Trabajo Fin de Grado

Estudiante de ingeniería en tecnologías industriales

Registrador de CANFD utilizando lenguaje C y una Raspberry PI

MEMORIA

Autor: Javier A. De Esteban Garau
Director: Manuel Moreno Eguíllaz

Convocatoria: Junio 2020

Escuela Técnica Superior
De Ingeniería Industrial de Barcelona

Resumen

Este documento describe el proceso de elaboración de un registrador de datos de CAN FD basado en una Raspberry Pi y lenguaje C.

Tras unos apartados de presentación, el escrito describe el sistema del proyecto, recorriendo cada uno de los componentes que lo forman con explicaciones sobre éstos y qué función le corresponde dentro del trabajo. Se analiza el protocolo protagonista de este trabajo, el CAN FD, la arquitectura del proyecto, la Raspberry Pi y una explicación del entorno de desarrollado utilizado.

Posteriormente, se explican los pasos a seguir para el desarrollo del proyecto, describiendo los pasos previos al inicio, como ha sido habilitar la Raspberry Pi y entender su arquitectura, el proceso, que es la programación del controlador del CAN FD para hacerlo funcional en la Raspberry Pi.

Por último, se presentan las validaciones realizadas para asegurar el correcto funcionamiento del sistema, donde se observan los resultados de los test realizados, con una explicación de lo que conllevan los resultados de cada uno.

Índice

ÍNDICE 4

1. **GLOSARIO** 7

2. **PREFACIO** 8

2.1. Origen del proyecto 8
2.2. Motivación 8
2.3. Requerimientos previos 8

3. INTRODUCCIÓN 10

3.1. Objetivos del proyecto 10
3.2. Alcance 10

4. SISTEMA 11

4.1. CAN FD 11
4.1.1. Protocolo CAN 11
4.1.2. CAN FD 13
4.2. MCP2517FD CLICK 14
4.2.1. Controlador MCP2517FD 14
4.2.2. Transceptor ATA6563 15
4.2.3. Interfaz SPI 15
4.2.4. Pines usados 16
4.3. Raspberry Pi 17
4.3.1. GPIO 18
4.3.2. Raspbian 18
4.4. Entorno desarrollo integrado 19
4.5. Sistema 19

5. DESARROLLO 21

5.1. Pasos previos 21
5.2. Desarrollo del controlador 21
5.2.1. Archivos necesarios 22
5.2.2. Validación librería pigpio 22
5.2.2.1. Driver SPI 24
5.2.3. Instrucciones SPI 25
5.2.3.1. Driver CAN FD SPI API 26
5.2.3.2. Validación de las instrucciones 29
5.2.4. Filtros y máscaras 31

5.2.5. Función main 32
5.2.5.1. Configuración de la función main 32
5.2.5.2. Configuración de los mensajes 34
5.2.6. Validación de controlador de CAN FD 35
5.3. Registrador de datos 39
5.3.1. Presentación de los datos 39
5.3.2. Registrador de datos 41
6. PRUEBAS Y RESULTADOS CON EL DATALOGGER 43
6.1. Registrador sin máscaras ni filtros 43
6.2. Registrador usando máscaras y filtros 45
6.2.1. Test 1 46
6.2.2. Test 2 48
6.2.3. Test 3 49
6.2.4. Test 4 51
7. IMPACTO ECONÓMICO 53
8. IMPACTO MEDIO AMBIENTAL 55
CONCLUSIONES Y FUTURO DEL PROYECTO 56
AGRADECIMIENTOS 57
BIBLIOGRAFÍA 58
Referencias bibliográficas 58

1. Glosario

CAN: Controller Area Network

CANFD: Controller Area Network with Flexible Data-rate

CRC: Cyclic Redundancy Check

FDF: Flexible Data-Rate Format

IDE: Integrated Development Environment

RoHS: Restriction of Hazardous Substances.

RPI: Raspberry PI

SRR: Subtitle Remote Request

SPI: Serial Peripheral Interface

Tasa de bits (bitrate): es la cantidad de datos consumidos para transmitir la secuencia de audio por unidad de tiempo.

Trama: Encapsulado de diferentes campos para transmitir información.

2. Prefacio

2.1. Origen del proyecto

El bus CAN es un protocolo de comunicaciones ampliamente usado en entornos y sistemas con requisitos de transferencia de infomación en tiempo real. Fue desarrollado por la compañía Robert Bosch GmbH en 1986 y surgió por la necesidad de conectar cada vez más y más dispositivos electrónicos en el interior de los coches. Debido a sus prestaciones, es frecuentemente utilizado en el sector de la automoción o la aeronáutica, en donde la fiabilidad en las comunicaciones es de vital importancia para el funcionamiento de los sistemas [1].

En 2012, Bosch lanzó al mercado el CAN FD, una actualización del protocolo CAN que optimizaba, entre otras cosas, la flexibilidad en los datos. Permite a cada unidad de control (ECU) cambiar dinámicamente la velocidad de envío de datos y permite una carga de datos de 64 bytes en vez de los 8 bytes a los que estaba limitado su predecesor [1].

Teniendo en cuenta que es relativamente nuevo este sistema de comunicaciones, estudiar el protocolo para poder optimizarlo al mínimo coste posible es vital.

2.2. Motivación

Las razones que han motivado la realización este proyecto son:

En primer lugar, quería hacer un proyecto relacionado con la ingeniería electrónica, ya que es uno de los campos que más me han interesado en la carrera y del cual, en mi opinión, hemos tenido una educación más escueta que con respecto otros ámbitos. Este proyecto era una oportunidad para experimentar un trabajo real en el mundo de la electrónica.

Por otro lado, el proyecto me brindaba la oportunidad de trabajar con software. Teniendo en cuenta que trabajo como informático, aunque sea utilizando Java y no lenguaje C, consideré que la experiencia en adaptarme a otros lenguajes partiendo de lo enseñado en la carrera me ayudaría más que un trabajo centrado en hardware.

2.3. Requerimientos previos

Para poder realizar el proyecto ha sido necesario:

Entender cómo funciona el protocolo CAN FD, paso a paso, para su correcto funcionamiento. Es importante también, entender cómo presenta la información el bus, ya que se ha de poder almacenar la información coherentemente en el registrador de datos. Es por ello por lo que conocer también la arquitectura del bus es imprescindible.

Paralelamente, es importante tener conocimientos suficientes para poder modificar el código base de partida, el cual está en lenguaje C y también para crear el registrador de datos, también desarrollado en dicho lenguaje [6]. Todo ello hace necesario tener nociones de programación y más específicamente de lenguaje C.

3. Introducción

3.1. Objetivos del proyecto

El principal objetivo del proyecto es crear un registrador de datos de CAN FD que sea funcional en una Raspberry Pi. El software del registrador debe estar escrito en lenguaje C.

Para poder realizar dicha meta, se debe completar un conjunto de objetivos previos, tales como instalar un sistema operativo y un entorno de desarrollo integrado IDE en la Raspberry PI, habilitar el controlador de SPI para hacerlo funcional en ese mismo hardware o estudiar y usar correctamente las conexiones entre la RPI y el SPI.

3.2. Alcance

El proyecto se va a centrar en adaptar un controlador o *driver* de CAN FD ofrecido por Microchip Inc. [3] para hacerlo compatible con una Raspberry PI, siendo ésta capaz de enviar y recibir mensajes a través de una red de CAN FD. El proyecto se basará en software, por lo que el hardware no requiere modificaciones.

El controlador de CAN FD será configurado para poder enviar y recibir mensajes a través de FIFOs, pudiéndose opcionalmente aplicar filtros y máscaras para los mensajes recibidos.

4. Sistema

4.1. CAN FD

4.1.1. Protocolo CAN

Como se ha mencionado con anterioridad, el protocolo CAN fue diseñado como método de comunicación entre componentes electrónicos. Basado en una topología bus, permite comunicarse sin necesidad de usar un ordenador como *host*. Fue una revolución para el mercado de la automoción, ya que, entre otras prestaciones, permitía reducir la cantidad de cables que requerían los sistemas de comunicación integrados en los vehículos, abaratando el coste de producción de los automóviles.

El protocolo, como su nombre indica, es una red en área. Esto quiere decir que cada bit de información se envía a la vez, de forma secuencial, a todos los controladores conectados al canal. Los receptores se encargarán de filtrar el mensaje o ignorarlo. El CAN clásico permite enviar mensajes de hasta 8 bytes.

En lo que a la arquitectura respecta, la transmisión de señales en un bus CAN se lleva a cabo a través de un par de cables trenzados. Las señales de estos cables se denominan CAN_H (*CAN high*) y CAN_L (*CAN low*) respectivamente. El bus tiene dos estados definidos: estado dominante y estado recesivo. En estado recesivo, los dos cables del bus se encuentran al mismo nivel de tensión (*common-mode voltage*), mientras que en el estado dominante hay una diferencia de tensión entre las dos señales. La transmisión de señales en forma de tensión diferencial es lo que permite al CAN tener una fuerte robustez contra el ruido electromagnético.

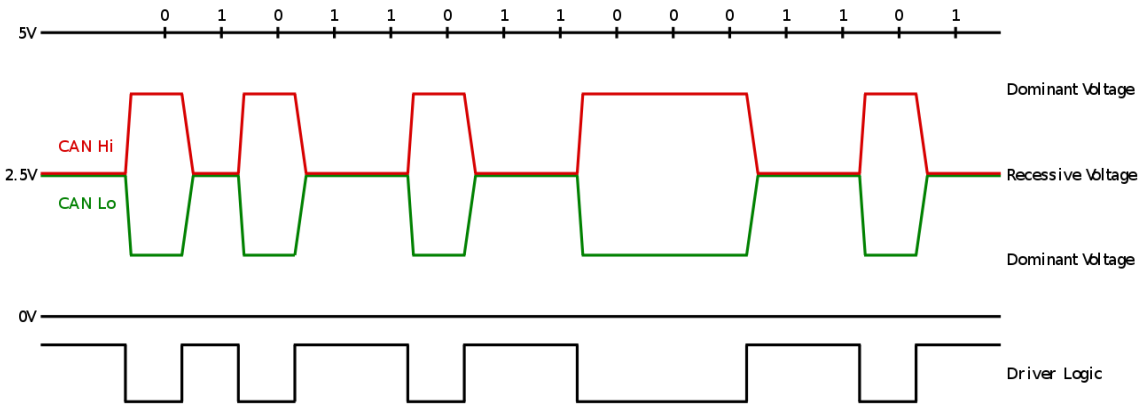

Fig. 1. Diferencia de voltaje en el bus CAN. Fuente: [2].

La trama de datos del CAN consisten en unos bits iniciales, que corresponden con el identificador del mensaje, la carga o *payload*, el CRC y algún otro bit de control extra. El mensaje puede ser *Estándar*, con un identificador de 11 bits, o *Extendido*, con un identificador de 29 bits.

Ya se ha hablado de los dos estados posibles en el bus, dominante y recesivo. Estos estados se refieren a los bits que entrega el bus, el bit dominante equivale a un 0 lógico mientras que el bit recesivo equivale a un 1 lógico. Durante una colisión, momento en el que dos o más nodos intentan transmitir bits, el bit más dominante es el que se impone respecto al resto.

Esto da paso al campo de arbitraje en el mensaje, que se produce durante la lectura del identificador del mensaje. El arbitraje comparará los identificadores de los mensajes que colisionan e identifica el más dominante entre ellos, el que tenga menor valor.

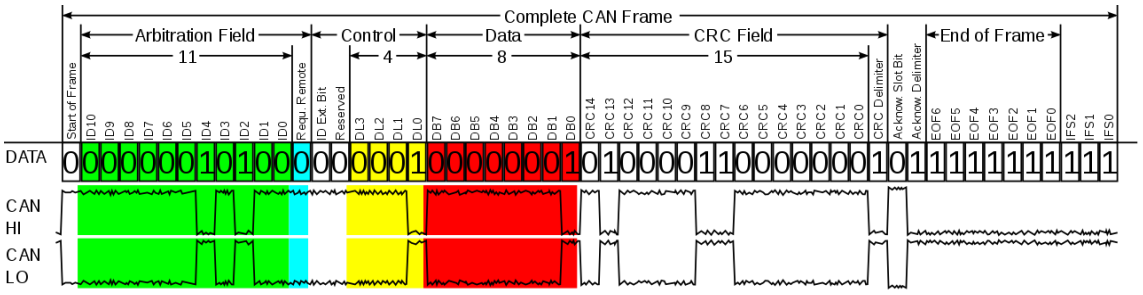

Fig. 2. Capa de datos del CAN. Fuente: [4].

La sincronización es especialmente importante en la fase de arbitraje, ya que durante ésta cada nodo debe ser capaz de observar tanto los datos transmitidos por él como los datos transmitidos por los demás nodos.

El controlador de bus CAN espera que una transición del bus de recesivo a dominante ocurra en un determinado intervalo de tiempo. Si la transición no ocurre en el intervalo esperado, el controlador reajusta la duración del siguiente bit en consecuencia. Dicho ajuste se lleva a cabo dividiendo cada bit en intervalos o *cuantos* de tiempo y asignando los intervalos a los cuatro segmentos de cada bit: sincronización, propagación, segmento de fase 1 y segmento de fase 2.

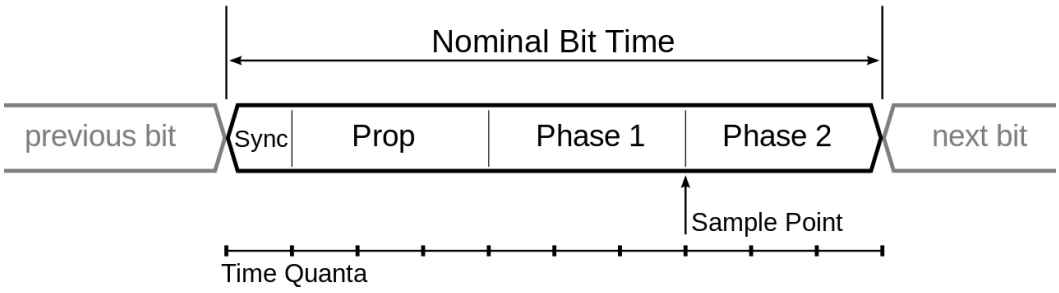

Fig. 3. Ejemplo de los tiempos de un bit en el CAN. Fuente: [5].

Para garantizar una correcta sincronización se añaden bits de relleno cada 5 bits consecutivos de la misma polaridad con polaridad opuesta, asegurando que haya suficientes transiciones entre bits recesivos y dominantes.

4.1.2. CAN FD

Como su predecesor, el CAN FD (Controller Area Network Flexible Data-Rate) es un protocolo de comunicación de equipos electrónicos. Fue desarrollado en 2011 y puesto en el mercado en 2012, debido la necesidad de modernizar el protocolo con transmisiones de datos en tiempo real más precisas.

El nuevo bus mejora la velocidad de transmisión de datos del CAN, la cual estaba limitada a 1 Mbit/s. El CAN FD también optimiza la capacidad de la carga, permitiendo transmitir hasta 64 bytes en vez de los 8 bytes de su predecesor. El aumento de capacidad se debe a que cuando solo hay un nodo transmitiendo, la tasa de bits aumenta. Sin embargo, después de la transmisión es necesario volver a sincronizar todos los nodos.

En lo que a datos se refiere, ambos buses utilizan el clásico identificador, el cual es de 11 o 29 bits. Por otro lado, en el control y en el campo CRC del CAN FD se usan más bits. Las principales diferencias son:

- El bit FDF, en el campo de control, indica si el siguiente bit de la secuencia debe interpretarse como un mensaje del CAN FD o del CAN.
- En el campo de control del CAN hay un bit RTR, mientras que el CAN FD utiliza el bit RRS. La función de este bit es prevenir que los mensajes con identificadores de 29 bits tengan menor prioridad que aquellos con 11 bit, ya que ambos pueden coexistir en el bus.
- El CAN FD cuenta con un bit para indicar el cambio de ratio de bits (BRS bit).

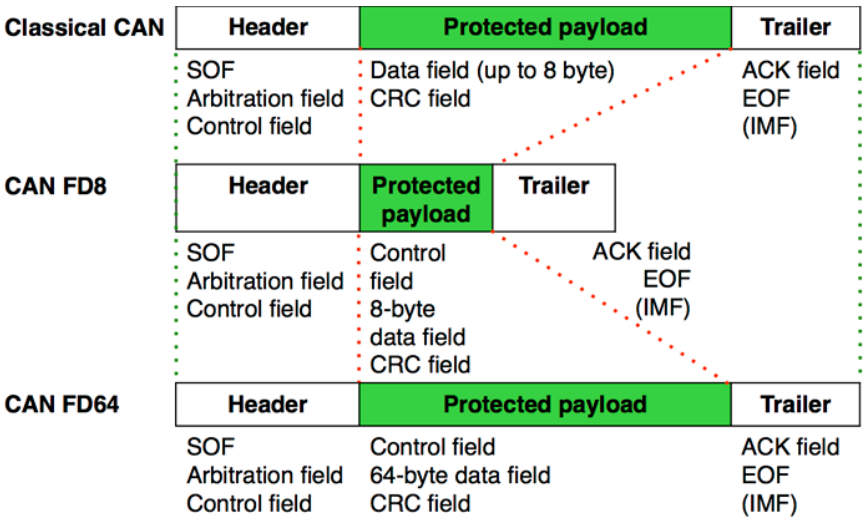

Fig. 4. Comparativa entre tramos de CAN y CAN FD. Fuente: [7].

Las principales mejoras que ofrece el CAN FD en comparación al CAN [8] son:

- El CAN FD cuenta con tramas más cortas, mejor rendimiento en tiempo real y mayor ancho de banda.
- Posibilidad de cambiar a una tasa más rápida de datos en la fase de datos. Cuentan con el mismo campo de arbitraje.
- Como se ha mencionado ya, el CAN FD permite enviar mensajes con más datos (de 8 bytes a 64 bytes). Con el aumento de datos, el software se simplifica, pero es necesario que con mensajes de alta carga aumente la tasa de bits para reducir el tamaño de los tramos y el tiempo que el tramo ocupa la línea de comunicación.
- El uso de más bits en el CRC reduce el riesgo de no detectar errores.

4.2. MCP2517FD CLICK

El MCP2517FD Click es un componente comercial compuesto por un controlador de CAN FD MCP2517FD, un transceptor de alta velocidad ATA6563 y un conector DB9 de 9 pines [9].

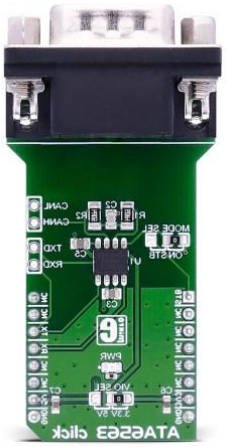

Fig. 5. MCO2517FD Click. Fuente: [9].

4.2.1. Controlador MCP2517FD

El MCP25FD es un controlador CAN FD de bajo coste y de tamaño ligero que se puede conectar a un microcontrolador a través de una interfaz SPI. Esto permite conectar un canal CAN FD a un microcontrolador pese a no tener periféricos para el CAN FD o suficientes canales del bus.

El controlador tiene capacidad para los datos tanto del formato clásico de CAN como del formato del CAN FD.

Los bloques que componen el MCP2517 son:

- Controlador CAN FD, implementa el protocolo CAN FD y contiene las FIFOs y los filtros.
- Interfaz SPI, controla el dispositivo accediendo a SFRs y la RAM.
- Controlador de RAM. Arbitra los accesos de la RAM entre el SPI y el módulo de control CAN FD.
- Mensajes RAM, se usa para archivar los datos de los mensajes.
- Oscilador, genera el reloj del sistema.
- Circuitos internos LDO y POR
- Controlador E/S (I/O).

4.2.2. Transceptor ATA6563

El ATA6563 [10] es un transceptor de alta velocidad que provee una interfaz entre el protocolo CAN y los cables trenzados del bus. Está diseñado para aplicaciones CAN de alta velocidad (de hasta 5 Mbit/s) en la industria de automoción, ofreciendo capacidades diferenciales de transmisión y recepción.

Cuenta con capacidades electromagnéticas mejoradas (EMC en inglés) y descarga electrostática (ESD) y tiene un comportamiento ideal en el CAN cuando se reduce el voltaje de alimentación.

Dicho transceptor usa un conector estándar DB9 de 9 pines.

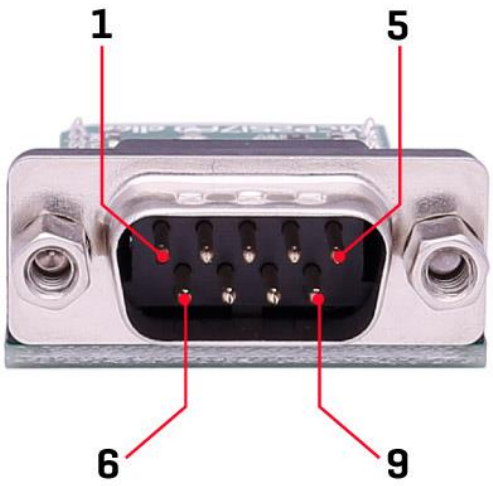

Fig. 6. Conector DB9. Fuente: [9].

4.2.3. Interfaz SPI

El bus SPI (Serial Peripheral Interface) fue desarrollado por Motorola en 1980 [11]. Sus ventajas respecto a otros sistemas han hecho que se convierta en un estándar en el mundo de la electrónica y automatización.

El bus SPI tiene una arquitectura de tipo maestro-esclavo**.** La comunicación de datos entre maestros y esclavo se realiza en dos líneas independientes, una del maestro a los esclavos,

y otra de los esclavos al maestro. Por tanto, la comunicación puede ser Full Duplex, es decir, el maestro puede enviar y recibir datos simultáneamente.

Otra característica de SPI es que es un bus síncrono. El dispositivo maestro proporciona una señal de reloj, que mantiene a todos los dispositivos sincronizados. Esto reduce la complejidad del sistema frente a los sistemas asíncronos.

Por lo que el SPI requiere de al menos 3 líneas:

- **MOSI** (*Master-out, slave-in*) para la comunicación del maestro al esclavo.
- **MISO** (*Master-in, slave-out*) para comunicación del esclavo al maestro.
- **SCK** (*Clock*) señal de reloj enviada por el maestro.

También cuenta con una línea adicional SS (*Slave Select*) por cada esclavo conectado que indica a qué esclavo va referido la información desde el maestro.

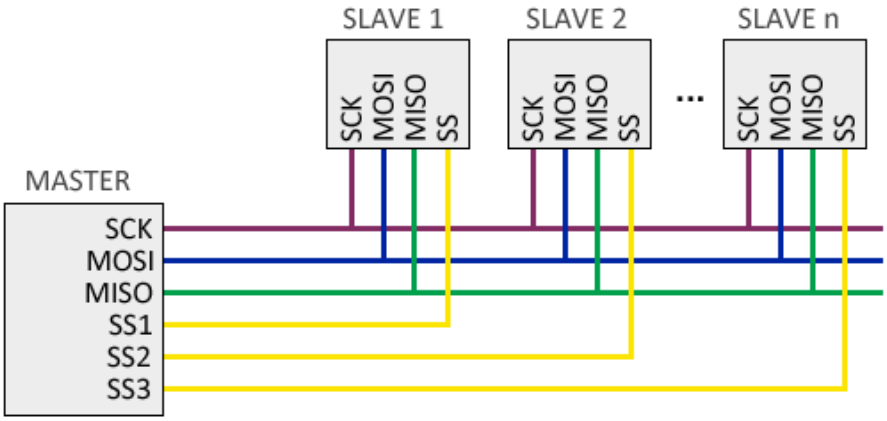

Fig. 7. Esquema de SPI con 1 maestro y 3 esclavos. Fuente: [11].

El problema de este diseño es que, si hay un número considerable de esclavos, se requieren demasiadas líneas. Existe una versión que conecta en cascada a los esclavos, por lo que solo es necesario una conexión SS, aunque esta versión tiene una respuesta más lenta, ya que la información debe llegar a todos los esclavos a la vez.

Por defecto el maestro mantiene en estado alto todas las líneas SS, lo que indica que el maestro no se comunica con ningún esclavo. Si el maestro quiere establecer comunicación con algún esclavo, cambiará el estado de su línea SS a estado bajo.

La trama no sigue ninguna regla, es decir, se puede enviar cualquier secuencia arbitraria de bits. Esto hace que los dispositivos conectados necesiten tener pre acordado la longitud y significado de los que van a enviar y recibir.

4.2.4. Pines usados

Solo se usan los siguientes pines del MCP2517FD:

Pin	Descripción
GND	Tierra
5 V	Suministro positivo
3V3	Suministro positivo
SDI	SPI data input
SDO	SPI data output
SCK	Spi clock input
nCS	SPI chip select input

Tabla 1. Tabla con lo pines usados del MCP2517FD Click. Fuente: propia.

4.3. Raspberry Pi

La Raspberry Pi es un ordenador de placa simple y bastante económica. Fue desarrollada en Reino Unido por la Raspberry Pi Fundation [12]. Pese a poder a tener un software de código abierto y permitiendo instalar cualquier sistema operativo, cuenta con uno oficial, el cual es una versión adaptada de Debian llamada Raspbian.

La flexibilidad y accesibilidad de este componente es lo que ha incentivado su elección para este proyecto, ya que desarrollar un *driver* de CAN FD en esta plataforma es dar acceso a operar con el bus a un público muy extenso. Cualquiera que sea poseedor de una Raspberry Pi podrá utilizar el CAN FD con ella.

Fig. 8. Imagen de una Raspberry PI. Fuente: *[13]*.

4.3.1. GPIO

Para este proyecto se utilizará una Raspberry Pi 3 Modelo B, puesta a la venta en el año 2016. Esta versión renueva procesador con el BCM2835, con un Quad-Core de 1.20GHz. Mantiene la RAM en 1GB. Su mayor novedad es la inclusión de Wi-Fi y Bluetooth (4.1 Low Energy) sin necesidad de adaptadores.

Dejando de lado el instrumental que nos permite trabajar con la RPI, monitor, teclado, ratón, etc., el punto interesante para el proyecto son los pines (GPIO) ya que se pueden usar como pines de entrada o pines de salida, lo que nos permitirá conectarnos al MCP2517FD Click para acceder al CAN FD.

Para poder controlar los pines con la Raspberry Pi se utiliza la librería de C *pigpio* [14]. La librería cuenta con funciones para el control de hardware. Dichas funciones devuelven un valor numérico a modo de control, en el caso de error la respuesta es inferior a 0. Se puede acceder a cada pin del GIPIO con el identificador numérico correspondiente a su posición.

Los pines que se usan para el proyecto son:

Pin	Descripción
Pin 39: Masa	Tierra
Pin 2: Alimentación 5V	Suministro positivo
Pin 1: Alimentación 3V3	Suministro positivo
Pin 19: SPI0 MOSI	SPI data input
Pin 21: SPI0 MISO	SPI data output
Pin 23: SPI0 SCLK	Spi clock input
Pin 11: SPI1 CE1	SPI chip select input

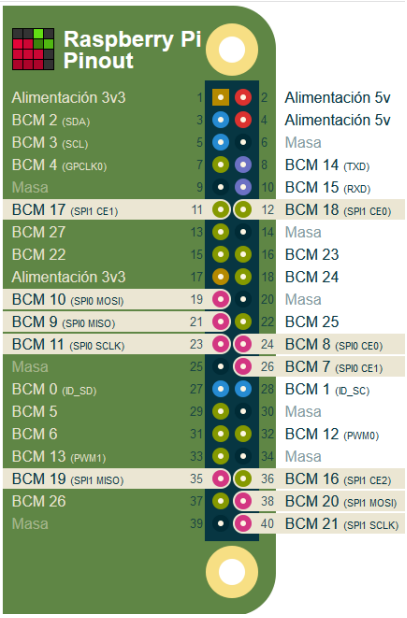

Fig. 9 y tabla 2. Descripción y uso de los pines GPIO de la Raspberry Pi. Fuente: [15].

4.3.2. Raspbian

También conocido como Raspberry Pi OS, Raspbian es una distribución del sistema operativo GNU/Linux basado en Debian y por lo tanto, libre para la SBC Raspberry Pi, orientado a la enseñanza de informática. El lanzamiento inicial fue en junio de 2012. Desde 2015, la Raspberry Pi Foundation lo ha proporcionado de forma oficial como el sistema operativo primario para la familia de placas SBC de Raspberry Pi. Hay varias versiones de Raspbian, siendo la actual Raspbian Buster [16].

Destaca también el menú "raspi-config", que permite configurar el sistema operativo sin tener que modificar archivos de configuración manualmente. Entre sus funciones, permite expandir la partición *root* para que ocupe toda la tarjeta de memoria, configurar el teclado, aplicar *overclock*, etc.

Al ser una distribución de GNU/Linux las posibilidades son infinitas. Todo software de código abierto puede ser recompilado en la propia Raspberry Pi para arquitectura ARMF, de manera que pueda ser utilizado en el propio dispositivo en caso de que el desarrollador no proporcione una versión ya compilada para esta arquitectura. De esta manera, el propio sistema operativo ayuda a modificar el controlador del CAN FD en la plataforma.

4.4. Entorno desarrollo integrado

Un entorno de desarrollo integrado (IDE) es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo de software. Normalmente, un IDE consiste de un editor de código fuente, herramientas de construcción automáticas y un depurador.

Dependiendo de cómo se instale el sistema operativo en la Raspberry Pi, puede ya tener instalados varios IDEs. Para el proyecto se optó por usar Geany IDE. El motivo se debe a que es un editor de texto de poco espacio basado en Scintilla, un componente de edición de código con características que facilitan la programación. A parte, únicamente requiere de bibliotecas GTK2 para su ejecución, se utiliza para el soporte grafico del IDE y acostumbra a ser, junto con Qt, a biblioteca de interfaces gráficas de usuario más popular en Linux. Por último, Geany admite cientos de lenguajes de programación entre los que evidentemente está el lenguaje C, lenguaje escogido para el proyecto.

4.5. Sistema

Una vez comprendidos los componentes, tanto de hardware como de software, que forman el proyecto, se va a explicar la arquitectura del sistema.

Como eje central del sistema está la Raspberry Pi, ya que es la plataforma a la que se conecta el controlador de CAN FD y la que permite desarrollar el controlador del bus, es decir, es el soporte del software y la conexión entre software y hardware. Se requiere conectar a la Raspberry Pi un monitor, un teclado y un ratón para posibilitar sus funciones como ordenador. Adicionalmente, se puede conectar un cable de Ethernet para facilitar el acceso a internet. También es en este dispositivo donde se instalará el sistema operativo Raspbian y la IDE Geany.

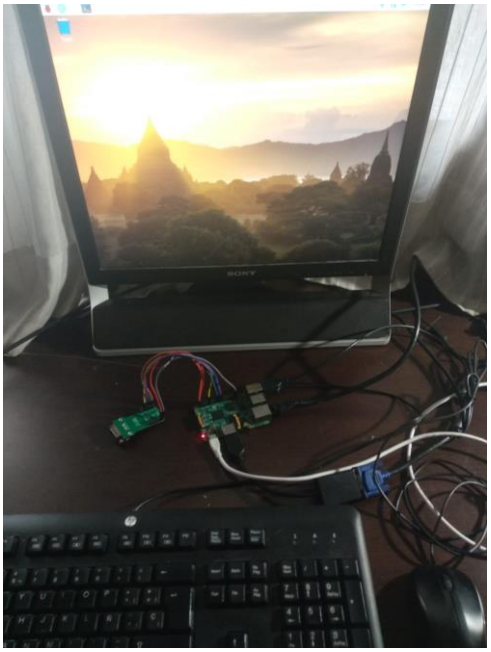

Fig. 10. Raspberry Pi conectada al monitor, teclado y ratón. Fuente: propia.

En lo que a hardware respecta, se conecta el MCP2517FD Click a la Raspberry Pi usando los pines GPIO mencionados en el apartado 4.3 y los pines del SPI mencionados en el apartado 4.2.

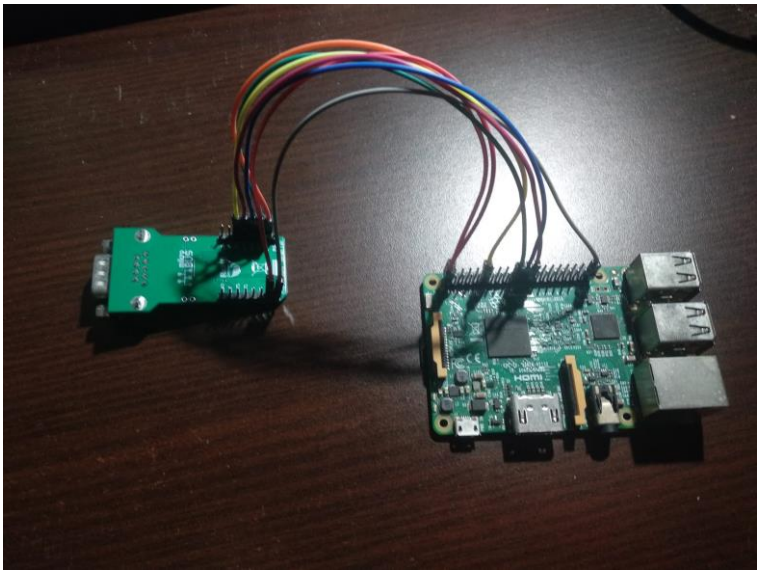

Fig. 11. Raspberry Pi conectada al MCP2517FD Click. Fuente: propia.

Finalmente, se debe conectar el CAN FD al MCP2517FD Click, aunque no es estrictamente necesario para el desarrollo del controlador y el registrador de datos, por lo que se ha decidido prescindir de él durante la fase de desarrollo para facilitar su movilidad.

5. Desarrollo

5.1. Pasos previos

Tras comprender cómo funciona el sistema, se ha de proceder al desarrollo del controlador. Para ello, son necesarios unos pasos previos antes de iniciar la programación del software del controlador de CAN FD.

Una vez conectados todos los componentes, es necesario configurar la Raspberry Pi para que se pueda usar. El primer paso es instalar el sistema operativo *Raspbian* [17]. Para ello se utiliza una imagen que la propia página web de Raspberry Pi ofrece. El procedimiento se resume en descargarse dicha imagen en una tarjeta SD, con un peso aproximado de 16 GB. Posteriormente, introducir la tarjeta en la Raspberry Pi y, tras una configuración, el sistema operativo es totalmente funcional. Antes de explicar la configuración necesaria hay que entender que la imagen usada para instalar *Raspbian* es una recopilación comprimida de archivos y carpetas de referencia utilizados para instalar y configurar un nuevo sistema operativo en un equipo capturados en un archivo de imagen. El proceso de instalar un sistema operativo a través de una imagen se utiliza bastante en la industria.

Para configurar y actualizar el sistema operativo es necesario tener la Raspberry Pi conectada a internet. La configuración se puede realizar a través de la ventana gráfica que ofrece el propio *Raspbian* o desde la consola de comandos. Para el proyecto se ha realizado la configuración desde la consola. Con tres sencillos comandos se puede tener el sistema correctamente actualizado [18]:

`sudo apt update`: Para actualizar la lista de paquetes instalados.

`sudo apt full-upgrade`: Para descargar las últimas versiones de los paquetes.

`sudo apt clean`: Para liberar espacio ocupado durante la actualización.

Una vez el sistema operativo está instalado es necesario instalar, o al menos actualizar, el IDE que se pretenda usar, en el caso de este proyecto Geany IDE. Cabe destacar que el sistema operativo cuenta con varios IDEs instalados, pero siempre es recomendable actualizarlos. Desde la misma terminal se puede instalar Geany [25], conectados a internet, con el comando: `sudo apt-get install geany`.

5.2. Desarrollo del controlador

Para el desarrollo del controlador se ha trabajado sobre dos *drivers* ya existentes, el primero ofrecido por la empresa Microchip Inc, *MCP2517FD canfdspi API* para *PIC32MX470* [19], y el desarrollado por el estudiante del GETI Oriol Garrobé en su trabajo de fin de grado [20]. Esto se debe a que el controlador desarrollado por Oriol Garrobé era una versión más

optimizada y ligera respecto al *driver* original, Sin embargo, su trabajo se centraba en desarrollar el controlador para un microcontrolador de la familia PIC18, lo cual tenía ciertas limitaciones al trabajar con una CPU de 8 bits. Con una Raspberry Pi no existe dicha limitación y es por eso por lo que se ha completado el proyecto usando los dos controladores.

5.2.1. Archivos necesarios

Como se ha mencionado anteriormente, el *driver* será una modificación de otros dos controladores de CAN FD. Por lo tanto, es necesario importar los archivos que éstos utilizan:

Archivos Cabecera	Archivos Fuente
drv_canfdspi_api.h	drv_canfdspi_api.c
drv_canfdspi_defines.h	drv_spi.c
drv_canfdspi_register.h	main.c
drv_canfdspi_spi.h	

Tabla 3. Archivos necesarios. Fuente: [19].

Debido a herencias que utiliza el código del *MCP2517FD canfdspi API* para *PIC32MX470* se han de eliminar las referencias a archivos que no se utilizan, con tal de eliminar posibles errores de compilación.

5.2.2. Validación librería pigpio

Dado que los códigos originales de los drivers que se usan están diseñados para operar en procesadores de 32 bits para el original y de 8 bits para el de Oriol Garrobé, y que en este proyecto se usa una Raspberry Pi, es necesario modificar el driver de SPI para que utilice la librería *pigpio* [14], que se encarga de controlar los pines de la RPI.

Debido a la multitud de pines que hay en los puertos GPIO de la Raspberry Pi, hay que seleccionar qué pin queremos usar para transmitir. El pin escogido es el pin número 17 de la librería *pigpio*, que se corresponde con el pin número 11 en el hardware, CE01. A partir de aquí, se asignarán los valores 0 ó 1 al pin seleccionado (el valor 1 implica que se está accediendo al SPI). Para modificar el valor del pin del se usa la función *gpioWrite,* en la que se le introduce como parámetro de entrada el índice del pin sobre el que se desea escribir y el valor que se le quiere dar, 0 ó 1.

Para inicializar el SPI se requiere la función *spiOpen,* además de indicar el pin que se pretende usar para acceder al SPI y ponerlo en valor alto. Para la transmisión de datos se utilizan las funciones *spiWrite* y *spiRead*.

Se ha realizado una prueba para comprobar el comportamiento de la librería en cuestión.

Dicha prueba consiste en un único programa ejecutable con una función muy simple: iniciar la configuración de pines y, dada una dirección y un mensaje, escribirlo en la memoria SPI y leerlo de esa misma dirección, usando las funciones *spiWrite* y *spiRead* para cada respectiva función en un bucle infinito.

Main del Test

```
1.  int main(void) {
2.      unsigned int address, k;
3.      char spiTransmitBuffer[2];
4.      char spiReceiveBuffer[4];
5.
6.      int h;
7.      if (gpioInitialise() < 0) {
8.          return 1;
9.      }
10.
11.     gpioSetMode(CS_PIN, PI_OUTPUT);
12.     gpioWrite(CS_PIN, 1);
13.
14.     h = spiOpen(0, 2000000, 0);
15.     if (h < 0) {
16.         return 2;
17.     }
18.
19.     address = 0xE00;
20.     spiTransmitBuffer[0] = ( char) ((cINSTRUCTION_READ << 4) + ((address >> 8)
    & 0xF));
21.     spiTransmitBuffer[1] = ( char) (address & 0xFF);
22.
23.     while(1) {
24.
25.         gpioWrite (CS_PIN, 0); //CS = 0
26.         spiWrite(h, spiTransmitBuffer, 2);
27.         spiRead(h, spiReceiveBuffer,4);
28.         gpioWrite(CS_PIN, 1); // CS  = 1
29.         printf("Read: ");
30.
31.         for(k = 0; k < 4; k++) {
32.         printf("%x", spiReceiveBuffer[k] );
33.
34.         }
35.         printf("\n");
36.
37.     }
38.
39.     spiClose(h); // CS = 1
40.
41. }
```

Resultados

```
Read: 60400
Read: 60400
Read: 60400
Read: 60400
Read: 60400
Read: 60400
```

```
Read: 60400
Read: 60400
Read: 60400
Read: 60400
Read: 60400
Read: 60400
```

Tabla 4. Test de librería pigpio y los resultados extraídos de la consola. Fuente: propia.

Al estar leyendo el 60400 se valida que la librería *pigpio* se puede utilizar para transmitir datos entre el SPI y la Raspberry Pi y, por lo tanto, se puede modificar el *driver* SPI para que se conecte a los pines de la RPI.

5.2.2.1. Driver SPI

El *driver* SPI está basado en el desarrollado por Oriol Garrobé, ya que es más simple y, por lo tanto, está más optimizado. Las diferencias principales entre ambos son el cómo se accede a la memoria RAM del SPI, dado a que en su proyecto solo existe un pin periférico SPI, debido al diseño del microprocesador que se usó, por lo que no necesita definir cuál es y utiliza la función *SPI*_CS para modificar el valor de dicho pin y así poder acceder al SPI. Sin embargo, en este proyecto es necesario definir qué pin se utilizará como periférico SPI y se requieren las funciones de la librería *pigpio* para acceder a éste y, por ende, a la memoria RAM.

La mayor diferencia se encuentra en la función *DRV_SPI_Initialize,* en la que, con una sentencia de código, el *driver* de Oriol Garrobé se inicializa, mientras que, el controlador de este proyecto requiere seleccionar el pin SPI y cambiarle el estado a nivel alto para poder acceder a éste. En las otras dos funciones del *driver*, *DRV_SPI_ChipSelectAssert* y *DRV_SPI_TransferData* las modificaciones son mínimas, cambiando en la primera las funciones *SPI_CS* por *gpioWrite* y en la segunda las funciones *WriteSPI* y *ReadSPI* por *spiWrite* y *spiRead*.

Código Original de Oriol Garrobé	Código modificado

`void DRV_SPI_Initialize(void)`
`{`
 	
`    OpenSPI(SPI_FOSC_4, MODE_00, SMPMID);`
 	
`    return;`
`}`
	`int8_t DRV_SPI_Initialize(void)`
	`{`
	
	`    if (gpioInitialise() < 0)`
	`    {`
	`        return 1;`
	`    }`
	
	`    gpioSetMode(CS_PIN, PI_OUTPUT);`
	`    gpioWrite(CS_PIN, 1); // CS = 1`
	`}`

```
13.
14.     h = spiOpen(0, 2000000, 0);
15.     if (h < 0)
16.     {
17.       return 2;
18.     }
19.     return 0;
20. }
```

Tabla 5. Modificaciones en el Driver SPI. Fuente: propia.

Para poder validar si el *driver* funciona correctamente, es necesario añadir instrucciones SPI para empezar a trabajar con el controlador CAN FD y comprobar si éste es capaz de transmitir información haciendo uso del *driver* SPI.

5.2.3. Instrucciones SPI

Las instrucciones SPI son las que permiten acceder a los SFRs (*Special Function Registers*) y a la RAM (*Random Access Memory*). Los SFRs se usan para controlar y leer el estado del controlador CAN FD. La RAM es donde se archivan los datos de los mensajes. El desarrollo del controlador se inicia con este bloque.

Varias funciones de transferencia de datos se requieren para poder escribir o leer a través del SPI. Son las funciones que requieren modificaciones para adaptarlas a las librerías de la Raspberry Pi. Las instrucciones SPI llamarán a las funciones *spiWrite* y *spiRead* para la transferencia de datos.

Para acceder a los SFRs se necesita trabajar en modo configuración. Por otro lado, la RAM está orientada a palabras. Esto implica enviar o recibir 4 bytes de información a la vez. Por lo tanto, cualquier múltiplo de 4 puede ser leído o escrito.

Cada una de las instrucciones SPI utilizan el mismo formato y estructura. La instrucción empieza definiendo las variables necesarias. Posteriormente, se escribe en el pin CS (*Chip Select*) un nivel bajo, es decir se cambia su valor de 1 a 0. El comando compuesto por los 4 primeros bits y la dirección, compuesta por los 12 restantes, se desplazan a la SDI. Durante la instrucción de escritura los bits se desplazan a la SDI en el flanco de subida del SCK. En cambio, en la instrucción de lectura, se desplazan fuera de la SDO en el flanco de bajada.

Por último, es necesario volver a cambiar el estado del pin CS al final de cada instrucción al estado alto. Se han modificado las funciones del *driver* original para cumplir con este requerimiento.

Las instrucciones SPI que se utilizan en el proyecto son:

- *DRV_CANFDSPI_ReadByte*
- *DRV_CANFDSPI_WriteByte*
- *DRV_CANFDSPI_ReadWord*

- *DRV_CANFDSPI_WriteWord*
- *DRV_CANFDSPI_ReadHalfWord*
- *DRV_CANFDSPI_WriteHalfWord*
- *DRV_CANFDSPI_ReadByteArray*
- *DRV_CANFDSPI_WriteByteArray*
- *DRV_CANFDSPI_ReadWordArray*
- *DRV_CANFDSPI_WriteWordArray*

La instrucción *DRV_CANFDSPI_WriteWordArray* no es imprescindible, pero se utiliza para hacer pruebas en el código.

5.2.3.1. Driver CAN FD SPI API

Las instrucciones SPI del *driver* CAN FD siguen basándose en el controlador desarrollado por Oriol Garrobé. Ya se ha mencionado anteriormente que dicho proyecto tenía problemas con el procesador de 8 bit que usaba, lo que limitaba la cantidad de información que podía transmitir instantáneamente, provocando el tener que desarrollar el *driver* separando y transmitiendo la información. Este proyecto no tiene esa limitación, dado a que trabajamos sobre una Raspberry Pi, por lo que no es necesario hacer modificaciones en todas las instrucciones SPI, presentadas en el apartado anterior. También, como pasaba en el *driver* SPI, se modifican las llamadas de acceso al SPI para que utilicen la librería *pigpio.*

A continuación, se presenta un ejemplo de las modificaciones realizadas en las instrucciones SPI, concretamente la instrucción *DRV_CANFDSPI_ReadWord.*

Código Modificado

```
1.  int8_t DRV_CANFDSPI_ReadWord(CANFDSPI_MODULE_ID index, uint16_t address, uint3
    2_t *rxd)
2.  {
3.
4.      uint8_t i;
5.      uint32_t x;
6.      int8_t spiTransferError = 0;
7.
8.
9.      gpioWrite(CS_PIN, 0); //CS = 0
10.
11.     // Compose command to WRITE
12.     spiTransmitBuffer[0] = (char) ((cINSTRUCTION_READ << 4) + ((address >> 8)
    & 0xF));
13.     spiTransmitBuffer[1] = (char) (address & 0xFF);
14.     spiWrite(h, spiTransmitBuffer, 2);
15.
16.     // Update data READ
17.     *rxd = 0;
18.     for (i = 2; i < 6; i++)
19.     {
20.         spiRead(h, spiReceiveBuffer, 1);
21.         x = (uint32_t) (spiReceiveBuffer[0]);
22.         *rxd += x << ((i - 2)*8);
```

```
    }

    gpioWrite(CS_PIN, 1); // CS = 1

    return spiTransferError;
}
```

Código de Oriol Garrobé

```
int8_t DRV_CANFDSPI_ReadWord(CANFDSPI_ODULE_ID index, uint16_t address, uint32_t *rxd)
{
    uint8_t i;
    uint32_t x;
    uint16_t spiTransferSize = 6; //Modified 16/04/2019
    int8_t spiTransferError = 0;
    SPI_CS = 0;

    // Compose command
    spiTransmitBuffer[0] = (uint8_t) ((cINSTRUCTION_READ << 4) + ((address >> 8) & 0xF));
    spiTransmitBuffer[1] = (uint8_t) (address & 0xFF);


    WriteSPI(spiTransmitBuffer[0]);
    WriteSPI(spiTransmitBuffer[1]);
// Update data
    *rxd = 0;
    for (i = 2; i < 6; i++) {
        x = (uint32_t) ReadSPI();
        *rxd += x << ((i - 2)*8);
}
/*
    spiTransferError = DRV_SPI_TransferData(index, spiTransmitBuffer, spiReceiveBuffer, spiTransferSize);
    if (spiTransferError) {
        return spiTransferError;
    }

    // Update data
    *rxd = 0;
    for (i = 2; i < 6; i++) {
        x = (uint32_t) spiReceiveBuffer[i];
        *rxd += x << ((i - 2)*8);
    }
*/
    SPI_CS = 1;
    return spiTransferError;
}
```

Tabla 6. Modificaciones realizadas desde el código del controlador de Oriol Garrobé. Fuente: propia.

Una vez todas las instrucciones SPI son modificadas para adaptarlas a este proyecto, se añadirán el resto de las funciones que conforman el *driver* CAN FD. Dichas funciones únicamente requieren de las instrucciones mencionadas anteriormente, por lo que se pueden reutilizar las funciones del controlador original *MCP2517FD canfdspi API* para *PIC32MX470* [19]. El *driver* desarrollado por Oriol Garrobé tiene modificaciones en varias de las funciones por sus problemas de espacio. Las funciones que se presentan a continuación son las que se encargan de la configuración del dispositivo y de realizar la transmisión de

información.

Las instrucciones restantes por añadir, que se reutilizaran del controlador original son:

- *DRV_CANFDSPI_Reset*
- *DRV_CANFDSPI_EccEnable*
- *DRV_CANFDSPI_RamInit*
- *DRV_CANFDSPI_ConfigureObjectReset*
- *DRV_CANFDSPI_Configure*
- *DRV_CANFDSPI_TransmitChannelConfigureObjectReset*
- *DRV_CANFDSPI_TransmitChannelConfigure*
- *DRV_CANFDSPI_ReceiveChannelConfigureObjectReset*
- *DRV_CANFDSPI_ReceiveChannelConfigure*
- *DRV_CANFDSPI_BitTimeConfigure*
- *DRV_CANFDSPI_OperationModeSelect*
- *DRV_CANFDSPI_BitTimeConfigureNominal40MHz*
- *DRV_CANFDSPI_BitTimeConfigureData40MHz*
- *DRV_CANFDSPI_BitTimeConfigureNominal20MHz*
- *DRV_CANFDSPI_BitTimeConfigureData20MHz*
- *DRV_CANFDSPI_BitTimeConfigureNominal10MHz*
- *DRV_CANFDSPI_BitTimeConfigureData10MHz*
- *DRV_CANFDSPI_TransmitChannelEventGet*
- *DRV_CANFDSPI_TransmitChannelLoad*
- *DRV_CANFDSPI_TransmitChannelUpdate*
- *DRV_CANFDSPI_DlctoDataBytes*
- *DRV_CANFDSPI_ReceiveChannelEventGet*
- *DRV_CANFDSPI_ReceiveMessageGet*
- *DRV_CANFDSPI_ReceiveChannelUpdate*
- *DRV_CANFDSPI_FilterObjectConfigure*
- *DRV_CANFDSPI_FilterDisable*
- *DRV_CANFDSPI_FilterToFifoLink*
- *DRV_CANFDSPI_FilterMaskConfigure*
- *DRV_CANFDSPI_FilterEnable*

Por otro lado, el proyecto de Oriol Garrobé tenía problemas de espacio en la memoria, por lo que eliminó varias funciones de la API que no se utilizaban ahorrando espacio. Dichas funciones sí están en el *driver* original. Sin embargo, para optimizar el controlador desarrollado no se añadirán, siendo ésta una de las ventajas de basar el driver en el proyecto de Oriol Garrobé.

5.2.3.2. Validación de las instrucciones

Para validar las instrucciones SPI se han realizado diversos test de manera iterativa. Las pruebas consisten en ir elaborando un programa ejecutable *main* en el que se controlen mensajes que contengan cada uno de los distintos formatos en los que se puede presentar los datos, bytes, palabras, listas de bytes y listas de palabra. El mensaje se debe escribir en una dirección y posteriormente leer en la misma dirección. Si coincide el mensaje enviado con el leído se considera que dicho par de instrucciones son funcionales y correctas, pues pueden transmitir información entre el SPI y la Raspberry Pi.

La última iteración es un conjunto de 6 pruebas diferentes, que consisten en:

- **TEST1**: leer el registro especial SFR denominado OSCON.
- **TEST2:** escribir y leer la semipalabra (2 bytes) 0x5A5A en la dirección 0x400.
- **TEST3:** escribir y leer la palabra 0x11443322 en la dirección 0x010.
- **TEST4:** escribir y leer el byte 0x98 en la dirección 0x010.
- **TEST5:** escribir y leer un array de bytes en la dirección 0x400.
- **TEST6:** escribir y leer un array de palabras en la dirección 0x400.

Tests de las instrucciones SPI

```
1.        #ifdef TEST1  // Read OSCON Register (address 0xE00)
2.      address = 0xE00;
3.      DRV_CANFDSPI_ReadWord(0, address, &value);
4.      printf("Read Register OSCON (0xE00): ");
5.      printf("0x%08x\n",value);
6.  #endif
7.
8.  #ifdef TEST2  // WriteHalfWord / ReadHalfWord test
9.      address = 0x010;
10.     DRV_CANFDSPI_WriteHalfWord(0, address, 0x5A5A);
11.     printf("HalfWord written in address 0x010: 0x5A5A\n");
12.     DRV_CANFDSPI_ReadHalfWord(0, address, &value);
13.     printf("Read HalfWord:  ");
14.     printf("0x%04x\n",value);
15. #endif
16.
17. #ifdef TEST3  // WriteWord / ReadWord test
18.     address = 0x400;
19.     DRV_CANFDSPI_WriteWord(0, address, 0x11443322);
20.     printf("Word written in address 0x400: 0x11443322\n");
21.     DRV_CANFDSPI_ReadWord(0, address, &value);
22.     printf("Read Word:  ");
23.     printf("0x%08x\n",value);
24. #endif
25.
26.
27. #ifdef TEST4 // WriteByte / ReadByte test
28.     address = 0x010;
29.     DRV_CANFDSPI_WriteByte(0, address, 0x98);
30.     printf("Byte written in address 0x400: 0x98\n");
31.     DRV_CANFDSPI_ReadByte(0, address, &value2);
32.     printf("Read Byte:  ");
33.     printf("0x%x\n",value2);
```

```
34.     printf("\n");
35. #endif
36.
37. #ifdef TEST5 // WriteByteArray / ReadByteArray test
38.     for (i=0; i<64; i++)
39.     {
40.         ip[i] = 63-i;   // 63...0, en HEX -> 0x3F...0x00
41.         iprx[i] = 0;
42.     }
43.     address = 0x400;
44.     DRV_CANFDSPI_WriteByteArray(0, address, ip, 64);
45.     printf("Byte Array written in address 0x400: 0x3F...0x00\n");
46.     DRV_CANFDSPI_ReadByteArray(0, address, iprx, 64);
47.     printf("Read Array:\n");
48.     for (i=0; i<64; i++) printf("0x%x, ",iprx[i]);
49.     printf("\n");
50. #endif
51.
52. #ifdef TEST6 // WriteByteArray / ReadByteArray test
53.     for (i=0; i<12; i++)
54.     {
55.         ip[i] = 63 - i;   // 63...0, en HEX -> 0x3F...0x00
56.         iprw[i] = 0;
57.     }
58.
59.     address = 0x400;
60.     DRV_CANFDSPI_WriteWordArray(0, address, ip, 64);
61.     printf("Word Array written in address 0x400: 0x3F...0x00\n");
62.     DRV_CANFDSPI_ReadWordArray(0, address, iprw, 64);
63.     printf("Read word Array:\n");
64.     for (i=0; i<64; i++) printf("0x%08x, ",iprw[i]);
65.     printf("\n");
66. #endif
```

Tabla 7. Tests a las instrucciones SPI. Fuente: propia.

Los resultados son los siguientes:

```
Read Register OSCON (0xE00): 0x00000460
HalfWord written in address 0x010: 0x5A5A
Read HalfWord:  0x5a5a
Word written in address 0x400: 0x11443322
Read Word:  0x11443322
Byte written in address 0x400: 0x98
Read Byte:  0x98

Byte Array written in address 0x400: 0x3F...0x00
Read Array:
0x3f, 0x3e, 0x3d, 0x3c, 0x3b, 0x3a, 0x39, 0x38, 0x37, 0x36, 0x35, 0x34, 0x33, 0x
32, 0x31, 0x30, 0x2f, 0x2e, 0x2d, 0x2c, 0x2b, 0x2a, 0x29, 0x28, 0x27, 0x26, 0x25
, 0x24, 0x23, 0x22, 0x21, 0x20, 0x1f, 0x1e, 0x1d, 0x1c, 0x1b, 0x1a, 0x19, 0x18,
0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10, 0xf, 0xe, 0xd, 0xc, 0xb, 0xa, 0x
9, 0x8, 0x7, 0x6, 0x5, 0x4, 0x3, 0x2, 0x1, 0x0,
Word Array written in address 0x400: 0x3F...0x00
Read word Array:
0x0000003f, 0x0000003e, 0x0000003d, 0x0000003c, 0x0000003b, 0x0000003a, 0x000000
39, 0x00000038, 0x00000037, 0x00000036, 0x00000035, 0x00000034, 0x00000033, 0x00
000032, 0x00000031, 0x00000030, 0x0000002f, 0x0000002e, 0x0000002d, 0x0000002c,
0x0000002b, 0x0000002a, 0x00000029, 0x00000028, 0x00000027, 0x00000026, 0x000000
25, 0x00000024, 0x00000023, 0x00000022, 0x00000021, 0x00000020, 0x0000001f, 0x00
00001e, 0x0000001d, 0x0000001c, 0x0000001b, 0x0000001a, 0x00000019, 0x00000018,
0x00000017, 0x00000016, 0x00000015, 0x00000014, 0x00000013, 0x00000012, 0x000000
11, 0x00000010, 0x0000000f, 0x0000000e, 0x0000000d, 0x0000000c, 0x0000000b, 0x00
00000a, 0x00000009, 0x00000008, 0x00000007, 0x00000006, 0x00000005, 0x00000004,
0x00000003, 0x00000002, 0x00000001, 0x00000000,
```

Fig. 12. Resultados de los Test de las instrucciones SPI. Fuente: propia.

En otras circunstancias estos test podrían haber sido más completos, al añadir un osciloscopio desde el que observar la salida de pulsos eléctricos y tener así una validación tanto por software como por hardware. Lamentablemente, no se ha podido disponer de dicha herramienta durante el desarrollo del proyecto.

5.2.4. Filtros y máscaras

En el CAN FD existe la posibilidad de recibir cientos de mensajes de distintos identificadores. Sin embargo, puede que solo interesen los mensajes de un grupo reducido de identificadores. Este filtro se puede aplicar mediante el software, pero para ahorrar tiempo de operación computacional el propio hardware incluye filtros y máscaras que pueden realizar la misma tarea.

La función de la máscara consiste en seleccionar los bits que se pretendan comparar con el filtro. El filtro es el que se encarga de aceptar o ignorar el mensaje, comparando su propio valor con el de los bits no enmascarados del mensaje.

La máscara permite seleccionar que bits del mensaje de CAN FD se compararan con el filtro, los bits del mensaje que se correspondan a bits con valor 0 en la máscara serán aceptados inmediatamente, mientras que los que se correspondan a un valor de 1 se comparan con el filtro.

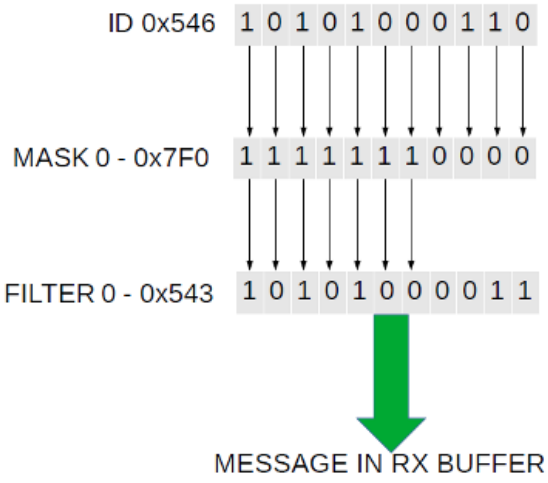

Fig. 13. Ejemplo de un mensaje aceptado por el filtro y la máscara. Fuente: [21].

En el *driver* se han desarrollado instrucciones que permiten configurar los filtros y las máscaras, ya que es una función imprescindible para un nodo de CAN FD poder seleccionar qué mensajes se deben recibir y cuales ignorar.

Los registros especiales que se utilizan para la configuración de filtros y máscaras son el *REG_CiFLTOBJ,* para los filtros y el *REG_CiMASK,* para las máscaras. Ambos son registros que contienen una palabra de 32 bits únicamente, es decir información de 4 bytes. Dicho

valor es el que se utiliza para seleccionar qué identificadores son ignorados y cuáles no.

REG_CiFLTOBJ	REG_CiMASK

`typedef union _REG_CiFLTOBJ {`
`  CAN_FILTEROBJ_ID bF;`
`  uint32_t word;`
`  uint8_t byte[4];`
`} REG_CiFLTOBJ;`
	`typedef union _REG_CiMASK {`
	`  CAN_MASKOBJ_ID bF;`
	`  uint32_t word;`
	`  uint8_t byte[4];`
	`} REG_CiMASK;`

Tabla 8. Objetos definidos para los Filtros y las Máscaras. Fuente: propia.

5.2.5. Función main

La función principal *main* sirve como punto de partida para la ejecución del programa. Normalmente, controla la ejecución del programa dirigiendo las llamadas a otras funciones del programa. Un programa deja de ejecutarse normalmente al final del *main*, aunque puede finalizar en otros puntos del programa por distintos motivos.

Se han definidos dos funciones principales *mains* diferentes en el transcurso del proyecto. La primera versión se codificó en una temprana fase del desarrollo del *driver* CAN FD para hacer pruebas de las instrucciones SPI. La segunda se diseñó como una simulación de un entorno más realista de transferencia de información para poner a prueba un controlador ya definido en su totalidad y es también donde se definirá el registrador de datos.

5.2.5.1. Configuración de la función main

La segunda versión función *main* es bastante más compleja que la primera versión y requiere de la mayoría de las instrucciones definidas en el controlador. Además, necesita una configuración para poder operar correctamente Por ello, el sistema debe trabajar en modo Configuración, ya que únicamente se puede acceder a ciertas de las instrucciones en ese modo, tales como el reset o la configuración de los canales FIFO. En este modo el sistema no puede acceder al CAN. Es necesario entonces que:

- El MCP2517FD se resetee sus valores a los iniciales, previniendo así un mal funcionamiento del sistema al utilizar un valor usada anteriormente.
- La memoria RAM debe iniciarse.
- Indicar si se usa el modo ISO CRC
- Preparar los canales FIFO indicando el número de canales necesarios y el tamaño de la *payload* de los mensajes que contendrá.
- Activar, si se necesitan, las máscaras y filtros. Debe enlazarse al canal FIFO donde se aplicará el filtro.
- Se debe habilitar el *Bit Time* para poder usar los dos valores de *bit-rate,* el NBR (*Nominal Bit Rate*), usado durante el arbitraje, y el DRB (Data Bit Rate), usado

durante la transferencia de datos.

- Finalmente, escoger el modo de operación, puede ser modo *Normal*, donde el dispositivo puede transferir mensajes con el modo CAN FD, con una tasa de datos de hasta 64 bytes. También puede usar el modo *LoopBack,* una variación del modo *Normal,* que permite la transmisión interna de mensajes con los canales FIFO de envío y recepción de datos.

Configuración de la función main

```
1.      gpioWrite(CS_PIN, 1); // CS = 1
2.     //CONFIG_SPI_CS = 0; // output
3.     DRV_SPI_Initialize();
4.
5.     DRV_CANFDSPI_Reset(0);
6.
7.     // Enable ECC and initialize RAM
8.     DRV_CANFDSPI_EccEnable(DRV_CANFDSPI_INDEX_0);
9.
10.    if (!ramInitialized) {
11.         DRV_CANFDSPI_RamInit(DRV_CANFDSPI_INDEX_0, 0x00); //0xff);
12.     //DRV_CANFDSPI_RamInit2(DRV_CANFDSPI_INDEX_0, 0xFF); //0xff);
13.         ramInitialized = true;
14.     }
15.
16.     // Configure device
17.     DRV_CANFDSPI_ConfigureObjectReset(&config);
18.     config.IsoCrcEnable = ISO_CRC;
19.     config.StoreInTEF = 0;
20.     config.TXQEnable = 0;
21.     DRV_CANFDSPI_Configure(0, &config);
22.
23.         // Test 1 TXFIFO
24.     DRV_CANFDSPI_TransmitChannelConfigureObjectReset(&txConfig);
25.     txConfig.FifoSize = 1;//7;
26.     txConfig.PayLoadSize = CAN_PLSIZE_64;
27.     txConfig.TxPriority = 0;
28.
29.     DRV_CANFDSPI_TransmitChannelConfigure(0, APP_TX_FIFO, &txConfig);
30.
31.         // Test 1 RXFIFO
32.     DRV_CANFDSPI_ReceiveChannelConfigureObjectReset(&rxConfig);
33.     rxConfig.FifoSize = 1;
34.     rxConfig.PayLoadSize = CAN_PLSIZE_64;
35.     rxConfig.RxTimeStampEnable = 1;
36.     DRV_CANFDSPI_ReceiveChannelConfigure(DRV_CANFDSPI_INDEX_0, APP_RX_FIFO, &r
    xConfig);
37.
38.     // Setup RX Filter
39.     fObj.word = 0x0;
40.     fObj.bF.SID = 0x01;
41.     fObj.bF.EXIDE = 0; //0;
42.     fObj.bF.EID = 0x00;
43.
44.     DRV_CANFDSPI_FilterObjectConfigure(DRV_CANFDSPI_INDEX_0, CAN_FILTER0, &fOb
    j.bF);
45.
46.     // Setup RX Mask
47.     mObj.word = 0;
48.     mObj.bF.MSID = 0x01;
```

```
49.     mObj.bF.MIDE = 1; // Only allow standard IDs
50.     mObj.bF.MEID = 0x0;
51.     DRV_CANFDSPI_FilterMaskConfigure(DRV_CANFDSPI_INDEX_0, CAN_FILTER0, &mObj.
    bF);
52.
53.     // Link FIFO and Filter
54.     DRV_CANFDSPI_FilterToFifoLink(DRV_CANFDSPI_INDEX_0, CAN_FILTER0, APP_RX_FI
    FO, true);
55.
56.     // Setup Bit Time
57.     DRV_CANFDSPI_BitTimeConfigure(DRV_CANFDSPI_INDEX_0, selectedBitTime, CAN_S
    SP_MODE_AUTO, CAN_SYSCLK_40M);
58.
59.     DRV_CANFDSPI_OperationModeSelect(DRV_CANFDSPI_INDEX_0, CAN_EXTERNAL_LOOPBA
    CK_MODE);
```

Tabla 9. Ejemplo de una configuración de la función main. Fuente: propia.

Nótese que la configuración mostrada hace uso de máscaras y filtros para leer únicamente mensajes con un identificador impar.

5.2.5.2. Configuración de los mensajes

Se ha mencionado que el *main* también se pretende usar como una simulación de casos de transferencia de datos, definiendo un mensaje, enviándolo y recibiéndolo. Este hecho requiere entonces configurar dicho mensaje también. Para ello, se han definido dos objetos para usarse para configurar la transferencia de información, txObj (objeto para transmitir información) y rxObj (objeto para recibir información). Existen 3 parámetros que definen los mensajes, todos ellos mencionados en la explicación del tramo de datos de CAN FD:

- **txObj.bF.id.SID**: es el identificador estándar, pese a ser con el que se trabaja también existe el txObj.bF.id.EID para la versión extendida.
- **txObj.bF.ctrl.DLC:** Es el tamaño, en bytes, de la carga del mensaje.
- **tx[**bytes**]:** La carga del mensaje, es un vector que contiene los bytes de datos del mensaje.

Existen otros parámetros que configuran el mensaje, como por ejemplo los bits de control. Sin embargo, no se modificarán en ninguna prueba.

Configuración de mensaje

```
1.  // Configure transmit message
2.  txObj.word[0] = 0;
3.  txObj.word[1] = 0;
4.
5.  txObj.bF.id.SID = 0x400;
6.  //FILE_START_ID;
7.  txObj.bF.id.EID = 0;
8.  txObj.bF.ctrl.BRS = 1;
9.  txObj.bF.ctrl.DLC = CAN_DLC_64;
10. txObj.bF.ctrl.FDF = 1;
11. txObj.bF.ctrl.IDE = 0;
12. txObj.bF.ctrl.SEQ = 0;
```

```
13.
14.  // Configure message data
15.  for (i = 0; i < txObj.bF.ctrl.DLC; i++)
16.  {
17.    txd[i] = rand() & 0xff;
18.  }
```

Tabla 10. Ejemplo de una configuración del mensaje de transmisión. Fuente: propia.

La configuración presenta un mensaje con identificador estándar 0x400, un DLC de 64 bytes y una *payload* de valores aleatorios.

Para facilitar hacer pruebas se ha incluido un ciclo infinito en la función *main,* que se encarga de generar mensajes y transferirlos.

5.2.6. Validación de controlador de CAN FD

Tras el desarrollo del código, se necesita validar que este cumple con su función, es decir, que el controlador sea capaz de transferir mensajes con la estructura de los tramos de CAN FD. Para ello, se ha desarrollado una segunda versión de la función *main,* que permite simular una situación de transferencia de datos en el tiempo.

El método usado para considerar como validado el *driver* no dista del utilizado hasta la fecha. Se definen mensajes de CANFD que deben ser enviados y posteriormente se leen. Si ambos mensajes coinciden, se da por hecho que el funcionamiento es adecuado. Es evidente que un único caso no es suficiente para validar si el controlador es funcional, por lo que se han definido 3 pruebas diferentes.

Las pruebas consisten en un primer test, donde se estudia un caso particular de un único mensaje con un único identificador. Se trabaja con un caso simple para facilitar el reconocimiento de errores, en caso de que los haya. Las dos siguientes pruebas consisten en ir añadiendo factores de aleatoriedad a las variables del test para volver el caso más complejo, tratando de ser más fidedigno a un caso real.

Por consiguiente, los test se presentan en la función *main*, con toda la configuración necesaria ya dispuesta, y se trabaja con el valor de tres parámetros:

- El identificador del mensaje.
- La *payload* del mensaje.
- El DLC (Data Length Code) de la *payload.*

En ninguno de estos casos se trabaja ni con filtros ni con máscaras.

Para la primera prueba, se pretende analizar un caso concreto en el que todos los parámetros son constantes. Los parámetros son:

- Identificador: 0xde.

- Carga: un contador ascendente que vaya de 0 al 63, en hexadecimal, que es como los datos se presentan, del 0x0 al 0x3F.
- DLC: 64.

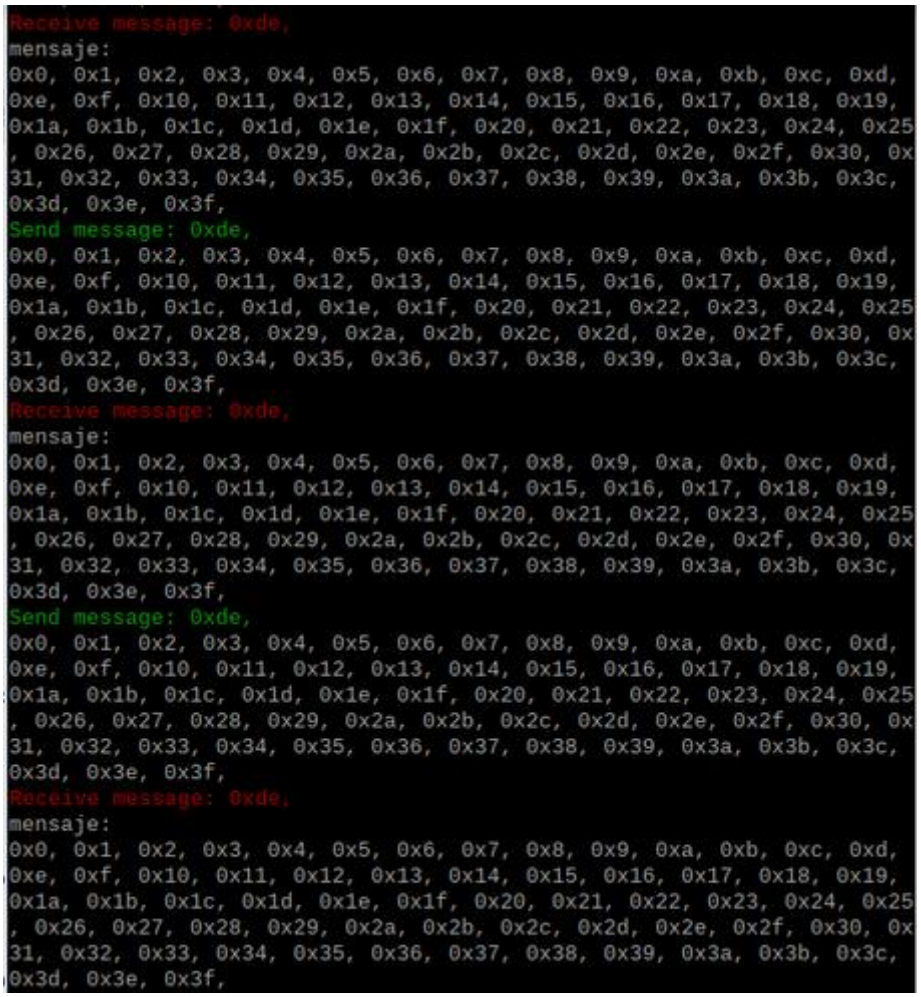

Fig. 14. Resultado del primer test al controlador CAN FD. Fuente: propia.

Tras ciertas correcciones en el código, la prueba muestra el patrón mostrado en la *Fig. 14.* Al observarse que el mensaje enviado coincide con el recibido, se considera que el primer test se puede concluir con éxito.

Para la segunda prueba, se añaden componentes de aleatoriedad en la carga y en la longitud de ésta, es decir el DLC. Los parámetros usados son:

- Identificador: 0xde.
- Carga: números aleatorios, en hexadecimal, con valores comprimidos entre 0x0 y 0xFF (que es el 255 en base decimal).
- DLC: números aleatorios entre sus 16 posibles valores.

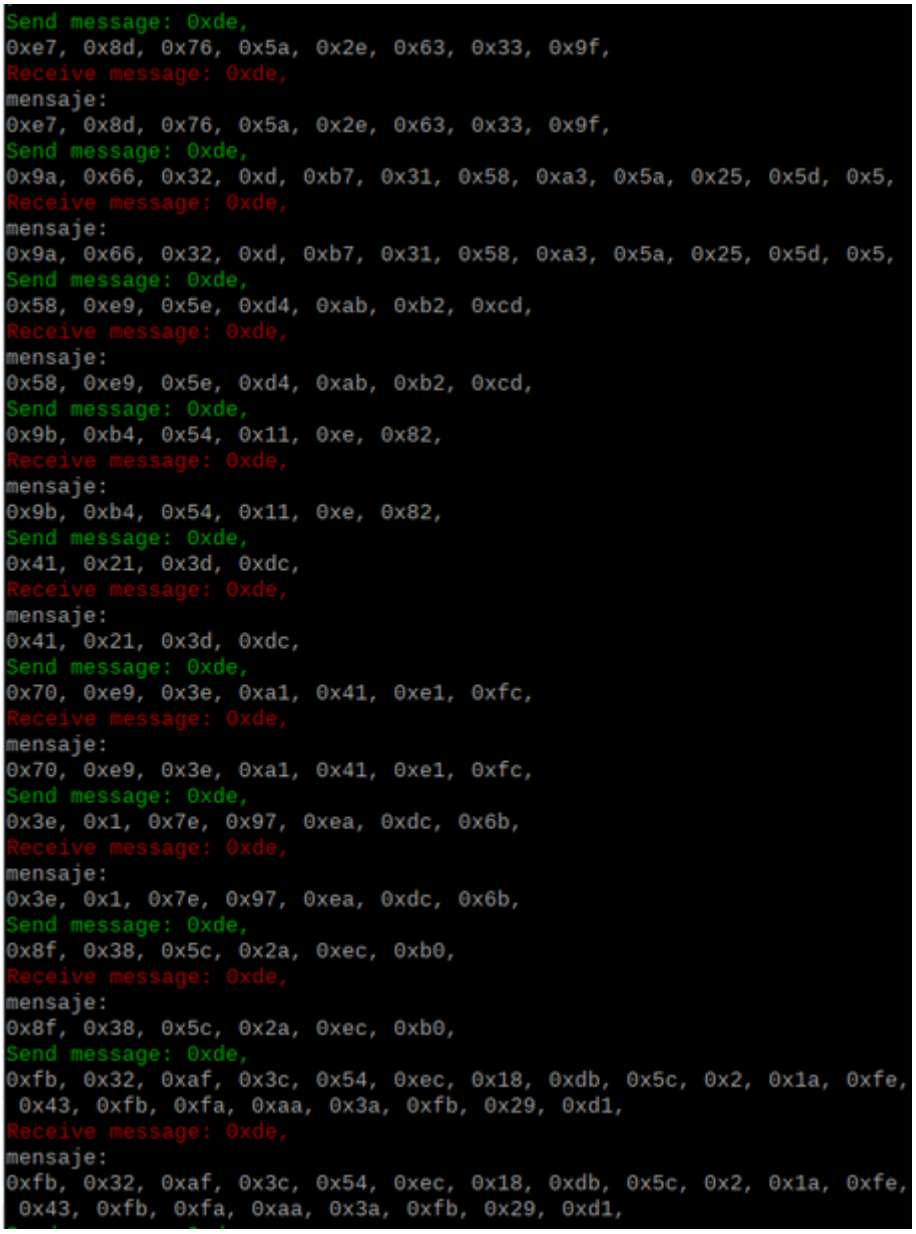

Fig. 15. Resultado del segundo test al controlador CAN FD. Fuente: propia.

Como era de esperar, los mensajes coinciden entre los transmitidos y los recibidos, por lo que se considera que se puede avanzar a la tercera prueba.

Para la tercera prueba, se aleatorizan todas las variables con las que trabaja el test, identificador, carga y DLC. Los parámetros son entonces,

- Identificador: números aleatorios, en hexadecimal, con valores comprimidos entre 0x0 y 0x7FF.
- Carga: números aleatorios, en hexadecimal, con valores comprimidos entre 0x0 y 0xFF (que es el 255 en base decimal).
- DLC: números aleatorios entre sus 16 posibles valores.

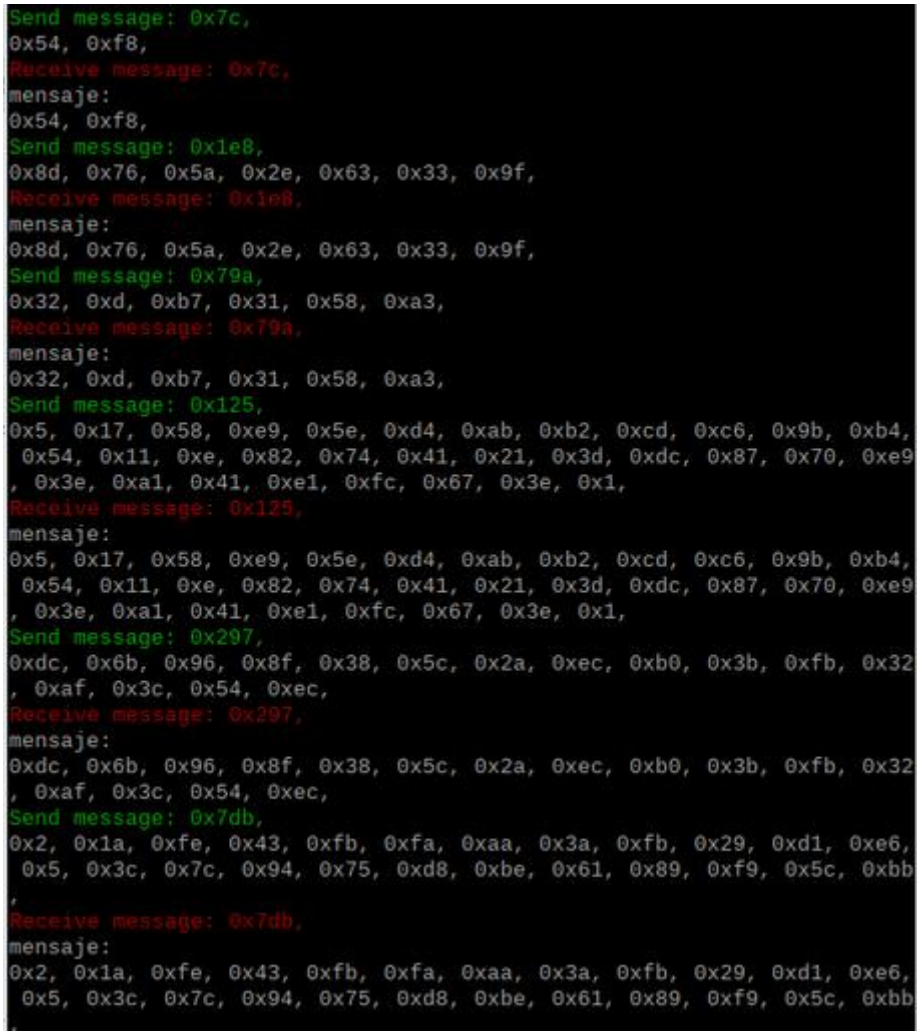

Fig. 16. Resultado del tercer test al controlador CAN FD. Fuente: propia.

Tras dejar al programa ejecutarse durante 30 minutos, no se detecta ningún error, por lo que se puede considerar que el *driver* cumple correctamente la función de enviar y recibir mensajes de CANFD en modo *Loopback*. Por lo tanto, queda validado el controlador, lo que implica que el CAN FD puede operar en la Raspberry Pi dando paso al desarrollo del *datalogger.*

5.3. Registrador de datos

5.3.1. Presentación de los datos

El controlador de CAN FD utiliza dos objetos para la transferencia de datos, uno para transmitir datos, *CAN_TX_MSGOBJ*, y otro para recibir datos, *CAN_RX_MSGOBJ*, ambos con una estructura bastante similar.

Dichos objetos guardan en un objeto propio los bits del identificador, los bits de control y el *timeStamp,* como medida temporal*,* reservando los primeros bits para estos atributos.

El objeto *CAN_MSGOBJ_ID* sirve para definir el identificador y reserva los 11 bits que ocupa la versión estándar, aunque también incluye 18 bits más para alcanzar los 29 bits que ocupa la versión extendida.

El objeto *CAN_TX_MSGOBJ_CTRL* sirve para definir los bits de control de los mensajes transmitidos, reservando 4 bits para el DLC, que mide la longitud de la carga, 1 bit para el IDE, el RTR y el BRS, también ocupa un bit el FDF y el ESI. Por último, sin contar los bits no implementados, reserva 5 bits para el FilterHit. El *CAN_RX_MSGOBJ_CTRL* sirve para definir los bits de control de los mensajes recibidos y tiene la misma estructura.

Finalmente, el objeto *CAN_MSG_TIMESTAMP,* que no es más que un entero (*int*) de 32 bits, sirve para disponer del tiempo UNIX como medida temporal.

CAN_MSGOBJ_ID	CAN_TX_MSGOBJ_CTRL	CAN_MSG_TIMESTAMP

`typedef struct _CAN_MSGOBJ_ID {`
`  uint32_t SID : 11;`
`  uint32_t EID : 18;`
`  uint32_t SID11 : 1;`
`  uint32_t unimplemented1 : 2;`
`} CAN_MSGOBJ_ID;`
	`typedef struct _CAN_TX_MSGOBJ_CTRL {`
	`  uint32_t DLC : 4;`
	`  uint32_t IDE : 1;`
	`  uint32_t RTR : 1;`
	`  uint32_t BRS : 1;`
	`  uint32_t FDF : 1;`
	`  uint32_t ESI : 1;`
	`  uint32_t unimplemented1 : 2;`
	`  uint32_t FilterHit : 5;`
	`  uint32_t unimplemented2 : 16;`
	`} CAN_TX_MSGOBJ_CTRL;`
		`typedef uint32_t CAN_MSG_TIMESTAMP;`

Tabla 11. Objetos propios de los objetos que definen los mensajes. Fuente: propia.

Los objetos de los mensajes contienen, además de los anteriormente mencionados, un contenedor de bytes para reservar la carga de los mensajes, a los cuales se puede acceder como palabras o bytes.

A continuación, se muestran cómo están definidos los objetos que definen los mensajes de envío y recepción de CAN FD según el driver.

CAN_TX_MSGOBJ

```
typedef union _CAN_TX_MSGOBJ {

    struct {
        CAN_MSGOBJ_ID id;
        CAN_TX_MSGOBJ_CTRL ctrl;
        CAN_MSG_TIMESTAMP timeStamp
;
    } bF;
    uint32_t word[3];
    uint8_t byte[12];
} CAN_TX_MSGOBJ;
```

CAN_RX_MSGOBJ

```
typedef union _CAN_RX_MSGOBJ {

    struct {
        CAN_MSGOBJ_ID id;
        CAN_RX_MSGOBJ_CTRL ctrl;
        CAN_MSG_TIMESTAMP timeStamp
;
    } bF;
    uint32_t word[3];
    uint8_t byte[12];
} CAN_RX_MSGOBJ;
```

Tabla 12. Objetos definidos del *driver* para la transmisión de datos. Fuente: propia.

Dentro de los bits de control están los 4 bits que definen el DLC, Data Length Code, que representa la longitud de los datos en bytes. Vale la pena recordar que una de las ventajas del CAN FD respecto al CAN es el poder enviar mensajes de hasta 64 bytes, mientras que el bus CAN está limitado a 8 bytes. En el controlador, la longitud de los mensajes puede variar en 16 posibles valores.

Data Length Code

```
typedef enum {
    CAN_DLC_0,
    CAN_DLC_1,
    CAN_DLC_2,
    CAN_DLC_3,
    CAN_DLC_4,
    CAN_DLC_5,
    CAN_DLC_6,
    CAN_DLC_7,
    CAN_DLC_8,
    CAN_DLC_12,
    CAN_DLC_16,
    CAN_DLC_20,
    CAN_DLC_24,
    CAN_DLC_32,
    CAN_DLC_48,
    CAN_DLC_64
} CAN_DLC;
```

Tabla 13. Posibles valores de DLC de los mensajes. Fuente: propia.

Los datos del mensaje, al igual que los identificadores, se denotan en base hexadecimal.

El *timeStamp* es una medida temporal que se presenta en milisegundos y está basada en el tiempo universal coordinado (UTC) [22]. Sin embargo, si la Raspberry Pi no cuenta con conexión a internet, la marca de tiempo se mantiene en la última marca registrada sin poder actualizarse al tiempo real Unix. Cabe añadir que, pese a no ofrecer una medida de tiempo universal, el *timeStamp* sirve como medida temporal relativa entre los mensajes, ya sean enviados o recibidos.

5.3.2. Registrador de datos

Para el *datalogger* se ha decidido presentar la información en archivos de texto plano tipo .txt. En él se incluye el *timeStamp* en milisegundos, el identificador en codificación hexadecimal, el DLC en codificación decimal y, por último, los datos del mensaje en codificación hexadecimal.

Se han diseñado dos registradores de datos, aunque con el mismo formato, uno dedicado a registrar la información de los mensajes enviados y el otro dedicado a registrar los mensajes recibidos. Esta división simplifica la codificación y resulta fácil para analizar por separado cada uno de los métodos de transferencia de datos. Cabe añadir que en una situación real existe una comunicación entre varios dispositivos al usar el bus, por lo que, tener un registro para cada uno de los dispositivos separado entre los mensajes enviados y los recibidos hace más entendible la información.

El formato que se ha decidido para el *datalogger* es:

Formato
timeStamp, identificador, DLC, mensaje
Ejemplo. *Datalogger* de enviar mensaje
1588291095, 0x1d0, 12, 73 51 ff 4a ec 29 cd ba ab f2 fb e3 1588291096, 0x80, 2, 54 f8 1588291097, 0x1f0, 7, 8d 76 5a 2e 63 33 9f 1588291098, 0x1e0, 6, 32 d b7 31 58 a3 1588291099, 0x171, 32, 5 17 58 e9 5e d4 ab b2 cd c6 9b b4 54 11 e 82 74 41 21 3d dc 87 70 e9 3e a1 41 e1 fc 67 3e 1 1588291100, 0xe1, 16, dc 6b 96 8f 38 5c 2a ec b0 3b fb 32 af 3c 54 ec 1588291101, 0x1e1, 24, 2 1a fe 43 fb fa aa 3a fb 29 d1 e6 5 3c 7c 94 75 d8 be 61 89 f9 5c bb 1588291102, 0xe1, 64, 95 b1 eb f1 b3 5 ef f7 0 e9 a1 3a e5 ca b cb d0 48 47 64 bd 1f 23 1e a8 1c 7b 64 c5 14 73 5a c5 5e 4b 79 63 3b 70 64 24 11 9e 9 dc aa d4 ac f2 1b 10 af 3b 33 cd e3 50 48 47 15 5c bb 6f 22 1588291103, 0x100, 20, 7d f5 b e1 1a 1c 7f 23 f8 29 f8 a4 1b 13 b5 ca 4e e8 98 32

```
1588291104, 0xf0, 12, 4d 3d 34 bc 5f 4e 77 fa cb 6c 5 ac
1588291105, 0x71, 20, aa 1a 55 a2 be 70 b5 73 3b 4 5c d3 36 94 b3 af
e2 f0 e4 9e
1588291106, 0x180, 5, 49 fd 82 4e a9
1588291107, 0x80, 4, b2 8a 29 54
1588291108, 0xe0, 16, bc d5 e 18 a8 44 ac 5b f3 8e 4c d7 2d 9b 9 42
1588291109, 0x50, 4, 33 af cd a3
1588291110, 0x81, 32, ad d4 76 47 de 32 1c ec 4a c4 30 f6 20 23 85
6c fb b2 7 4 f4 ec b b9 20 ba 86 c3 3e 5 f1 ec
1588291111, 0x71, 3, b7 99 50
1588291112, 0xf1, 4, d3 d9 34 f7
1588291113, 0x1f0, 2, 10 a8
```

Tabla 14. Formato registrador de datos. Fuente: propia.

Para generar el registrador es necesario crear un archivo en el código y "printarlos" con la información que nos interesa. El *timeStamp* se puede obtener haciendo una llamada a la función de la biblioteca *time.h* y "casteando" el resultado como un valor entero *(int)time(NULL).* El identificador y el DLC se extraen del objeto de transferencia sobre el que se están registrando valores. Por último, se recorre la carga del mensaje y se registran cada uno de sus valores.

A continuación, se enseña un ejemplo de código de registrador de datos, concretamente el que registra los datos transmitidos.

Código del datalogger

```
1.
2.  FILE *pFile1;
3.  pFile1 = fopen("sendMessage.txt", "a");
4.  fprintf(pFile1, "%d, 0x%x, %d,", timestamp, txObj.bF.id, DRV_CANFDSPI_DlcToDat
    aBytes(txObj.bF.ctrl.DLC));
5.  for(i=0;i<txObj.bF.ctrl.DLC;i++)
6.  {
7.     fprintf(pFile1," %x",txd[i]);
8.  }
9.  fprintf(pFile1,"\n");
10. fclose(pFile1);
```

Tabla 15. Código de datalogger. Fuente: propia.

6. Pruebas y Resultados con el Datalogger

6.1. Registrador sin máscaras ni filtros

Dado que no en todas las aplicaciones del CAN FD es estrictamente necesario filtrar los mensajes, se ha tomado la decisión de presentar dos registradores de datos, uno sin configuración ni uso de filtros ni máscaras y otro, en el que sí están presentes.

Los registradores o *dataloggers* utilizan el formato ya explicado. Se crea un archivo por cada método de transferencia de información que hay, es decir, enviar y recibir datos. El archivo generado para el envío de datos se denomina “sendMessage.txt” y el archivo generado para la recepción de datos se denomina “receiveMessage.txt”. Debido a la falta de filtros y que la comunicación de *driver* solo se está probando consigo mismo, ambos archivos tendrán el mismo contenido en condiciones normales, es decir, si no hay errores de ningún tipo.

Es necesario tener configurado el *driver* y definir los mensajes en la función *main* para hacer funcionar el programa y registrar datos.

A continuación, se prueba, ejecutando el programa, la generación de ambos archivos.

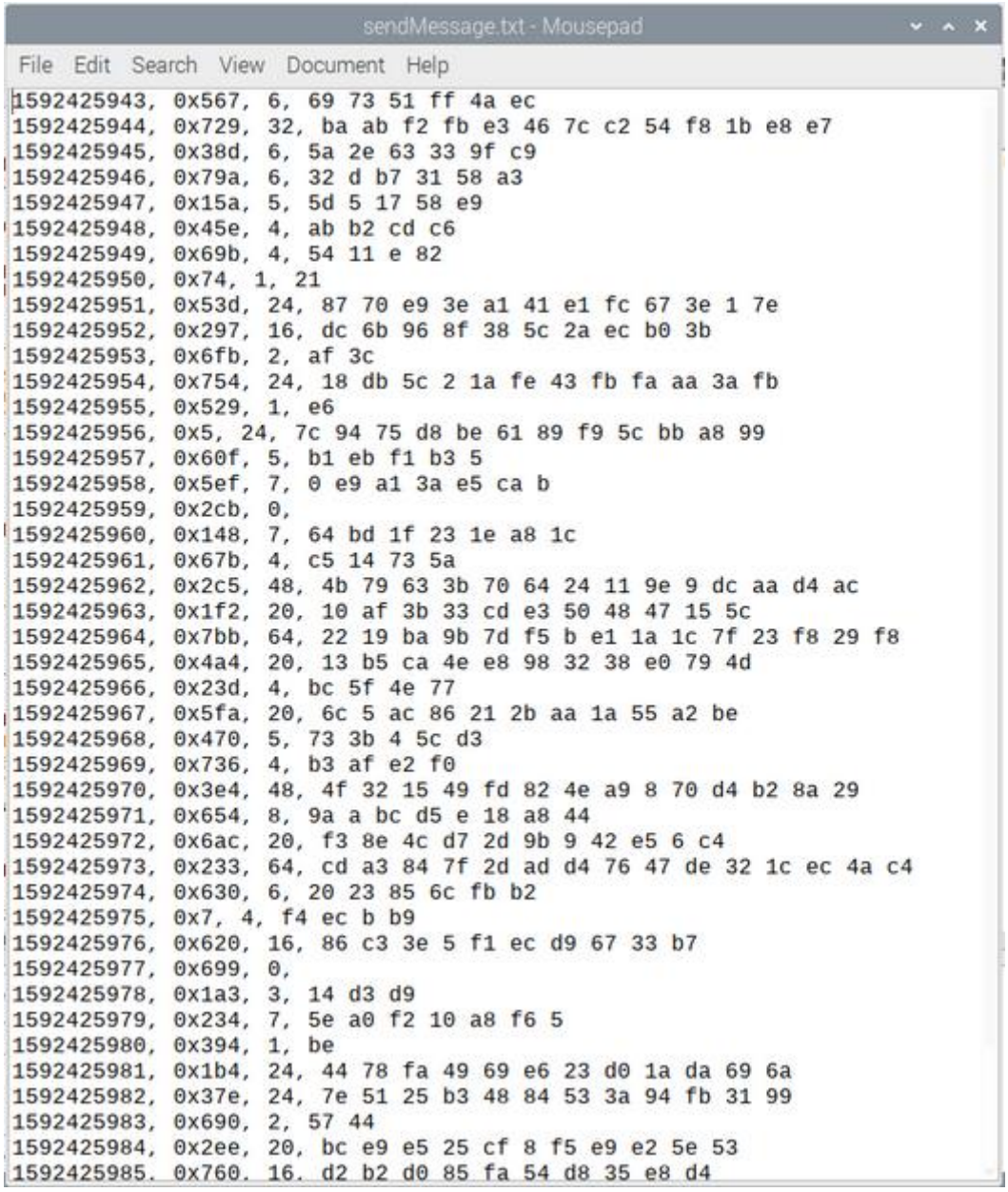

Fig. 17. Archivo generado para registro de datos enviados “sendMessage.txt”. Fuente: propia.

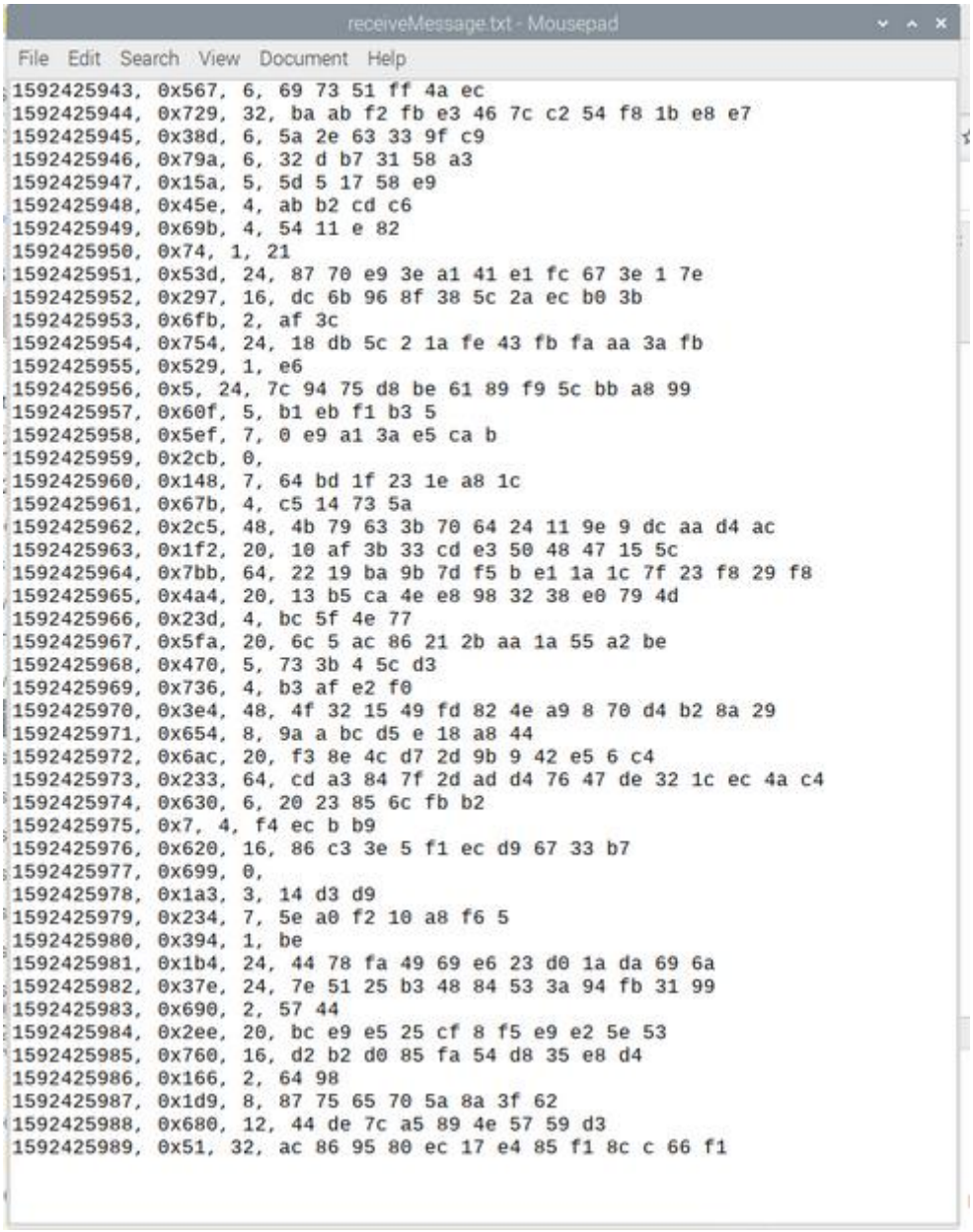

Fig. 18. Archivo generado para registro de datos recibidos “receiveMessage.txt”. Fuente: propia.

Como era de esperar, ambos archivos coinciden con la información registrada, tanto el tiempo en que se realizó la transferencia, el identificador del mensaje y la carga de éste (el segundo se ve con más datos porque la imagen se capturó desde una perspectiva más amplia). Nótese que el *payload* contiene la misma cantidad de bytes indicada en el DLC, cosa que reafirma la validación del *driver.*

6.2. Registrador usando máscaras y filtros

En esta versión del registrador, los mensajes de CANFD se pueden filtrar. Por lo tanto, se van a realizar diversas pruebas para poner a prueba el *dataloger.* Siguen generándose dos archivos, uno para datos enviados y otro para datos recibidos. En todos los casos el registro

de datos enviados se denomina “sendMessage.txt”, aunque se le renombra para identificarse con la prueba que se esté realizando. Por el contrario, el nombre del archivo con los mensajes recibidos tendrá el nombre “Testx.txt”, siendo la x el número del test que se está realizando.

Ahora que los mensajes se pueden filtrar, los ficheros que se generen ya no deben coincidir si se aplica un filtro, aunque todos los datos que aparezcan en el archivo de recepción de datos deben aparecer en el “sendMessage.txt” correspondiente.

Es necesario tener configurado el *driver* y definir mensajes en la función *main* para poder hacer funcionar el programa y registrar datos.

Para estos test sólo se modifica la configuración de las máscaras y filtros. La carga y el identificador de los mensajes se generan aleatoriamente.

6.2.1. Test 1

Aplicar máscara y filtro para que únicamente se lean mensajes con identificador igual a 0x200.

Configuración filtro y máscara	
Filtro	0x200
Máscara	0x7FF

Tabla 16. Filtro y máscara del test 1. Fuente: propia.

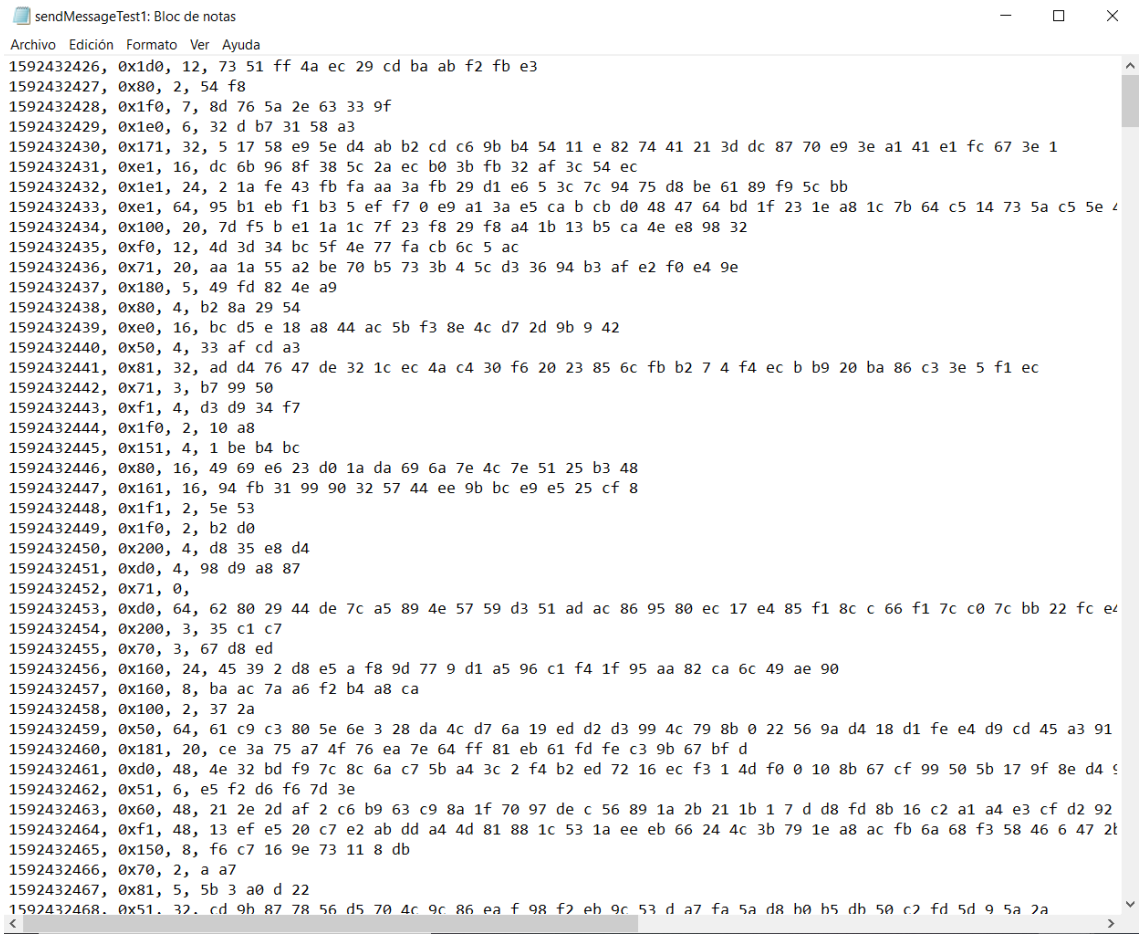

Fig. 19. Archivo generado para registro de datos enviados “sendMessageTest1.txt”. Fuente: propia.

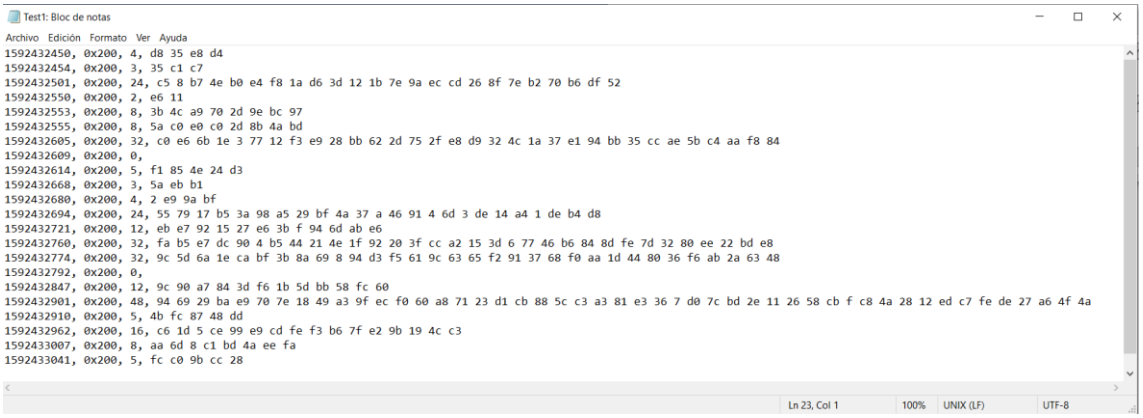

Fig. 20. Archivo generado para registro de datos recibidos “Test1.txt”. Fuente: propia.

Como se puede observar, se han enviado mensajes con identificadores aleatorios, pero solo se han registrado mensajes CANFD recibidos con el identificador 0x200. Por lo tanto, se concluye que el funcionamiento de los filtros y máscaras es correcto.

6.2.2. Test 2

Aplicar máscara y filtro para que únicamente se lean mensajes con identificadores terminados en 31 en codificación hexadecimal.

Configuración filtro y máscara	
Filtro	0x31
Máscara	0x0FF

Tabla 17. Filtro y máscara del test 3. Fuente: propia.

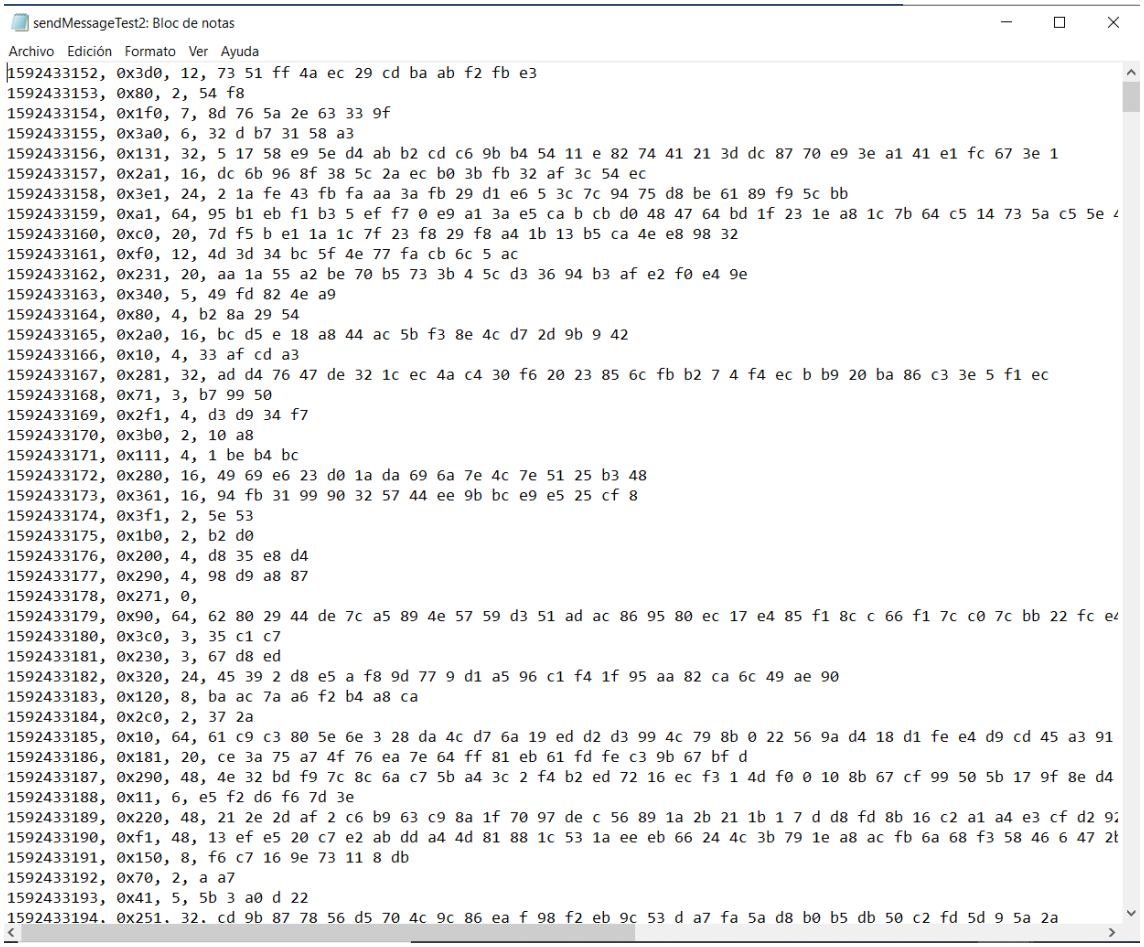

Fig. 21. Archivo generado para registro de datos enviados “sendMessageTest2.txt”. Fuente: propia

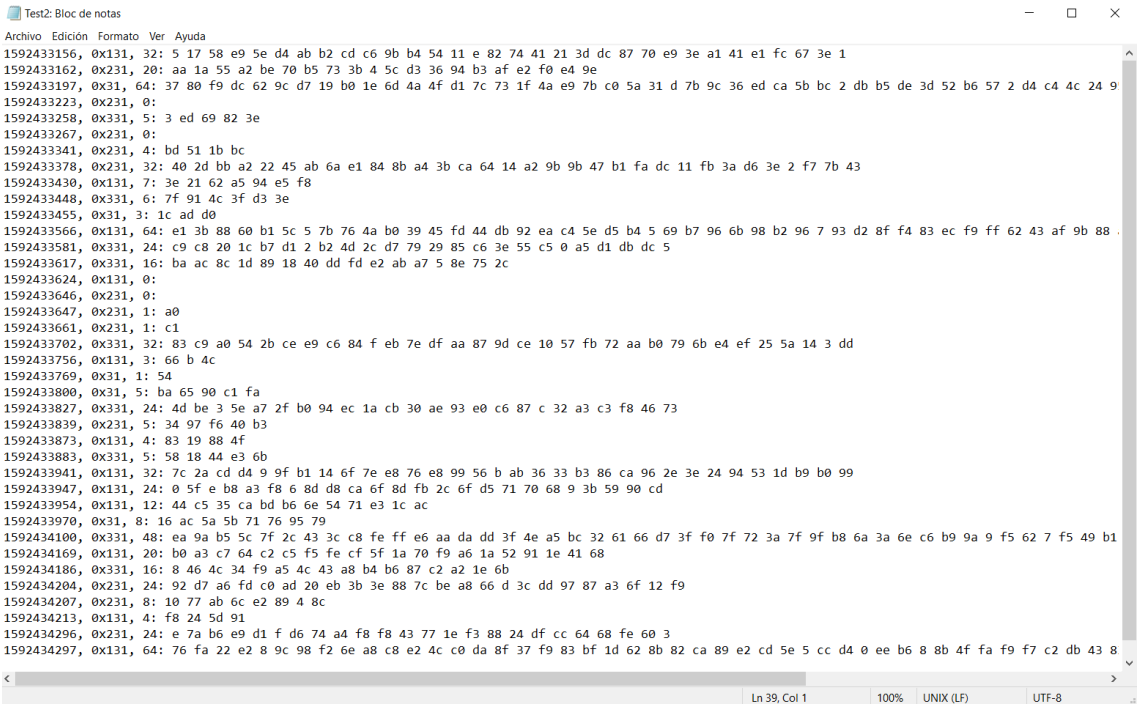

Fig. 22. Archivo generado para registro de datos recibidos “Test2.txt”. Fuente: propia.

Como se puede observar, se han enviado mensajes con identificadores aleatorios, pero sólo se han registrado mensajes CANFD recibidos con el identificador terminado en 0x31. Por lo tanto, se concluye que el funcionamiento de filtros y máscaras es correcto.

6.2.3. Test 3

Aplicar máscara y filtro para que únicamente se lean mensajes con identificador impar.

Configuración filtro y máscara	
Filtro	0x01
Máscara	0x01

Tabla 18. Filtro y máscara del test 3. Fuente: propia.

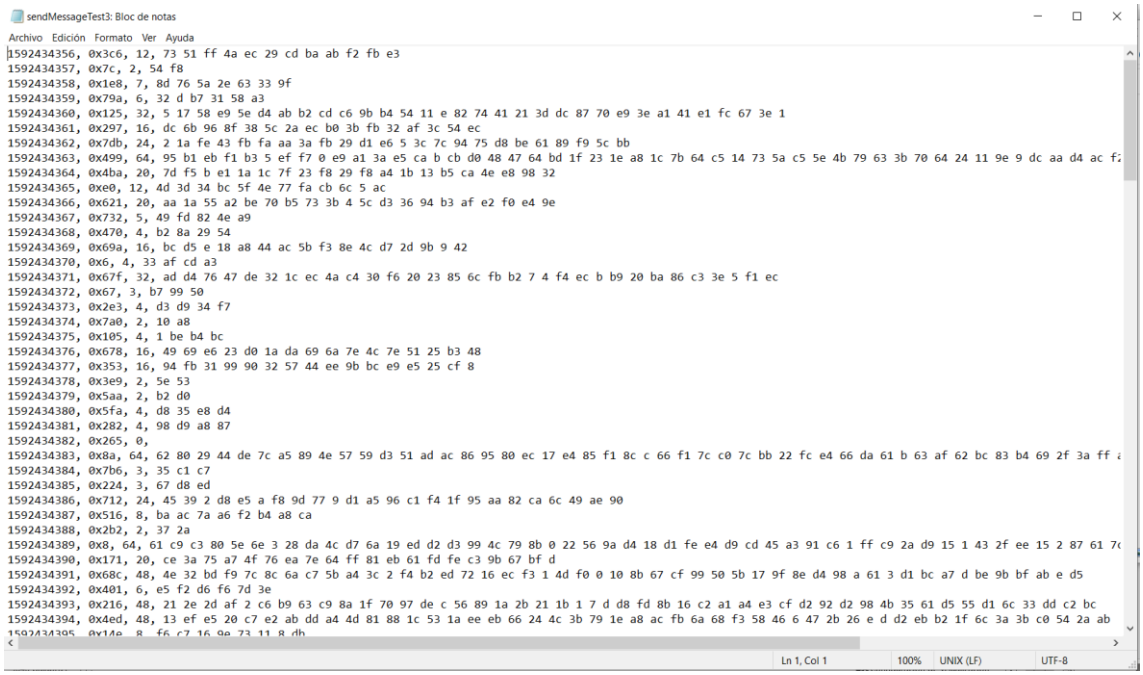

Fig. 23. Archivo generado para registro de datos enviados “sendMessageTest3.txt”. Fuente: propia

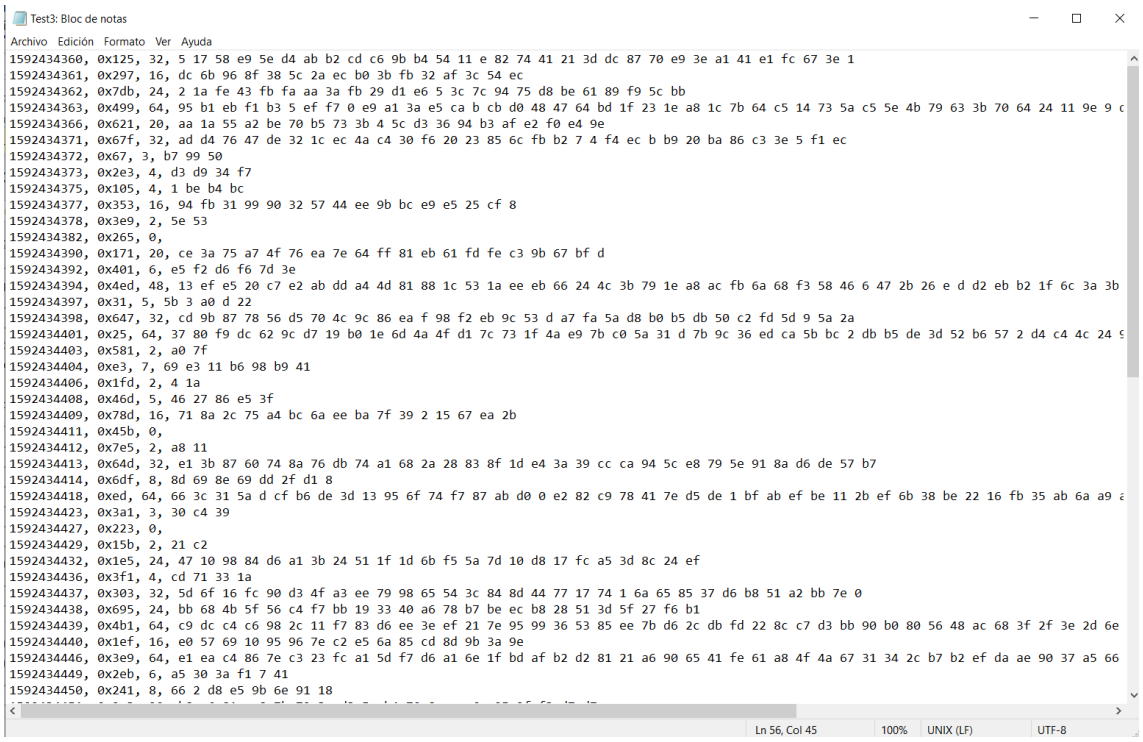

Fig. 24. Archivo generado para registro de datos recibidos “Test3.txt”. Fuente: propia.

Como se puede observar, se han enviado mensajes con identificadores aleatorios, pero sólo se han registrado datos recibidos con el identificador impar (en hexadecimal los números b, d y f son impares, ya que corresponden a los números decimales 11, 13 y 15 respectivamente). Por lo tanto, se concluye que el funcionamiento es correcto.

6.2.4. Test 4

Aplicar máscara y filtro para que únicamente se lean mensajes con identificador par.

Configuración filtro y máscara	
Filtro	0x00
Máscara	0x01

Tabla 19. Filtro y máscara del test 4. Fuente: propia.

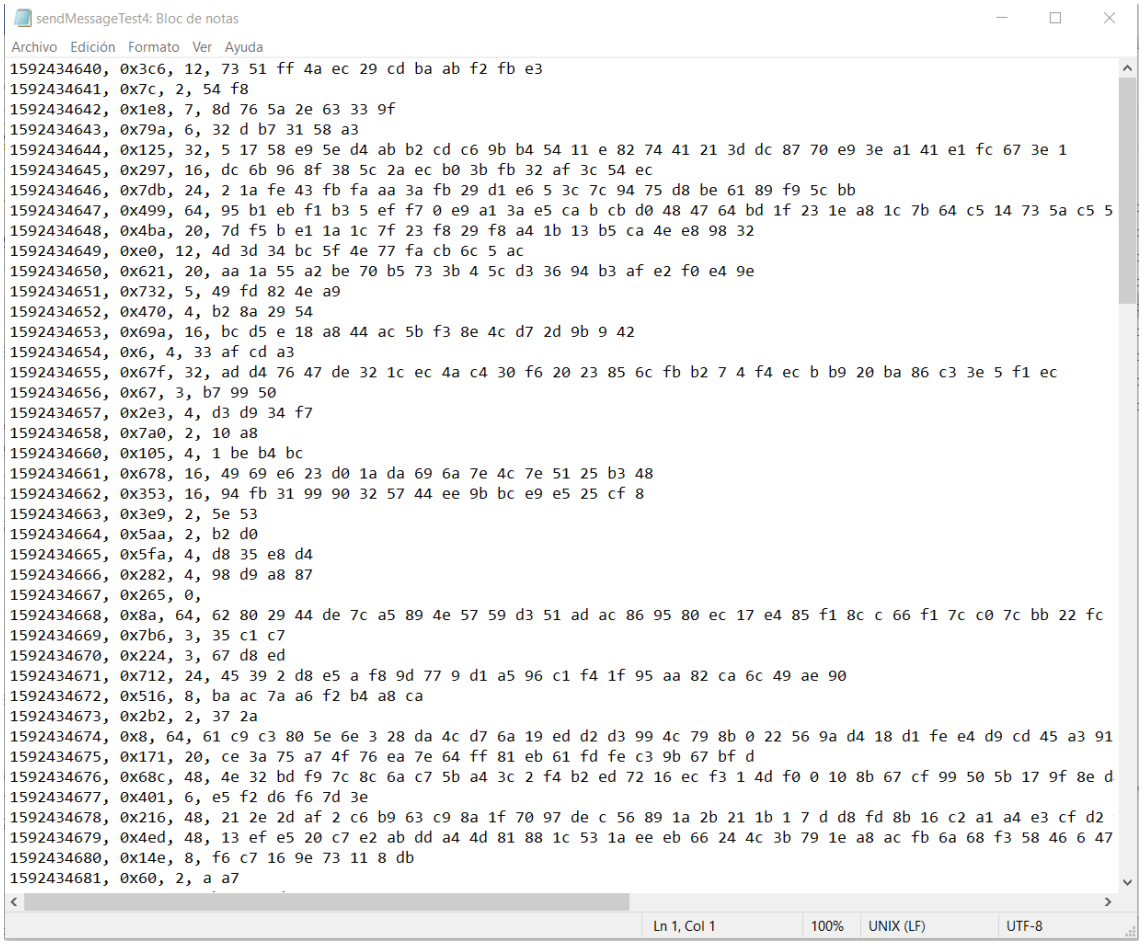

Fig. 25. Archivo generado para registro de datos enviados “sendMessageTest4.txt”. Fuente: propia

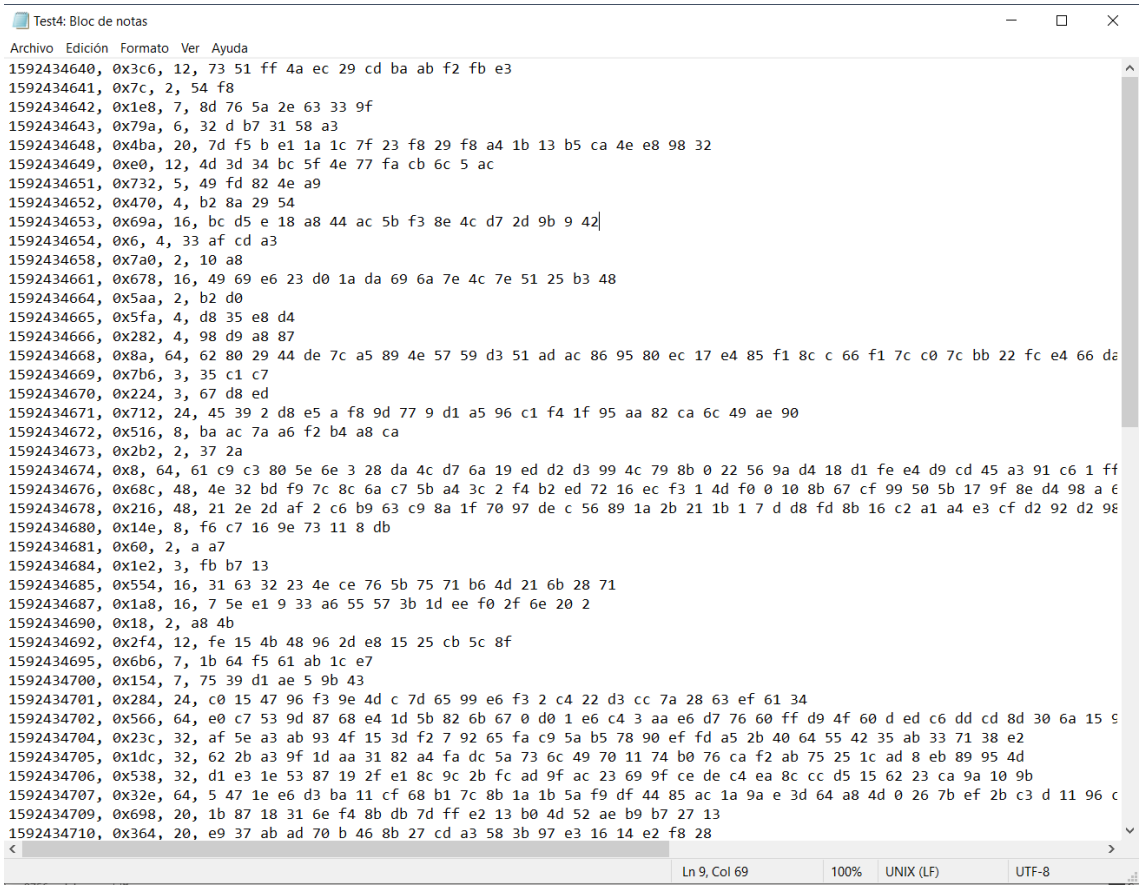

Fig. 26. Archivo generado para registro de datos recibidos “Test4.txt”. Fuente: propia.

Como se puede observar, se han enviado mensajes con identificadores aleatorios, pero sólo se han registrado mensajes CANFD recibidos con el identificador par (en hexadecimal los números a, c y e son pares, ya que corresponden a los números decimales 10, 12 y 14 respectivamente). Por lo tanto, se concluye que el funcionamiento es correcto.

7. Impacto Económico

Para poder analizar el impacto económico generado por el proyecto se va a estudiar el gasto mínimo necesario generado durante el desarrollo del trabajo.

En primer lugar, se presentan los costes para la obtención de material. Esto incluye la Rasberry Pi, el MCP2517FD Click, monitor, teclado y ratón para poder operar en la RPI, el sistema operativo Raspbian, la tarjeta SD donde el sistema operativo se almacenará y el compilador Geany IDE.

A continuación, se muestra una tabla a modo de resumen, donde se presenta el componente, el proveedor que se ha encontrado y su precio:

Componente	Proveedor	Coste
Raspberry Pi	Raspipc.es [23]	39,87 €
MCP2517FD Click	MIKROE [9]	25,00 $ - 22,37 €
Monitor LED – AOC value E970SWN HD	MediaMarkt [24]	65,99 €
Teclado – Genius KB-116	MediaMarkt [24]	9,90 €
Ratón – Logitech M90	MediaMarkt [24]	8, 90 €
Raspbian	Raspberry Pi Foundation [17]	Gratuito
Geany IDE	Raspberry-projects.com [25]	Gratuito
Tarjeta microSD	Pc Components [26]	6,98 €

Tabla 20. Coste de los componentes del sistema. Fuente: propia.

Resultando en un coste de los componentes de valor 154,01 €. Cabe destacar que, los costes del monitor, teclado y ratón no son, en todos los casos obligatorios, ya que, actualmente, es común encontrar dichos dispositivos en la mayoría de los hogares, oficinas y laboratorios de hoy en día. Por lo tanto, esta inversión en un caso práctico probablemente requiera un menor capital.

Por otro lado, el tiempo invertido es otro factor para tener en cuenta en el impacto económico. En el desarrollo de este proyecto, incluyendo fases de aprendizaje, desarrollo, pruebas y

redactado, se han invertido aproximadamente 325 horas de un ingeniero y, cabe destacar, que, a groso modo, el 40% de las horas invertidas han requerido supervisión de un tutor, por lo tanto, esta figura habría invertido 130 horas en el proyecto.

Por lo tanto, el coste del tiempo invertido seria, considerando un sueldo de 27 €/h para el ingeniero de desarrollo y un sueldo de 40 €/h para el supervisor, 8.775 € por la parte del desarrollador y 5.200 euros por la parte de supervisión. Dejando un total de 13.975 € de costes en tiempo requerido.

Resumiendo, el coste de componentes es de 154,01 € y el coste del tiempo invertido es de 13.975 € sumando un total de 14.129,01 €. De manera que, el impacto económico de producir el prototipo desarrollado en el proyecto requiere una inversión de capital con el valor ya mencionado.

8. Impacto medio ambiental

Para el análisis del impacto ambiental se va a estudiar la estimación de la contaminación generada durante el proceso de desarrollo del proyecto.

El primer paso es examinar la huella ecológica que produce el componente eje del proyecto, la Raspberry Pi.

La Rasberry Pi Fundation, concienciada de los riesgos medioambientales que azotan a la sociedad actualmente, es una de las pocas empresas que produce en su totalidad sus productos en un único país, concretamente en Reino Unido, Gales [28]. Inicialmente sus piezas se ensamblaban en China, pero al pactar con la filial inglesa de Sony pudieron trasladar la producción a Reino Unido. De manera que el impacto generado por el transporte de piezas está muy minimizado con respecto a otras empresas. Por otro lado, la propia industria del ordenador de placa simple ha firmado en su RoHS [27] que ninguno de sus dispositivos contiene más de un 0,1% de peso dedicado a *Sustancias de Muy Alta Preocupación* (SVHC) asegurando así, un producto confiable de desechar.

Por otro lado, en lo que al CAN FD respecta, ya se ha mencionado que es una versión actualizada y optimizada del protocolo CAN, permitiendo reducir el consumo durante la transferencia de datos al poder trabajar con tasas de datos más elevadas, reduciendo el tiempo de ejecución y por ende su consumo energético.

En conclusión, el impacto medioambiental generado minimiza su efecto en la producción al usar una Raspberry Pi y reduce sus exigencias energéticas al usar un protocolo actualizado como, el que es, el CAN FD.

Conclusiones y futuro del proyecto

Como conclusión se puede considerar que el objetivo se ha cumplido exitosamente. Se ha desarrollado un registrador de datos de bus CAN FD capaz registrar tanto datos de transmisión como de recepción.

Por otro lado, el proyecto está en una fase muy temprana, es decir aún hay potencial de desarrollo para futuros proyectos basado en este. A continuación, se mencionan ciertas mejoras que se podrían incorporar en el trabajo:

- Incorporar funcionalidades WiFi que permitan controlar el sistema desde un dispositivo remoto y recibir los datos registrado haciendo uso de una conexión WiFi entre éste y la Raspberry Pi.
- Implementar el detector de errores del CAN FD, el CRC (Cyclic Redundancy Check). Controlando así, que el sistema no contenga errores durante la transmisión.
- Diseñar un sistema de comunicación entre dispositivos que implemente un bus CAN FD y su registrador de datos.
- En el proyecto únicamente se han mencionado dos modos de operación, *Nominal* y *LoopBack,* y solo se ha utilizado el segundo. En el MCP2517FD Click permite trabajar en otros modos que permiten al CAN FD operar más eficientemente.

No tan solo se ha desarrollado el registrador de datos. El proyecto concluye con un *driver* que dota de la capacidad al CAN FD de operar en una Raspberry Pi, siendo un elemento muy accesible al público permitiendo a un amplio sector usar el bus e implementar nuevas aplicaciones de el en la plataforma. Además, el proyecto ha permitido explorar más a fondo el potencial del protocolo CAN FD.

Desde un aspecto más personal, el proyecto ha sido una experiencia enriquecedora que ha permitido ampliar los conocimientos del ámbito de la electrónica y explorar, desde un punto de vista más cercano, las herramientas que se utilizan en el sector. Ampliar el conocimiento del modo de operación de una Raspberry Pi o lograr un mayor entendimiento de los sistemas de comunicaciones entre dispositivos electrónicos son tan solo un par de menciones a todo lo aprendido durante el trabajo.

Agradecimientos

En primer lugar, me gustaría agradecerle al tutor del proyecto, Manuel Moreno Eguíllaz, su soporte durante el desarrollo, estando siempre disponible para cualquier situación que requiriese su supervisión y mostrando siempre interés en presentar un trabajo de calidad.

Agradecer también a las personas que he conocido durante la carrera, tanto profesores como alumnos, que me han enseñado los conocimientos necesarios para el desarrollo el proyecto.

En necesario mencionar a dos alumnos del GETI que han sido un refuerzo más para este trabajo, siendo estos Oriol Garrobé y Albert Burgés. Pese no haber tenido un contacto directo, agradecer a Oriol su excepcional trabajo de fin de grado, el cual ha demostrado ser una fuente de conocimiento mucho más valido que otros recursos que puedes encontrar hoy en día. A Albert, agradecerle el apoyo mutuo brindado durante el transcurso de sendos trabajos, ya que al hacer proyectos similares ha sido un soporte más con el que contrastar información.

Finalmente, agradecer a mi familia ofrecerme los medios suficientes para acceder a los conocimientos universitarios y el apoyo incondicional por su parte para superar cualquier obstáculo que se presente.

Bibliografía

Referencias bibliográficas

[1] CAN IN AUTOMATION, CAN knowledge . Artículo, 9 de Junio de 2020.
[2] WIKIPEDIA, VOLTAJE LEVELS. Imagen, 11 de Junio de 2020.
[3] CONTROLADOR CAN FD OFRECIDO POR MICROCHIP INC. Página oficial, 1989.
[4] WIKIPEDIA COMMONS, CAN-Frame mit Pegeln mit Stuffbits. Imagen, 5 de Abril de 2020
[5] WIKIPEDIA, NOMINAL BIT TIME. Imagen, 11 de Junio de 2020.
[6] TUTORIALS POINT, Tutorial lenguaje C. 15 Mayo 2020.
[7] CAN IN AUTOMATION, CAN FD, the basic idea. Artículo, 12 de Juno de 2020.
[8] KENT LENNARTSSON – KVASER, Comparing CAN FD with Classical CAN. Artículo, 12 de Junio de 2019.
[9] MIKROE EMBEDDED TOOLS, MCP2517FD Click. 21 de Junio de 2020.
[10] ATMEL, ATA6560/ATA6561 Datasheet. 13 de Junio de 2020.
[11] EL BUS SPI EN ARDUINO. Artículo, 13 de Junio de 2016.
[12] DOCUMENTAION OF RASPBERRY PI. Página oficial, 10 de Junio de 2020.
[13] RASPBERRY PI MODEL B. Imagen, 11 de Junio de 2020.
[14] PIGPIO LIBRARY. Página oficial, 15 de Marzo de 2020.
[15] SPI PINS. Imagen, 20 de Marzo de 2020.
[16] Raspbian SO. Página oficial, 7 de Junio de 2020.
[17] RASPBERRYPI.ORG, INSTALLISING OPERATING SYSTEM IMAGES. Documentación oficial, 15 de Marzo de 2020.
[18] RASPBERRYPI.ORG, UPDATING AND UPGRADING RASPBERRY PI OS. Documentación oficial, 15 de Marzo de 2020.
[19] MICROCHIP, Firmware Drivers. Página oficial, 13 Junio 2020.
[20] Trabajo Fin de Grado de Oriol Garrobé. 20 de Junio de 2020.
[21] CAN BUS Mask on Raspberry Pi. Imagen, 13 Junio 2020.
[22] WIKIPEDIA, Tiempo Coordinado Universal. Artículo, 13 Junio 2020.
[23] RASPIPC.ES, Buy Rasberry Pi. 21 de Junio de 2020.
[24] MEDIAMARKT, Buy components. 21 de Junio de 2020.
[25] RASPBERRY-PROJECTS, Installing Geany IDE. 20 de Marzo de 2020.
[26] PC COMPONETS, Buy SD. 21 de Junio de 2020.
[27] RASPBERRYPI.ORG, RoHS de Raspberry Pi. Certificado, 21 de Junio de 2020
[28] RASPBERRYPI.ORG, Made in the UK!. Artículo, 21 de Junio de 2020.