Universitat Oberta de Catalunya

uoc.edu

Implementación Low Cost de un Sistema de Comunicación BLE

Carmen Llamas Bellido
Máster en Ingeniería de Telecomunicación
Área de Electrónica

Aleix López Martín

Junio 2022

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Implementación Low Cost de un Sistema de Comunicación BLE*
Nombre del autor:	*Carmen Llamas Bellido*
Nombre del consultor/a:	*Aleix López Martín*
Nombre del PRA:	*Carlos Monzo Sánchez*
Fecha de entrega (mm/aaaa):	06/2022
Titulación:	*Máster en Ingeniería de Telecomunicación*
Área del Trabajo Final:	*Electrónica*
Idioma del trabajo:	*Castellano*
Palabras clave	*BLE, nRF52, Thingy:52, Sniffer, Arduino, UART*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Bluetooth ha sido una tecnología muy utilizada para la transferencia de información. Sin embargo, con el auge de los dispositivos IoT como wearables se puso de manifiesto la necesidad de perfeccionar esta tecnología. Esta mejora estuvo enfocada al bajo consumo y potencia, y tuvo como resultado el Bluetooth Low Energy.

Este proyecto se centra en la implementación de dos subsistemas de comunicación BLE. Cada subsistema se centrará en un rol del SoC BLE (central/periférico) que reciba/envíe datos mediante BLE.

El primer subsistema adquiere los datos de un sensor de nivel de agua en un móvil. El sensor es el encargado de enviar los datos captados hacia el microcontrolador el cual los envía hacia el SoC y este mediante BLE al teléfono móvil.

El segundo adquiere los datos de un sensor en un ordenador. El sensor está integrado en una plataforma junto con el SoC BLE (que actuará como periférico) que enviará los datos hacia otro SoC BLE el cual mostrará los datos en el ordenador.

Primero se ha llevado a cabo un estudio de diferentes dispositivos a usar para cada uno de los subsistemas eligiendo el óptimo en cada caso. Posteriormente

se realizado el diseño hardware y la programación firmware.

Obtendremos dos prototipos sobre los que se realizarán las pruebas pertinentes que permitan corroborar su funcionamiento correcto y que se ha logrado un diseño de bajo consumo, potencia y coste.

Las conclusiones versarán sobre los resultados obtenidos y las líneas futuras que optimizarían el diseño realizado.

Abstract (in English, 250 words or less):

Bluetooth has been a widely used technology for information transfer. However, with the rise of IoT devices such as wearables, the need to improve this technology became apparent. This improvement was focused on low power consumption and low power, and resulted in Bluetooth Low Energy.

This project focuses on the implementation of two BLE communication subsystems. Each subsystem will focus on one role of the BLE SoC (central/peripheral) that receives/sends data via BLE.

The first subsystem acquires data from a water level sensor in a mobile phone. The sensor sends the captured data to the microcontroller, which sends it to the SoC and the SoC sends it via BLE to the mobile phone.

The second one acquires the data from a sensor in a computer. The sensor is integrated in a platform together with the BLE SoC (which will act as a peripheral) which will send the data to another BLE SoC which will display the data on the computer.

First, a study of different devices to be used for each of the subsystems has been carried out, choosing the optimal one in each case. Subsequently, the hardware design and firmware programming were carried out.

We will obtain two prototypes on which the relevant tests will be carried out to corroborate that they work correctly and that a low consumption, power and cost design has been achieved.

The conclusions will be based on the results obtained and the future lines that would optimise the design carried out

Índice

Lista de figuras

Lista de tablas

1. Introducción

1.1 Contexto y justificación del Trabajo

Actualmente Bluetooth es una tecnología muy útil para la transferencia de datos que se encuentra en varios dispositivos y es popular para todo. Su uso es muy variado y se podría decir que no tiene límites en cuanto al campo de aplicación.

Sin embargo, Bluetooth como tal era especialmente problemático para los dispositivos con una potencia limitada, como los dispositivos IoT. Bluetooth Low Energy BLE se diseñó para solucionar este problema ya que posee muchas características que hicieron triunfar a Bluetooth, pero centrándose en el bajo consumo.

BLE ha hecho posible que una amplia gama de pequeños dispositivos IoT, como sensores y *tags*, se comuniquen a pesar de no tener grandes baterías ya que solo necesitan una transferencia pequeña de datos.

Se podría decir que BLE hizo que salieran a la luz muchos de los *wearables*, *tags* y otros dispositivos inteligentes que utilizamos hoy en día ya que su bajo consumo, coste y potencia hizo posible que se pudieran incorporar en cualquier dispositivo, sobre todo en los *wearables* tan comunes hoy en día.

Es de gran interés por tanto conocer cómo funciona esta tecnología de comunicación, sus ventajas e inconvenientes y su evolución. Pero no solo es interesante la teoría puesto que este trabajo tiene como uno de los objetivos la implementación de un sistema de bajo coste basado en comunicación BLE pudiendo ver así de primera mano su funcionamiento.

1.2 Objetivos del Trabajo

Los objetivos que se pretenden alcanzar a través de este trabajo son:

- Alcanzar una visión completa sobre BLE: evolución, características, protocolos.

- Estudiar diferentes integrados que utilicen BLE.

- Desarrollar una interfaz para interconectar diferentes dispositivos mediante BLE y UART.

- Implementar un prototipo donde el SoC actúe como central para recibir datos mediante BLE

- Implementar un prototipo donde el SoC actúe como periférico para enviar datos mediante BLE

- Realizar pruebas sobre los sistemas verificando su comportamiento.

1.3 Estructura del Trabajo Fin de Máster

Este documento consta de 10 capítulos donde se abarcan diversos puntos que se han realizado para realizar el trabajo completo.

El primer capítulo pretende contextualizar BLE y porqué este TFM gira en torno a esa tecnología. También se expone la motivación que hizo que este proyecto se llevara a cabo y los objetivos que se definieron y se han cumplido a lo largo del desarrollo del proyecto.

El segundo capítulo trata sobre el estado del arte de BLE. Se hace un repaso por la evolución que ha sufrido, desde Bluetooth 4.0 hasta Bluetooth 5.2 y los cambios que se fueron incorporando para mejorar BLE. También se detallan las características de BLE (muchas compartidas con Bluetooth) y la pila de protocolos en la que basa su tecnología. Por último, se exponen las ventajas y desventajas de BLE y algunos campos de aplicación puesto que BLE se utiliza en diversos campos y dispositivos.

El tercer capítulo busca dar una pequeña introducción a los dos subsistemas que se quieren implementar sin entrar en detalle del *hardware* ni del *software*, ya que esto se detalla más adelante.

El cuarto capítulo versa sobre la estructura hardware. Contextualiza los dispositivos a utilizar dando una pequeña visión del mercado, de las características del propio dispositivo y de la función a desempeñar en el sistema. También se añaden otros dispositivos que se consideraron, pero se descartaron para el proyecto.

El quinto capítulo muestra la estructura software del sistema, empezando por una breve y concisa explicación de la arquitectura software genérica del SoC elegido. Posteriormente se detallan y se da una explicación de los códigos implementados. También este punto incluye los diferentes programas complementarios que se han usado para la realización completa del prototipo. Este capítulo termina con unos diagramas para clarificar más el código utilizado.

Finalmente, en el capítulo seis se realizan las pruebas sobre cada subsistema. Se detallan los pasos seguidos, los intercambios de mensajes observados y los datos recibidos por el “sistema final” tras las comunicaciones BLE/UART pertinentes.

Por último, en el capítulo siete se hace una recapitulación para mostrar las conclusiones halladas tras el proyecto y las líneas futuras y mejoras que se podrían realizar.

El capítulo ocho contiene un glosario de los términos usados y el capítulo nueve recoge la bibliografía en la que se ha basado el proyecto.

El capítulo 10 consta de Anexos donde entre otros, se detalla el código utilizado y los servicios BLE que son usados en un subsistema.

1.4 Motivación

Es innegable el auge del IoT y de nuevos protocolos de comunicación donde reina el bajo coste y consumo. Para muchas aplicaciones el envío de datos no ha de ser constante, ya que lo que se pretende es que la vida útil del dispositivo sea lo mayor posible.

Uno de los aspectos que hace tan interesante el BLE es que la mayoría de los dispositivos (televisiones, smartphones, *wearables*…) integran ya este estándar, esto ha propiciado que se utilice mucho en la domótica (bombillas, enchufes, cámaras…) con un coste asequible por lo que mucha parte de la población ya los integra en sus hogares.

La ingeniería de las telecomunicaciones debe estar orientada a la creación e implementación de protocolos que se tornen hacia una gestión más eficiente de la energía. Por ello, el sistema BLE despierta una motivación extra.

El hecho de poder integrar el diseño hardware y firmware en un mismo proyecto es de gran interés y también un desafío para cualquier ingeniero puesto que abarca dos campos clave de la electrónica.

Durante la realización del TFM se han tenido en cuenta los siguientes entregables mostrados en forma de un diagrama WBS

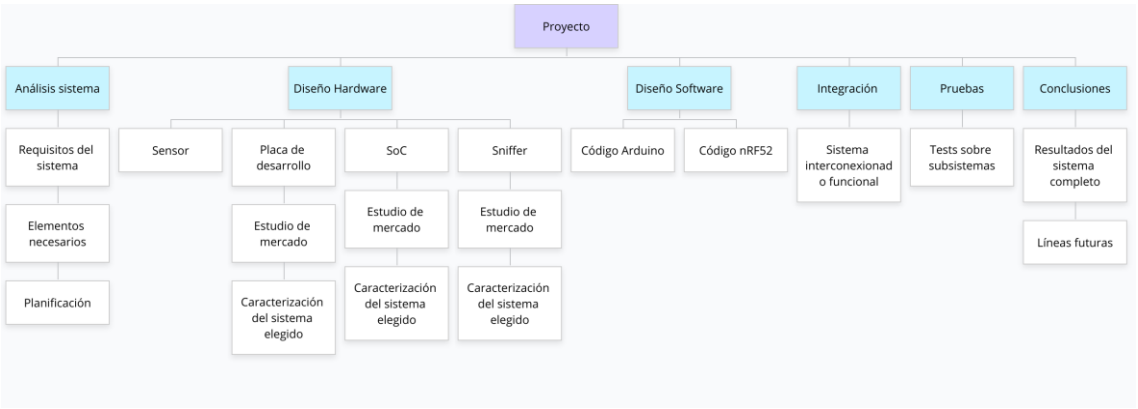

Ilustración 1: WBS del proyecto

2. Estado del arte

2.1 Introducción – ¿Cómo surge?

Bluetooth Low Energy (BLE) o Bluetooth Smart surge como un subconjunto del ya conocido por todos Bluetooth (a partir de ahora lo denominares Bluetooth clásico) que se introduce como una especificación de Bluetooth 4.0. Pero ¿cómo se ha llegado al punto de necesitar BLE y no seguir usando Bluetooth clásico?

El contenido que consumimos y enviamos ha pasado de ser simples textos, audios o imágenes hasta el envío de vídeo (que aglutina lo anterior). Pero no solo ha evolucionado el contenido transmitido, sino que los escenarios también, porque ahora las comunicaciones no son persona a persona, sino que son de persona a cosa y hasta de cosa a cosa.

Esta evolución también ha propiciado que los requisitos de la transmisión sean cada vez mayores, es decir, buscamos tecnologías que consuman poco, que sea barato y que sea una comunicación estable (fiable).

Desde los inicios siempre se ha utilizado la tecnología Bluetooth para el envío de información de manera inalámbrica. Sin embargo, el hecho de que la mayoría de los dispositivos, como auriculares, wearables… demandaran un alto consumo, era necesario encontrar una solución.

Es así como nació BLE, la cual es una tecnología Bluetooth de baja potencia que resuelve el problema del alto consumo de los dispositivos inalámbricos. Es un protocolo que fue pensado para abordar las necesidades de la eficiencia energética sin olvidarse del bajo coste. Actualmente (casi) todos los smartphones cuentan con BLE sin importar el SO que tenga, por lo que es un estándar muy usado.

BLE fue desarrollado por la compañía Nordic Semiconductor, la cual empezó el desarrollo estándar de la ULP – Ultra Low Power, sin embargo, después fue adoptado por el SIG (Special Interest Group) conformado por las compañías Ericsson, Nokia, Toshiba, IBM e Intel [1].

Desde entonces, diversas compañías se han dedicado al desarrollo de chips BLE, pero entre ellos, podemos nombrar a Texas Instrument (TI), Nordic Semiconductor y Dialog. Esta disputa de poder en el mercado de los chips BLE hace que cada vez su tecnología sea más refinada y el mercado se encuentre en una situación próspera de crecimiento.

Cuando surgió BLE, Texas Instrument (TI) lanzó los chips CC2540 y CC2541. Ambas soluciones SoC eran rentables y de bajo consumo para aplicaciones Bluetooth de bajo consumo. Cuentan con memoria flash de 128 o 256 KB (dependiendo del modelo) y combinan un excelente transceptor de RF con un

MCU 8051, memoria flash programable en el sistema, RAM de 8 KB y otros periféricos [2].

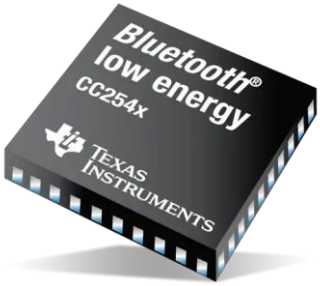

Ilustración 2: CC254x chip [2]

Por otro lado, la compañía Dialog lanzó el chip DA14580. Este SoC cuenta con un núcleo ARM Cortex-M0 de 16MHz (32bits) con 84KB de ROM y 50KM de RAM. Cuenta también con una ROM OTP de 32KB. Este SoC revolucionó el mercado de los wearables puesto que su tamaño era muy reducido y el consumo de energía era muy bajo. [3]

Ilustración 3: DA14580 chip [3]

Nordic lanzó el chip nRF51822, que utiliza el núcleo ARM Cortex M0, integra 256 KB de Flash y 16 KB de RAM. Este chip también fue muy utilizado en el mercado de las pulseras. [4]

Ilustración 4: nRF51822 chip [4]

A medida que los estándares de Bluetooth evolucionaban, también lo hacían los chips de las diferentes compañías.

2.2 Evolución/Versiones

2.2.1 Bluetooth 4.0

Esta versión sacada en 2010 es la primera en implementar la tecnología BLE, permitiendo el ahorro de consumo. Mantiene una tasa de transferencia de 24 Mb/s, aunque frente a la versión 4.1 tiene una tasa de transferencia un 15% menor. En esta versión se incluye por primera vez la banda ISM de los 2.4GHz, logrando así un alcance de 60 m y una velocidad de hasta 32 Mb/s.

2.2.2 Bluetooth 4.1

En esta versión de 2013 reconocemos una revisión de la anterior, mejora la tasa de transferencia un 15% y se centra en el Internet de las Cosas (IoT) permitiendo de este modo la conexión entre dispositivos pequeños sin necesidad de un intermediario.

Otra mejora fue la actualización de la topología de la capa de enlace, esto facilitó la coexistencia maestro-esclavo en un solo dispositivo. También permite varios esclavos y no solo uno. Cabe destacar que esta versión admite UUID de 32 bits, el cual se transporta en el paquete de transmisión. El UUID de 128 bits se puede conseguir a través de un mapeo de UUID de 32 bits de la transmisión. Se puede obtener, por tanto, la mayor longitud efectiva de datos de transmisión en el paquete de transmisión.

Otras características nuevas compatibles con esta versión incluyen:

- Tiempo parcial de descubrimiento
- Conexión L2CAP
- Topología y modo dual
- Topología de capa de enlace LE
- Escaneo completo entrelazado
- Servicios de señalización de coexistencia inalámbrica móvil

2.2.3 Bluetooth 4.2

Se lanza en 2014, la principal mejora es el modo de cifrado de emparejamiento. En las versiones anteriores este punto se basa en el cifrado AES-CCM, cuyo inconveniente provenía de compartir la misma clave entre receptor y transmisor, por tanto, existía el riesgo de ser descifrado. En esta nueva versión el emparejamiento está encriptado por un algoritmo de intercambio de claves Diffie-Hellman. Este nuevo método de cifrado consiste en que cada dispositivo dispone de dos claves, una pública y otra privada. Esta última se guarda sola mientras que la pública se revela a la otra parte.

Otra mención importante es la nueva y única dirección MAC de Bluetooth, que es propia para el dispositivo Bluetooth conocida como BD_ADDR, que desempeñará un rol en la identificación de dispositivos como el emparejamiento y la conexión. Con BD_ADDR se puede rastrear el equipo de logística, no obstante, la nueva MAC no está expuesta a la supervisión del

dispositivo maestro en algunas de las aplicaciones. Sin embargo, esta versión ofrece una opción flexible para solventar el inconveniente. El dispositivo esclavo puede optar por enviar una dirección de dispositivo aleatoria en el modo de transmisión, de manera que el dispositivo maestro pueda obtener su BD_ADDR real tras emparejarse y analizarse.

Una mejora relevante es la capacidad de transmisión de datos, pasa de 23 bytes de la versión anterior a un máximo de 255 bytes, mejorando considerablemente la tasa de transferencia de datos. Añadir que en esta versión se implementa la compatibilidad con el protocolo IPv6.

2.2.4 Bluetooth 5.0

En 2016 se lanza la nueva versión de Bluetooth 5.0, con la cual se puede alcanzar una tasa de transferencia de datos de 50Mb/s y el alcance aumenta hasta los 100 m. Esto se produce porque la nueva versión admite dos PHY, es decir, 1M PHY y 2M PHY frente a 1M PHY de las versiones anteriores, además mantiene la admisión de un solo paquete de 255 bytes de transmisión de datos. Esto se traduce en el alcance de una velocidad de datos neta de aproximadamente en 16Mb/s con sobrecarga.

Se ha de recalcar que con la ayuda de los códigos de corrección de errores de renvió y el mapeo de patrones el RX mejora significativamente la capacidad de corrección de errores de todo el paquete de carga útil en el proceso y con ello el aumento de la distancia de transmisión. El rango 4X se obtiene reduciendo la tasa de datos en 2Mb/s, 1 Mb/s, 500Kb/s y 125 kb/s, es decir, que cuanto menor sea la velocidad de datos, el rango alcanzado aumenta.

Además, en esta nueva versión se agrega una PDU extendida sobre la base de la original PDU, lo que permite que se pase de una admisión de datos de 31 bytes a 255 bytes. Se ha de tener en cuenta que en la PDU original el canal de transmisión se limita a 37, 38 y 39 canales, en la versión 5.0 los datos se envían en 37 canales que no sean 37, 38 y 39.

2.2.5 Bluetooth 5.2

Es la última versión conocida, lanzada en enero de 2020, se modifica con un LE Audio que permito que varios dispositivos compartan datos. No obstante, tiene un límite de dos dispositivos. A través de la combinación de mejoras en el perfil de atributo genérico y una versión mejorada del protocolo de atributo (A), nace el nuevo protocolo de atributos mejorados (COMER). Con ello, los usuarios finales pueden reducir la latencia de un extremo a otro con el desarrollo de la sensibilidad de las aplicaciones.

Cabe mencionar que estos dispositivos 5.2 tienen un control LE Power el cual ejerce una mejora en la potencia de transmisión cuando se encuentran conectados dos dispositivos. También pueden exigirá para menguar el uso de energía y estabilizar la calidad de la señal el cambio en la potencia de transmisión.

Gracias al LE Audio se transmiten los datos de sonido en dispositivos de espectro de baja calidad, utiliza un nuevo algoritmo de compresión para mantener la calidad. También incluye el nuevo códec LC3 (Códec de comunicación de baja complejidad) de alta calidad y poca robustez, aportando un audio de mejor calidad y disminuyendo el consumo de energía.

2.3 Características

Como hemos introducido, BLE es un subconjunto del estándar Bluetooth que se caracteriza por un bajo consumo. BLE sacrifica el rendimiento y el rango operativo a favor de un menor consumo de energía. Estas son algunas de sus características:

2.3.1 Seguridad

Una característica principal que trajo consigo BLE fue el cifrado de 128 bits y la autenticación. También se añaden dos puntos más que mejoran la seguridad, los cuales son:

- **Salto de frecuencia adaptable (AFH):** El estándar Bluetooth permite que los dispositivos en comunicación se pongan de acuerdo sobre qué canales utilizar de los 37 canales de datos disponibles. Si el dispositivo central detecta una gran interferencia en un canal, puede iniciar una actualización del mapa de canales. La ventaja de AFH es que los dispositivos en comunicación están continuamente monitorizando su entorno en busca de interferencias y cambian continuamente el mapa de canales para hacer frente a las interferencias. [5]

- **Corrección de errores hacia adelante (FEC):** Es un método que añade información redundante en las transmisiones para detectar errores.

Realmente la seguridad es clave en la fase de emparejamiento que es donde se encuentra el punto débil de los sistemas BLE, ya que una vez emparejados, la comunicación sí es segura. En el emparejamiento tenemos dos fases, una primera donde se intercambian información de los dispositivos y sus capacidades la cual no está encriptada y una segunda donde se generan e intercambian las claves.

Una vez terminado el emparejamiento, se lleva a cabo la vinculación donde los dispositivos guardan los datos de autenticación para futuras conexiones. Por lo tanto entramos en un poco de detalle en los diferentes métodos de emparejamiento que existen: [6]

- **Just Works:** Es el predeterminado, por lo tanto el más común pero el menos seguro ya que la clave temporal que se intercambia durante la 2º fase de emparejamiento se establece en cero por lo que sería muy vulnerable a ataques de cualquier dispositivo que quiera conectarse con la clave temporal de 0.

- **Out of Band (OOB):** En este emparejamiento se usa otro tipo de protocolo para enviar datos críticos.

- **Claves de paso:** Mediante este método se establece una clave de 6 dígitos que el usuario debe ingresar en ambos dispositivos.

- **Comparación numérica:** Este método es muy parecido al anterior pero se utiliza un número aleatorio que intercambian previamente para generar la clave.

2.3.2 Banda de frecuencia

BLE y Bluetooth clásico comparten ciertas características, como que ambos operan en la banda ISM de 2.4GHz, pero aquí BLE opera en modo suspensión la mayoría del tiempo, lo que hace que consuma menos y alargue la vida útil de las baterías en ciertas aplicaciones. También utiliza las modulaciones GFSK (por la que los pulsos de datos se filtran con un filtro gaussiano antes de aplicarse para alterar la frecuencia de la portadora, con el fin de que las transiciones de frecuencia sean más suaves) y FHSS para minimizar los efectos del desvanecimiento y el ruido ya que comparten banda con la tecnología WiFi.

Sin embargo, BLE utiliza solo 40 canales con un espaciado entre canales de 1 a 2 MHz, a diferencia de Bluetooth clásico (que utilizaba 79 con un espaciado de 1MHz). Tres canales están reservados para el proceso de advertising (37, 38, 39) utilizándose el resto para el intercambio de información [7]

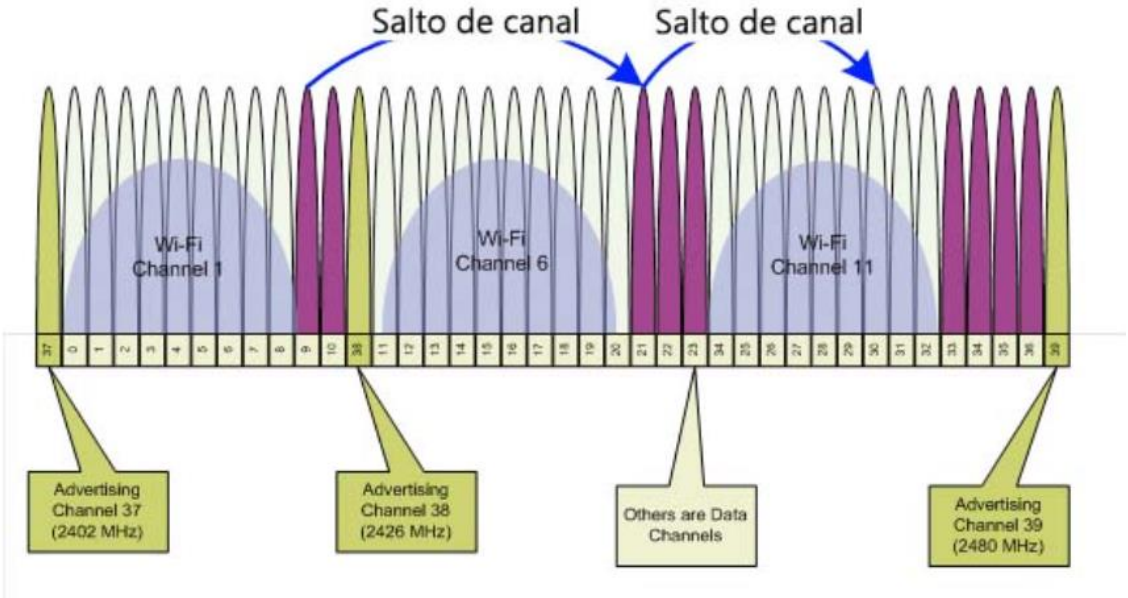

Ilustración 5: Bandas de frecuencias BLE [7]

2.3.3 Tasa de datos

Aquí es donde vemos un descenso del valor puesto que pasamos de un valor teórico de hasta 3Mbps en Bluetooth clásico a un 1Mbps en BLE. En las últimas versiones de BLE ya se puede llegar hasta los 2Mbps.
Se envía un bit por símbolo. El throughput ronda los 0.27-1,37Mbit/s.

2.3.4 Alcance

Teóricamente posee un alcance de hasta 100 metros, aunque esto se puede ver decrementado si la señal se va encontrando con obstáculos.

2.3.5 Latencia

Típicamente es de 6ms, comparado con los 100ms en el Bluetooth clásico.

2.3.6 Consumo

Aquí es donde más brilla el protocolo ya que se sitúa entre los 0,01-0.50 W (comparados con 1W que era antes). Para reducir el consumo de energía, un dispositivo BLE se mantiene en modo de suspensión la mayor parte del tiempo. Cuando ocurre un evento, el dispositivo se activa y se comienza a transferir la información.

El consumo pico suele ser inferior a los 15mA y el promedio es aproximadamente 1µA.

Estos datos hacen que para aplicaciones donde el dispositivo se alimente a través de baterías/pilas, la vida útil se pueda alargar hasta los diez años.

2.3.7 Topología

Utilizan la topología *scatternet* que se ilustra en la siguiente figura

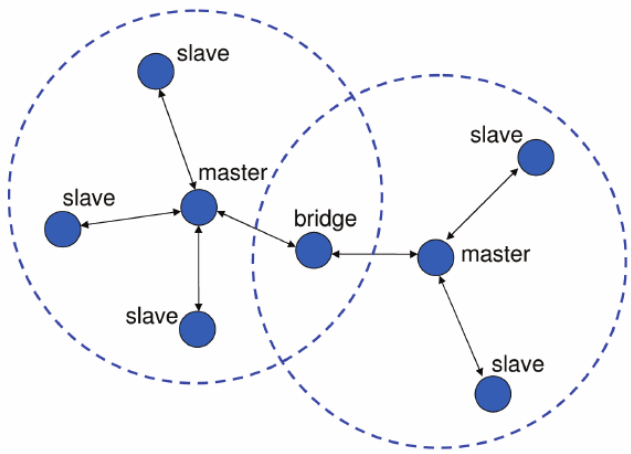

Ilustración 6: Topología scatternet

2.4 Pila de Protocolos

BLE dispone de una pila de protocolos en referencia a la capa OSI que se ilustra en la siguiente figura [8]:

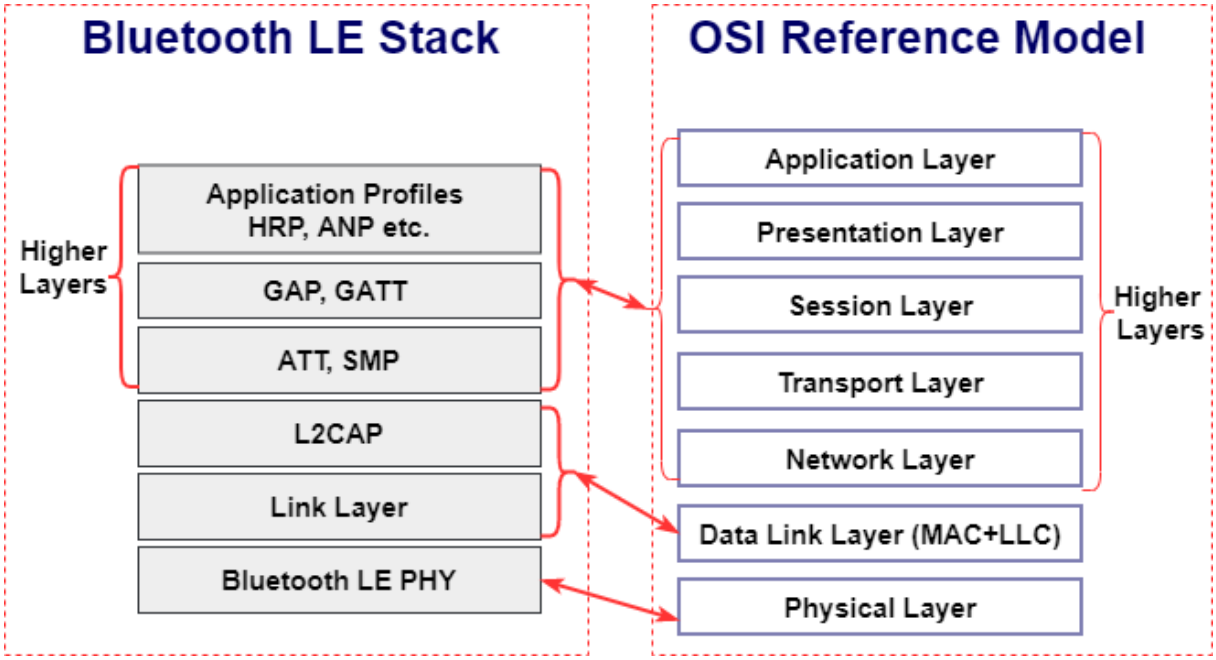

Ilustración 7: Stack BLE [8]

2.4.1 Capa física

La **capa física** es la que se encarga de proporcionar la comunicación física. Como se ha comentado en el punto anterior, se utiliza la banda ISM de 2.4GHz con 40 canales. Para encontrar el canal óptimo se utiliza el AFH.

2.4.2 Capa de enlace

La **capa de enlace** lleva a cabo la gestión de las características de la comunicación, la definición de roles *(Advertiser, Scanner, Master and Slave)* y el control de cambios de parámetros de la conexión/encriptación. Estos roles se definen en la máquina de estados de la capa: [9]

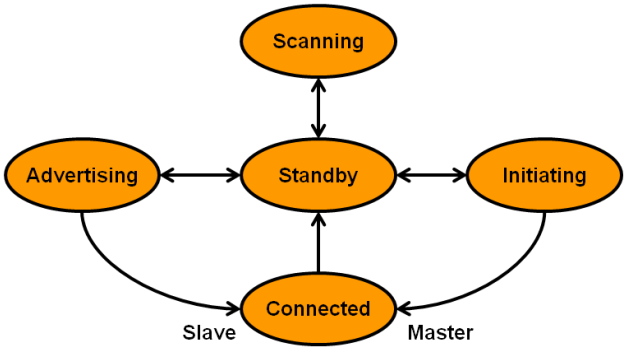

Ilustración 8: Roles máquina de estados capa de enlace [9]

Los dispositivos maestros buscan otros dispositivos. Normalmente, el maestro es un smartphone/tableta/PC (dispositivo inteligente Bluetooth). Los dispositivos esclavos anuncian y esperan conexiones. Por lo general, el esclavo es un pequeño dispositivo como un wearable. Un esclavo solo se puede conectar a un maestro, pero un maestro se puede conectar a varios esclavos. El número máximo de esclavos estará definido por la pila de Bluetooth que utilice el maestro.

Hemos de ir introduciendo también los términos cliente/servidor. Los dispositivos cliente acceden a recursos remotos a través de un enlace BLE utilizando el protocolo GATT (veremos a continuación). Por lo general, el maestro es el cliente. Los dispositivos del servidor tienen una base de datos local y métodos de control de acceso, y proporcionan recursos al cliente remoto. Por lo general, el esclavo es el servidor.

2.4.3 Capa L2CAP

La capa **L2CAP** (Logic Link Control and Adaptation Protocol) se encarga de la multiplexación, la fragmentación y la recombinación. Esta capa por tanto es la encargada de dar acceso y soporte a los dos protocolos fundamentales: [10]

- **ATT** (Attribute Protocol) un protocolo basado en atributos presentados por dispositivo, con arquitectura cliente-servidor, que permite el intercambio de información
  ATT también se encarga de realizar la organización de datos en atributos como se muestra en la figura [8]:

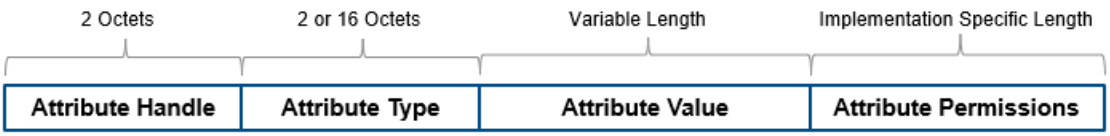

Ilustración 9: Trama ATT [8]

  Los atributos del dispositivo se representan como:

  - **El identificador de atributo** es un valor de identificador de 16 bits asignado por el servidor para permitir que un cliente haga referencia a esos atributos.
  - **El tipo de atributo** es un identificador único universal (**UUID**) definido por Bluetooth SIG. Por ejemplo, UUID 0x2A37 representa una medida de frecuencia cardíaca.
  - **El valor del atributo** es un campo de longitud variable.
  - **Los permisos de atributo** son conjuntos de valores de permiso asociados con cada atributo. Estos permisos especifican los privilegios de lectura y escritura y su nivel de seguridad.

- **SMP** (Security Manager Protocol) protocolo que proporciona un framework para generar y distribuir claves de seguridad entre dos dispositivos.

2.4.4 Capa GAT

La **capa GAT** (Generic Acces Profile) permite que un dispositivo sea visible para los demás y determina cómo pueden interoperar, es decir, los roles de iniciación, los modos de operación, como llevan a cabo el establecimiento de comunicación, la seguridad…

2.4.5 Capa GATT

Finalmente, la **capa GATT** (Generic Attribute Profile) define cómo debe ser la transferencia de información entre los dispositivos BLE. Podemos decir que encapsula el ATT y coordina el intercambio de perfiles en un enlace Bluetooth LE. Los perfiles incluyen información y datos.

Dentro de GATT debemos pararnos un poco para aclarar un par de términos. GATT organiza los atributos los cuales se organizan GATT Server Profiles que se agrupan en Services que contienen Characteristics que contienen Atributos (Declaration, Value, Descriptor). Esto se comprende mejor en la siguiente ilustración [11]

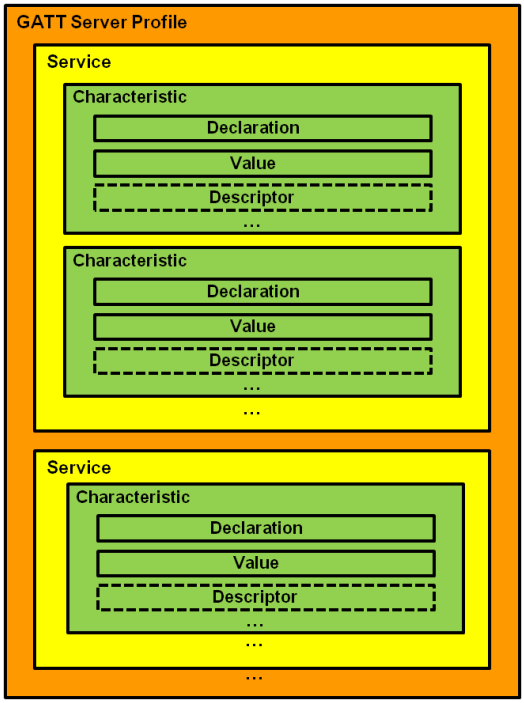

Ilustración 10: Ilustración de un perfil GATT [11]

Una **definición de servicio** es una colección de datos y comportamientos asociados para lograr una función particular. Los servicios se clasifican en Públicos o Privados.

- Público - definido por Bluetooth SIG (UUID de 16 bits)
- Privado definido por el proveedor (UUID de 128 bits)

Las **características** son esencialmente contenedores para los datos del usuario.

Podemos ver el ejemplo para el GATT Heart Rate Service [11]

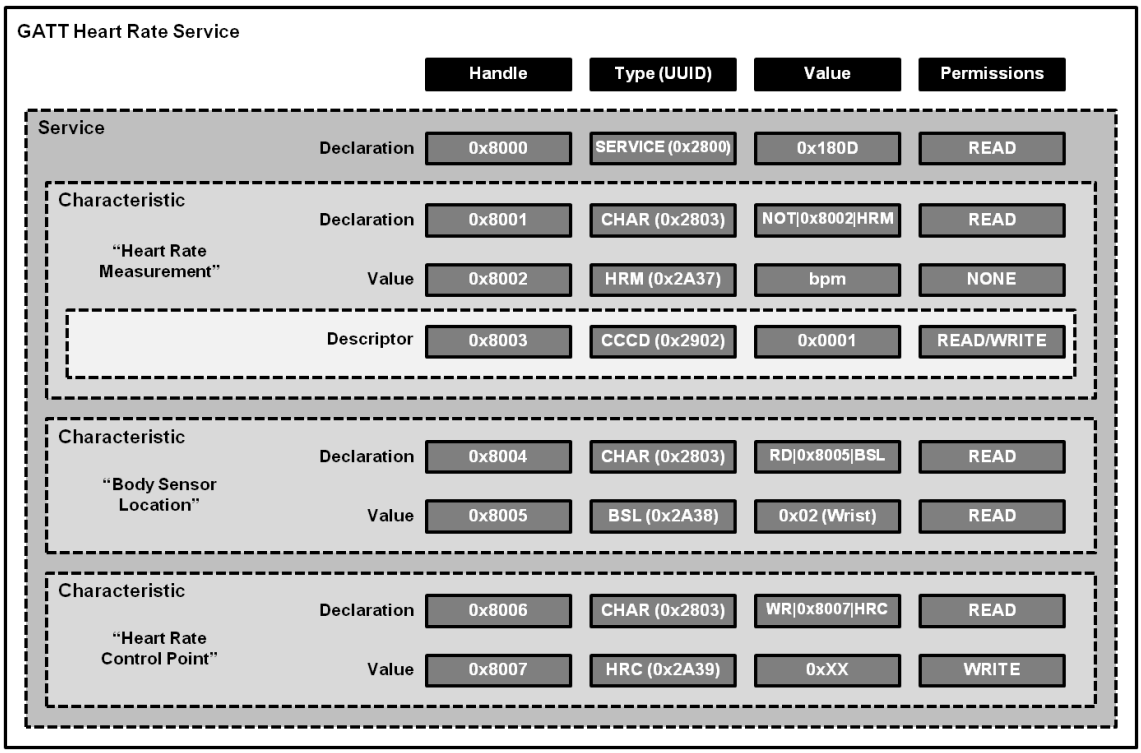

Ilustración 11: Ejemplo GATT Heart Rate Service [11]

2.5 Ventajas y desventajas

Entre las ventajas del uso de BLE nos encontramos:

- Simplicidad
- Bajo consumo (eficiencia energética)
- Baja latencia
- Alta aceptación, es mundialmente usado

Por otro lado, esta tecnología tiene sus puntos débiles:

- El alcance es limitado, teóricamente son menos de 100m
- El alcance aún puede verse más mermado dependiendo de los alrededores ya que esto degrada las ondas BLE.
- Bajo ancho de banda y baja velocidad.
- La seguridad, aunque ha mejorado, sigue siendo un punto crítico ya que durante el establecimiento de la conexión se pueden realizar ataques.
- Compatibilidad entre versiones, BLE no es compatible con el Bluetooth clásico.

Hemos de ser conscientes de que dependiendo de la aplicación que vayamos a realizar, las desventajas puede que no sean determinantes, por ejemplo, para un *wearable,* que el alcance sea pequeño no es muy determinante puesto que, en la mayoría de los casos, el *smartphone* estará cerca.

Hemos de decantarnos por BLE si nuestra aplicación busca un: bajo coste, bajo consumo, una tasa de datos reducida y un corto alcance.

2.6 Campos de aplicación

BLE es usado en un grandísimo número de aplicaciones. Todos los últimos *smartphones* y portátiles, *wearables lo* incluyen, por ello su uso está tan aceptado. La siguiente figura nos muestra todos los posibles campos de aplicación donde se podría utilizar BLE [12]

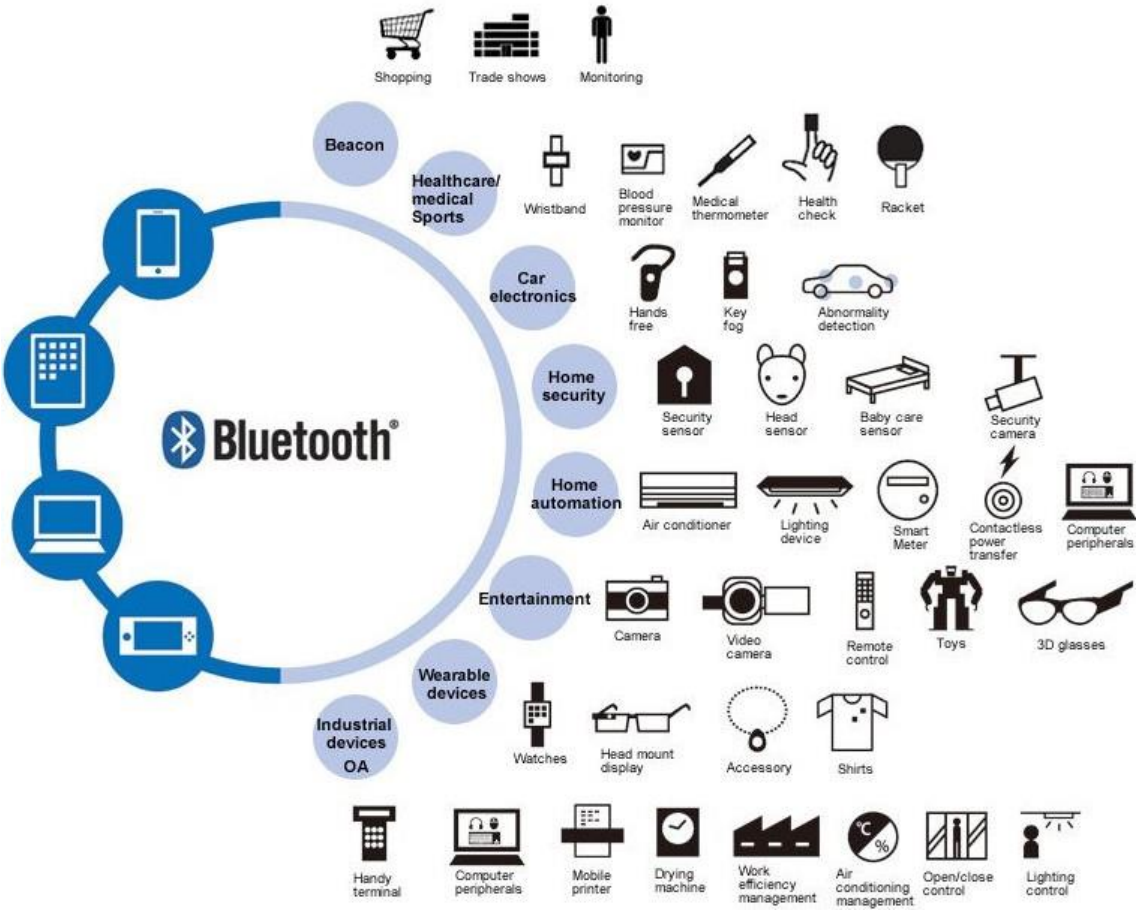

Ilustración 12: Campos de aplicación BLE [12]

Algunas aplicaciones novedosas son:

- **Carga inalámbrica:** Un BLE compatible con un estándar de interfaz para la transferencia de energía eléctrica inalámbrica basada en los principios de resonancia magnética como es Rezence, desarrollada por A4WP, permite cargar varios dispositivos sin la necesidad de estar en contacto con un cargador.

- **Medicina:** Ciertas máquinas podrían transmitir la información recabada en un paciente a través de Bluetooth BLE, como un monitor de glucosa en sangre o presión. Algo muy interesante son las llamadas de emergencia, en cuanto se marca el 911 el dispositivo móvil activa automáticamente sus capacidades de ubicación, recogiendo todos los datos posibles de ubicación para enviarlas inmediatamente al receptor de la llamada.

- **Redes sociales** Se utiliza para detectar usuarios cercanos como base para el mapeo de las redes sociales.
- **Cocina inteligente** El tenedor inteligente ya se ha inventado utilizando esta tecnología BLE. Este invento tan innovador permite al usuario monitorear y rastrear sus hábitos alimenticios para perder peso. Mide el tiempo entre comida, la duración de ellas, la cantidad de porciones, etc.

- **Etiquetas inteligentes** Estas Etiquetas demuestran el potencial de la tecnología BLE. Dichas etiquetas se comunican con el dispositivo móvil a través de Bluetooth lo que permitirá al mismo realizar un seguimiento de la ubicación de esta. Por tanto, se pueden adjuntar estas etiquetas a diferentes objetos para tener una ubicación conocida del objeto en sí.

3. Subsistemas del Proyecto

El proyecto abarca dos subsistemas independientes en donde el SoC BLE principal actuará como periférico y como central. De esta manera se podrá ver como se llevan a cabo las diferentes comunicaciones entre los dispositivos.
En este capítulo se mostrará una primera visión de los subsistemas. Los componentes que lo conforman se definirán en el capítulo 4. Estructura Hardware del Proyecto. Su funcionamiento se detallará en 5. Estructura Software del Proyecto y finalmente podremos observar el comportamiento en 6. Resultados obtenidos.

3.1 Subsistema 1

Con este primer subsistema se pretende adquirir los datos de un sensor medioambiental (de nivel de agua) en un teléfono móvil.

El sensor medioambiental es el encargado de enviar los datos captados hacia el microcontrolador. En este subsistema el sensor será de profundidad, es decir, el sensor nos marcará el nivel de agua de la superficie.

El microcontrolador deberá acondicionar los datos recibidos a través de uno de sus pines digitales/analógicos e integrarlos en una cadena de texto legible. También cada minuto hará una media de los valores que ha ido recibiendo y lo codificará en una cadena. Dichas cadenas serán enviadas cada 5 segundos aproximadamente hacia el SoC BLE, el cual, mediante notificaciones, enviará los datos (las cadenas) a un teléfono móvil.

En este subsistema se integrarán los siguientes dispositivos:
- Sensor medioambiental
- Microcontrolador
- SoC BLE
- Teléfono móvil

Se llevarán a cabo las siguientes comunicaciones:
- Comunicación Serie (UART)
- Comunicación Inalámbrica (BLE)

El diagrama del subsistema responde a la siguiente figura:

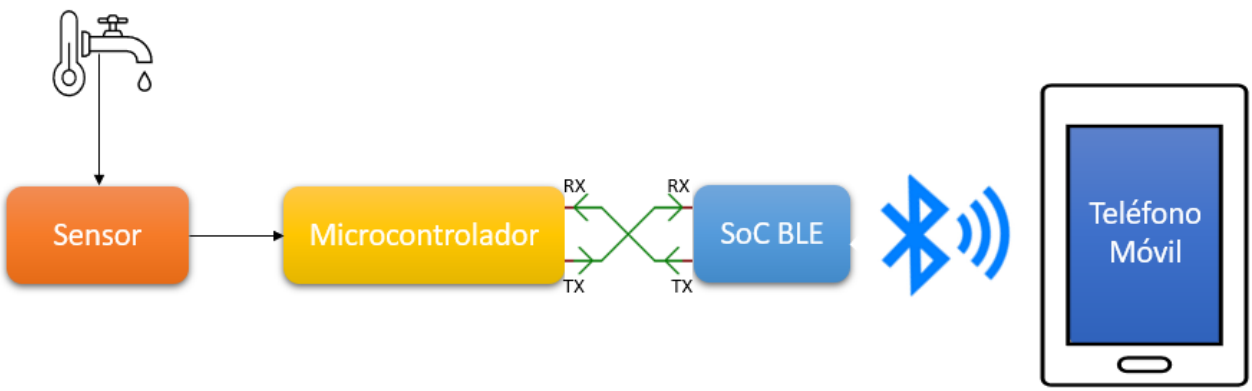

Ilustración 13: Diagrama Subsistema 1

3.2 Subsistema 2

Con este segundo subsistema se pretende adquirir los datos de un sensor medioambiental en un ordenador.

El sensor medioambiental en este caso está integrado en una plataforma junto con el SoC BLE (que actuará como periférico). Dicho sensor captará datos medioambientales tales como: temperatura, presión, humedad, calidad del aire/gas…

La plataforma al contar con el SoC BLE enviará los datos adquiridos por los sensores a través de BLE mediante notificaciones hacia otro SoC BLE. Tras recibirlos, el SoC BLE que actúa como central, enviará los datos a través de la UART hacia el ordenador.

En este subsistema se integrarán los siguientes dispositivos:

- Multi-sensor con SoC BLE (periférico)
- SoC BLE (central)
- Portátil

Se llevarán a cabo las siguientes comunicaciones:

- Comunicación Inalámbrica (BLE)
- Comunicación Serie (UART)

El diagrama del subsistema responde a la siguiente figura:

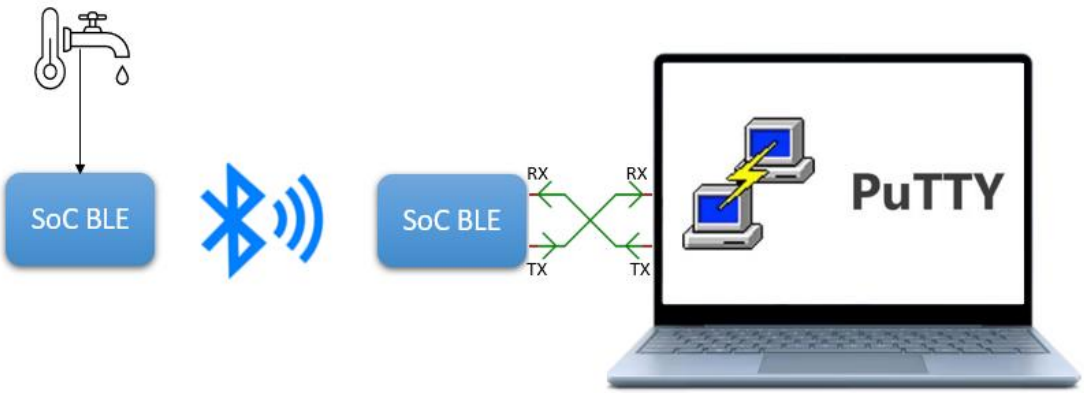

Ilustración 14: Diagrama Subsistema 2

4. Estructura Hardware del Proyecto

En este capítulo se detallará el hardware elegido para el proyecto y el lugar que cada uno ocupa en cada subsistema desarrollado. Sobre cada hardware elegido se dará una breve descripción de sus características y el papel que desempeña en el subsistema.
También se detallarán de una manera más breve otros dispositivos que fueron considerados pero que finalmente no fueron escogidos.

4.1 Microcontrolador Arduino UNO

4.1.1 Descripción del dispositivo

Antes de detallar el microcontrolador elegido vamos a dar una pequeña descripción de qué es. Un microcontrolador se podría definir como un circuito integrado diseñado para realizar una acción específica en un sistema integrado. Normalmente el chip del microcontrolador incluye el procesador, la memoria y periféricos de entrada/salida.

- El procesador (CPU) es el encargado de realizar las operaciones aritméticas y lógicas del programa. Es el que gobierna el sistema.
- La memoria del controlador almacena los datos del sistema además del propio código del programa. La información puede ser volátil o no volátil; por ello normalmente hablamos de memoria RAM (volátil – usada para información temporal) y memoria FLASH (no volátil – usada para información que necesita persistir y para el código del programa)
- Los periféricos permiten intercambiar información con otros dispositivos a través de los pines diseñados para ello. Estos periféricos interactúan con la CPU y la memoria.

La elección del microcontrolador a usar debe estar basado en los requisitos del proyecto que vayamos a desarrollar. Las características básicas a tener en cuenta deberán ser: memoria disponible, capacidad de cómputo, número de pines E/S, periféricos disponibles (ADC/buses de comunicación…), coste…

La búsqueda se centró en los microcontroladores desarrollados por Arduino debido a la gran comunidad que lo sostiene (es un proyecto *open-source*) y a la gran gama de proyectos que se pueden realizar.

Arduino UNO fue la primera versión de las tarjetas Arduino. Es una de las más populares ya que fue la que hizo despegar el proyecto Arduino. Una de las diferencias frente a sus predecesoras es que no utiliza el chip controlador de USB a serie de FTD, sino que cuenta con el Atmega16U2 (Atmega8U2 hasta la versión R2) programado como convertidor USB a serie.

Esta placa evolucionó hacia Arduino UNO R3. Posteriormente también se creó la versión Arduino UNO R3 SMD. La diferencia es que en UNO R3, el procesador ATmega está encapsulado en un paquete de doble hilera (*dual in-*

line package DIP) mientras que en la versión SMD, como su nombre indica utiliza un empaquetado SMD, esto hace que sea ligeramente más barato.

En la siguiente figura se pueden ver las diferentes tarjetas:

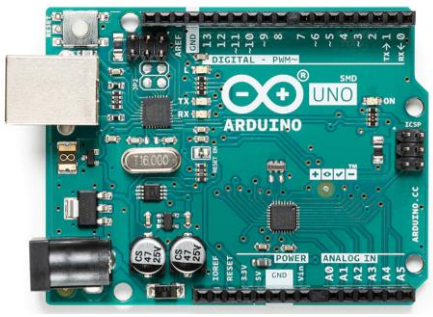

Ilustración 16: Arduino UNO R3 SMD

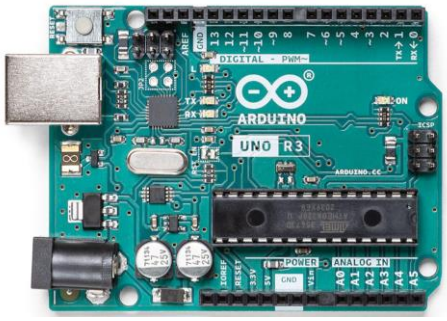

Ilustración 15: Arduino UNO R3

Esta tarjeta es la seleccionada para la realización del proyecto, en concreto la versión SMD. Las principales características son las siguientes [13]:

Microcontrolador	ATmega328P
Voltaje de operación	5V
Pines E/S Digitales	14 (6 con salida PWM)
Pines E Analógicos	6
Memoria Flash	32KB (ATmega328P)
Memoria SRAM	2KB (ATmega328P)
Memoria EEPROM	1KB (ATmega328P)
Velocidad del reloj	16 MHz
Dimensiones	68,6 mm x 53,4 mm
Peso	25g

Tabla 1: Características Arduino UNO R3

Los pines se detallan en la siguiente figura:

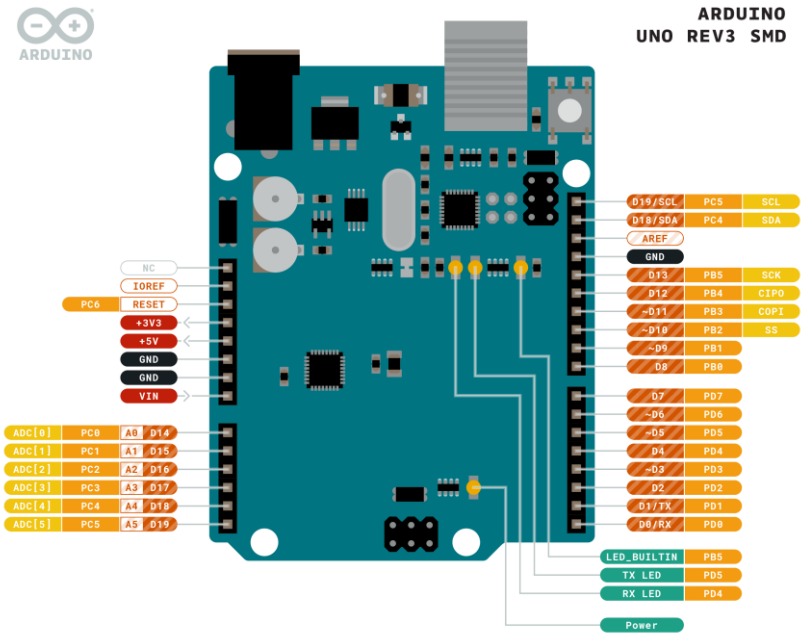

Ilustración 17: Pinout Arduino UNO R3 SMD [13]

En cuanto a los pines de alimentación:

- `Vin`: Si se utiliza una fuente de alimentación externa como alimentación, podemos acceder a través de este pin. También se puede alimentar a través de la clavija de alimentación.
- `5V`: Este pin de salida ofrece una señal de 5V a través del regulador.
- `3V3`. : Este pin de salida ofrece una señal de 3,3V a través del regulador.
- `GND`: Pin de tierra.

Los pines digitales con los que cuenta operan a 5V y dan/reciben una corriente de 20mA. Cuentan con una resistencia de pull-up interna (desconectada por defecto) de 20-50kΩ. La corriente máxima es de 40mA y no debe sobrepasarse.

También nos encontramos con unos pines digitales que permiten llevar a cabo la comunicación serie: 0 (RX) y 1 (TX). Estos son utilizados para recibir (RX) y transmitir (TX) datos serie TTL a los pines correspondientes del chip ATmega8U2 USB-to-TTL Serial.

También soporta comunicación I2C y SPI. El software Arduino (IDE) incluye una librería `Wire` para simplificar el uso del bus I2C. Si se utilizan las funciones de la librería `SoftwareSerial` podríamos realizar la comunicación serie a través de cualquier otro pin digital.

El precio de estas tarjetas en la página oficial es de 20 euros. Sin embargo, al ser el proyecto Arduino de código abierto, numerosos fabricantes venden adaptaciones de las placas originales, se las conoce como Arduino UNO genérico, las cuales son aún más baratas.

También en otras páginas no oficiales se pueden encontrar a un precio inferior a 10 euros lo que las hace unas tarjetas perfectas para la iniciación o proyectos que no necesiten de demasiado cómputo. También se venden muchos kits de iniciación a la programación en Arduino que incluye la tarjeta Arduino UNO R3 (SMD) junto a numerosos sensores y una *protoboard* con resistencias para realizar diferentes conexionados.

4.1.2 Funcionalidad

Este dispositivo será usado en el subsistema número 1. Su función es alimentar y recoger los datos del sensor de nivel de agua, acondicionarlos e integrarlos en una cadena de texto que será enviada a través de la UART hacia el SoC BLE.

4.2. nRF52 DK (PCA10040)

4.2.1 Descripción del dispositivo

Otra pieza clave fundamental en el proyecto será el SoC que lleve a cabo la comunicación mediante BLE.
Se ha de encontrar una placa de desarrollo que integre un SoC BLE y ciertos periféricos para llevar a cabo las funcionalidades descritas en el capítulo 3. Subsistemas del Proyecto.
El elegido para el proyecto es el nRF52 DK, también conocido como PCA10040. Es una placa de desarrollo recomendada para BLE, Bluetooth, ANT y aplicaciones de 2.4GHz.

Algunas de las características del kit son: [14]

- Solución SoC basada en flash nRF52832 ANT™/ANT+™, Bluetooth® Low Energy.
- Botones y LEDs para la interacción con el usuario
- Interfaz de E/S para módulos de Arduino (compatibilidad con el estándar)
- Depurador SEGGER J-Link OB
- Interfaz de puerto COM virtual a través de UART
- Programación de dispositivos de almacenamiento masivo mediante arrastrar y soltar
- Soporte del modo de escucha NFC-A
- Alimentación a través de USB, de una pila CR2032 o de alimentación externa.

Este kit se recomienda para numerosas aplicaciones en el ámbito del *IoT*, *wearables*, *appacessories*, domótica, juguetes…

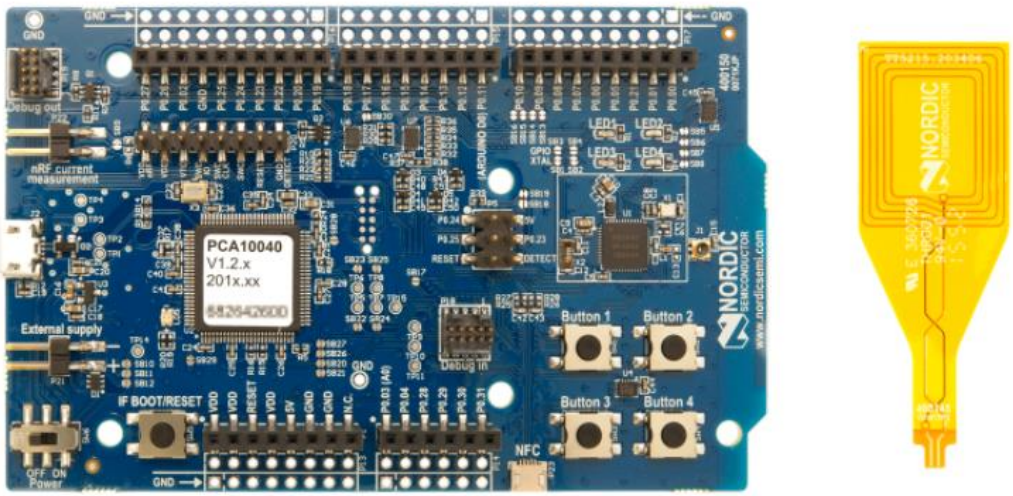

Ilustración 18: PCA10040 DK

En cuanto al SoC BLE que integra es el nRF52832 el cual soporta BLE. En cuanto a las características: [15]

CPU	64MHz ARM Cortex-M4
Memoria Flash	512 KB
Memoria RAM	64 KB
Cache	8 KB
Seguridad	AES-128/ECB/CCM/AAR
Protocolos inalámbricos	BLE, Bluetooth Mesh, NFC, ANT
Tasa	2/1 Mbps (BLE) 36+
Potencia TX	(+4, -20) dBm
Sensibilidad RX	-96 dBm a 1Mpbs y -89 dBm a 2Mbps
Consumo radio BLE	(5.4 , 7.5) mA
Interfaces digitales	SPI, TWI, I^2S, UART, PDM, QDEC
Voltaje de operación	(1.7 , 3,6) V
Rango de temperatura	-40ºC a 85ºC

Tabla 2: Características PCA10040

En la siguiente figura se observa el diagrama de conectores:

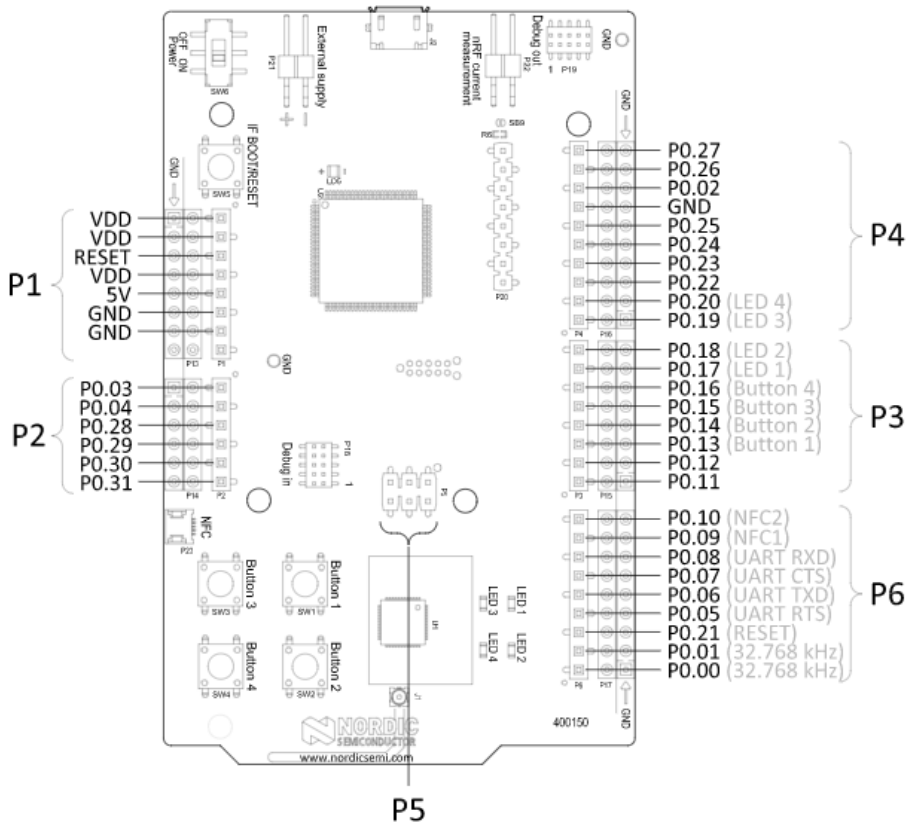

Ilustración 19: Conectores PCA10040 [14]

En las aplicaciones desarrolladas se utilizarán los conectores UART RXD y UART TXD.

La interconexión entre los bloques del kit de desarrollo es la siguiente:

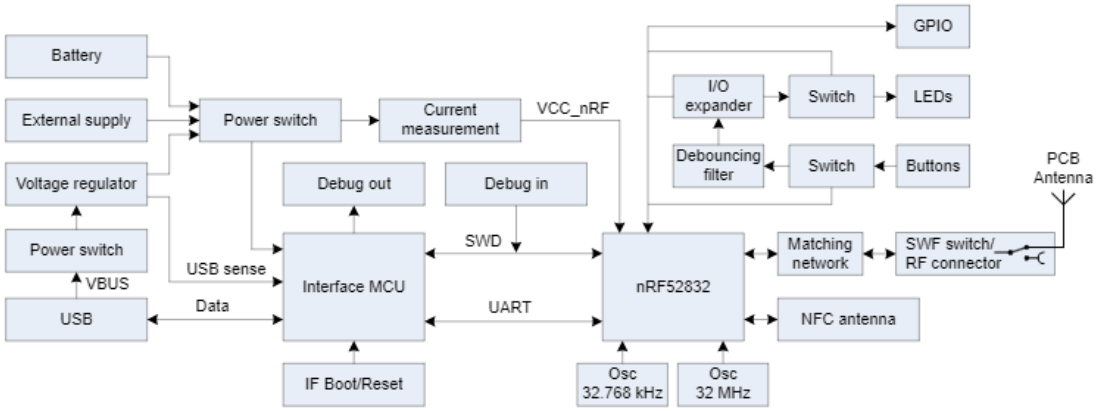

Ilustración 20: Interconexión bloques PCA10040

En cuanto a la arquitectura Software se detallará en el capítulo [5.1. Arquitectura software nRF](#).

4.2.2 Funcionalidad

Este dispositivo será usado en ambos subsistemas, pero desarrollará diferentes funciones:

- Subsistema 1: En este subsistema el nRF52 DK se encargará de recibir los datos (cadenas de texto que contienen los datos del sensor de agua) por la UART (pines 06 y 08) y posteriormente los enviará por BLE hacia el teléfono móvil
- Subsistema 2: En este subsistema el nRF52 DK se encargará de recibir los datos a través de BLE y los mostrará en el ordenador mediante la UART.

4.3 Thingy:52 (PCA20020)

4.3.1 Descripción del dispositivo

Thingy:52 o PCA20020 está construido alrededor del nRF52832 al igual que el DK anterior. Es una plataforma que integra varios sensores junto con el SoC, haciéndolo una opción perfecta a la hora de desarrollar prototipos y similares sin tener que modificar su firmware.

Ilustración 21: Thingy:52

Sus características principales son: [16]

- Compatible Bluetooth LE y NFC
- Botón y LEDs programables por el usuario
- Sensor de movimiento de nueve ejes
- Acelerómetro de bajo consumo
- Sensores de humedad, presión atmosférica y temperatura
- Sensor de calidad del aire/gas
- Sensor de color
- Altavoz y micrófono
- Batería recargable de Li-Po con capacidad de 1440 mAh

Sus diferentes sensores se detallan en la siguiente figura:

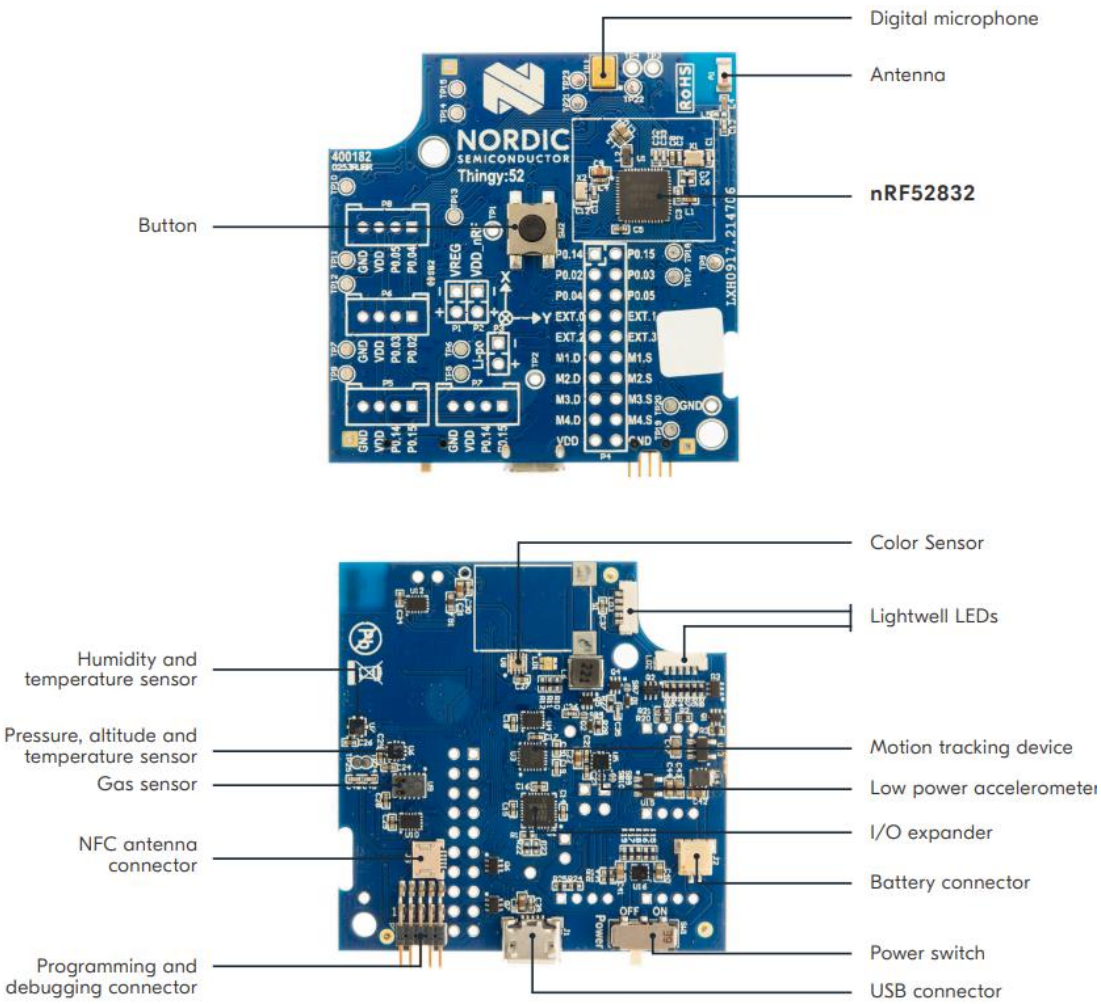

Ilustración 22: Sensores Thingy:52 [17]

En cuanto a la arquitectura Software se detallará en el capítulo 5.1. Arquitectura software nRF.

4.3.2 Funcionalidad

Este dispositivo será utilizado en el subsistema 2 para enviar recopilar datos de los diferentes sensores que posee y enviarlos hacia el nRF52.

4.4 Sensor de nivel de agua

4.4.1 Descripción del dispositivo

Para completar el subsistema 1 se necesita un sensor externo que capte la magnitud a medir, en este caso, el nivel de agua. El sensor por el que se ha optado es el siguiente:

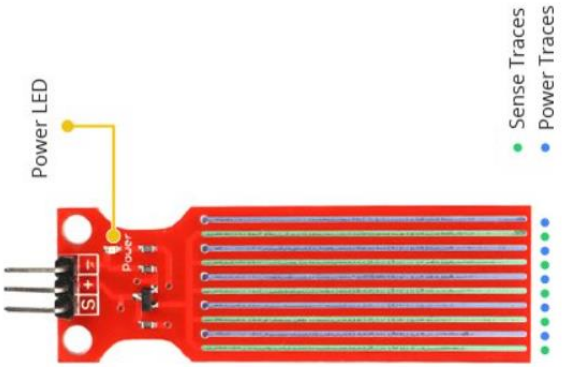

Ilustración 23: Sensor de profundidad [18]

Este sensor está formado por diez tiras de cobre, 5 de ellas de potencia y cinco de detección los cuales se “unen” cuando se sumergen. Su funcionamiento está basado en que los conductores actúan como una resistencia variable, cuyo valor es modificado según la distancia desde la parte superior del sensor hasta la superficie del agua. La resistencia obtenida es por tanto inversamente proporcional a la altura del agua:

- A más agua sumergida, mejor conductividad y por tanto menor resistencia
- A menos agua sumergida, peor conductividad y por tanto mayor resistencia.

La salida del sensor será por tanto un voltaje que dependerá de la resistencia que obtenemos en la placa, por lo que la salida será analógica. Los pines que dispone el sensor son los siguientes:

- S: es la salida analógica que contiene el nivel de agua. Deberá ser conectada a una entrada analógica del Arduino UNO.
- +: es la alimentación del sensor.
- -: es la conexión a tierra.

4.4.2 Funcionalidad

El sensor será utilizado en el subsistema 1 para obtener la medida del nivel de agua. El sensor será alimentado a través del Arduino y le enviará el nivel a través de un pin analógico.

4.5 nRF52840 – Dongle (PCA10059)

4.5.1 Descripción del dispositivo

Para visualizar correctamente el intercambio de mensajes a través de BLE no es solo suficiente con el uso de herramientas como nRF Connect. Para visualizar todo el tráfico que se obtiene a través de BLE es necesario utilizar un *sniffer*.

Para ello se ha optado por utilizar el nRF52840 Dongle, el cual es un pequeño dongle USB de bajo coste para Bluetooth Low Energy, Bluetooth mesh, Thread, Zigbee y 802.15.4 entre otros.

Ilustración 24: nRF52840 Dongle [19]

Algunas de sus características son: [19]

- Bluetooth 5.2
- Soporte de la capa física 802.15.4
- 15 GPIO
- Interfaz USB directa con el SOC
- Antena 2.4GHz integrada
- Botón, RGB LED y LED programables
-

Sobre este dispositivo también pueden realizarse aplicaciones para IoT y wearables entre otras. Sus características como sniffer se resumen en la siguiente tabla: [20]

Feature	nRF52840 Dongle
LE Secure Connections	Yes
Max Data Packet Size	251
Max Advertising Packet Size	255
Radio Fast Ramp-up	Yes
LE 2M Phy	Yes
LE Coded PHY	Yes
USB	Yes

Tabla 3: Características PCA10059

4.5.2 Funcionalidad

Este dispositivo no es necesario para que los subsistemas funcionen, sin embargo, tiene carácter didáctico y complementario ya que se podrán observar todos los pasos de mensajes durante el desarrollo de las pruebas en el software Wireshark.

4.6 Otros

En esta sección se exponen otras opciones que se barajaron pero que no se consideraron finalmente para el proyecto.

4.6.1 Arduino Mega 2560 Rev3

Arduino Mega 2560 Rev3 está basado en ATmega2560 y fue la evolución de la tarjeta Arduino Mega a la cual reemplaza.

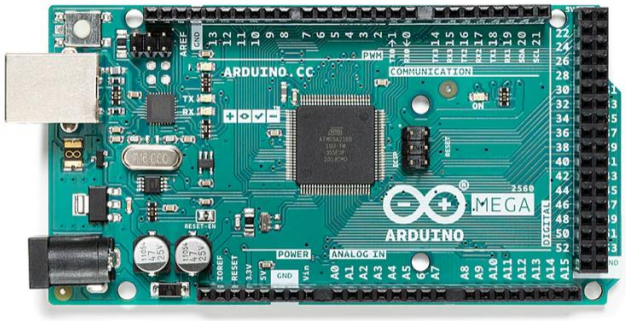

Ilustración 25: Arduino ATmega 2560 Rev3 [21]

Sus características son las siguientes: [21]

Microcontrolador	ATmega2560
Voltaje de operación	5V
Pines E/S Digitales	54 (15 con salida PWM)
Pines E Analógicos	16
Memoria Flash	256KB (ATmega328P)
Memoria SRAM	8KB (ATmega328P)
Memoria EEPROM	4KB (ATmega328P)
Velocidad del reloj	16 MHz
Dimensiones	101,52 mm x 53,3 mm
Peso	37g

Tabla 4: Características Arduino Mega 2560 Rev3

En cuanto a la alimentación es igual que la ya vista en la subsección anterior. Sin embargo, en cuanto a puertos de comunicación en este caso contamos con cuatro puertos serie (4 TX y 4 RX) lo cual nos permitiría llevar a cabo 4 comunicaciones serie con diferentes dispositivos.

El precio de esta tarjeta en la página oficial asciende a 36,80 euros, como vemos es superior a la anterior ya que incluye más pines digitales y analógicos, más espacio de memoria, más puertos serie hardware y un microcontrolador más potente.

Esta opción sería más acertada si quisiéramos recibir datos de varios sensores a través de las diferentes UARTs.

En este caso, el Arduino se utilizará en el subsistema que recogerá los datos del sensor. Dicho sensor deberá enviar los datos hacia un pin digital/analógico. El Arduino tratará los datos y los enviará mediante puerto serie (UART) hacia la placa BLE.

Por lo que la opción más acertada sería el Arduino UNO R3 dado sus características y precio.

El Arduino MEGA también nos encajaría en el proyecto, sin embargo, para la funcionalidad que requerimos el Arduino, la cual es recibir medidas y enviarlas por UART, sería desaprovechar la capacidad de la tarjeta.

4.6.2 Arduino Nano 33 BLE

El Arduino Nano 33 BLE es la evolución del Arduino Nano tradicional, que incluye el procesador nRF52840 de Nordic Semiconductors, una CPU ARM Cortex®-M4 de 32 bits. Esto hace que pueda ser cliente y host Bluetooth Low Energy y Bluetooth

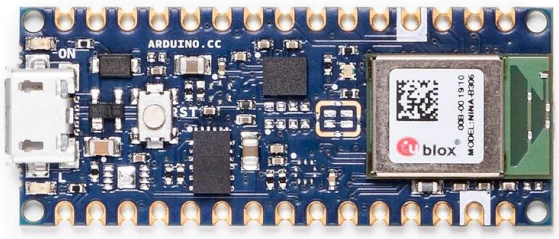

Ilustración 26: Arduino Nano 33 BLE [22]

Incluye una unidad de medición inercial de 9 ejes (incluye un acelerómetro, un giroscopio y un magnetómetro con resolución de 3 ejes cada uno).

Sus características son: [22]

Microcontrolador	nRF52840
Voltaje de operación	3,3V
Pines E/S Digitales	14 (todos con salida PWM)
Pines E Analógicos	8
Memoria Flash	1MB (ATmega328P)
Memoria SRAM	256KB (ATmega328P)
Memoria EEPROM	0
Velocidad del reloj	64 MHz
Dimensiones	45 mm x 18 mm
Peso	5g

Tabla 5: Características Arduino Nano 33 BLE

Su precio en la página oficial es de 18,40 euros.

Por otro lado, el Arduino NANO 33 BLE también hubiera sido una buena opción si el subsistema a implementar utilizara como datos los obtenidos por la unidad de medición inercial y estos fueran enviados por BLE en lugar de por UART. Sin embargo, el subsistema estaba enfocado a datos mediambientales (como temperatura, humedad…).

4.6.3 nRF51 Dongle (PCA10031)

Este dispositivo es un dongle de desarrollo USB versátil y de bajo coste para aplicaciones propietarias de BLE, ANT y 2,4 GHz que utilizan el SoC de la serie nRF51.

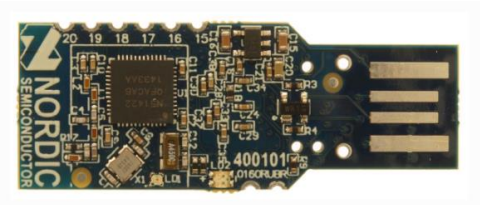

Ilustración 27: nRF51 Dongle (PCA10031) [23]

Sus características como sniffer se resumen en la siguiente tabla: [20]

Feature	nRF51 Dongle
LE Secure Connections	No
Max Data Packet Size	27
Max Advertising Packet Size	37
Radio Fast Ramp-up	No
LE 2M Phy	No
LE Coded PHY	No
USB	No

Tabla 6: Características PCA10031

Esta opción finalmente no se consideró puesto que sus características como sniffer no iban a ser suficientes para el proyecto.

4.7 Diagrama del proyecto

4.7.1 Subsistema 1

Los dispositivos son alimentados a través del puerto USB del portátil y entre ellos, se interconectan de la siguiente manera:

Sensor	Arduino
+	D7
-	GND
S	A5

Arduino	PCA10040
TX (1)	RX (P 0.6)
RX(2)	TX (P 0.8)

Tabla 7: Conexión pines Subsistema 1

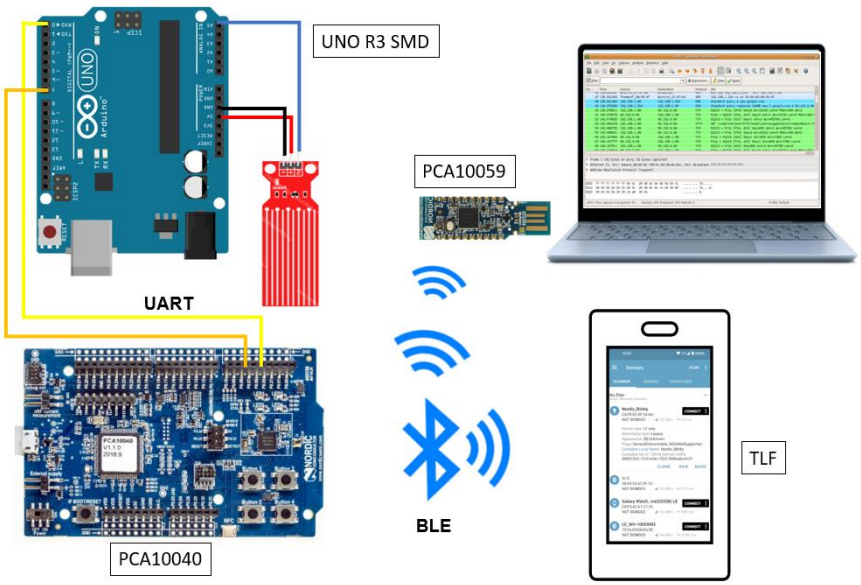

Ilustración 28: Diagrama conexión Subsistema 1

4.7.2 Subsistema 2

Los dispositivos son igualmente alimentados a través del puerto USB del portátil y entre ellos, no hay ninguna interconexión ya que la comunicación es a través de BLE.

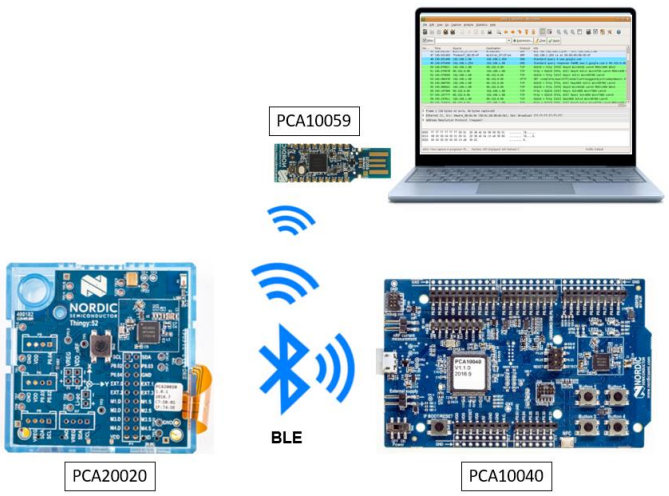

Ilustración 29: Diagrama conexión Subsistema 2

5. Estructura Software del Proyecto

La implementación del software del proyecto debe llevarse a cabo siguiendo las herramientas recomendadas por los fabricantes en cada caso.
Además, necesitaremos programas complementarios para poder completar el proyecto.

En este capítulo se definirán: la arquitectura software de los dispositivos, los IDEs para cada hardware y los programas complementarios que se han usado y una descripción del código realizado.

5.1 Arquitectura software nRF

Este punto se centrará en la arquitectura firmware de los dispositivos que utilizan BLE que en nuestro caso son: nRF52 y Thingy:52

La arquitectura de software del nRF52 ofrece una separación entre el código de la aplicación y los *SoftDevices* (la pila de protocolos de Nordic). Esto permite que el código de la aplicación puede ser desarrollado y compilado independientemente de la pila de pila de protocolos reduciendo así los errores.

El *SoftDevice* con el que se desarrollará el proyecto es el S132 (pila de protocolo Bluetooth 5 de alto rendimiento para los SoCs nRF52810 y nRF52832). El S132 no deja de ser un binario ya precompilado y linkado que nos ofrece la pila de protocolos para nuestro SoC. Su descarga es gratuita y se hará desde la página del fabricante, nordicsemi.com [24]

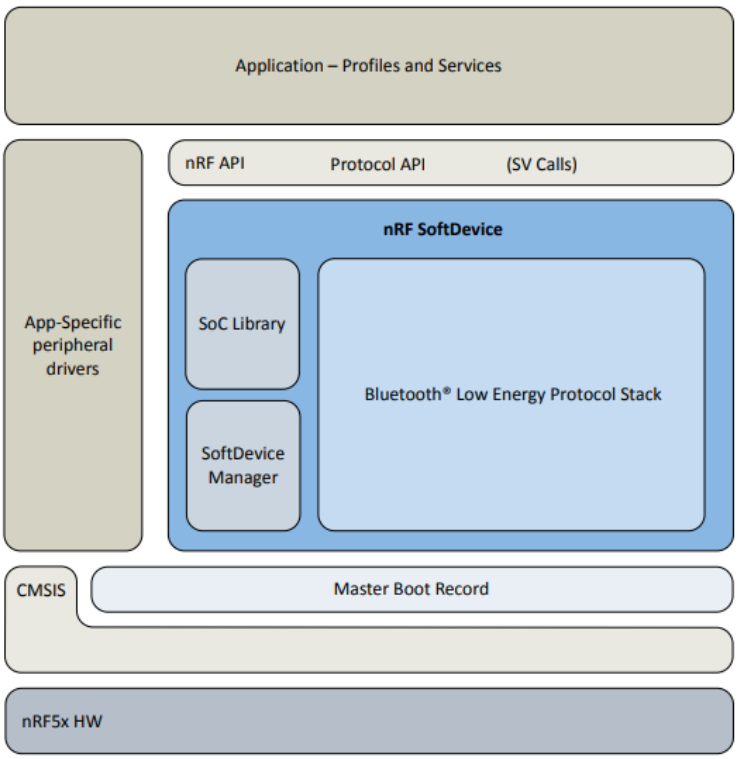

Ilustración 30: Arquitectura Software de una aplicación con SoftDevice [25]

El *SoftDevice* como vemos en la figura superior, consta de tres componentes principales: [25]
- ***SoC Library***: implementación e interfaz de programación de aplicaciones (API) de nRF para la gestión de recursos del hardware compartido
- ***SoftDevice Manager (SDM)*:** implementación y API de nRF para la gestión de SoftDevice (habilitar/desactivar el SoftDevice, etc.)
- ***BLE Protocol Stack***: implementación de la pila de protocolos y de la API

El *SoftDevice* utiliza tanto memoria Flash como memoria RAM. La cantidad de memoria Flash está fijada mientras que la cantidad de RAM depende de si el *SoftDevice* está habilitado y la configuración de la pila.

El *SoftDevice* permite que las aplicaciones implementen perfiles estándar de BLE y también casos de uso propios. La API está definida sobre el Protocolo Genérico de Atributos (GATT), el Perfil Genérico de Acceso (GAP) y el Protocolo de Adaptación y Control de Enlace Lógico (L2CAP). Otros protocolos, como el Protocolo de Atributos (ATT), el Gestor de Seguridad (SM) y la Capa de Enlace (LL), son gestionados por las capas superiores del SoftDevice, como se muestra en la siguiente figura [25]

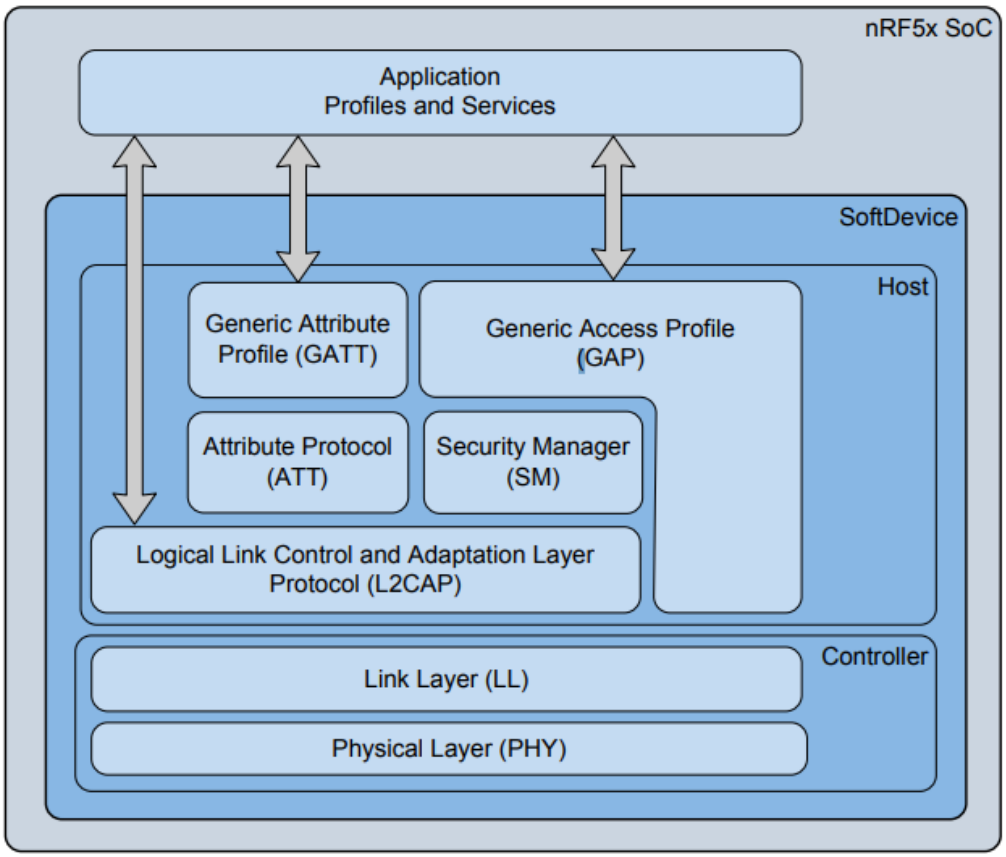

Ilustración 31: Arquitectura del stack del SoftDevice [25]

La arquitectura de Thingy:52 responde a la misma arquitectura básica:

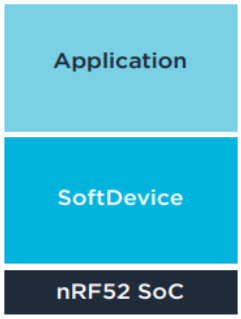

Ilustración 32: Arquitectura Software Thingy:52 [26]

Pero con algunas modificaciones que se muestran en la siguiente figura:

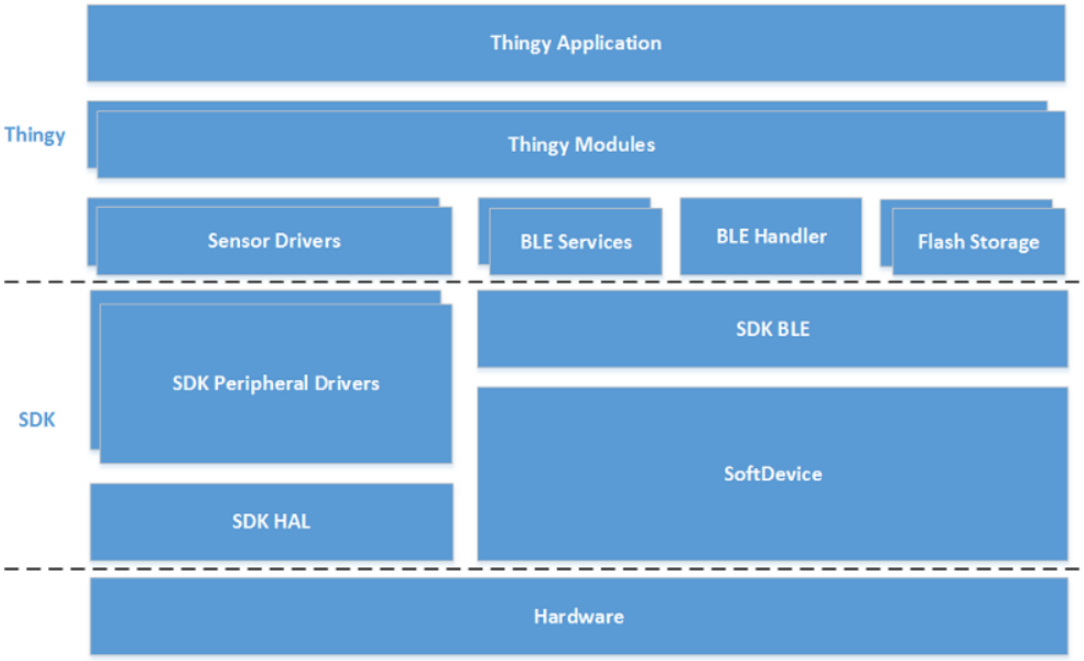

Ilustración 33: Arquitectura Firmware detallada Thingy:52 [26]

Las diferencias radican en los módulos encima del SDK: [26]

- ***Sensor Drivers***: El Thingy:52 se caracterizaba por ser una plataforma multi-sensor. Thingy utiliza los controladores periféricos del SDK y las capas de acceso al hardware (HAL) e implementa sus propios controladores de sensores que realizan las operaciones sobre ellos (habilitarlos, deshabilitarlos, cambiar la configuración, adquirir los datos…)
- ***BLE Services***: Es la capa que incluye los servicios BLE propios que utiliza el Thingy Module. Todos los servicios podrán consultarse en el Anexo 10.2. BLE Services Thingy:52. El UUID base es EF68xxxx-9B35-4933-9B10-52FFA9740042 y xxxx dependerá de la característica en concreto. Un breve resumen de los servicios es: [27]
    - ***Thingy configuration service***: Es el responsable de manejar la configuración general de los parámetros que no se corresponden

a otro módulo, como, por ejemplo: Device name, firmware version…
    - ***Environment servic***e: Es el responsable de leer la temperatura, presión, humedad, calidad del aire, intensidad de luz, además de enviar los datos y su configuración. Sobre este módulo se centrará el subsistema 2.
    - ***User interface service:*** Es el responsable de manejar los LEDs, botones y otros componentes del UI. Este módulo, en concreto los botones, también se trabajará en el subsistema 2.
    - ***Motion service***: Es el responsable de la lectura del sensor de movimiento de 9 ejes y del acelerómetro.
    - ***Sound service***: Es el responsable del micrófono y del altavoz.
    - ***Battery service***: Es el responsable de manejar el nivel de batería. Esta característica en envía a través de notificaciones cuando el nivel cambia. Su UUID es el 180F.
    - ***DFU service***: Es el responsable de llevar a cabo las actualizaciones del firmware en el Thingy. Su UUID es 0000xxxx-0000-1000-8000-00805f9b34fb
- ***BLE Handler***: el cual maneja todas las comunicaciones Bluetooth y envía los eventos al procesador.
- ***Flash storage***: memoria flash para el almacenamiento de los datos.
- ***Thingy Modules***: Capa que contienen la funcionalidad a alto nivel, controla los drivers de los sensores y los correspondientes servicios BLE.

5.2 IDEs de desarrollo

En esta sección se describen los IDEs usados para desarrollar el código y una explicación del código implementado (las funciones y parámetros críticos en ellas)

5.2.1 Arduino

El programa utilizado para el desarrollo, compilación y actualización de código para la placa Arduino UNO es Arduino IDE (1.8.19). Es un IDE de código abierto bastante sencillo de manejar.

Cuando se abre un nuevo proyecto, hay siempre dos funciones básicas:

- **`setup():`** Es la primera función que se ejecuta y su función es inicializar algunas variables y los módulos necesarios. Esta función solo se ejecuta una vez al principio del programa.
- **`loop()`**: El resto del programa se ejecuta bajo esta función. Se ejecuta en bucle como su nombre indica y en ella se realizará todo el cómputo.

En la interfaz disponemos de dos botones:

- **Verificar**: comprueba que el código escrito no tiene ningún problema sintáctico. Su tarea es compilar el programa, al final de su ejecución, nos indica el espacio que ha usado el *sketch*.
- **Subir**: Se encarga de enviar el binario generado en la etapa de verificación a la placa. Para ello hemos de asegurarnos que bajo la

pestaña Herramientas, en las pestañas “Placa” y “Puerto” está seleccionada la nuestra.

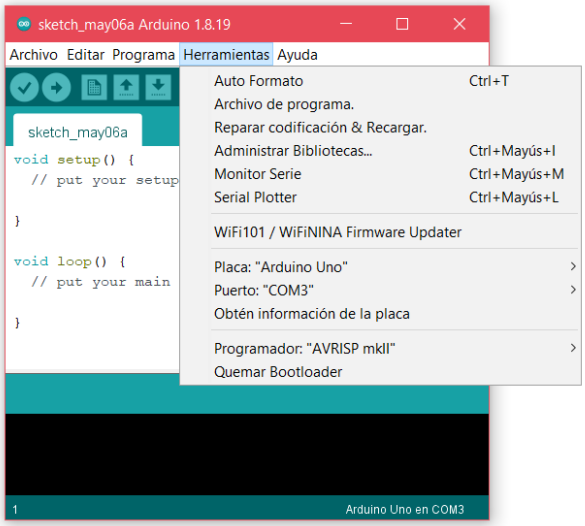

Ilustración 34: IDE Arduino

Una vez descrito la interfaz se procede a describir el código realizado.

5.2.1.1 Sketch Water Sensor

Al ser un proyecto de código abierto, la cantidad de ejemplos y de librerías creadas ayudan a la implementación rápida y eficaz de código, en concreto, la implementación realizada tiene como base el ejemplo “Arduino – Water Sensor” [27]. El código desarrollado está adjuntado en 10.1.1.1 Código Arduino.

Las funciones principales (junto a su descripción) son:

- **`setup()`:** Se encarga de inicializar la comunicación serie indicando el *baud rate* que en este caso es de 115200 y de configurar el pin digital 7 como salida (a través de este se alimentará el sensor). Se opta por hacerlo de esta manera ya que podemos apagar el sensor mientras no obtenemos medidas alargando así su vida útil.

- **`loop()`:** Se encarga de encender el sensor, adquirir la medida a través del pin analógico A5, mapear el valor obtenido a un valor entre 1 y 4 y enviar la cadena de texto siguiente: “*Water value: XX level: YY*” a través de la UART. Esta función cada minuto (12 iteraciones de 5 segundos) llama a `resetAndSendMeasures()`. Las medidas se obtienen con la API `analogRead()`, y el mapeo de la medida entre los niveles se hace mediante `map()`. Esta función solo se ejecutará si no hemos recibido la cadena “stop” a través de RX. Para ello utilizamos la función `checkRX()`.

- **`CheckRX()`**: Esta función se encarga de recibir los datos por la UART. Para ello, cuando tenga datos disponibles, los guardará en un buffer hasta que se encuentre con el carácter de fin de línea. Tras ello

comparará la cadena recibida con las cadenas “stop” y “start”, en el caso de que encuentre la cadena “stop”, una variable global se activará evitando así que se ejecute el resto del programa. Si se recibe “start” se desactivará esta variable permitiendo que el programa continúe.

- **`resetBuckets():`** Se encarga de inicializar y limpiar las variables que contienen las medidas.

- **`resetAndSendMeasures():`** Se encarga de calcular la media obtenida en el último minuto y enviarlo a través de la UART: “*[MEAN] Water value: XX level: YY*”. Una vez computada la media se llama a `resetBuckets().`

Tras compilar el código se obtiene la siguiente salida:

```
El Sketch usa 5266 bytes (16%) del espacio de almacenamiento de programa. El máximo es 32256 bytes.
Las variables Globales usan 413 bytes (20%) de la memoria dinámica, dejando 1635 bytes para las variables locales. El máximo es 2048 bytes.
```

Se observa como el programa no ocupa mucha memoria.

5.2.2 nRF DK IDE

Por otro lado, se ha de desarrollar también el código para el nRF52 DK. Para ello hay numerosos IDEs pero nos centraremos en dos:

- SEGGER Embedded Studio for ARM
- µVision

Cada uno de ellos es usado en un subsistema.
En principio el dispositivo Thingy:52 no cambiará su *firmware*. Por otro lado, el *sniffer* utilizará un código HEX ya precompilado el cual se detallará más en 5.3.3. nRF Connect (Desktop).

5.2.2.1 Código Subsistema 1 PCA10040

En este caso se utilizará el IDE de SEGGER Embedded Studio. Su descarga es gratuita. Para desarrollar las aplicaciones, tal y como se vio en el capítulo 5.1. Arquitectura software nRF, necesitamos el *SDK* y el *SoftDevice*.
Para esta aplicación en concreto se utilizará:

- nRF5_SDK_17.1.0_ddde560
- SoftDevice S132

Primero se necesitará programar el *SoftDevice*. Para ello se utilizará el fichero precompilado que se ubica en:
`..\nRF5_SDK_17.1.0_ddde560\components\softdevice\s132\hex` llamado `s132_nrf52_7.2.0_softdevice.hex.`
Dicho fichero se programará utilizando el software nRF Connect (Desktop).

Una vez tenemos el SoftDevice programado, nuestra aplicación estará basada en su mayoría en el código de ejemplo que se encuentra en el SDK bajo la siguiente ruta:
`..\nRF5_SDK_17.1.0_ddde560\examples\ble_peripheral\ble_app_uart\pca10040\s132`
El código del main.c se puede encontrar en los anexos: 10.1.1.2. Código PCA10040.

En esta carpeta nos encontramos el ejemplo en diversos IDEs, escogemos el que se encuentra bajo la carpeta `/ses/` y abrimos el fichero `ble_app_uart_pca10040_s132.emProject`.

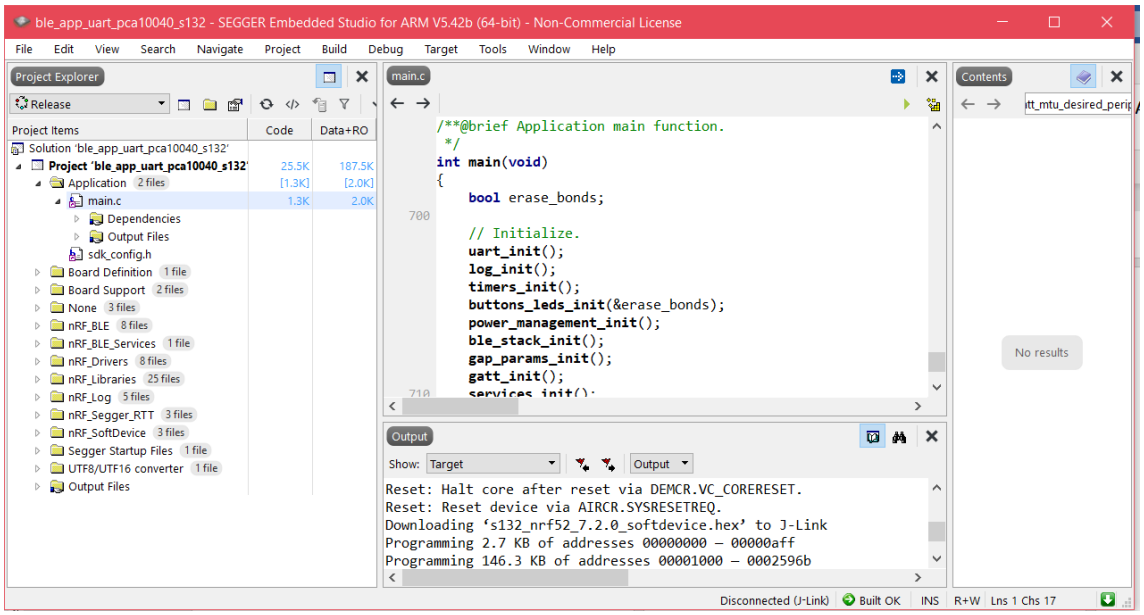

Ilustración 35: IDE SEGGER Embedded Studio

El nombre del dispositivo se indicará a través del `#define DEVICE_NAME` que en este caso será `"Subsistema 1"`.

Nos centramos en las siguientes funciones:

- **`uart_init()`:** mediante esta función se inicializa la UART, para ello, entre otras cosas, hemos de indicarle en la variable de tipo `app_uart_comm_params_t` el `rx_pin_no` y el `tx_pin_no` junto al `baud_rate`.
- **`nus_data_handler()`:** esta función maneja los datos recibidos a través de BLE (el evento es `BLE_NUS_EVT_RX_DATA`), en este caso serán los datos que el teléfono móvil escribe en la `RX Charasteristics`. Este mismo mensaje que recibe lo envía por la UART mediante `app_uart_put()`.
- **`uart_event_handle`**(): Esta función se encarga de manejar los datos que se reciben por la UART. Al igual que en el código del Arduino, los caracteres recibidos se guardan en un buffer hasta que nos encontramos con '\n' o '\r'. Tras ello, envía el mensaje por BLE con la función `ble_nus_data_send()`.

- **advertising_start**(): Función que inicializa el *advertising* ya que en este subsistema recordamos que el nRF52 realizará el papel de periférico, mientras que el teléfono móvil será la central.

El resto de las funciones que se implementa en el código son:

- **timers_init():** Inicializa el módulo de *timer*
- **gap_params_init():** Esta función configura todos los parámetros GAP (Generic Access Profile) necesarios del dispositivo (como la latencia, el timeout…). También establece los permisos y la apariencia.
- **services_init():** Inicializa los servicios que serán utilizados por la aplicación como el *Queued Write Module*, el NUS (Nordic UART Service)
- **conn_params_init():** Función que inicializa el módulo de los parámetros de conexión.
- **ble_stack_init():** Función que inicializa el *SoftDevice*.
- **advertising_init():** Función que inicializa la funcionalidad de *Advertising.*
- **buttons_leds_init():** Función que inicializa los botones y los LEDs.
- **ble_evt_handler():** Función que maneja los eventos BLE, como cuando nos conectamos (`BLE_GAP_EVT_CONNECTED`), nos desconectamos (`BLE_GAP_EVT_DISCONNECTED`)…
- **gatt_evt_handler():** Función que maneja los eventos de la librería GATT
- **gatt_init():** Función que inicializa la librería GATT
- **bsp_event_handler():** Función que inicializa los eventos del módulo BSP (estos eventos se generan al pulsar el botón)
- **idle_state_handle**(): Función que maneja el estado idle, en caso de que no tengamos operaciones pendientes, hace que el módulo duerma.
- **on_conn_params_evt():** Esta función será llamada para todos los eventos del Módulo de Parámetros de Conexión que se pasen a la aplicación.
- **on_adv_evt():** Función que se llama para manejar los eventos de *advertising*.
- **sleep_mode_enter():** Pone el chip en modo bajo consumo, para despertarlo se utilizará el botón de wakeup.
- **main():** Es la función que se ejecuta cuando el programa arranca. Esta función llama a todas aquellas funcionas que inicializan los módulos. Una vez todo está inicializado se procede a llamar a la función `advertising_start()`. Si nos conectamos a un dispositivo, llegará el evento BLE_GAP_EVT_CONNECTED. En este caso vemos como el nRF actúa como periférico porque está realizando el *advertising* y es el teléfono el que realiza el *scan*.
  Tras activar las CCCDs a través del dispositivo móvil, ya nos empezarán a llegar mensajes a través de la UART (`uart_event_handle()`) y estos serán redirigidos a la interfaz BLE para ser enviados. Si el teléfono envía una cadena de texto, esta será tratada en con el evento BLE_NUS_EVT_RX_DATA y será enviada por la UART.

5.2.2.2 Código Subsistema 2 PCA10040

En este caso se utilizará el IDE de µVision. Su descarga es gratuita. Igual que en el caso anterior necesitamos el *SDK* y el *SoftDevice*. Para esta aplicación en concreto se utilizará:

- SDK 15.0.0
- SoftDevice S132 6.0.0

Esta aplicación está basada en el código *nRF52xx-to-Thingy-52* [28] el cual a su vez está basado en el ejemplo *Bluetooth 5 multi link demo* [29].

Para poder utilizar la aplicación hemos de integrarla en el SDK siguiendo estos pasos:

1. Añadir el directorio `ble_app_thingy_to_52_dk_c` a la siguiente ruta `<SDK>\examples\ble_central\.`
2. Mover las carpetas en la ruta `src` hacia la siguiente ruta `<SDK path>\components\ble\ble_services\`. Bajo este directorio tenemos dos subcarpetas: ble_tes_c que incluye el Thingy Environment service y ble_tbs_c que incluye el código original del LED button service junto a ciertos cambios que definen el UUID específico del Thingy.

Una vez hemos añadido el proyecto, pasamos a modificar ciertos valores:

1. Hemos de realizar cambios en el main.c para poder integrarlo con el Thingy:52 del que dispongamos. Hemos de cambiar el valor de la variable `m_target_periph_name[]` para que contenga el *device name* del Thingy:52, que en este caso es “Sebas”.
2. Hemos de añadir el UUID del Thingy en los ficheros src:

```
#define LBS_UUID_BASE        {0x42, 0x00, 0x74, 0xA9, 0xFF, 0x52, 0x10, 0x9B,
0x33, 0x49, 0x35, 0x9B, 0x00, 0x00, 0x68, 0xEF}
#define LBS_UUID_SERVICE     0x0300
#define LBS_UUID_BUTTON_CHAR 0x0302
#define LBS_UUID_LED_CHAR    0x0301
```

3. Hemos de añadir los siguientes defines en el fichero sdk_config.h

```
// <q> BLE_TES_C_ENABLED  - ble_tes_c - Nordic Thingy Environment Service
Client

#ifndef BLE_TES_C_ENABLED
#define BLE_TES_C_ENABLED 1
#endif

// <o> BLE_TES_C_BLE_OBSERVER_PRIO
// <i> Priority with which BLE events are dispatched to the Thingy
Environment Service Client.

#ifndef BLE_TES_C_BLE_OBSERVER_PRIO
#define BLE_TES_C_BLE_OBSERVER_PRIO 2
#endif

// <q> BLE_TBS_C_ENABLED  - ble_tes_c - Nordic Thingy Button Service Client

#ifndef BLE_TBS_C_ENABLED
```

```
#define BLE_TBS_C_ENABLED 1
#endif

// <o> BLE_TBS_C_BLE_OBSERVER_PRIO
// <i> Priority with which BLE events are dispatched to the Thingy Button
Service Client.

#ifndef BLE_TBS_C_BLE_OBSERVER_PRIO
#define BLE_TBS_C_BLE_OBSERVER_PRIO 2
#endif
```

Una vez hechos los cambios procedemos a abrir el proyecto utilizando el IDE µVision.

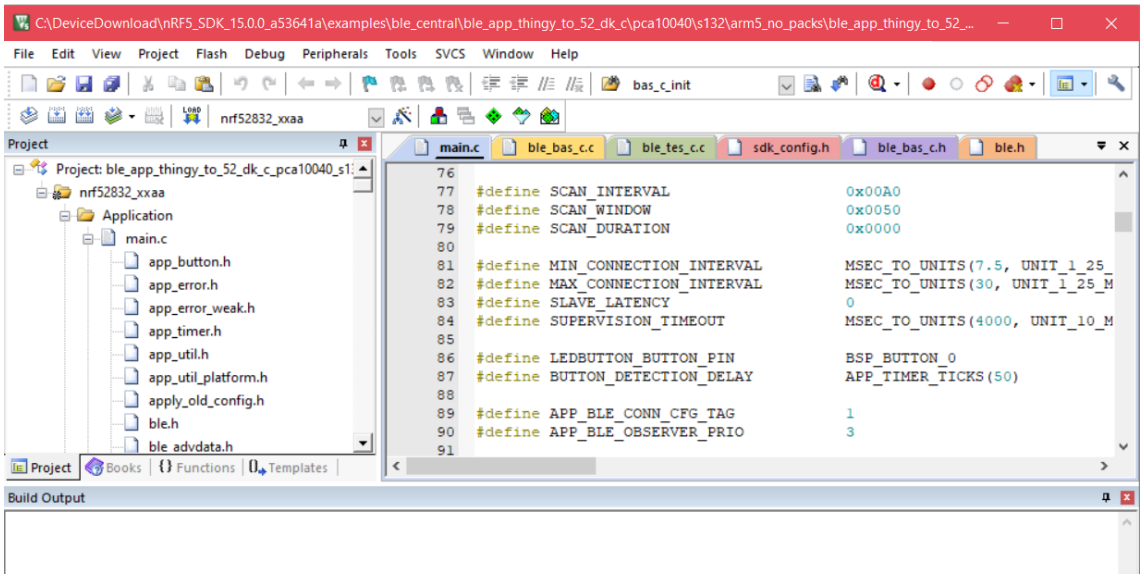

Ilustración 36: IDE µVision

Las funciones que se desarrollan en el main.c y que se encargan de manejar los datos son:

- **`tbs_c_evt_handler()`**: Función que maneja los eventos que vienen a través del módulo *Thingy Button Central*. Los eventos que podemos recibir son:
  - **BLE_TBS_C_EVT_DISCOVERY_COMPLETE:** Cuando se descubre el servicio Thingy Button. Una vez descubierto, se pasa a asignar los *handlers* y a habilitar las notificaciones a través de la función ble_tbs_c_button_notif_enable().
  - **BLE_TBS_C_EVT_BUTTON_NOTIFICATION**: Cuando se recibe una notificación. Esto se generará al pulsar el botón en el Thingy:52. La acción que se realiza es Encender el LED. Es decir, el LED se encenderá siempre y cuando el botón del Thingy:52 esté pulsado.
- **`tes_c_evt_handler()`**: Función que maneja los eventos que vienen a través del módulo *Thingy Environment Central*. Los eventos que podemos recibir son:
  - **`BLE_TES_C_EVT_DISCOVERY_COMPLETE:`** Cuando se descubre el servicio *Thingy Environment*. Una vez descubierto, se pasa a

asignar los *handlers* y a habilitar las notificaciones a través de la función ble_tes_c_XXXXXXXXX_notif_enable ().

- **BLE_TES_C_EVT_TEMPERATURE_NOTIFICATION:** Cuando se recibe una notificación de la característica de temperatura. La acción que se realiza es enviar por la UART la temperatura mediante la siguiente cadena “Got temperature: XX,XX”
- **BLE_TES_C_EVT_PRESSURE_NOTIFICATION**: Cuando se recibe una notificación de la característica de presión. La acción que se realiza es enviar por la UART la presión mediante la siguiente cadena “Got pressure: XX,XX”
- **BLE_TES_C_EVT_HUMIDITY_NOTIFICATION**: Cuando se recibe una notificación de la característica de humedad. La acción que se realiza es enviar por la UART la humedad mediante la siguiente cadena “Got humidity: XX”
- **BLE_TES_C_EVT_GAS_NOTIFICATION**: Cuando se recibe una notificación de la característica de gas. La acción que se realiza es enviar por la UART el valor mediante la siguiente cadena “Got C02: XX”, “Got organic componentes: XX”
- **BLE_TES_C_EVT_COLOR_NOTIFICATION**: Cuando se recibe una notificación de la característica de color. La acción que se realiza es enviar por la UART el valor mediante la siguiente cadena “Got color: RXX, GXX, BXX, CXX”

Otras funciones del código son:

- **scan_start()**: Función que inicializa el escaneo basado en los parámetros definidos en m_scan_params. Mientras se esté escaneando el LED1 estará encendido y el LED2 apagado. En este subsistema, el nRF52 actúa como central mientras que Thingy:52 actuará como periférico, por ello el *advertising* será realizado por Thingy:52 y el *scan* por el nRF52.
- **on_adv_report()**: Función que maneja los eventos de *advertising* del *SoftDevice.*
- **tbs_c_init()**: Función que inicializa el cliente *Thingy Butto*n.
- **tes_c_init()**: Función que inicializa el cliente *Thingy Environment*
- **db_disc_handler()**: Función que maneja los eventos del módulo de descubrimiento de la base de datos. Dependiendo de los UUIDs que se descubran, esta función debería reenviar los eventos a sus respectivos servicios. Por lo que se deben llamar a las funciones ble_XXX_ on_db_disc_evt()
- **db_discovery_init()**: Función que inicializa la base de datos del descubrimiento.
- **main()**: Función que es llamada al inicio del código. Igual que en el anterior ejemplo, lo primero que hace es inicializar los módulos. Cuando la ejecución comienza empieza a escanear buscando el dispositivo que se le ha indicado, en este caso “*Sebas*”. Al contrario que en caso anterior, en este caso el nRF actúa como central ya que es él quien escanea buscando al dispositivo. Cuando lo encuentre pasará a descubrir las características y a activar las notificaciones, posteriormente el Thingy pasará a enviarle sus parámetros.

Sobre este código base se han añadido funcionalidad nueva para incorporar también la lectura de la batería restante (ya que en en el subsistema el dispositivo utilizará su batería interna). Necesitamos incluir dos ficheros: `ble_bas_c.c` y `ble_bas_c.h` que se encuentran en la ruta: `\ble\ble_services\ble_bas_c`. Además, se implementan nuevas funciones:

- **`bas_c_evt_handler()`**: Función que maneja los eventos que vienen a través del módulo *Thingy Battery Central*. Los eventos que podemos recibir son:
    - **`BLE_BAS_C_EVT_DISCOVERY_COMPLETE`**: Cuando se descubre el servicio *Thingy Battery*. Una vez descubierto, se pasa a asignar los *handlers* y a habilitar las notificaciones a través de la función ble_bas_c_bl_notif_enable().
    - **`BLE_BAS_C_EVT_BATT_NOTIFICATION`**: Cuando se recibe una notificación. Esto se generará solamente si el nivel de batería cambia tal y como dice la documentación. Una vez recibido se mandará la siguiente cadena por UART: "Got Battery Level: X"
- **`bas_c_init()`**: Inicializa el cliente *Thingy Battery*.

La programación de este código consta de dos pasos:

1. Programar el *SoftDevice* mediante nRF Connect (for Desktop)

2. Compilar y cargar el binario mediante la interfaz de usuario

Ilustración 37: Interfez usuario IDE µVision para compilar y flashear

5.3 Softwares complementarios

Además de los IDEs detallados en el apartado anterior, es necesario disponer de otros programas complementarios que nos ayudarán en el desarrollo del código y a la realización de las pruebas para la verificación de los sistemas.

5.3.1 Putty

PuTTY es un cliente SSH, Telnet, rlogin, y TCP raw con licencia libre. El uso que se le dará en nuestro proyecto es para recibir los datos en el Subsistema 2, ya que el nRF52 enviará los datos recibidos del Thingy:52 mediante esta interfaz.

Para ello en la ventana del programa se ha de indicar simplemente el puerto COM en el que está conectado el dispositivo y el baud rate (115200)

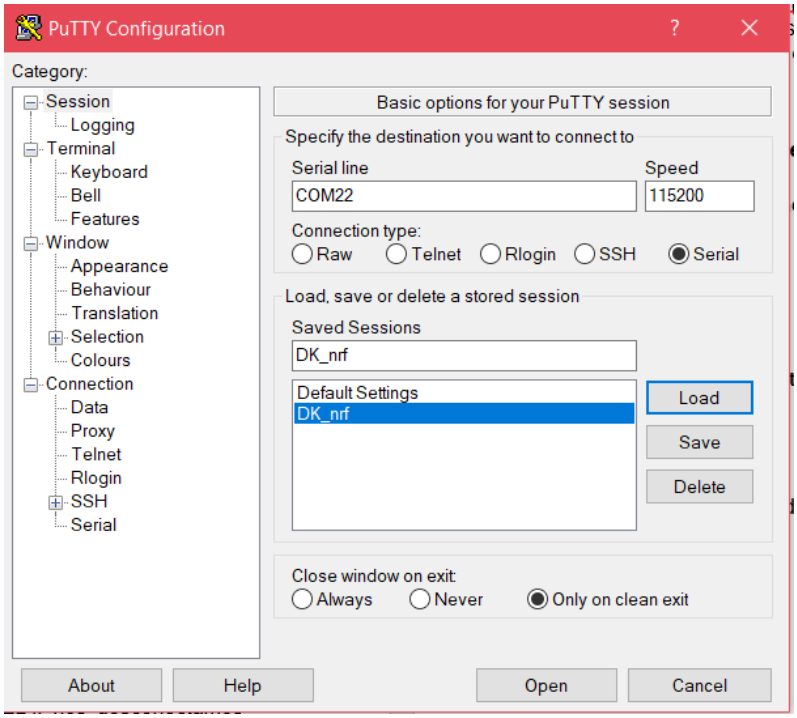

Ilustración 38: Interfaz PuTTY

5.3.2 Wireshark

Wireshark es el analizador de protocolos más usado. Es un programa gratuito muy utilizado en la enseñanza. Decodifica una gran cantidad de protocolos, entre ellos BLE. Para utilizarlo necesitamos un Sniffer el cual se encargará de redirigir el tráfico BLE al puerto del ordenador, en nuestro caso esa será la misión del PCA-10059.

Para poder utilizar Wireshark junto a BLE se deben seguir ciertos pasos que se detallan en el Infocenter de Nordic en la sección *Installing the nRF Sniffer capture tool* [30]. Además, se ha de añadir el perfil tal y como se detalla en la sección *Adding a Wireshark profile for the nRF Sniffer* [31].

Un requisito fundamental es que nuestro *Sniffer* soporte BLE. Además, debemos *flashear* un binario precompilado para que nuestro dongle actúe como *Sniffer*. Para ello se siguen los pasos en la sección *Programming the nRF Sniffer firmware* [32]

Una vez se inicia el programa, lo primero será seleccionar el puerto por el que vamos a captura, el cual se corresponde con el puerto del Sniffer.

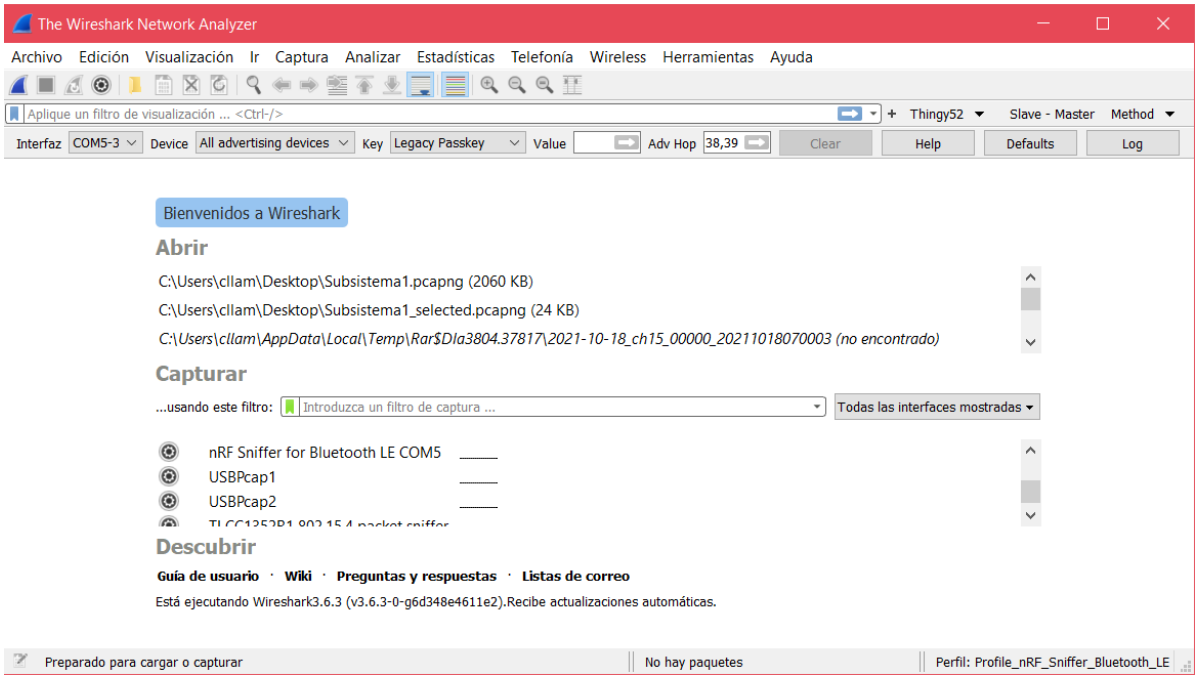

Ilustración 39: Pantalla inicial Wireshark

Se puede iniciar la captura indicando un filtro, el más común sería filtrar por RSSI, ya que las comunicaciones BLE se dan en muchos dispositivos y la captura se llenaría de paquetes.

Una vez seleccionada la interfaz comenzamos a ver tráfico BLE.

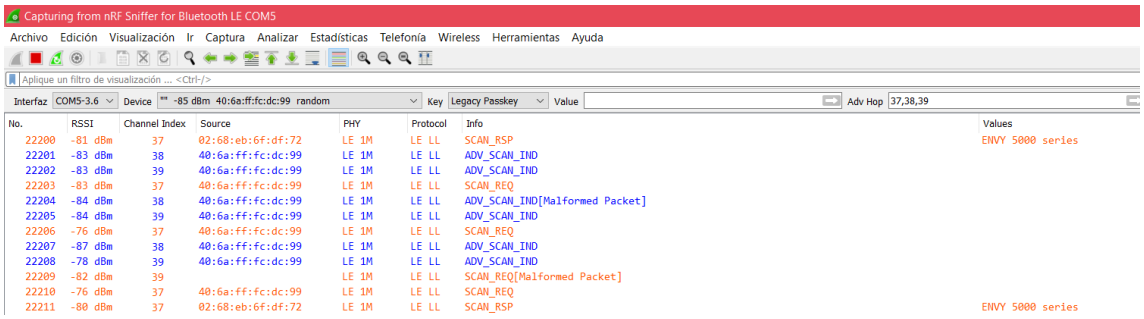

Ilustración 40: Interfaz Wireshark durante la captura

Cuando se establece la comunicación se puede utilizar en el *Toolbox* la pestaña *Device* y seleccionar al dispositivo, entonces se comenzará a ver el tráfico solo de ese dispositivo.

Durante la captura se pueden utilizar filtros que nos permitan seleccionar ciertos tipos de paquetes. El listado de filtros se encuentra en la documentación de Wireshark [33]

Otra herramienta que ayuda a visualizar correctamente las comunicaciones son las reglas de coloreado. Estas permiten relacionar un filtro con un color o un sombreado, mejorando así la interpretación visual de la traza. En este proyecto se han definido las siguientes:

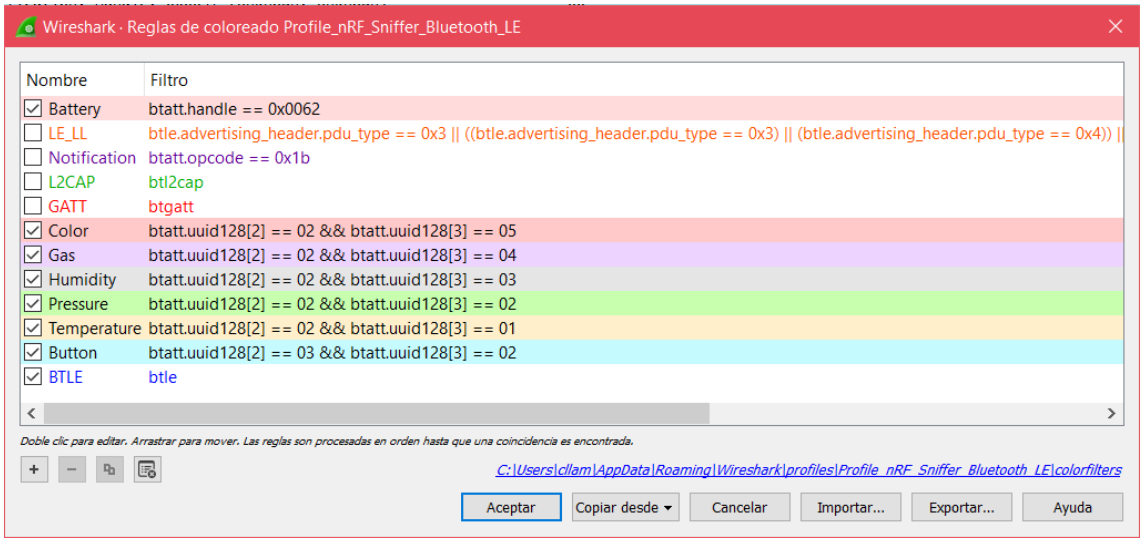

Ilustración 41: Reglas coloreado Wireshark

En el capítulo 6. Resultados obtenidos se detallarán algunas trazas obtenidas.

5.3.3 nRF Connect (Desktop)

nRF Connect es un marco de herramientas multiplataforma gratuita para ayudar al desarrollo en dispositivos nRF. Contiene muchas aplicaciones para probar, supervisar, medir, optimizar y programar sus aplicaciones [34].

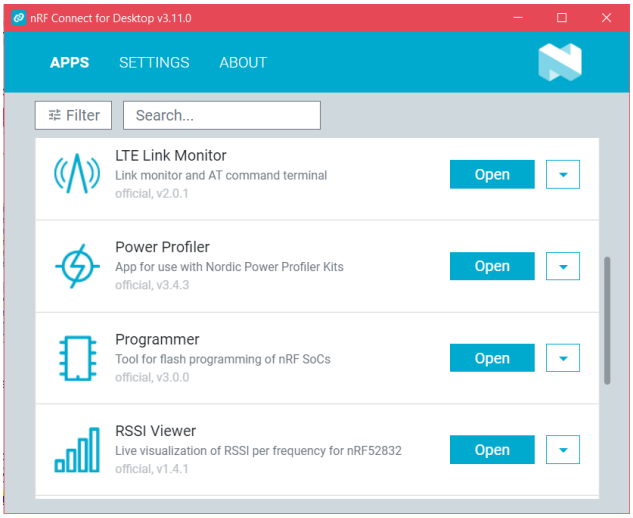

Ilustración 42: Pantalla inicial nRF Connect (Desktop)

La herramienta que usaremos será *Programmer.* Esta herramienta ha sido utilizada para dos funciones:

- Escribir el binario precompilado y linkado al Dongle USB para que actúe como Sniffer.
- Escribir el binario precompilado y linkado en el nRF52 que contiene el *SoftDevice.*

Para ello seleccionamos la aplicación con Open y se nos abrirá un menú como este:

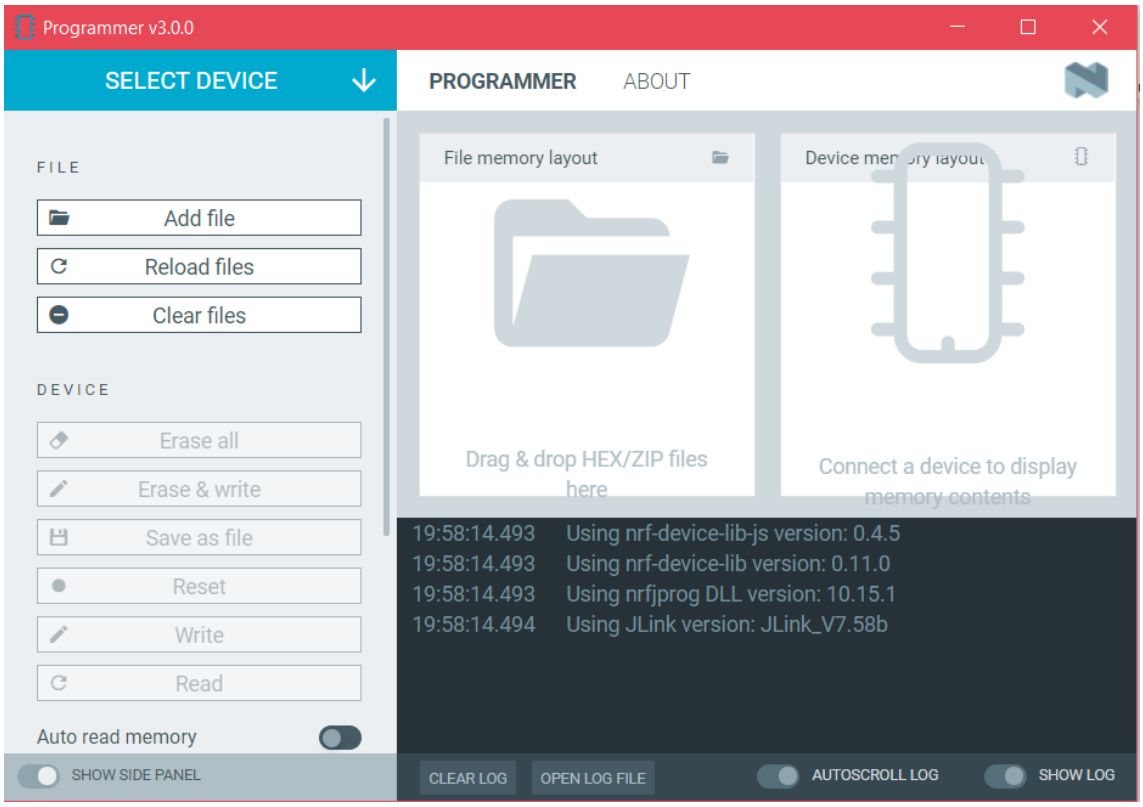

Ilustración 43: Programmer v3.0.0

Donde se debe seleccionar el dispositivo y arrastrar el binario en hexadecimal. Después se seleccionará la función Erase & Write. En la consola se podrá ver el desarrollo del proceso.

5.3.4 nRF Connect (app)

Esta aplicación es gratuita para iOS y Android y nos permite escanear y conectarnos a dispositivos Bluetooth. Se usará para visualizar las características de los dispositivos y poder acceder y modificar ciertos campos.

Su uso se detallará en el capítulo 6. Resultados obtenidos.

5.4 Diagramas de los sistemas

5.4.1 Subsistema 1

5.4.1.1 Arduino

El código del Arduino se localiza principalmente dentro de la función `loop()`.Se muestra también un diagrama de la función `checkRX()` dado que es la que se encarga de recibir los datos enviados por el nRF52 a través de la UART.

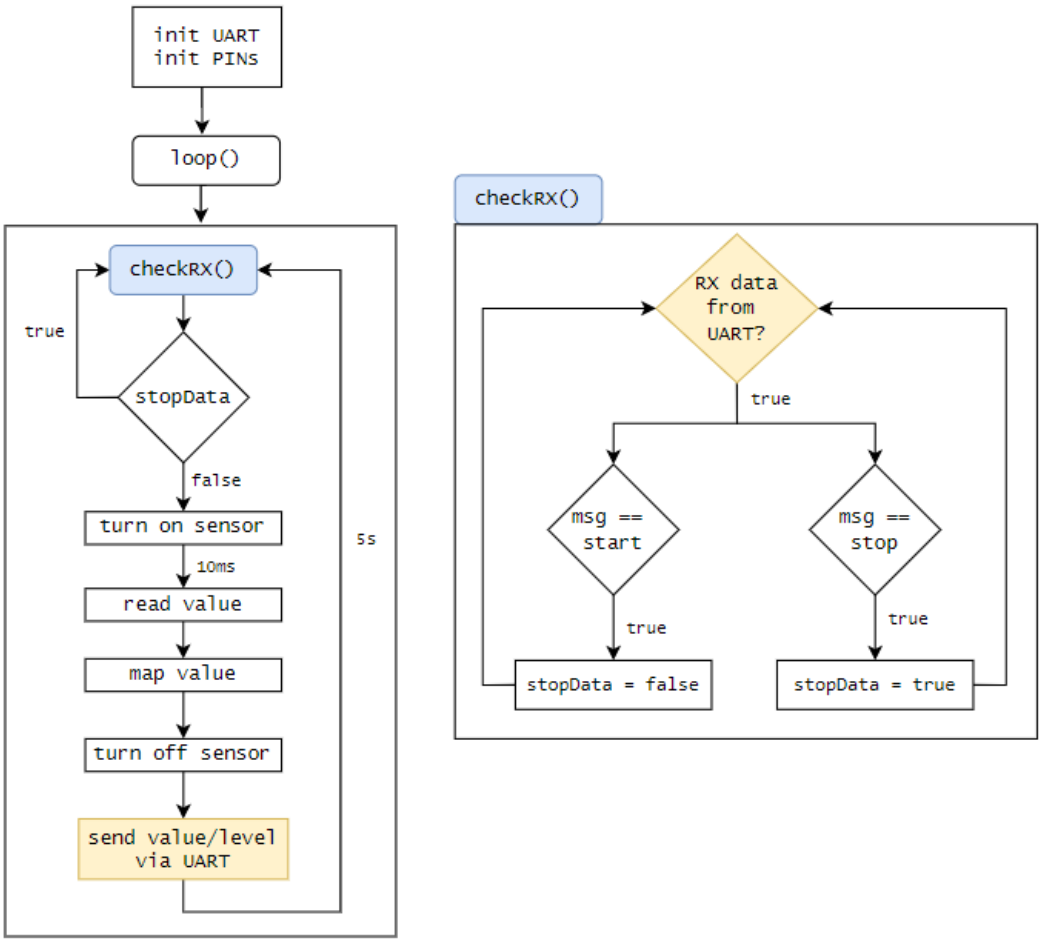

Ilustración 44: Diagrama código Arduino Subsistema 1

5.4.1.2 PCA10040

El código es complejo por lo que se crearan subdiagramas. El primero recogería las acciones que realiza en main.c, que es: inicializar todos los módulos, servicios, *stacks,* comunicación…, comenzar el *advertising* y esperar en un bucle infinito (`for(;;)`)

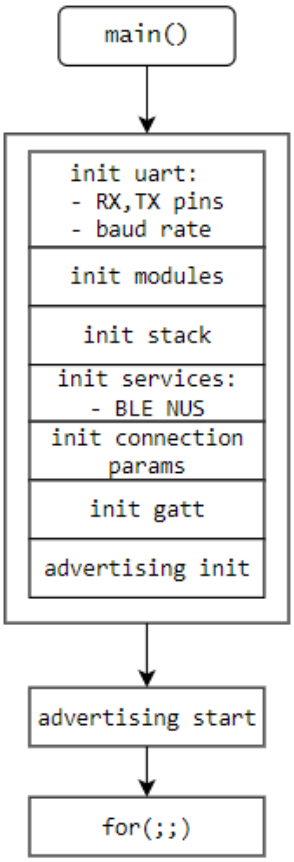

Ilustración 45: Diagrama main() PCA10040 Subsistema 1

Luego por otra parte tenemos dos manejadores de eventos importantes: el que maneja los datos recibidos/enviados por UART, y el que maneja los datos enviados/recibidos por BLE.

Estas funciones son las siguientes junto a una breve descripción:

- **`nus_data_handler():`** esta función maneja los datos recibidos a través de BLE (el evento es `BLE_NUS_EVT_RX_DATA`), en este caso serán los datos que el teléfono móvil escribe en la `RX Charasteristics`. Este mismo mensaje que recibe lo envía por la UART mediante `app_uart_put().`
- **`uart_event_handle`**(): Esta función se encarga de manejar los datos que se reciben por la UART. Al igual que en el código del Arduino, los caracteres recibidos se guardan en un buffer hasta que nos encontramos con '\n' o '\r'. Tras ello, envía el mensaje por BLE con la función `ble_nus_data_send().`

Sus diagramas son los siguientes:

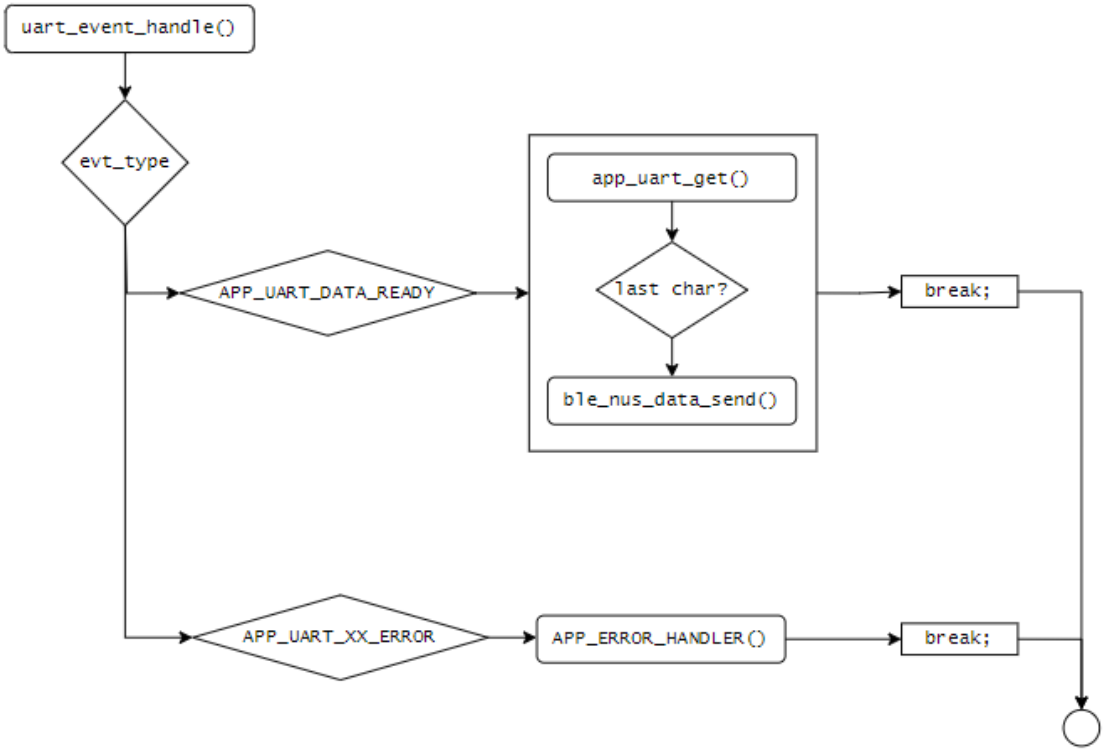

Ilustración 47: Diagrama uart_event_handle() PCA10040 Subsistema 1

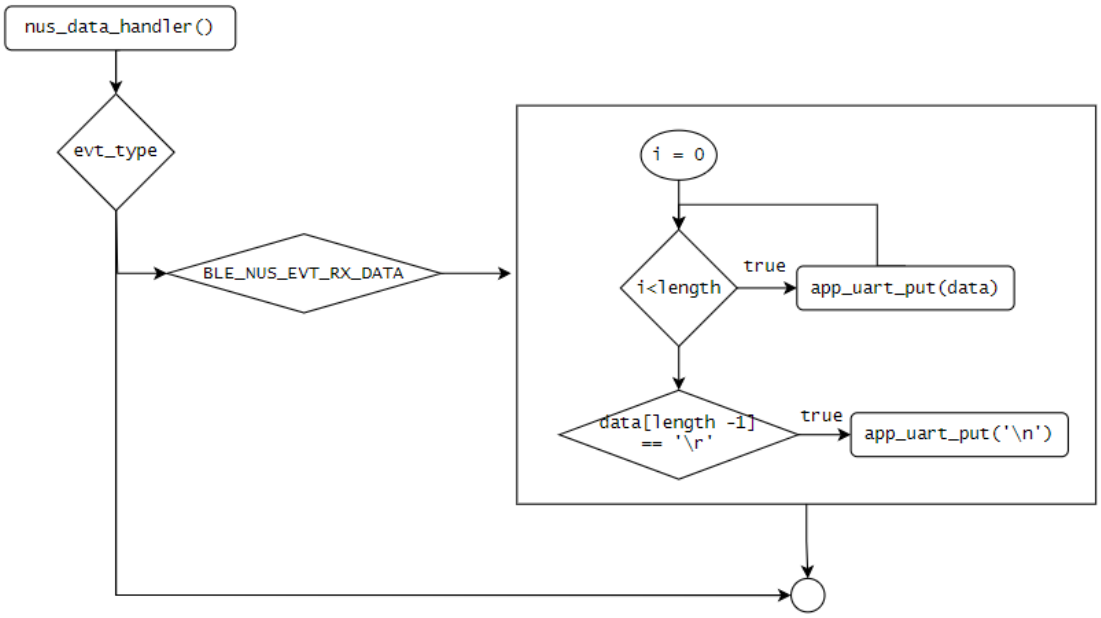

Ilustración 46: Diagrama nus_data_handler() PCA10040 Subsistema 1

5.4.1.3 Sistema completo

Los siguientes diagramas muestran la interconexión entre las funciones del código de los dispositivos.

En esta primera ilustración se muestra el inicio de conexión de los dispositivos y un primer envío periódico de mensajes donde entran en juego los diagramas mostrados anteriormente.

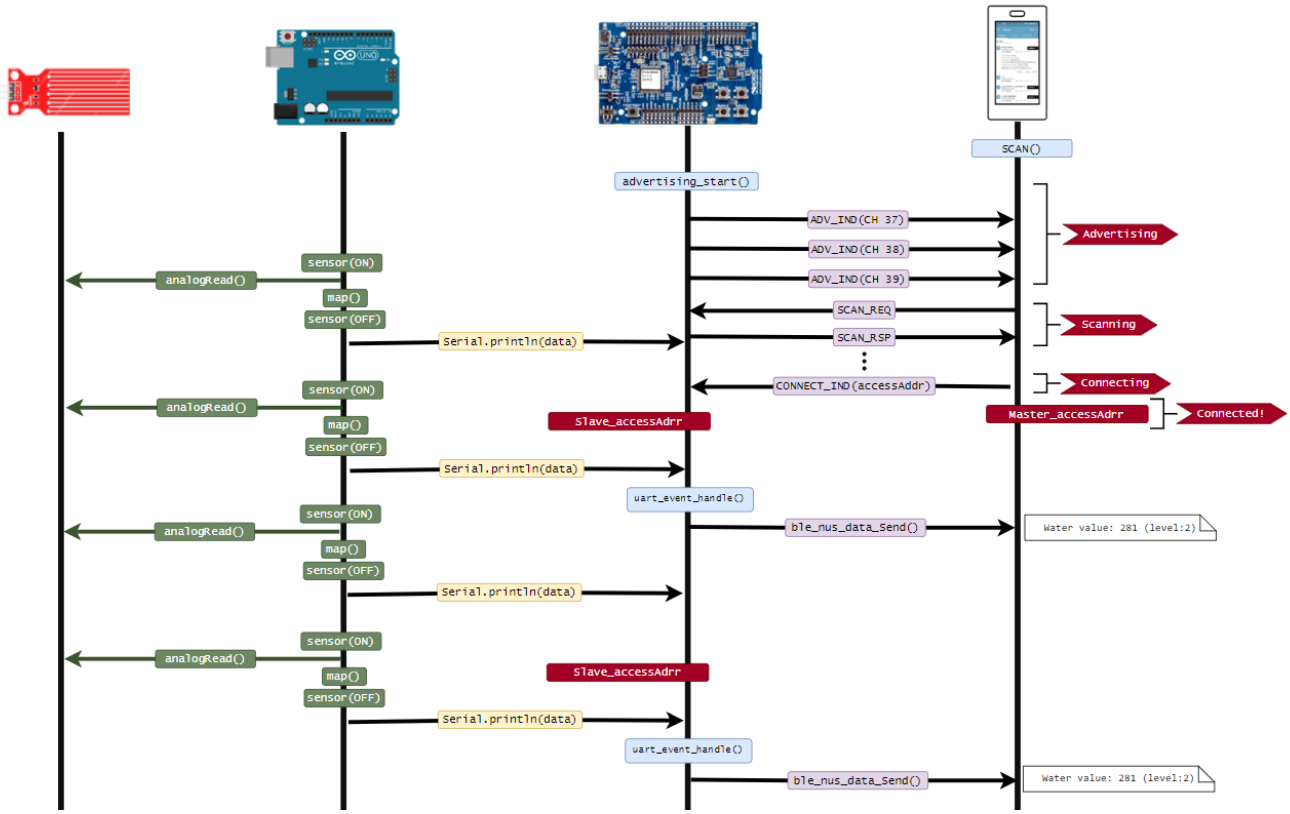

Ilustración 48: Diagrama 1 Subsistema 1

El diagrama siguiente muestra la interconexión cuando se manda la cadena "*stop*" y "*start*" desde el teléfono móvil utilizando la aplicación nRF Connect.

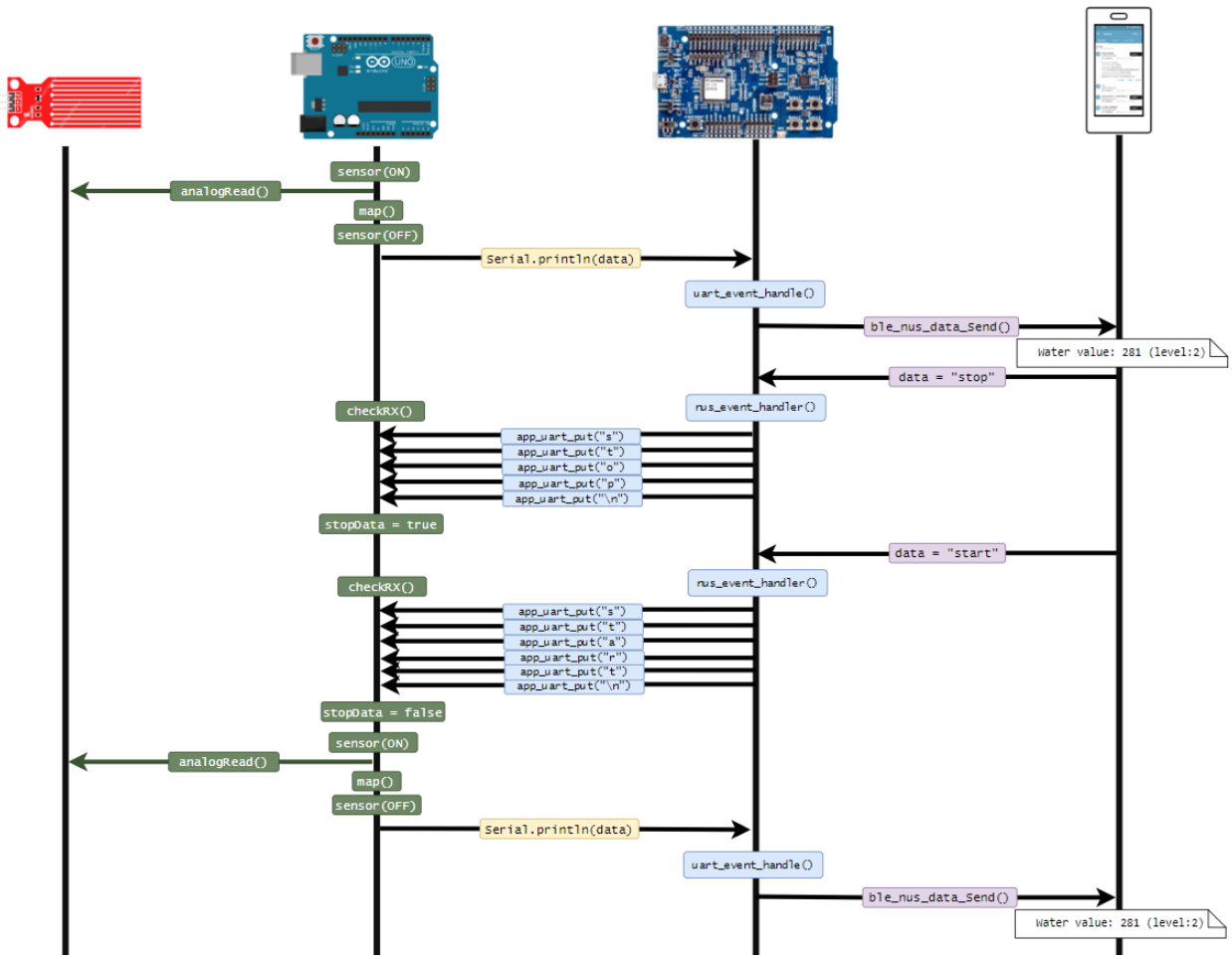

Ilustración 49: Diagrama 2 Subsistema 1

5.4.2 Subsistema 2

5.4.2.1 PCA10040

Al igual que en caso anterior, el código es complejo por lo que se crearan subdiagramas. El primero recogería las acciones que realiza en main.c, que es: inicializar todos los módulos, servicios, *stacks,* comunicación…, comenzar el *scanning* y esperar en un bucle infinito (`for(;;)`)

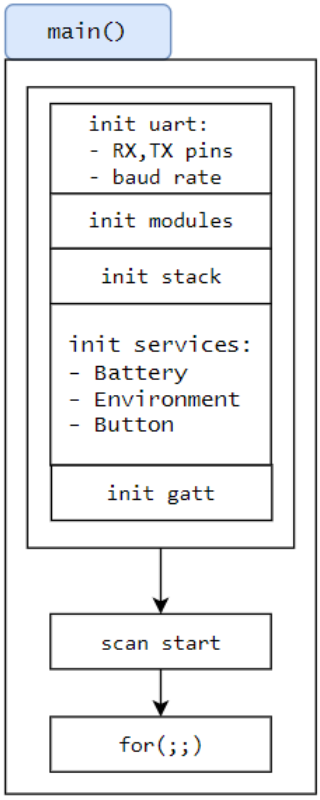

Ilustración 50: Diagrama main() PCA10040 Subsistema 2

Las funciones principales que manejarán la conexión y el manejo de los datos se mostrarán en los siguientes diagramas. Las funciones son:

- **`ble_evt_handler()`**: la cual maneja los eventos BLE tales como conexión, desconexión, *advertising indication…*
- **`bas_c_evt_handler()`**: que maneja los eventos del *Thingy battery module.*
- **`tes_c_evt_handler()`**: que maneja los eventos del *Thingy Environment central module.*
- **`tbs_c_evt_handler()`**: que maneja los eventos del *Thingy Button central module.*

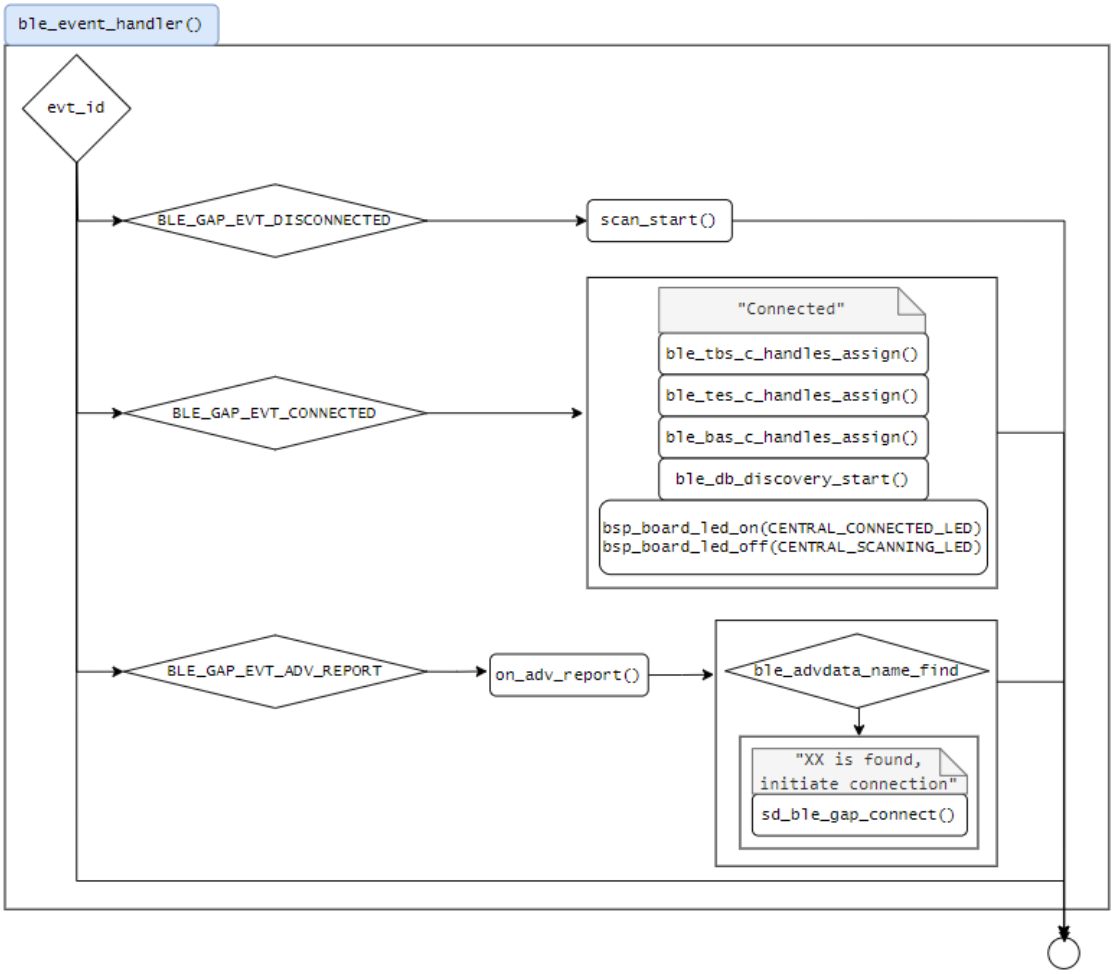

Ilustración 52: Diagrama ble_event_handler() PCA10040 Subsistema 2

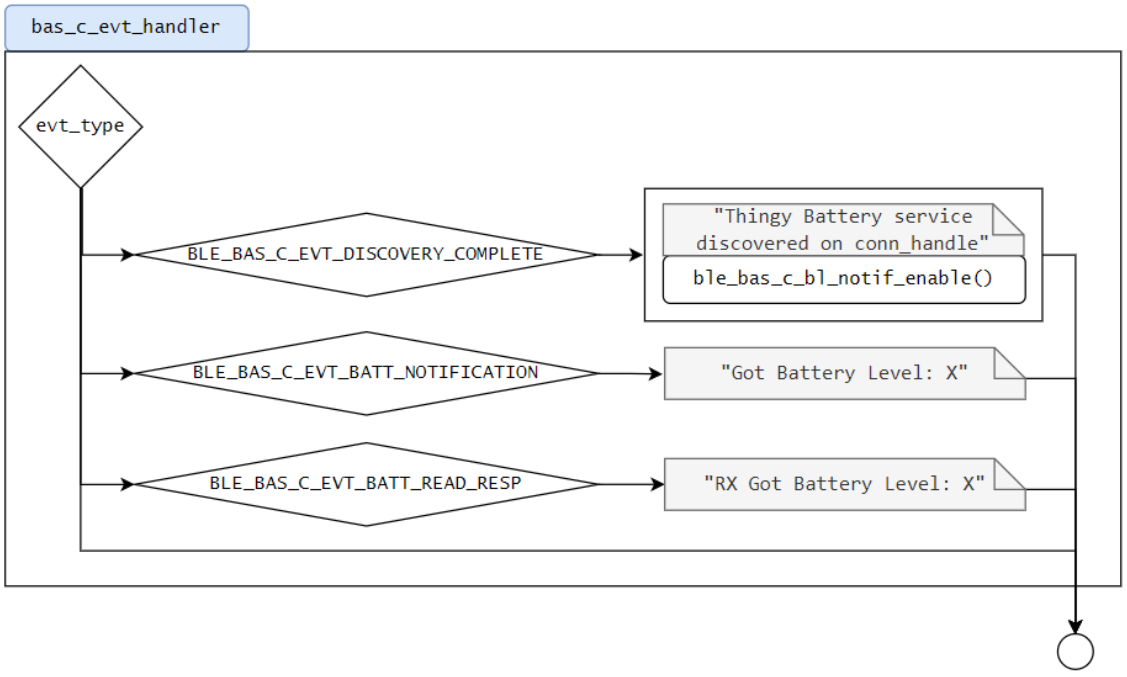

Ilustración 51: Diagrama bas_c_evt_handler() PCA10040 Subsistema 2

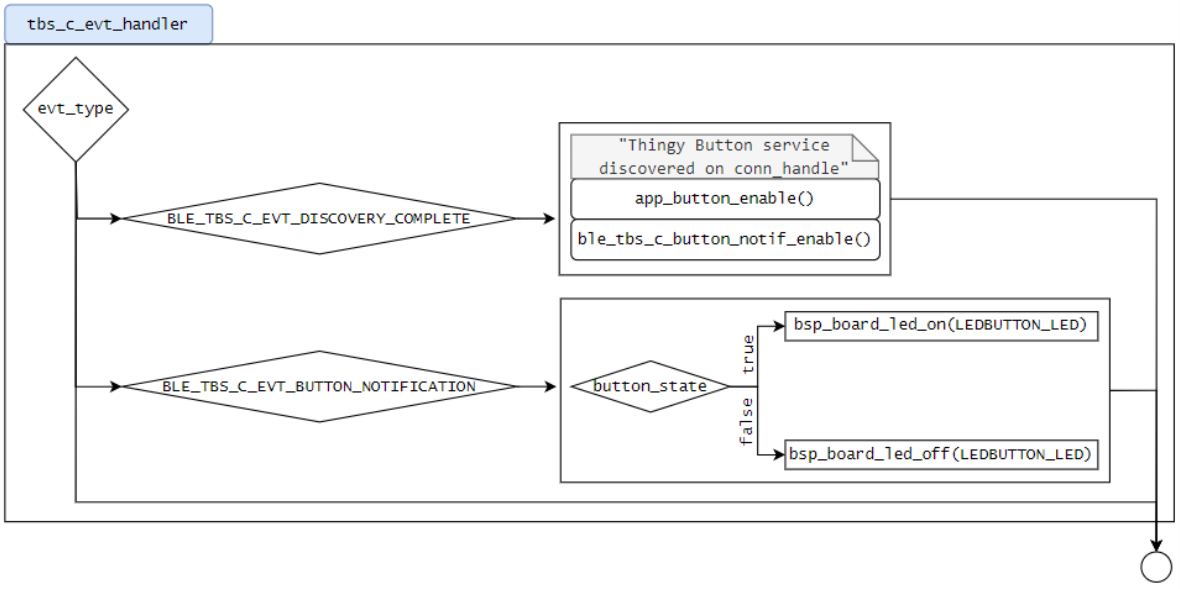

Ilustración 54: Diagrama tbs_c_evt_handler() PCA10040 Subsistema 2

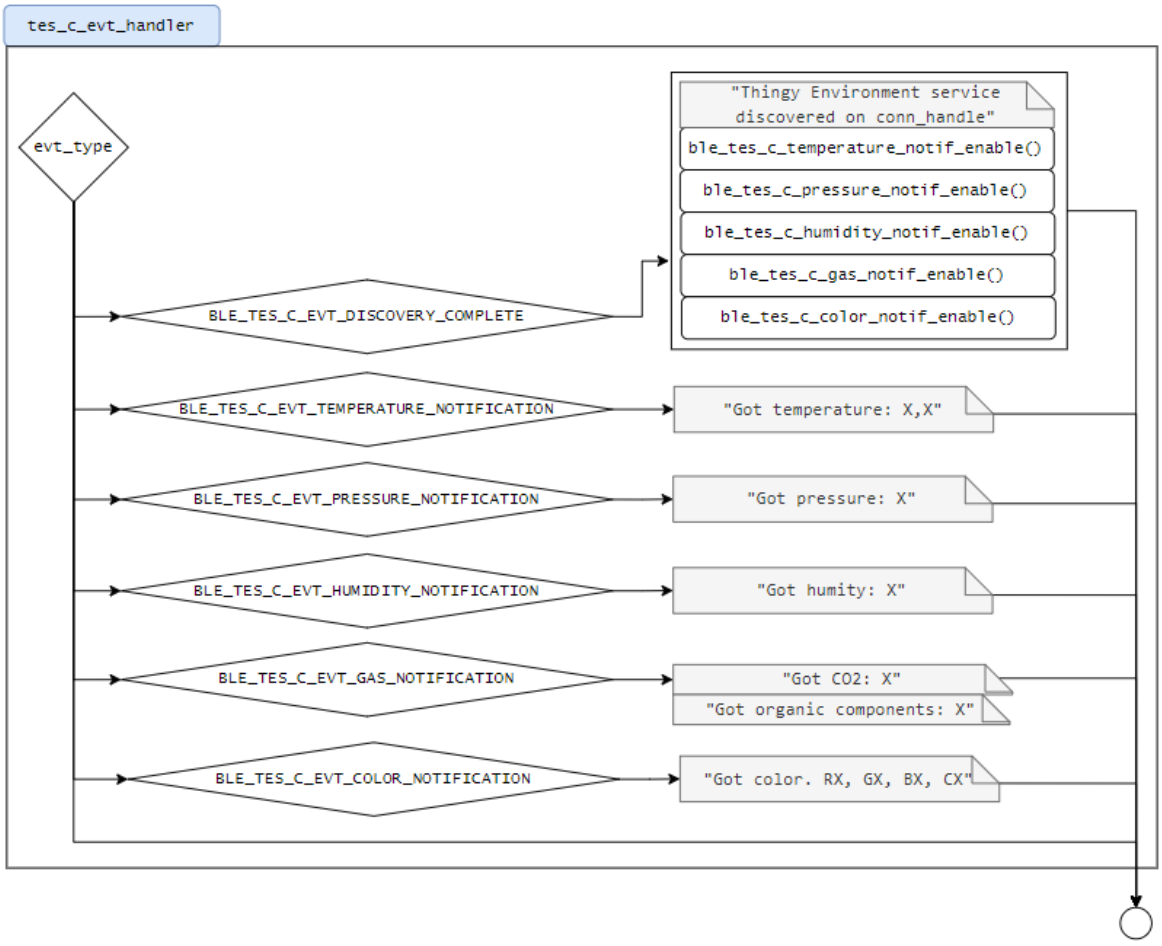

Ilustración 53: Diagrama tes_c_evt_handler() PCA10040 Subsistema 2

5.4.2.2. Sistema completo

Este diagrama muestra la interconexión de los dispositivos y las funciones principales que intervienen en la comunicación.

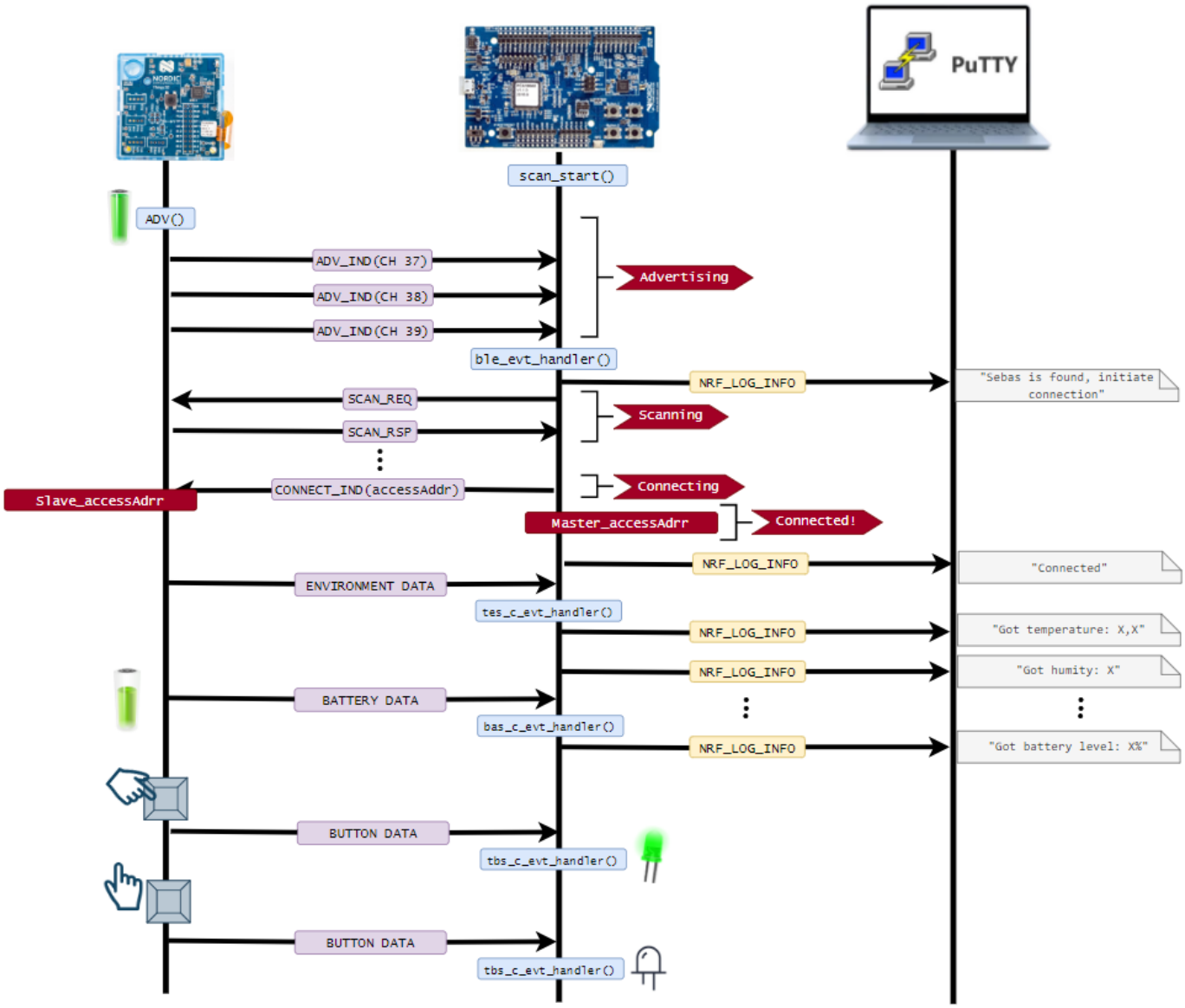

Ilustración 55: Diagrama Subsistema 2

6. Resultados obtenidos

En este capítulo se detallarán las pruebas realizadas en cada subsistema y los resultados obtenidos acompañados de las evidencias.

6.1 Subsistema 1

El sistema completo se muestra en la siguiente figura:

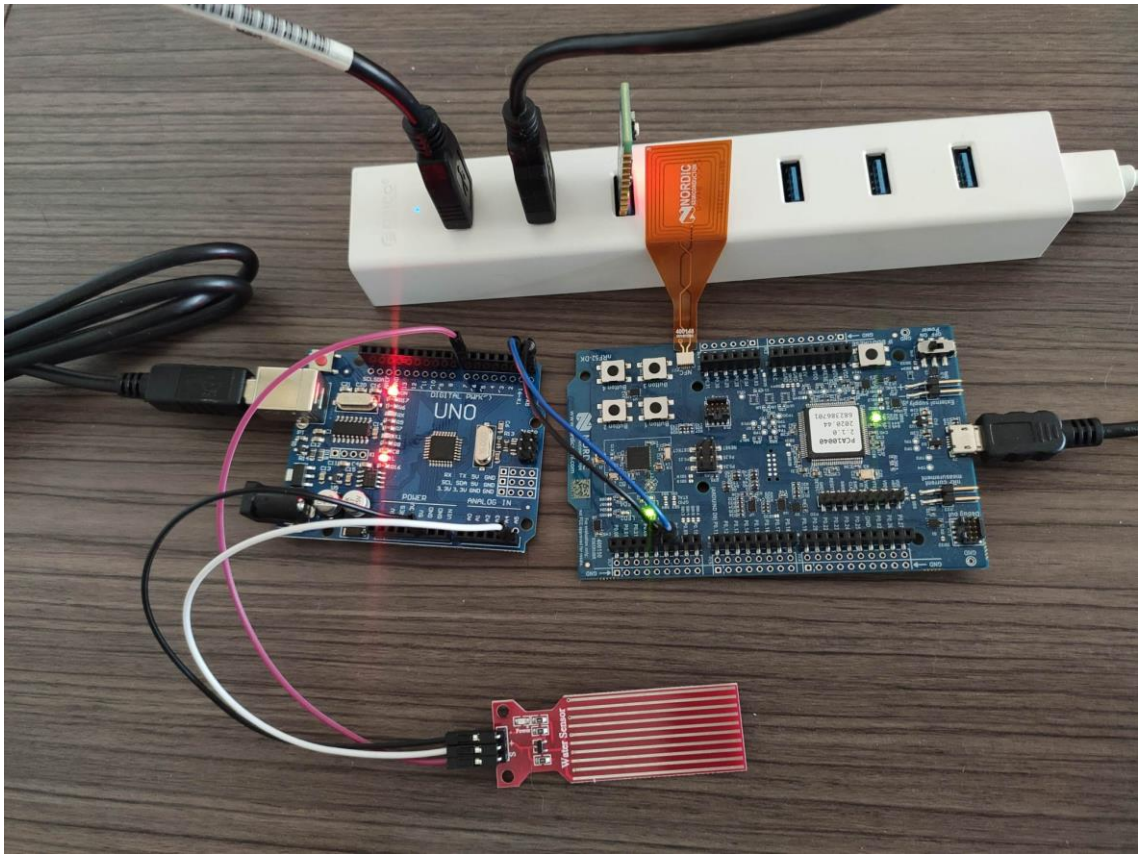

Ilustración 56: Subsistema 1 Interconectado

Ambos dispositivos se alimentan a través del HUB-USB. Entre ellos, se interconectan las UARTs. El sensor de agua es alimentado a través del Arduino UNO.

Se llevan a cabo dos pruebas diferentes para verificar el sistema.

6.1.1 Prueba TX Characteristics

En esta prueba se busca recibir las medidas obtenidas por el sensor, para ello se siguen los siguientes pasos:

1. Se programan ambos dispositivos (Arduino UNO y nRF52 DK)
2. Se conecta el sniffer y se inicia Wireshark para capturar las trazas
3. Se abre la aplicación nRF Connect en el dispositivo móvil

4. Se habilitan los CCCDs en el dispositivo ya que por defecto están deshabilitados
5. Aparece el servicio: Nordic UART Service
6. Dentro de él se puede observar:
    a. ***RX Characteristics***: Es usado para enviar datos hacia el Arduino
    b. ***TX Characteristics*** junto con ***Descriptor***: Es usado para recibir los datos del Arduino
7. Se selecciona el log del sistema, para ello se accede al menú de opciones que está junto a DISCONNECT en la esquina superior derecha y se pulsa sobre Show log (la última opción); ahí se podrá ver el intercambio de mensajes, como, por ejemplo:

```
18:08:51.791 Notification received from 6e400003-b5a3-f393-e0a9-
e50e24dcca9e, value: (0x) 57-61-74-65-72-20-76-61-6C-75-65-3A-20-32-
38-31-20-28-6C-65-76-65-6C-3A-32-29-0D
18:08:51.791 "Water value: 281 (level:2)
```

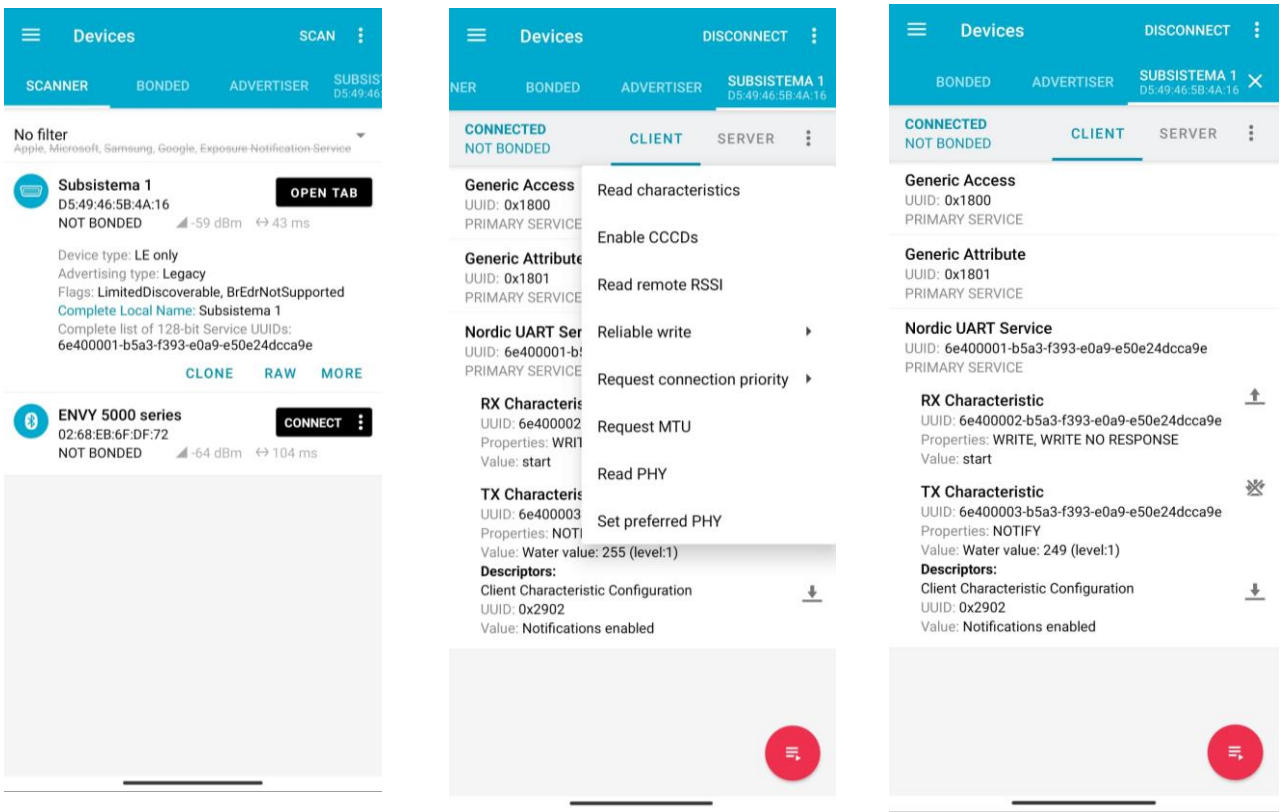

Ilustración 57: Pruebas con nRF Connect (app)

Las figuras (de izquierda a derecha) muestran:

- Los dispositivos que el teléfono móvil ha escaneado
- Una vez ya se ha hecho la conexión con el dispositivo, se acceden a las opciones que se encuentran a la derecha de SERVER y se habilitan las notificaciones a través de “Enable CCCDs”
- Una vez notificaciones están activas, en *TX Characteristics, Value* se observa la cadena de texto enviada por el Arduino.

Las trazas BLE obtenidas se muestran en la siguiente figura:

No.	RSSI	Channel Index	Source	PHY	Protocol	Info	Values
7	-23 dBm	24	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_FEATURE_REQ	
8	-58 dBm	24	Slave_0xac2724c7	LE 1M	ATT	Rcvd Exchange MTU Request, Client Rx MTU: 247	
9	-23 dBm	28	Master_0xac2724c7	LE 1M	ATT	Sent Exchange MTU Response, Server Rx MTU: 247	
10	-58 dBm	28	Slave_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_FEATURE_RSP	
11	-24 dBm	28	Master_0xac2724c7	LE 1M	ATT	Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0001.…	
12	-23 dBm	21	Master_0xac2724c7	LE 1M	ATT	Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0001.…	
13	-23 dBm	21	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_LENGTH_REQ	
14	-22 dBm	3	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_LENGTH_REQ	
15	-24 dBm	25	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_LENGTH_REQ	
16	-56 dBm	25	Slave_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_LENGTH_REQ	
17	-57 dBm	25	Slave_0xac2724c7	LE 1M	ATT	Rcvd Read By Group Type Response, Attribute List Length: 2, Generic Access Profile,…	
18	-23 dBm	29	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_LENGTH_RSP	
19	-57 dBm	29	Slave_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_LENGTH_RSP	
20	-24 dBm	29	Master_0xac2724c7	LE 1M	ATT	Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x000b.…	
21	-23 dBm	21	Master_0xac2724c7	LE 1M	ATT	Sent Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x000b.…	
22	-23 dBm	21	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_CONNECTION_UPDATE_IND	
23	-24 dBm	34	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_CONNECTION_UPDATE_IND	
24	-58 dBm	34	Slave_0xac2724c7	LE 1M	ATT	Rcvd Read By Group Type Response, Attribute List Length: 1, Nordic UART Service	
25	-23 dBm	26	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Include Declaration, Handles: 0x0001..0x0009	
26	-57 dBm	21	Slave_0xac2724c7	LE 1M	ATT	Rcvd Error Response - Attribute Not Found, Handle: 0x0001 (Generic Access Profile)	
27	-23 dBm	23	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0001..0x0009	
28	-56 dBm	17	Slave_0xac2724c7	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 4, Device Name, Appearance, Peri…	
29	-23 dBm	25	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0009..0x0009	
30	-56 dBm	16	Slave_0xac2724c7	LE 1M	ATT	Rcvd Error Response - Attribute Not Found, Handle: 0x0009 (Generic Access Profile: …	
31	-22 dBm	18	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Include Declaration, Handles: 0x000b..0xffff	
32	-57 dBm	29	Slave_0xac2724c7	LE 1M	ATT	Rcvd Error Response - Attribute Not Found, Handle: 0x000b (Nordic UART Service)	
33	-24 dBm	22	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_VERSION_IND	
34	-23 dBm	22	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x000b..0xffff	
35	-24 dBm	27	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x000b..0xffff	
36	-56 dBm	27	Slave_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_VERSION_IND	
37	-58 dBm	13	Slave_0xac2724c7	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 2, Nordic UART Tx, Nordic UART Rx	
38	-24 dBm	31	Master_0xac2724c7	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x000f..0xffff	
39	-55 dBm	4	Slave_0xac2724c7	LE 1M	ATT	Rcvd Error Response - Attribute Not Found, Handle: 0x000f (Nordic UART Service: Nor…	
40	-23 dBm	0	Master_0xac2724c7	LE 1M	ATT	Sent Find Information Request, Handles: 0x0010..0xffff	
41	-58 dBm	35	Slave_0xac2724c7	LE 1M	ATT	Rcvd Find Information Response, Handle: 0x0010 (Nordic UART Service: Nordic UART Rx…	
42	-22 dBm	3	Master_0xac2724c7	LE 1M	ATT	Sent Find Information Request, Handles: 0x0011..0xffff	
43	-58 dBm	11	Slave_0xac2724c7	LE 1M	ATT	Rcvd Error Response - Attribute Not Found, Handle: 0x0011 (Nordic UART Service: Nor…	
44	-24 dBm	11	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_CONNECTION_UPDATE_IND	
45	-55 dBm	26	Slave_0xac2724c7	LE 1M	L2CAP	Connection Parameter Update Request	
46	-22 dBm	18	Master_0xac2724c7	LE 1M	LE LL	Control Opcode: LL_CONNECTION_UPDATE_IND	
47	-22 dBm	18	Master_0xac2724c7	LE 1M	L2CAP	Connection Parameter Update Response (Accepted)	
48	-22 dBm	27	Master_0xac2724c7	LE 1M	L2CAP	Connection Parameter Update Response (Accepted)	
49	-57 dBm	27	Slave_0xac2724c7	LE 1M	LE LL	L2CAP Fragment[BoundErrorUnreassembled Packet]	
50	-25 dBm	18	Master_0xac2724c7	LE 1M	ATT	Sent Write Request, Handle: 0x0010 (Nordic UART Service: Nordic UART Rx: Client Cha…	0x0001
51	-60 dBm	13	Slave_0xac2724c7	LE 1M	ATT	Rcvd Write Response, Handle: 0x0010 (Nordic UART Service: Nordic UART Rx: Client Ch…	
52	-53 dBm	36	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 275 (level:2)\r
53	-49 dBm	33	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 168 (level:1)\r
54	-48 dBm	5	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 302 (level:2)\r
55	-51 dBm	11	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 239 (level:1)\r
56	-52 dBm	19	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 0 (level:0)\r
57	-53 dBm	24	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 314 (level:2)\r
58	-54 dBm	21	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 71 (level:0)\r
59	-58 dBm	2	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 18 (level:0)\r
60	-60 dBm	29	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 284 (level:2)\r
61	-57 dBm	20	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 5 (level:0)\r
62	-50 dBm	34	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 208 (level:1)\r
63	-56 dBm	19	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	[MEAN] Water value: 162 (level:0)\r
64	-56 dBm	19	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 210 (level:1)\r
65	-55 dBm	32	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 52 (level:0)\r
66	-59 dBm	1	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 287 (level:2)\r
67	-58 dBm	7	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 46 (level:0)\r
68	-55 dBm	9	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 172 (level:1)\r
69	-55 dBm	9	Slave_0xac2724c7	LE 1M	LE LL	L2CAP Fragment Start	
70	-55 dBm	9	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 240 (level:1)\r
71	-56 dBm	14	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 40 (level:0)\r
72	-54 dBm	31	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 296 (level:2)\r
73	-55 dBm	25	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 105 (level:0)\r
74	-57 dBm	20	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 103 (level:0)\r
75	-54 dBm	9	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 282 (level:2)\r
76	-60 dBm	2	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 58 (level:0)\r
77	-61 dBm	3	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	[MEAN] Water value: 157 (level:0)\r
78	-61 dBm	3	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 155 (level:1)\r
79	-57 dBm	7	Slave_0xac2724c7	LE 1M	LE LL	L2CAP Fragment[BoundErrorUnreassembled Packet]	
80	-59 dBm	17	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 238 (level:1)\r
81	-58 dBm	24	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 51 (level:0)\r
82	-56 dBm	7	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 293 (level:2)\r
83	-59 dBm	27	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 125 (level:0)\r
84	-62 dBm	1	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 81 (level:0)\r
85	-57 dBm	4	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 292 (level:2)\r
86	-52 dBm	35	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 188 (level:1)\r
87	-61 dBm	1	Slave_0xac2724c7	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 71 (level:0)\r

Ilustración 58: Traza 1 Wireshark Subsistema 1

En las trazas se puede ver en color verde aquellas que utilizan el protocolo ATT. Por otro lado, las moradas, son aquellas que son una notificación. Y por último las azules, el resto de los mensajes BLE .

Se ve como en la última columna se imprime el mensaje enviado por el nRF al teléfono. Se puede comprobar como en cada iteración el nivel de agua cambia. Y cada doce paquetes, se recibe el mensaje que comienza con [MEAN] que incluye la media del último minuto.

Algunos paquetes que resaltar en la comunicación son:

- **Paquete 50**: En este paquete, el maestro (teléfono móvil) habilita las CCCDs del esclavo (nRF) escribiendo un 0x0001.

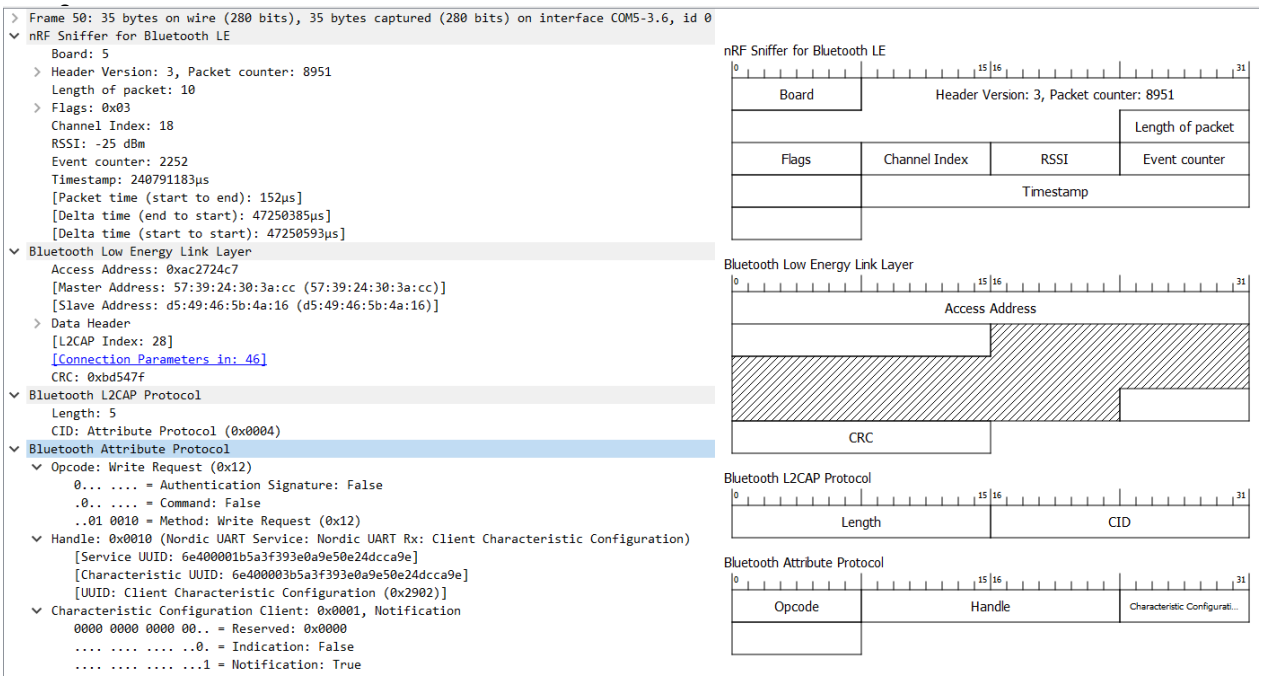

Ilustración 59: Descripción paquete 50 Prueba 1 Subsistema 1

- **Paquetes (morados) 52 -> …** : En estos paquetes se observa como el nRF52 envía las notificaciones cada 5 segundos al Maestro utilizando el protocolo ATT. El mensaje es el siguiente:

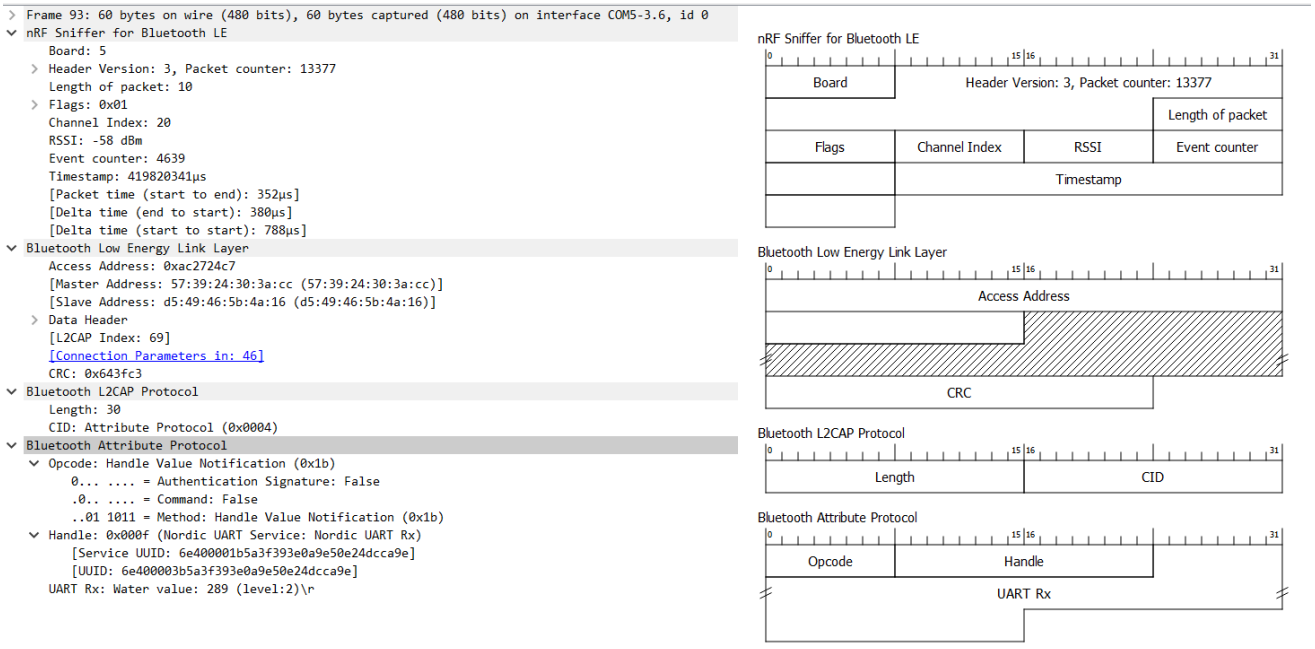

Ilustración 60: Descripción paquete 93 Prueba 1 Subsistema 1

6.1.2 Prueba RX y TX Characteristics

En esta prueba lo que se quiere verificar es el envío de un mensaje a través del teléfono hacia el Arduino para parar/continuar con la ejecución de programa. Se siguen los siguientes pasos partiendo de que ya hemos hecho el test anterior:

1. Se envía la cadena “stop” a través de RX Characteristics, para ello se selecciona la flecha al lado del panel y se escribe el valor de la cadena de texto

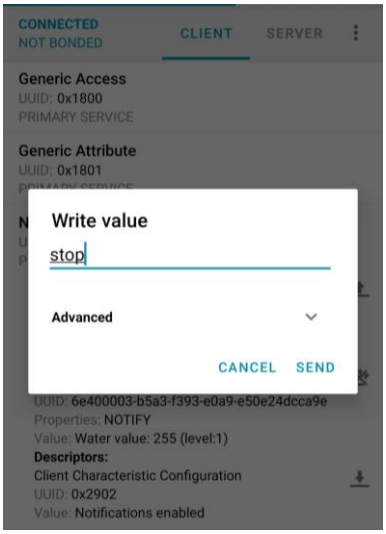

Ilustración 61: Envío "stop" mediante nRF Connect (app)

2. Se deben observar dos mensajes por parte del nRF y que ya no se recibirán más mensajes con las notificaciones de los valores:

3204	-47 dBm	7	Master_0xa864c2f9	LE 1M	ATT	Sent Write Request, Handle: 0x000d (Nordic UART Service: Nordic UART Tx)	stop
3207	-41 dBm	30	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Write Response, Handle: 0x0010 (Nordic UART Service: Nordic UART Rx: Client Ch…	
3237	-39 dBm	13	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Data TX is stopped!!\r
3239	-42 dBm	13	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Use 'start' to continue the TX\r
3308	-43 dBm	18	Master_0xa864c2f9	LE 1M	LE LL	Control Opcode: LL_CHANNEL_MAP_IND	
3350	-47 dBm	7	Master_0xa864c2f9	LE 1M	LE LL	Control Opcode: LL_FEATURE_REQ	
3447	-39 dBm	33	Master_0xa864c2f9	LE 1M	LE LL	Control Opcode: LL_CHANNEL_MAP_IND	
3647	-40 dBm	9	Master_0xa864c2f9	LE 1M	LE LL	Control Opcode: LL_CHANNEL_MAP_IND	
3719	-42 dBm	13	Master_0xa864c2f9	LE 1M	LE LL	Control Opcode: LL_CHANNEL_MAP_IND	

Ilustración 62: Traza 2 Wireshark Subsistema 1

3. Se espera un poco y se envía la cadena “start”.

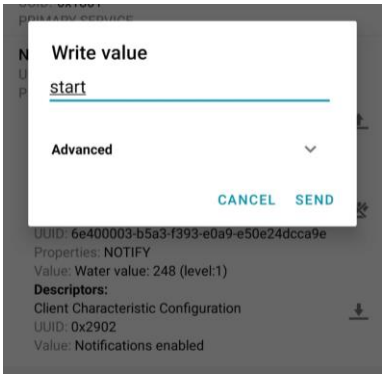

Ilustración 63: Envío "start" mediante nRF Connect (app)

4. Lo que se debe observar son dos mensajes por parte del nRF y que se vuelven a recibir más mensajes:

8363	-37 dBm	21	Master_0xa864c2f9	LE 1M	ATT	Sent Write Request, Handle: 0x000d (Nordic UART Service: Nordic UART Tx)	start
8366	-38 dBm	23	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Write Response, Handle: 0x0010 (Nordic UART Service: Nordic UART Rx: Client Ch…	
8368	-37 dBm	23	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Data TX starts again!\r
8370	-38 dBm	23	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Use 'stop' to pause the TX\r
8372	-38 dBm	23	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 254 (level:1)\r
8504	-39 dBm	16	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 271 (level:2)\r
8638	-38 dBm	18	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 203 (level:1)\r
8675	-57 dBm	5	Master_0xa864c2f9	LE 1M	LE LL	Control Opcode: LL_CHANNEL_MAP_IND	
8778	-40 dBm	30	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 262 (level:2)\r
8918	-38 dBm	10	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	[MEAN] Water value: 252 (level:1)\r
8920	-38 dBm	10	Slave_0xa864c2f9	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x000f (Nordic UART Service: Nordic UART Rx)	Water value: 259 (level:1)\r

Ilustración 64: Traza 3 Wireshark Subsistema 1

A resaltar en este caso podemos analizar el **paquete 8368** el cual contiene la cadena que se ha enviado a través del teléfono vía BLE

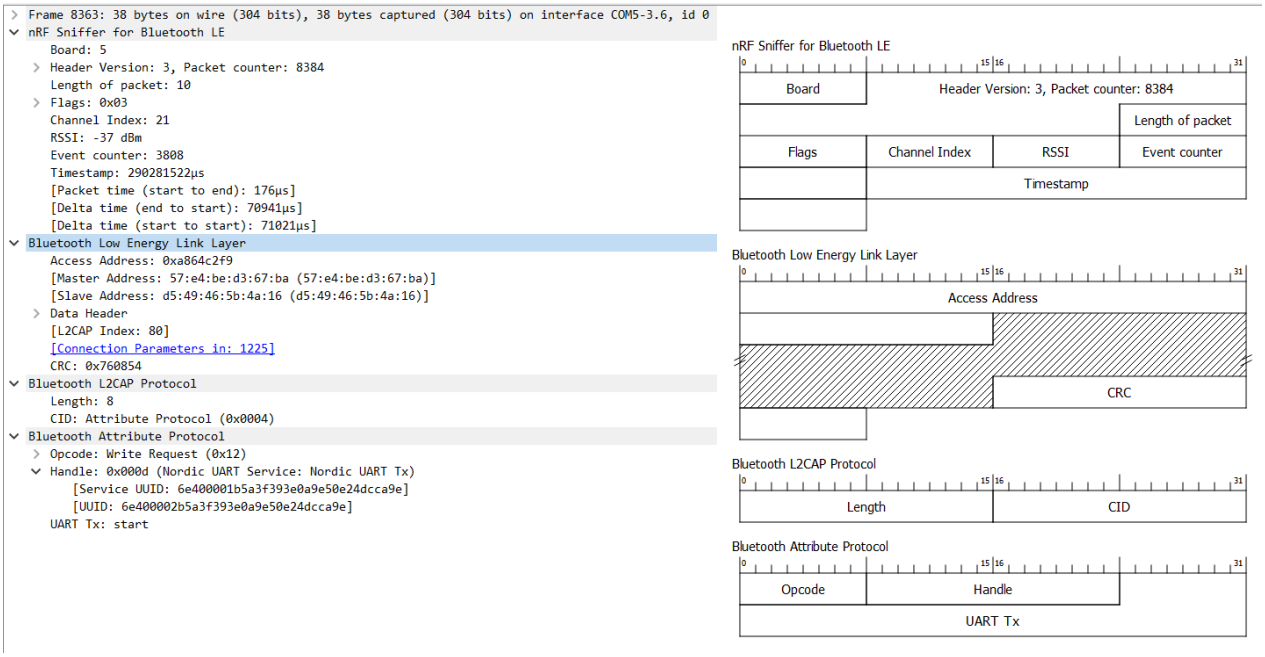

Ilustración 65: Descripción paquete 8363 Prueba 2 Subsistema 1

6.2 Subsistema 2

El sistema completo se muestra en la figura:

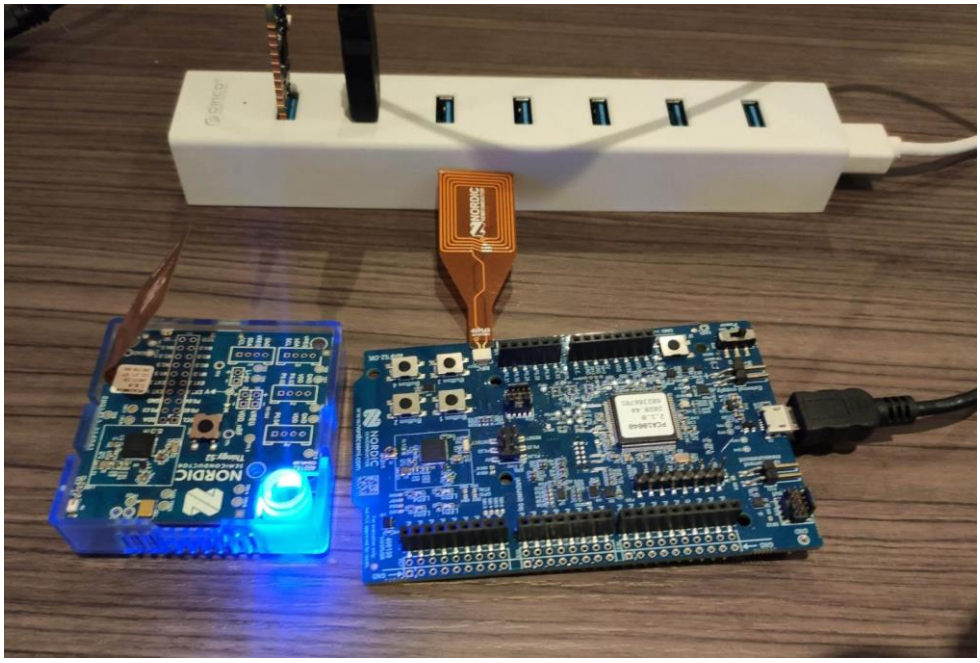

Ilustración 66: Subsistema 2 Interconectado

Vemos como en este caso el Thingy está funcionando a través de su alimentación (pila) y el nRF sí está alimentado a través del HUB-USB.
En este caso solo se lleva a cabo una prueba para verificar el sistema siguiendo los siguientes pasos:

1. Programar nRF52 DK
2. Conectar el sniffer y abrir Wireshark para capturar las trazas
3. Abrir la aplicación nRF Connect en el dispositivo móvil
4. Buscar el dispositivo *Sebas* y observar los diferentes servicios que se nos detallan:

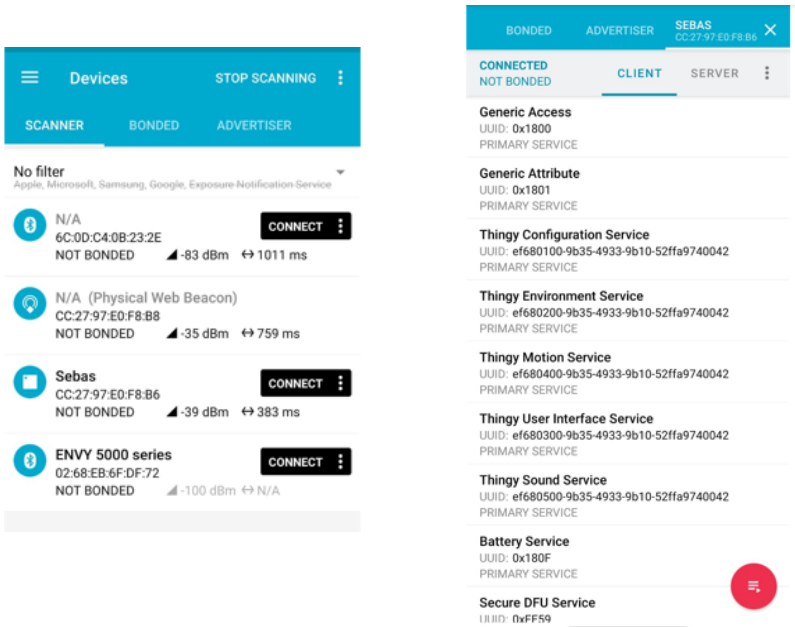

Ilustración 67: Escaneo mediante nRF Connect (app)

5. Se observa como aparecen los servicios Environment, Button y Battery

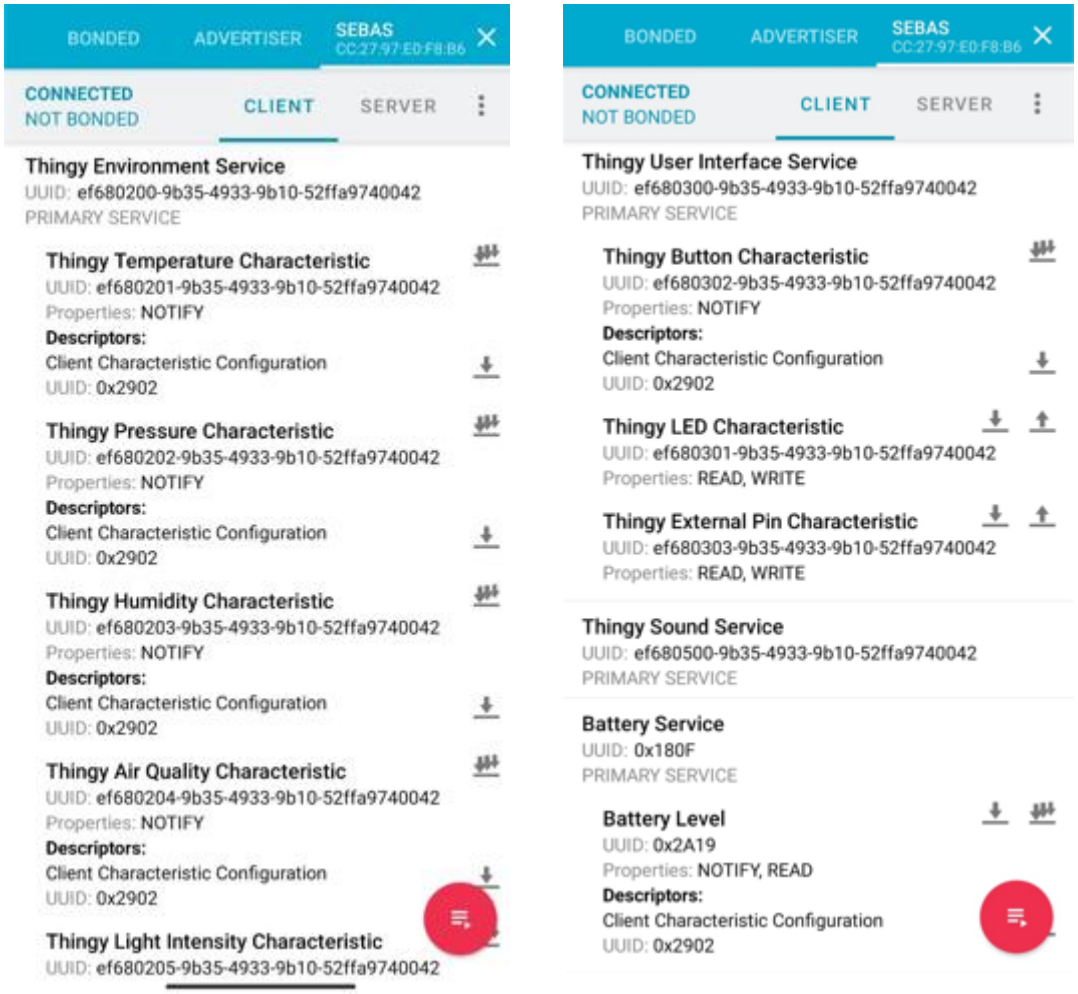

Ilustración 68: Servicios Thingy:52 a través de nRF Connect (app)

6. Una vez se comprueban los servicios se apaga Thingy:52 y se alimenta el nRF52 DK.
7. Se habilita una sesión en el PuTTy hacia el dispositivo indicando el puerto serie y el *baud rate* y se reinicia con el botón, se imprimen las siguientes líneas:

```
<info> app: Start example 2 - Thingy:52 to nRF52 DK
```

8. Se enciende Thingy:52, cuando conecte, reiniciamos el dispositivo, una vez hecho esto en Wireshark, en el Toolbox, se selecciona el dispositivo Thingy para que se analice su comunicación

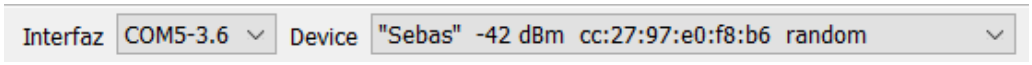

Ilustración 69: Toolbox Wireshark para analizar la conexión

9. Encender Thingy:52 de nuevo y ver como se reconecta.

```
<info> app: Sebas is found, initiate connection
<info> app: Connected.
```

10. A continuación, se recibirá por la sesión del PuTTY el descubrimiento de los servicios del Thingy:52 y se configurarán los CCCDs.

```
<info> app: Thingy Battery service discovered on conn_handle 0x0.
<info> ble_tbs_c: Thingy Button Service discovered at peer.
<info> app: Thingy Button service discovered on conn_handle 0x0.
<info> ble_tbs_c: Configuring CCCD. CCCD Handle = 80, Connection Handle = 0
<info> ble_tes_c: Thingy Enviroment Service discovered at peer.
<info> app: Thingy Environment service discovered on conn_handle 0x0.
<info> ble_tes_c: Configuring CCCD. CCCD Handle = 32, Connection Handle = 0
<info> ble_tes_c: Configuring CCCD. CCCD Handle = 35, Connection Handle = 0
<info> ble_tes_c: Configuring CCCD. CCCD Handle = 38, Connection Handle = 0
<info> ble_tes_c: Configuring CCCD. CCCD Handle = 41, Connection Handle = 0
<info> ble_tes_c: Configuring CCCD. CCCD Handle = 44, Connection Handle = 0
```

11. A continuación, ya se recibirán todos los datos en bucle mediante las notificaciones previamente configuradas.

```
<info> app: Got temperature: 26,50
<info> app: Got pressure: 1018,68
<info> app: Got humidity: 49.
<info> app: Got color. R479, G21, B8, C44
<info> app: Got temperature: 26,52
<info> app: Got pressure: 1018,69
<info> app: Got humidity: 48.
<info> app: Got color. R475, G21, B6, C44
<info> app: Got C02: 0.
<info> app: Got organic components: 0.
<info> app: Got Battery Level: 62
...
```

12. Si pulsamos el botón en el Thingy:52 observamos la siguiente línea además de ver como se enciende el LED en el nRF.

```
<info> app: Button state changed on peer to 0x1.
<info> app: Button state changed on peer to 0x0.
```

Ahora se visualizan los mensajes de la comunicación en Wireshark para mostrar los paquetes interesantes en la siguiente figura.

Primero se descubren los servicios y se habilitan las notificaciones. Posteriormente recibiremos los valores de las características periódicamente.

No.	RSSI	Channel Index	Source	PHY	Protocol	Info	Values
2852	-37 dBm	14	Master_0x10b26b3d	LE 1M	ATT	Sent Find By Type Value Request, GATT Primary Service Declaration, Handles: 0x0001…	0f18
2855	-40 dBm	28	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Find By Type Value Response	
2856	-36 dBm	5	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0060..0x0063	
2859	-39 dBm	19	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Battery Level	
2860	-34 dBm	33	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0063..0x0063	
2863	-38 dBm	10	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Error Response - Attribute Not Found, Handle: 0x0063 (Unknown)	
2864	-34 dBm	24	Master_0x10b26b3d	LE 1M	ATT	Sent Find Information Request, Handles: 0x0063..0x0063	
2867	-39 dBm	1	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Find Information Response, Handle: 0x0063 (Unknown: Battery Level: Client Char…	
2868	-36 dBm	15	Master_0x10b26b3d	LE 1M	ATT	Sent Find By Type Value Request, GATT Primary Service Declaration, Handles: 0x0001…	420074a9ff52109b3349359b000368ef
2871	-41 dBm	29	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Find By Type Value Response	
2872	-36 dBm	6	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x004d..0x0054	
2873	-35 dBm	20	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x004d..0x0054	
2876	-40 dBm	34	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2877	-35 dBm	11	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0050..0x0054	
2880	-40 dBm	25	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2881	-35 dBm	2	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0053..0x0054	
2884	-39 dBm	16	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2885	-34 dBm	30	Master_0x10b26b3d	LE 1M	ATT	Sent Find Information Request, Handles: 0x0050..0x0050	
2888	-39 dBm	7	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Find Information Response, Handle: 0x0050 (Unknown: Unknown: Client Characteri…	
2889	-36 dBm	21	Master_0x10b26b3d	LE 1M	ATT	Sent Find By Type Value Request, GATT Primary Service Declaration, Handles: 0x0001…	420074a9ff52109b3349359b000268ef
2892	-40 dBm	35	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Find By Type Value Response	
2893	-36 dBm	12	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x001d..0x002e	
2896	-40 dBm	26	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2897	-36 dBm	3	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0020..0x002e	
2900	-39 dBm	17	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2901	-34 dBm	31	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0023..0x002e	
2904	-38 dBm	8	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2905	-36 dBm	22	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0026..0x002e	
2908	-39 dBm	36	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2909	-36 dBm	13	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x0029..0x002e	
2912	-40 dBm	27	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	
2913	-36 dBm	4	Master_0x10b26b3d	LE 1M	ATT	Sent Read By Type Request, GATT Characteristic Declaration, Handles: 0x002c..0x002e	
2916	-39 dBm	18	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Read By Type Response, Attribute List Length: 1, Unknown	

Ilustración 70: Traza 1 Wireshark Subsistema 2

En este intercambio de mensajes, los paquetes interesantes son 2859, 2876, 2896, 2900, 2904, 2908, 2912 donde el Slave (Thingy:52) muestra su lista de atributos, por ejemplo, el mensaje 2908 contiene la siguiente información:

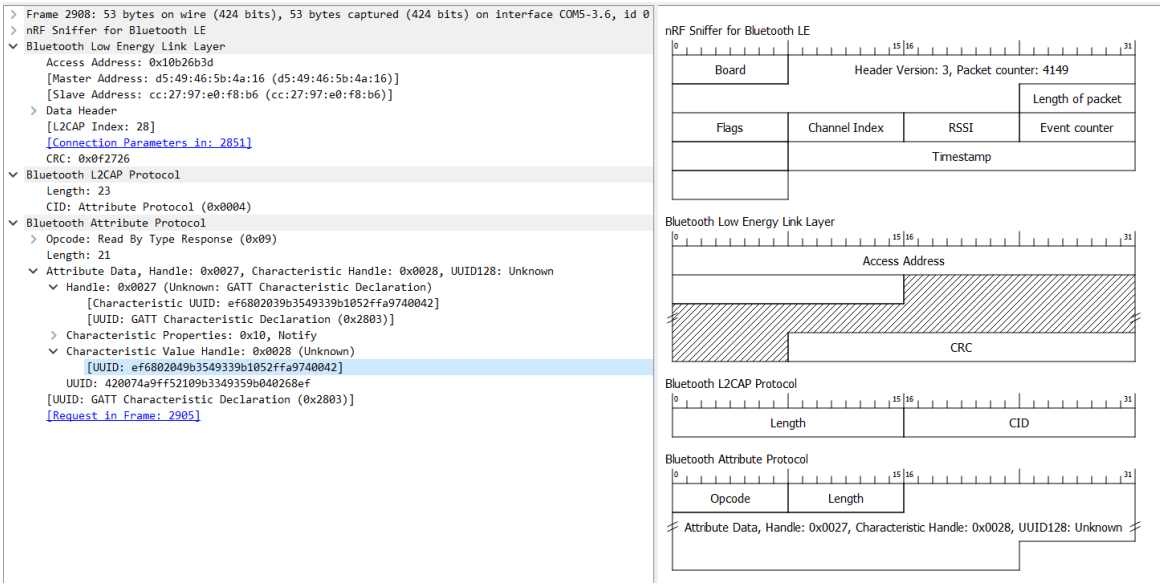

Ilustración 71: Descripción paquete 2908 Prueba 1 Subsistema 2

El UUID que se ha descubierto es ef6802049b3549339b1052ffa9740042 que si accedemos a la documentación se observa que se corresponde con la calidad de aire. Los servicios se pueden consular en [10.2 BLE Services Thingy:52]

Lo mismo se realiza para todos los UUID que se han definido en el código, en concreto son:

- Batería: `180F`
- Color: `EF680205-9B35-4933-9B10-52FFA9740042`
- Calidad del aire/gas: `EF680204-9B35-4933-9B10-52FFA9740042`
- Humedad: `EF680203-9B35-4933-9B10-52FFA9740042`
- Presión: `EF680202-9B35-4933-9B10-52FFA9740042`
- Temperatura: `EF680201-9B35-4933-9B10-52FFA9740042`
- Botón: `EF680302-9B35-4933-9B10-52FFA9740042`

Para mejorar la visualización de los datos se incorpora el siguiente código de colores basado en los UUID del mensaje:

Battery	btatt.handle == 0x0062
Color	btatt.uuid128[2] == 02 && btatt.uuid128[3] == 05
Gas	btatt.uuid128[2] == 02 && btatt.uuid128[3] == 04
Humidity	btatt.uuid128[2] == 02 && btatt.uuid128[3] == 03
Pressure	btatt.uuid128[2] == 02 && btatt.uuid128[3] == 02
Temperature	btatt.uuid128[2] == 02 && btatt.uuid128[3] == 01
Button	btatt.uuid128[2] == 03 && btatt.uuid128[3] == 02
BTLE	btle

Ilustración 72: Reglas colores Subsistema 2 Wireshark

Los siguientes mensajes interesantes son aquellos donde se activan las notificaciones , en el caso del subsistema, habrá que hacerlo para los diferentes atributos que deseemos recibir.

2938	-35 dBm	1	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x0063 (Unknown: Battery Level: Client Characteristic C…	0x0001
2941	-40 dBm	15	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x0063 (Unknown: Battery Level: Client Characteristic …	
2942	-35 dBm	29	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x0050 (Unknown: Unknown: Client Characteristic Configu…	0x0001
2945	-39 dBm	6	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x0050 (Unknown: Unknown: Client Characteristic Config…	
2946	-36 dBm	20	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x0020 (Unknown: Unknown: Client Characteristic Configu…	0x0001
2949	-40 dBm	34	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x0020 (Unknown: Unknown: Client Characteristic Config…	
2951	-40 dBm	34	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x001f (Unknown: Unknown)	1a32
2952	-36 dBm	11	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x0023 (Unknown: Unknown: Client Characteristic Configu…	0x0001
2955	-40 dBm	25	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x0023 (Unknown: Unknown: Client Characteristic Config…	
2957	-40 dBm	25	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x0022 (Unknown: Unknown)	fa03000044
2958	-35 dBm	2	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x0026 (Unknown: Unknown: Client Characteristic Configu…	0x0001
2961	-39 dBm	16	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x0026 (Unknown: Unknown: Client Characteristic Config…	
2963	-39 dBm	16	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Handle Value Notification, Handle: 0x0025 (Unknown: Unknown)	31
2964	-34 dBm	30	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x0029 (Unknown: Unknown: Client Characteristic Configu…	0x0001
2967	-39 dBm	7	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x0029 (Unknown: Unknown: Client Characteristic Config…	
2968	-36 dBm	21	Master_0x10b26b3d	LE 1M	ATT	Sent Write Request, Handle: 0x002c (Unknown: Unknown: Client Characteristic Configu…	0x0001
2971	-40 dBm	35	Slave_0x10b26b3d	LE 1M	ATT	Rcvd Write Response, Handle: 0x002c (Unknown: Unknown: Client Characteristic Config…	

Ilustración 73: Traza 2 Wireshark Subsistema 2

En concreto estos paquetes son los 2938, 2942, 2946, 2952, 2958, 2964, 2968. El paquete 2942 contiene lo siguiente:

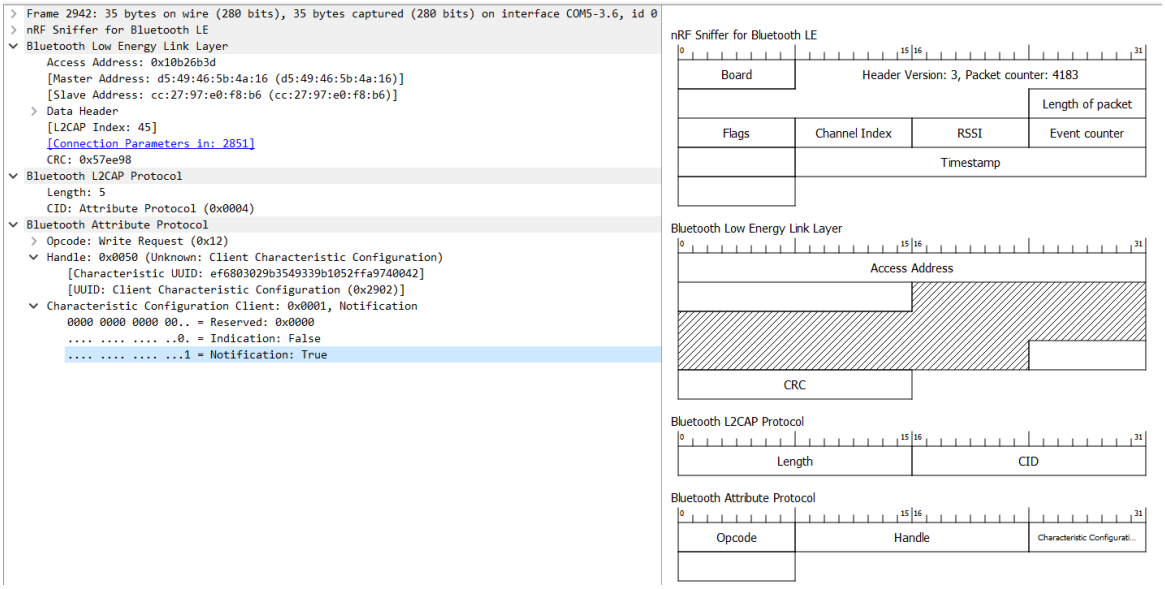

Ilustración 74: Descripción paquete 2942 Prueba 1 Subsistema 2

Se ha puesto a 1 el bit correspondiente a las notificaciones para el servicio ef6803029b3549339b1052ffa9740042 que corresponde al Button Service.

Finalmente, ya se observan los valores de las características las cuales hemos habilitado.

En la siguiente tabla se muestran unos extractos de paquetes de cada una de las características:

Temperatura	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x001f (Unknown) [UUID: ef6802019b3549339b1052ffa9740042] Value: 1a50
Presión	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x0022 (Unknown) [UUID: ef6802029b3549339b1052ffa9740042] Value: fa03000049
Humedad	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x0025 (Unknown) [UUID: ef6802039b3549339b1052ffa9740042] Value: 2f
Gas	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x0028 (Unknown) [UUID: ef6802049b3549339b1052ffa9740042] Value: 90010000
Color	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x002b (Unknown) [UUID: ef6802059b3549339b1052ffa9740042] Value: db01140006002e00
Batería	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x0062 (Battery Level) [UUID: Battery Level (0x2a19)] Battery Level: 62%
Botón	Bluetooth Attribute Protocol Opcode: Handle Value Notification (0x1b) 0... .... = Authentication Signature: False .0.. .... = Command: False ..01 1011 = Method: Handle Value Notification (0x1b) Handle: 0x004f (Unknown) [UUID: ef6803029b3549339b1052ffa9740042] Value: 01

7. Conclusiones

En este capítulo se expondrán las conclusiones halladas y las posibles líneas futuras que podrían continuar este trabajo o mejorar su implementación.

7.1 Resultados obtenidos

Lo primero que hemos de hacer al llegar a este punto es verificar que hemos cumplido los objetivos que se marcaron en la introducción de este documento.

El estudio de la tecnología BLE se llevó a cabo en el estudio del estado del arte. Para implementar cualquier sistema electrónico es vital realizar un buen estudio de mercado teniendo en cuenta los requisitos del sistema, sus características y presupuesto. Por ello esta parte fue abordada mientras se diseñó la estructura hardware del proyecto. También se ha conseguido realizar y modificar códigos para el Arduino y el SoC para realizar dos subsistemas donde nuestro dispositivo principal actuara como central y periférico para la recepción y envío de datos mediante BLE. Una vez los dispositivos tenían el código correcto también se han conseguido conectarlos para el envío de datos mediante BLE/UART. Ha sido también importante realizar las diversas pruebas sobre los sistemas para corroborar su funcionamiento.

En cuanto aalgunas características:

- ***Low Cost***: el sistema está basado en unos componentes accesibles y baratos, además que los IDEs y SDK son gratutitos.

Subsistema 1		
Componente	Precio	Comprado en
nRF52 DK	42,96	Farnell
Arduino UNO	5,66	Aliexpress
Sensor Agua	0,46	Aliexpress
TOTAL	**49,08**	

Subsistema 2		
Componente	Precio	Comprado en
nRF52 DK	42,96	Farnell
Thingy:52*	45,81	Farnell
TOTAL	**88,77**	

Tabla 8: Coste Subsistemas

* El dispositivo Thingy:52 se ha usado en lugar de otro sensor más barato ya que era una excelente opción al ser una plataforma multisensor óptima para la realización de prototipos.

- **Longitud de las tramas**: Las tramas enviadas por los subsistemas son pequeñas, al igual que la distancia en la que se busca utilizar el sistema, por lo que no ha sido necesario un sistema con alta potencia.

- ***Low Power-consumption***: En cuanto al consumo final del sistema no se han realizado medidas puesto que un proyecto comercial no constaría de un DK ya que este contiene muchos periféricos que aumentan el consumo notablemente. Sin embargo, los picos de consumo del SoC del nRF52 son pequeños según el fabricante.

Además, el código utilizado consta de un modo de bajo consumo donde entra periódicamente si no tiene nada que hacer.

Este proyecto también ha servido para ver como BLE es una muy buena opción para el envío y recogida periódica de datos en un escenario como podría ser un hogar o jardín, ya que las distancias son pequeñas.

7.2 Líneas futuras

Para la realización de este trabajo se ha tenido un tiempo y recursos limitados, por lo que para diseñar un sistema Low Cost comercial se deberían realizar muchas mejoras para poder alcanzar también un menor consumo.

A continuación, se muestran algunas líneas que se podrían seguir para mejorar el trabajo realizado y/o darle otro enfoque:

- Integrar un modo *bajo consumo* para que cuando el SoC funciona como periférico esté enviando tramas solo durante 30s cada media hora, de esta manera se le da un enfoque más *Low Power.* Esto también dependerá de los datos a enviar, ya que si no son críticos la periodicidad podría ser incluso mayor.

- Conectar el Arduino mediante BLE al SoC que actúe de central. De esta manera el proyecto no se necesitaría cableado y el proyecto sería más versátil.

- Añadir el servicio *Motion* del Thingy:52 al proyecto, haciendo que el sensor reporte los datos del sensor de movimiento de 9 ejes y del acelerómetro tras activar sus notificaciones

- Añadir más sensores y actuadores en el Arduino aprovechando las UART Software.

- Integrar los datos recibidos en un bot en *Telegram* para tomar decisiones inteligentes basados en los valores recibidos.

- Implementar que el SoC funcione como central y periférico. Algunos ejemplos se proporcionan en el SDK.

8. Glosario

BLE	Bluetooth Low Energy
LE	Low Energy
UART	Universal Asynchronous Receiver-Transmitter
SoC	System On Chip
IoT	Internet Of Things
WBS	Work Breakdown Structure
ROM	Read-Only Memory
RAM	Random Access Memory
UUID	Universally Unique Identifier
PHY	Physical Layer
RX	Reception
TX	Transmission
GFSK	Gaussian Frequency Shift Keying
GAP	Generic Access Profile
GATT	Generic Attribute Profile
ATT	Attribute Profile
SMP	Security Manager Protocol
L2CAP	Logical Link Control and Adaptation Protoco
LL	Link Layer
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
IDE	Integrated Development Environment
DK	Development Kit
NFC	Near Field Communication
SDM	Soft Device Manager
DFU	Device Firmware Update
CCCD	Client Characteristic Configuration Descriptor

9. Bibliografía

[1] R. Tencio, "Fundamentos y Experimentación con Bluetooth Low Energy - COSTARICAMAKERS.com", COSTARICAMAKERS.com, 2022. https://costaricamakers.com/fundamentos-y-experimentacion-con-bluetooth-low-energy/#:~:text=El%20Bluetooth%20de%20Baja%20Energ%C3%ADa%20fue%20desarrollado%20por%20Nordic%20Semiconductor,con%20Especial%20Inter%C3%A9s)%2C%20que%20fue

[2] "Texas Instruments CC2540 2.4GHz BLUETOOTH® System-on-Chip". Mouser. https://www.mouser.es/new/texas-instruments/ti-cc2540-socs/.

[3] "Dialog Semiconductor DA14580 Low Power Bluetooth Smart SoC". Mouser. https://www.mouser.es/new/dialog-semiconductor/dialog-semi-da14580-soc/.

[4] "nRF51822 BLUETOOTH® Low Energy 2.4GHz Wireless". Mouser. https://www.mouser.es/new/nordic-semiconductor/nordic-nrf51822-multiprotocol-soc/.

[5] "Adaptive Frequency Hopping - v3.3 - Bluetooth API Documentation Silicon Labs". Software Developer Docs - Silicon Labs. https://docs.silabs.com/bluetooth/latest/general/system-and-performance/adaptive-frequency-hopping.

[6] "Cómo funciona Bluetooth Low Energy: el protocolo estrella de IoT | WeLiveSecurity". WeLiveSecurity. https://www.welivesecurity.com/la-es/2020/03/17/como-funciona-bluetooth-low-energy/#:~:text=Seguridad%20en%20Bluetooth%20Low%20Energy,de%20errores%20hacia%20adelante%20(FEC).

[7] "Bluetooth Low Energy: Un estudio sobre sensores inalámbricos en Internet de las cosas - **Electrodaddy**". **Electrodaddy**. https://www.electrodaddy.com/bluetooth-low-energy-un-estudio-sobre-sensores-inalambricos-en-internet-de-las-cosas/#:~:text=A%20diferencia%20de%20Bluetooth%20Classic,de%201%20a%202%20MHz.

[8] "Bluetooth Protocol Stack- MATLAB & Simulink- MathWorks España". MathWorks - Creadores de MATLAB y Simulink - MATLAB y Simulink - MATLAB & Simulink. https://es.mathworks.com/help/bluetooth/ug/bluetooth-protocol-stack.html;jsessionid=4b27e7c8e2ec1de4ad509e8928b8.

[9] "BLE Link Layer Roles and States - Developer Help". Home - Developer Help. https://microchipdeveloper.com/wireless:ble-link-layer-roles-states

[10] "BLE (Bluetooth Low Energy) | Elt". Innovation in Lighting Technology | ELT. https://www.elt.es/ble-bluetooth-low-energy.

[11] "Attribute and Data Hierarchy - Developer Help". Home - Developer Help. https://microchipdeveloper.com/wireless:ble-gatt-data-organization.

[12] "New IoT Platform for Wireless Device- Bluetooth Low Energy | AumRaj". AumRaj. https://aumraj.com/index.php/new-iot-platform-for-wireless-device-bluetooth-low-energy/

[13] "UNO R3 | Arduino Documentation | Arduino Documentation". Arduino Docs | Arduino Documentation | Arduino Documentation. https://docs.arduino.cc/hardware/uno-rev3.

[14] "Nordic Semiconductor Infocenter". Nordic Semiconductor Infocenter. https://infocenter.nordicsemi.com/index.jsp?topic=/ug_nrf52832_dk/UG/nrf52_DK/intro.html.

[15] Nordic Semiconductors. "nRF52832 SoC Product Brief Version 2.0". Nordic Semiconductor | Specialists in Low Power Wireless - nordicsemi.com. https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52832-product-brief.pdf?la=en&hash=2F9D995F754BA2F2EA944A2C4351E682AB7CB0B9.

[16] "Nordic Thingy:52 - Get started". Nordic Semiconductor | Specialists in Low Power Wireless - nordicsemi.com. https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-52/GetStarted?lang=en.

[17] Nordic Semiconductors. "Nordic Thingy:52 IoT Sensor Kit". https://nsscprodmedia.blob.core.windows.net/prod/software-and-other-downloads/product-briefs/nordic-thingy52-product-brief.pdf.

[18] L. M. Engineers. "In-Depth: How Water Level Sensor Works and Interface it with Arduino - Last Minute Engineers". Last Minute Engineers. https://lastminuteengineers.com/water-level-sensor-arduino-tutorial/.

[19] Nordic Semiconductors. "nRF52840 Dongle". Nordic Semiconductor | Specialists in Low Power Wireless - nordicsemi.com. https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52840-Dongle-product-brief.pdf?la=en&hash=8DDD17CCF2E574021A06A05C71B46C505D492361

[20] "nRF Sniffer for Bluetooth LE". Nordic Semiconductor Infocenter. https://infocenter.nordicsemi.com/index.jsp?topic=/ug_sniffer_ble/UG/sniffer_ble/min_requirements.html.

[21] "Arduino Mega 2560 Rev3". Arduino Official Store. http://store.arduino.cc/products/arduino-mega-2560-rev3.

[22] "Nano 33 BLE | Arduino Documentation | Arduino Documentation". Arduino Docs | Arduino Documentation | Arduino Documentation. https://docs.arduino.cc/hardware/nano-33-ble.

[23] "nRF51 Dongle". Nordic Semiconductor | Specialists in Low Power Wireless - nordicsemi.com. https://www.nordicsemi.com/Products/Development-hardware/nRF51-Dongle .

[24] "S132 SoftDevice - Downloads". Nordic Semiconductor | Specialists in Low Power Wireless - nordicsemi.com. https://www.nordicsemi.com/Products/Development-software/s132/download.

[25] "S132 SoftDevice, SoftDevice Specification v7.1". Nordic Semiconductor Infocenter. https://infocenter.nordicsemi.com/pdf/S132_SDS_v7.1.pdf.

[26] "Nordic Thingy:52 v2.2.0 : Firmware architecture". GitHub Pages. https://nordicsemiconductor.github.io/Nordic-Thingy52-FW/documentation/firmware_architecture.html.

[27] "BLE Services, Nordic Thingy:52 v2.2.0 : Firmware architecture". GitHub Pages. https://nordicsemiconductor.github.io/Nordic-Thingy52-FW/documentation/firmware_architecture.html#fw_arch_ble_services.

[28] "GitHub - crfosse/Thingy-52-to-nRF52xx: An example of how you can get sensor data over BLE from Nordic Thingy:52 to a nrf52xx development kit". GitHub. https://github.com/crfosse/Thingy-52-to-nRF52xx.

[29] "GitHub - NordicPlayground/nrf52-ble-multi-link-multi-role". GitHub. https://github.com/NordicPlayground/nrf52-ble-multi-link-multi-role.

[30] "Installing the nRF Sniffer capture tool". Nordic Semiconductor Infocenter. https://infocenter.nordicsemi.com/index.jsp?topic=/comp_matrix_nrf52833/COMP/nrf52833/nRF52833_ic_rev_sdk_sd_comp_matrix.html.

[31] "Adding a Wireshark profile for the nRF Sniffer". Nordic Semiconductor Infocenter. https://infocenter.nordicsemi.com/index.jsp?topic=/comp_matrix_nrf52833/COMP/nrf52833/nRF52833_ic_rev_sdk_sd_comp_matrix.html.

[32] "Programming the nRF Sniffer firmware". Nordic Semiconductor Infocenter. https://infocenter.nordicsemi.com/index.jsp?topic=/comp_matrix_nrf52833/COMP/nrf52833/nRF52833_ic_rev_sdk_sd_comp_matrix.html.

[33] "Wireshark · Display Filter Reference: Bluetooth Low Energy Link Layer". Wireshark · Go Deep. https://www.wireshark.org/docs/dfref/b/btle.html.

[34] "nRF Connect for Desktop". Nordic Semiconductor | Specialists in Low Power Wireless - nordicsemi.com. https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-desktop.

10. Anexos

10.1 Código

En este anexo se muestran los códigos utilizados en cada subsistema para cada dispositivo.

10.1.1 Subsistema 1

10.1.1.1 Código Arduino

```
/*********************************************************************************
 * Sketch to obtain and populate the water measures from a sensor to another device
 * through serial port.
 * Device needed:
 *  - Arduino UNO
 *  - Water sensor
 *  - Laptop with PuTTy or another device
 *  - Wires to connect the devices
 *********************************************************************************/

 // Define section
#define POWER_PIN      7U       // Power pin
#define SIGNAL_PIN     A5       // Sensor Data PIN
#define DELAY_MEASURES 5000U    // Delay between measures [ms]
#define BUCKET_SIZE    12U      // Bucket sise
#define SENSOR_MIN 0            // Sensor minimum value
#define SENSOR_MAX 521          // Sensor maximum value
#define BAUD_RATE 115200        // Rate of serial communcation
#define MAX_MESSAGE_LENGTH 6U   // Maximum msg length
#define STOP "stop"             // Msg to stop the program
#define START "start"           // Msg to continue the program

// Global variables
int value = 0;                                  // variable to store the sensor's value
int level = 0;                                  // variable to store the water's level
static int value_bucket[BUCKET_SIZE] = {0U};    // variable to store all the values
static int level_bucket[BUCKET_SIZE] = {0U};    // variable to store all the levels
static long sum_value = 0U;                     // variable to store the sum of the values
static long sum_level = 0U;                     // variable to store the sum of the levels
static int iteration = 0;                       // variable to indicate the iteration of the loop
static bool stopData = false;                   // variable to indicate if the program must
continue

/*
 * This function initialize the devices and serial communcation
 */
void setup() {
  Serial.begin(BAUD_RATE);      // Start serial communication with nRF52 DK
  pinMode(POWER_PIN, OUTPUT);   // Configure D7 pin as an OUTPUT
  digitalWrite(POWER_PIN, LOW); // Turn the sensor OFF
  delay(2000U);                 // Delay to start
}


/*
 * This function resets the buckets to zero.
 */
static void resetBuckets()
{
  memset(value_bucket,0U,sizeof(value_bucket));
  memset(level_bucket,0U,sizeof(level_bucket));

  sum_value = 0U;
  sum_level = 0U;
}

/*
 * This function calls the reset function and also computes the mean of the measures
```

```
 * when the bucket is full
 */
static void resetAndSendMeasures()
{
  for(int it = 0; it < BUCKET_SIZE ; it++) // Loop through the measures and do the sum
  {
    sum_value += value_bucket[it];
    sum_level += level_bucket[it];
  }
  String str = String("[MEAN] Water value: ") + String(sum_value/BUCKET_SIZE) + String("
(level:") + String(sum_level/BUCKET_SIZE) + String(")"); // Print the mean
  Serial.println(str);                     // Send string through the serial port

  resetBuckets();                          // Reset the buckets
}

/*
 * This function checks the msg received
 */
static void checkRX()
{
 //Check to see if anything is available in the serial receive buffer
 while (Serial.available() > 0)
 {
   static char message[MAX_MESSAGE_LENGTH]; // Variable to store the message
   static unsigned int message_pos = 0;

   //Read the next byte in the serial receive buffer
   char inByte = Serial.read();

   //Check if we have receibe a full msg
   if ( inByte != '\n' && (message_pos < MAX_MESSAGE_LENGTH - 1) )
   {
     message[message_pos] = inByte;
     message_pos++;
   }
   // Full msg received
   else
   {
     message[message_pos] = '\0'; //Add null character to string

     if(memcmp(message,STOP,4U) == 0U) // "stop" string received
     {
         stopData = true;
         Serial.println("Data TX is stopped!!");
         Serial.println("Use 'start' to continue the TX");
     }

     if(memcmp(message,START,5U) == 0U) // "start" string received
     {
         stopData = false;
         Serial.println("Data TX starts again!");
         Serial.println("Use 'stop' to pause the TX");
     }
     message_pos = 0;
   }
 }
}


void loop()
{
  checkRX();                                          // check the data received

  // Continue with the program if we have not recieved the "stop"
  if(stopData == false)
  {
    digitalWrite(POWER_PIN, HIGH);                    // turn the sensor ON
    delay(10);                                        // wait 10 milliseconds
    value = analogRead(SIGNAL_PIN);                   // read the analog value from sensor
    level = map(value, SENSOR_MIN, SENSOR_MAX, 0, 4); // 4 levels

    digitalWrite(POWER_PIN, LOW);                     // turn the sensor OFF
```

```
    if(iteration == BUCKET_SIZE)                    // Bucket is full
    {
      //Populate the mean of the measures
      resetAndSendMeasures();
      iteration = 0;
    }

    value_bucket[iteration] = value;                // Log the value
    level_bucket[iteration] = level;                // Log the level

    String str = String("Water value: ") + String(value) + String(" (level:") + String(level) +
String(")");
    Serial.println(str);                            // Send string through the serial port
    iteration++;                                    // Increment the iteration
    delay(DELAY_MEASURES);                          // Wait for the next measure
  }
}
```

10.1.1.2 Código PCA10040

```
/** @file
 *
 * @defgroup ble_sdk_uart_over_ble_main main.c
 * @{
 * @ingroup  ble_sdk_app_nus_eval
 * @brief    UART over BLE application main file.
 *
 * This file contains the source code for a sample application that uses the Nordic UART
service.
 * This application uses the @ref srvlib_conn_params module.
 */


#include <stdint.h>
#include <string.h>
#include "nordic_common.h"
#include "nrf.h"
#include "ble_hci.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "nrf_sdh.h"
#include "nrf_sdh_soc.h"
#include "nrf_sdh_ble.h"
#include "nrf_ble_gatt.h"
#include "nrf_ble_qwr.h"
#include "app_timer.h"
#include "ble_nus.h"
#include "app_uart.h"
#include "app_util_platform.h"
#include "bsp_btn_ble.h"
#include "nrf_pwr_mgmt.h"

#if defined (UART_PRESENT)
#include "nrf_uart.h"
#endif
#if defined (UARTE_PRESENT)
#include "nrf_uarte.h"
#endif

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"

#define APP_BLE_CONN_CFG_TAG            1                                       /**< A tag
identifying the SoftDevice BLE configuration. */

#define DEVICE_NAME                     "Subsistema 1"                           /**< Name
of device. Will be included in the advertising data. */
#define NUS_SERVICE_UUID_TYPE           BLE_UUID_TYPE_VENDOR_BEGIN              /**< UUID
type for the Nordic UART Service (vendor specific). */

#define APP_BLE_OBSERVER_PRIO           3                                       /**<
Application's BLE observer priority. You shouldn't need to modify this value. */

#define APP_ADV_INTERVAL                64                                      /**< The
advertising interval (in units of 0.625 ms. This value corresponds to 40 ms). */

#define APP_ADV_DURATION                18000                                   /**< The
advertising duration (180 seconds) in units of 10 milliseconds. */

#define MIN_CONN_INTERVAL               MSEC_TO_UNITS(20, UNIT_1_25_MS)         /**< Minimum
acceptable connection interval (20 ms), Connection interval uses 1.25 ms units. */
#define MAX_CONN_INTERVAL               MSEC_TO_UNITS(75, UNIT_1_25_MS)         /**< Maximum
acceptable connection interval (75 ms), Connection interval uses 1.25 ms units. */
#define SLAVE_LATENCY                   0                                       /**< Slave
latency. */
#define CONN_SUP_TIMEOUT                MSEC_TO_UNITS(4000, UNIT_10_MS)         /**<
Connection supervisory timeout (4 seconds), Supervision Timeout uses 10 ms units. */
```

```
#define FIRST_CONN_PARAMS_UPDATE_DELAY  APP_TIMER_TICKS(5000)                       /**< Time
from initiating event (connect or start of notification) to first time
sd_ble_gap_conn_param_update is called (5 seconds). */
#define NEXT_CONN_PARAMS_UPDATE_DELAY   APP_TIMER_TICKS(30000)                      /**< Time
between each call to sd_ble_gap_conn_param_update after the first call (30 seconds). */
#define MAX_CONN_PARAMS_UPDATE_COUNT    3                                           /**< Number
of attempts before giving up the connection parameter negotiation. */

#define DEAD_BEEF                       0xDEADBEEF                                  /**< Value
used as error code on stack dump, can be used to identify stack location on stack unwind. */

#define UART_TX_BUF_SIZE                256                                         /**< UART TX
buffer size. */
#define UART_RX_BUF_SIZE                256                                         /**< UART RX
buffer size. */


BLE_NUS_DEF(m_nus, NRF_SDH_BLE_TOTAL_LINK_COUNT);                                   /**< BLE NUS
service instance. */
NRF_BLE_GATT_DEF(m_gatt);                                                           /**< GATT
module instance. */
NRF_BLE_QWR_DEF(m_qwr);                                                             /**< Context
for the Queued Write module.*/
BLE_ADVERTISING_DEF(m_advertising);                                                 /**<
Advertising module instance. */

static uint16_t   m_conn_handle          = BLE_CONN_HANDLE_INVALID;                 /**< Handle
of the current connection. */
static uint16_t   m_ble_nus_max_data_len = BLE_GATT_ATT_MTU_DEFAULT - 3;            /**< Maximum
length of data (in bytes) that can be transmitted to the peer by the Nordic UART service module.
*/
static ble_uuid_t m_adv_uuids[]          =                                          /**<
Universally unique service identifier. */
{
    {BLE_UUID_NUS_SERVICE, NUS_SERVICE_UUID_TYPE}
};


/**@brief Function for assert macro callback.
 *
 * @details This function will be called in case of an assert in the SoftDevice.
 *
 * @warning This handler is an example only and does not fit a final product. You need to
analyse
 *          how your product is supposed to react in case of Assert.
 * @warning On assert from the SoftDevice, the system can only recover on reset.
 *
 * @param[in] line_num    Line number of the failing ASSERT call.
 * @param[in] p_file_name File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
    app_error_handler(DEAD_BEEF, line_num, p_file_name);
}

/**@brief Function for initializing the timer module.
 */
static void timers_init(void)
{
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);
}

/**@brief Function for the GAP initialization.
 *
 * @details This function will set up all the necessary GAP (Generic Access Profile) parameters
of
 *          the device. It also sets the permissions and appearance.
 */
static void gap_params_init(void)
{
    uint32_t                err_code;
    ble_gap_conn_params_t   gap_conn_params;
    ble_gap_conn_sec_mode_t sec_mode;
```

```
    BLE_GAP_CONN_SEC_MODE_SET_OPEN(&sec_mode);

    err_code = sd_ble_gap_device_name_set(&sec_mode,
                                          (const uint8_t *) DEVICE_NAME,
                                          strlen(DEVICE_NAME));
    APP_ERROR_CHECK(err_code);

    memset(&gap_conn_params, 0, sizeof(gap_conn_params));

    gap_conn_params.min_conn_interval = MIN_CONN_INTERVAL;
    gap_conn_params.max_conn_interval = MAX_CONN_INTERVAL;
    gap_conn_params.slave_latency     = SLAVE_LATENCY;
    gap_conn_params.conn_sup_timeout  = CONN_SUP_TIMEOUT;

    err_code = sd_ble_gap_ppcp_set(&gap_conn_params);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling Queued Write Module errors.
 *
 * @details A pointer to this function will be passed to each service which may need to inform
the
 *          application about an error.
 *
 * @param[in]   nrf_error   Error code containing information about what went wrong.
 */
static void nrf_qwr_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for handling the data from the Nordic UART Service.
 *
 * @details This function will process the data received from the Nordic UART BLE Service and
send
 *          it to the UART module.
 *
 * @param[in] p_evt       Nordic UART Service event.
 */
/**@snippet [Handling the data received over BLE] */
static void nus_data_handler(ble_nus_evt_t * p_evt)
{

    if (p_evt->type == BLE_NUS_EVT_RX_DATA)
    {
        uint32_t err_code;

        NRF_LOG_DEBUG("Received data from BLE NUS. Writing data on UART.");
        NRF_LOG_HEXDUMP_DEBUG(p_evt->params.rx_data.p_data, p_evt->params.rx_data.length);

        for (uint32_t i = 0; i < p_evt->params.rx_data.length; i++)
        {
            do
            {
                err_code = app_uart_put(p_evt->params.rx_data.p_data[i]);
                if ((err_code != NRF_SUCCESS) && (err_code != NRF_ERROR_BUSY))
                {
                    NRF_LOG_ERROR("Failed receiving NUS message. Error 0x%x. ", err_code);
                    APP_ERROR_CHECK(err_code);
                }
            } while (err_code == NRF_ERROR_BUSY);
        }
        if (p_evt->params.rx_data.p_data[p_evt->params.rx_data.length - 1] == '\r')
        {
            while (app_uart_put('\n') == NRF_ERROR_BUSY);
        }
    }

}
/**@snippet [Handling the data received over BLE] */
```

```
/**@brief Function for initializing services that will be used by the application.
 */
static void services_init(void)
{
    uint32_t           err_code;
    ble_nus_init_t     nus_init;
    nrf_ble_qwr_init_t qwr_init = {0};

    // Initialize Queued Write Module.
    qwr_init.error_handler = nrf_qwr_error_handler;

    err_code = nrf_ble_qwr_init(&m_qwr, &qwr_init);
    APP_ERROR_CHECK(err_code);

    // Initialize NUS.
    memset(&nus_init, 0, sizeof(nus_init));

    nus_init.data_handler = nus_data_handler;

    err_code = ble_nus_init(&m_nus, &nus_init);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling an event from the Connection Parameters Module.
 *
 * @details This function will be called for all events in the Connection Parameters Module
 *          which are passed to the application.
 *
 * @note All this function does is to disconnect. This could have been done by simply setting
 *       the disconnect_on_fail config parameter, but instead we use the event handler
 *       mechanism to demonstrate its use.
 *
 * @param[in] p_evt  Event received from the Connection Parameters Module.
 */
static void on_conn_params_evt(ble_conn_params_evt_t * p_evt)
{
    uint32_t err_code;

    if (p_evt->evt_type == BLE_CONN_PARAMS_EVT_FAILED)
    {
        err_code = sd_ble_gap_disconnect(m_conn_handle, BLE_HCI_CONN_INTERVAL_UNACCEPTABLE);
        APP_ERROR_CHECK(err_code);
    }
}


/**@brief Function for handling errors from the Connection Parameters module.
 *
 * @param[in] nrf_error  Error code containing information about what went wrong.
 */
static void conn_params_error_handler(uint32_t nrf_error)
{
    APP_ERROR_HANDLER(nrf_error);
}


/**@brief Function for initializing the Connection Parameters module.
 */
static void conn_params_init(void)
{
    uint32_t               err_code;
    ble_conn_params_init_t cp_init;

    memset(&cp_init, 0, sizeof(cp_init));

    cp_init.p_conn_params                  = NULL;
    cp_init.first_conn_params_update_delay = FIRST_CONN_PARAMS_UPDATE_DELAY;
    cp_init.next_conn_params_update_delay  = NEXT_CONN_PARAMS_UPDATE_DELAY;
    cp_init.max_conn_params_update_count   = MAX_CONN_PARAMS_UPDATE_COUNT;
    cp_init.start_on_notify_cccd_handle    = BLE_GATT_HANDLE_INVALID;
    cp_init.disconnect_on_fail             = false;
    cp_init.evt_handler                    = on_conn_params_evt;
```

```
    cp_init.error_handler                  = conn_params_error_handler;

    err_code = ble_conn_params_init(&cp_init);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for putting the chip into sleep mode.
 *
 * @note This function will not return.
 */
static void sleep_mode_enter(void)
{
    uint32_t err_code = bsp_indication_set(BSP_INDICATE_IDLE);
    APP_ERROR_CHECK(err_code);

    // Prepare wakeup buttons.
    err_code = bsp_btn_ble_sleep_mode_prepare();
    APP_ERROR_CHECK(err_code);

    // Go to system-off mode (this function will not return; wakeup will cause a reset).
    err_code = sd_power_system_off();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling advertising events.
 *
 * @details This function will be called for advertising events which are passed to the
application.
 *
 * @param[in] ble_adv_evt  Advertising event.
 */
static void on_adv_evt(ble_adv_evt_t ble_adv_evt)
{
    uint32_t err_code;

    switch (ble_adv_evt)
    {
        case BLE_ADV_EVT_FAST:
            err_code = bsp_indication_set(BSP_INDICATE_ADVERTISING);
            APP_ERROR_CHECK(err_code);
            break;
        case BLE_ADV_EVT_IDLE:
            sleep_mode_enter();
            break;
        default:
            break;
    }
}


/**@brief Function for handling BLE events.
 *
 * @param[in]   p_ble_evt   Bluetooth stack event.
 * @param[in]   p_context   Unused.
 */
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    uint32_t err_code;

    switch (p_ble_evt->header.evt_id)
    {
        case BLE_GAP_EVT_CONNECTED:
            NRF_LOG_INFO("Connected");
            err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);
            APP_ERROR_CHECK(err_code);
            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
            err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr, m_conn_handle);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_GAP_EVT_DISCONNECTED:
            NRF_LOG_INFO("Disconnected");
```

```
            // LED indication will be changed when advertising starts.
            m_conn_handle = BLE_CONN_HANDLE_INVALID;
            break;

        case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
        {
            NRF_LOG_DEBUG("PHY update request.");
            ble_gap_phys_t const phys =
            {
                .rx_phys = BLE_GAP_PHY_AUTO,
                .tx_phys = BLE_GAP_PHY_AUTO,
            };
            err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GAP_EVT_SEC_PARAMS_REQUEST:
            // Pairing not supported
            err_code = sd_ble_gap_sec_params_reply(m_conn_handle,
BLE_GAP_SEC_STATUS_PAIRING_NOT_SUPP, NULL, NULL);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_GATTS_EVT_SYS_ATTR_MISSING:
            // No system attributes have been stored.
            err_code = sd_ble_gatts_sys_attr_set(m_conn_handle, NULL, 0, 0);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_GATTC_EVT_TIMEOUT:
            // Disconnect on GATT Client timeout event.
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                                             BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
            break;

        case BLE_GATTS_EVT_TIMEOUT:
            // Disconnect on GATT Server timeout event.
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                                             BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
            break;

        default:
            // No implementation needed.
            break;
    }
}


/**@brief Function for the SoftDevice initialization.
 *
 * @details This function initializes the SoftDevice and the BLE event interrupt.
 */
static void ble_stack_init(void)
{
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}
```

```
/**@brief Function for handling events from the GATT library. */
void gatt_evt_handler(nrf_ble_gatt_t * p_gatt, nrf_ble_gatt_evt_t const * p_evt)
{
    if ((m_conn_handle == p_evt->conn_handle) && (p_evt->evt_id ==
NRF_BLE_GATT_EVT_ATT_MTU_UPDATED))
    {
        m_ble_nus_max_data_len = p_evt->params.att_mtu_effective - OPCODE_LENGTH -
HANDLE_LENGTH;
        NRF_LOG_INFO("Data len is set to 0x%X(%d)", m_ble_nus_max_data_len,
m_ble_nus_max_data_len);
    }
    NRF_LOG_DEBUG("ATT MTU exchange completed. central 0x%x peripheral 0x%x",
                  p_gatt->att_mtu_desired_central,
                  p_gatt->att_mtu_desired_periph);
}


/**@brief Function for initializing the GATT library. */
void gatt_init(void)
{
    ret_code_t err_code;

    err_code = nrf_ble_gatt_init(&m_gatt, gatt_evt_handler);
    APP_ERROR_CHECK(err_code);

    err_code = nrf_ble_gatt_att_mtu_periph_set(&m_gatt, NRF_SDH_BLE_GATT_MAX_MTU_SIZE);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling events from the BSP module.
 *
 * @param[in]   event   Event generated by button press.
 */
void bsp_event_handler(bsp_event_t event)
{
    uint32_t err_code;
    switch (event)
    {
        case BSP_EVENT_SLEEP:
            sleep_mode_enter();
            break;

        case BSP_EVENT_DISCONNECT:
            err_code = sd_ble_gap_disconnect(m_conn_handle,
BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            if (err_code != NRF_ERROR_INVALID_STATE)
            {
                APP_ERROR_CHECK(err_code);
            }
            break;

        case BSP_EVENT_WHITELIST_OFF:
            if (m_conn_handle == BLE_CONN_HANDLE_INVALID)
            {
                err_code = ble_advertising_restart_without_whitelist(&m_advertising);
                if (err_code != NRF_ERROR_INVALID_STATE)
                {
                    APP_ERROR_CHECK(err_code);
                }
            }
            break;

        default:
            break;
    }
}


/**@brief   Function for handling app_uart events.
 *
```

```
 * @details This function will receive a single character from the app_uart module and append it
to
 *          a string. The string will be be sent over BLE when the last character received was a
 *          'new line' '\n' (hex 0x0A) or if the string has reached the maximum data length.
 */
/**@snippet [Handling the data received over UART] */
void uart_event_handle(app_uart_evt_t * p_event)
{
    static uint8_t data_array[BLE_NUS_MAX_DATA_LEN];
    static uint8_t index = 0;
    uint32_t       err_code;

    switch (p_event->evt_type)
    {
        case APP_UART_DATA_READY:
            UNUSED_VARIABLE(app_uart_get(&data_array[index]));
            index++;

            if ((data_array[index - 1] == '\n') ||
                (data_array[index - 1] == '\r') ||
                (index >= m_ble_nus_max_data_len))
            {
                if (index > 1)
                {
                    NRF_LOG_DEBUG("Ready to send data over BLE NUS");
                    NRF_LOG_HEXDUMP_DEBUG(data_array, index);

                    do
                    {
                        uint16_t length = (uint16_t)index;
                        err_code = ble_nus_data_send(&m_nus, data_array, &length,
m_conn_handle);
                        if ((err_code != NRF_ERROR_INVALID_STATE) &&
                            (err_code != NRF_ERROR_RESOURCES) &&
                            (err_code != NRF_ERROR_NOT_FOUND))
                        {
                            APP_ERROR_CHECK(err_code);
                        }
                    } while (err_code == NRF_ERROR_RESOURCES);
                }

                index = 0;
            }
            break;

        case APP_UART_COMMUNICATION_ERROR:
            APP_ERROR_HANDLER(p_event->data.error_communication);
            break;

        case APP_UART_FIFO_ERROR:
            APP_ERROR_HANDLER(p_event->data.error_code);
            break;

        default:
            break;
    }
}
/**@snippet [Handling the data received over UART] */


/**@brief  Function for initializing the UART module.
 */
/**@snippet [UART Initialization] */
static void uart_init(void)
{
    uint32_t                     err_code;
    app_uart_comm_params_t const comm_params =
    {
        .rx_pin_no    = RX_PIN_NUMBER,
        .tx_pin_no    = TX_PIN_NUMBER,
        .rts_pin_no   = RTS_PIN_NUMBER,
        .cts_pin_no   = CTS_PIN_NUMBER,
        .flow_control = APP_UART_FLOW_CONTROL_DISABLED,
        .use_parity   = false,
```

```
#if defined (UART_PRESENT)
        .baud_rate    = NRF_UART_BAUDRATE_115200
#else
        .baud_rate    = NRF_UARTE_BAUDRATE_115200
#endif
    };

    APP_UART_FIFO_INIT(&comm_params,
                       UART_RX_BUF_SIZE,
                       UART_TX_BUF_SIZE,
                       uart_event_handle,
                       APP_IRQ_PRIORITY_LOWEST,
                       err_code);
    APP_ERROR_CHECK(err_code);
}
/**@snippet [UART Initialization] */


/**@brief Function for initializing the Advertising functionality.
 */
static void advertising_init(void)
{
    uint32_t               err_code;
    ble_advertising_init_t init;

    memset(&init, 0, sizeof(init));

    init.advdata.name_type          = BLE_ADVDATA_FULL_NAME;
    init.advdata.include_appearance = false;
    init.advdata.flags              = BLE_GAP_ADV_FLAGS_LE_ONLY_LIMITED_DISC_MODE;

    init.srdata.uuids_complete.uuid_cnt = sizeof(m_adv_uuids) / sizeof(m_adv_uuids[0]);
    init.srdata.uuids_complete.p_uuids  = m_adv_uuids;

    init.config.ble_adv_fast_enabled  = true;
    init.config.ble_adv_fast_interval = APP_ADV_INTERVAL;
    init.config.ble_adv_fast_timeout  = APP_ADV_DURATION;
    init.evt_handler = on_adv_evt;

    err_code = ble_advertising_init(&m_advertising, &init);
    APP_ERROR_CHECK(err_code);

    ble_advertising_conn_cfg_tag_set(&m_advertising, APP_BLE_CONN_CFG_TAG);
}


/**@brief Function for initializing buttons and leds.
 *
 * @param[out] p_erase_bonds  Will be true if the clear bonding button was pressed to wake the
application up.
 */
static void buttons_leds_init(bool * p_erase_bonds)
{
    bsp_event_t startup_event;

    uint32_t err_code = bsp_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS, bsp_event_handler);
    APP_ERROR_CHECK(err_code);

    err_code = bsp_btn_ble_init(NULL, &startup_event);
    APP_ERROR_CHECK(err_code);

    *p_erase_bonds = (startup_event == BSP_EVENT_CLEAR_BONDING_DATA);
}


/**@brief Function for initializing the nrf log module.
 */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}
```

```c
/**@brief Function for initializing power management.
 */
static void power_management_init(void)
{
    ret_code_t err_code;
    err_code = nrf_pwr_mgmt_init();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling the idle state (main loop).
 *
 * @details If there is no pending log operation, then sleep until next the next event occurs.
 */
static void idle_state_handle(void)
{
    if (NRF_LOG_PROCESS() == false)
    {
        nrf_pwr_mgmt_run();
    }
}


/**@brief Function for starting advertising.
 */
static void advertising_start(void)
{
    uint32_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST);
    APP_ERROR_CHECK(err_code);
}


/**@brief Application main function.
 */
int main(void)
{
    bool erase_bonds;

    // Initialize.
    uart_init();
    log_init();
    timers_init();
    buttons_leds_init(&erase_bonds);
    power_management_init();
    ble_stack_init();
    gap_params_init();
    gatt_init();
    services_init();
    advertising_init();
    conn_params_init();

    // Start execution.
    NRF_LOG_INFO("Start Example 1");
    advertising_start();

    // Enter main loop.
    for (;;)
    {
        idle_state_handle();
    }
}
```

10.1.2 Subsistema 2

10.1.2.1 Código PCA10040

```
/**
 * @brief Environment sensor data and button press from Thingy:52 to nRf52xx DK application main file.
 *
 * This file contains the source code for a sample client application using the Thingy Button service and the Thingy Environment service.
 */

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "nrf_sdh.h"
#include "nrf_sdh_ble.h"
#include "nrf_sdh_soc.h"
#include "nrf_pwr_mgmt.h"
#include "app_timer.h"
#include "boards.h"
#include "bsp.h"
#include "bsp_btn_ble.h"
#include "ble.h"
#include "ble_hci.h"
#include "ble_advdata.h"
#include "ble_advertising.h"
#include "ble_conn_params.h"
#include "ble_db_discovery.h"
#include "ble_tbs_c.h"
#include "ble_tes_c.h"
#include "ble_bas_c.h"
#include "nrf_ble_gatt.h"

#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"


#define CENTRAL_SCANNING_LED            BSP_BOARD_LED_0                     /**< Scanning LED will be on when the device is scanning. */
#define CENTRAL_CONNECTED_LED           BSP_BOARD_LED_1                     /**< Connected LED will be on when the device is connected. */
#define LEDBUTTON_LED                   BSP_BOARD_LED_2                     /**< LED to indicate a change of state of the the Button characteristic on the peer. */

#define SCAN_INTERVAL                   0x00A0                              /**< Determines scan interval in units of 0.625 millisecond. */
#define SCAN_WINDOW                     0x0050                              /**< Determines scan window in units of 0.625 millisecond. */
#define SCAN_DURATION                   0x0000                              /**< Timout when scanning. 0x0000 disables timeout. */

#define MIN_CONNECTION_INTERVAL         MSEC_TO_UNITS(7.5, UNIT_1_25_MS)    /**< Determines minimum connection interval in milliseconds. */
#define MAX_CONNECTION_INTERVAL         MSEC_TO_UNITS(30, UNIT_1_25_MS)     /**< Determines maximum connection interval in milliseconds. */
#define SLAVE_LATENCY                   0                                   /**< Determines slave latency in terms of connection events. */
#define SUPERVISION_TIMEOUT             MSEC_TO_UNITS(4000, UNIT_10_MS)     /**< Determines supervision time-out in units of 10 milliseconds. */

#define LEDBUTTON_BUTTON_PIN            BSP_BUTTON_0                        /**< Button that will write to the LED characteristic of the peer */
#define BUTTON_DETECTION_DELAY          APP_TIMER_TICKS(50)                 /**< Delay from a GPIOTE event until a button is reported as pushed (in number of timer ticks). */

#define APP_BLE_CONN_CFG_TAG            1                                   /**< A tag identifying the SoftDevice BLE configuration. */
#define APP_BLE_OBSERVER_PRIO           3                                   /**< Application's BLE observer priority. You shouldn't need to modify this value. */
```

```
BLE_TBS_C_DEF(m_ble_tbs_c);                                     /**< Main structure used by the
TBS client module. */
BLE_BAS_C_DEF(m_ble_bas_c);                                     /**< Main structure used by the
BAS client module. */
BLE_TES_C_DEF(m_ble_tes_c);                                     /**< Main structure used by the
TES client module. */
NRF_BLE_GATT_DEF(m_gatt);                                       /**< GATT module instance. */
BLE_DB_DISCOVERY_DEF(m_db_disc);                                /**< DB discovery module
instance. */

static char const m_target_periph_name[] = "Sebas";             /**< Name of the device we try
to connect to. This name is searched in the scan report data*/

/**@brief Parameters used when scanning. */
static ble_gap_scan_params_t const m_scan_params =
{
    .active   = 1,
    .interval = SCAN_INTERVAL,
    .window   = SCAN_WINDOW,

    .timeout           = SCAN_DURATION,
    .scan_phys         = BLE_GAP_PHY_1MBPS,
    .filter_policy     = BLE_GAP_SCAN_FP_ACCEPT_ALL,
};

static uint8_t m_scan_buffer_data[BLE_GAP_SCAN_BUFFER_MIN]; /**< buffer where advertising
reports will be stored by the SoftDevice. */

/**@brief Pointer to the buffer where advertising reports will be stored by the SoftDevice. */
static ble_data_t m_scan_buffer =
{
    m_scan_buffer_data,
    BLE_GAP_SCAN_BUFFER_MIN
};

/**@brief Connection parameters requested for connection. */
static ble_gap_conn_params_t const m_connection_param =
{
    (uint16_t)MIN_CONNECTION_INTERVAL,
    (uint16_t)MAX_CONNECTION_INTERVAL,
    (uint16_t)SLAVE_LATENCY,
    (uint16_t)SUPERVISION_TIMEOUT
};


/**@brief Function to handle asserts in the SoftDevice.
 *
 * @details This function will be called in case of an assert in the SoftDevice.
 *
 * @warning This handler is an example only and does not fit a final product. You need to
analyze
 *          how your product is supposed to react in case of Assert.
 * @warning On assert from the SoftDevice, the system can only recover on reset.
 *
 * @param[in] line_num     Line number of the failing ASSERT call.
 * @param[in] p_file_name  File name of the failing ASSERT call.
 */
void assert_nrf_callback(uint16_t line_num, const uint8_t * p_file_name)
{
    app_error_handler(0xDEADBEEF, line_num, p_file_name);
}


/**@brief Function for the LEDs initialization.
 *
 * @details Initializes all LEDs used by the application.
 */
static void leds_init(void)
{
    bsp_board_init(BSP_INIT_LEDS);
}


/**@brief Function to start scanning.
```

```
 */
static void scan_start(void)
{
    ret_code_t err_code;

    (void) sd_ble_gap_scan_stop();

    err_code = sd_ble_gap_scan_start(&m_scan_params, &m_scan_buffer);
    APP_ERROR_CHECK(err_code);

    bsp_board_led_off(CENTRAL_CONNECTED_LED);
    bsp_board_led_on(CENTRAL_SCANNING_LED);
}


/**@brief Handles events coming from the Thingy Button central module.
 */
static void tbs_c_evt_handler(ble_tbs_c_t * p_tbs_c, ble_tbs_c_evt_t * p_tbs_c_evt)
{
    switch (p_tbs_c_evt->evt_type)
    {
        case BLE_TBS_C_EVT_DISCOVERY_COMPLETE:
        {
            ret_code_t err_code;

            err_code = ble_tbs_c_handles_assign(&m_ble_tbs_c,
                                                p_tbs_c_evt->conn_handle,
                                                &p_tbs_c_evt->params.peer_db);
            NRF_LOG_INFO("Thingy Button service discovered on conn_handle 0x%x.", p_tbs_c_evt-
>conn_handle);

            err_code = app_button_enable();
            APP_ERROR_CHECK(err_code);

            // Thingy Button service discovered. Enable notification of Button.
            err_code = ble_tbs_c_button_notif_enable(p_tbs_c);
            APP_ERROR_CHECK(err_code);
        } break; // BLE_TBS_C_EVT_DISCOVERY_COMPLETE

        case BLE_TBS_C_EVT_BUTTON_NOTIFICATION:
        {
            NRF_LOG_INFO("Button state changed on peer to 0x%x.", p_tbs_c_evt-
>params.button.button_state);
            if (p_tbs_c_evt->params.button.button_state)
            {
                bsp_board_led_on(LEDBUTTON_LED);
            }
            else
            {
                bsp_board_led_off(LEDBUTTON_LED);
            }
        } break; // BLE_TBS_C_EVT_BUTTON_NOTIFICATION

        default:
            // No implementation needed.
            break;
    }
}

/**@brief Handles events coming from the Thingy Environment central module.
 */
static void tes_c_evt_handler(ble_tes_c_t * p_tes_c, ble_tes_c_evt_t * p_tes_c_evt)
{

    switch (p_tes_c_evt->evt_type)
    {
        case BLE_TES_C_EVT_DISCOVERY_COMPLETE:
        {
            ret_code_t err_code;

            err_code = ble_tes_c_handles_assign(&m_ble_tes_c,
                                                p_tes_c_evt->conn_handle,
                                                &p_tes_c_evt->params.peer_db);
```

```
            NRF_LOG_INFO("Thingy Environment service discovered on conn_handle 0x%x.",
p_tes_c_evt->conn_handle);

            // Thingy Environment service discovered. Enable notification of sensor data.
            err_code = ble_tes_c_temperature_notif_enable(p_tes_c);
            APP_ERROR_CHECK(err_code);
            err_code = ble_tes_c_pressure_notif_enable(p_tes_c);
            APP_ERROR_CHECK(err_code);
            err_code = ble_tes_c_humidity_notif_enable(p_tes_c);
            APP_ERROR_CHECK(err_code);
            err_code = ble_tes_c_gas_notif_enable(p_tes_c);
            APP_ERROR_CHECK(err_code);
            err_code = ble_tes_c_color_notif_enable(p_tes_c);
            APP_ERROR_CHECK(err_code);
//            err_code = ble_tes_c_config_notif_enable(p_tes_c);
//            APP_ERROR_CHECK(err_code);

        } break; // BLE_TES_C_EVT_DISCOVERY_COMPLETE

        case BLE_TES_C_EVT_TEMPERATURE_NOTIFICATION:
        {
            ble_tes_temperature_t temperature = p_tes_c_evt->params.value.temperature_data;
            NRF_LOG_INFO("Got temperature: %d,%d", temperature.integer, temperature.decimal);
        } break; // BLE_TES_C_EVT_TEMPERATURE_NOTIFICATION
        case BLE_TES_C_EVT_PRESSURE_NOTIFICATION:
        {
            ble_tes_pressure_t pressure = p_tes_c_evt->params.value.pressure_data;
            NRF_LOG_INFO("Got pressure: %d,%d", pressure.integer, pressure.decimal);
        } break; // BLE_TES_C_EVT_PRESSURE_NOTIFICATION
        case BLE_TES_C_EVT_HUMIDITY_NOTIFICATION:
        {
            ble_tes_humidity_t humidity = p_tes_c_evt->params.value.humidity_data;
            NRF_LOG_INFO("Got humidity: %d.", humidity);
        } break; // BLE_TES_C_EVT_HUMIDITY_NOTIFICATION
        case BLE_TES_C_EVT_GAS_NOTIFICATION:
        {
            ble_tes_gas_t gas = p_tes_c_evt->params.value.gas_data;
            NRF_LOG_INFO("Got C02: %d.", gas.eco2_ppm);
            NRF_LOG_INFO("Got organic components: %d.", gas.tvoc_ppb);
        } break; // BLE_TES_C_EVT_GAS_NOTIFICATION
        case BLE_TES_C_EVT_COLOR_NOTIFICATION:
        {
            ble_tes_color_t color = p_tes_c_evt->params.value.color_data;
            NRF_LOG_INFO("Got color. R%d, G%d, B%d, C%d", color.red, color.green, color.blue,
color.clear);
        } break; // BLE_TES_C_EVT_COLOR_NOTIFICATION
        case BLE_TES_C_EVT_CONFIG_NOTIFICATION:
        {
            // No implementation.
        } break; // BLE_TES_C_EVT_CONFIG_NOTIFICATION

        default:
            // No implementation needed.
            break;
    }
}

/**@brief Handles events coming from the Thingy battery module.
 */
static void bas_c_evt_handler(ble_bas_c_t * p_bas_c, ble_bas_c_evt_t * p_bas_c_evt)
{
          switch (p_bas_c_evt->evt_type)
    {
        case BLE_BAS_C_EVT_DISCOVERY_COMPLETE:
        {
            ret_code_t err_code;

            err_code = ble_bas_c_handles_assign(&m_ble_bas_c,
                                                p_bas_c_evt->conn_handle,
                                                &p_bas_c->peer_bas_db);
            NRF_LOG_INFO("Thingy Battery service discovered on conn_handle 0x%x.", p_bas_c_evt-
>conn_handle);

            // Thingy Environment service discovered. Enable notification of sensor data.
```

```
            err_code = ble_bas_c_bl_notif_enable(p_bas_c);
            APP_ERROR_CHECK(err_code);
        } break; // BLE_TES_C_EVT_DISCOVERY_COMPLETE

        case BLE_BAS_C_EVT_BATT_NOTIFICATION:
        {
            uint8_t battLevel = p_bas_c_evt->params.battery_level;
            NRF_LOG_INFO("Got Battery Level: %d", battLevel);
        } break; // BLE_TES_C_EVT_TEMPERATURE_NOTIFICATION

                                case BLE_BAS_C_EVT_BATT_READ_RESP:
        {
            uint8_t battLevel = p_bas_c_evt->params.battery_level;
            NRF_LOG_INFO("RX Got Battery Level: %d", battLevel);
        } break; // BLE_TES_C_EVT_TEMPERATURE_NOTIFICATION
          }
}
/**@brief Function for handling the advertising report BLE event.
 *
 * @param[in] p_adv_report  Advertising report from the SoftDevice.
 */
static void on_adv_report(ble_gap_evt_adv_report_t const * p_adv_report)
{
    ret_code_t err_code;

    if (ble_advdata_name_find(p_adv_report->data.p_data,
                              p_adv_report->data.len,
                              m_target_periph_name))
    {
                                NRF_LOG_INFO("Sebas is found, initiate connection")
        // Name is a match, initiate connection.
        err_code = sd_ble_gap_connect(&p_adv_report->peer_addr,
                                      &m_scan_params,
                                      &m_connection_param,
                                      APP_BLE_CONN_CFG_TAG);
        APP_ERROR_CHECK(err_code);
    }
    else
    {
        err_code = sd_ble_gap_scan_start(NULL, &m_scan_buffer);
        APP_ERROR_CHECK(err_code);
    }
}


/**@brief Function for handling BLE events.
 *
 * @param[in]   p_ble_evt   Bluetooth stack event.
 * @param[in]   p_context   Unused.
 */
static void ble_evt_handler(ble_evt_t const * p_ble_evt, void * p_context)
{
    ret_code_t err_code;

    // For readability.
    ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;

    switch (p_ble_evt->header.evt_id)
    {
        // Upon connection, check which peripheral has connected (HR or RSC), initiate DB
        // discovery, update LEDs status and resume scanning if necessary. */
        case BLE_GAP_EVT_CONNECTED:
        {
            NRF_LOG_INFO("Connected.");
            err_code = ble_tbs_c_handles_assign(&m_ble_tbs_c, p_gap_evt->conn_handle, NULL);
            APP_ERROR_CHECK(err_code);

            err_code = ble_tes_c_handles_assign(&m_ble_tes_c, p_gap_evt->conn_handle, NULL);
            APP_ERROR_CHECK(err_code);

                        err_code = ble_bas_c_handles_assign(&m_ble_bas_c, p_gap_evt-
>conn_handle, NULL);
            APP_ERROR_CHECK(err_code);
```

```
            err_code = ble_db_discovery_start(&m_db_disc, p_gap_evt->conn_handle);
            APP_ERROR_CHECK(err_code);

            // Update LEDs status, and check if we should be looking for more
            // peripherals to connect to.
            bsp_board_led_on(CENTRAL_CONNECTED_LED);
            bsp_board_led_off(CENTRAL_SCANNING_LED);
        } break;

        // Upon disconnection, reset the connection handle of the peer which disconnected,
update
        // the LEDs status and start scanning again.
        case BLE_GAP_EVT_DISCONNECTED:
        {
            NRF_LOG_INFO("Disconnected.");
            scan_start();
        } break;

        case BLE_GAP_EVT_ADV_REPORT:
        {
            on_adv_report(&p_gap_evt->params.adv_report);
        } break;

        case BLE_GAP_EVT_TIMEOUT:
        {
            // We have not specified a timeout for scanning, so only connection attemps can
timeout.
            if (p_gap_evt->params.timeout.src == BLE_GAP_TIMEOUT_SRC_CONN)
            {
                NRF_LOG_DEBUG("Connection request timed out.");
            }
        } break;

        case BLE_GAP_EVT_CONN_PARAM_UPDATE_REQUEST:
        {
            // Accept parameters requested by peer.
            err_code = sd_ble_gap_conn_param_update(p_gap_evt->conn_handle,
                                        &p_gap_evt-
>params.conn_param_update_request.conn_params);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GAP_EVT_PHY_UPDATE_REQUEST:
        {
            NRF_LOG_DEBUG("PHY update request.");
            ble_gap_phys_t const phys =
            {
                .rx_phys = BLE_GAP_PHY_AUTO,
                .tx_phys = BLE_GAP_PHY_AUTO,
            };
            err_code = sd_ble_gap_phy_update(p_ble_evt->evt.gap_evt.conn_handle, &phys);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GATTC_EVT_TIMEOUT:
        {
            // Disconnect on GATT Client timeout event.
            NRF_LOG_DEBUG("GATT Client Timeout.");
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gattc_evt.conn_handle,
                                             BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
        } break;

        case BLE_GATTS_EVT_TIMEOUT:
        {
            // Disconnect on GATT Server timeout event.
            NRF_LOG_DEBUG("GATT Server Timeout.");
            err_code = sd_ble_gap_disconnect(p_ble_evt->evt.gatts_evt.conn_handle,
                                             BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION);
            APP_ERROR_CHECK(err_code);
        } break;

        default:
            // No implementation needed.
```

```
            break;
    }
}


/**@brief Thingy Button client initialization.
 */
static void tbs_c_init(void)
{
    ret_code_t       err_code;
    ble_tbs_c_init_t tbs_c_init_obj;

    tbs_c_init_obj.evt_handler = tbs_c_evt_handler;

    err_code = ble_tbs_c_init(&m_ble_tbs_c, &tbs_c_init_obj);
    APP_ERROR_CHECK(err_code);
}

/**@brief Thingy Environment client initialization.
 */
static void tes_c_init(void)
{
    ret_code_t       err_code;
    ble_tes_c_init_t tes_c_init_obj;

    tes_c_init_obj.evt_handler = tes_c_evt_handler;

    err_code = ble_tes_c_init(&m_ble_tes_c, &tes_c_init_obj);
    APP_ERROR_CHECK(err_code);
}


/**@brief Thingy Battery client initialization.
 */
static void bas_c_init(void)
{
    ret_code_t       err_code;
    ble_bas_c_init_t bas_c_init_obj;

    bas_c_init_obj.evt_handler = bas_c_evt_handler;

    err_code = ble_bas_c_init(&m_ble_bas_c, &bas_c_init_obj);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the BLE stack.
 *
 * @details Initializes the SoftDevice and the BLE event interrupts.
 */
static void ble_stack_init(void)
{
    ret_code_t err_code;

    err_code = nrf_sdh_enable_request();
    APP_ERROR_CHECK(err_code);

    // Configure the BLE stack using the default settings.
    // Fetch the start address of the application RAM.
    uint32_t ram_start = 0;
    err_code = nrf_sdh_ble_default_cfg_set(APP_BLE_CONN_CFG_TAG, &ram_start);
    APP_ERROR_CHECK(err_code);

    // Enable BLE stack.
    err_code = nrf_sdh_ble_enable(&ram_start);
    APP_ERROR_CHECK(err_code);

    // Register a handler for BLE events.
    NRF_SDH_BLE_OBSERVER(m_ble_observer, APP_BLE_OBSERVER_PRIO, ble_evt_handler, NULL);
}

/**@brief Function for handling database discovery events.
 *
```

```
 * @details This function is callback function to handle events from the database discovery
module.
 *          Depending on the UUIDs that are discovered, this function should forward the events
 *          to their respective services.
 *
 * @param[in] p_event  Pointer to the database discovery event.
 */
static void db_disc_handler(ble_db_discovery_evt_t * p_evt)
{
    ble_tbs_on_db_disc_evt(&m_ble_tbs_c, p_evt);
    ble_tes_on_db_disc_evt(&m_ble_tes_c, p_evt);
          ble_bas_on_db_disc_evt(&m_ble_bas_c, p_evt);
}

/**@brief Database discovery initialization.
 */
static void db_discovery_init(void)
{
    ret_code_t err_code = ble_db_discovery_init(db_disc_handler);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the log.
 */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}


/**@brief Function for initializing the timer.
 */
static void timer_init(void)
{
    ret_code_t err_code = app_timer_init();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the Power manager. */
static void power_management_init(void)
{
    ret_code_t err_code;
    err_code = nrf_pwr_mgmt_init();
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for initializing the GATT module.
 */
static void gatt_init(void)
{
    ret_code_t err_code = nrf_ble_gatt_init(&m_gatt, NULL);
    APP_ERROR_CHECK(err_code);
}


/**@brief Function for handling the idle state (main loop).
 *
 * @details Handle any pending log operation(s), then sleep until the next event occurs.
 */
static void idle_state_handle(void)
{
    NRF_LOG_FLUSH();
    nrf_pwr_mgmt_run();
}


int main(void)
{
```

```
    // Initialize.
    log_init();
    timer_init();
    leds_init();
    power_management_init();
    ble_stack_init();
    gatt_init();
    db_discovery_init();
          bas_c_init();
    tbs_c_init();
    tes_c_init();


    // Start execution.
    NRF_LOG_INFO("Start example 2 - Thingy:52 to nRF52 DK");
    scan_start();

    // Turn on the LED to signal scanning.
    bsp_board_led_on(CENTRAL_SCANNING_LED);

    // Enter main loop.
    for (;;)
    {
        idle_state_handle();
    }
}
```

10.2 BLE Services Thingy:52

10.2.1 Thingy configuration service

Name	UUID	Type	Data	Description
Base UUID	EF68xxxx-9B35-4933-9B10-52FFA9740042			
Thingy configuration service	0100			
Device name characteristic	0101	Write/Read	max 10 bytes	Device name as ASCII string.

Advertising param characteristic	0102	Write/Read	3 bytes	Advertising parameters (in units):
				• uint16_t - Adv interval in ms (unit 0.625 ms).
				  ◦ min 32 -> 20 ms
				  ◦ max 8000 -> 5 s
				• uint8_t - Adv timeout in s (unit 1 s).
				  ◦ min 0 -> 0 s
				  ◦ max 180 s -> 3 min

Connection param characteristic	0104	Write/Read	8 bytes	Connection parameters:
				• uint16_t - Min connection interval (unit 1.25 ms).
				  ◦ min 6 -> 7.5 ms
				  ◦ max 3200 -> 4 s
				• uint16_t - Max connection interval (unit 1.25 ms).
				  ◦ Same as above.
				• uint16_t - Slave latency (number of connection events).
				  ◦ Range 0-499
				• uint16_t - Supervision timeout (unit 10 ms).
				  ◦ Min 10 -> 100 ms
				  ◦ Max 3200 -> 32 s
				The following constraint applies: conn_sup_timeout * 4 > (1 + slave_latency) * max_conn_interval that corresponds to the following Bluetooth Spec requirement: The Supervision_Timeout in milliseconds must be larger than (1 + Conn_Latency) * Conn_Interval_Max * 2, where Conn_Interval_Max is given in milliseconds.

Eddystone URL	0105	Write/Read	3-17 bytes	Eddystone URL. Data format: • Byte 0: URL Scheme Prefix • Bytes 1+: The rest of the URL using Eddystone URL encoding Write 0 bytes to disable Eddystone advertising. See Eddystone-URL for more information.
Cloud token	0106	Write/Read	max 250 bytes	Cloud token to be used for identification against cloud services.
FW version	0107	Read	3 bytes	• uint8_t - major • uint8_t - minor • uint8_t - patch

MTU request	0108	Write/Read req	3 bytes	MTU request. Not retained.
				• uint8_t - Peripheral exchange request
				  ◦ 0x00 - Peripheral does not send an MTU exchange request. (Default behavior)
				  ◦ 0x01 - Peripheral sends an MTU exchange request, when written to 0x01.
				• uint16_t - MTU size requested (23 - 276 bytes).

NFC Tag content characteristic	0109	Write/Read	max 247 bytes	Content of NFC tag.

10.2.2 Environment service

Name	UUID	Type	Data	Description
Base UUID	EF68xxxx-9B35-4933-9B10-52FFA9740042			
Weather station service	0200			
Temperature characteristic	0201	Notify	2 bytes	Temperature in Celsius • int8_t - integer • uint8_t - decimal
Pressure characteristic	0202	Notify	5 bytes	Pressure in hPa • int32_t - integer • uint8_t - decimal
Humidity characteristic	0203	Notify	1 bytes	Relative humidity in % • uint8_t - RH
Gas (Air quality) characteristic	0204	Notify	4 bytes	• uint16_t - eCO2 ppm • uint16_t - TVOC ppb
Color characteristic	0205	Notify	8 bytes	• uint16_t - Red • uint16_t - Green • uint16_t - Blue • uint16_t - Clear

Configuration characteristic	0206	Write/Read	12 bytes	• uint16_t - Temperature interval in ms (100 ms - 60 s).
				• uint16_t - Pressure interval in ms (50 ms - 60 s).
				• uint16_t - Humidity interval in ms (100 ms - 60 s).
				• uint16_t - Color interval in ms (200 ms - 60 s).
				• uint8_t - Gas mode
				  ◦ 1 = 1 s interval
				  ◦ 2 = 10 s interval
				  ◦ 3 = 60 s interval
				• Color sensor LED calibration:
				  ◦ uint8_t - Red intensity [0 - 255]
				  ◦ uint8_t - Green intensity [0 - 255]
				  ◦ uint8_t - Blue intensity [0 - 255]

10.2.3 User Interface service

Name	UUID	Type	Data	Description
Base UUID	EF68xxxx-9B35-4933-9B10-52FFA9740042			
UI service	0300			

LED characteristic	0301	Write req/Read	Max 5 bytes	RGB Value:
				• uint8_t - Mode (retained for BLE connected and disconnected):
				  ◦ 0 = Off
				  ◦ 1 = Constant
				  ◦ 2 = Breathe
				  ◦ 3 = One Shot
				Constant mode (retained for BLE connected):
				• uint8_t - R intensity (0 - 255)
				• uint8_t - G intensity (0 - 255)
				• uint8_t - B intensity (0 - 255)
				Breathe mode (retained for BLE connected):
				• uint8_t - color:
				  ◦ 0x01 - RED
				  ◦ 0x02 - GREEN
				  ◦ 0x03 - YELLOW
				  ◦ 0x04 - BLUE
				  ◦ 0x05 - PURPLE
				  ◦ 0x06 - CYAN
				  ◦ 0x07 - WHITE
				• uint8_t - intensity (1-100) [%]
				• uint16_t - delay [ms] (50 ms - 10 s)
				One shot mode (retained for BLE connected):
				• uint8_t - color:
				  ◦ 0x01 - RED
				  ◦ 0x02 - GREEN
				  ◦ 0x03 - YELLOW
				  ◦ 0x04 - BLUE
				  ◦ 0x05 - PURPLE
				  ◦ 0x06 - CYAN
				  ◦ 0x07 - WHITE
				• uint8_t - intensity (1-100) [%]
				Default connected behavior:
				• Mode: Breathe
				• Color: CYAN
				• Intensity: 20%
				• Delay: 3500 ms

Button characteristic	0302	Notify	1 byte	Boolean button state: • 0: Button released • 1: Button pressed
EXT pin characteristic	0303	Write req/Read	4 bytes	External pin control. Range 0 - 255. 0 is off, 255 is on. PWM is not implemented. So either 0 or 255. • uint8_t - MOS_1 • uint8_t - MOS_2 • uint8_t - MOS_3 • uint8_t - MOS_4

10.2.4 Motion service

Name	UUID	Type	Data	Description
Base UUID	EF68xxxx-9B35-4933-9B10-52FFA9740042			
Thingy motion service	0400			

Config characteristic	0401	Write/Read	9 bytes	Motion configuration:
				• uint16_t - Step counter interval in ms (100 ms - 5 s).
				• uint16_t - Temperature compensation interval in ms (100 ms - 5 s).
				• uint16_t - Magnetometer compensation interval in ms (100 ms - 1 s).
				• uint16_t - Motion processing unit frequency in Hz (5 - 200 Hz).
				• uint8_t - Wake-on-motion
				  ◦ 0x00 = Off
				  ◦ 0x01 = On

Tap characteristic	0402	Notify	2 bytes	Direction and count of taps:
				• uint8_t - Direction:
				  ◦ 0x01 = TAP_X_UP
				  ◦ 0x02 = TAP_X_DOWN
				  ◦ 0x03 = TAP_Y_UP
				  ◦ 0x04 = TAP_Y_DOWN
				  ◦ 0x05 = TAP_Z_UP
				  ◦ 0x06 = TAP_Z_DOWN
				• uint8_t - Count
				  ◦ Number of taps in the given direction

Orientation characteristic	0403	Notify	1 byte	• uint8_t - Orientation
				  ◦ 0x00 = Portrait
				  ◦ 0x01 = Landscape
				  ◦ 0x02 = Reverse portrait
				  ◦ 0x03 = Reverse landscape

Quaternion characteristic	0404	Notify	16 bytes	Attitude represented with quaternions (2Q30 fixed point): • int32_t - w • int32_t - x • int32_t - y • int32_t - z
Step counter characteristic	0405	Notify	8 bytes	Step counter: • uint32_t - Steps • uint32_t - Time [ms]

Raw data characteristic	0406	Notify	18 bytes	Motion sensor raw data:
				• Accelerometer
				  ◦ int16_t - x [G] (6Q10 fixed point)
				  ◦ int16_t - y [G] (6Q10 fixed point)
				  ◦ int16_t - z [G] (6Q10 fixed point)
				• Gyroscope
				  ◦ int16_t - x [deg/s] (11Q5 fixed point)
				  ◦ int16_t - y [deg/s] (11Q5 fixed point)
				  ◦ int16_t - z [deg/s] (11Q5 fixed point)
				• Compass
				  ◦ int16_t - x [µT] (12Q4 fixed point)
				  ◦ int16_t - y [µT] (12Q4 fixed point)
				  ◦ int16_t - z [µT] (12Q4 fixed point)

Euler characteristic	0407	Notify	12 bytes	Attitude represented in Euler angles (16Q16 fixed point) • int32_t - roll [degrees] • int32_t - pitch [degrees] • int32_t - yaw [degrees]
Rotation matrix char	0408	Notify	18 bytes	Attitude in rotation matrix (2Q14 fixed point) • int16_t [9] - 3 x 3 matrix
Heading characteristic	0409	Notify	4 bytes	Heading (16Q16 fixed point) • int32_t - Heading [degrees]
Gravity vector	040A	Notify	12 bytes	Attitude represented by a gravity vector: • float - x • float - y • float - z

10.2.5 Sound service

Name	UUID	Type	Data	Description
Base UUID	EF68xxxx-9B35-4933-9B10-52FFA9740042			
Thingy sound service	0500			

Config characteristic	0501	Write/Read	2 bytes	Sound configuration
				• uint8_t - Speaker mode:
				  ◦ 0x01 - Frequency and duration.
				  ◦ 0x02 - 8-bit PCM
				  ◦ 0x03 - Sample
				• uint8_t - Microphone mode
				  ◦ 0x01 - ADPCM
				  ◦ 0x02 - SPL

Speaker data characteristic	0502	Write without resp	Max 273 bytes	In frequency mode:
				• uint16_t - Frequency in Hz (0 - stop forever).
				• uint16_t - Duration in ms (0 = forever).
				• uint8_t - Volume in % (0-100).
				In PCM mode:
				• uint8_t [20 - 273] 8-bit PCM samples
				In sample mode:
				• uint8_t - sample ID (0x00 - 0x08)
				  ◦ 0 Collect_Point_00.wav
				  ◦ 1 Collect_Point_01.wav
				  ◦ 2 Explosion_02.wav
				  ◦ 3 Explosion_04.wav
				  ◦ 4 Hit_00.wav
				  ◦ 5 Pickup_01.wav
				  ◦ 6 Pickup_03.wav
				  ◦ 7 Shoot_00.wav
				  ◦ 8 Shoot_01.wav

Speaker status characteristic	0503	Notify	1 byte	Speaker status • uint8_t - 0x00 - Finished • uint8_t - 0x01 - Buffer warning • uint8_t - 0x02 - Buffer ready • uint8_t - 0x10 - Packet disregarded • uint8_t - 0x11 - Invalid command
Microphone characteristic	0504	Notify	Max 273 bytes	ADCPM mode • ADPCM frame

10.2.6. Battery service

Name	UUID	Type	Data	Description
UUID	180F	Notify/Read	1 byte	• uint8_t State of charge [%] (0-100) Uses a lookup table to convert from battery voltage to state of charge (SoC). Due to the ADC configuration and battery model, certain percentage values are skipped. Will only update/notify if there is a change in remaining battery level.

10.2.7. DFU service

Name	UUID	Type	Data	Description
Secure DFU Service	FE59			
Base UUID	0000xxxx-0000-1000-8000-00805f9b34fb			
DFU Control Point characteristic	8EC90001-F315-4F60-9FB8-838830DAEA50	Notify/Write	1 byte	Steps to put Thingy into Buttonless DFU mode: • Enable notifications for the DFU Control point characteristic • Write 0x01 to the characteristic