establecimiento de conexión con un dispositivo *Central* BLE.

```
240  void deviceConnectedCallback(BLEStatus_t status, uint16_t handle) {
241    switch (status) {
242      case BLE_STATUS_OK:
243        Serial.println("Device connected!");
244        connected_id = handle;
245        Serial.print("Device connected handle: ");
246        Serial.println(connected_id);
247
248        break;
249      default: break;
250    }
251  }
```

Figura 84. Función *deviceConnectedCallback()*

En segundo lugar se define la función *deviceDisconnectedCallback()*, que toma como parámetro de entrada el identificador de la conexión, como se muestra en la Figura 85. La función *onDisconnectedCallback()* es la encargada de registrar esta función, que es a la que se debe invocar cuando el dispositivo se desconecta, y se da por finalizada una conexión BLE.

```
253 /**
254  * @brief Disconnect handle.
255  *
256  * @param[in]  handle   Connect handle.
257  *
258  * @retval None
259  */
260 void deviceDisconnectedCallback(uint16_t handle) {
261   Serial.println("Device disconnected!");
262   connected_id = 0xFFFF;
263 }
```

Figura 85. Función *deviceDisconnectedCallback()*

En tercer lugar se define la función de *callback gattReadCallback()*, que es la función registrada por la función *onDataReadCallback()*, y a la que se invoca cuando el dispositivo al que está conectado lee el valor de una de las Características definidas anteriormente y devuelve la longitud del valor de dicha Característica. Esta función se muestra en la Figura 86.

```
265 /**
266  * @brief Callback for reading event.
267  *
268  * @note  If characteristic contains client characteristic configuration,
269  *        then client characteristic configuration handle is value_handle+1.
270  *        Now can't add user_descriptor.
271  *
272  * @param[in]  value_handle
273  * @param[in]  buffer
274  * @param[in]  buffer_size    Ignore it.
275  *
276  * @retval  Length of current attribute value.
277  */
278 uint16_t gattReadCallback(uint16_t value_handle, uint8_t * buffer, uint16_t buffer_size) {
279   uint8_t characteristic_len = 0;
280
281   Serial.print("Read value handler: ");
282   Serial.println(value_handle, HEX);
283
284   if (characterl_handle == value_handle) {
285     Serial.print("  - characteristic1 read: ");
286     memcpy(buffer, characteristic1_data, CHARACTERISTIC1_MAX_LEN);
287     characteristic_len = CHARACTERISTIC1_MAX_LEN;
288     for (uint8_t index = 0; index < CHARACTERISTIC1_MAX_LEN; index++) {
289       Serial.print(buffer[index], HEX);
290     }
291     Serial.println();
292   }
293   else if (character2_handle == value_handle) {
294     Serial.print("  - characteristic2 read: ");
295     memcpy(buffer, (uint8_t*) caracter, CHARACTERISTIC2_MAX_LEN);
296     characteristic_len = CHARACTERISTIC2_MAX_LEN;
297     for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN; index++) {
298       Serial.print(buffer[index], HEX);
299     }
300     Serial.println();
301   }
302   return characteristic_len;
303 }
```

Figura 86. Función *gattReadCallback()*

En cuarto lugar se define la función de *callback gattWriteCallback()* (Figura 87), función registrada por *onDataWriteCallback()*, y que es invocada cuando el dispositivo al que está conectado escribe el valor de una de las Características definidas anteriormente, o escribe el valor del *CCCD*. Esta función muestra por el terminal serie el valor escrito de la Característica, y devuelve un valor de tipo *int*.

A pesar de que en el presente TFG no se contempla ninguna funcionalidad asociada a la escritura de un valor en alguna de las características definidas por parte del dispositivo *Central*, se tiene en cuenta para futuros usos.

```
306 /**
307  * @brief Callback for writing event.
308  *
309  * @param[in]  value_handle
310  * @param[in]  *buffer       The buffer pointer of writing data.
311  * @param[in]  size          The length of writing data.
312  *
313  * @retval
314  */
315 int gattWriteCallback(uint16_t value_handle, uint8_t *buffer, uint16_t size) {
316   Serial.print("Write value handler: ");
317   Serial.println(value_handle, HEX);
318
319   if (characterl_handle == value_handle) {
320     memcpy(characteristicl_data, buffer, size);
321     Serial.print("characteristicl write value: ");
322     for (uint8_t index = 0; index < size; index++) {
323       Serial.print(characteristicl_data[index], HEX);
324       Serial.print(" ");
325     }
326     Serial.println(" ");
327   }
328   else if (character2_handle+1 == value_handle) {   // Client Characteristic C
329     Serial.print("characteristicl CCCD write value: ");
330     for (uint8_t index = 0; index < size; index++) {
331       Serial.print(buffer[index], HEX);
332       Serial.print(" ");
333     }
334     Serial.println(" ");
335   }
336   return 0;
337 }
```

Figura 87. Función *gattWriteCallback()*

Por último, en la Figura 88 se define la función *characteristic2_notify()*, que permite enviar las notificaciones para la Característica *char2_uuid*. Como puede verse en la línea 347, esta función recibe como parámetro el carácter identificado tras el proceso de clasificación. En las líneas 351 y 352 se define el vector *characteristic2_data* teniendo en cuenta que el tamaño máximo de los datos a enviar es de 2 bytes, ya que la letra recibida puede ser un conjunto de caracteres, como es el caso de “CH”, “LL” y “RR”. Además, en la línea 354 se especifica el valor ASCII de la letra “Ñ” para mostrarla

en pantalla cuando se detecte. Finalmente, se envía la notificación con la información requerida (línea 363).

```
339 /**
340  * @brief Send notify of characteristic to client when a character is detected.
341  *
342  * @param[in]  value_handle
343  * @param[in]  *caracter
344  *
345  * @retval
346  */
347 static void characteristic2_notify(char *caracter) {
348   if((connected_id != 0xFFFF) && ble.attServerCanSendPacket()) {
349     Serial.print("Characteristic2_notify: ");
350
351     characteristic2_data[CHARACTERISTIC2_MAX_LEN-2] = (uint8_t) caracter[1];
352     characteristic2_data[CHARACTERISTIC2_MAX_LEN-1] = (uint8_t) caracter[2];
353     Serial.print("caracter = ");
354     if ((caracter[2] == 0xC3) && (caracter[1] == 0xB1)) {
355         Serial.print("ñ");
356     }
357     else {
358       for (uint8_t index = 0; index < CHARACTERISTIC2_MAX_LEN+1; index++) {
359         Serial.print(caracter[index]);
360       }
361     }
362     Serial.println("");
363     ble.sendNotify(caracter2_handle, characteristic2_data, CHARACTERISTIC2_MAX_LEN);
364   }
365 }
```

Figura 88. Función *characteristic2_notify()*

Por otro lado, en la función *setup()* del código desarrollado se encuentran:

- La inicialización de la interfaz HCI entre el *Host* y el controlador, así como la inicialización del controlador al estado predeterminado (Figura 89)

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

Figura 89. Inicialización de la interfaz HCI

- El registro de las funciones de *callback* (Figura 90)

```
423   // Register BLE callback functions.
424   ble.onConnectedCallback(deviceConnectedCallback);
425   ble.onDisconnectedCallback(deviceDisconnectedCallback);
426   ble.onDataReadCallback(gattReadCallback);
427   ble.onDataWriteCallback(gattWriteCallback);
```

Figura 90. Registro de las funciones de *callback* en el dispositivo *Peripheral*

- La incorporación de los Servicios y Características tanto del *GAP* como del *GATT Server* (Figura 91)

- El establecimiento de los parámetros y los datos del proceso de *Advertising* y la llamada a la función *ble.startAdvertising()* para que el dispositivo comience el proceso de *Advertising* (Figura 92)

```
// 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));

// Add GATT service and characteristics
ble.addService(BLE_UUID_GATT);
ble.addCharacteristic(BLE_UUID_GATT_CHARACTERISTIC_SERVICE_CHANGED,
                      ATT_PROPERTY_INDICATE, change, sizeof(change));

// Add primary service1.
ble.addService(service1_uuid);
// Add characteristic to service1, return value handle of characteristic.
character1_handle = ble.addCharacteristicDynamic(char1_uuid,
                    ATT_PROPERTY_READ|ATT_PROPERTY_NOTIFY|ATT_PROPERTY_WRITE_WITHOUT_RESPONSE,
                    characteristic1_data, CHARACTERISTIC1_MAX_LEN);
character2_handle = ble.addCharacteristicDynamic(char2_uuid,
                    ATT_PROPERTY_READ|ATT_PROPERTY_NOTIFY,
                    characteristic2_data, CHARACTERISTIC2_MAX_LEN);
```

Figura 91. Incorporación de los Servicios y las Características del GAP en el dispositivo *Peripheral*

```
    // Set BLE advertising parameters
    ble.setAdvertisementParams(&adv_params);

    // Set BLE advertising and scan respond data
    ble.setAdvertisementData(sizeof(adv_data), adv_data);
    ble.setScanResponseData(sizeof(scan_response), scan_response);

    // Start advertising.
    ble.startAdvertising();
    Serial.println("BLE start advertising.");
}
```

Figura 92. Incorporación de los Servicios y Características del GATT Server

Cabe destacar que la Característica *character1* posee propiedades de notificación (ATT_PROPERTY_NOTIFY) y de escritura sin respuesta (ATT_PROPERTY_WRITE WITHOUT_RESPONSE), mientras que la Característica *character2* posee propiedades de notificación (ATT_PROPERTY_NOTIFY) y de lectura (ATT_PROPERTY_READ).

Por último, dentro de la función *loop(),* se realiza la llamada a la función *characteristic2_notify()* después de realizar cada predicción y tomando como parámetro de entrada el carácter detectado, como se muestra en la Figura 93. De esta manera, el dispositivo *Peripheral* enviará una notificación al dispositivo conectado cada vez que se produzca un cambio en el valor de la Característica, representando con fines de depuración por el terminal serie el valor de la misma.

```
666             svm_predict(sensors);
667
668             available = 0;
669             // stop recording
670             sampleTimer.end();
671             isSampling = false;
672
673
674             characteristic2_notify(caracter);
675         }
676       }
677     }
678   }
```

Figura 93. Sección de código de la función *loop()* correspondiente a la parte de BLE

4.5.2 Validación funcional del código implementado

Una vez implementada la funcionalidad del dispositivo *Peripheral BLE* a partir del *firmware* de usuario proporcionado por *RedBear*, se comprueba inicialmente su correcto funcionamiento mediante la aplicación *BLE Scanner* [32] para Android como dispositivo *Centra BLE*.

Al iniciar la aplicación *BLE Scanner* una vez se ha montado la plataforma y cargado el código en el dispositivo *RedBear Duo*, se seleccionará la opción *BLE Scanner* de la parte inferior izquierda de la pantalla (Figura 94) para que el dispositivo *Central* inicie el proceso de *Scan* en busca de dispositivos *Peripheral* que se encuentren en su entorno en proceso de *Advertising*.

Figura 94. Aplicación *BLE Scanner* – Opciones

Una vez hecho esto, aparece el dispositivo *Peripheral* denominado SIGNTRL (Figura 95), tal como se configuró en el código implementado y comentado anteriormente.

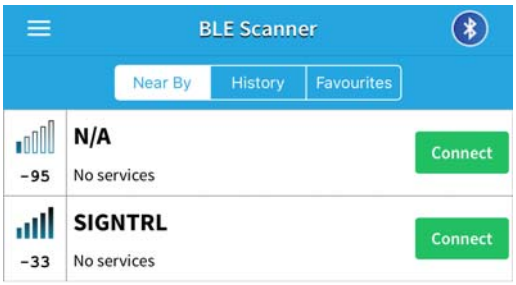


Figura 95. Aplicación BLE Scanner - Dispositivos Peripheral descubiertos

A continuación, se pulsa el botón *Connect*, situado al lado del nombre del dispositivo *Peripheral*, para iniciar la conexión entre la aplicación *BLE Scanner*, que actúa como dispositivo *Central,* y el dispositivo *Peripheral*. Es entonces cuando aparece el Servicio implementado en el dispositivo *Peripheral*, con UUID 713D0000-503E-4C75-BA94-3148F18D941E, como se muestra en la Figura 96.

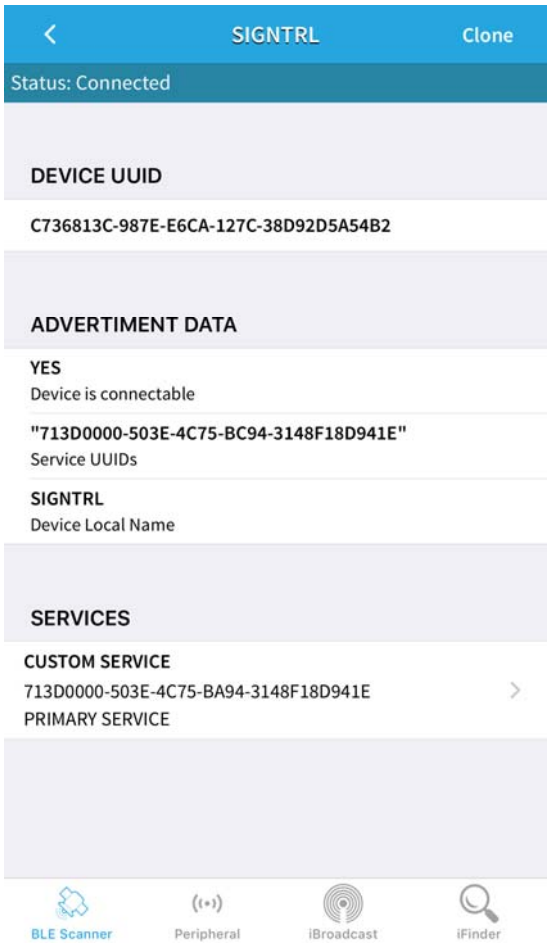


Figura 96. Aplicación BLE Scanner - Servicio del dispositivo *Peripheral*

Finalmente, se pulsa en la opción CUSTOM SERVICE para desplegar todas las Características definidas en el Servicio *service1* del dispositivo *Peripheral* y habilitar la recepción de las notificaciones asociadas a la Característica *character2* con UUID 713D0003-503E-4C75-BA94-3148F18D941E. En la Figura 97 se puede ver cómo inicialmente el dispositivo *Central* no está recibiendo ninguna notificación por parte del dispositivo *Peripheral,* y seguidamente en el apartado "Value" recibe correctamente la letra "a" (Figura 98).

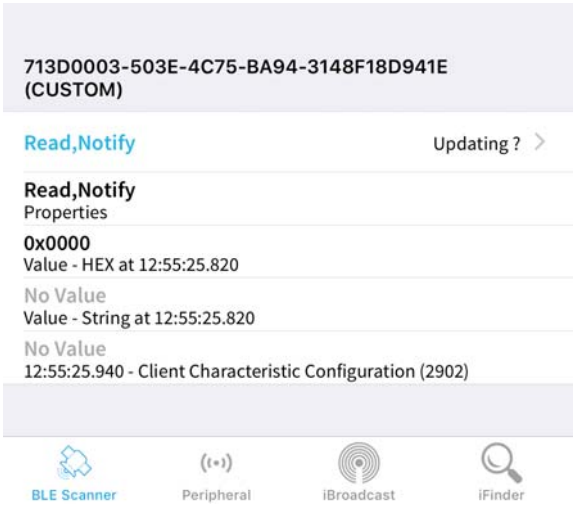


Figura 97.Aplicación BLE Scanner - Notificaciones recibidas (No Value)

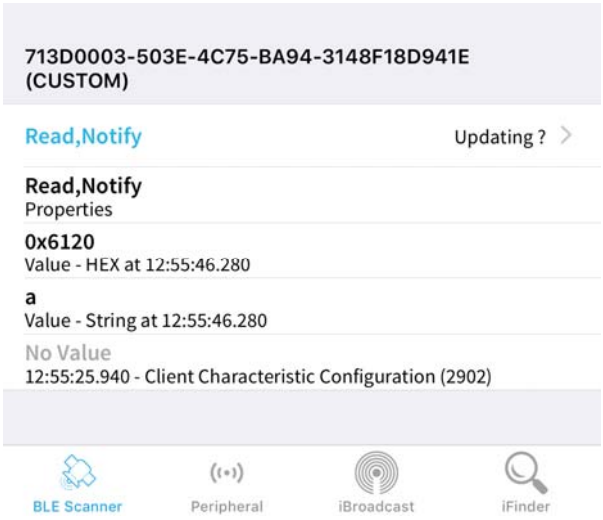


Figura 98. Aplicación BLE Scanner - Notificaciones recibidas (letra a)

Capítulo 5. Plataforma HW/SW final

En este capítulo se detalla la interconexión de todos los dispositivos que constituyen la plataforma HW/SW final, así como el *firmware* desarrollado para la correcta interpretación de todos los símbolos del alfabeto dactilológico de la Lengua de Signos Española. Por último, se recoge la validación funcional de la plataforma a partir de la realización de diferentes pruebas experimentales de uso.

5.1 Conexión de los componentes

En este apartado se describe la interconexión de los componentes *hardware* que forman parte de la plataforma HW/SW final desarrollada en el presente TFG, así como los resultados obtenidos a partir de su validación funcional tras integrar todos los componentes.

Cabe destacar que para esta última versión se utiliza el dispositivo *SparkFun Photon IMU Shield* [33] mostrado en la Figura 99, que además de integrar el módulo LSM9DS1 utilizado en la versión inicial de la plataforma HW/SW, facilita la conexión directa con el dispositivo *RedBear Duo*, sin necesidad de realizar ningún cableado externo, pues la disposición de los pines de ambos coincide.

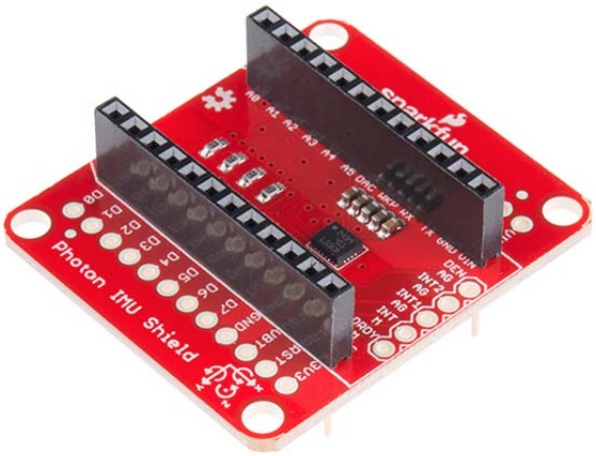


Figura 99. Dispositivo *Photon IMU Shield* de Sparkfun

Por lo tanto, la plataforma final está formada por los siguientes componentes HW: el dispositivo *RedBear Duo*, el complemento *Photon IMU Shield* que integra el módulo LSM9DS1, cinco *flex sensors* asociados a cada uno de los dedos de la mano, y las resistencias necesarias para completar el montaje de los sensores flexibles. Además, con el fin de lograr que la plataforma no dependa de un PC para realizar el proceso de identificación de los símbolos, una vez realizado el entrenamiento, se añade una batería recargable para alimentar adecuadamente los diferentes elementos de la plataforma, y un pulsador adicional cuya finalidad se explicará con posterioridad.

Así, en la Figura *100* se ilustra un esquema del conexionado de todos los dispositivos requeridos en la implementación de la plataforma HW/SW final, que puede ser alimentada, o bien a través del puerto USB del dispositivo *RedBear Duo*, o bien mediante la batería externa añadida.

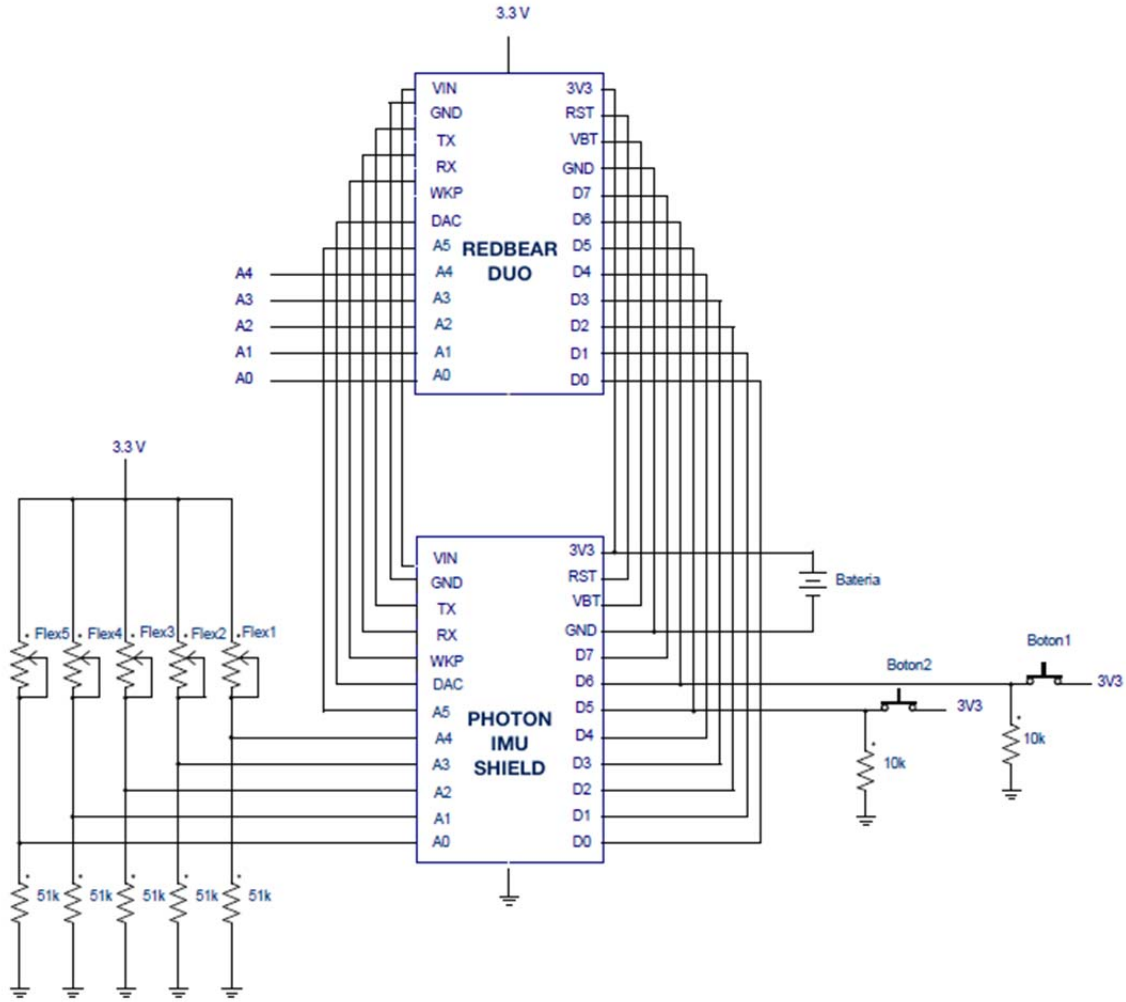


Figura 100. Esquemático de la plataforma final

Así, tras conectar todos los elementos sobre una *protoboard* de tamaño más reducido que la utilizada en el desarrollo de la plataforma HW/SW inicial, y pegarla al guante, se obtiene la plataforma HW/SW final mostrada en la Figura 101. Como puede verse, se consigue el grado de autonomía que se pretendía mediante el uso de la batería recargable.

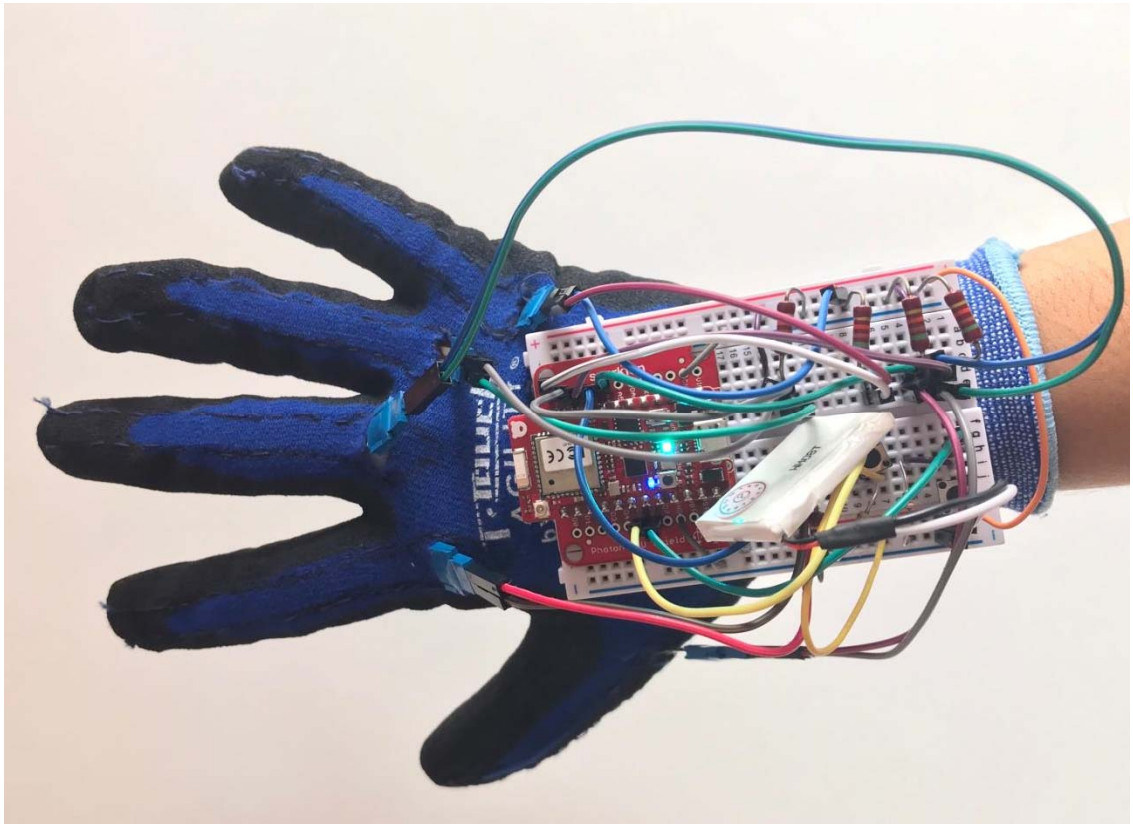


Figura 101. Plataforma HW/SW final

5.2 *Desarrollo Firmware*

Atendiendo al desarrollo del código asociado al *firmware* de la plataforma HW/SW, contenido en el fichero `svm-train-predict-ble.ino`, con respecto a la versión inicial se ha ampliado el menú de usuario a cuatro opciones, tal como se muestra en la Figura 102. El menú de usuario se sitúa dentro de la función *setup()* junto con la inicialización de los parámetros utilizados, y parte del código asociado a la comunicación BLE, tal como se comentó en el Capítulo 4.

```
407   Serial.println("MENU:");
408   Serial.println("-----------------------------------------------");
409   Serial.println("1. Capture Hand Gesture Data for training SVM");
410   Serial.println("2. Print Data for training SVM");
411   Serial.println("3. SVM Scale Data for training");
412   Serial.println("4. Predict Hand Gestures");
413   Serial.println("-----------------------------------------------");
414   Serial.print ("Enter the number corresponding to the menu option to perform: ");
```

Figura 102. Menú de usuario de la plataforma final

Además, se ha incluido el mecanismo asociado al LED azul situado junto al pin *D7* utilizado en la plataforma inicial para la lectura de datos, en el proceso de reconocimiento de la opción 4, de tal manera que se recogen los datos durante el tiempo que permanece el pin a nivel lógico *HIGH,* es decir, mientras el LED esté encendido.

Otra modificación a tener en cuenta es que en esta plataforma final se definen seis medidas por cada una de las letras a entrenar, como puede verse en el valor de la constante NUMBER_MEAN_VALUES en la Figura *103*.

```
17  #define NUMBER_MEAN_VALUES 6 // number of mean values per button press
18  #define NUMBER_SAMPLES 50    // number of samples per mean value (50)
19  #define SAMPLE_PERIOD  40    // sample period (40) (hms)
```

Figura 103. Definición de las constantes para la toma de muestras

La sección de *firmware* que se ejecuta en cada momento dependerá de la opción seleccionada por el usuario: al seleccionar 1 como opción en la interfaz que se muestra en el PC a través del puerto serie USB, se realizará la captura de datos de los diferentes sensores conectados al dispositivo *RedBear Duo* con el fin de completar el entrenamiento SVM para cada uno de los símbolos, mientras que con la opción 2 se imprimen por pantalla los datos recogidos, con la opción 3 se realiza en el dispositivo *RedBear Duo* el escalado de estos datos, y con la opción 4 se activan las funciones que permiten predecir la clase a la que pertenece el gesto realizado en cada momento.

Así, dentro de la función *loop()* del *firmware* se lee qué opción del menú es seleccionada en cada momento, como puede verse en la sección de código recogida en la Figura *104*. Cabe destacar que en la plataforma HW/SW final se ha añadido un segundo botón para activar la variable *enablePrediction* (líneas 477-482) y acceder así a las funcionalidades de la opción 4, ya que de esta manera no se necesita de un PC una vez realizado el entrenamiento. Al seleccionar la opción 1, se crea la variable *inID* para poder recibir dos bytes por el puerto serie cuando se introduce la letra a entrenar, permitiendo con ello la lectura de las letras “CH”, “LL” y “RR”. Con esta modificación también se hace posible leer la letra “Ñ”, cuya codificación en formato Unicode es 0xC3B1, como se especifica en la línea 506. Así, los números asociados a las

letras introducidas en la opción 1 se asignan a la variable *symbol,* y seguidamente se activa la variable *enableSampling* poniéndose a *true* para permitir la ejecución de la sección de código asociada a esta primera opción.

```
474  void loop()
475  {
476
477    bool predictbuttonPressed = digitalRead(PREDICTPIN);
478    if (predictbuttonPressed && !predictbuttonAlreadyPressed && !enableSampling && !enablePrediction)
479    {
480      enablePrediction = true;
481    }
482    predictbuttonAlreadyPressed = predictbuttonPressed;
483
484
485    if ((Serial.available() > 0) && (!enableSampling)) {
486       char inOption = Serial.read();
487
488       Serial.println(inOption);
489       if (inOption == '1') {
490          Serial.print ("Enter the letter for the hand gesture data: ");
491          char inID[2]={0};
493          while (!(Serial.available() > 0)) Particle.process(); // wait for serial port
494          int bytesRead = Serial.readBytes(inID, 2);
495          for (int i = 0; i < bytesRead; i++)
496             Serial.print(inID[i]);
497
498          Serial.println("");
499          if ((bytesRead == 1) && (inID[0] == ' ')) {
500            enableSampling = false;
501            Serial.println("");
502            Serial.print ("Enter the number corresponding to the menu option to perform: ");
503          }
504          else {
505            if ( ((bytesRead == 1) && (((inID[0] >= 'a') && (inID[0] <= 'z')))) || ((bytesRead == 2) &&
506            ((inID[0] == 0xC3) && (inID[1] == 0xB1))) ) {
507              enableSampling = true;
508              n_values = 0;
509              symbol = (int) inID[0]; // the symbol associated to the sampled mean values
510            }
511            else if ((bytesRead == 2) && (((inID[0] == 'c') && (inID[1] == 'h')) || ((inID[0] == 'l') &&
512            (inID[1] == 'l')) || ((inID[0] == 'r') && (inID[1] == 'r')))) {
513              enableSampling = true;
514              n_values = 0;
515              symbol = (int) (inID[0] + inID[1]); // the symbol associated to the sampled mean values
516            }
517            else {
518              enableSampling = false;
519              Serial.println("");
520              Serial.print ("Enter the number corresponding to the menu option to perform: ");
521            }
522          }
523       }
524       else if (inOption == '2') {
525          printtrainValues();
526          Serial.println("");
527          Serial.print ("Enter the number corresponding to the menu option to perform: ");
528       }
529       else if (inOption == '3') {
530          scaleValues();
531          Serial.println("");
532          Serial.print ("Enter the number corresponding to the menu option to perform: ");
533       }
534       else if (inOption == '4') {
535          enablePrediction = true;
536       }
537       else {
538          Serial.println("Not a valid option");
539          Serial.println("");
540          Serial.print ("Enter the number corresponding to the menu option to perform: ");
541      }
542  }
```

Figura 104. Función loop() de la plataforma final – primera parte

La sección de código correspondiente a la opción 1 también se encuentra dentro de la función *loop().* En la Figura *105* se muestra el diagrama de flujo del código correspondiente a esta primera opción. Atendiendo a este diagrama de flujo, una vez seleccionada la letra a entrenar, el mecanismo para tomar las muestras durante el entrenamiento es el siguiente: se pulsa el Botón 1 para iniciar la recogida de muestras, momento en el que se encenderá el LED azul *(LEDPIN)* durante dos segundos con el fin de mostrar el tiempo disponible para representar el símbolo. Una vez transcurrido ese tiempo, se realizan los cálculos necesarios para generar el modelo de predicción. Mientras no se seleccione otra opción, este mecanismo se repite hasta obtener seis muestras de la letra seleccionada, es decir, cuando *n_values* coincida con el valor establecido en la constante NUMBER_MEAN_VALUES definida.

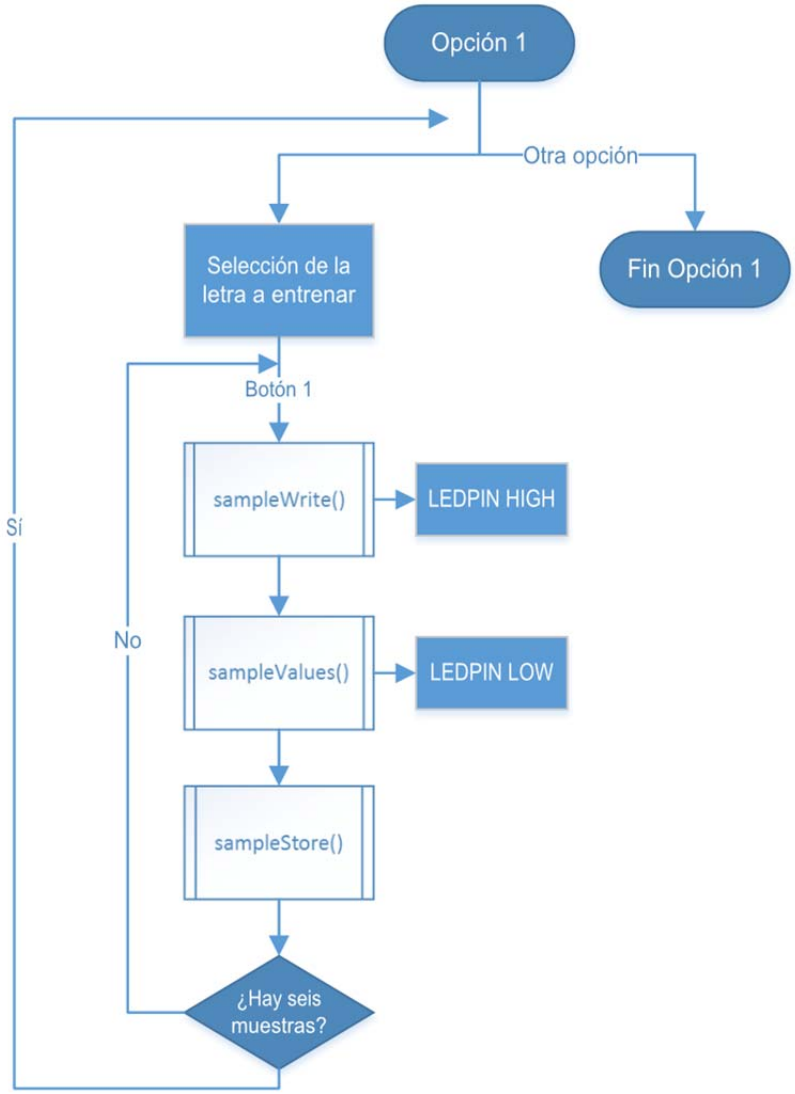


Figura 105. Diagrama de flujo opción 1 de la plataforma final

En la Figura *106* se recoge la sección código de la función *loop()* una vez la variable *enableSampling* se ha establecido a *true.* Como puede verse en las líneas 554-559, al pulsar el Botón 1 *(BUTTONPIN)* se activa la variable *enableCaptureSampling*, momento en el que se iniciará un *timer* implementado a partir del uso de la librería *SparkIntervalTimer*, que a su vez llamará a la función *sampleWrite()* para, a partir de los sensores, tomar 50 muestras con un periodo de muestreo de 40 ms. Esto queda definido en la línea 565. Además, durante este tiempo se encenderá el LED azul *(LEDPIN)*, como puede verse en la línea 566. Una vez tomadas las muestras establecidas, se apaga el LED azul y se llama a la función *sampleValues().* Seguidamente se llama a la función *sampleStore(),* y en la línea 592 se comprueba si el número de muestros realizados (*n_values*) es 6; en caso afirmativo se pasa al entrenamiento de una nueva letra y se le suman seis nuevos vectores al valor de la variable *n_train_vectors* (línea 598).

```
549   if (enableSampling)
550   {
551     // Checks if BUTTONPIN is pressed to take NUMBER_SAMPLES samples
552     // (one sample each SAMPLE_PERIOD ms) in order to generate
553     // NUMBER_MEAN_VALUES mean values
554     bool buttonPressed = digitalRead(BUTTONPIN);
555     if(buttonPressed && !buttonAlreadyPressed)
556     {
557       enableCaptureSampling = true;
558     }
559     buttonAlreadyPressed = buttonPressed;
560
561     if (enableCaptureSampling)
562     {
563       if (!isSampling)
564       {
565         sampleTimer.begin(sampleWrite, SAMPLE_PERIOD, hmSec);
566         digitalWrite(LEDPIN, HIGH);
567         isSampling = true;
568       }
569       else
570       {
571         if (available == NUMBER_SAMPLES)
572         {
573           // print the symbol
574           Serial.print(symbol);
575           Serial.print(" ");
576
577           digitalWrite(LEDPIN, LOW);
578
579           available = 0;
580
581           sampleValues();
582           if (indx_y[symbol] == -1)
583              sampleStore(n_train_vectors);
584           else
585              sampleStore(indx_y[symbol]);
586
587           // stop recording
588           sampleTimer.end();
589           isSampling = false;
590           enableCaptureSampling = false;
591           n_values++;
592
593
594           if (n_values == NUMBER_MEAN_VALUES)
595           {
```

```
596             if (indx_y[symbol] == -1) {
597               indx_y[symbol] = n_train_vectors;
598               n_train_vectors = n_train_vectors + NUMBER_MEAN_VALUES;
599             }
600
601             Serial.print ("Enter the letter for the hand gesture data: ");
602             char inID[2]={0};
603             Serial.setTimeout(500);
604             while (!(Serial.available() > 0)) Particle.process(); // wait for serial port
605             int bytesRead = Serial.readBytes(inID, 2);
606             for (int i = 0; i < bytesRead; i++)
607                Serial.print(inID[i]);
608
609             Serial.println("");
610             if ((bytesRead == 1) && (inID[0] == ' ')) {
611               enableSampling = false;
612               Serial.println("");
613               Serial.print ("Enter the number corresponding to the menu option to perform: ");
614             }
615             else {
616               if ( ((bytesRead == 1) && (((inID[0] >= 'a') && (inID[0] <= 'z')))) || ((bytesRead == 2)
617               && ((inID[0] == 0xC3) && (inID[1] == 0xB1))) ) {
618                 enableSampling = true;
619                 n_values = 0;
620                 symbol = (int) inID[0]; // the symbol associated to the sampled mean values
621               }
622               else if ((bytesRead == 2) && (((inID[0] == 'c') && (inID[1] == 'h')) || ((inID[0] == 'l')
623               && (inID[1] == 'l')) || ((inID[0] == 'r') && (inID[1] == 'r')))) {
624                 enableSampling = true;
625                 n_values = 0;
626                 symbol = (int) (inID[0] + inID[1]); // the symbol associated to the sampled mean values
627               }
628               else {
629                 enableSampling = false;
630                 Serial.println("");
631                 Serial.print ("Enter the number corresponding to the menu option to perform: ");
632               }
633             }
634           }
635         }
636       }
637     }
638   }
```

Figura 106. Función *loop()* – Sección de código correspondiente a la opción 1 de la plataforma final

En la Figura *107* se muestra la sección de código correspondiente a la función *sampleWrite()*. Por un lado (líneas 771-798), se recogen los valores leídos por los pines analógicos para, siguiendo el procedimiento visto en el apartado 2.3.2, estimar el ángulo de doblez de cada una de los sensores flexibles. Por otro lado, se recogen los valores asociados al acelerómetro integrado en el dispositivo LSM9DS1 del complemento *Photon IMU Shield* (líneas 800-810) para calcular el *pitch* en la línea 812 y el *roll* en la línea 813. Seguidamente, en las líneas 815 y 816 se convierten de radianes a grados los valores de *pitch* y *roll*, respectivamente. Por último, en las líneas 819-840 se recogen en distintas matrices todos los datos necesarios durante el proceso de muestreo, es decir, los ángulos de doblez de cada sensor flexible, las medidas de *pitch* y *roll*, y el valor de cada uno de los tres ejes del acelerómetro.

```
770 void sampleWrite(void) {
771     // Read the ADC, and calculate voltage and resistance from it
772     int flexADC0 = analogRead(FLEX_PIN0);
773     float flexV0 = flexADC0 * VCC / 4095.0;
774     float flexR0 = R_DIV*(VCC/flexV0-1.0);
775 
776     int flexADC1 = analogRead(FLEX_PIN1);
777     float flexV1 = flexADC1 * VCC / 4095.0;
778     float flexR1 = R_DIV*(VCC/flexV1-1.0);
779 
780     int flexADC2 = analogRead(FLEX_PIN2);
781     float flexV2 = flexADC2 * VCC / 4095.0;
782     float flexR2 = R_DIV*(VCC/flexV2-1.0);
783 
784     int flexADC3 = analogRead(FLEX_PIN3);
785     float flexV3 = flexADC3 * VCC / 4095.0;
786     float flexR3 = R_DIV*(VCC/flexV3-1.0);
787 
788     int flexADC4 = analogRead(FLEX_PIN4);
789     float flexV4 = flexADC4 * VCC / 4095.0;
790     float flexR4 = R_DIV*(VCC/flexV4-1.0);
791 
792     // Use the calculated resistance to estimate the sensor's bend angle:
793     float value[NUMBER_SENSORS];
794     value[0] = map(flexR0, STRAIGHT_RESISTANCE0, BEND_RESISTANCE0,0.0, 90.0);
795     value[1] = map(flexR1, STRAIGHT_RESISTANCE1, BEND_RESISTANCE1,0.0, 90.0);
796     value[2] = map(flexR2, STRAIGHT_RESISTANCE2, BEND_RESISTANCE2,0.0, 90.0);
797     value[3] = map(flexR3, STRAIGHT_RESISTANCE3, BEND_RESISTANCE3,0.0, 90.0);
798     value[4] = map(flexR4, STRAIGHT_RESISTANCE4, BEND_RESISTANCE4,0.0, 90.0);
799 
800     float valueacc[NUMBER_ACC];
801     int16_t a[NUMBER_MMACC];
802 
803     uint8_t lectura = imu.readAccel();
804     if (lectura == 0) {
805       Serial.println("*** ERROR imu.readAccel");
806     }
807 
808     a[0] = imu.ax;
809     a[1] = imu.ay;
810     a[2] = imu.az;
811 
812     valueacc[0] = atan2(a[1], a[2]);
813     valueacc[1] = atan2(-a[0], sqrt(a[1] * a[1] + a[2] * a[2]));
814     // Convert from radians to degrees:
815     valueacc[0] *= 180.0 / M_PI;
816     valueacc[1]  *= 180.0 / M_PI;
817 
818 
819     if (isSampling)
820     {
821       if (available == NUMBER_SAMPLES)
822         return;
823 
824       for (int k = 0; k < NUMBER_SENSORS; k++)
825         mBuffer[k][mWritePos] = value[k];
826 
```

```
826
827       for (int k = 0; k < NUMBER_ACC; k++)
828         maccBuffer[k][mWritePos] = valueacc[k];
829
830       for (int k = 0; k < NUMBER_MMACC; k++)
831         mmmaccBuffer[k][mWritePos] = a[k];
832
833       mWritePos++;
834
835       if (mWritePos == NUMBER_SAMPLES)
836         mWritePos = 0;
837
838       available++;
839     }
```

Figura 107. Función *sampleWrite()* en la plataforma final

La sección de código de la función *sampleValues()* se recoge en la Figura *108*. Tiene tres partes diferenciadas: en la primera parte (líneas 706-720) se calcula el valor medio para cada sensor flexible, en la segunda parte (líneas 722-735) se calcula el valor medio para las medidas de *pitch* y *roll* del acelerómetro, y en la tercera parte (líneas 738-760) se calcula la desviación típica de los tres ejes (x, y, z) del acelerómetro. Esto último se debe a que, tras unas pruebas iniciales con una especialista en interpretación de la Lengua de Signos, se llegó a la conclusión de que las medidas de *pitch* y *roll* no eran suficientes para distinguir letras como la "Y", donde la posición de la mano es similar a la de las letras "I", "J" y "Z", tal como se vio en el capítulo 2.

Por lo tanto, a diferencia de en la plataforma inicial, donde los sensores 8, 9 y 10 se correspondían con la máxima diferencia en el movimiento asociado a cada uno de los ejes del acelerómetro, los sensores 8, 9 y 10 de la plataforma HW/SW final se corresponden con los valores de la desviación típica.

```
705 void sampleValues(void) {
706     // calculate the mean value for each sensor
707     float sum_value[NUMBER_SENSORS];
708     for (int j = 0; j < NUMBER_SENSORS; j++)
709     {
710        sum_value[j] = 0;
711        for (int i = 0; i < NUMBER_SAMPLES; i++)
712        {
713           sum_value[j] = (sum_value[j] + mBuffer[j][i]);
714        }
715        mean_value[j] = sum_value[j] / (float) NUMBER_SAMPLES;
716        Serial.print(j+1);
717        Serial.print(":");
718        Serial.print((int) mean_value[j]);
719        Serial.print(" ");
720     }
```

```
721     // calculate the mean value for each acc
722     float sum_value_acc[NUMBER_ACC];
723     for (int j = 0; j < NUMBER_ACC; j++)
724     {
725        sum_value_acc[j] = 0;
726        for (int i = 0; i < NUMBER_SAMPLES; i++)
727        {
728           sum_value_acc[j] = (sum_value_acc[j] + maccBuffer[j][i]);
729        }
730        mean_value_acc[j] = sum_value_acc[j] / (float) NUMBER_SAMPLES;
731        Serial.print(j+NUMBER_SENSORS+1);
732        Serial.print(":");
733        Serial.print((int) (mean_value_acc[j]));
734        Serial.print(" ");
735     }
736
737     // calculate the stdev value for each mmacc
738     float sum_value_mmacc[NUMBER_MMACC];
739     for (int j = 0; j < NUMBER_MMACC; j++)
740     {
741        sum_value_mmacc[j] = 0;
742        for (int i = 0; i < NUMBER_SAMPLES; i++)
743        {
744           sum_value_mmacc[j] = (sum_value_mmacc[j] + (float) mmmaccBuffer[j][i]);
745        }
746        mean_value_mmacc[j] = sum_value_mmacc[j] / (float) NUMBER_SAMPLES;
747
748        // stdev
749        float sqDevSum = 0.0;
750        for(int i = 0; i < NUMBER_SAMPLES; i++)
751        {
752           sqDevSum += pow((mean_value_mmacc[j] - ((float) (mmmaccBuffer[j][i]))), 2);
753        }
754        stDev[j] = sqrt(sqDevSum / (float) NUMBER_SAMPLES);
755
756        Serial.print(j+NUMBER_SENSORS+NUMBER_ACC+1);
757        Serial.print(":");
758        Serial.print((int) (stDev[j] / 10));
759        Serial.print(" ");
760     }
761     Serial.println("");
762  }
```

Figura 108. Función *sampleValues()* de la plataforma final

Finalmente, se recoge la función *sampleStore()* en la Figura *109*, encargada de almacenar todos los valores recogidos de los sensores durante el entrenamiento .

```
693  void sampleStore(int n) {
694       prob_y[n+n_values] = (double) symbol;
695       for (int j = 0; j < NUMBER_SENSORS; j++)
696         prob_x[n+n_values][j] = (int) mean_value[j];
697       for (int j = 0; j < NUMBER_ACC; j++)
698         prob_x[n+n_values][j+NUMBER_SENSORS] = (int) mean_value_acc[j];
699       for (int j = 0; j < NUMBER_MMACC; j++)
700         prob_x[n+n_values][j+NUMBER_SENSORS+NUMBER_ACC] = (int) (stDev[j] / 10);
701  }
```

Figura 109. Función *sampleStore()*

La opción 2 se ha añadido para facilitar la recogida de todos los datos generados durante el proceso de entrenamiento. Como puede verse en la Figura *110*, esta opción únicamente activa la función *printtrainValues(),* mostrada en la Figura *111*, que haciendo uso de la variable *n_train_vectors*, imprime por pantalla todos los datos registrados para cada una de las letras utilizadas en el proceso de entrenamiento.

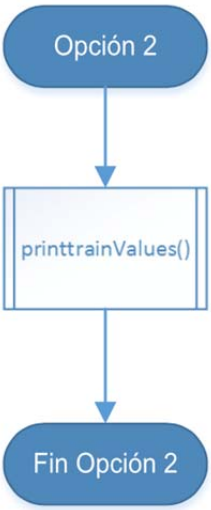


Figura 110. Diagrama de flujo de la opción 2 de la plataforma final

```
void printtrainValues(void) {
    if (n_train_vectors == 0) {
        Serial.println("ERROR - no training vectors");
    }
    else {
      for (int i = 0; i < n_train_vectors; i++)
      {
        Serial.print((int) prob_y[i]);
        Serial.print(" ");
        for (int j = 0; j < (NUMBER_SENSORS + NUMBER_ACC + NUMBER_MMACC); j++)
        {
           Serial.print(j+1);
           Serial.print(":");
           Serial.print(prob_x[i][j]);
           Serial.print(" ");
        }
        Serial.println();
      }
    }
}
```

Figura 111. Función printtrainValues()

La opción 3 es la asociada a la función *scaleValues()*, encargada de realizar el escalado de los datos entrenados en el dispositivo *RedBear Duo*. Esta función permite que sólo sea necesario recurrir a la librería LIBSVM a través de la ventana de comandos en un PC, para la generación del modelo de entrenamiento. En la Figura *112* pueden verse las

funciones utilizadas en esta opción. El código de la función *scaleValues()* es el recogido en la Figura *113*. En las líneas 879-892 se seleccionan los valores máximos y mínimos de cada sensor, mientras que en las líneas 898-912 escala los valores obtenidos por cada sensor durante el proceso de entrenamiento.

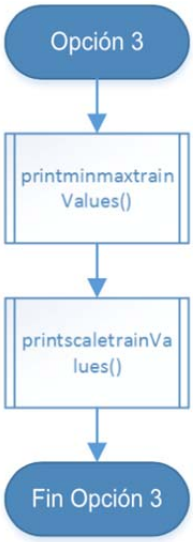


Figura 112. Diagrama de bloques de la opción 3 de la plataforma final

```
877 void scaleValues(void) {
878     if (n_train_vectors == 0) {
879         Serial.println("ERROR - no training vectors");
880     }
881     else {
882       // calculate max/min values from training data for each sensor
883       for (int j = 0; j < (NUMBER_SENSORS + NUMBER_ACC + NUMBER_MMACC); j++)
884       {
885         low_train[j] = INT_MAX;
886         high_train[j] = -INT_MAX;
887       }
888
889       for (int i = 0; i < n_train_vectors; i++)
890       {
891         for (int j = 0; j < (NUMBER_SENSORS + NUMBER_ACC + NUMBER_MMACC); j++)
892         {
893             if (prob_x[i][j] < low_train[j])
894               low_train[j] = prob_x[i][j];
895             if (prob_x[i][j] > high_train[j])
896               high_train[j] = prob_x[i][j];
897         }
898       }
899       printminmaxtrainValues();
900
901       // scale values from training data for each sensor
902       for (int i = 0; i < n_train_vectors; i++)
903       {
904         for (int j = 0; j < (NUMBER_SENSORS + NUMBER_ACC + NUMBER_MMACC); j++)
905         {
906             if (prob_x[i][j] == low_train[j]) {
907               scaledprob_x[i][j] = (float) scaleParprob_x[0];
908             }
909             else if (prob_x[i][j] == high_train[j]) {
910               scaledprob_x[i][j] = (float) scaleParprob_x[1];
911             }
912             else {
913               scaledprob_x[i][j] = ((float)scaleParprob_x[0] +((float)scaleParprob_x[1] -((float)scaleParprob_x[0]))*
914               ((float)prob_x[i][j]-(float)low_train[j])/((float)high_train[j]-(float)low_train[j]));
915             }
916         }
917       }
918       printscaledtrainValues();
919     }
920 }
```

Figura 113. Función *scaleValues()*

Las funciones *printmaxtrainValues()* y *printscaledtrainValues()* de la Figura *114*, llamadas en la función *scaleValues(),* son las encargadas de imprimir los datos los necesarios para la generación de la información contenida inicialmente en el fichero *range*, y los datos escalados, respectivamente.

```
923 void printminmaxtrainValues(void) {
924     Serial.print(scaleParprob_x[0]);
925     Serial.print(" ");
926     Serial.println(scaleParprob_x[1]);
927     for (int j = 0; j < (NUMBER_SENSORS + NUMBER_ACC + NUMBER_MMACC); j++)
928     {
929       Serial.print(j+1);
930       Serial.print(" ");
931       Serial.print(low_train[j]);
932       Serial.print(" ");
933       Serial.println(high_train[j]);
934     }
935     Serial.println();
936 }
937
938
939 void printscaledtrainValues(void) {
940     for (int i = 0; i < n_train_vectors; i++)
941     {
942       Serial.print((int) prob_y[i]);
943       Serial.print(" ");
944       for (int j = 0; j < (NUMBER_SENSORS + NUMBER_ACC + NUMBER_MMACC); j++)
945       {
946          Serial.print(j+1);
947          Serial.print(":");
948          Serial.print(scaledprob_x[i][j], 6);
949          Serial.print(" ");
950       }
951       Serial.println();
952     }
953 }
```

Figura 114. Funciones *printmaxtrainValues()* y *printscaledtrainValues()*

Finalmente, la opción 4 activa las funciones de predicción. En la Figura *115* se muestra el diagrama de flujo asociado al código de la opción 4. El mecanismo para la predicción de signos es similar al seguido en la opción 1: en primer lugar, se presiona el Botón 1 para iniciar la recogida de muestras, aunque en este caso, antes de recoger las muestras es necesario asegurar que el usuario está preparado para realizar el signo, por lo que la función *sampleHandMov()* detecta cuándo está la mano quieta. Seguidamente, se encenderá el led azul *(LEDPIN)* durante 1,25 segundos con el fin de mostrar el tiempo disponible para representar el símbolo que se desea identificar. Una vez pasado ese tiempo, con los datos obtenidos se realizan los cálculos necesarios a partir de la información proporcionada por los diferentes sensores integrados en la plataforma HW/SW y se envían a la función de predicción, la cual, mediante un proceso de votación, es capaz de identificar la letra asociada al signo interpretado.

Finalmente, se transmite la letra identificada mediante comunicación BLE. Mientras no se vuelva a pulsar el Botón 1, este mecanismo se repite cada vez que se detecte la mano quieta.

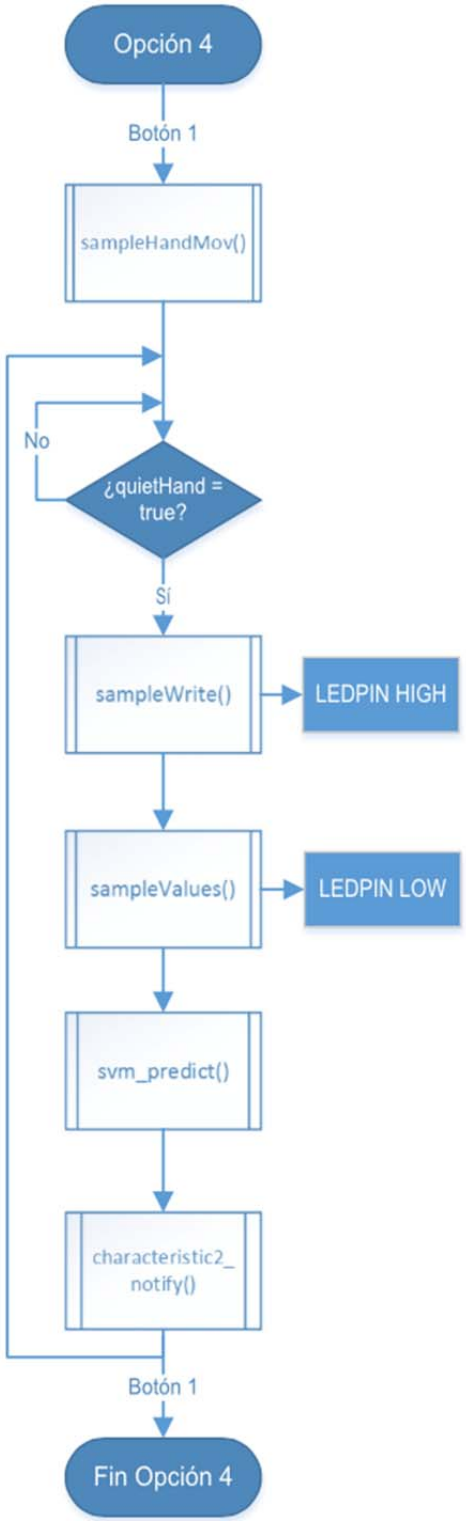


Figura 115. Diagrama de flujo de la opción 4 de la plataforma final

En la Figura *116* se muestra la sección de código modificada de la función *svm_predict()* vista en el apartado 3.3.1. La modificación introducida en el código se debe a que, en esta última versión, en lugar de imprimir el número asociado a la letra reconocida, se imprime directamente la letra. Para ello se utiliza la variable *label,* un *array* que contiene todas las letras del alfabeto español en código ASCII.

```
1225    // output of the predicted class.
1226    Serial.print("> recognized letter = ");
1227     if ((label[recognizedClass] >= 'a') && (label[recognizedClass] <= 'z')){
1228      Serial.println((char) label[recognizedClass]);
1229      caracter[0] = '\0';
1230      caracter[1] = (char) label[recognizedClass];
1231      caracter[2] = ' ';
1232      delay(500);
1233        for(int q = 0; q < NR_CLASS; q++){
1234            result[q]=0;
1235      }
1236    } else if (label[recognizedClass] == 0xC3)  {
1237      Serial.println("ñ");
1238      caracter[0] = '\0';
1239      caracter[1] = 0xB1;
1240      caracter[2] = 0xC3;
1241      delay(500);
1242      for(int q = 0; q < NR_CLASS; q++){
1243            result[q]=0;
1244      }
1245    } else if (label[recognizedClass] == ((int) ('c')) + ((int) ('h'))) {
1246      Serial.println("ch");
1247      caracter[0] = '\0';
1248      caracter[1] = 'c';
1249      caracter[2] = 'h';
1250      delay(500);
1251      for(int q = 0; q < NR_CLASS; q++){
1252            result[q]=0;
1253      }
1254    } else if (label[recognizedClass] == ((int) ('l')) + ((int) ('l')))   {
1255      Serial.println("ll");
1256      caracter[0] = '\0';
1257      caracter[1] = 'l';
1258      caracter[2] = 'l';
1259      delay(500);
1260      for(int q = 0; q < NR_CLASS; q++){
1261        result[q]=0;
1262    }
1263    } else if (label[recognizedClass] == ((int) ('r')) + ((int) ('r')))   {
1264      Serial.println("rr");
1265      caracter[0] = '\0';
1266      caracter[1] = 'r';
1267      caracter[2] = 'r';
1268      delay(500);
1269      for(int q = 0; q < NR_CLASS; q++){
1270        result[q]=0;
1271    }
1272   }
1273  }
```

Figura 116. Sección de código añadida a la función *svm_predict()*

5.3 Validación funcional de la plataforma HW/SW final

La validación funcional de la plataforma HW/SW final desarrollada en el presente TFG se divide en dos partes. En primer lugar, se realiza una validación sin transmitir los datos vía BLE, y seguidamente se añade esta funcionalidad. Para comprobar el correcto funcionamiento de la plataforma HW/SW final desarrollada en el presente TFG se siguen los pasos siguientes:

1. Se carga el *firmware* desarrollado en el dispositivo *RedBear Duo*. Para ello, habrá que conectar el dispositivo a un PC, momento en el que el LED RGB empezará a parpadear en verde hasta conectarse a la red WiFi, que pasará a parpadear más lentamente en color celeste. A continuación, desde el *IDE Particle Build* en la web de Particle, se selecciona el dispositivo *RedBear Duo* asociado a la cuenta de usuario y se carga el fichero `svm-train-predict-ble.ino` desarrollado, utilizando el botón *Flash*, tal como se detalla en el apartado 2.3.1. En ese momento el LED RGB comenzará a parpadear rápidamente en magenta.
2. Se abre el terminal para la comunicación serie USB con el PC. En el presente TFG se ha utilizado la aplicación *PuTTy* como terminal serie. Una vez abierta la aplicación, se selecciona el puerto COM correspondiente, y entre tanto, el LED RGB habrá pasado a parpadear en color verde. Al conectarse a la red WiFi, y una vez seleccionado el puerto COM correspondiente, aparecerá el menú diseñado, tal como se muestra en la Figura *117*.

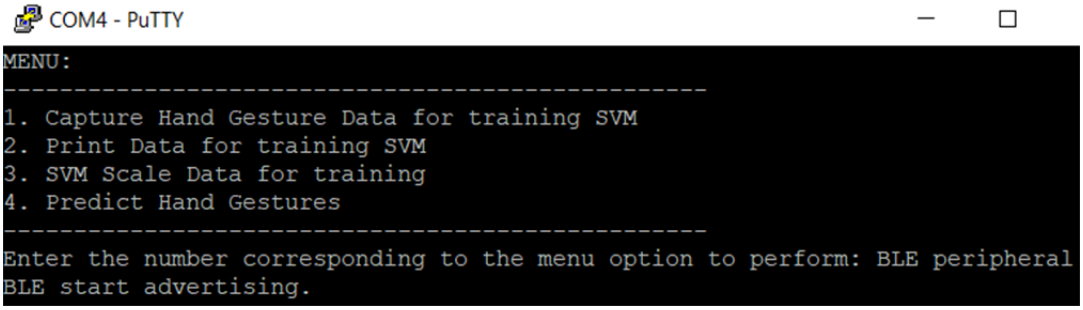


Figura 117. Menú plataforma inicial en el terminal

3. Se selecciona la opción 1 para realizar el entrenamiento. En la Figura *118* puede verse que se toman seis muestras de cada letra, donde los valores de 1, 2, 3, 4 y 5 se corresponden con los datos medidos en cada una de las resistencias flexibles, 6 y 7 son las medidas de *pitch* y *roll*, y 8, 9 y 10 los valores de la desviación típica de cada uno de los tres ejes del acelerómetro x/y/z.

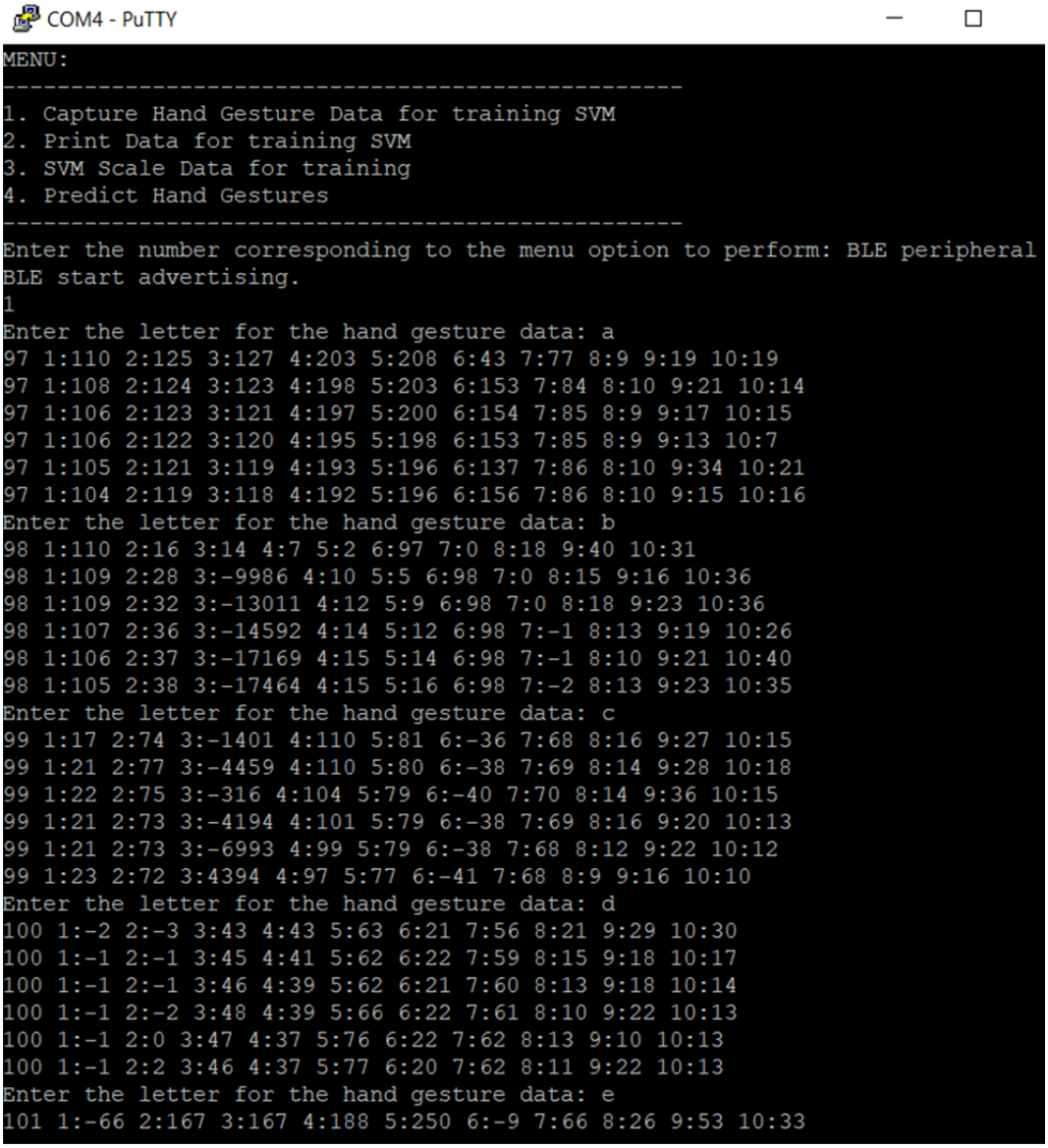


Figura 118. Datos obtenidos en el terminal con la opción “1”

4. Se selecciona la opción 2 para recoger los datos de todas las letras entrenadas, tal como se ilustra en la Figura *119*.

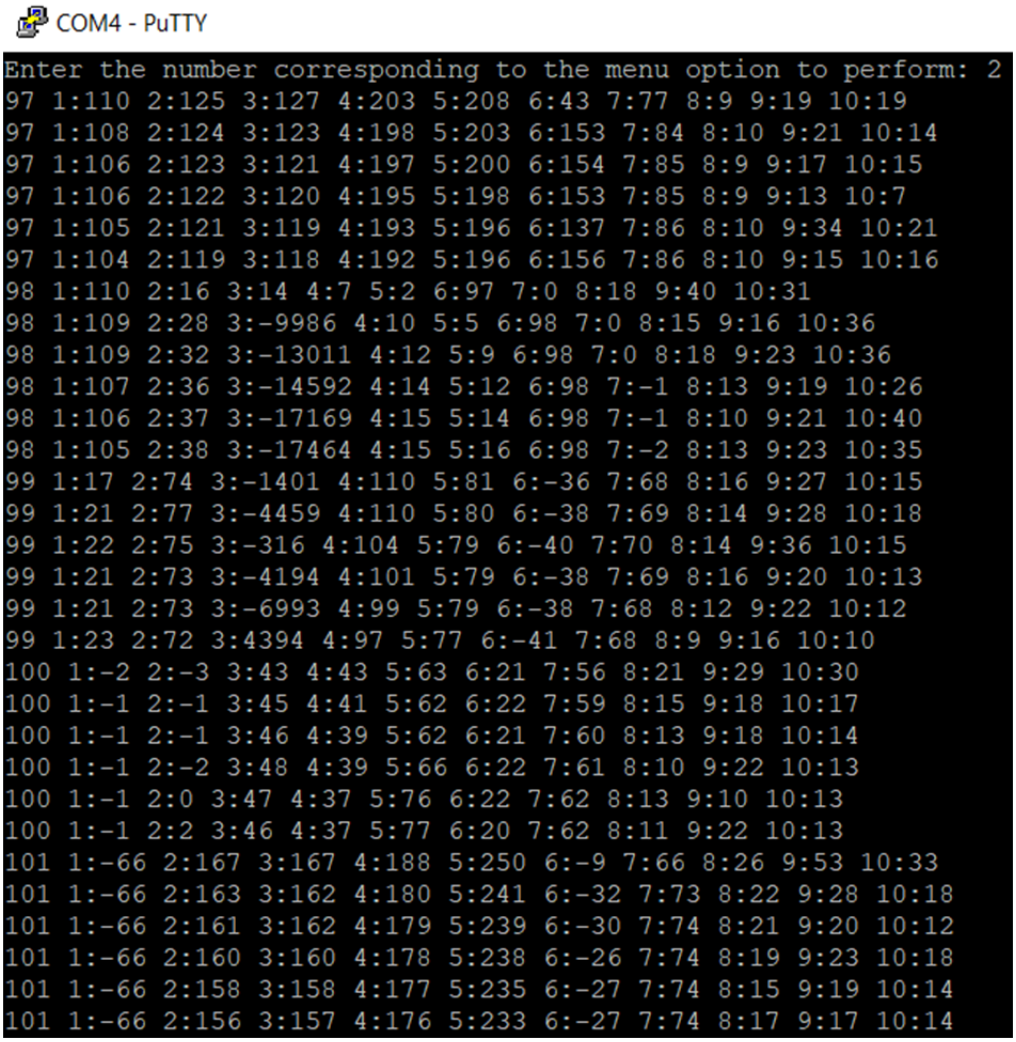


Figura 119. Datos obtenidos en el terminal con la opción "2"

5. Se selecciona la opción “3” para obtener los valores máximos y mínimos de cada sensor, y a partir de ellos, los valores máximos y mínimos para cada uno de los 10 parámetros obtenidos a partir de los sensores durante la fase de entrenamiento, y los datos escalados, tal como se muestra en la Figura 120

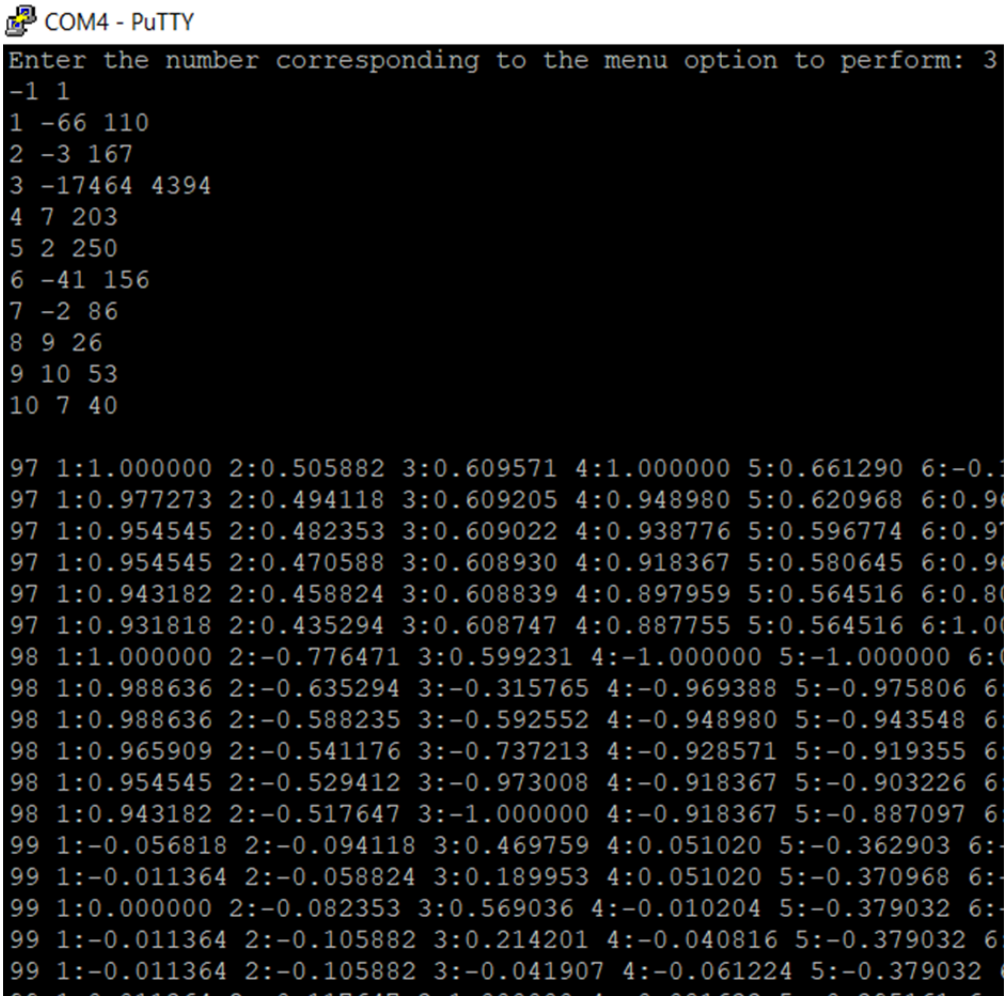


Figura 120. Datos obtenidos en el terminal con la opción "3"

6. A partir de los datos escalados se crea el fichero *range* y, utilizando el comando *svm-train* de LIBSVM en una ventana de comandos en el PC, el modelo de clasificación.
7. Se genera un *sketch* `.ino` con los vectores de soporte y se copian en el fichero `svm-train-predict-ble.ino`. Esto se consigue utilizando la herramienta *Arduino_SVM* desarrollada en Java, y los ficheros *range* y *model* generados en el paso anterior, tal como se vio en el apartado 3.4.
8. Una vez hecho esto, se actualiza el *firmware* desde el *IDE Particle Build*, de la misma manera que en el paso 1 y se repite el paso 2.
9. Por último, se selecciona la opción 4 para realizar la predicción de clases, proceso que se inicia al pulsar el Botón 1. Así, interpretando algunos signos con

el guante diseñado, se obtienen los resultados mostrados en la Figura *121* por el terminal.

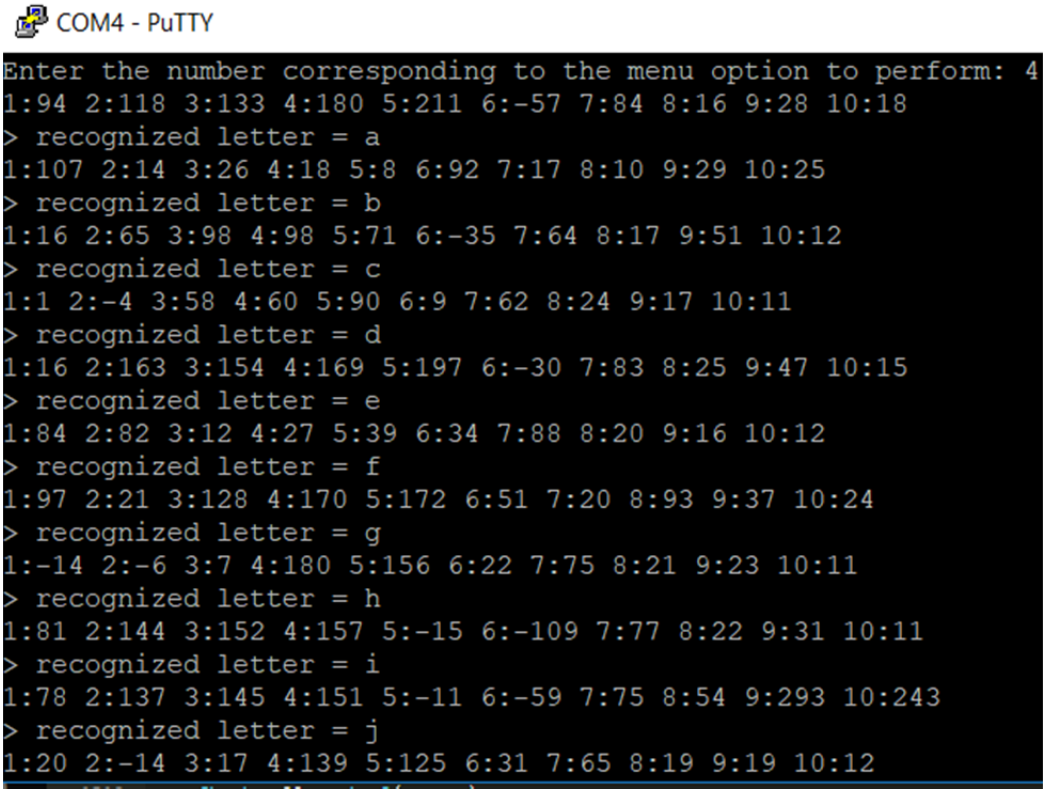


Figura 121. Datos obtenidos en el terminal con la opción "4"

A continuación, se realiza la validación con comunicación BLE. Para ello, se añade la batería recargable a la plataforma para comprobar que, una vez realizado el entrenamiento, y copiados los vectores de soporte en el código, es posible prescindir del PC para la ejecución de la opción 4. Así, partiendo del punto 7 y una vez actualizado el *firmware*, se siguen los siguientes pasos:

1. Se abre la aplicación móvil *BLE Scanner* y se inicia la conexión con el dispositivo *RedBear Duo*, tal como se explicó en el apartado 4.5.2.
2. Se pulsa el Botón 2 añadido a la plataforma HW/SW final para acceder a las funciones correspondientes a la opción 4, y seguidamente se acciona el Botón 1 para iniciar el proceso de predicción. En la Figura 122 se muestran algunos de los resultados obtenidos en el dispositivo móvil durante la validación. En este caso, se trata del reconocimiento de los gestos asociados a las letras “L”, “E” y “Q”.

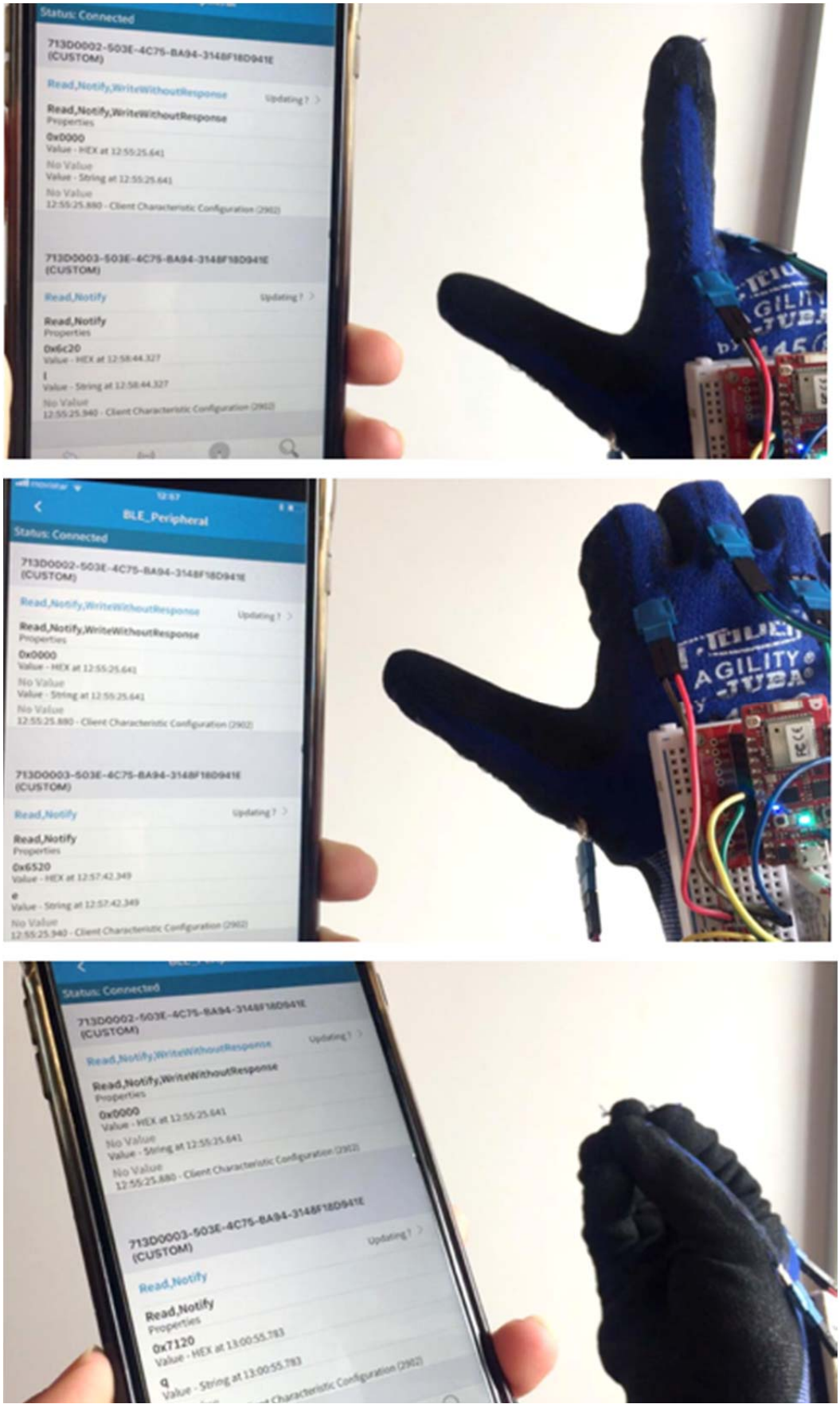


Figura 122. Validación de la plataforma final (I)

Cabe destacar que la funcionalidad de la versión final de la plataforma HW/SW fue validada por la Técnico en Interpretación de la Lengua de Signos de la ULPGC, obteniendo un reconocimiento satisfactorio para cada uno de los 30 símbolos del alfabeto dactilológico de la Lengua de Signos Española. En la Figura *123* y en la Figura *124* se recogen algunas fotografías tomadas durante la sesión en la que se realizó este proceso de validación.

Figura 123. Validación de la plataforma final (II)

Figura 124. Validación de la plataforma final (III)

Capítulo 6. Conclusiones y líneas futuras

6.1 Conclusiones

Tras finalizar el proceso de validación funcional, se puede concluir que con el presente Trabajo Fin de Grado se ha logrado el objetivo principal inicialmente establecido: el desarrollo de una plataforma HW/SW capaz de interpretar los signos asociados al alfabeto dactilológico de la Lengua de Signos Española (LSE) y transmitirlos mediante conexión BLE a un dispositivo móvil.

Para ello, se ha realizado un estudio de los 30 signos que componen el alfabeto dactilológico de la LSE, a partir del cual se han determinado los recursos HW necesarios para poder identificarlos correctamente. Tras ese análisis inicial, se integraron sensores flexibles para detectar la posición de cada uno de los dedos de la mano, un acelerómetro que permitiese identificar el movimiento de la mano en los diferentes ejes, y un dispositivo capaz de procesar localmente los datos recogidos de

los sensores, que en el caso concreto de este TFG, en la fase inicial de desarrollo fue el dispositivo *Photon* de la empresa Particle. Por otro lado, se ha adaptado la librería LIBSVM con el fin de poder ejecutar las funciones de escalado y clasificación en el dispositivo *Photon*. Finalmente, se añade la posibilidad de transmitir las letras identificadas vía BLE desde la plataforma HW/SW desarrollada, a un dispositivo móvil, para lo cual se integró el dispositivo *RedBear Duo*.

Además, se ha conseguido realizar un diseño más compacto y con un elevado grado de autonomía que permite a la plataforma HW/SW, una vez completado el proceso de entrenamiento, no depender de un PC para la fase de reconocimiento de símbolos, lo cual representa una novedad frente a otras propuestas.

Por otro lado, atendiendo a la experiencia adquirida durante la realización de este TFG, se pueden establecer las siguientes conclusiones:

- La documentación relativa al dispositivo *Photon* está disponible en la web y ha sido de gran ayuda durante el desarrollo de este TFG. De la misma manera, los foros de la comunidad de Particle son de gran utilidad, y han permitido resolver con eficacia muchas de las dudas que surgieron.
- El entorno de desarrollo *IDE Particle Build* es muy sencillo e intuitivo. Sin embargo, tiene funciones que son poco prácticas, como es el caso de la opción de búsqueda. Es por ello que para desarrollar el código es preferible usar un editor de texto libre como *Notepad++*, y luego copiar el código en el IDE.
- Además, cabe destacar la importancia de haber organizado el TFG en módulos, pues es lo que ha permitido progresar sin arrastrar errores durante su desarrollo.
- Un aspecto a tener muy en cuenta es que, a la hora de realizar el entrenamiento, es fundamental ejecutar el movimiento completo del signo en el tiempo que el LED azul permanece encendido, pues el no hacerlo correctamente puede provocar errores en el momento de reconocer letras de movimientos prolongados, como es el caso de la “Z”.
- Al tratarse de una plataforma de código abierto, es posible modificar el código para añadir funcionalidades adicionales.

6.2 Líneas futuras

Tras alcanzar todos los objetivos propuestos, se da por finalizado el presente Trabajo Fin de Grado. Partiendo del mismo, se proponen las siguientes líneas futuras:

- Integrar la función asociada a la generación de los vectores soporte, que actualmente se debe realizar a partir de la función *svm_train* de la librería LIBSVM en un PC, con el fin de que la plataforma HW/SW sea completamente autónoma en la realización de todos los procesos asociados al entrenamiento y la clasificación de las letras del alfabeto dactilológico de la LSE.
- Realizar las modificaciones necesarias para poder realizar el reconocimiento y transmisión de palabras asociadas a la Lengua de Signos Española, y no únicamente a las letras del alfabeto dactilológico.
- Incluir la posibilidad de determinar el principio y el fin de una palabra con el fin de transmitir de manera conjunta las letras del alfabeto dactilológico que la forman.

Bibliografía

[1] *OMS | Informe mundial sobre la discapacidad*. [Online]. Available: http://www.who.int/disabilities/world_report/2011/es/. [Accessed: Dec 2017]

[2] *OMS | Sordera y pérdida de la audición*. [Online]. Available: http://www.who.int/mediacentre/factsheets/fs300/es/.[Accessed: Dec 2017]

[3] Ministerio de Educación y Ciencia, "Población con discapacidad auditiva." "Available: http://ares.cnice.mec.es/informes/17/contenido/19.htm [Accessed: Dec 2017]

[4] M. C. Domingo, "An oweview of IoT for people with disabilities," *Journal of Network and Computer Applications*, pp. 584-596, 2011.

[5] P. Kumari, P. Goel and S. R. N. Reddy, "PiCam: IoT based wireless alert system for deaf and hard of hearing," in 2015, pp. 39-44.

[6] *Traductor de lengua de signos a voz en tiempo real - Showleap*. [Online] Available: http://www.showleap.com. [Accessed: Dec 2017]

[7] T. Starner, J. Weaver and A. Pentland, "Real-time American sign language recognition using desk and wearable computer based video," *Tpami,* vol. 20, no. 12, pp. 1371-1375, 1998. [Online] Available: DOI: 10.1109/34.735811.

[8] "Un guante inteligente que traduce el lenguaje de signos a texto y audio." 2015. Available: https://descubrearduino.com/un-guante-inteligente-que-traduce-el-lenguaje- de-signos-a-texto-y-audio/. [Accessed: Jan 2018]

[9] "Guante traductor de lengua de señas," 2015. Available: http://neuropsicologia.saludyeducacionintegral.com/guante-traductor-de-lenguaje-sordomudo/. [Accessed: May 2018]

[10] *Particle*. Available: https://docs.particle.io/datasheets/photon-(wifi)/photon-datasheet/. [Accessed: Jan 2018]

[11] *RedBear Duo*. Available: https://redbear.cc/product/wifi-ble/redbear-duo.html. [Accessed: Jan 2018]

[12] *Flex Sensor 2.2"*. Available: https://www.sparkfun.com/products/10264. [Accessed: Jan 2018]

[13] “Confederación Estatal de Personas Sordas," 2013. Available: http://www.cnse.es/lengua.php. [Accessed: Jun 2018]

[14] *El alfabeto dactilológico*. Available: http://lsefacil.usefedora.com/courses/3640/lectures/71282. [Accessed: Jun 2018]

[15] *Photon Datasheet*. Available: https://docs.particle.io/datasheets/photon-(wifi)/photon-datasheet/. [Accessed: May 2018]

[16] "Flex Sensor Data Sheet," *Asia-Pacific Biotech News,* vol. 22, *no.5,* pp. 18-27, 2018. DOI: 10.1142/S0219030318000356.

[17] *Flex Sensor Hookup Guide*. Available: https://learn.sparkfun.com/tutorials/flex-sensor-hookup-guide. [Accessed: May 2018]

[18] H. Barragán, «¿Que es Wiring? », *DEARQ Rev. Arquit. Univ. Los Andes*, no. 8, pp. 156–158, 2011.

[19] S. Monk, *Getting Started with the Photon: Making Things with the Affordable, Compact, Hackable WiFi Module.* San Francisco: Maker Media, 2015.

[20] *TINKERING WITH "TINKER"*. [Online]. Available: https://docs.particle.io/guide/getting-started/tinker/photon/. [Accessed: May 2018]

[21] *Particle*. [Online]. Available: https://docs.particle.io/guide/getting-started/intro/photon/. [Accessed: May 2018]

[22] *FLASH APPS WITH PARTICLE BUILD*. [Online]. Available: https://docs.particle.io/guide/getting-started/build/photon/. [Accessed: May 2018]

[23] *Particle CLI*. [Online]. Available: https://docs.particle.io/guide/tools-and-features/cli/photon/. [Accessed: May 2018]

[24] "SparkFun_LSM9DS1_Arduino_Library," Enero, 2017. [Online] Available: https://github.com/sparkfun/SparkFun_LSM9DS1_Arduino_Library/blob/master/examples/LSM9DS1_Basic_I2C/LSM9DS1_Basic_I2C.ino [Accessed: May 2018]

[25] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning,* vol. 20*, no.3,* pp. 273-297, 1995. DOI: 10.1007/BF00994018.

[26] E. Carmona, "Tutorial sobre Máquinas de Vectores de Soporte (SVM)," 2014.

[27] Chris Albon, "SVC Parameters When Using RBF Kernel," Dec, 2017.

[28] Chih-Wei Hsu, Chih-Chung Chang and Chih-Jen Lin, "A Practical Guide to Support Vector Classification," May, 2016.

[29] C. Chang and C. Lin, "LIBSVM," *ACM Transactions on Intelligent Systems and Technology (TIST),* vol. 2, *n.3,* pp. 1-27, 2011. Available: http://dl.acm.org/citation.cfm?id=1961199. DOI: 10.1145/1961189.1961199.

[30] C. Cufí*l*, *Getting Started with Bluetooth Low Energy.* (1st ed.) California: O'Reilly Media, 2014.

[31] Gabriel Recio. 2018. *Estudio de la tasa de transferencia del dispositivo IoT RedBear Duo en comunicaciones BLE*. Universidad de las Palmas de Gran Canaria, España.

[32] "BLE Scanner 4.0," [Online]. Available: https://itunes.apple.com/us/app/ble-scanner-4-0/id1221763603?mt=8

[33] "SparkFun Photon IMU Shield," [Online]. Available: https://www.sparkfun.com/products/13629

Pliego de Condiciones

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

PL.1 Condiciones Hardware

Durante el desarrollo de este TFG se han usado los dispositivos hardware recogidos en la Tabla 9.

Tabla 9. Relación de equipos *hardware*

Equipo	Modelo	Fabricante/Comerciante
Ordenador	CPU Intel Core i7 16 GB RAM	Dell
Particle *Photon*	Photon	Particle
RedBear Duo	RedBear Duo	RedBear
LSM9DS1	SparkFun 9DoF IMU Breakout	Sparkfun Electronics
Flex sensors	Flex Sensors 2,2”	Spectra Symbol
Photon IMU Shield	SparkFun Photon IMU Shield	Sparkfun Electronics
Batería LiPo	BET573 (3.7V / 180mAh)	Electronic NIMO

PL.2 Condiciones Software

Por otro lado, en la Tabla *10* se exponen las herramientas software utilizadas, especificando su versión.

Tabla 10. Relación de herramientas *software*

Aplicación	Versión
Sistema Operativo Windows	Microsoft Windows 10
Microsoft Office	2016
Microsoft Visio	2013

Notepad++	7.5.3
Particle Build	
Putty	0.70
Arduino_SVM	3.3.6
BLE Scanner	2.0.1

PL.3 Condiciones Firmware

Por último, en la Tabla *11* se expone el *firmware* utilizado y su versión.

Tabla 11. Relación de *firmware*

Aplicación	**Versión**
Dispositivo *Photon*	v0.7.0
Dispositivo *RedBear Duo*	v0.3.1
Librería *SparkIntervalTimer*	v1.2.8
Librería *SparkFunLSM9DS1*	V1.1.3

Presupuesto

Este capítulo contiene el presupuesto que recoge los gastos generados en la realización del presente Trabajo Fin de Grado. Dicho presupuesto se divide en las siguientes partes:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material, 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.

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

P.1 Trabajo tarifado por tiempo empleado

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

Según la tabla retributiva de personal contratado en proyectos de investigación elaborada por la ULPGC en el año 2016, este salario, con una dedicación de 20 horas semanales, asciende a 896,31€ mensuales. Este TFG se ha desarrollado a lo largo de cuatro meses de trabajo, por lo que se calcula el coste total por tiempo empleado en:

$$896{,}31 \cdot 4 = 3.585{,}24\ €$$

Por lo tanto, el trabajo tarifado por tiempo empleado asciende a la cantidad de *tres mil quinientos ochenta y cinco euros con veinticuatro céntimos.*

P.2 Amortización del inmovilizado material

En el inmovilizado material se consideran tanto los recursos hardware como software empleados para la realización de este TFG.

Para estipular el coste de amortización en un periodo de 3 años se utiliza un sistema de amortización lineal, en el que se supone que el inmovilizado material se deprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula de la siguiente forma:

$$Cuota\ anual = \frac{Valor\ de\ adquisición - Valor\ residual}{Número\ de\ años\ de\ vida\ útil}$$

donde el valor residual se corresponde con el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil.

P.2.1 Amortización del material hardware

Debido a que la duración de este Trabajo Fin de Grado es de 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 *12* se especifica el hardware amortizable necesario para la realización del trabajo, indicando su valor de adquisición y su amortización, teniendo en cuenta un tiempo de uso de 4 meses.

Tabla 12. Costes y amortización del *hardware (I)*

Elemento	Valor de adquisición	Amortización
Ordenador portátil Dell Inspiron	1120,00 €	123,00 €
Total	**1120,00 €**	**123,00 €**

Por otro lado, en la Tabla *13* se muestra el resto de material hardware utilizado y por el que, debido a su bajo precio, su amortización coincide con su valor de adquisición.

Tabla 13. Costes y amortización del *hardware* (II)

Elemento	Valor de adquisición	Amortización
Photon	16,19 €	16,19 €
RedBear Duo	19,31 €	19,31 €
LSM9DS1 Breakout	12,75 €	12,75 €
Photon IMU Shield	21,41 €	21,41 €
Flex sensor 2.2” (x5)	34,12 €	34,12 €
Batería LiPo	8,10 €	8,10 €

Protoboard	7,09 €	7,09 €
Total	**103,78€**	**118,97 €**

Finalmente, tras realizar la suma de ambos se obtiene el coste total del material hardware, tal y como se muestra en la Tabla *14*.

Tabla 14. Costes y amortizaciones totales del *hardware*

Elemento	Coste
Costes y amortización hardware (I)	123,00 €
Costes y amortización hardware (II)	118,97 €
Total	**241,97 €**

El coste total del material hardware asciende a *doscientos cuarenta y un euros con noventa y siete céntimos.*

P.2.2 Amortización del 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 *15* muestra los elementos software necesarios para la realización del trabajo, así como su valor de adquisición y su amortización.

Tabla 15. Costes y amortización del *software*

Elemento	Valor de adquisición	Amortización
Sistema Operativo Windows	0,00 €	0,00 €

Microsoft Office	0,00 € (*)	0,00 € (*)
Notepad++	0,00 € (**)	0,00 € (**)
Particle Build	0,00 € (**)	0,00 € (**)
Putty	0,00 € (**)	0,00 € (**)
Arduino_SVM	0,00 € (**)	0,00 € (**)
BLE Scanner	0,00 € (**)	0,00 € (**)
Total	**0,00 €**	**0,00 €**

(*) Licencia de uso proporcionada por la ULPGC.

(**) Software libre.

Por tanto, el coste total del material software es de *cero euros.*

P. 3 Redacción del trabajo

Se ha utilizado (7) para determinar el coste asociado a la redacción de la memoria del presente Trabajo Fin de Grado.

$$R = 0{,}07 \times P \times C_n, \quad (7)$$

donde:

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

El valor del presupuesto P se calcula sumando los costes del trabajo tarifado por tiempo empleado y de la amortización del inmovilizado material, tanto hardware como software. El resultado de los costes se muestra en la Tabla *16*.

Tabla 16. Presupuesto, incluyendo trabajo tarifado y amortización del material

Concepto	Coste
Trabajo tarifado por tiempo empleado	3.585,24 €
Amortización del material hardware	226,78 €
Amortización del software	0,00 €
Total	**3.812,02 €**

Como el coeficiente de ponderación *Cn* para presupuestos menores de 30.050,00€ viene definido por el COITT con un valor de 1.00, el coste derivado de la redacción del Trabajo Fin de Grado es de:

$$R = 0{,}07 \times 3.812{,}02\ € \times 1 = 266{,}84\ €$$

Ascendiendo de esta forma el coste de la redacción del trabajo a *doscientos sesenta y seis euros con ochenta y cuatro céntimos.*

P. 4 Derechos de visado del COITT

El COITT establece que, para proyectos técnicos de carácter general, los derechos de visado para 2016 se calculan en base a (8).

$$V = 0{,}006 \times P_1 \times C_1 + 0{,}003 \times P_2 \times C_2, \quad (8)$$

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 P2.

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 Tabla *17*. Al igual que en el caso anterior, el coeficiente C_1 para proyectos de presupuesto inferior a 30.050,00€ es de 1,00, asimismo el valor de P_2 es de 0,00€ ya que no se realiza ninguna obra.

Tabla 17. Presupuesto, incluyendo trabajo tarifado, amortización y redacción del trabajo

Concepto	Coste
Trabajo tarifado por tiempo empleado	3.585,24 €
Amortización del material hardware	241,97 €
Amortización del software	0,00 €
Redacción del trabajo	266,84 €
Total	**4.094,05 €**

De esta forma, aplicando a (8) los datos de la Tabla 17 y el coeficiente especificado se obtiene:

$$V = 0{,}006 \times 4094{,}05\,€ \times 1 = 24{,}56\,€$$

Los costes por derechos de visado del presupuesto ascienden a *veinticuatro euros con cincuenta y seis céntimos.*

P. 5 Gastos de tramitación y envío

Los gastos de tramitación y envió están estipulados en seis euros (6,00€) por cada documento visado de forma telemática.

P. 6 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 *18* se muestran los costes derivados de estos recursos.

Tabla 18. Costes de material fungible

Concepto	Coste
Folios	10,00 €
Guante	4,00 €
Tóner de la impresora	30,00 €
Encuadernado	4,00€
Tres CDs	6,00€
Total	**54,00 €**

Los costes de material fungible ascienden a *cincuenta y cuatro euros.*

P. 7 Aplicación de impuestos y coste total

La realización del presente TFG está gravada por el Impuesto General Indirecto Canario (IGIC) en un siete por ciento (7 %). En la Tabla *19* se muestra el presupuesto final con los impuestos aplicados.

Tabla 19. Presupuesto total del Trabajo Fin de Grado

Concepto	Coste

Trabajo tarifado por tiempo empleado	3.585,24 €
Amortización del material hardware	241,97 €
Amortización del software	0,00 €
Redacción del trabajo	266,84 €
Derechos de visado del COIT	24,56 €
Gastos de tramitación y envío	6,00 €
Costes de material fungible	54 €
Total (Sin IGIC)	**4.178,61 €**
IGIC (7%)	292,50 €
Total	**4.471,11 €**

El presupuesto total del Trabajo Fin de Grado “Plataforma para la interpretación del alfabeto dactilológico de la Lengua de Signos Española basada en dispositivos IoT” asciende a *cuatro mil cuatrocientos setenta y un euros con once céntimos.*

Fdo.: D. Claudia R. Rivero Santana

En Las Palmas de Gran Canaria a 8 de junio de 2018

Anexo

Adjunto a la memoria del presente Trabajo Fin de Grado se encuentra disponible un *Compact Disc Read-Only Memory* (CD-ROM). En este anexo se describe la estructura de su contenido a partir de la lista que se expone a continuación.

- En el directorio `Memoria del TFG` se encuentra la memoria del TFG *"Plataforma para la interpretación del alfabeto dactilológico de la Lengua de Signos Española basada en dispositivos IoT"* en formato PDF.
- En el directorio `Vídeo` se encuentra un vídeo en el que se demuestra el funcionamiento de la plataforma HW/SW desarrollada.