Trabajo Final de Máster

Máster Universitario en Ingeniería Industrial

Puerta de enlace CANFD-Bluetooth Low Energy basada en Arduino

MEMORIA

Autor: Ana Gómez Gejo
Director: Manuel Moreno Eguílaz

Convocatoria: Abril 2022

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Resumen

En este documento se describe el proceso de diseño, desarrollo y validación de una puerta de enlace CAN FD – BLE basada en Arduino. El objetivo principal de este proyecto es la construcción de un prototipo funcional, capaz de leer mensajería CAN FD y enviarla a un dispositivo receptor mediante *Bluetooth Low Energy* (BLE).

En primer lugar, se realiza un breve estudio de mercado para conocer las soluciones que existen en el mercado, detectar sus puntos fuertes y débiles. El estudio se realiza con la finalidad de conocer qué factor determinante debería incluir el prototipo de este proyecto.

En segundo lugar, se realiza una breve explicación teórica de los distintos protocolos de comunicación: CAN, CAN FD, Bluetooth y Bluetooth Low Energy. Se incluye su origen, motivación y aplicación. En el caso de CAN, o en su versión más avanzada CAN FD, también incluye una breve explicación de la arquitectura de una trama/mensaje de CAN, ya que es esencial para la comprensión de este proyecto.

En tercer lugar, se detallan los componentes que conforman la solución propuesta y cómo se interconectan. Se presentan las placas utilizadas, los lenguajes de programación y las librerías empleadas.

Una vez presentada la solución, se explica paso a paso el proceso seguido durante el desarrollo del dispositivo y las pruebas realizadas para comprobar el correcto funcionamiento. Esta parte del documento incluye una explicación de los problemas encontrados y cómo se han solventado.

Los últimos aspectos incluidos en la memoria corresponden a la planificación, estimación de costes e impacto ambiental del proyecto.

Índice

1. Glosario

BLE: *Bluetooth Low Energy*

CAN: *Controller Area Network*

CAN FD: *Controller Area Network with Flexible Data-Rate*

CPU: *Central Processing Unit*

ECU: *Electronic Control Unit*

ISM: *Industrial, Scientific and Medical portions of the radio spectrum*.

SPI: *Serial Peripheral Interface*

UUID: *Universally Unique Identifier*

2. Presentación

2.1. Origen del proyecto

Hace aproximadamente tres décadas, los fabricantes de automóviles conectaban todos los dispositivos utilizando cableado punto a punto. Este método proporcionaba buenos resultados debido a los pocos sistemas eléctricos y electrónicos existentes en un automóvil, ya que aparte del sistema de gestión del motor, la mayoría eran sistemas puramente mecánicos.

Con el paso del tiempo fue necesario introducir nuevos sistemas electrónicos, para aumentar la seguridad y funcionalidad de los automóviles. La solución de interconectarlos punto a punto dejó de ser viable por distintos motivos como, por ejemplo:

- Aumento considerable del cableado
- Falta de espacio
- Aumento del peso
- Aumento del coste

Llegados a este punto, fue necesario desarrollar una nueva arquitectura del vehículo.
La utilización de un bus de comunicaciones se hizo esencial para poder reducir en gran medida el cableado en los vehículos. En la Figura 1 se ejemplifica esta problemática y la solución propuesta.

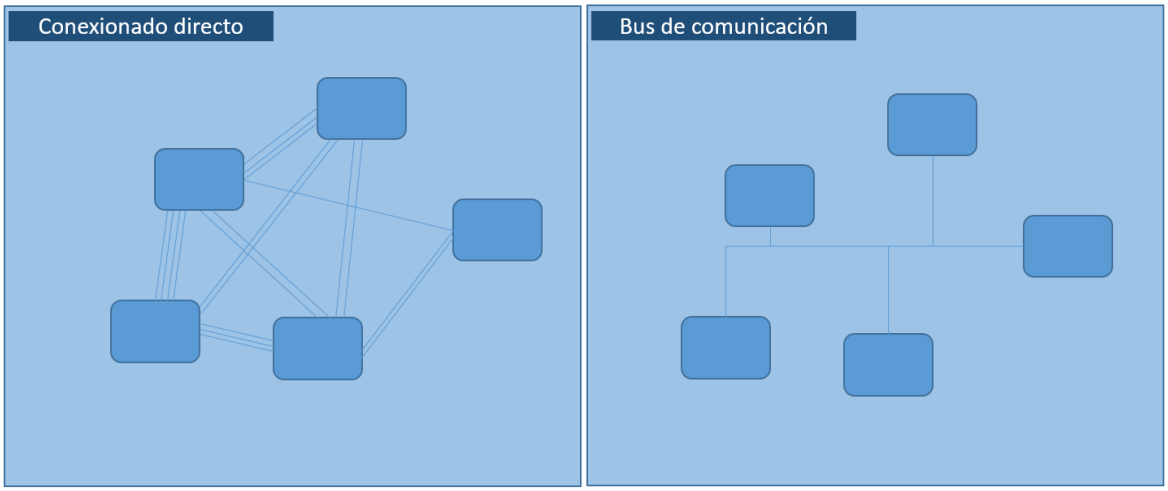

Figura 1.- Representación conexionado directo y bus de comunicación. Fuente: propia.

Basándose en esta idea, a finales de los 80, Bosch creó el protocolo de comunicaciones CAN bus para automoción [1] (En este trabajo se expondrá más detalladamente en qué

consiste). En concreto, nos centraremos en la versión más evolucionada, el CAN FD.

Este proyecto nace con la idea de facilitar la comunicación inalámbrica entre el vehículo y el usuario. Actualmente existen varios protocolos de comunicación en los vehículos, pero el más utilizado en automoción es el CAN (o CAN FD en su versión más avanzada). Por lo tanto, para poder analizar el estado de un vehículo, es esencial tener la capacidad de leer la información que viaja por un bus CAN o CAN FD.

Por otro lado, no sólo basta con leer, también es necesario un proceso de envío y recepción para que el usuario pueda tener la información.

Actualmente, dos de los protocolos de comunicación inalámbrica más comunes son el WiFi y el Bluetooth. La idea es construir un prototipo capaz de leer CAN y mostrarlo en un dispositivo. Mi compañera Marta de Pfaff se encargó de hacer la puerta de enlace entre CANFD y WiFi [2]. La idea es desarrollar algo similar, pero entre CAN FD y Bluetooth Low Energy (BLE).

2.2. Motivación

A finales de 2019 tuve la posibilidad de realizar las prácticas curriculares del Grado en Ingeniería en Tecnologías Industriales en SEAT, concretamente en el Departamento de Desarrollo Electrónico y Validación de Iluminación.

Durante esa estancia como becaria descubrí que el mundo de la automoción es muy amplio y cada vez más diverso. Si observamos la evolución, los vehículos cada vez integran más funciones, la mayoría de ellas basadas en la electrónica.

Otro punto importante para mí es el hecho que se trata de un trabajo que combina parte teórica con parte práctica. Para poder retener bien los conocimientos considero esencial tener una parte práctica en la que poder “pelearse”. De esta forma, podré poner en práctica los conocimientos de programación en C++ y Python adquiridos durante el grado y el máster.

Este trabajo final de máster (TFM) me permite profundizar más en el mundo de la automoción y, a la vez, adquirir nuevos conocimientos sobre dos protocolos de comunicación y lenguajes de programación ampliamente utilizados en el mundo laboral.

3. Introducción

3.1. Objetivos del proyecto

El objetivo principal del proyecto es desarrollar y construir una puerta de enlace que permita enviar los mensajes que circulan por un bus CAN FD a un sistema que disponga de comunicaciones inalámbricas basadas en Bluetooth Low Energy. A partir de este objetivo principal nacen los siguientes objetivos secundarios:

- Conocer los distintos protocolos de comunicación que se van a utilizar y sus variantes más avanzadas.
- Buscar una solución que sea funcional y más económica que las que hay actualmente en el mercado, realizando un estudio de mercado.
- Adquirir una metodología para el desarrollo y validación del dispositivo.

3.2. Estudio de mercado

Actualmente existen varios dispositivos que realizan la función mencionada. A continuación, se listan los dispositivos más comunes, junto con una breve descripción.

- **CANblue II**: Dispositivo creado por Ixxat que permite transmitir datos CAN por Bluetooth de manera rápida y confiable. Es una solución robusta y segura para comunicaciones inalámbricas. Este dispositivo está específicamente diseñado para aplicaciones industriales más que automoción [3].

  Aunque la mayoría de los dispositivos CANBlue II funcionan como enlace entre CAN clásico y Bluetooth Low Energy (BLE), la empresa fabrica bajo demanda dispositivos con controlados de CAN FD (no solo controladores de CAN clásico [4]. Ofrece la posibilidad de funcionar como un puente entre dos protocolos de comunicación (CAN/Bluethooth), puente universal e incluye una interfaz de progamación para PC [5]. Permite gestionar los datos recibidos y filtrarlos. En el distribuidor *RS Components* tiene un precio de 390 € [6].

Figura 2.- Ixxat CANblueII. Fuente: [4].

- **CAN-Bluetooth Gateway AX141100**: Dispositivo de Axiomatic Global Electronic Solutions. Dispositivo específicamente diseñado para automoción para enviar datos de CAN por Bluetooth (Clásico o BLE) a un ordenador, teléfono móvil, pantalla… [7]. En comparación con el dispositivo anterior, se trata de una solución más pequeña y manejable, pero con menor funcionalidad (solo puede enviar información, no puede tratar los datos, solo recibe y transmite).

  Presenta un puerto para conexión de CAN. En este caso, el conector TE Deutsch de 8 pines en lugar del ampliamente utilizado DB-9. Rango de precio no disponible.

Figura 3.- AX141100 CAN-Bluetooth Gateway. Fuente: [7].

- **GCAN-203 Bluetooth to CAN converter:** En dispositivo de GCAN, convierte CAN o CANFD a bluetooth clásico (v2.0) pero no tiene la opción de *low-energy*. Las aplicaciones más habituales de estos dispositivos son las siguientes [8]:
  - Monitoreo de un bus de comunicación de una línea de producción
  - Monitoreo de datos de un vehículo o aparato médico en una aplicación móvil a tiempo real

- Monitoreo remoto por bluetooth de un conjunto de dispositivos electrónicos

En precio aproximado de este producto en Amazon es de 200 €.

Figura 4.- GCAN-203 Bluetooth CAN Gateway. Fuente: [8].

Existen dispositivos ODS que permiten leer determinada información del coche, pero están limitados a unas direcciones de diagnosis en concreto. Lo que se pretende conseguir en este proyecto es poder leer la información de un bus CAN FD y también poder enviar (no solo leer).

Es en este punto donde nos interesa construir un dispositivo de enlace entre CANFD – BLE que sea funcional y más barato que las soluciones comerciales existentes.

3.3. Alcance del proyecto

Con la realización de este proyecto se pretende construir un dispositivo funcional capaz de trabajar como puerta de enlace entre un bus de comunicación CANFD y un dispositivo conectado por Bluetooth Low Energy (BLE).

Dentro de este proyecto se engloban los siguientes puntos:

- Estudio del mercado para ver las soluciones actuales
- Conocer los distintos protocolos de comunicación que se van a emplear
- Definir una solución e implementarla
- Validar el funcionamiento del dispositivo

Quedaría fuera del alcance de este proyecto:

- Crear una aplicación para la gestión de los datos enviados/recibidos por BLE.

4. Protocolos de comunicación

4.1. CAN

CAN (Controller Area Network) es un protocolo de comunicaciones desarrollado por la firma alemana Bosch a inicios de 1980, basado en una topología bus para la transmisión de mensajes en entornos distribuidos. Permite gestionar la comunicación entre distintas unidades de control, en automoción, también conocidas como CPU o ECU, sin necesidad de disponer de una unidad central [1].

Como se ha mencionado anteriormente, originalmente fue diseñado con el fin de simplificar y ahorrar en el cableado de los vehículos. Desde su origen a principios de los 80, se ha estandarizado a través de ISO11898 e ISO11519, y se ha convertido en un protocolo estándar para la mayoría de los fabricantes de automóviles [9]. Con los años, se ha implementado en otros contextos industriales.

Para cada dispositivo conectado al bus CAN, los datos de una trama se transmiten de forma serie y son recibidos por todos los dispositivos, incluido el que transmite. Existe un orden de prioridad por si se da el caso en que más de un dispositivo transmite al mismo tiempo (el de mayor prioridad transmite mientras que el resto queda a la espera).

Físicamente, la transmisión de señales en un bus CAN se lleva a cabo a través de dos cables trenzados, conocidos como CAN_H (CAN High) y CAN_L (CAN Low). El estado viene determinado por la diferencia de potencial entre ambos. El bus tiene dos estados definidos [10]:

- Estado recesivo (*common-mode voltaje*): Mismo nivel de tensión de los dos cables. Valor lógico 1.
- Estado dominante: Diferencia de tensión entre los dos cables > 1,5V. Valor lógico 0.

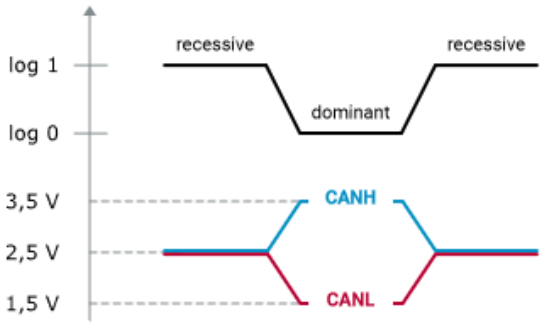

Figura 5.- Niveles de tensión del bus CAN. Fuente: [10].

Existen los siguientes tipos de tramas:

- Trama de datos (*data frame*) – Envío de datos.
- Trama remota (*remote frame*) – Solicitar envío de datos a un nodo.
- Trama de error (*error frame*) – Se transmite cuando un nodo detecta un mensaje erróneo.
- Trama de sobrecarga (*overload frame*) – Es transmitida por un nodo que se encuentra muy ocupado.

En el presente proyecto, nos centraremos en leer y enviar tramas de datos. En la Figura 6 se puede ver el ejemplo de una trama de datos CAN formato base, tanto la señal física recibida como los valores lógicos que le corresponden.

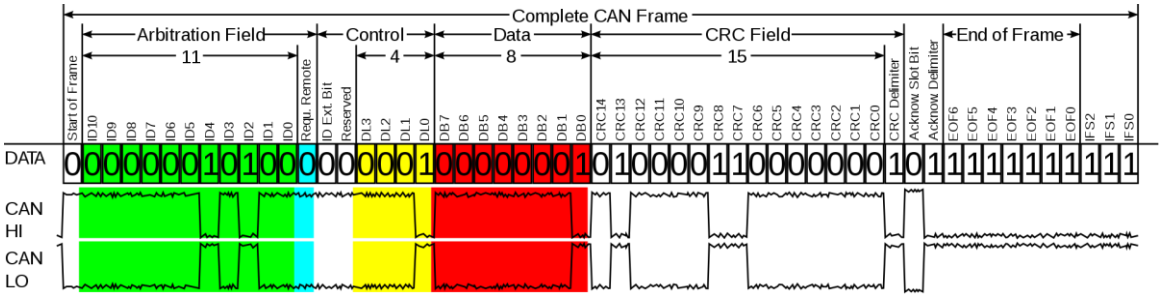

Figura 6.- Trama de datos de CAN en formato base. Fuente: [11].

Nombre del campo	Nº bits	Finalidad
SOF (start-of-frame)	1	Indica el inicio de la transmisión
IdentificadorA - ID	11	Un identificador (único) que también representa la prioridad de la trama
RTR – Petición de transmisión remota	1	Dominante (0) para tramas de datos, recesivo (1) para tramas de peticiones remotas
IDE - Bit de extensión de identificador	1	Dominante (0) para formato base, recesivo (1) para formato extendido
Bit reservado R0	1	Bit reservado. Debe ser dominante (0) pero se pueden aceptar ambos.
DLC- Código de longitud de datos	4	Número de bytes de datos del mensaje, entre 0 y 8. Si este campo es mayor que 8, el mensaje será de 8 bytes como máximo.
Campo de datos	0-64	Datos de la trama (la longitud viene dada por el DLC)
CRC	15	Código que verifica que los datos fueron enviados correctamente
Delimitador CRC	1	Debe ser recesivo (1)
Hueco de acuse de recibo	1	El transmisor emite recesivo (1) y cualquier receptor emite dominante (0)
Delimitador ACK	1	Debe ser recesivo (1)
Fin de trama EOF	1	Debe ser recesivo (1)

Tabla 1.- Campos de una trama de datos en formato base. Fuente: [11].

También existen las tramas de datos en formato extendido. La principal diferencia es que aparece un segundo conjunto de 18 bits que se utilizan para completar el identificador. Se obtiene un identificador de 29 bits en total. Se puede ampliar la información del tipo de tramas en el protocolo CAN en el siguiente enlace [12].

4.2. CAN FD

Cuando el estándar de CAN fue definido, el número de componentes electrónicos en un automóvil era reducido y con los parámetros que se definieron eran suficientes. Con los años, el número de centralitas en un automóvil se incrementó de forma considerable provocando que la limitación de la tasa de bits y de la cantidad de datos (*payload*) del CAN empezó a impedir actividades [13].

A partir de esta problemática y el trabajo de Bosch, junto con las empresas del sector del automóvil, en 2011 nació el protocolo CAN FD (Controller Area Network Flexible Data).

La principal diferencia entre ambos es que el CANFD permite variar la velocidad (tasa de bits) y ampliar la longitud de los datos. Es decir, los mensajes pueden tener mayor longitud y se pueden transmitir a una velocidad superior (permite empaquetar más datos en una misma trama).

El protocolo mejorado supera los límites de CAN clásico [1]:

- Permite transmitir datos más rápido que 1Mbit/s, en teoría hasta 8 Mbit/s.
- Mayor longitud de datos (*payload*): hasta un máximo de 64 bytes.

La idea es sencilla. Cuando un nodo se encuentra transmitiendo, la tasa de bits puede aumentar porque no es necesario sincronizar ningún nodo. La sincronización se realiza al inicio de la transmisión, no durante la misma. Este aumento de la tasa de bits permite empaquetar más datos en la misma unidad de tiempo.

El uso de una relación de 1:8 para las tasas de bits en la fase de arbitraje y de datos provoca un rendimiento aproximado seis veces mayor (un poco inferior a 8 veces debido a que las tramas de CANFD utilizan más bits en la cabecera y en el campo CRC) [1].

Como resumen, en la Tabla 2 se comparan los dos protocolos de comunicación.

	Bus CAN	Bus CAN FD (Flexible Date-rate)
Nomenclatura	CAN Clásico	CAN FD
Máxima velocidad de datos	1 Mbit/s	8 Mbit/s
Máxima carga de datos (payload)	8 bytes	64 bytes
Tramas admitidas	Solo admite tramas CAN clásico	Admite tramas de CAN clásico y CAN FD

Tabla 2.- Comparativa CAN vs CAN FD. Fuente: propia.

El aumento de la velocidad de datos y el mayor *payload* se logran modificando el formato de la trama de CAN. En la Figura 7 tenemos pequeña comparación entre los bits de una trama de datos de CAN clásica y una trama de datos CAN FD (ambas en formato base).

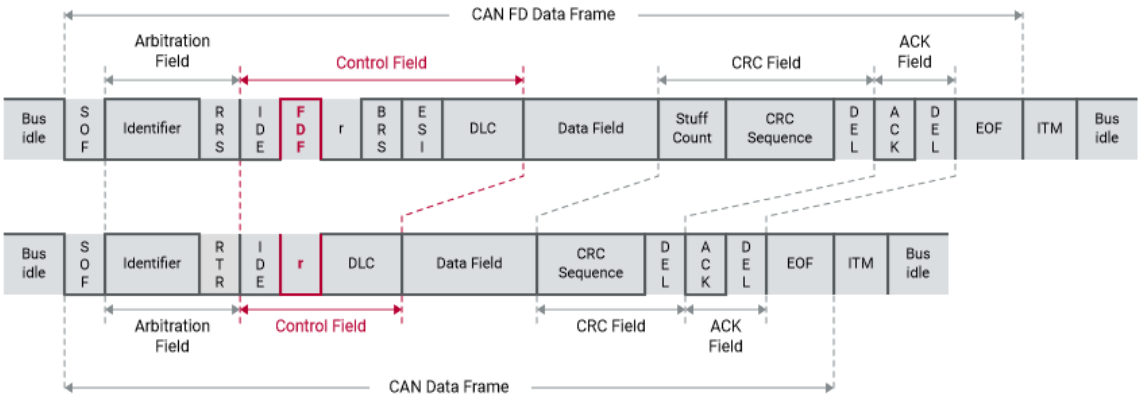

Figura 7.- Descripción de los bits en una trama de CAN vs CANFD. Fuente: [10].

Existen algunas diferencias entre una trama de datos de CAN clásico y una trama de CAN FD, sobre todo en los bits de la parte de arbitraje. Lo más importante para este proyecto, que después se deberá tener en cuenta a la hora de escribir el código para generar los mensajes de CAN FD emulados, se encuentra en los 4 bits destinados al DLC.

CAN FD utiliza un nuevo método de codificación DLC para indicar la longitud de los datos del mensaje (empleado los 4bits de control). Para mensajes inferiores a 8 bytes, utiliza la misma codificación que los mensajes de CAN clásicos. Para mensajes de mayor longitud utiliza la codificación mostrada en la siguiente tabla.

No. of data bytes	Data length code (DLC)			
	DLC3	DLC2	DLC1	DLC0
0 *to* 8	As in Classical CAN			
12	r	d	d	r
16	r	d	r	d
20	r	d	r	r
24	r	r	d	d
32	r	r	d	r
48	r	r	r	d
64	r	r	r	r

Tabla 3.- Método de codificación DLC CAN-FD. Fuente: [1].

Como hemos visto en el apartado anterior, las tramas de datos del CAN FD las podemos encontrar como formato base (identificador 11-bits) y como formato extendido (identificador 29-bits).

4.3. Bluetooth

Bluetooth es un protocolo de comunicación inalámbrica para la transmisión de datos y voz entre diferentes dispositivos a corta distancia, creado por Bluetooth Special Interest Group en 1989. La comunicación se realiza mediante un enlace por radiofrecuencia en la banda ISM de los 2,4 GHz (de 2,4 a 2,49 GHz) [14].

Los dispositivos se clasifican en distintas clases en referencia a su potencia de transmisión. En la Tabla 4 se pueden apreciar las distintas clases, la potencia máxima permitida para cada una de ellas y el alcance aproximado.

Clase	Potencia máxima permitida		Alcance (aproximado)
Clase 1	100 mW	20 dBm	100 m
Clase 2	2,5 mW	4 dBm	5-10 m
Clase 3	1 mW	0 dBm	1 m
Clase 4	0,5 mW	-3 dBm	0,5 m

Tabla 4.- Dispositivos Bluetooth según potencia máxima. Fuente: [15].

Como se muestra en la Tabla 5, los dispositivos con Bluetooth también pueden clasificarse según su versión y ancho de banda.

Versión	Año de publicación	Ancho de banda
Versión 1.2	2003	1 Mbit/s
Versión 2.0+EDR	2004	3 Mbit/s
Versión 3.0 + HS	2009	24 Mbit/s
Versión 4.0	2010	32 Mbit/s
Versión 5.0	2017	50 Mbit/s

Tabla 5.- Dispositivos Bluetooth según versión. Fuente: [15].

Todas las versiones de los estándares de Bluetooth están concebidas para la retrocompatibilidad, es decir, permite que el último estándar englobe a todas las versiones anteriores. Para este proyecto, nos centraremos la versión 4.0 del protocolo, también conocida como Bluetooth Low Energy (BLE) [14].

4.4. Bluetooth Low Energy (BLE)

En 2009, el grupo *Bluetooth Special Group* presenta la versión 4.0 del protocolo Bluetooth. Bluetooth 4.0 incluye tanto el Bluetooth clásico como el Bluetooth Low Energy (BLE) [14].

La tecnología incorporada en esta versión optimiza el consumo de energía en los distintos modos de funcionamiento (pleno rendimiento, medio y reposo), sobre todo, para un uso a bajas velocidades de datos.

A diferencia de la comunicación Bluetooth clásica, basada en una conexión serie asíncrona, una comunicación por BLE actúa como un tablón de anuncios. Los dispositivos que se conectan a ella son como los miembros de la comunidad que leen el tablón de anuncios. Los dispositivos conectados tienen un rol asignado, cada dispositivo actúa como tablón de anuncios (dispositivo periférico) o como lector de información (dispositivo central) [16].

En la Figura 8 se representa este concepto y los roles de los distintos dispositivos conectados.

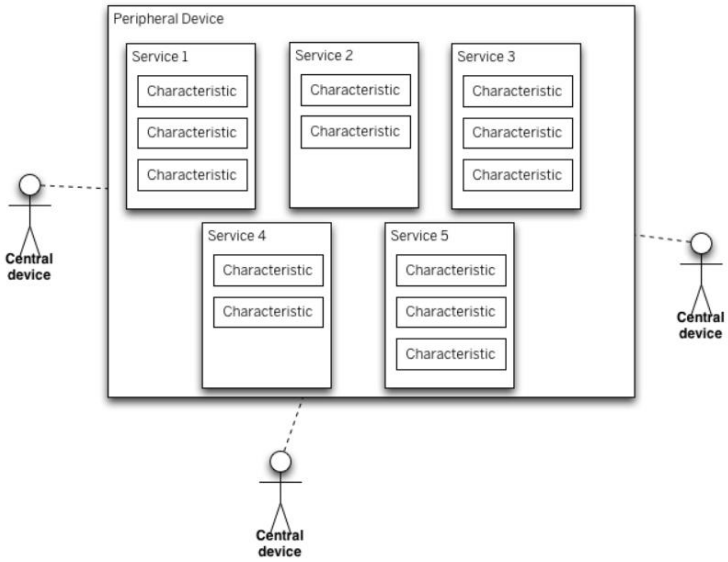

Figura 8.- Representación roles de los dispositivos conectados por BLE. Fuente: [16].

La información que proporciona un periférico está estructurada en servicios, cada uno de los cuales se subdivide en características. Tanto los servicios como las características tienen un identificador único (UUID de hasta 128 bits). Aunque el identificador puede formarse con una combinación aleatoria de caracteres, en las especificaciones del protocolo existe un listado de los identificadores recomendados dependiendo del tipo de aplicación. Estas recomendaciones se pueden consultar en la página web de Bluetooth [14].

Si el dispositivo es periférico, solo debe preocuparse de actualizar la información de las características cuando sea necesario sin preocuparse de si los dispositivos centrales las han leído o no. Por otro lado, un dispositivo central necesita conectarse al periférico y leer las características que le interese.

Para poner un ejemplo, un dispositivo periférico podría ser una estación meteorológica con tres servicios: lluvia, temperatura y humedad. En el caso del servicio de temperatura se podría tener tres características: temperatura actual, temperatura máxima del día y temperatura mínima. Los distintos usuarios pueden conectarse al periférico y consultar la característica o características que necesite, sin necesidad de coger el paquete completo de información (solo la necesaria).

Las características pueden ser solo leíbles o leíbles y escribibles. En el primer caso, solo el periférico puede actualizar el valor, mientras que, en el segundo, tanto el periférico como el central pueden.

4.5. SPI

El bus SPI es un estándar de comunicaciones síncrono, utilizado principalmente para la transferencia de información entre circuitos integrados [17]. En nuestro proyecto se va a utilizar para la comunicación entre la placa de Arduino y la placa controladora del CANFD.

Los dispositivos se comunican utilizando una relación maestro-esclavo, donde el dispositivo principal (máster) es el que inicia la transmisión. Este protocolo permite tener uno o varios dispositivos esclavos, pero un único dispositivo máster.

La sincronización y la transmisión de datos se realiza por medio de 4 señales:

- SCLK (*Serial Clock*): Es el pulso que marca la sincronización entre el Máster y el/los esclavo/s. Esta señal está siempre gobernada por el dispositivo máster.
- MOSI (*Master Output Slave Input*): Salida de datos del dispositivo máster y entrada de datos al esclavo.
- MISO (*Master Input Slave Output*): Salida de datos del dispositivo esclavo y entrada de datos al máster.
- CS (*Chip Select*): Se encarga seleccionar y habilitar un esclavo.

En la Figura 9 se representa el conexionado necesario para el funcionamiento del protocolo SPI en una configuración con tres esclavos.

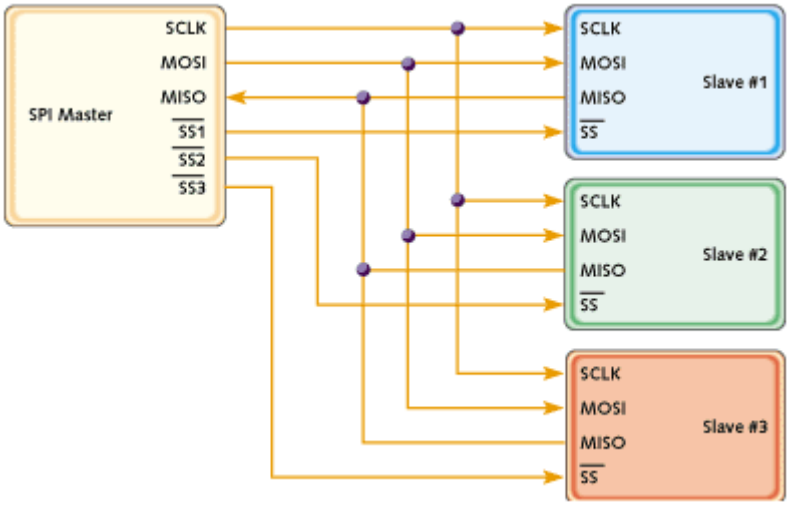

Figura 9.- Conexionado SPI máster con múltiples esclavos. Fuente: [18].

En el caso particular de este proyecto solo tenemos un dispositivo máster y uno esclavo. Por lo tanto, el conexionado sería el mostrado en la Figura 10.

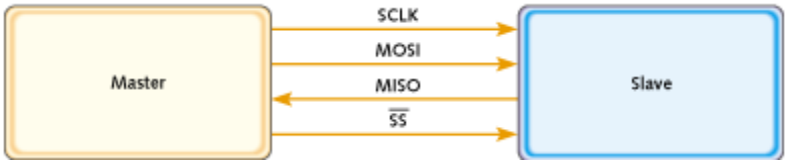

Figura 10.- Conexionado SPI escenario un máster con un esclavo. Fuente: [18].

5. Solución propuesta

Como se ha mencionado anteriormente, en el mercado ya existen dispositivos que satisfacen la mayoría de las necesidades que tenemos, pero normalmente la parte del coste es donde fallan. Es decir, son dispositivos excesivamente caros. Por lo tanto, en este proyecto se busca desarrollar una versión de bajo coste (*low cost*).

5.1. Hardware

Se utiliza una placa comercial MCP2517FD Click como solución completa para la parte de CAN FD. Esta placa comercial es un controlador de CAN FD económico y de tamaño reducido, que contiene el controlador de CAN FD, el transceptor (*transceiver*) de Microchip y el conector DB9 [19]. Esta placa requiere alimentación tanto de 3,3 V como 5 V y se comunica con el microcontrolador mediante SPI.

Se propone utilizar una de las últimas placas comerciales lanzadas por Arduino, *Nano33 IoT,* que permite recibir la mensajería CAN FD por SPI y enviarla mediante bluetooth (BLE) gracias al módulo WIFININA que tiene incorporado [20].

Para la recepción de los datos enviados mediante BLE, inicialmente se propone emplear un teléfono móvil. Debido a la evolución del proyecto, finalmente se opta por utilizar una RaspberryPi 3B+ [21].

5.2. Software

Se puede dividir en dos partes: C++ y Python. A grandes rasgos, el primer lenguaje de programación se utiliza para programar en el entorno de Arduino y el segundo para crear una aplicación en el dispositivo cliente, que permita mostrar los mensajes de CAN e interactuar con el dispositivo.

En Arduino se utilizan las librerías ACAN2517FD y SPI que permiten que el dispositivo reciba los mensajes CAN FD adecuadamente, además de utilizar la librería ArduinoBLE para la conexión de la placa de Arduino con el terminal del usuario.

En la Raspberry PI se emplea la librería Bluepy para la conexión de la placa con el dispositivo receptor.

5.3. Arquitectura final

Como se indica en la Figura 11, las tres placas se acoplan creando el dispositivo final, las dos primeras (MCP2517FD y Arduino) se acoplan físicamente mediante cableado, mientras que la tercera (Raspberry Pi) se conecta mediante Bluetooth.

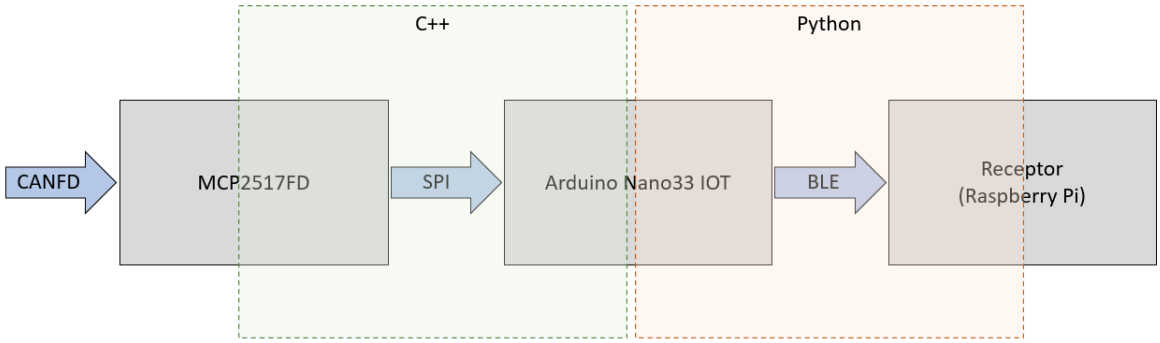

Figura 11.- Representación componentes de la solución propuesta. Fuente: propia.

El sistema completo tiene dos posibles configuraciones de funcionamiento: mensajes de CAN FD emulados y mensajes CAN FD reales. En los siguientes capítulos se profundiza más en las dos posibles configuraciones.

6. Desarrollo del prototipo

En este apartado se detallan los diferentes procedimientos que se han llevado a cabo durante el desarrollo de la solución, desde la conexión física de las placas, conexión SPI, código de Arduino para la generación de mensajes CANFD, conexión BLE… También se detallan los problemas encontrados y cómo se han solventado.

Para facilitar la construcción del prototipo, se decidió fraccionar el trabajo en tres fases principales:

- Fase 1 MCP2517FD – Arduino: Configuración SPI, envío y recepción de mensajes CANFD.
- Fase 2 Conexión Arduino – Dispositivo Receptor: Configuración y envío de datos por Bluetooth Low Energy (BLE).
- Fase 3: Desarrollo de la puerta de enlace (Gateway) CANFD-BLE completa.

En cada una de las fases, se realizan un test de validación para comprobar el funcionamiento correcto de cada uno de los componentes. Fraccionar el desarrollo permite centrarse en tareas más simples. Los problemas o funcionamientos incorrectos se pueden aislar y resolver de forma ágil.

6.1. Fase 1: MCP2517FD – Arduino Nano33 IoT

6.1.1. Instalación Arduino IDE y configuración placa Nano33 IoT

Para programar en el entorno de Arduino, compilar los programas y pasarlos a la placa Nano33 IoT, se ha utilizado el IDE de Arduino.

En este apartado de la memoria, se detallan los pasos seguidos durante la instalación y configuración del IDE de Arduino.

Desde el https://www.arduino.cc/en/software se puede descargar el ejecutable e instalar Arduino IDE. El proceso de instalación es sencillo y rápido. Ante cualquier duda, Arduino proporciona un manual de instalación y una guía para principiantes [22].

Una vez instalado IDE, se debe seleccionar la placa de trabajo. Por defecto, Nano33 IoT no viene instalada y, por lo tanto, es necesario buscar la tarjeta electrónica y seleccionar la placa. Nano33 IoT se encuentra dentro del conjunto SAMD Boards (32-bits ARM Cortex).

Es necesario abrir el Gestor de tarjetas e instalar Arduino SAMD Boards (32-bits ARM Cortex). En la Figura 12 se muestra el proceso descrito de forma visual.

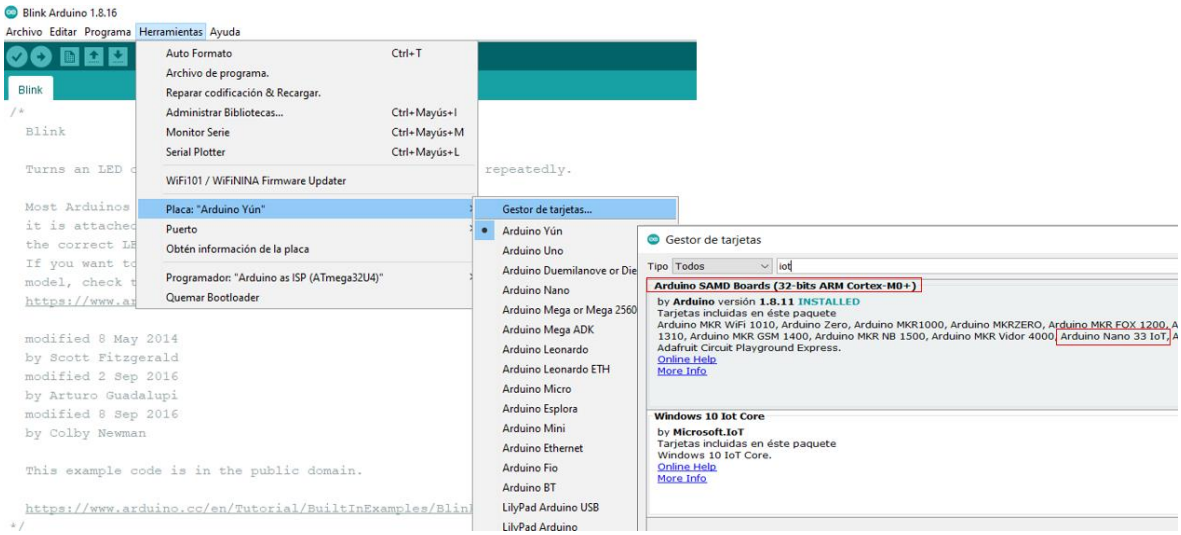

Figura 12.- Instalación tarjeta gráfica Arduino SAMD Boards. Fuente: propia.

Una vez instalada la tarjeta correcta, dentro del listado de placas, ya aparece Nano33 IoT y se puede seleccionar. Se puede ver que se ha configurado correctamente, porque aparecerá en la parte inferior derecha de la pantalla de IDE o en Herramientas --> Placa: Arduino NANO 33 IoT (véase la Figura 13).

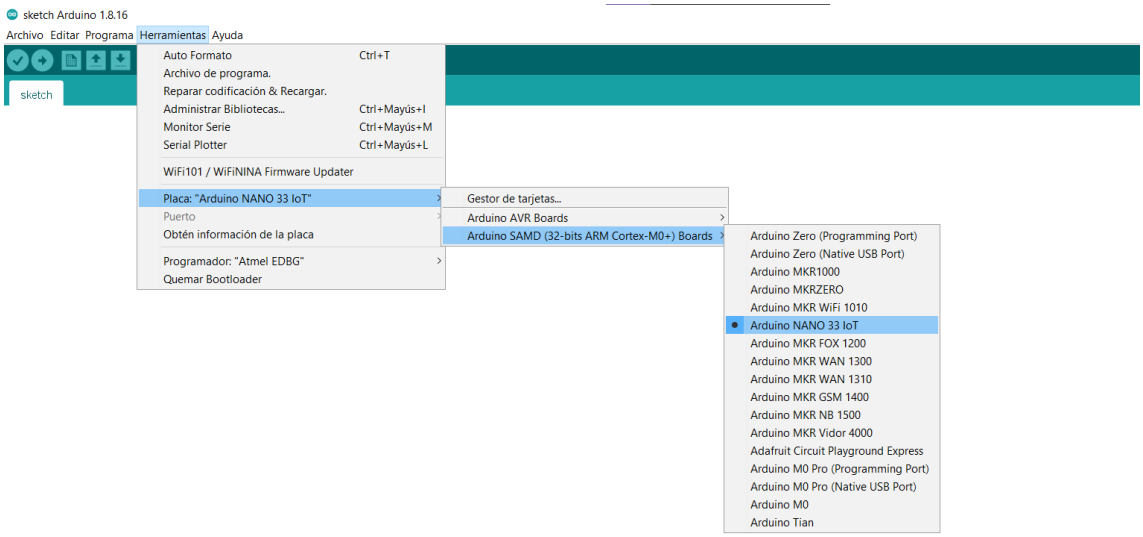

Figura 13.- Selección placa de trabajo en Arduino IDE. Fuente: propia.

6.1.2. Instalación librerías: SPI y ACAN2517FD

En este apartado se explican los pasos seguidos para instalar las librerías de CANFD y SPI.

A modo resumen, hay dos formas de instalar una librería en Arduino:

- Librería propia de Arduino: Buscar la librería mediante el administrador de bibliotecas del IDE de Arduino.
- Librería no propia de Arduino: Descargar el .zip de la librería y añadirlo con la opción *Añadir biblioteca .ZIP…*

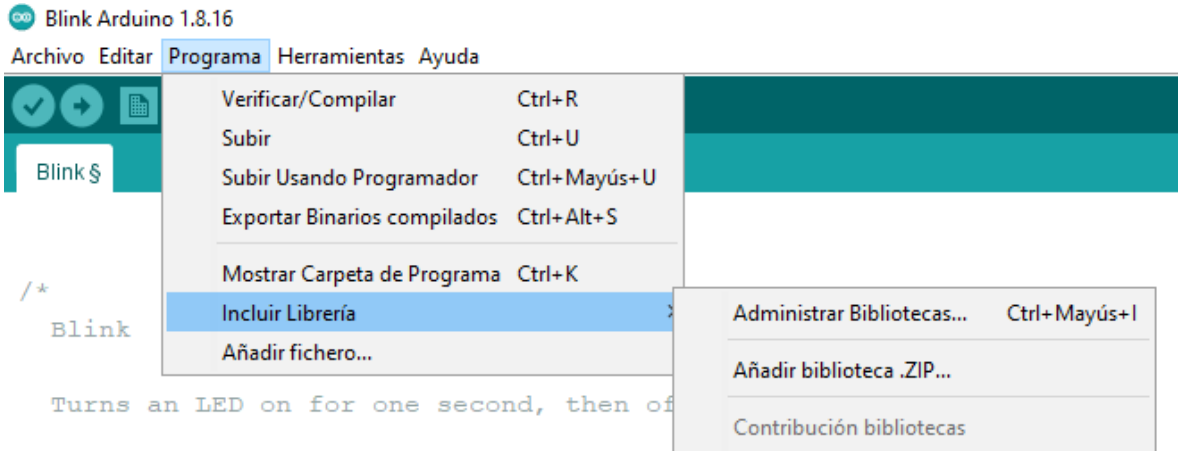

Figura 14.- Incluir librerías en IDE de Arduino. Fuente: propia.

En el caso de la librería SPI, no hace falta instalarla, porque ya viene por defecto al instalar el IDE de Arduino. Si no fuera el caso, se podría instalar desde el administrador de bibliotecas buscando *SPI*.

Para la parte de control de mensajería CAN y CAN FD, se utiliza la librería externa ACAN2517FD y está disponible para descargar en el siguiente enlace: https://github.com/pierremolinaro/acan2517FD. Una vez descargado el fichero .zip, el proceso que hay que seguir es el siguiente:

- En el IDE de Arduino: *Programa* > *Incluir Librería* > *Añadir biblioteca .ZIP* …
- Se abrirá un cuadro de diálogo y hay que indicar el directorio donde hemos guardado el zip descargado.
- Una vez introducido el directorio y seleccionada la carpeta zip, se instalará la librería.
- Al finalizar la instalación, la nueva biblioteca aparece en el listado de bibliotecas disponibles para utilizar.

Como se ha mencionado anteriormente, Arduino proporciona un manual básico de usuario, donde se encuentra detallado este procedimiento y se acompaña con capturas de pantalla para facilitar el seguimiento, además de incluir alternativas y solución a los problemas de instalación más recurrentes (configuración de puertos, librerías, placas …) [22].

Cuando se incluyen librerías en Arduino, éstas se instalan en el siguiente directorio: *C:\Users\.....\Documents\Arduino*. Importante tenerlo presente si en algún momento es necesario consultar o modificar algún parámetro de la librería.

6.1.3. Primeras pruebas con Arduino Nano 33 IoT

Al recibir la placa de Arduino y configurar el IDE de Arduino se realizan una serie de pruebas para comprobar que la placa que tenemos no tiene desperfectos y no existen incompatibilidades con las librerías que se van a utilizar (ACAN2517FD, SPI y BLE).

Al recibir la placa de Arduino se sueldan los pines y se comprueba que no hay ningún cortocircuito entre ellos, mirando continuidad entre los pines correlativos.

Se conecta el Arduino Nano al ordenador mediante el USB. Se observa como el IDE detecta automáticamente qué puerto se está utilizando. No es el caso, pero suele ser un fallo bastante habitual que no se reconozca. En el manual de usuario de Arduino se detalla como configurar el puerto si no se detecta automáticamente [22].

Una vez detectada la placa correctamente, se cargan dos ejemplos para detectar si las librerías que necesitamos en este punto funcionan correctamente.

- Ejemplo: *Basics Blink* → El programa consiste en hacer parpadear el led inferior del Arduino Nano de forma intermitente. Como se puede observar en la captura de pantalla de la Figura 15, el programa se puede cargar con éxito en la placa y funciona correctamente. Se realiza la modificación de los tiempos de encendido y apagado del LED con resultado satisfactorio.

- Ejemplo: SPI *BarometricPressureSensor* → Como en el caso anterior, el programa se puede cargar y no aparece ningún mensaje de error. En una primera prueba, no existen incompatibilidades con las librerías instaladas.

Figura 15.- Ejemplo Blink cargado correctamente en Arduino Nano33 IoT. Fuente: propia.

6.1.4. Conexionado entre MCP2517FD y Arduino Nano33 IoT

Como se ha mencionado anteriormente, para la transferencia de información entre estas dos placas se utiliza un bus de comunicación SPI con un dispositivo maestro (Arduino) y un único dispositivo esclavo (MCP2517FD).

Con la ayuda de los *datasheet* de las placas (Arduino Nano33 IoT [20] y MCP2517FD click board [19]), se han buscado los pines destinados para el SPI y se ha determinado el conexionado. El único pin que queda a nuestra elección es el pin del Arduino que hace la función de *Chip Select*. Se selecciona el pin físico 20 porque es un pin digital configurable como salida.

La alimentación de la placa de Arduino se realiza mediante el cable micro-usb. En el caso de la MCP2517FD necesita dos alimentaciones: una a 3,3 V y otra a 5 V. La primera alimentación la proporciona la placa de Arduino, mientras que la segunda se proporciona mediante un cable de alimentación USB.

En las Figuras 16 y 17 se muestra el esquema del conexionado y el resultado final.

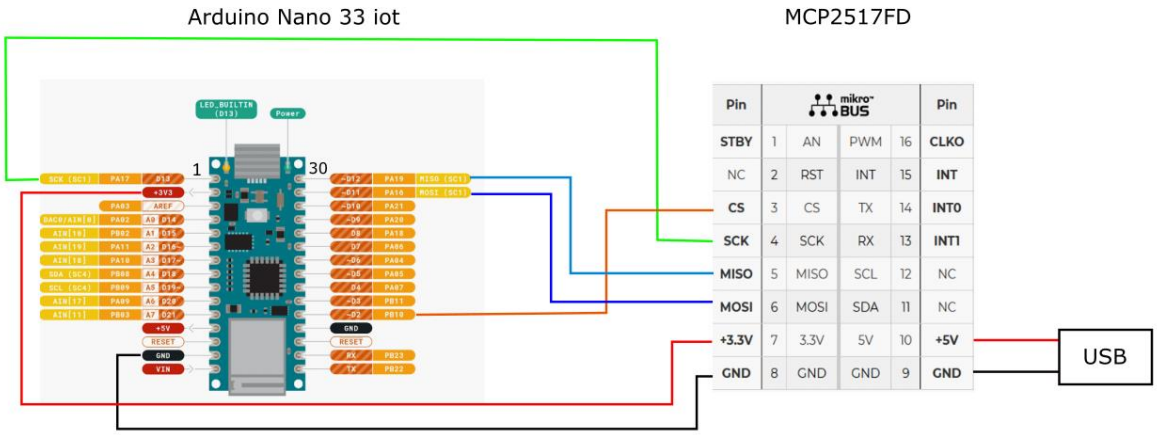

Figura 16.- Esquema conexionado SPI. Fuente: propia.

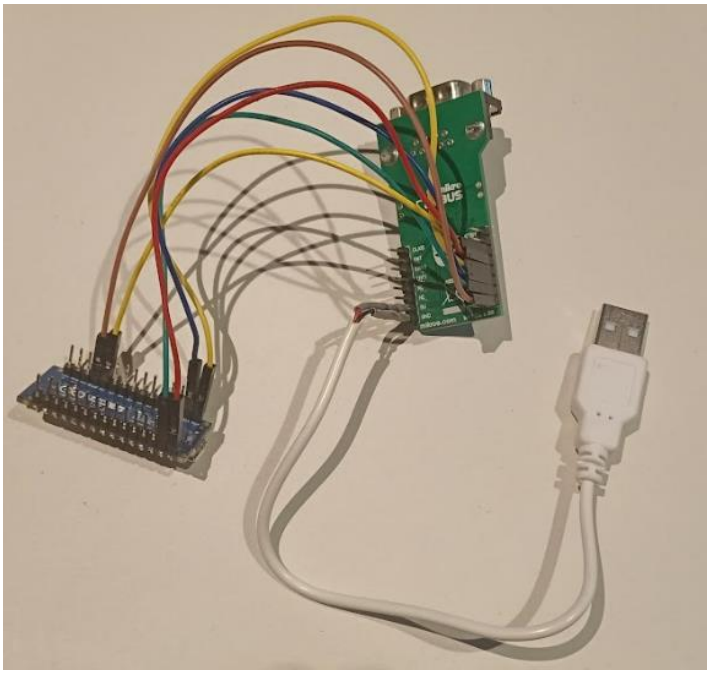

Figura 17.- Conexionado final MCP2517FD-Arduino Nano33 IoT. Fuente: propia.

6.1.5. Lectura y envío de mensajes CAN FD (emulados)

En este apartado se explica el procedimiento seguido para la creación del programa que permite la lectura y el envío de mensajes CAN FD emulados, es decir, sin la necesidad de tener una fuente/centralita que genere estos mensajes.

Para desarrollar el programa, se utiliza un programa ejemplo disponible en la librería ACAN2517FD. Esta librería no dispone de ningún ejemplo preparado para Arduino Nano33 IoT. Por lo tanto, se escoge como base un programa para la placa Arduino Uno y se adapta a las necesidades del proyecto.

Ejemplo base seleccionado: *LoopBackDemoArduinoUnoNoInt*. Este programa envía y recibe un mensaje de CAN sin necesidad de tener la placa conectada externamente. Es decir, genera un bucle interno y los mensajes enviados son los que se reciben. La placa se configura para que funcione en modo *InternalLoopBack mode*.

Después de adaptar los pines del programa a los pines del Arduino Nano33 IoT, se intenta hacer funcionar el programa.

Primer error encontrado: Al cargar el programa aparece el siguiente mensaje de error 0x1. Se consulta la documentación de la librería y se observa que corresponde a un problema de configuración. El error aparece cuando se excede el tiempo previsto de configuración.

Solución: Después de analizar con el osciloscopio las señales físicas de los pines, se observa que los pulsos digitales presentes en el pin asignado como CS (*chip select*) no son correctos. El problema se encuentra en la definición del pin. Se indicó como *chip select* el pin número 20, pero Arduino reconoce ese pin como número 2 (es el pin digital D2, aunque físicamente sea el pin 20 de la placa).

- *static const byte MCP2517_CS = 20* → Definición incorrecta
- *static const byte MCP2517_CS = 2* → Definición correcta

Las placas comerciales de Arduino se utilizan para infinidad de proyectos. En internet podemos encontrar gran cantidad de documentación acerca de ellas, en ocasiones, documentación ambigua y contradictoria, lo que provoca un error a la hora de identificar los pines.

Se recomienda siempre comprobar si el pin declarado corresponde al deseado. La prueba es rápida y evita horas de trabajo buscando errores. Consiste en añadir el siguiente bucle y observar cómo se comporta la señal que sale del pin.

```
8  while(1)
9    {
10     digitalWrite(2, HIGH);
11     delay(50);
12     digitalWrite(2, LOW);
13     delay(50);
14   }
15
```

Figura 18.- Alternancia de estado del pin2. Fuente: propia.

De esta forma, si el pin se declara correctamente, se observa una alternancia en la salida.

Segundo error encontrado: Después de solventar el error en la definición del pin CS, el programa sigue sin funcionar y aparece un mensaje de error distinto. En este caso, error 0x40000, diferencia entre el mensaje enviado y el recibido. Se revisa el perfil del resto de pines del SPI y se observa que el pin SDO (MISO) no presenta el comportamiento habitual, típicos pulsos rectangulares. En la salida del pin aparece un pulso en el momento inicial y la señal cae justo después.

Inicialmente se sospecha que la placa es defectuosa. Para determinar si se trata de un problema de *hardware* o de *software*, se sustituye la placa por otra igual. Como el comportamiento incorrecto se mantiene, se descarta un mal funcionamiento de la placa y solo puede ser un error de configuración.

Se revisan los tres parámetros de la declaración de ACAN2517FD *settings*.

```
ACAN2517FDSettings settings (ACAN2517FDSettings::OSC_20MHz, 125UL * 1000UL, DataBitRateFactor::x1) ;
```

Se observa que cambiando el valor de la frecuencia de oscilación de 40 MHz a 20 MHz, el programa funciona correctamente. Se desconoce el motivo, pero hay que aplicar un factor del dos a la hora de configurar los parámetros.

En el programa *comparación_envio_recepcion_completo* se comprueba que el mensaje enviado corresponde al mensaje recibido. Para ello, declaramos una función que genera mensajes de CAN de forma aleatoria, *EnviarCAN()*. En la Figura 19 se observa el comportamiento de la función.

```
50 void EnviarCAN()
51 {
52    CANFDMessage frame1;
53    frame1.id = random(1, 0x7FF); //formato base: identificador de 11 bits, en el caso extendido (29bits)
54    uint8_t longitud[] = {0,1,2,3,4,5,6,7,8,12,16,20,26,32,48,64};
55    frame1.len= longitud[random(0, 16)];
56    for(int i=0; i<frame1.len; i++)
57    {
58       frame1.data[i]=random(0,255);
59    }
60    const bool ok = can.tryToSend (frame1) ;
61    Serial.print("Identificador mensaje enviado: ");
62    Serial.print(frame1.id, HEX);
63    Serial.print("  Longitud mensaje enviado: ");
64    Serial.println(frame1.len);
65    Serial.print("mensaje enviado: ");
66    for(int i=0; i<frame1.len; i++)
67       {
68       Serial.print(frame1.data[i], HEX);
69       }
70    Serial.println();
```

Figura 19.- Declaración función EnviarCAN. Fuente: propia.

Se genera un identificador de la trama con un valor aleatorio entre 1 y 0x7FF, que corresponde a los 11 bits de un identificador de una trama en formato base. Si en algún momento se requiere generar tramas en formato extendido (29 bits de identificador) se debería sustituir por: *frame.id = random(1, 0x1FFFFFFF).*

Como se ha mencionado en el capítulo 4, los mensajes de CAN y CAN FD tienen 4 bits reservados para el DLC donde se indica la longitud de los datos del mensaje. Esta longitud no puede ser aleatoria. Por lo tanto, se genera un *array* con los valores permitidos de longitud en bytes y se escoge aleatoria un ítem.

Por último, se genera aleatoriamente los datos del mensaje hasta llegar a la longitud que indica el DLC. El bucle principal del programa se encarga de ir generando mensajes y recibirlos (*InternalLoopBack mode*). Se muestran por el monitor serie ambos mensajes para poder compararlos.

Como se observa en la Figura 20, los mensajes enviados corresponden a los mensajes recibidos.

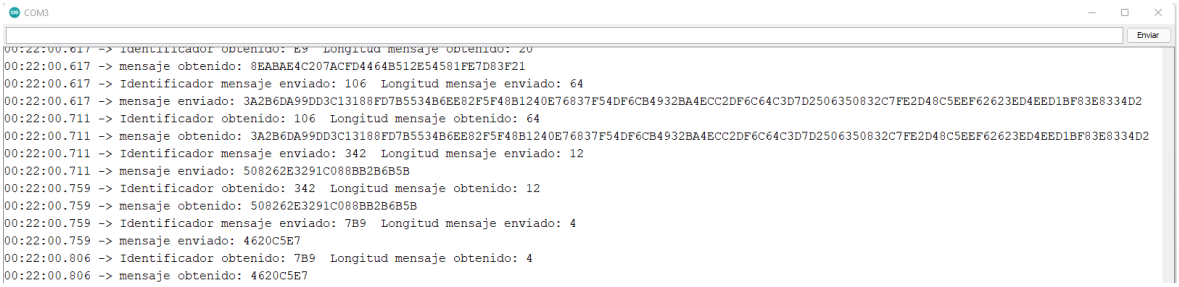

Figura 20.- Comparativa mensaje enviado vs mensaje recibido. Fuente: propia.

Para acabar de comprobar el correcto funcionamiento del envío y recepción de mensajes CAN, se observa con el osciloscopio el perfil de las señales físicas de CAN FD generadas. En la Figura 21 se observa la evolución temporal de las señales CANH y CANL.

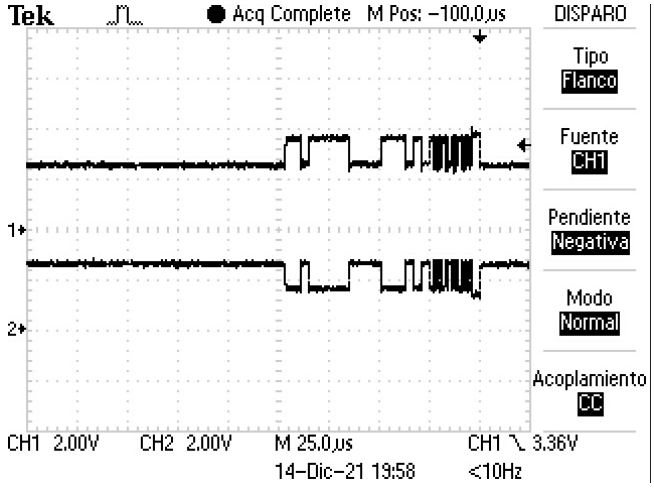

Figura 21.- Perfil CANH y CANL obtenido. Fuente: propia.

6.1.6. Lectura y envío de mensajes CAN FD (reales)

Uno de los requisitos del dispositivo es que sea capaz de leer los mensajes CAN FD reales

de una maqueta y que posteriormente los envíe por Bluetooth a un usuario conectado. Por lo tanto, ha de ser capaz de leer los mensajes generados por la maqueta del laboratorio.

La maqueta se conecta al prototipo desarrollado mediante un conector DB-9 estándar.

Preparación maqueta: Se debe reiniciar la configuración de las tres centralitas mediante los pulsadores correspondientes. En la maqueta se puede seleccionar la velocidad de transmisión del CAN FD mediante el pulsador *baudrate*. Podemos seleccionar una de las 4 combinaciones (velocidad arbitraje / velocidad de datos) según el número de veces que se pulse:

- Configuración1: 125 kbit/s + 1 Mbit/s → 1 parpadeo LED azul
- Configuración2: 250 kbit/s + 2 Mbit/s → 2 parpadeos LED azul
- Configuración3: 500 kbit/s + 4 Mbit/s → 3 parpadeos LED azul
- Configuración4: 1000 kbit/s + 8 Mbit/s → 4 parpadeos LED azul

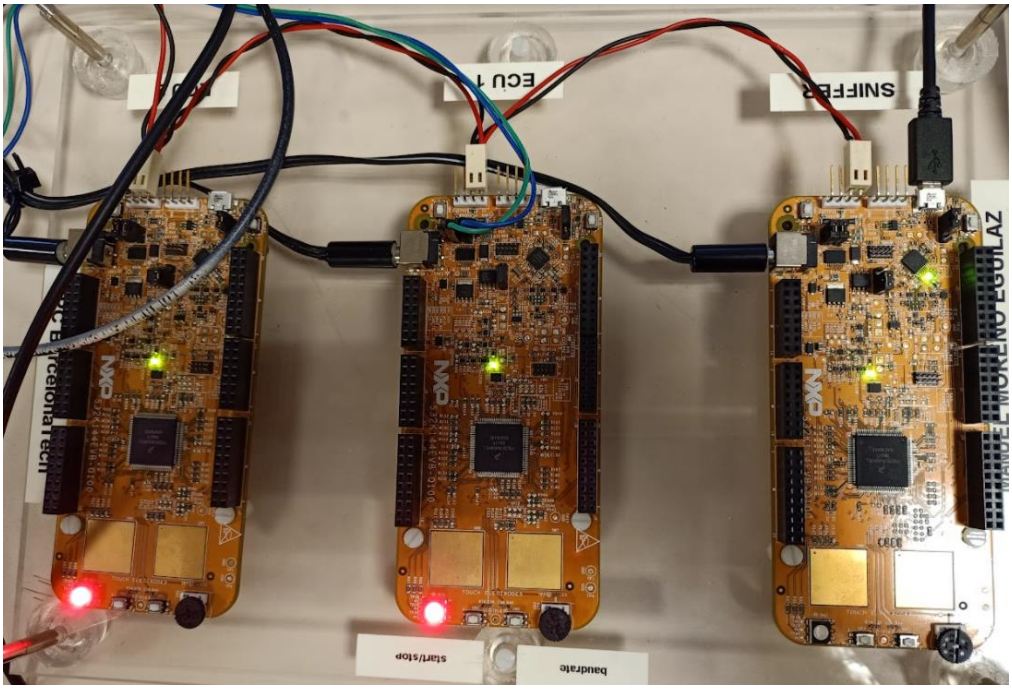

Figura 22.- Maqueta utilizada para la validación del prototipo. Fuente: propia.

Como se puede observar en la imagen de la maqueta (Figura 22), las centralitas disponen de dos pulsadores en la parte inferior con los que podemos cambiar la configuración *baurate* y activar/desactivar el envío *Start-stop*.

Preparación programa Arduino: Se emplea como base el programa que utilizamos para comparar los mensajes emulados, *comparación_envio_recepcion_completo* con las siguientes modificaciones:

- Se cambia el modo de funcionamiento InternalLoopBack → NormalFD.

```
settings.mRequestedMode = ACAN2517FDSettings::NormalFD ;
```

- Se comenta la llamada a la función *EnviarCAN()* del bucle principal del programa.
- Se selecciona la configuración *ACAN2517FD settings* para que coincida con los valores de la maqueta.

El programa de Arduino utilizado en este apartado está guardado con el siguiente nombre: *comparacion_envio_recepcion_maqueta.* Como el resto de los programas mencionados en la memoria, se pueden encontrar en los anexos.

Montaje completo:

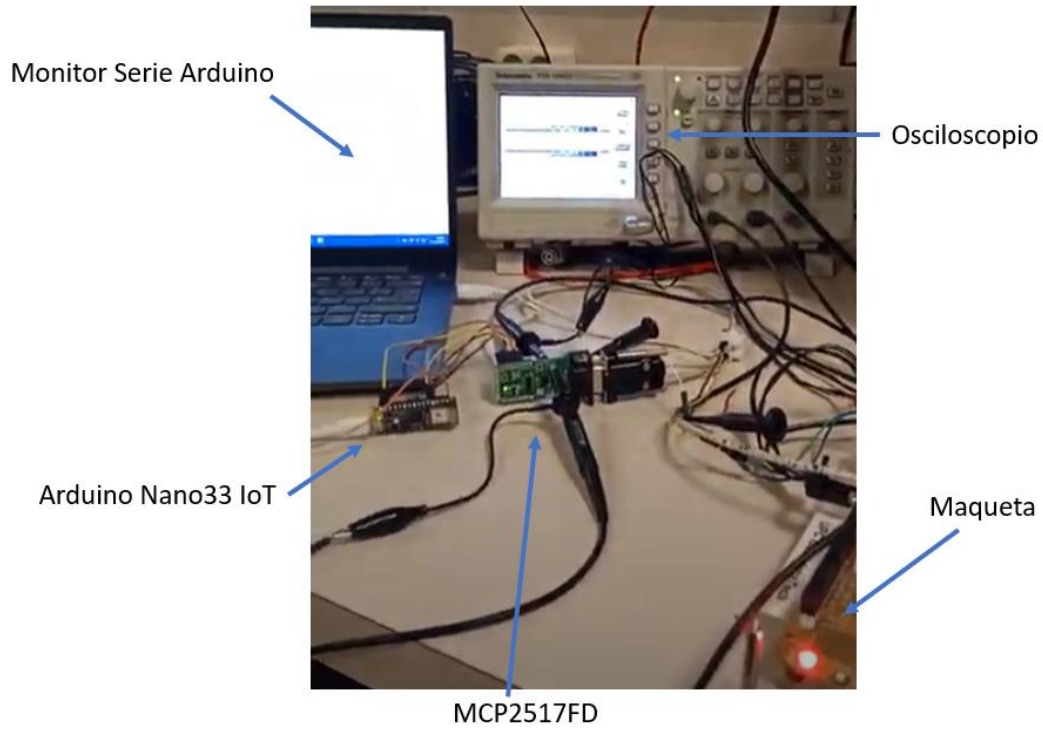

Figura 23.- Montaje completo para el envío y la lectura de mensajes CANFD. Fuente: propia.

Se realizan tres pruebas para validar el funcionamiento correcto del prototipo. Las dos primeras pruebas consisten en generar los mensajes en la maqueta y leerlos en el Arduino. La última prueba consiste en generar los mensajes en Arduino y leerlos en el ordenador. De esta forma, se puede validar el correcto funcionamiento tanto en el envío como en la recepción.

Envío de un mensaje en particular: Desde el ordenador de la maqueta se puede configurar el envío de un mensaje concreto. Se verifica que el mensaje enviado por el ordenador/maqueta (Figura 24) se corresponde al mensaje obtenido en el monitor serie de Arduino (Figura 25).

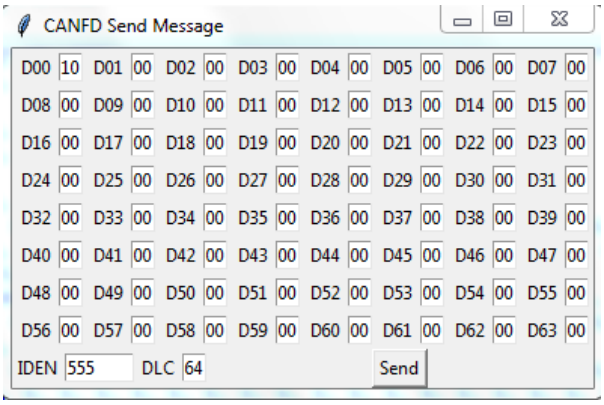

Figura 24.- Mensaje enviado por la maqueta. Fuente: propia.

```
19:17:27.087 -> Identificador obtenido: 555  Longitud mensaje obtenido: 64
19:17:27.087 -> 10000000000000000000000000000000000000000000000000000000000000000000000000000000
19:17:38.451 -> Identificador obtenido: 555  Longitud mensaje obtenido: 64
19:17:38.451 -> 10000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Figura 25.- Mensaje recibido por Arduino Nano33 IoT. Fuente: propia.

Envío de una lista de 100 mensajes de CAN FD: La segunda prueba consiste en enviar un paquete de 100 mensajes de CAN FD (mediante las ECU1 o 2) y comprobar que se reciben en la placa de Arduino. Se adjunta una captura de los mensajes enviados y recibidos para poderlos comparar.

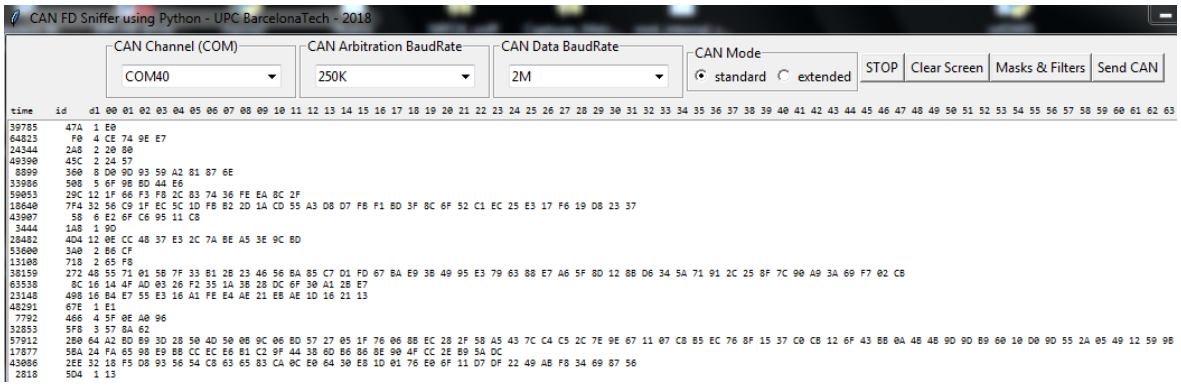

Figura 26.- Captura de los mensajes enviados por la maqueta. Fuente: propia.

```
19:11:09.739 -> B6CF
19:11:09.880 -> Identificador obtenido: 718  Longitud mensaje obtenido: 2
19:11:09.880 -> 65F8
19:11:09.974 -> Identificador obtenido: 272  Longitud mensaje obtenido: 48
19:11:09.974 -> 557115B7F33B12B234656BA85C7D1FD67BAE93B4995E3796388E7A65F8D128BD6345A71912C258F7C90A93A69F72CB
19:11:10.069 -> Identificador obtenido: 8C  Longitud mensaje obtenido: 16
19:11:10.069 -> 144FAD326F2351A3B28DC6F30A12BE7
19:11:10.163 -> Identificador obtenido: 498  Longitud mensaje obtenido: 16
19:11:10.163 -> B4E755E316A1FEE4AE21EBAE1D162113
19:11:10.257 -> Identificador obtenido: 67E  Longitud mensaje obtenido: 1
19:11:10.257 -> E1
19:11:10.352 -> Identificador obtenido: 466  Longitud mensaje obtenido: 4
19:11:10.352 -> 5FEA096
19:11:10.446 -> Identificador obtenido: 5F8  Longitud mensaje obtenido: 3
19:11:10.446 -> 578A62
19:11:10.540 -> Identificador obtenido: 2B0  Longitud mensaje obtenido: 64
19:11:10.540 -> A2BDB93D28504D50B9C6BD572751F7668BEC282F58A5437CC4C52C7E9E67117C8B5EC768F1537C0CB126F43BBA4B4B9D9DB96010D09D552A54912599B
19:11:10.682 -> Identificador obtenido: 5BA  Longitud mensaje obtenido: 24
19:11:10.682 -> FA6598E9BBCCECE6B1C29F44386DB6868E904FCC2EB95ADC
19:11:10.776 -> Identificador obtenido: 2EE  Longitud mensaje obtenido: 32
19:11:10.776 -> 18F5D8935654C8636583CACE06430E81D176E06F11D7DF2249ABF834698756
19:11:10.876 -> Identificador obtenido: 5D4  Longitud mensaje obtenido: 1
19:11:10.876 -> 13
```

Figura 27.- Captura de los mensajes recibidos en Arduino. Fuente: propia.

Envío de mensajes CAN FD desde Arduino: En esta última prueba, volvemos a activar la función *EnviarCAN()* del programa y se envía un listado de mensajes para leerlos en el ordenador. En este caso, se observa como los mensajes leídos por parte del ordenador coincide con los mensajes generados y enviados por parte de Arduino. En el caso del monitor serie solo imprimimos los valores del identificador y longitud del mensaje para agilizar la comparativa.

```
18:54:09.013 -> Identificador mensaje enviado: 6EC  Longitud mensaje enviado: 3
18:54:09.531 -> Identificador mensaje enviado: 65B  Longitud mensaje enviado: 5
18:54:10.002 -> Identificador mensaje enviado: 624  Longitud mensaje enviado: 12
18:54:10.521 -> Identificador mensaje enviado: 1FB  Longitud mensaje enviado: 16
18:54:11.040 -> Identificador mensaje enviado: C4  Longitud mensaje enviado: 48
18:54:11.511 -> Identificador mensaje enviado: 444  Longitud mensaje enviado: 2
18:54:12.030 -> Identificador mensaje enviado: 71D  Longitud mensaje enviado: 1
18:54:12.502 -> Identificador mensaje enviado: 41  Longitud mensaje enviado: 32
18:54:13.020 -> Identificador mensaje enviado: 367  Longitud mensaje enviado: 7
18:54:13.539 -> Identificador mensaje enviado: 680  Longitud mensaje enviado: 5
```

Figura 28.- Captura de mensajes enviados por Arduino. Fuente: propia.

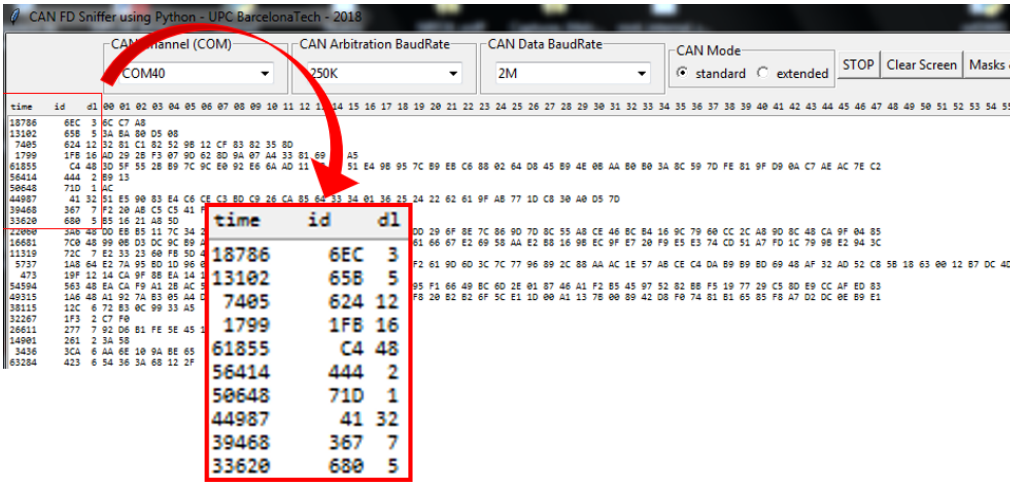

Figura 29.- Captura mensajes recibidos por la maqueta. Fuente: propia.

Se observa que los mensajes enviados desde las centralitas y ordenador corresponde a los

mensajes obtenidos, así que se da por finalizada la parte de envío y recepción de mensajería CAN FD (fase 1).

6.2. Fase 2: Arduino Nano33 IoT – Dispositivo receptor

En esta parte del proyecto es necesario trabajar con la librería ArduinoBLE. Por defecto, ya viene instalada al descargar el IDE. Si no fuera el caso, se debería instalar (siguiendo los pasos descritos en el apartado 6.1.2 o en el manual básico de usuario [22].

Inicialmente se decide utilizar un móvil con una versión de Bluetooth superior a 4.0 para que incluya BLE. Se pretende crear una pequeña aplicación, mediante *AppInventor,* para mostrar los datos recibidos. Al ser un dispositivo móvil con sistema operativo Android se pensó que la aplicación sería sencilla de crear con esta herramienta creada por el MIT (*Massachusetts Institute of Technology*) [23].

Como se detallará en los siguientes subapartados, finalmente se descarta esta opción y se opta por desarrollar un programa en Python y utilizar, como dispositivo receptor, una Raspberrypi 3B que incluye *Bluetooth Low Energy (BLE)*.

En este apartado se describen los pasos seguidos para desarrollar la parte del prototipo que solventa el envío y recepción de datos por BLE.

6.2.1. Primeros pasos librería ArduinoBLE y AppInventor

El primer ejercicio realizado es cargar un programa ejemplo de la librería ArduinoBLE (LED) encargado de buscar los dispositivos de Bluetooth y conectarse a uno de ellos. Una vez realizada la conexión permite controlar el encendido y apagado del LED de la placa.

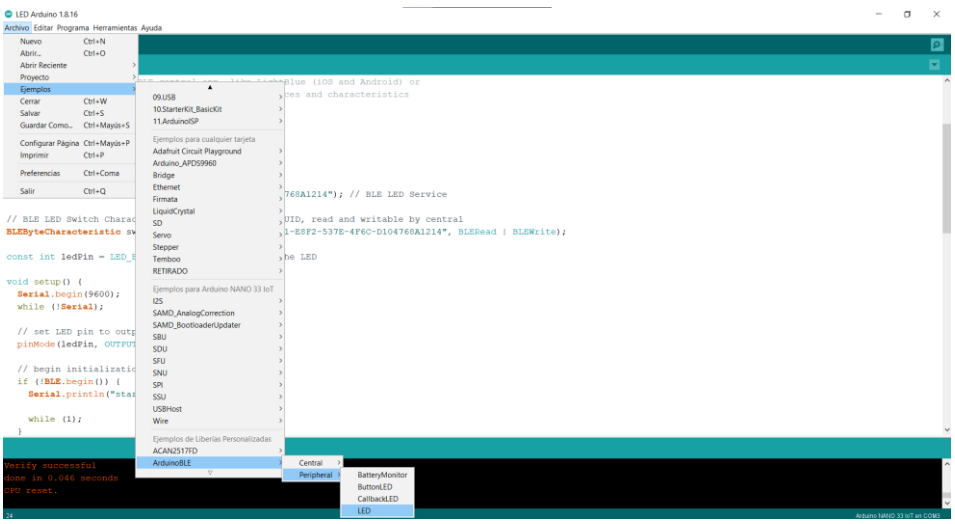

Figura 30.- Ejemplo librería ArduinoBLE utilizado. Fuente: propia.

Es un programa básico que permite ver cómo inicializar e incluir un servicio BLE y cómo se define una característica de un servicio (*BLEStringCharacteristic, BLEByteCharacteristic, BLEBoolCharacteristic* …) y las propiedades que se le pueden asignar (*BLEread, BLEwrite, BLEnotify*). Toda la documentación de la librería y ejemplos están disponible para consultar en internet [16].

En este punto, todavía no se desarrolla la aplicación en AppInventor. Para poder conectarse e interactuar con Arduino Nano33 IoT se utiliza la aplicación LightBlue. LightBlue permite escanear, conectarse y explorar los dispositivos BLE cercanos.

LightBlue permite conectarse a los dispositivos disponibles, ver los servicios y características que tienen y si éstas lo permiten (es decir, son características leíbles y escribibles), modificarlas.

En el caso del control del LED, funciona perfectamente. Como se puede observar en la Figura 31, se puede realizar la conexión y desconexión del dispositivo y se puede controlar el encendido del LED, modificando el valor de la característica (0 --> LED apagado “LED off”, valor diferente de 0 --> LED apagado “LED off”).

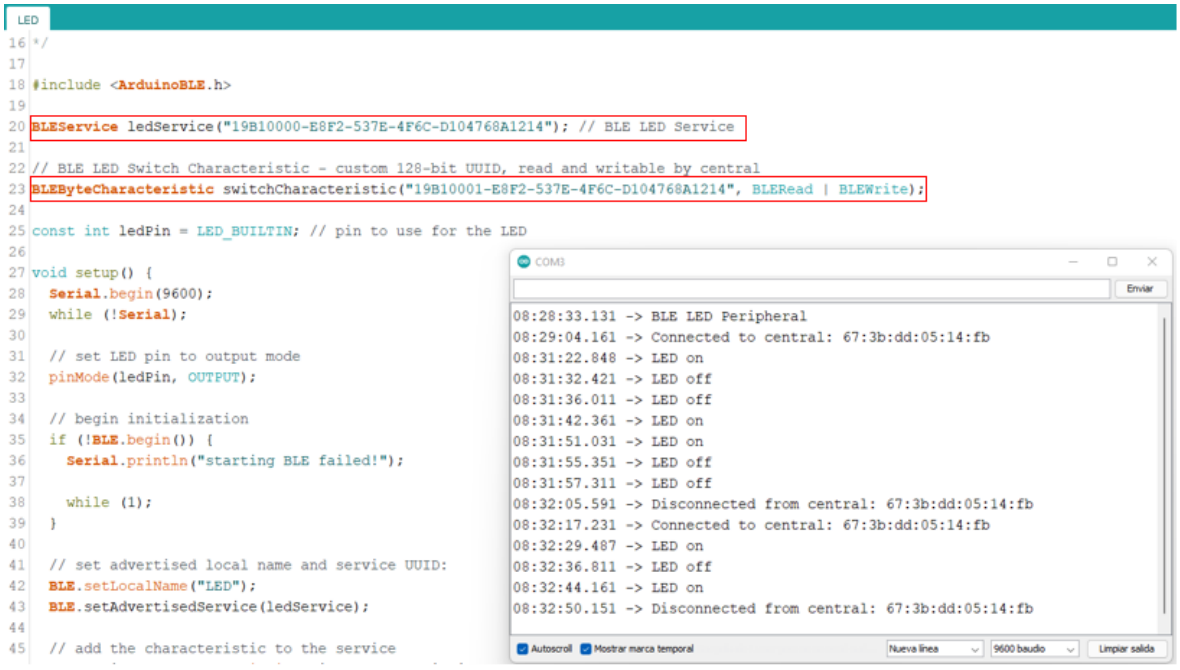

Figura 31.- Captura monitor serie control del LED mediante LightBlue. Fuente: propia.

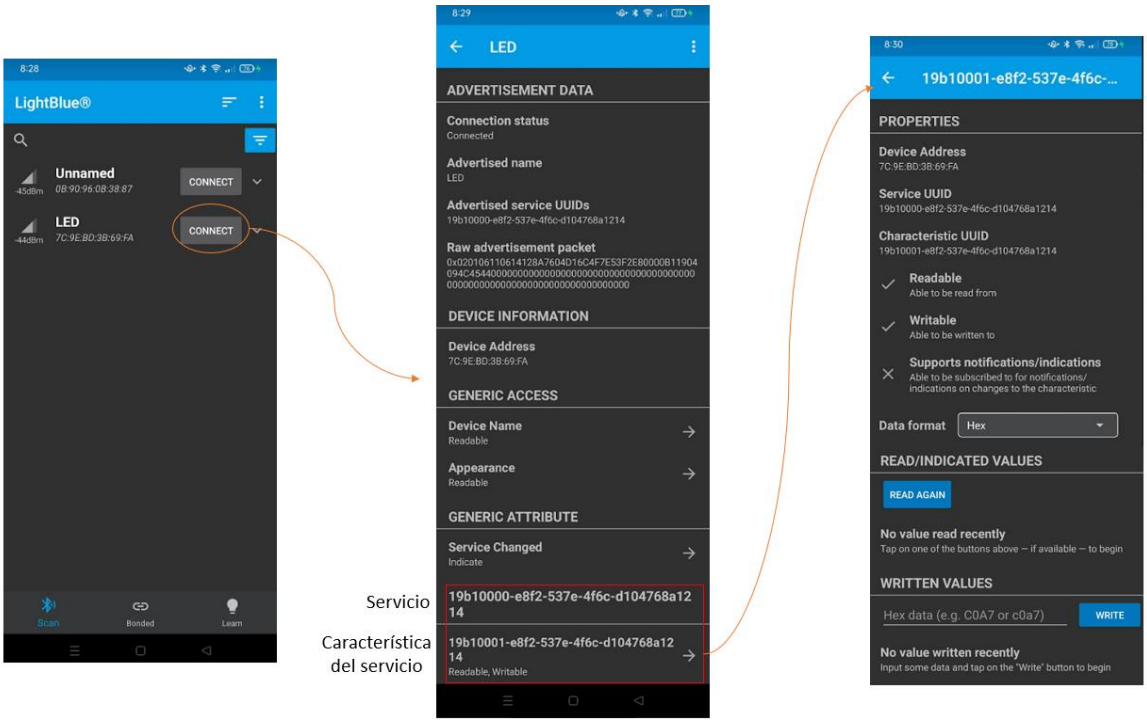

Figura 32.- Conexión e información proporcionada por LightBlue. Fuente: propia.

En este punto se desarrolla una primera aplicación en *AppInventor*, que permite conectarse y desconectarse de los dispositivos y controlar el LED.

El desarrollo de aplicaciones mediante *AppInventor* se basa en dos partes: diseño (*designer*) y bloques (*blocks*). Una primera parte, *designer*, donde se introducen los componentes que se requieren en la aplicación (botones, listas, casillas de texto…) y una segunda parte, *blocks*, donde se describe la lógica de funcionamiento de los componentes introducidos. Es decir, si se ha introducido un botón en la parte *designer*, en *blocks* se define el funcionamiento que tendrá la aplicación si el usuario selecciona el botón.

En el anexo está incluido el programa de Arduino para el control del led (*Control_LED.ino*) y la aplicación de AppInventor creada (*Control_led.aia*).

El la Figura 33 , se muestra una captura de pantalla de ambas partes de la aplicación.

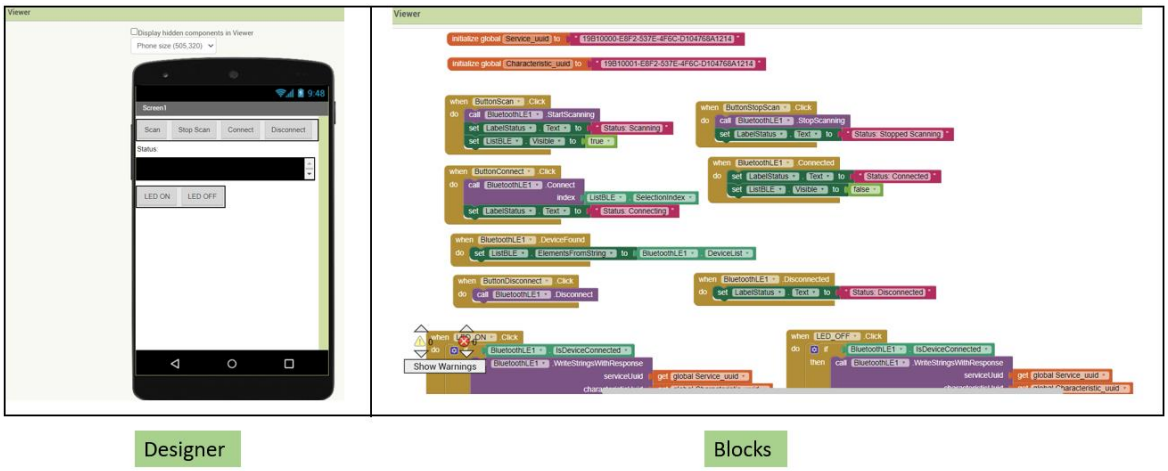

Figura 33.- Aspecto aplicación AppInventor para el control del led. Fuente: propia.

Para el control del LED, la aplicación desarrollada es simple. Con los 4 botones de la parte superior se realiza la conexión por Bluetooth. Durante la fase de búsqueda de dispositivos, se hace visible una lista con todos los dispositivos disponibles para conectar (véase Figura 34).

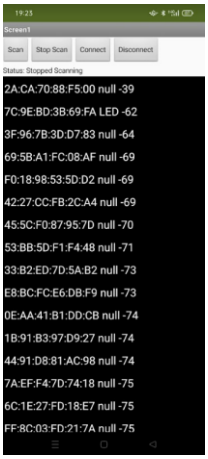

Figura 34.- Listado dispositivos BLE disponibles. Fuente: propia.

Problema encontrado: Al buscar los dispositivos (*Scan*) se observaba que no hay dispositivos disponibles (en la aplicación aparece el listado vacío), aunque por el monitor serie se indica que Arduino está intentando conectar (*advertising*) y con la aplicación LightBlue se detectan los dispositivos.

Se investiga en el foro de AppInventor si otros usuarios han tenido el mismo problema y cómo se puede solventar [24]. El problema venía provocado porque la aplicación no tenía permisos para acceder a la ubicación del dispositivo. Por lo tanto, se accedió a la configuración del dispositivo y se le activaron los permisos.

Una vez emparejado con el Arduino Nano33 IoT, este listado deja de ser visible y con los dos botones inferiores se puede encender y apagar el led. Al seleccionar los botones, la aplicación escribe sobre el valor de la característica. Asignando un 0 para apagar el LED y un 4 para encender (serviría cualquier valor distinto de 0).

Con esta primera prueba se ha realizado con éxito:

- Establecer la conexión mediante BLE entre el Arduino Nano33 IoT y el teléfono móvil.
- Añadir un servicio y una característica.
- Modificar el valor de la característica.

Después de este primer ejercicio con la comunicación BLE, hay dos opciones para el envío de la información. Por un lado, el envío de la información completa en una única característica. Por otro lado, fraccionar la información y enviarla en distintas características. En la se puede ver los dos escenarios posibles.

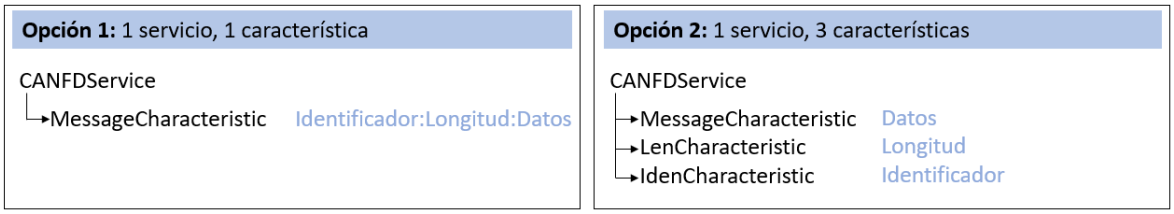

Figura 35.- Posibles escenarios para el envío del CAN FD por BLE. Fuente: propia.

6.2.2. Envío de mensajes CAN FD por BLE

De los dos escenarios posibles, se prefiere el primero, donde se envía toda la información en una única característica. En este apartado se describe la prueba realizada para poder comprobar si es viable o no. Es decir, si es posible que una única característica pueda contener el valor de un mensaje completo de CAN FD (64 bytes).

El programa de Arduino creado para conocer si es posible enviar toda la información en una misma característica está guardado en el anexo como: *limite_envio_BLE*. Es el resultado de combinar los programas de *envio_recepción_completo*, *control_led* y las siguientes modificaciones:

- Se selecciona el modo *InternalLoopBack*.
- Se define el servicio *CANFDService* y la característica *MessageCaracterístic*.
- El valor de la característica está conformado por: el identificador, la longitud de los datos y los datos. Todos ellos separados por dos puntos.
- Se fija la longitud de datos enviados a 64 bytes. De esta forma, se fuerza a tener el mensaje más largo posible.

- Se fija que los datos del mensaje sean todos igual, en este caso, la letra A, para agilizar la comprobación.
- Con el objetivo de comprobar que la información presente en la característica corresponde al mensaje CAN FD enviado, se muestra por pantalla: el mensaje generado, el que se recibe (por el bucle interno) y el valor de la característica.

En este apartado no es necesario desarrollar una aplicación en AppInventor, ya que únicamente necesitamos que el Arduino se conecte con un dispositivo para que empiece a enviar datos. En el bucle principal del programa está programado así. Arduino no llama a la función *EnviarCAN()* hasta que no hay un dispositivo conectado. En este punto, se vuelve a utilizar la aplicación LightBlue.

Problema: Como se puede ver en la Figura 36, la característica no muestra el mensaje completo.

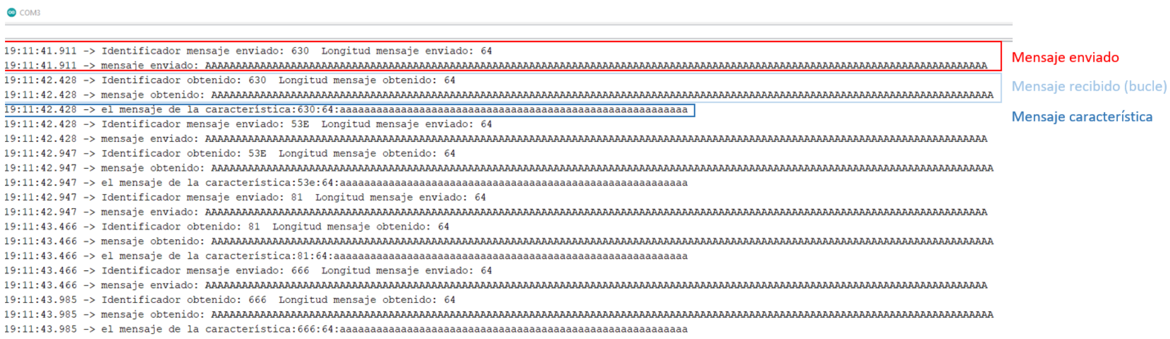

Figura 36.- Diferencia mensaje generado/recibido/característica. Fuente: propia.

Se observa como la característica muestra 64 caracteres en total, un valor muy lejano al que se necesita para representar un mensaje completo de CAN FD.

Solución: Se cambia el valor del tamaño máximo de la característica. Una vez introducido el cambio, el valor de la característica coincide con el mensaje enviado.

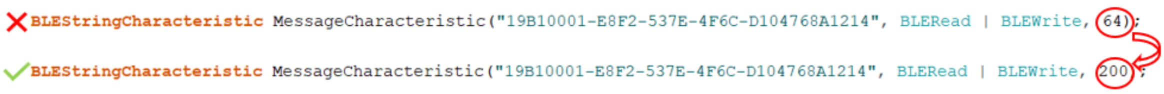

Figura 37.- Cambio en la definición para corregir el error de longitud. Fuente: propia.

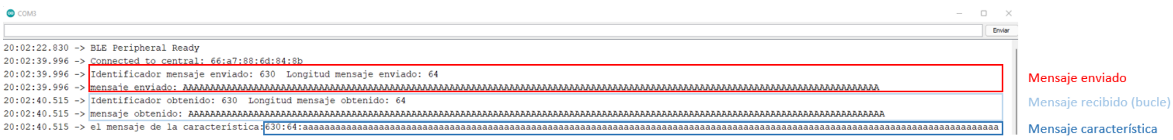

Figura 38.- Comparativa mensaje enviado/recibido/característica. Fuente: propia.

Por último, para acabar de comprobar el correcto funcionamiento del envío de información,

se modifica el programa para que el identificador, la longitud y los datos sean aleatorios. Este nuevo programa se guarda como *limite_envio_BLE_aleatorio*. Tal y como se muestra en la Figura 39, el funcionamiento es correcto.

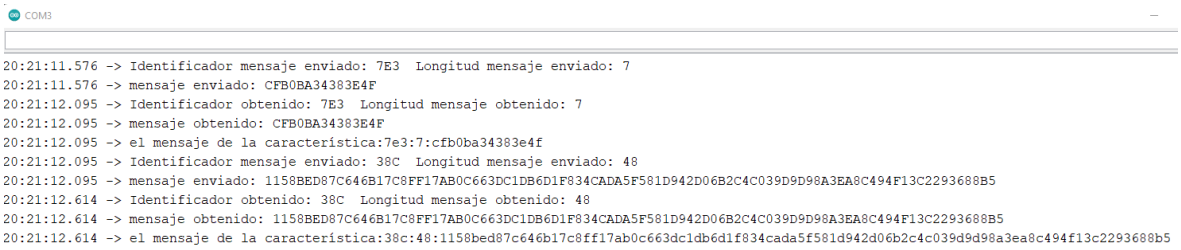

Figura 39.- Comparativa envío por BLE con mensajes aleatorios. Fuente: propia.

Con este resultado se da por finalizado el envío de mensajería CAN FD por BLE. Arduino es capaz de enviar la información correcta por BLE. El siguiente paso es desarrollar la recepción de los datos enviados.

6.2.3. Recepción mensajes CAN FD por BLE (teléfono móvil)

Una vez solucionada la parte del envío de información, falta desarrollar la parte de recepción. Es decir, tener acceso a la característica y capturar el valor por parte del dispositivo receptor.

Tal y como se indica en el alcance del proyecto, en este proyecto no se realiza un tratamiento de los datos a posterior. Por lo tanto, se considera que el dispositivo receptor ha de poder coger la información y almacenarla en un documento de texto.

Después de muchas horas invertidas en el desarrollo de la aplicación se consigue obtener una aplicación capaz de leer información (*Lectura_CANFD_archivo.aia*), pero no funcional, debido a que solo lee una parte pequeña de los mensajes enviados.

En la Figura 40 se muestra una captura de pantalla del documento de texto que se guarda en la aplicación. Al comparar los mensajes (identificador, longitud y datos) del archivo y los del monitor serie, se observa la cantidad de mensajes que no se reciben.

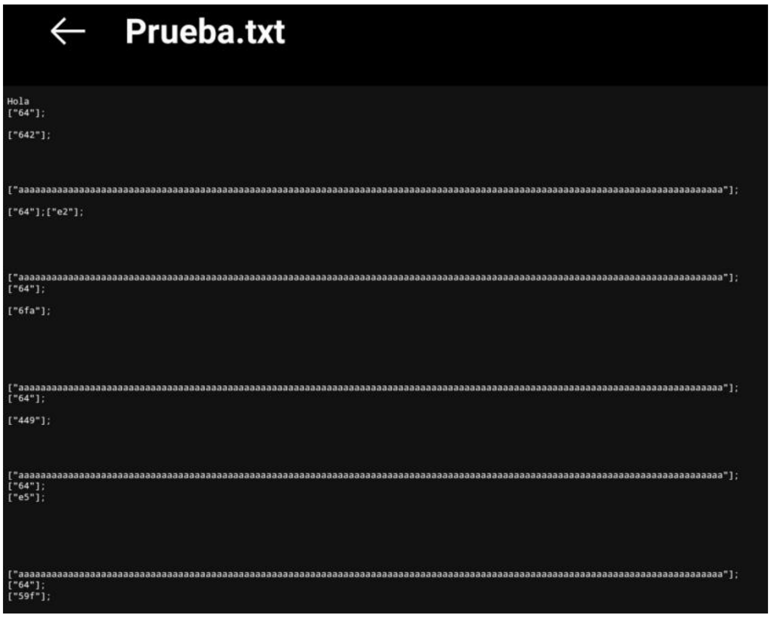

Figura 40.- Captura documento de texto generado con AppInventor. Fuente: propia.

Se ha probado a fraccionar el mensaje en distintas características y aumentar los tiempos entre mensajes. Aun aumentando el tiempo entre mensajes correlativos enviados, la cantidad de mensajes perdidos supera el 80%. Por lo tanto, no se considera cómo solución válida.

Esta pérdida de información se puede deber a los siguientes factores:

- Como la aplicación se genera a través de un conjunto de bloques estándar, el código seguramente no sea el óptimo, haciendo más lentos los procesos.

- La extensión de *AppInventor BluetoothLE* está pensada para la transmisión de datos provenientes de los sensores. Estos datos suelen ser cortos y de longitud constante. En este proyecto los datos son largos y con extensión variable. En el foro de *AppInventor* [24] existen varias consultas por el mismo tema sin una solución clara.

En este punto del proyecto se decide cambiar de estrategia con el dispositivo receptor para poder cumplir el objetivo de terminar el prototipo y que sea funcional.

6.2.4. Recepción mensajes CAN FD por BLE (Raspberry Pi)

Para poner en funcionamiento la Raspberry Pi, es necesario descargar el sistema operativo. En este proyecto utilizaremos Raspberry PI OS y se puede descargar desde la página web oficial (https://www.raspberrypi.com/software/operating-systems/).

Se dispone de tres opciones para descargar el sistema operativo: Raspberry Pi OS Lite,

Raspberry Pi OS con escritorio y Raspberry Pi OS con escritorio más aplicaciones recomendadas. En este proyecto se ha descargado con escritorio sin las aplicaciones recomendadas.

Una vez finalizada la descarga, se guarda en la tarjeta de memoria Micro SD para poder arrancar el sistema operativo y almacenar la información. Al alimentar la Raspberry Pi, se inicia el sistema operativo y se puede configurar. Es necesario conectar un ratón, un teclado y una pantalla.

Al arrancar el sistema se comprueba que ya viene instalado Python3. Si no fuera el caso, se debería instalar.

Para la conexión por BLE es necesario instalar la librería *Bluepy* de Python. La librería y su documentación están disponibles para descargar en el siguiente enlace: https://github.com/IanHarvey/bluepy. Con el objetivo de finalizar la instalación, es necesario introducir los siguientes comandos en el terminal:

```
$ sudo apt-get install python3-pip libglib2.0-dev
$ sudo pip3 install bluepy
```

Para desarrollar esta parte de código se coge como base la estructura del programa ejemplo de la librería *notification.py* y se adapta a las necesidades del proyecto. Se configura correctamente el servicio y la característica que queremos consultar. Es decir, se introduce correctamente el valor de los UUID del servicio y la característica para que coincida con el establecido en el programa de Arduino.

En el programa de Python es necesario indicar la dirección MAC de Bluetooth del Arduino Nano33 IoT utilizado. Hay distintas formas para conocer la dirección del dispositivo. A continuación, se explican, de forma resumida, dos de ellas.

Por un lado, se puede emplear la aplicación LightBlue, solo siendo necesario cargar un programa donde se utilice la comunicación por BLE. Por otro lado, se puede introducir un nuevo comando en el programa de Arduino que mientras espere la conexión con otro dispositivo muestre por pantalla su dirección MAC.

Ambas formas de conseguir la dirección MAC se representan en la Figura 41:

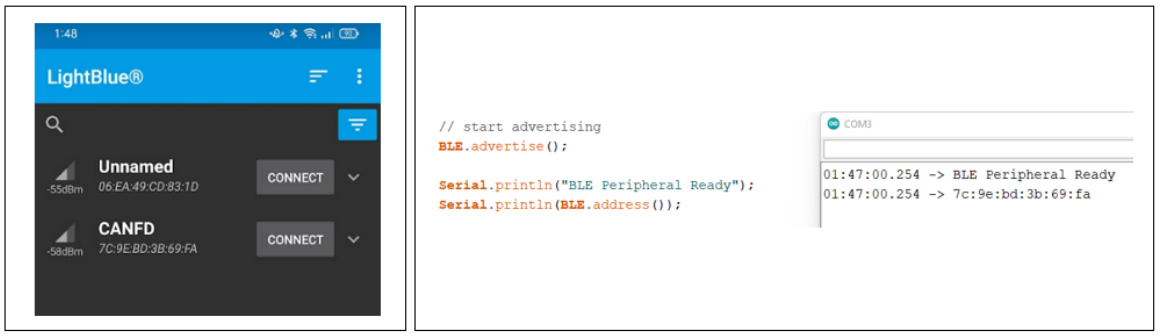

Figura 41.- Dirección MAC BLE Nano33 IoT utilizando: (a) LightBlue, (b) IDE. Fuente: propia.

Se escoge este ejemplo muestra, debido a que se quiere actualizar el valor de la característica cada vez que cambia. Las características de BLE pueden tener la opción de notificar al dispositivo cuando se ha producido un cambio significativo en el valor de la característica. El dispositivo central puede recibir la notificación y extraer el valor, sin necesidad de ir consultando cada intervalo de tiempo. De esta forma, se ahorra tiempo y energía.

En Arduino cargamos el programa *BatteryMonitor_ejemplo* donde cada 200 milisegundos se actualiza un valor de batería simulado (se genera un valor aleatorio entre 1 y 10) y se muestra por pantalla. Para poder disponer de una característica con este servicio de notificaciones, es necesario introducir la propiedad *BLENotify* cuando se crea la característica.

Problema encontrado: Al ejecutar los programas se observa como la parte de Arduino funciona correctamente, es decir, es capaz de ir mostrando los valores de batería, mientras que la Raspberry pi solo muestra una parte, es decir, corta el valor (véase Figuras 42 y 43).

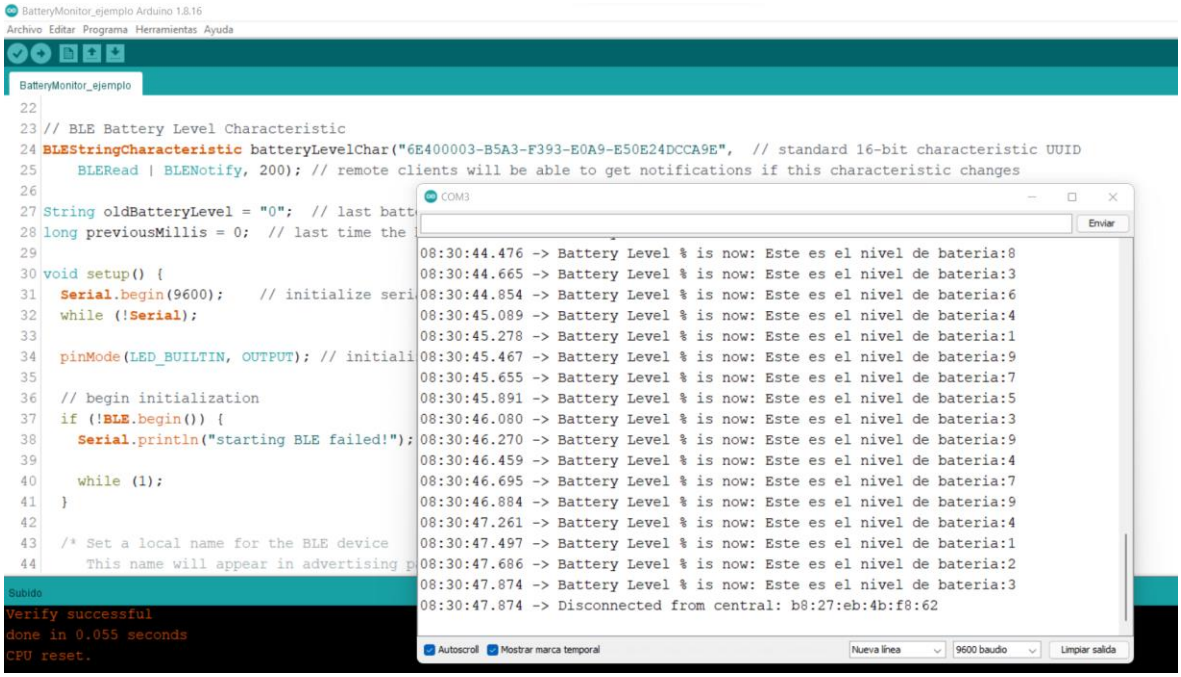

Figura 42.- Nivel de batería mostrado por Arduino. Fuente: propia.

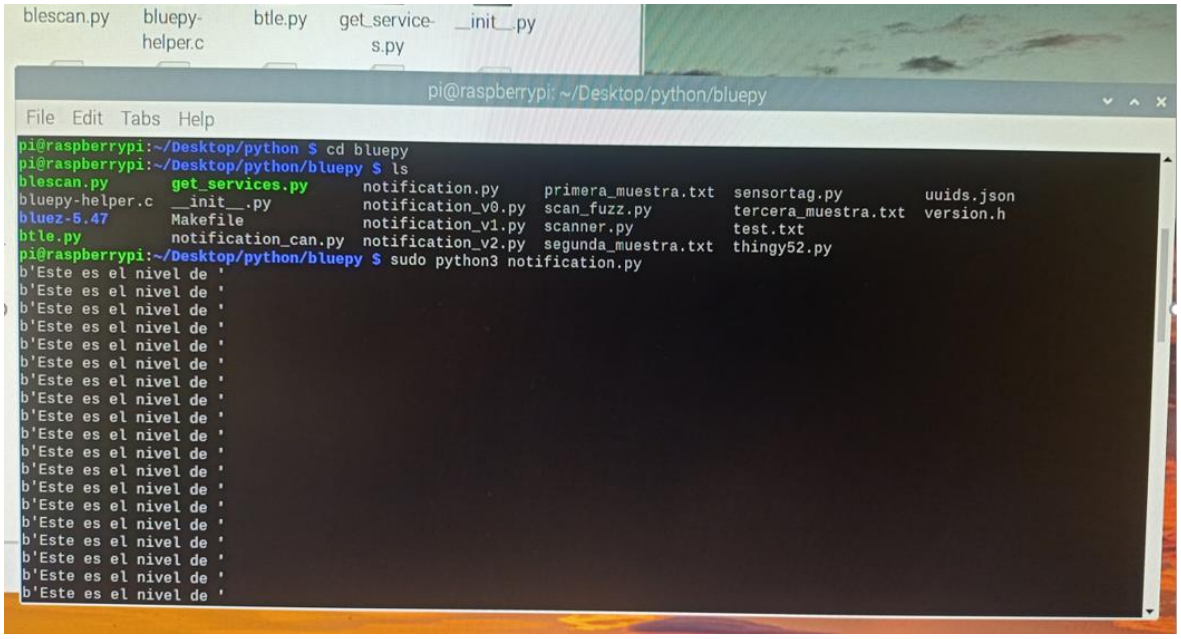

Figura 43.- Nivel de batería mostrado por RaspberryPi. Fuente: propia.

Solución: Introducir en la inicialización del programa de Python la instrucción para ampliar el MTU (*p.setMTU(200)*). Una vez introducido este cambio, el programa funciona correctamente y se puede considerar que funciona adecuadamente.

Figura 44.- Nivel de batería al cambiar el valor de MTU. Fuente: propia.

Después de modificar el código para recibir el mensaje completo, se da por finalizada la conexión entre Arduino Nano33 IoT y la Raspberry Pi. El único punto pendiente por verificar es la cantidad de mensajes que se reciben y si hay alguno que se pierde.

Para esta última comprobación, el ejercicio realizado consiste en enviar 100 mensajes de CAN FD con identificadores correlativos y así poder detectar si hay pérdida o no de información. Los mensajes enviados se dejan almacenados en un archivo de texto (*test.txt*) en la misma carpeta de trabajo.

Los archivos cargados son:

- Raspberry Pi: *notification_v1.py*
- Arduino: *identificadores_correlativos1* (solo los datos son aleatorios, la longitud del mensaje es la máxima 64 bytes y los identificadores correlativos).

Desde que se realiza la conexión hasta el primer mensaje guardado, se pierden algunos mensajes. Una vez se guarda el primer valor, el resto se recibe sin problemas. En las Figuras 45 y 46, se observa como los mensajes enviados son los mismo que se reciben en la Raspberry Pi y en Arduino (*InternalLoopBack*).

Figura 45.- Últimos mensajes enviados con Identificadores_correlativos1. Fuente: propia.

Figura 46.- Últimos mensajes recibidos con Identificadores_correlativos1. Fuente: propia.

Se ha tratado de ampliar el tiempo añadiendo un retraso (*delay*) entre el inicio de la conexión y el primer envío, pero sin resultados favorables. También se ha ampliado el tiempo entre dos envíos correlativos, al ralentizar el envío perdemos menos mensajes, pero se siguen perdiendo.

En el programa *Identificadores_correlativos1* se ha introducido un retraso de 20 ms entre envío de dos mensajes correlativos. Con este tiempo entre envíos, se pierden, de media, los 15 primeros mensajes.

No se considera un problema importante. Es un punto para tener en cuenta de cara a futuras ampliaciones y mejoras.

Se hace una segunda versión del programa de Arduino con identificadores correlativos (*identificadores_correlativos2*), donde tanto la longitud como los datos son completamente aleatorios. El funcionamiento sigue siendo correcto si no se tiene en cuenta los mensajes iniciales perdidos. No se adjuntan las capturas para comparar los valores, pero se puede consultar en el anexo.

6.3. Fase 3: Gateway CANFD – BLE

En esta última fase de experimentación se comprueba el funcionamiento correcto del conjunto. Inicialmente se comprueba con mensajes CAN FD emulados y, por último, se comprueba con la maqueta.

6.3.1. Gateway CAN FD – BLE (mensajes emulados)

Se emplean los siguientes programas para el funcionamiento del envío y recepción de mensajes CAN FD emulados:

- Arduino: *Lectura_can_completa*
- Raspberry Pi: *notification_v1.py*

Tal y como se muestra en las Figura 47 y 48, los mensajes enviados (mostrados por el monitor serie) se corresponden a los mensajes recibidos (guardados en el documento de texto).

```
22:52:42.017 -> Identificador mensaje enviado: 3CF  Longitud mensaje enviado: 12
22:52:42.017 -> mensaje enviado: 34C972FA871D2E207D8B268F
22:52:42.536 -> Identificador obtenido: 3CF  Longitud mensaje obtenido: 12
22:52:42.536 -> mensaje obtenido: 34C972FA871D2E207D8B268F
22:52:42.536 -> Identificador mensaje enviado: 677  Longitud mensaje enviado: 1
22:52:42.536 -> mensaje enviado: BB
22:52:43.016 -> Identificador obtenido: 677  Longitud mensaje obtenido: 1
22:52:43.016 -> mensaje obtenido: BB
22:52:43.016 -> Identificador mensaje enviado: DF  Longitud mensaje enviado: 26
22:52:43.016 -> mensaje enviado: B572EDA0B36447C5287176C93D32E31E18118591966F8FDE5D
22:52:43.536 -> Identificador mensaje enviado: 104  Longitud mensaje enviado: 26
22:52:43.536 -> mensaje enviado: 10ED8A5DD4BAA5FA6AC4DFA3E0ACA315405DED543524E82E3086
22:52:44.046 -> Identificador mensaje enviado: 465  Longitud mensaje enviado: 64
22:52:44.046 -> mensaje enviado: 6E8C2884934611B3244830D8769C6EC9482D94E397BBCE6CC5112754AF915297E349D5C69AC5C5B15694CAC54EAED4E43A5647DC86CF947B0CD4FBBA0F1837F
22:52:44.556 -> Identificador obtenido: 465  Longitud mensaje obtenido: 64
22:52:44.556 -> mensaje obtenido: 6E8C2884934611B3244830D8769C6EC9482D94E397BBCE6CC5112754AF915297E349D5C69AC5C5B15694CAC54EAED4E43A5647DC86CF947B0CD4FBBA0F1837F
22:52:44.556 -> Identificador mensaje enviado: 68F  Longitud mensaje enviado: 16
22:52:44.556 -> mensaje enviado: EC2591E80E070DE8D37B6CB0148253
22:52:45.077 -> Identificador obtenido: 68F  Longitud mensaje obtenido: 16
22:52:45.077 -> mensaje obtenido: EC2591E80E070DE8D37B6CB0148253
22:52:45.077 -> Identificador mensaje enviado: 241  Longitud mensaje enviado: 6
22:52:45.077 -> mensaje enviado: B43D6FE09FA
22:52:45.556 -> Identificador obtenido: 241  Longitud mensaje obtenido: 6
22:52:45.556 -> mensaje obtenido: B43D6FE09FA
22:52:45.556 -> Identificador mensaje enviado: 4FE  Longitud mensaje enviado: 32
22:52:45.556 -> mensaje enviado: E6BDA5CC3CB6C3CD1FBA8A1EF8161381F45219260465C9D31C5A64E79B011E
22:52:46.086 -> Identificador obtenido: 4FE  Longitud mensaje obtenido: 32
22:52:46.086 -> mensaje obtenido: E6BDA5CC3CB6C3CD1FBA8A1EF8161381F45219260465C9D31C5A64E79B011E
22:52:46.086 -> Identificador mensaje enviado: 14B  Longitud mensaje enviado: 6
22:52:46.086 -> mensaje enviado: 385294287861
22:52:46.556 -> Identificador obtenido: 14B  Longitud mensaje obtenido: 6
22:52:46.556 -> mensaje obtenido: 385294287861
22:52:46.556 -> Identificador mensaje enviado: 7F2  Longitud mensaje enviado: 8
22:52:46.592 -> mensaje enviado: D6C432882891D1C
22:52:47.066 -> Identificador obtenido: 7F2  Longitud mensaje obtenido: 8
22:52:47.066 -> mensaje obtenido: D6C432882891D1C
```

Figura 47.- Mensajes aleatorios generados. Fuente: propia.

```
b'3cf:12:34c972fa871d2e207d8b268f'
b'677:1:bb'
b'465:64:6e8c2884934611b3244830d8769c6ec9482d94e397bbce6cc5112754af915297e349d5c69ac5c5b15694cac54eaed4e43a5647dc86cf947b0cd4fbba0f1837f'
b'68f:16:ec2591e80e070de8d37b6cb0148253'
b'241:6:b43d6fe09fa'
b'4fe:32:e6bda5cc3cb6c3cd1fba8a1ef8161381f45219260465c9d31c5a64e79b011e'
b'14b:6:385294287861'
b'7f2:8:d6c432882891d1c'
```

Figura 48.- Mensajes aleatorios recibidos. Fuente: propia.

En la memoria del proyecto solo se incluye una parte de los mensajes enviados y recibidos con la finalidad de realizar la comparativa. Los archivos de texto con el conjunto completo se pueden consultar en el anexo.

Se considera que el prototipo funciona correctamente con mensajes emulados.

6.3.2. Gateway CAN FD – BLE (mensajes reales)

Para finalizar con la validación del prototipo, se realizan un conjunto de tests en un entorno real de trabajo, conectándolo a la maqueta CANFD del laboratorio.

Se emplean los siguientes programas para el funcionamiento del envío y recepción de mensajes CAN FD reales:

- Arduino: *maqueta_completa*
- Raspberry Pi: *notification_v1.py*

Para poder recibir los mensajes proporcionados por la maqueta, se fija el modo *ListenOnly* y se anula la instancia que llama a la función *EnviarCAN().* Es importante comprobar que las velocidades del CAN FD de la maqueta corresponden a las introducidas en la configuración inicial de la librería ACAN2517FD.

Una vez establecida la conexión BLE de las dos placas, se envía un conjunto de 100 mensajes CAN FD desde la ECU1. Se realiza el mismo ejercicio con la ECU2. En ambos casos, se reciben correctamente los datos enviados.

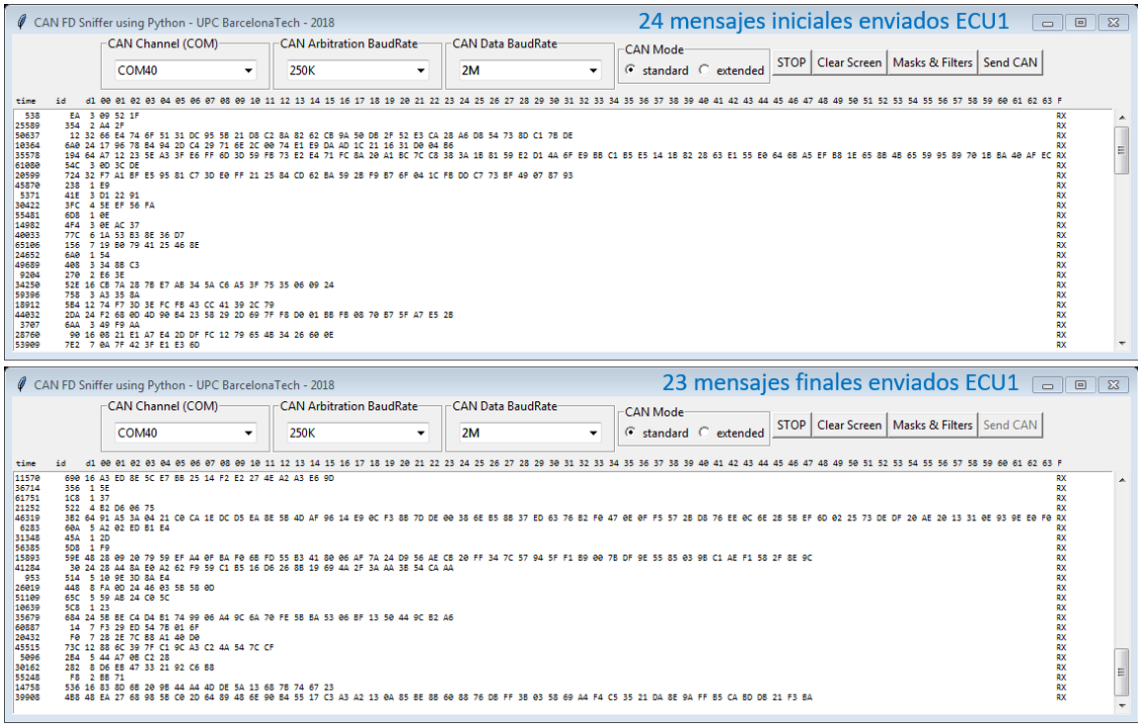

Figura 49.- Captura de los mensajes iniciales y finales enviados por ECU1. Fuente: propia.

24 mensajes iniciales recibidos

```
File  Edit  Format  View  Help
b'ea:3:9521f'
b'354:2:a42f'
b'12:32:66e4746f5131dc955b21d8c28a8262cb9a50db2f52e3ca28a6d854738dc17bde'
b'6a0:24:179678b4942dc429716e2c074e1e9daad1c211631d04b6'
b'194:64:a712235ea33fe6ff6d3d59fb73e2e471fc8a20a1bc7cc8383a1b8159e2d14a6fe9bbc1b5e5141b822863e155e0646ba5efb81e658b4b65599589701bba40afec'
b'54c:3:d3cde'
b'724:32:f7a1bfe59581c73de0ff212584cd62ba592bf9b76f41cfbddc773bf4978793'
b'238:1:e9'
b'41e:3:d12291'
b'3fc:4:5eef56fa'
b'6d8:1:e'
b'4f4:3:eac37'
b'77c:6:1a53b38e36d7'
b'156:7:19b0794125468e'
b'6a0:1:54'
b'408:3:348bc3'
b'270:2:e63e'
b'52e:16:cb7a287be7ab345ac6a53f75356924'
b'758:3:a3358a'
b'5b4:12:74f73d3efcfb43cc41392c79'
b'2da:24:f268d4d90b42358292d697ff8d01bbfb870b75fa7e52b'
b'6aa:3:49f9aa'
b'90:16:821e1a7e42ddfffc1279654b342660e'
b'7e2:7:a7f423fe1e36d'

...

b'690:16:a3ed8e5ce7bb2514f2e2274ea2a3e69d'
b'356:1:5e'
b'1c8:1:37'
b'522:4:b2d6675'
b'3b2:64:91a53a421c0ca1edcd5ea8e5b4daf9614e9cf38b7dde0386eb58b37ed6376b2f047eff5572bd876eec6e2b5bef6d22573dedf20ae201331e939ee0f0'
b'60a:5:a22edb1e4'
b'45a:1:2d'
b'5d8:1:f9'
b'59e:48:289207959efa4fbaf06bfd55b341806af7a24d956aecb20ff347c57945ff1b907bdf9e558539bc1aef1582f8e9c'
b'30:24:28a48ae0a262f959c1b516d6268b19694a2f3aaa3b54caaa'
b'514:5:109e3d8ae4'
b'448:8:fad244635b58d'
b'65c:5:59ab24c05c'
b'5c8:1:23'
b'684:24:5bbec4d4b174996a49c6a70fe5bba536bf1350449cb2a6'
b'14:7:f329ed547b16f'
b'f0:7:282e7cb8a140d0'
b'73c:12:886c397fc19ca3c24a547ccf'
b'2b4:5:44a7bc228'
b'282:8:d6eb47332192c6b8'
b'f8:2:bb71'
b'536:16:838d6b209b44a44dde5a13687b746723'
b'4b8:48:ea2768985bc02d6489486e90b45517c3a3a213a85be8b608876dbff3b35869a4f4c53521da8e9affb5cabddb21f3ba'
```

23 mensajes finales recibidos

Figura 50.- Captura de los mensajes iniciales y finales recibidos (ECU1). Fuente: propia.

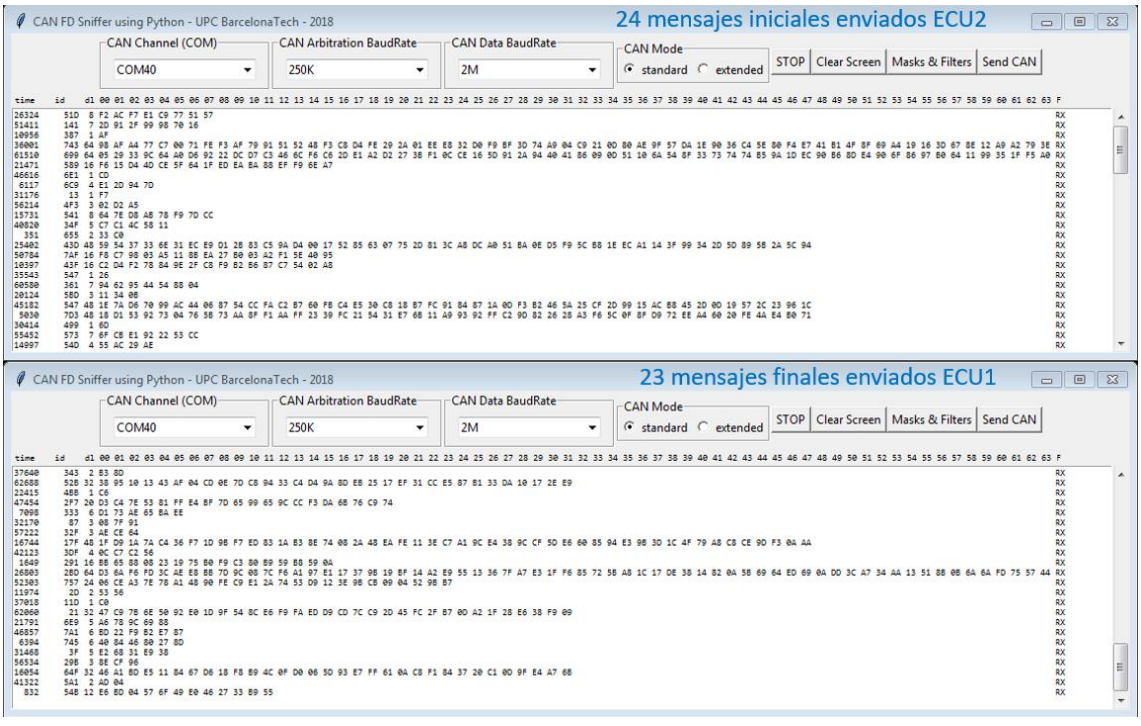

Figura 51.- Captura de los mensajes iniciales y finales enviados por ECU2. Fuente: propia.

ETSEIB

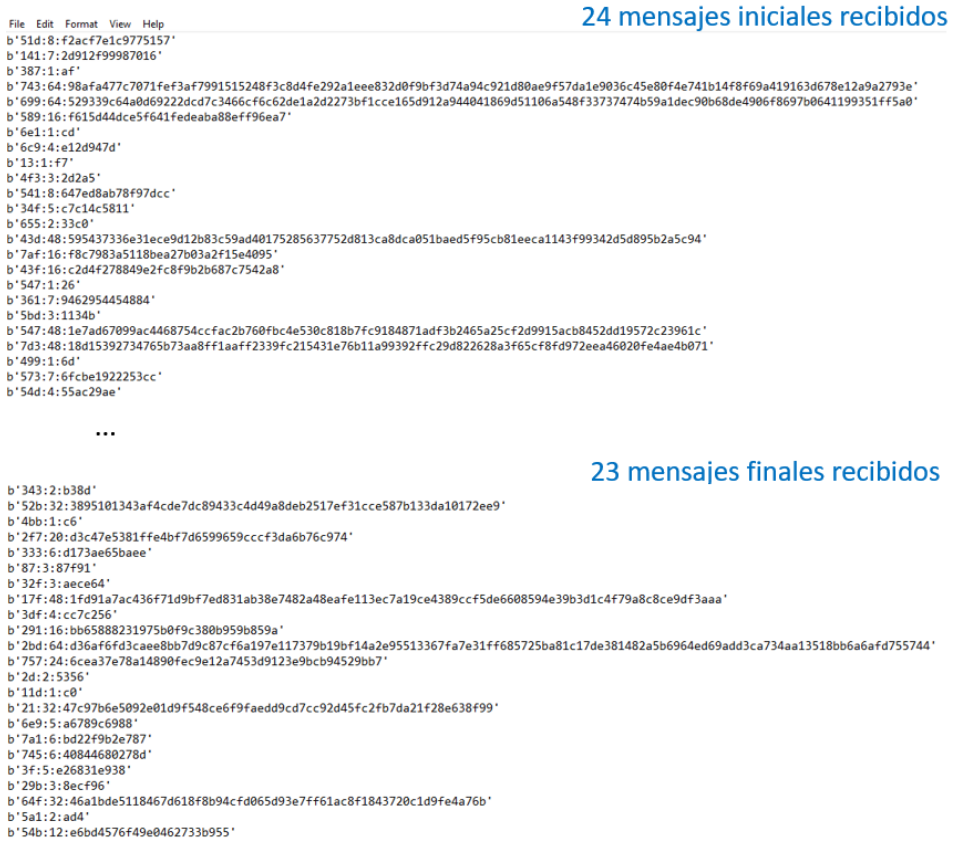

Figura 52.- Captura de los mensajes iniciales y finales recibidos (ECU2). Fuente: propia.

Cabe destacar que, en este caso, no se produce una pérdida de los datos iniciales como sucedía al trabajar con mensajes emulados. Se recibe el conjunto completo de los mensajes enviados.

Para finalizar la validación del dispositivo completo, se envían un conjunto de mensajes individuales y se compara con los mensajes recibidos.

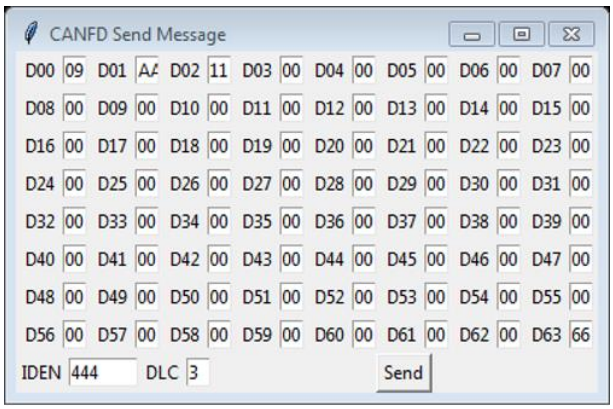

Figura 53.- Último mensaje individual enviado por la maqueta. Fuente: propia.

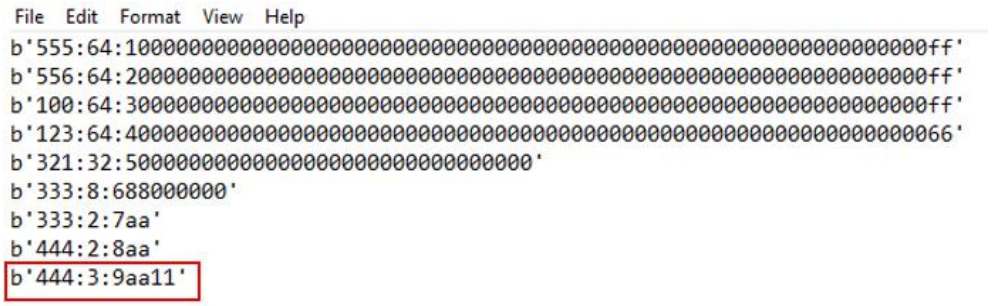

Figura 54.- Mensajes individuales recibidos. Fuente: propia.

7. Planificación y presupuesto del proyecto

7.1. Planificación temporal de las fases del proyecto

El inicio real de este trabajo de final de máster fue en septiembre de 2021, aunque durante el mes de julio de 2021 se realizaron un par de reuniones previas con el tutor para investigar temas, opciones de trabajos y formalizar la matrícula.

Inicialmente, la idea era presentar el proyecto en enero de 2022 (final del cuatrimestre de otoño), pero por problemas familiares y carga laboral, se decidió prorrogar la entrega y disponer de tres meses adicionales (8 meses en total).

La primera fase del proyecto se inicia en el momento en que se conoce la temática del trabajo. En esta primera fase se realiza una lectura de los proyectos que preceden a éste para saber hasta dónde han llegado los compañeros, su metodología de trabajo y los problemas que se encontraron para intentar evitarlos y no trabajar en vano [2][25].

Es en esta primera fase de proyecto donde se determinan los objetivos y el alcance del proyecto, siempre teniendo en cuenta que se trata de un trabajo final de máster de 12 créditos y que corresponde, aproximadamente, a unas 300h de trabajo (1ECTS ≈ 25h).

Durante el mes de septiembre y octubre de 2021 se decide que módulos se van a utilizar y como se va a realizar la comunicación entre ellos. Para ver si la arquitectura de la solución propuesta es compatible con los elementos seleccionados, tanto a nivel de *hardware* (pines físicos y módulos) *como software* (librerías). Durante estos dos meses se realiza un trabajo de predesarrollo para detectar posibles incompatibilidades.

A mediados de octubre se recibe el Arduino Nano33 IoT. Es en este punto donde se empieza a trabajar con el desarrollo y la validación del prototipo de la puerta de enlace CAN FD-BLE. Constituye la fase principal (o de más carga de trabajo) de este proyecto y se extiende hasta finales de marzo. Como se ha mencionado en el capítulo 6 de la memoria, para facilitar el proceso, la fase de desarrollo y validación se ha dividido en las siguientes tres sub-fases:

- CANFD-Arduino
- Arduino-dispositivo receptor: inicialmente se utiliza un dispositivo móvil con una aplicación creada mediante *AppInventor,* pero se descarta esta alternativa y finalmente se emplea como receptor una *RaspberryPi*.
- Puerta de enlace CAN FD – BLE completa

Para finalizar el proyecto, durante el mes de abril, se realiza un pequeño estudio sobre el impacto medioambiental del proyecto y una estimación de los costes.

Paralelamente, a este conjunto de fases, se lleva a cabo la redacción de la memoria. Sin embargo, esta actividad toma un peso muy importante durante los dos últimos meses de proyecto.

En la Figura 55 se muestra la distribución temporal de las fases del proyecto anteriormente descritas. En la parte superior, se muestran los hitos más significativos de la evolución del proyecto.

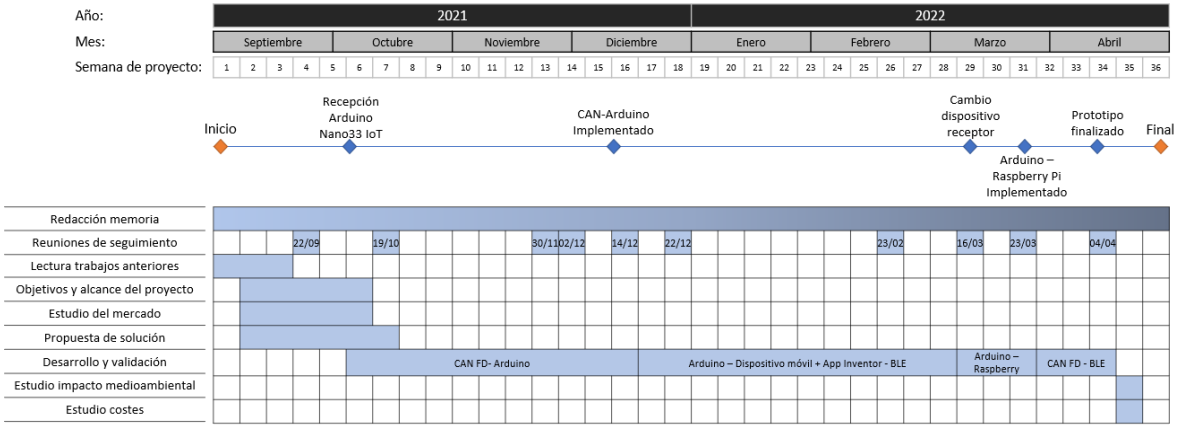

Figura 55.- Distribución temporal de las fases del proyecto. Fuente: propia.

Como se observa en la distribución temporal, durante el proyecto se dedica mucho tiempo para hacer funcionar la conexión por BLE entre Arduino y el dispositivo móvil. El módulo de BLE desarrollado para utilizar en el *AppInventor* no acaba de funcionar correctamente y se dedica tiempo extra para buscar el fallo y la solución.

7.2. Presupuesto del proyecto

Para elaborar un presupuesto del proyecto, inicialmente se han realizado una estimación de costes en función de diferentes factores. Como se observa en la Tabla 6, el coste total del proyecto se divide en distintos bloques/partidas.

Para el cálculo del material utilizado, hemos contabilizado el precio de compra de las distintas placas empleadas, el cable USB del Arduino Nano33 IoT y el cable HDMI para conectar la RaspberryPi a la pantalla. Se ha despreciado el precio del cableado empleado para realizar la conexión SPI entre la placa MCP2517FD y Arduino Nano33 IoT.

En el apartado del coste personal se ha tenido en cuenta las horas dedicadas por el

investigador y el director del proyecto. En el caso del investigador, se considera un precio de 20 €/h, mientras que las horas dedicadas al seguimiento del proyecto por el director corresponden a 40 €/h.

Se ha realizado una estimación de consumo eléctrico y el coste que este representa en el proyecto. Se considera 120 W consumo medio del ordenador portátil + placas y 30 W de iluminación del lugar de trabajo (4 bombillas LED x 6 W/bombilla). Para el coste se ha seleccionado el precio medio de la luz del 01/04/2022 que fue de 0,30523 €/kWh [26].

Para el cálculo de las amortizaciones se ha empleado la siguiente fórmula:

$$\text{Amortización (A)} = \frac{\text{valor adquisición [€]}}{\text{tiempo de vida útil [h]}} \times \text{Tiempo de uso [h]}$$

Únicamente se ha considerado amortización del ordenador portátil empleado para el desarrollo del proyecto y el teléfono móvil (*smartphone*). Aunque también se ha necesitado puntualmente una pantalla, ratón y teclado, se considera que es un coste no significativo como para tenerlo en cuenta.

El tiempo de vida útil de estos dos dispositivos se ha estimado en 4 años, cuando se considera que queda obsoleto (no deja de ser funcional). Aproximadamente 8000 h de trabajo (4 años x 250 días laborables/año x 8 h de trabajo/día laboral).

	Concepto	Cantidad	[€]/unidad		Total [€]
Material	MCP2517FD Click board	1	23,05		23,05
	Arduino Nano33 IoT	1	16		16
	RaspberryPi 3b+	1	42,3		42,3
	USB Cable Type A Male to Micro Type B Male	1	4,9		4,9
	HDMI Cable	1	4,5		4,5
	Concepto	Cantidad [h]	[€]/h		Total [€]
Coste personal	Investigador Junior	300	20		6000
	Director del proyecto	50	40		2000
	Concepto	Potencia [Kw]	Uso [h]	Precio [€/KWh]	Total [€]
Consumo eléctrico	Ordenador	0,12	300	0,30523	10,99
	Iluminación lugar de trabajo	0,045	300	0,30523	4,12
	Concepto	Precio compra [€]	Vida útil estimada [h]	Uso [h]	Total [€]
Amortizaciones	Ordenador	600	8000	300	22,50
	Teléfono móvil	250	8000	50	1,56
	Concepto				Total [€]
Totales	Costes totales antes de impuestos				8129,92
	IVA	21%			1707,28
	Coste total del proyecto				9837,20

Tabla 6.- Cálculo del coste del proyecto. Fuente: propia.

8. Impacto medioambiental

Todas las actividades tienen asociadas diferentes elementos que pueden llegar a interactuar con el medio ambiente y, por lo tanto, pueden tener un posible impacto en el medio ambiente.

El estudio del impacto ambiental es obligatorio para todos los proyectos. La realización de este estudio permite identificar los aspectos susceptibles de producir impacto ambiental y minimizarlos.

En este caso, al tratarse de un proyecto de investigación con una parte de experimentación, el mayor impacto será el que pueda provocar la energía consumida por las máquinas utilizadas.

8.1. Emisiones

La principal fuente de emisión de este proyecto es aquella producida como consecuencia de la energía eléctrica consumida. La energía eléctrica consumida en España puede provenir de diferentes tipos de producción, por lo tanto, para el cálculo de emisiones se utilizará el valor del MIX energético 2016 proporcionado por WWF [27].

La energía consumida durante la realización de este proyecto es de 49,5 kWh (véase en Presupuesto del proyecto).

Principales emisiones	MIX (WWF España 2016)	Total emisiones
Dióxido de carbono (CO_2)	0,174 kg/kWh	8,613 kg
Dióxido de azufre	0,366 g/kWh	18,117 g
Óxido de Nitrógeno (NO_x)	0,261 g/KWh	12,9195 g

Tabla 7.- Principales emisiones provenientes del consumo de electricidad. Fuente: propia.

8.2. Residuos

El único residuo que se puede considerar tras la realización de este proyecto es el prototipo construido. Una vez finalizado el proyecto, estas placas se pueden reutilizar en otros proyectos, ya sea como pack completo o como componentes individuales.

8.3. Vertidos

Los principales vertidos son las aguas domésticas (grises y negras). Se considera que no tienen un impacto significativo. Siempre se puede recomendar el uso adecuado del agua.

8.4. Ruido

La exposición continuada de una persona al ruido puede causarle molestia. En casos de ruido elevado, las consecuencias pueden ser más importantes como, por ejemplo, fatiga, estrés, insomnio…

La OMS (Organización Mundial de la Salud) recomienda que para zonas de oficina y trabajo administrativo haya un nivel de ruido inferior a 65 dB [28].

En este proyecto, la principal fuente de ruido proviene del ordenador, pero no se considera que tenga un impacto significativo, puesto que, corresponde a un valor entre 20 y 40 dB.

8.5. Conclusiones del estudio de impacto medioambiental

Como se ha mencionado anteriormente, el mayor impacto medioambiental en este proyecto está provocado por el consumo de energía eléctrica. En ningún caso, el impacto ambiental es significativo.

Se debe considerar que utilizar CANFD y BLE es mucho más eficiente que utilizar CAN y Bluetooth clásico. Por lo tanto, en automoción, puede suponer un impacto positivo de cara al medioambiente.

9. Conclusiones

En este apartado se presentan las conclusiones más relevantes que se han obtenido al finalizar el proyecto. Además, se proponen unas líneas de trabajo para futuras investigaciones.

Fijándose en el resultado final del proyecto, se puede afirmar que ha sido un éxito. Se ha cumplido con el objetivo principal de desarrollar una puerta de enlace CAN FD – BLE funcional. El dispositivo funciona correctamente en los dos escenarios posibles, trabajando con mensajes de CAN FD emulados y reales.

Durante el desarrollo del prototipo se dedicó mucho tiempo intentando desarrollar una aplicación para el teléfono móvil (dispositivo receptor) utilizando *AppInventor*. Se puede concluir que fue una estrategia errónea. Al tratarse de un desarrollador de aplicaciones por bloques estándar, las aplicaciones obtenidas no son óptimas y, por lo tanto, suelen emplear más tiempo en realizar las instrucciones. En el caso de mensajería CAN FD, donde es necesario poder transmitir gran cantidad de datos con longitudes variables, no es una buena solución.

El cambio de estrategia respecto al dispositivo receptor provocó un retraso en la planificación, haciendo imposible la introducción de filtros o post tratamiento de datos por falta de tiempo.

Con el estudio del mercado realizado, se pone de manifiesto que las soluciones existentes en el mercado son muy caras. Centrarse en buscar y construir un dispositivo de bajo coste puede ser muy rentable debido a la baja oferta.

Desde luego, el desarrollo de la puerta de enlace CAN FD – BLE no acaba aquí. Este proyecto es la base de dónde se puede partir para acabar teniendo un dispositivo realmente competitivo.

Como proyectos futuros se propone:

- Introducir un filtrado para los mensajes de CAN FD.
- Estudiar la viabilidad de fusionar este proyecto con el proyecto previo de CANFD – Wifi [2]. Arduino Nano33 IoT incorpora el chip que permite la conexión Bluetooth y Wifi de la placa.
- Crear una aplicación para *Smartphones* donde se puedan recibir y mostrar los mensajes.

Agradecimientos

Antes de finalizar este trabajo, me gustaría poder agradecer a las personas que han sido, de forma directa o indirecta, partícipes de este proyecto.

Agradecer al director del proyecto Manuel Moreno por el apoyo y dedicación durante estos últimos meses. Disponible para guiarme o resolver cualquier duda en todo momento.

Por último, agradecer a mi familia y amigos por el estímulo constante para que pueda mejorar día a día, capaces de motivarme en los momentos más difíciles.

Bibliografía

[1] CIA, «CAN in Automation (CiA): History of the CAN technology,» Agosto 2015. [En línea]. Disponible: https://www.can-cia.org/can-knowledge/can/can-history. [Último acceso: Diciembre 2021].

[2] M. De Pfaff Ganduxer, «Puerta de enlace CANFD-Wifi basada en Arduino,» ETSEIB, Universitat Politècnica de Catalunya, Barcelona, Enero 2020. [Último acceso: Abril 2022].

[3] Automatización Industrial - Info PLC «CANBlue II Transmite datos CAN por Bluetooth,» Agosto 2013. [En línea]. Disponible: https://www3.infoplc.net/noticias/item/101313-canblue-ii-transmite-datos-can-por-bluetooth. [Último acceso: Diciembre 2021].

[4] Ixxat by HMS Networks, «CANblue II,» Octubre 2020. [En línea]. Disponible: https://www.ixxat.com/products/automotive-solutions/overview/can-bridges-and-gateways/canblue-ii. [Último acceso: Enero 2022].

[5] Ixxat, «Can and CAN FD solutions,» Enero 2020. [En línea]. Disponible: https://docs.rs-online.com/0b39/A700000008006684.pdf. [Último acceso: Enero 2022].

[6] RS Components, «Ixxat CANblue II ext. Antenna,» Enero 2022. [En línea]. Disponible: https://es.rs-online.com/web/p/routers-industriales/2262737. [Último acceso: Enero 2022].

[7] Axiomatic Global Electronic Solutions, «AX141100: CAN-Bluetooth Gateway,» Febrero 2018. [En línea]. Disponible: https://www.axiomatic.com/protocol-converter/can-bluetooth/. [Último acceso: Enero 2022].

[8] GCAN - Guangcheng CAN-Bus mall, «GCAN-203 Bluetooth to CAN converter,» Enero 2022. [En línea]. Disponible: http://www1.gcanbox.com/fsd/canzxwg/GCAN-203.html. [Último acceso: Enero 2022].

[9] Tecnologia ALIFPGA - Hangzhou, «Descripción general del bus CAN,» 2021. [En línea]. Disponible: https://programmerclick.com/article/22561472838/. [Último acceso: Diciembre 2021].

[10] Vector, «Introduction to CAN - Learning Module CAN,» Febrero 2022. [En línea]. Disponible: https://elearning.vector.com/mod/page/view.php?id=333.

[11] Wikipedia, «Bus CAN,» Agosto 2021. [En línea]. Disponible: https://es.wikipedia.org/wiki/Bus_CAN. [Último acceso: Diciembre 2021].

[12] Kvaser, «Una introducción básica al bus CAN,» [En línea]. Disponible: https://www.kvaser.com/es/course/conceptos-basicos-de-can/. [Último acceso: Diciembre 2021].

[13] Embitel , «Classical CAN vs CAN FD,» Julio 2018. [En línea]. Disponible: https://www.embitel.com/blog/embedded-blog/classical-can-vs-can-fd-decoding-their-data-transfer-capabilities-and-compatibility-with-the-bootloader-software. [Último acceso: Enero 2022].

[14] Bluetooth, «Bluetooth Technology Overview,» Enero 2022. [En línea]. Disponible: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/. [Último acceso: Febrero 2022].

[15] Wikipedia, «Bluetooth - Wikipedia,» 11 Abril 2022. [En línea]. Disponible: https://en.wikipedia.org/wiki/Bluetooth. [Último acceso: Febrero 2022].

[16] Arduino, «ArduinoBLE,» Marzo 2022. [En línea]. Disponible: https://www.arduino.cc/reference/en/libraries/arduinoble/. [Último acceso: Abril 2022].

[17] Microchip, «Serial Peripheral Interface (SPI),» Enero 2021. [En línea]. Disponible: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/core-independent-and-analog-peripherals/communication-connectivity-peripherals/spi-peripherals. [Último acceso: Diciembre 2021].

[18] David Kalinsky y Roe Kalinsky, «Introduction to Serial Peripheral Interface,» Enero 2002. Disponible:https://web.archive.org/web/20061111110015/http://www.embedded.com/showArticle.jhtml?articleID=9900483. [Último acceso: Diciembre 2021].

[19] Mikroe, «MCP2517FD click - Mikroe,» 2021. [En línea]. Disponible: https://www.mikroe.com/mcp2517fd-click. [Último acceso: Febrero 2022].

[20] Arduino Nano33 IoT, «Arduino Nano 33 IoT - Arduino documentation,» Noviembre 2021. [En línea]. Disponible: https://docs.arduino.cc/hardware/nano-33-iot. [Último acceso: Abril 2022].

[21] Raspberry Pi, «Raspberry Pi 3 Model B+,» Marzo 2022 2021. [En línea]. Disponible:

https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. [Último acceso: Abril 2022].

[22] Arduino Learn, «Arduino Learn,» Enero 2022. [En línea]. Disponible: https://docs.arduino.cc/learn/. [Último acceso: Abril 2022].

[23] MIT App Inventor, «MIT App inventor - About Us,» Enero 2022. [En línea]. Disponible: http://appinventor.mit.edu/about-us. [Último acceso: Abril 2022].

[24] MIT App Inventor , «MIT App Inventor Community,» Febrero 2019. [En línea]. Disponible: https://community.appinventor.mit.edu/. [Último acceso: Marzo 2022].

[25] A. Sastre Martinez, «CANFD Data Logger based on Arduino,» ETSEIB, Universitat Politècnica de Catalunya, Barcelona, 2020.

[26] Tarifaluzhora, «Precio de la luz por horas,» Abril 2022. [En línea]. Disponible: https://tarifaluzhora.es/. [Último acceso: Abril 2022].

[27] WWF España, «Observatorio de la Electricidad año 2016,» 2017. [En línea]. Disponible: http://awsassets.wwf.es/downloads/oe_anual_2016.pdf. [Último acceso: Abril 2022].

[28] World Health Organization, «Guidelines for community noise,» Abril 1999. [En línea]. Disponible: https://www.who.int/publications/i/item/a68672. [Último acceso: Abril 2022].