Master Thesis

Master's Degree in Industrial Engineering

Black-Box based on Sigfox and CAN-FD

REPORT

Author: Jordi Torné Chertó
Supervisor: Manuel Moreno Eguílaz

Call: January 2022

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Abstract

This report describes the process of developing an automotive CAN-FD Black-Box that uses Sigfox communication.

At first, the document introduces the components and programs that have been used, explaining their reason to be and the role they play. In the same way, CAN-FD and SPI protocols are analysed too, emphasizing their behaviour.

Subsequently, it catalogues and describes the different drivers that had been used. Besides, it shows step by step the different parts that establish the main file to make it more understandable. After that, the code is validated to ensure its correct performance.

Finally, the modification of the main file to send Sigfox messages and accept external CAN-FD messages is presented. However, this time the validation has been done using a demonstrator that includes some CAN-FD nodes. The results are shown in a linked video which confirms the correct operation of the designed Black-Box.

Index

1. Glossary

ACK: Acknowledge

ADC: Analog to Digital Converter

BRS: Bit Rate Switch

CAN FD: Controller Area Network Flexible Data-Rate

CAN: Controller Area Network

CRC: Cyclic Redundancy Check

CS: Chip Select

DAC: Digital to Analog Converter

DLC: Data Length Code

EDR: Event Data Recorder

EOF: End of Frame

ESI: Error State Indicator

EVB: Evaluation Board

GPIO: General Purpose Input Output

GPS: Global Positioning System

I2C: Inter-Integrated Circuit

IDE: Identifier Extension Flag

IDE: Integrated Development Environment

IoT: Internet of Things

ISP: In-System Programming

LED: Light-Emitting Diode

LSB: Least Significant Bit

LVLLT: Low Voltage Transistor-Transistor Logic

MISO: Master In Slave Out

MOSI: Master Out Slave In

MSB: Most Significant Bit

PCB: Printed Circuit Board

RF: Radio-Frequency

SCLK: Serial Clock

SDK: Software Development Kit

SOF: Start of Frame

SPI: Serial Peripheral Interface

SRR: Substitute Remote Request

SS: Slave Select

UART: Universal Asynchronous Receiver-Transmitter

USB: Universal Serial Bus.

2. Introduction

The Controlled Area Network (CAN) protocol has become the most widely used in-vehicle networking for electronic control units (ECUs) and sensors communications. However, with the growing consumer, safety, and legislative demands on cars, the number of embedded electronics has been rising in the same way that the amount of data being, making the protocol increasingly insufficient for the needs [1].

To solve the deficiencies/limitations of the CAN protocol, the Controlled Area Network Flexible Data-Rate (CAN-FD) is born. This protocol not only increases the bandwidth but also shares the same physical layer as its predecessor, minimizing significantly the total costs of upgrading the system.

Due to the above and knowing that new legislations are forcing the introduction of Black-Boxes in cars, getting one that communicates with CAN-FD is essential in these times.

2.1. Objectives

The main objective of the project is to create a functional Black-Box, which is an event data recorder installed in some vehicles to measure and track parameters such as it is acceleration, speed, braking or steering wheel angle.

At the very beginning, a Black-Box was only designed to help investigators determine the causes of accidents, flashing to a non-volatile memory the data recorded just before the crash. However, today a few numbers of Black-Boxes include communication systems to transmit data allowing new functionalities such as tracking in real-time the behaviour of a vehicle. This is especially interesting for insurance to determine more accurately the driver risk factor.

For that reason, in this project a CAN-FD Black-Box with the ability to send data messages via wireless networks is going to be developed. To do that, some devices are going to be programmed with a code created specifically for this purpose. The language that has been chosen is the C language because is the language of drivers that will be used.

As a secondary goal of the project, it has been tried to create a clear and easy-to-understand code to be reused in future projects.

2.2. Scope

The scope of the project can be divided into four parts: Setting the connections of the used devices, modifying the drivers to fit with an SPI protocol implemented via software, developing a source file with the code that defines the behaviour of the Black-Box, and doing the proper validations to ensure its correct operation.

It will not enter in the scope of the project the real implementation of the Black-Box as it will be only used in a CAN-FD demonstrator. In the same way, creating an extended list of warnings/alarms will not be included because to demonstrate its correct functioning there is only needed to define a few.

2.3. State of the art

These last years the inclusion of a Black-Box in a vehicle has aroused great interest due to the regulations of different countries. Some examples are China, the United States and now (in July 2022) European Union that are forcing to equip a Black-Box in all the new vehicles.

The Black-box that the normative refers to is an Event data recorded, which records and stores the data of a period shortly before, during and immediately after the collision. The data must include vehicle speed, braking, position and tilt of the vehicle on the road, the state and rate of activation of all its safety systems, 112-based eCall in-vehicle system, brake activation and relevant input parameters of the on-board active safety and accident avoidance systems, with a high level of accuracy and ensured survivability of data [2].

However, recently has appeared another type of Black-Box that include communication systems. These are especially interesting in detecting rare issues, getting field data, developing predictive maintenance and sending the crash report to a server, easing the labour of getting the data from the vehicle.

Nowadays, there exists some manufacturers that provide this new Black-Box but they are very expensive, difficult to buy for a particular or they use communications systems such as 3G/4G, which does not have complete coverage everywhere. An example is CANedge2 from CSS ELECTRONICS.

Getting the ideas from the Black-Boxes already created, this project is going to develop a low-cost Black-Box with a great coverage communication system based on CAN-FD. The starting point will be the TFG by Javier A. De Esteban Garau named "***Registrador de CAN-***

FD utilizando lenguaje C y una Raspberry PI", which created a datalogger for CAN-FD protocol.

3. Tools

This project has needed three different electronic boards to provide CAN-FD and Sigfox communication to the Black-Box. The part of the software has been implemented using C because examples, library and builders for the evaluation board were designed in that way.

3.1. Devices

3.1.1. Telecom design EVB for TD1204

3.1.1.1. TD1204

TD1204 is a high performance, low current SIGFOX gateways, RF transceiver and GPS receiver.

This module includes some features such as an ARM Cortex M3 processor, sensitive and powerful RF (radio-frequency) transceiver, high-efficiency GPS receiver, 3D accelerometer, LVTTL (low voltage transistor-transistor logic) UART, I^2C bus, multiple timers, high-speed ADC and DAC and numerous GPIOs. Thanks to those features, TD1204 can provide a large number of applications in an efficient environment with very low consumption [3].

Some examples of applications of TD1204 are SIGFOX transceiver (fully certified), geolocation and tracking, universal timing and synchronization, sensor network, health monitors, home security and alarm, industrial control, remote control, vehicles and objects tracking or people and pet's geolocation.

Fig. 3.1 shows its functional block diagram:

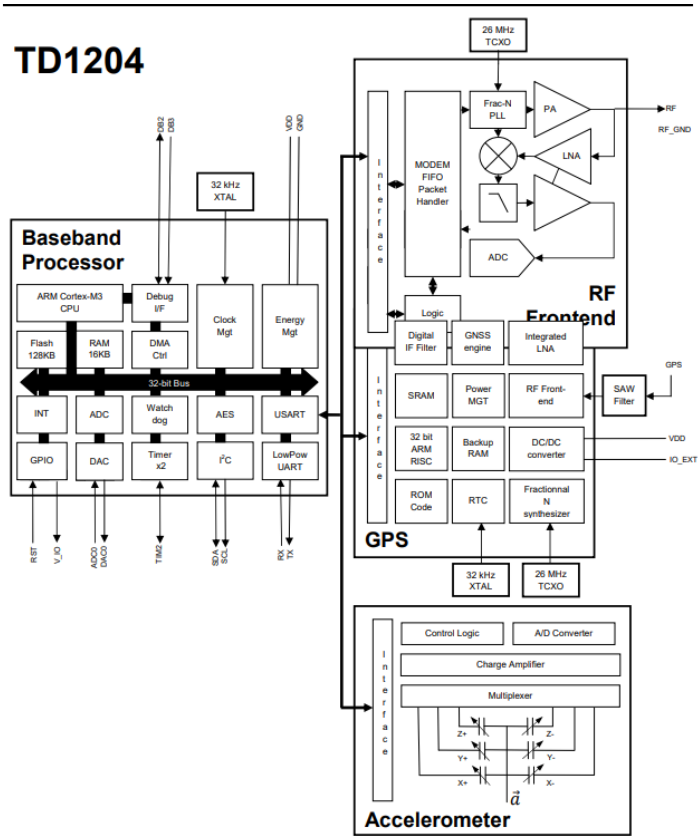


Figure 3.1. TD1204 Functional bock diagram. Source: [3].

3.1.1.2. TD1204 Evaluation Board

TD1204 EVB is an evaluation board that includes the TD1204 module with some more components, providing a rich development and demonstration platform.

The features it includes are FTDI USB to the serial cable connector, drop out voltage regulator, current measurement jumper, super blue LED, SMA Sigfox antenna connector, U.FL GPS antenna connector, ISP header and breakout header.

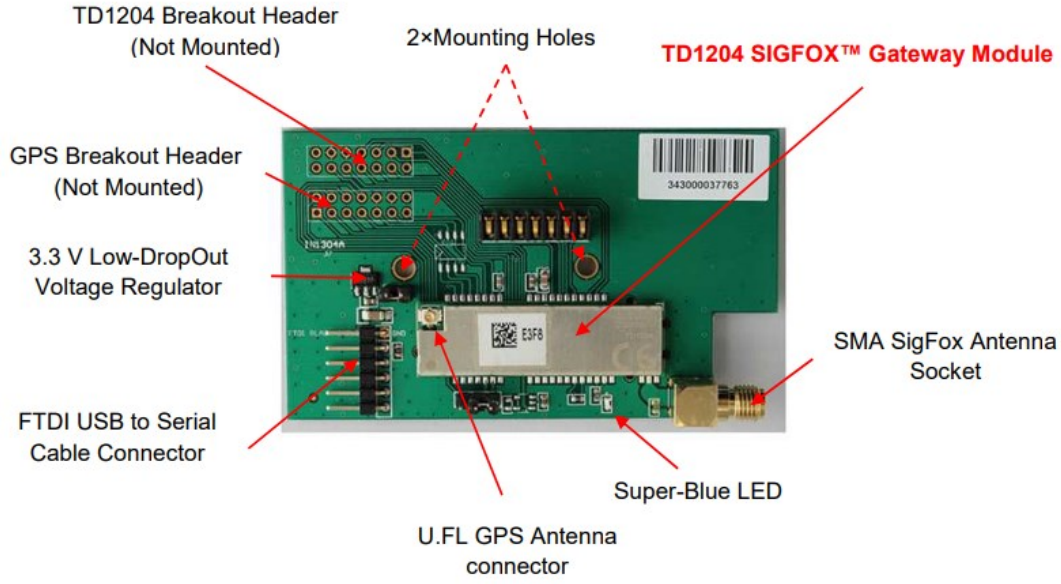


Figure 3.2. TD1204 EVB PCB. Source: [4].

One of the most interesting things that this evaluation board contains is the FTDI USB cable connector, which allows us to communicate between our computer, the FTDI hardware device and the ISP header which provides an in-situ Programming/Debugging interface.

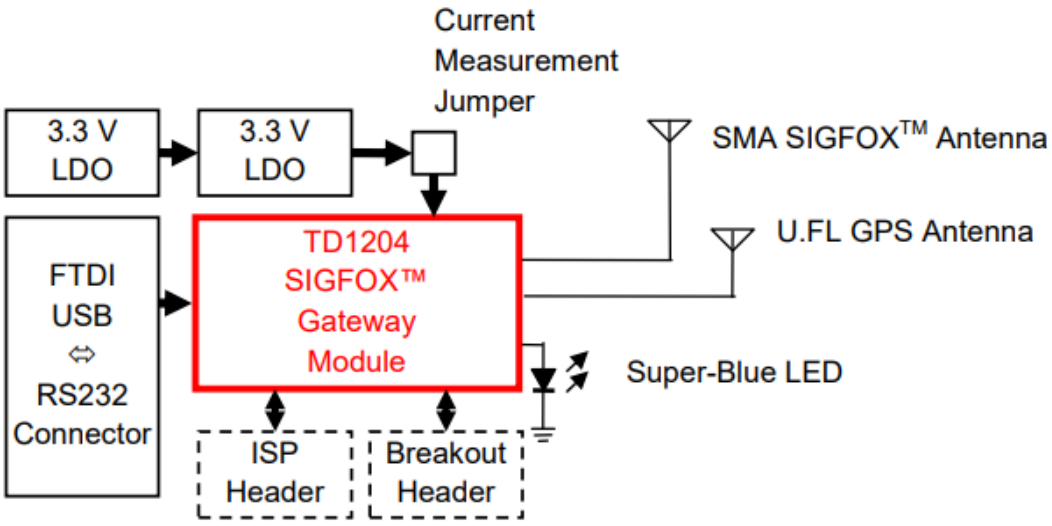


Figure 3.3. TD1204 EVB Block Diagram. Source: [4].

3.1.2. Starter kit EFM32TG-STK3300

The starter kit is used for evaluation, prototyping and application development for the EFM32TG family with the ARM Cortex-M3 CPU core. In other words, this device had been

used to program the microprocessor from the evaluation board using the computer.

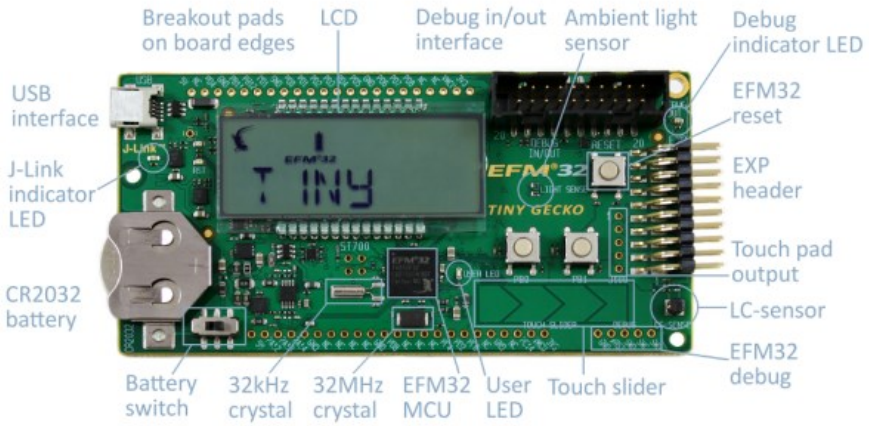


Figure 3.4. Starter Kit EMF32TG-SKT3300 PCB. Source: [5].

Its main features are:

- Advanced Energy Monitoring provides real-time visibility into the energy consumption of an application or prototype design.
- On-board debugger with debug out functionality.
- 160-segment Energy Micro LCD.

As mentioned before, in this project the board has been exclusively used to provide the debug out functionality. This feature is reached thanks to the J-link it includes.

J-link is not more than a debug probe that optimizes the debug and flash program experience, allowing to read and write registers and memory, support breakpoint debugging. Also, it can be directly communicated via SPI protocol and is fully supported by Eclipse IDE and ARM Cortex-M3.

Debug OUT: In this mode, the on-board debugger can be used to debug a supported Silicon Labs device mounted on a custom board.

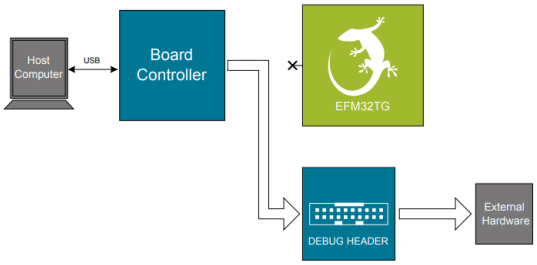


Figure 3.5. Debug OUT mode. Source: [5].

As shown in Fig. 3.5, in the debug OUT mode the microcontroller EFM32TG will not be used, although the host computer, board computer and the debug header will.

The Starter kit's block diagram can be seen in Fig. 3.6.

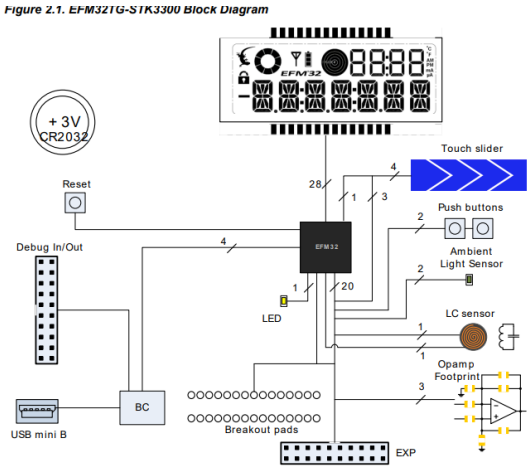


Figure 3.6. Starter Kit EMF32TG-SKT3300 Block diagram. Source: [5].

3.1.3. MCP2517FD CLICK

The last board is the MCP2517FD CLICK, which is a complete CAN/CAN-FD solution. It includes the MCP2517FD CAN-FD controller and the ATA6563 high-speed CAN transceiver from Microchip.

The MCP2517FD is an external CAN-FD controller with an SPI interface. This component is specially used to provide a CAN-FD channel when the microcontroller is either lacking a CAN-FD peripheral or when there are not enough CAN-FD channels. It supports the CAN

frames in the classical format (CAN2.0B) and CAN flexible data rate (CAN-FD).

Its block diagram is shown in Fig. 3.7:

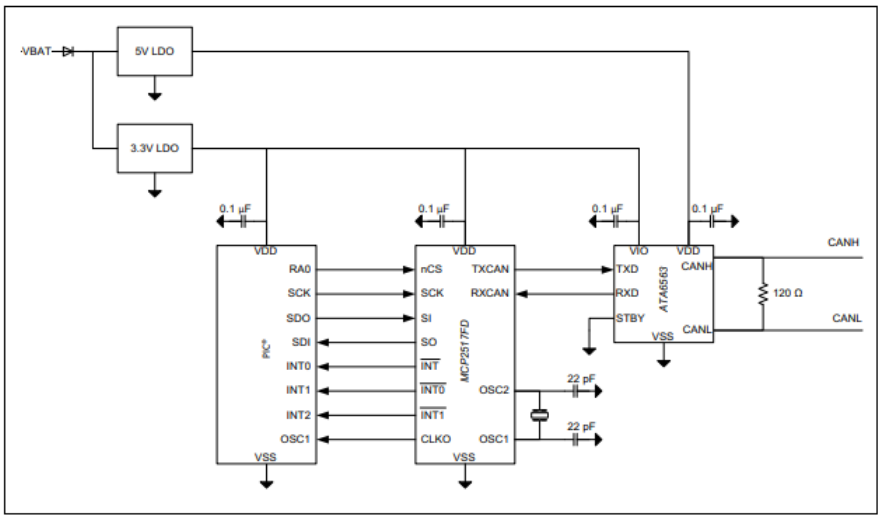


Figure 3.7. MCP2517FD Interfacing with a microcontroller. Source: [6].

The ATA6563 is nothing more than a high-speed CAN transceiver that provides an interface between a CAN protocol controller and the physical two-wire CAN bus. Its block diagram is shown in Fig. 3.8.

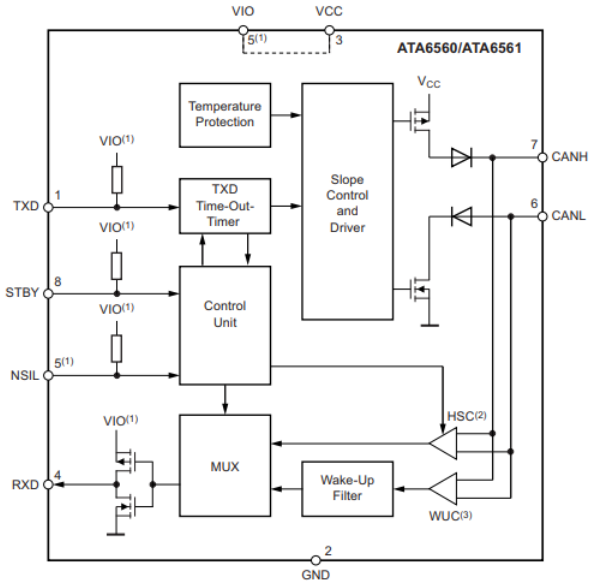


Figure 3.8. ATA6563 Block diagram. Source [7].

As you can imagine this board is the one that provides CAN-FD connectivity to the device.

3.2. Connections

3.2.1. Connections between TD1204 EVB and Starter kit EFM32TG-STK3300

Some people may think that linking different PCBs and making them work properly is a simple thing, but beyond reality, sometimes is one of the headaches that engineers face. Hence, to ease that, this part of the chapter will summarize how they were connected.

However, before starting talking about the connection it has to be explained that the mode Starter kit EFM32TG-STK3300 has to be changed to debug OUT. This mode is needed to debug and flash programs into the microprocessor (ARM cortex-M3) of the evaluation board. The change on the mode has been done with Simplicity Studio, a program that allows changing the J-Link adapter to debug mode.

Once done, the connection between the Starter kit EFM32TG-STK3300 and TD1204 EVB can be set using the debug In/Out connector and the ISP header. Fig. 3.9 shows the wiring.

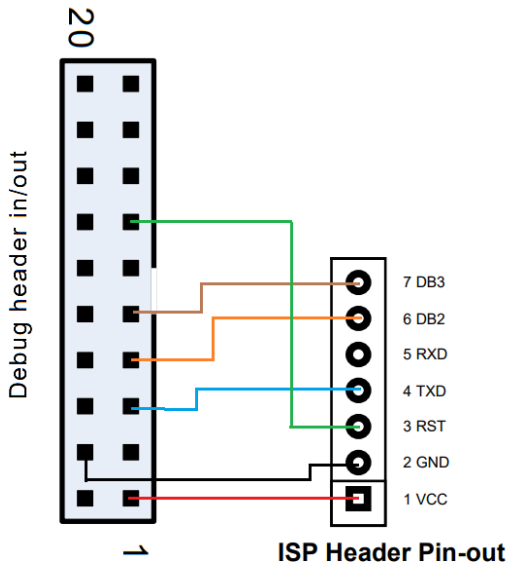


Figure 3.9. Wiring between TD1204 EVB and Starter kit EFM32TG-STK3300. Source: Own.

Where:

- VTARGED (Pin 1): Target reference voltage on the debugged application. It is connected to the VCC pin of the TD1204 EVB.
- TMS/SWDIO (Pin 7): Serial wire data I/O, this signal provides the SWD programming/ debugging signal interface to the integrated TD1204 ARM CPU. It is connected to the DB2 pin.
- TCK/SWCLK (Pin 9): Serial wire clock that provides the signal to the integrated TD1204 ARM CPU. It is connected to the DB3 pin.
- TDO/SWO (pin 13): Serial wire output, this signal provides the UART data coming from the host application processor going to the TD1204 module. This pin is connected to the RXD (Data Receive Signal) of the ISP header, which is a Low-Power UART Data Receive Signal.
- TDI (Pin 5): This pin is called JTAG Test Data In. It provides the UART data going from the TD1204 module out to the host application processor. This pin is connected to the TXD (Data Transmit Signal) of the ISP header.
- /RESET (Pin 15): This signal resets the target device and it is connected to the RST pin of the ISP header, which resets the TD1204 to the initial state.

3.2.2. Connections between TD1204 EVB and MCP2517FD CLICK

The other connection involves the TD1204 EVB's break-out Header pin-out and the MCP2517FD click SPI pins.

Here a little problem appears because the evaluation board does not include SPI pins, making it to be implemented using other pins. However, this will carry some extra work related to including libraries and modifying some existing ones.

The connection between them has to be done using the SPI bus. However, the evaluation board does not include a hardware SPI. Because of that, in this project the SPI is implemented by software, using several GPIO pins of the TD1204 EVB.

The SPI protocol is a four-wire synchronous serial communication protocol, where each wire carries a different logical signal. That means that 4 pins will be needed:

- SCLK: Serial Clock (is always output from master)
- MOSI: Master Out Slave In (master sends data to slave)

- MISO: Master In Slave Out (slave sends data to master)
- CS/SS: Chip Select /Slave Select (is a master output and often is active low)

Before deciding which pins will be used it has to be ensured that they will work properly. For this reason, a simple program has been launched to the TD1204 board. That program made the pins fluctuate periodically between 3.3 V and 0 V (setting and clearing the pin).

```
// Define the LED pin as an output in push-pull mode
GPIO_PinModeSet(DAC0_PORT,DAC0_BIT, gpioModePushPull, 1);
while(1)
    {
    GPIO_PinOutSet(DAC0_PORT, DAC0_BIT);
    TD_RTC_Delay(T5MS);
    GPIO_PinOutClear(DAC0_PORT, DAC0_BIT);
    TD_RTC_Delay(T5MS);
    }
```

Figure 3.10. Code used to check each GPIO pin. Source: Own.

If a pin fluctuates in the frequency defined by the program and with the correct voltage would mean that the pin could be used for the SPI software implementation. The way it has been proven the viability of each pin was using an oscilloscope.

After testing some pins, it was decided to use the SDA (pin1) as SCK, SCL (pin 3) as MOSI, ADC0 (pin 8) as MISO and DAC0 (pin 10) as SS.

However, as mentioned before, it is not as easy as wiring them directly, and it has to be included in an SPI header file that implements that type of communication via software.

```
#define SCK_PORT    SDA_PORT            // SCK (pin 1 TD1204 Break-out
Header)
#define SCK_BIT     SDA_BIT

#define MOSI_PORT   SCL_PORT            // MOSI (pin 3 TD1204 Break-out
Header)
#define MOSI_BIT    SCL_BIT

#define MISO_PORT   ADC0_PORT           // MISO (pin 8 TD1204 Break-out
Header)
#define MISO_BIT    ADC0_BIT
```

```
#define SS_PORT     DAC0_PORT           // SS (pin 10 TD1204 Break-out
Header)
#define SS_BIT      DAC0_BIT
```

Figure 3.11. Code used to link each pin to a constant. Source: Own.

After that, it only remains the connection of the wires, where the DAC0 pin will be connected with CS, SDA with SCK, ADC0 with SDO, SCL with SDI, VCC with 3V3, GND with GND and external USB which provides 5 V with 5V pin.

Figure 3.12. Wiring between TD1204 EVB and MCP2517 FD. Source: Own.

3.3. Eclipse IDE

The IDE (Integrated Development Environment) used in this project was the Eclipse, which is a free and open-source software specially oriented to develop applications in a large number of languages via plug-ins, including C.

As an IDE, it could be said that it is an embedded IDE because almost every embedded environment has Eclipse-based IDE with integrated debugger support, and as such, it is simple to use.

However, the main reason to use Eclipse was that the software development kit (SDK) of the TD1204 was designed for this IDE, which includes the stand-alone Eclipse package for the

part of compiling and the flash/debug launchers.

The part of setting up the Eclipse to work with TD1204 is a bit complex, but it is defined in the Telecom Design RF Module Software Development Kit.

3.4. Sigfox

As mentioned in the introduction, the Black-Box needs wireless communication to provide features such as remote tracking of the vehicle. In this case, the idea is to develop a device that sends alerts/warnings when an event occurs, these events could be exceeding the maximum speed or RPM.

Because of the nature of the Black-Box, it could be said that the feature that the communication must have, is a wide-area network. However, things like bit rates and data length are not important at all, as the device will not usually transmit messages, and when it does, they will not be necessarily long.

Taking into consideration the aforementioned, I should therefore point out that a low-power wide-area network (LPWAN) would be convenient because of its coverage, low consumption and costs that it has associated.

There exist a few convenient options to provide this communication such as LoRa or Sigfox. Both of them are LPWAN but while the first has a bigger bandwidth (allowing more data transmission), the other has more installed networks.

Because of the importance of the installed networks, it makes more ease to use, Sigfox had been chosen. However, it has other characteristics that make it interesting such as its low cost, low energy consumption, wide-area network, coverage anywhere and great device operability using Sigfox Backend.

This Sigfox Back-End is the endpoint of the communication and in this case will act as a server, where all the alarms/warnings transmitted by the Blackbox will end there. Those messages could be up to 12 bytes and are shown in Hex format.

Moreover, this server provides a web application interface for device management and configuration of data integration, as well as standards-based web APIs to automate the device management and implement the data integration.

3.5. Black-Box

This last chapter has the purpose of making more understandable the behaviour of the Black-Box, clarifying the function of each device and the function of the whole system.

In a first instance, it could say that the Black-Box has some ECUs connected to the Black-Box. This connection has been done using the MCP2517FD Click, which not only provides a CAN-FD transceiver, but also a channel for the protocol.

The next part of the system is the TD1204 EVB, which provides the processor and contains the antenna to send the Sigfox messages. However, to program, the board has needed the Starter kit EFM32TG-STK3300 as it is compatible with the processor and with the ECLIPSE IDE.

Finally, the last part of the system is the Sigfox Back-End, which receives the warnings of the Black-Box and stores them.

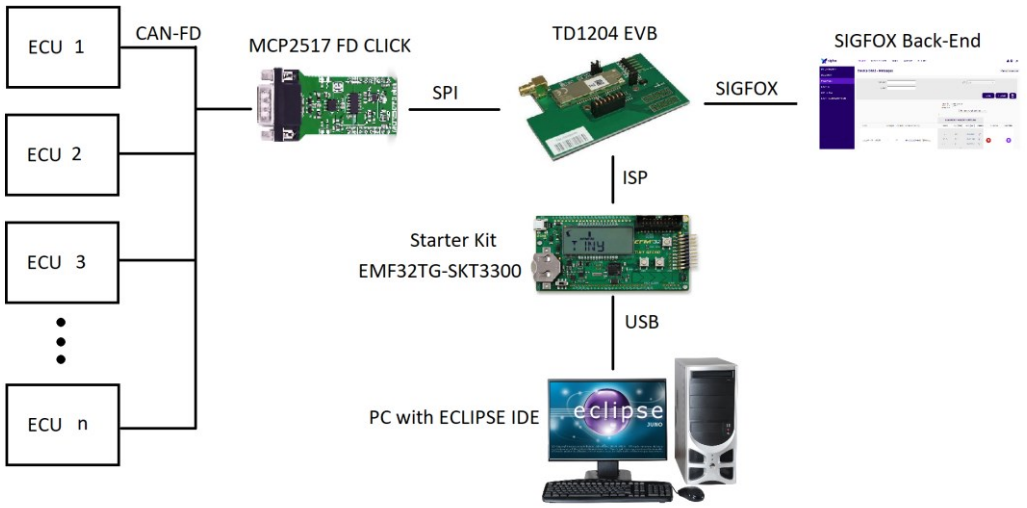


Figure 3.13. Black-Box system sketch. Source: Own.

4. Communication protocols

4.1. SPI (Serial Peripheral Interface)

SPI is a synchronous serial communication interface developed by Motorola which requires 4 wires MOSI, MISO, SCK and SS. It is a full-duplex communication protocol, meaning that can send and receive messages simultaneously. This protocol is commonly used in circuit boards as an intra-system protocol.

Serial Peripheral Interface requires at least two participants, one master and one or more slaves, where both can receive and send data, which is transmitted bit by bit through MOSI or MISO line.

When the master sends the data, it is transmitted through MOSI (Master Output Slave Input) wire to the slave. The data sent from the master to the slave is usually sent with the most significant bit first (MSB).

In the case where the slave transmits data back to the master, the used line is the MISO (Master Input Slave Output). However, in this case, the data is usually sent with the least significant bit first (LSB).

The steps of SPI data transmission are 4: [8]

1. Master transmits the clock signal.
2. Master switches the SS/CS (Chip Select) pin to a low voltage state, which activates a specific slave.
3. Master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received.
4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. Master reads the bits as they are received.

In this project, as mentioned in chapter 3.2.2, SPI has to be implemented via software and because of that, the used code is commented next.

The first instruction that is defined is the one that initialises the 4 SPI signals. In this case, it puts all the signals to 0, except for the Chip Select because it is activated when it has a low voltage.

```
void init_SPI(void)
{
    // Configuration of GPIOs
    GPIO_PinModeSet(SS_PORT, SS_BIT, gpioModePushPull, 1); // SS = 1
    GPIO_PinModeSet(SCK_PORT, SCK_BIT, gpioModePushPull, 0); // SCK = 0
    GPIO_PinModeSet(MOSI_PORT, MOSI_BIT, gpioModePushPull, 0); // MOSI = 0
    GPIO_PinModeSet(MISO_PORT, MISO_BIT, gpioModeInputPull, 0); // MISO = 0
}
GPIO_PinOutClear(SS_PORT, SS_BIT); //CS = 0
```

Figure 4.1. Code to initialize the SPI signals. Source: Own.

The other instructions are used to send and receive data, two of them writes, or send, data to the slave, and the other read, or receive, data from the slave to the master.

Writing using the software-based SPI needs the *spiWrite* and *SPI_transfer* functions. The first one, *spiWrite*, has the purpose of calling *SPI_transfer* sequentially. As it can be seen in Fig. 4.2, the function requires 2 arguments, where one is a pointer to the array of data (tx*) and the other the number of bytes (n).

```
void spiWrite(unsigned char *tx, unsigned char n)
{
    unsigned char i;
    for(i=0;i<n;i++)
    {
        SPI_transfer(tx[i]);
    }
}
```

Figure 4.2. spiWrite function. Source: Own.

SPI_transfer is the function in charge of transmitting data bit by bit via the MOSI line. This one has an algorithm that changes the state of the MOSI depending on the value of the bits. Furthermore, it also read data from the MISO at the same time it writes, because of that it was said that SPI is a full-duplex communication protocol.

The algorithm can be defined in 5 steps:

1. First, it changes the logical value of MOSI according to if the most significant bit of the byte is 1 or 0

2. Move the position of the byte to compare the following MSB in the next iteration

3. Set the clock to 1
4. Read the MISO and save it on the LSB of the Byte
5. Set the clock to 0 and repeat all the processes for 8 times

```
unsigned char SPI_transfer(unsigned char byte)
{
    unsigned char counter;

    for(counter = 8; counter; counter--)
    {
        if (byte & 0x80)
        {
            Set_MOSI_1;  // MOSI = 1;
        }
        else
        {
            Set_MOSI_0;  // MOSI = 0;
        }
        byte <<= 1;
        Set_SCK_1; // SCK = 1; /* a slave latches input data bit */
        if (Read_MISO) //(MISO)
        {
            byte |= 0x01;
        }
        Set_SCK_0; // SCK = 0; /* a slave shifts out next output data bit */
    }
    return(byte);
```

Figure 4.3. SPI_transfer function. Source: Own.

Reading using the software-based SPI requires the *spiRead* function, which is very similar to the last part of the *SPI_Transfer*, but in this case, it saves the values of MISO inside one of the arguments.

```
void spiRead(unsigned char *rx, unsigned char n)
{
    unsigned char counter;
    unsigned char i;
    char byte;

    for(i=0;i<n;i++)
```

```
    {
        byte = 0;
        for(counter = 8; counter; counter--)
        {
            byte <<= 1;
            Set_SCK_1; // SCK = 1; /* a slave latches input data bit */
            if (Read_MISO) //(MISO)
            {
                byte |= 0x01;
            }
            Set_SCK_0; // SCK = 0; /* a slave shifts out next output data
bit */
        }
        rx[i] = byte;
    }
}
```

Figure 4.4. spiRead function. Source: Own.

4.2. CAN-FD (Controller Area Network Flexible Data-Rate)

CAN-FD is an expansion of the CAN Bus. It is defined by the standards of ISO 11898-1:2015, which respond to the increased bandwidth requirements for automotive and industrial networks.

CAN-FD may be defined as a data communication protocol that uses a multi-master serial bus to connect the electronic control units (ECUs), also called nodes in the automotive sector.

The major benefits that CAN-FD offers are: [9]

- Increase the data length to 64 bytes instead of the 8 bytes of classic CAN.
- Increase the bit rate within the data phase to 5 Mbit/s, going beyond the previous limit of 1 Mbit/s.
- Reliability improvement with its new cyclic redundancy check (CRC) and the addition of protected stuff-bit.
- Smooth transition because CAN-FD and CAN ECUs can be mixed.

Software and hardware changes between CAN and CAN-FD are minimal. In software, both

message format is very similar and in hardware, the only difference is to use a new protocol controller with a qualified transceiver that allows faster speeds [1].

The physical layer of CAN-FD and High-speed CAN (ISO 11898-2) is the same and consists of a pair of twisted wires of 120-ohm resistor that connects the nodes. Both cables are differential wired-AND signals which allow 2 states, dominant (logical 0) and recessive (logical 1). The cables are called CANH (high) and CANL (low).

The dominant state happens when the voltage of CANH is bigger than the CANL, where the first is towards 3.5 V and the other towards 1.5 V.

The recessive state, unlike the other, takes place when no device is transmitting a dominant signal. In that situation, the terminating resistors return passively the two wires to the recessive state where both are near 2.5 V (it is considered recessive if the differential voltage between CANH and CANL is no more than 0.5 V) [10].

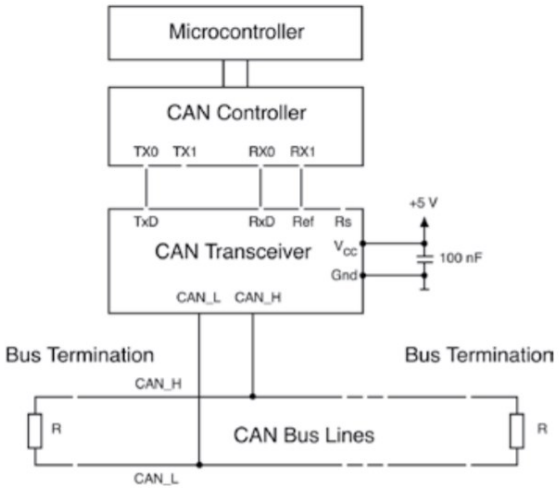


Figure 4.5. CAN and CAN-FD physical layer. Source: [11].

4.2.1. CAN-FD frame

CAN-FD allows two types of frames, FD Base Frame Format (FBFF) and FD Extended Frame Format (FEFF) with an 11-Bit identifier and a 29-Bit identifier, respectively. The other fields of the frame remain identical for both formats.

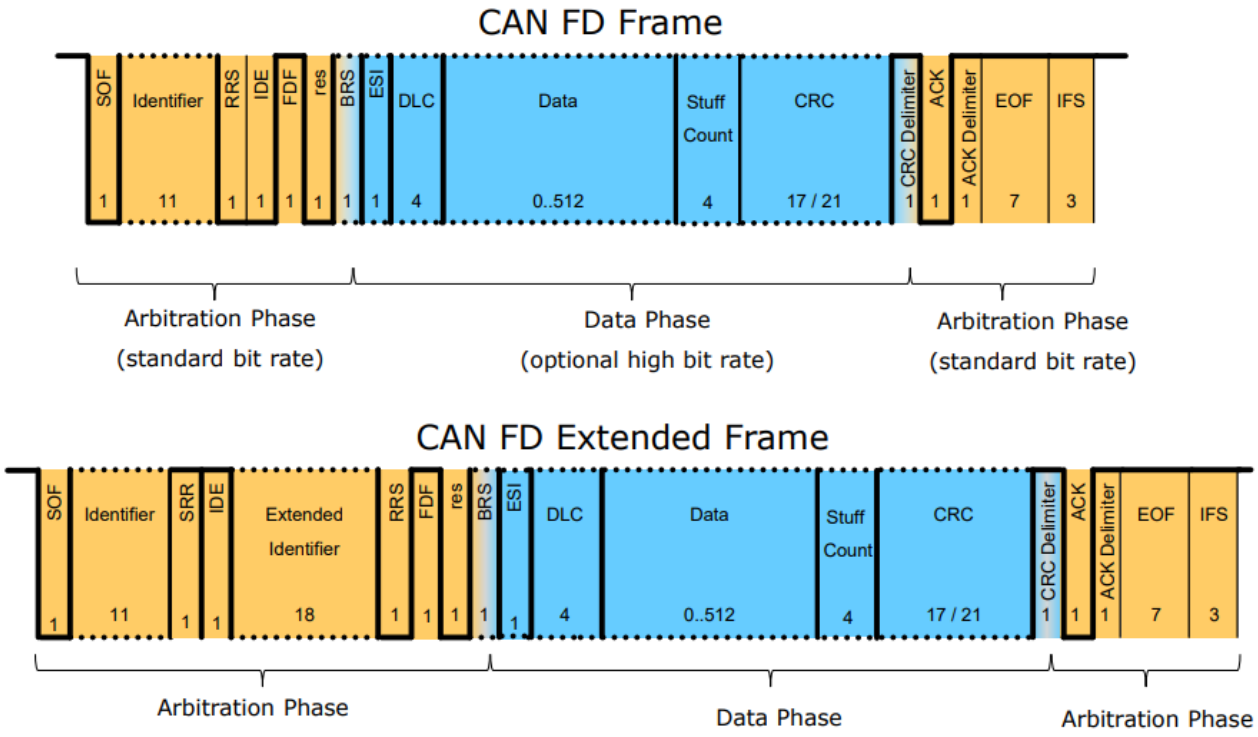


Figure 4.6. CAN-FD frame. Source: [12].

Fig. 4.6 shows the seven fields which compose the CAN-FD frame: Start of Frame (SOF), Arbitration field, Control Field, Data Field, CRC field, ACK field and End of Frame (EOF).

CAN-FD frame includes another type of bit, which appears when there are more than 5 bits in a row at the same level. This bit is usually called a stuff-bit and add an extra bit in the frame with inversed polarity to ensure that level changes, also has the utility to synchronise the sample point [13].

4.2.1.1. Start Of frame and arbitration Field

Start Of Frame: Points out the beginning of the Data frame with a single dominant bit (0). A station is only allowed to start the transmission when the bus is in an idle state.

Arbitration Field: The structure of the Arbitration Field is different for base format and extended format frames.

In base format, the Arbitration Field consists of the basic identifier and the RRS bit. The base identifier is 11 bits long denoted ID-28…ID 18.

In extended format, the arbitration field consists of the extended identifier, the SRR bit, the

IDE bit and the RRS (or r1) bit. As shown in Fig. 4.7 the first section is the base identifier and the second is the identifier extension.

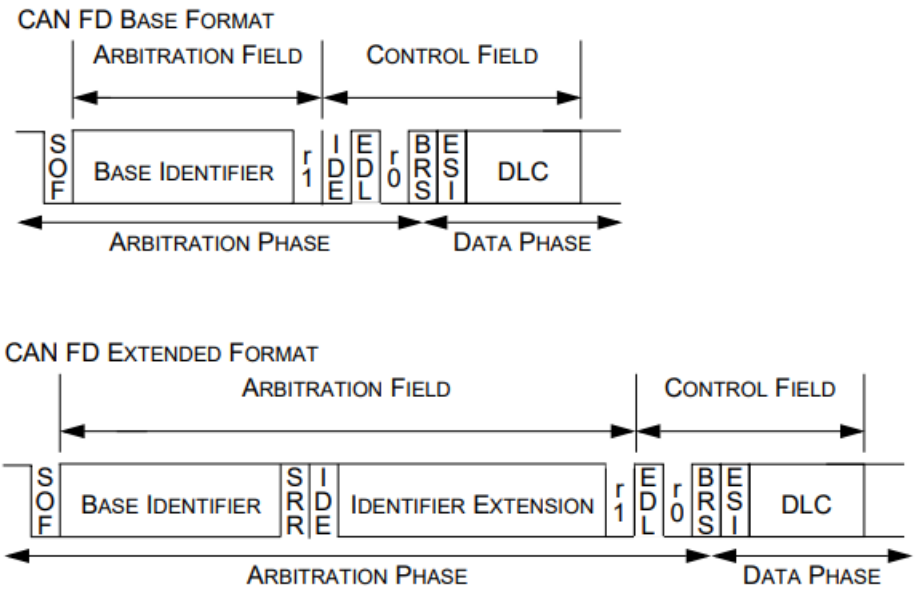


Figure 4.7. CAN-FD Arbitration and control field. Source: [14].

In the arbitration phase, it is also decided which message will prevail when several are sent simultaneously. When that happens the most dominant identifier will be the one to remain, in other words, the one with a lower identifier.

The frame of this phase is mostly the same as in standard CAN, with the exception that RTR (Remote Transmission Request) is changed to RRS.

The meaning of the arbitration phase bits is:

- SRR (Substitute Remote Request): This bit is always recessive (1) and it prevents a 29-bit message from having a higher priority than an 11-bit.
- IDE (Identifier extension flag): it is part of the arbitration field in the extended format and part of the control field in the base format. The bit is transmitted dominant (logical 0) in the base format, whereas in the other recessive (logical 1).
- RRS, RTR or r1 bit: This bit in CAN-FD has no real use, and it is always a dominant reserved bit (logical 0). However, standard CAN was used to request a remote transmission, which is not possible in CAN-FD.

4.2.1.2. Control field and Data field

Control field: The structure of this field is different for the base and extended format. Also, it differs from standard CAN including EDL, BRS and ESI bits.

In base format, the control field consists of the bits of IDE, FDF, res, BRS, ESI and 4 DLC bits. Otherwise, the extended format is similar to the base one, with the only difference being that the IDE bit was already put in the arbitrary part.

The explanation of the use of the bits that are part of the control field is defined below:

- FDF (FD format): Distinguishes between CAN and CAN-FD. A dominant bit (0) means that the message is sent in CAN and a recessive bit that is in CAN-FD.
- Res: Reserved bit let for future applications. It is always dominant (0).
- BRS (Bit rate Switch): Allows CAN-FD frames to be sent at a higher bit rate. If the bit is dominant (0) the bit rate would not be modified, in other words, would be the same that in the arbitration field. However, if the bit is recessive (1) the bit rate from now to the DRC delimiter bit would be increased.
- ESI (Error State Indicator): This bit acts as a flag that is normally set dominant (0). If the CAN-FD frame sender becomes error-passive, this bit will change to recessive (1).
- DLC (Data length code): These 4 bits indicates the number of bytes in the data field. The next table shows the codes to express the number of data bytes.

	Number of Data Bytes	Data Length Code			
		DLC3	DLC2	DLC1	DLC0
Codes in CAN and CAN FD Format	0	0	0	0	0
	1	0	0	0	1
	2	0	0	1	0
	3	0	0	1	1
	4	0	1	0	0
	5	0	1	0	1
	6	0	1	1	0
	7	0	1	1	1
CAN Format	8	1	0/1	0/1	0/1
Codes in CAN FD Format	8	1	0	0	0
	12	1	0	0	1
	16	1	0	1	0
	20	1	0	1	1
	24	1	1	0	0
	32	1	1	0	1
	48	1	1	1	0
	64	1	1	1	1

Figure 4.8. CAN-FD DLC codes. Source: [14].

Data field: It is the field that contains the data that is going to be transmitted, and it could allow up to 64 bytes. The bits of the byte are sent as MSB first.

4.2.1.3. CRC field

CRC field: CRC (Cyclic redundancy check) includes the Stuff count (4 bits), CRC sequence and the recessive CRC delimiter bit.

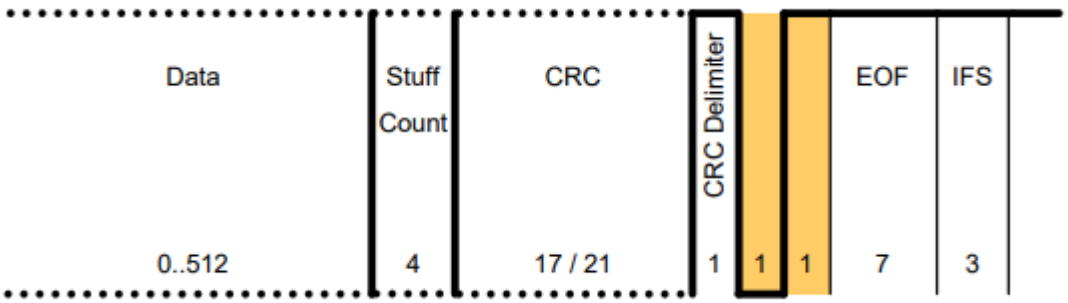


Figure 4.9. CAN-FD CRC field. Source: [12].

Stuff-count are 4 bits used to detect single failures under all conditions. The first 3 bits corresponds to the number of stuff bits that had appeared during the frame, they are presented in Gray-coding format. The last bit of stuff count is a parity bit, which takes a dominant value (0) if it is not pair and recessive (1) if it is.

The CRC has three different sequences named CRC_15, CRC_17 and CRC_21 sequences, which are used depending on certain situations. The first, CRC_15, is used for CAN frames, the second when the frame in CAN-FD has data up to 16 bytes and the last one when the data field is longer than 16 bytes.

Even though these three sequences are generated using different equations, all of them result in a Hamming distance of 6, meaning that they can detect up to 5 randomly distributed errors. These equations are shown in the following image.

- CRC_15 0xC599 $(x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1)$
  = $(x+1)\cdot(x^{7}+x^{3}+1)\cdot(x^{7}+x^{3}+x^{2}+x+1)$
- CRC_17 0x3685B $(x^{17}+x^{16}+x^{14}+x^{13}+x^{11}+x^{6}+x^{4}+x^{3}+x^{1}+1)$
  = $(x+1)\cdot(x^{16}+x^{13}+x^{10}+x^{9}+x^{8}+x^{7}+x^{6}+x^{3}+1)$
- CRC_21 0x302899 $(x^{21}+x^{20}+x^{13}+x^{11}+x^{7}+x^{4}+x^{3}+1)$
  = $(x+1)\cdot(x^{10}+x^{3}+1)\cdot(x^{10}+x^{3}+x^{2}+x^{1}+1)$

Figure 4.10. CRC equations. Source: [14].

The method of calculating the CRC sequence begins at the start of the frame where each sequence is calculated in all nodes, and the node that wins the arbitration phase sends the CRC sequence selected by the values of the frames FDF and DLC. The receivers shall consider the selected CRC polynomial to check for a CRC-Error.

The bit string used to calculate CRC consists of the SOF, Arbitration Field, Control Field, Data field, if it is present, and supplemented with CRC bits of value ‘0’. However, in CAN-FD format the stuff bits are included in the relevant bitstream for CRC calculation.

The polynomial formed by the bitstream is divided (calculated modulo-2) by the generator-polynomial (e.g.,0xC599, 0x3685B or 0x302899). The remainder of this polynomial is the CRC sequence transmitted over the bus.

The last bit of the CRC field is the delimiter whit a recessive signal (1). It is usually transmitted as 1 bit, but because of the phase shift between nodes, it accepts up to 2-bit times [12].

4.2.1.4. ACK field and End of Frame

ACK field: Acknowledge field contains two bits, the ACK slot and the ACK delimiter. In this field, the bit rate returns to the normal value if a higher one was set in the control field. Because of that change, one additional bit-time tolerance is allowed both before and after the edge from recessive to dominant. In other words, the protocol accepts one additional bit before and one after when passing from CRC delimiter to ACK slot.

The ACK slot is the receiver response that has received a valid message, setting a dominant bit (0). The method used to validate the message is by comparing the CRC sequence that the receiver has calculated with the one calculated by the transmitter.

The last bit of this field is the ACK delimiter, which is nothing more than one recessive bit (1).

End of Frame field: This is the last field that contains the End of Frame (EOF) and the Inter-Frame Spacing (IFS).

The End of Frame is a group of seven recessive bits (seven logical 1) that delimitates the ACK bits with the IFS bits.

With the same line, IFS are the minimum space between 2 CAN or CAN-FD frames and consists of 3 recessive bits (1).

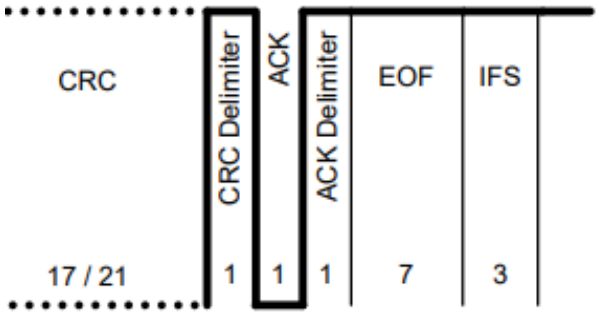


Figure 4.11. CAN-FD End of Frame field. Source: [12].

5. SPI and CAN-FD files

This project had used different drivers to implement both SPI and CAN-FD protocols. Most of these were provided by Microchip, which is the same manufacturer as the MCP2517 FD controller. Even though, some drivers had been modified to fit with the Black-Box application.

These drivers are separated into header and source files, where the first are declarations and macro definitions while the other defines the functions declared on the header file.

Due to both source files being compiled together with the main file, they do not need to be included as the header files.

The name of the files used in this project are shown in the next table:

Header files (.h)	Source files (.c)
drv_CAN-FDspi_api.h	drv_CAN-FDspi_api.c
drv_CAN-FDspi_defines.h	spi.c
drv_CAN-FDspi_register.h	Sigfox.c
spi.h	

Table 5.1. Source and header files. Source: Own.

The driver named *spi* (*spi.h* and *spi.c*) is the one that defines the SPI transfer data. Its header file defines and declares constants and functions, meanwhile, the source file defines the functions that the protocol needs.

Since it was already detailed in chapter 4.1, it will not be explained in this chapter.

5.1. CAN-FD SPI files

As mentioned before, to make possible the communication using CAN-FD protocol some header and source files have been used. Those files were originally designed by Microchip, which is the same manufacturer as the CAN-FD controller. Even though, because the TD1204 EVB has no SPI implemented by hardware, some files have been modified to work correctly.

Each of the different CAN-FD drivers is focused on defining some specific things, because of that, there is one related to the registers of the controller, another with the object definitions, such as the configuration of the FIFO channel for transmission messages, and another one which defines the CAN-FD functions used in the main file.

5.1.1. drv_CAN-FDspi_register.h

The first file named *drv_CAN-FDspi_register.h* is a header that has the purpose of defining the MCP2517 FD registers. It could be said that this file is nothing more than listing the datasheet of the controller in a code format.

The declarations of this file involve:

- SPI instruction set.
- Register and RAM addresses.
- Structure of registers.
- Registers to reset values.

5.1.2. drv_CAN-FDspi_defines.h

The second header file is the *drv_CAN-FDspi_defines,* which includes two very different sections, the implementation and object definitions.

The implementation section is where depending on the chosen device or revision, it defines the number of implemented filters and FIFOs, transmit queue, the maximum size of TX/RX object and data bytes in messages.

The object definition section, in contrast, involves the things related to the message transmission and reception, such as channels, operational modes, events, CAN-FD framework and a large etcetera.

5.1.3. drv_CAN-FDspi_api.h and drv_CAN-FDspi_api.c

The last library used to provide CAN-FD is the one that contains *drv_CAN-FDspi_api.h* and *drv_CAN-FDspi_api.c*. Both of them are also developed by Microchip and they are the

application programming interface (API), which provide the functions used to communicate via CAN-FD.

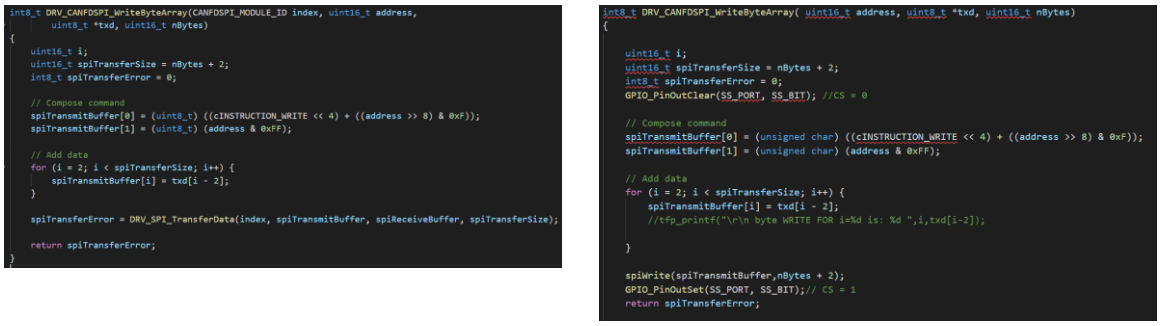
The definition of the API file could be that it is such as an upper layer where the other files (with exception of the main Sigfox file) are included to provide more sophisticated functions.

Some functions included in the API are:

- Reset
- Write and read bytes/half-words (2 bytes) / words (4 bytes) byte array/word array
- ECC enable and initialization of RAM
- CAN-FD frame, transmit and receive channel, filter and mask configuration, and a large etcetera.

However, the original file was dedicated for the PIC32MX470, which considers that the SPI was implemented via hardware and not by software. For that reason, the file had needed some fixes in the write and read instructions.

Here a comparison between the original and modified write byte array instruction is shown:



```
int8_t DRV_CANFDSPI_WriteByteArray(CANFDSPI_MODULE_ID index, uint16_t address,
        uint8_t *txd, uint16_t nBytes)
{
    uint16_t i;
    uint16_t spiTransferSize = nBytes + 2;
    int8_t spiTransferError = 0;

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

    // Add data
    for (i = 2; i < spiTransferSize; i++) {
        spiTransmitBuffer[i] = txd[i - 2];
    }

    spiTransferError = DRV_SPI_TransferData(index, spiTransmitBuffer, spiReceiveBuffer, spiTransferSize);

    return spiTransferError;
}
```

```
int8_t DRV_CANFDSPI_WriteByteArray( uint16_t address, uint8_t *txd, uint16_t nBytes)
{

    uint16_t i;
    uint16_t spiTransferSize = nBytes + 2;
    int8_t spiTransferError = 0;
    GPIO_PinOutClear(SS_PORT, SS_BIT); //CS = 0

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

    // Add data
    for (i = 2; i < spiTransferSize; i++) {
        spiTransmitBuffer[i] = txd[i - 2];
        //tfp_printf("\r\n byte WRITE FOR i=%d is: %d ",i,txd[i-2]);

    }

    spiWrite(spiTransmitBuffer,nBytes + 2);
    GPIO_PinOutSet(SS_PORT, SS_BIT);// CS = 1
    return spiTransferError;
```

Figure 5.1. Changes between original and modified writeByteArray function. Source: Own.

As you can see the difference are not noticeable at all, but the new one uses some functions included in the SPI library to transmit and receive data using SPI ports, while the original one only the *DRV_SPI_TransferData* instruction.

Moreover, another difference is that in the modified one, the chip select (CS) has to be set to 1 after the instruction ends.

5.2. Sigfox file (main)

Sigfox file is the main source file of the project and it is the one that implements all the functionalities that the Black-Box will have, from the configuration of the controller to the management of the received data.

The file is divided into two big differenced sections, the user setup and the user loop.

5.2.1. User setup

User setup, as its name indicates, is the section where the desired configuration is set, most of the configuration involves modifying registers from MCP2517 FD to communicate properly using CAN-FD protocol. An important part of this part of code had been got it from the TFG by Javier A. De Esteban Garau named "*Registrador de CAN-FD utilizando lenguaje C y una Raspberry PI"*. Even though, some changes have been done to fit with the project [15].

The different parts of this function are listed in the following sections.

5.2.1.1. UART Configuration

Opening the UART port in itself is not mandatory, but it helps a lot in terms of development. The main utility is to show the outputs of the microprocessor in the same computer, allowing to flash the program instead of debugging.

```
TD_UART_Options_t options = {LEUART_DEVICE, LEUART_LOCATION, 9600, 8, 'N',1,
 false};
TD_UART_Open(&options, TD_STREAM_RDWR);
```

Figure 5.2. UART Configuration. Source: Own.

5.2.1.2. Initialize SPI and reset MCP2517 FD

The first own instruction of the program is the one that initializes the SPI. The importance of

doing that is to secure that will not appear problems in SPI instructions when running the program again.

Initializing SPI means configuring the GPIOs named SS, SCK, MOSI and MISO and, as mentioned before, is a must to assure the communication between the microprocessor ARM Cortex M3 and the controller MCP2517 FD.

Though the SPI had been initialized, it is needed to reset the CAN-FD controller before start modifying the corresponding registers. That instruction not only sets to default all its registers but also changes the state of the device to configuration, a mandatory mode for configuring FIFO channels and message objects.

```
init_SPI();
DRV_CAN-FDSPI_Reset();
```

Figure 5.3. Initialization and reset MCP2517 FD. Source: Own.

5.2.1.3. Error Correcting Code and initialize RAM

ECC or error-correcting code is a technology implemented in the RAM used to check and correct errors, protecting against undetected memory data corruption. This functionality is especially worth in critical applications such as the Black-Box, and for this reason, it is activated.

On the other hand, RAM is initialized here to ensure that it does not pull unwanted data from the previous run of the program.

```
DRV_CAN-FDSPI_EccEnable();
DRV_CAN-FDSPI_RamInit(0x00);
```

Figure 5.4. ECC enable and initialize RAM. Source: Own.

Here, as you can see, it is first enabling the ECC and then putting all the bits of all RAM addresses to 0.

5.2.1.4. CAN control register configuration

This part consists of modifying the CAN control register (CiCON) to:

- Enable ISO CRC in CAN frames, which includes stuff bits on the CRC field and use

Non-Zero CRC initialization.

- Do not save transmitted messages in transmit event FIFO bit (TEF).
- Do not reserve space in RAM for Transmit Queue bit (TXQ).

```
CAN_CONFIG config;
DRV_CAN-FDSPI_ConfigureObjectReset(&config);
config.IsoCrcEnable = ISO_CRC;
config.StoreInTEF = 0;
config.TXQEnable = 0;
DRV_CAN-FDSPI_Configure(&config);
```

Figure 5.5. CAN control register configuration. Source: Own.

At first, this code defines config as a *CAN_CONFIG* data type, which is nothing more than a variable with the same structure as the *CiCON* register.

Secondly, the code uses a function named *DRV_CAN-FDSPI_ConfigureObjectReset,* which copies the default state of the *CiCON* register bits to that variable.

After that, it changes the state of some members of the variable and finally only last to put those changes inside the register of the controller. The instruction that does that is the *DRV_CAN-FDSPI_Configure*.

5.2.1.5. Transmitted objects (TX) and received objects (RX) configuration

FIFO channels are used to receive and transmit messages, and for this reason, it has to be modified according to our preferences. Even though, before explaining how was done it would be great to briefly explain what FIFO channels are.

FIFO channels (First In First Out) are a type of channel where the first entry is processed first. These can be configured to send (TX) or receive (RX) messages, and since in CAN-FD we need both things we will need at least 1 of each type.

For a TX FIFO, the User Address points to the address in RAM where the data for the next transmit message shall be stored, meanwhile, for the RX FIFO, the User Address points to the address in RAM where the data of the next received message shall be read [6].

The register that involves the configuration of the TX and RX FIFO channel is the *CiFIFOCON*.

Some changes have been applied in TX FIFO from the default values, some of them were setting the FIFO size to 8 bytes, the payload size to 64 bytes (the same as the maximum bytes that CAN-FD could transmit) and setting the transmit priority bits of the message to the lowest state, which is TX priority of 0.

Similarly, RX FIFO applied the same changes as the TX FIFO, but instead of modifying the transmit priority bits to 0, it sets the capture timestamp of the received message object in RAM, meaning that now it captures the time stamp between two messages in a row.

```
#define APP_TX_FIFO CAN_FIFO_CH2
#define APP_RX_FIFO CAN_FIFO_CH1
CAN_TX_FIFO_CONFIG txConfig;
DRV_CAN-FDSPI_TransmitChannelConfigureObjectReset(&txConfig);
txConfig.FifoSize = 8;
txConfig.PayLoadSize = CAN_PLSIZE_64;
txConfig.TxPriority = 0;
DRV_CAN-FDSPI_TransmitChannelConfigure(APP_TX_FIFO, &txConfig);
```

Figure 5.6. Transmitted objects configuration. Source: Own.

```
CAN_RX_FIFO_CONFIG rxConfig;
DRV_CAN-FDSPI_ReceiveChannelConfigureObjectReset(&rxConfig);
rxConfig.FifoSize = 8;
rxConfig.PayLoadSize = CAN_PLSIZE_64;
rxConfig.RxTimeStampEnable = 1;
DRV_CAN-FDSPI_ReceiveChannelConfigure(APP_RX_FIFO, &rxConfig);
```

Figure 5.7. Received objects configuration. Source: Own.

As it can be seen, the biggest difference between transmit channel and the received channel is that one uses the argument *APP_TX_FIFO* while the other the *APP_RX_FIFO*.

In the same way, both codes share the same procedures as the code used to configure the CAN control register. The steps to do that are:

- Defining a variable with the members of the register.
- Put that variable to the default values.
- Change the desired members of the variable.
- Set the register with those changes.

5.2.1.6. Configure filter and mask

Filter and mask are used to reduce the valid number of IDs in CAN-FD. That functionality is especially useful when it is only wanted to receive data from some specific devices.

Talking more accurately about filters and masks, it can be said that the first compares the value of its bits with the ID, while the other says which bits have to be compared. The following picture complements this explanation with a graphic that shows how they work.

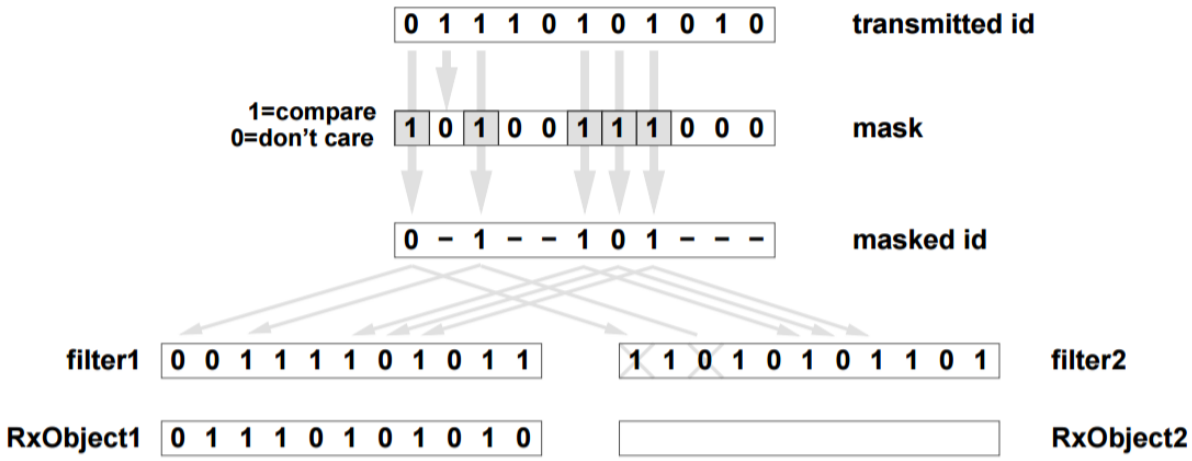


Figure 5.8. Filter and mask graphical explanation. Source: [16].

In this case, filters and masks could be implemented via software but it was decided to implement them via hardware to save computational resources, as the MCP2517 FD contains the appropriate registers to do that by hardware.

Regarding the filter, it is configurated using the *CiFLTOBJ* register where most of its bits are related to the ID we want to give it.

```
#define filter0 CAN_FILTER0
CAN_FILTEROBJ_ID filterConfig;
filterConfig.SID=0x11;
filterConfig.EID=0;
filterConfig.SID11=0;
DRV_CAN-FDSPI_FilterObjectConfigure(filter0,&filterConfig);
```

Figure 5.9. Filter configuration. Source: Own.

The previous code shows that *filter0* is configured with a standard ID that has a value of 0x11. The function *DRV_CAN-FDSPI_FilterObjectConfigure* is the one that has been used to change the register of the filter.

As the filter, the implementation of the mask has been done using a specific resister called *CiMask*, which is the configuration mask register. The following code shows how it was configured and as you will see it has been only put an ID of 1 to only compare the LSB of the ID and filter, but it could be changed by putting another number.

```
CAN_MASKOBJ_ID maskConfig;
maskConfig.MSID=1;
maskConfig.MEID=0;
maskConfig.MSID11=0;
maskConfig.MIDE=0;
DRV_CAN-FDSPI_FilterMaskConfigure(filter0,&maskConfig);
```

Figure 5.10. Mask configuration. Source: Own.

As you can see, the register of the mask is practically the same as the filter where most of its bits are equal.

Finally, the last thing to do is to link the filter with the FIFO channel. To do that it is only needed to write the next instruction, which involves the *CiFLTCON* (filter control register).

```
DRV_CAN-FDSPI_FilterToFifoLink(CAN_FILTER0, APP_RX_FIFO, true);
```

Figure 5.11. Filter linking to a channel. Source: Own.

5.2.1.7. Setup bit time

This section involves the setup of the bit rates for the two different velocities that CAN-FD allows. These different rates are named Arbitration and Data, and each of them defines the bit transmission velocity of the different bits of the frame.

Arbitration rate is bit time which contains the bits from the start of the frame to BRS (Bit Rate Switch) and from the ACK slot until the end. The maximum bits transmitted per second that can reach is the same as in the standard CAN which is 1 Mbit/s, the register in charge is the nominal bit time configuration (CiNVTCFG).

The other rate that CAN-FD could have is the one called data. This rate contains the bits from the ESI to DRC delimiter. The data rate, in contrast to the arbitration one, permits higher velocities reaching up to 5 Mbit/s, which is the same as saying 5 velocities up to 5 times higher than the arbitration rate. The register in charge of the data rate is the data bit time configuration (CiDBTCFG).

```
CAN_BITTIME_SETUP selectedBitTime = CAN_250K_2M;//CAN_500K_2M;
DRV_CAN-FDSPI_BitTimeConfigure(selectedBitTime, CAN_SSP_MODE_AUTO,
CAN_SYSCLK_40M);
```

Figure 5.12. Setting the bit time. Source: Own.

Here, as it can be seen, the bit rate is configured for 250 Kbits/s in the arbitration section and 2 Mbits/s in the data section.

5.2.1.8. Operational Mode

The last part of the *user_setup* function is the one that changes the operational mode from configuration to external loopback mode.

The main reason to use external loopback mode instead of the others is that it treats its own transmitted messages as received messages, and also stores them (if pass acceptance filtering) in the RX FIFO. This operational mode is especially interesting for self-test, being incredibly useful in detecting troubleshooting.

```
DRV_CAN-FDSPI_OperationModeSelect(CAN_EXTERNAL_LOOPBACK_MODE);
```

Figure 5.13. Operational mode select. Source: Own.

5.2.2. User loop

The user loop is the section in charge of transmitting, receiving and subsequently managing messages. As before, it has been divided into different parts to ease the comprehension of how it works.

5.2.2.1. Message configuration

The message configuration is the part that sets the CAN-FD to be sent with the desired form and data. As you could imagine, in this section there have only to be put the bits of the frame that are not fixed or are calculated automatically as the CRC.

```
CAN_TX_MSGOBJ txObj;
uint8_t txd[MAX_DATA_BYTES];
txObj.word[0] = 0;
txObj.word[1] = 0;
txObj.bF.id.SID = 0x121;
tfp_printf("\n\r ID of TX is: %x",txObj.bF.id.SID);
txObj.bF.id.EID = 0;

txObj.bF.ctrl.BRS = 1;
txObj.bF.ctrl.DLC = rand()%15;//0xf;
uint8_t DLC_TX;
DLC_TX = DRV_CAN-FDSPI_DlcToDataBytes(txObj.bF.ctrl.DLC);
tfp_printf("\n\r DLC of TX is: %d",DLC_TX);

txObj.bF.ctrl.FDF = 1;
txObj.bF.ctrl.IDE = 0;
txObj.bF.ctrl.SEQ = 0;

uint8_t i;
uint8_t rxd[MAX_DATA_BYTES];
tfp_printf("\n\r The data TX send are: [")
for (i = 0; i < DLC_TX; i++)
{
    txd[i] = 64-i;
    rxd[i] = 0;
    if (i!= DLC_TX -1){
        tfp_printf("%x, ",txd[i]);
        }
    else tfp_printf("%x] ",txd[i]);;

}
```

Figure 5.14. Message configuration. Source: Own.

The code starts declaring txObj as CAN_TXMSGOBJ, which is nothing more than a data type that takes into account the bits of the arbitration and control phase of the CAN-FD frame.

After that, the frame can be defined according to our preferences. In this case, it remains in such a way:

- Set the ID as 0x121, without using the extended format.
- Set to 1 the FDF and BRS bits, making to use CAN-FD and a flexible bit rate.
- Set to 0 the IDE as now the code will not be in extended format.

- Put the sequence bits on 0.
- Set the DLC bit to 64 bytes.
- Write each of the 64 bytes with some data.

5.2.2.2. Message transmission

The message transmission is the part of the code that transmits the CAN-FD frame to the desired channel.

```
CAN_TX_FIFO_EVENT txFlags;
DRV_CAN-FDSPI_TransmitChannelEventGet(APP_TX_FIFO, &txFlags);
while (!(txFlags & CAN_TX_FIFO_NOT_FULL_EVENT));
uint8_t n;
int8_t code;
n = DLC_TX;
code = DRV_CAN-FDSPI_TransmitChannelLoad(APP_TX_FIFO, &txObj, txd, n, true);
```

Figure 5.15. Message transmission. Source: Own.

The listed code starts defining a variable that takes the event in the TX FIFO channel, to ensure that the program is not sending messages when the FIFO channel is full.

Later this code unifies all the frames, decides which addresses of RAM has to use, and after that, it transmits the message.

5.2.2.3. Message reception

The last part of the base program is the message reception, and as mentioned before, using the external loopback operation permits reading the messages transmitted by the same device.

```
CAN_RX_MSGOBJ rxObj;
TD_RTC_Delay(T100MS)
CAN_RX_FIFO_EVENT rxFlags;
```

```
code = DRV_CAN-FDSPI_ReceiveChannelEventGet(APP_RX_FIFO, &rxFlags);
if (rxFlags & CAN_RX_FIFO_NOT_EMPTY_EVENT){
        GPIO_PinOutSet(LED_PORT, LED_BIT);
        CAN_RX_MSGOBJ rxObj;
        code = DRV_CAN-FDSPI_ReceiveMessageGet(APP_RX_FIFO, &rxObj, rxd,
MAX_DATA_BYTES);
        int DLC_RX = DRV_CAN-FDSPI_DlcToDataBytes(rxObj.bF.ctrl.DLC);
        tfp_printf("\n\r Message received");
        tfp_printf("\n\r ID of RX is: %x",rxObj.bF.id.SID);//ID
        tfp_printf("\n\r DLC of RX is: %d",DLC_RX);
        tfp_printf("\n\r The data RX received are: [");
        for (i = 0; i < DLC_RX; i++)
            {
            if (i!=DLC_RX-1){
                tfp_printf("%x, ",rxd[i]);
            }
            else tfp_printf("%x] ",rxd[i]);;
            }
        tfp_printf("\n\r End of Message ");

        GPIO_PinOutClear(LED_PORT, LED_BIT);
        // Process rxObj
}
else tfp_printf("\n\r ID do not match with filter and Mask");
```

Figure 5.16. Message reception. Source: Own.

At first, the code looks if there is something in the FIFO RX channel, and if it is, which is the same as saying if it is not empty, it copies the information saved in the corresponding RAM addresses.

The function used to get the data, ID, DLC and others bits from the CAN-FD frame is *DRV_CAN-FDSPI_ReceiveMessageGet*. The required arguments of this function are the channel, variables, where to save the parts of the frame and the maximum bits it can read.

6. Software validation

Software validation is the part of the project in charge of checking if the developed code works correctly. Some of the features that have been tested are:

- SPI instructions were modified in this project to fit with the software implementation.
- Sending and reception of CAN-FD messages using random and non-random values.
- Mask and filters.
- CAN-FD electrical signals.

6.1. SPI instructions to write and read Registers

SPI instructions are used to access Special Function Registers (SFRs) and RAM of the CAN-FD controller MCP2517FD, in other words, they allow to read and write both SFRs and RAM registers.

The main differences between SFR and RAM accesses are that SFR first is byte-oriented and the other is word-oriented (4 bytes). Additionally, it exists another variance which is that to modify most of the SFR registers the device needs to be in the configuration mode.

Regarding SPI instructions, they have mostly the same format and structure, starting with a low level and ending with a high level in the Chip Select (CS) pin. However, all of them share that their command consists of a 4-bit command followed by a 12-bit address.

Name	Format	Description
RESET	C = 0b0000; A = 0x000	Resets internal registers to default state; selects Configuration mode.
READ	C = 0b0011; A; D = SDO	Read SFR/RAM from address A.
WRITE	C = 0b0010; A; D = SDI	Write SFR/RAM to address A.
READ_CRC	C = 0b1011; A; N; D = SDO; CRC = SDO	Read SFR/RAM from address A. N data bytes. Two bytes CRC. CRC is calculated on C, A, N and D.
WRITE_CRC	C = 0b1010; A; N; D = SDI; CRC = SDI	Write SFR/RAM to address A. N data bytes. Two bytes CRC. CRC is calculated on C, A, N and D.
WRITE_SAFE	C = 0b1100; A; D = SDI; CRC = SDI	Write SFR/RAM to address A. Check CRC before write. CRC is calculated on C, A and D.
Legend: C = Command (4 bit), A = Address (12 bit), D = Data (1 to n bytes), N = Number of Bytes (1 byte), CRC (2 bytes)		

Figure 6.1. SPI instruction list. Source: [6].

In the table shown in Fig. 6.1, there are 6 different types of SPI instructions. However, in this project, only 3 had been used because the remaining only provides an extra layer of reliability on the SPI communication.

Those SPI instructions can be found inside some functions of the *DRV_CAN-FDSPI_api* library, and because of that, all of the functions that involve SPI instructions have to be checked.

The functions that involve SPI instructions are:

- *DRV_CAN-FDSPI_WriteByte*
- *DRV_CAN-FDSPI_ReadByte*
- *DRV_CAN-FDSPI_WriteWord*
- *DRV_CAN-FDSPI_ReadWord*
- *DRV_CAN-FDSPI_WriteHalfWord*
- *DRV_CAN-FDSPI_ReadHalfWord*
- *DRV_CAN-FDSPI_WriteByteArray*
- *DRV_CAN-FDSPI_ReadByteArray*
- *DRV_CAN-FDSPI_WriteWordArray*
- *DRV_CAN-FDSPI_ReadWordArray*

The tests done to check if they work correctly are the same that Javier Esteban did in his final degree project. [15]

- Test 1: Read OSCON register (Address 0xE00)

```
uint32_t rxd=0x00;
DRV_CAN-FDSPI_ReadWord(0xE00,&rxd);
tfp_printf("\r\n TEST 1: the register is: %d\r\n ",rxd);
```

Figure 6.2. Read OSCON register code. Source: Own.

- Test 2: Write and read half word (2 bytes)

```
uint16_t rxd1=0x00;
DRV_CAN-FDSPI_WriteHalfWord(0x010,0x5A5A);
tfp_printf("\r\n TEST 2: Half word has been written in address 0x010 with a
value of 0x5a5a");
DRV_CAN-FDSPI_ReadHalfWord(0x010,&rxd1);
tfp_printf("\r\n The half word read is: %x\r\n ",rxd1);
```

Figure 6.3. Write and read half word. Source: Own.

Test 3: Write and read a word (4 bytes)

```
uint32_t rxd2=0x00;
DRV_CAN-FDSPI_WriteWord(0x400, 0x11443322);
tfp_printf("\r\n TEST 3: The word has been written in address 0x400 with a
value of 0x11443322 ");
DRV_CAN-FDSPI_ReadWord(0x400,&rxd2);
tfp_printf("\r\n Word read is: %x\r\n ",rxd2);
```

Figure 6.4. Write and read a word. Source: Own.

Test 4: Write and read a byte

```
uint8_t rxd3=0x00;
DRV_CAN-FDSPI_WriteByte(0x010, 0x98);
tfp_printf("\r\n TEST 4: The byte has been written in address 0x010: 0x98");
DRV_CAN-FDSPI_ReadByte(0x010,&rxd3);
tfp_printf("\r\n Byte read is: %x\r\n ",rxd3);
```

Figure 6.5. Write and read a byte. Source: Own.

Test 5: Write and read a byte array

```
unsigned char i;
uint8_t ip[64];
uint8_t iprx[64];
for (i=0;i<64;i++){
    ip[i]=63-i;//0x3F...0x00
    iprx[i]=0;
}
DRV_CAN-FDSPI_WriteByteArray(0x400,ip,64);ds
tfp_printf("\r\n TEST 5: The byte array written in address 0x400 has the
following values: 0x3F...0x00");
DRV_CAN-FDSPI_ReadByteArray(0x400,iprx,64);
```

```
tfp_printf("\r\n Byte array read are: [ ");
for (i=0;i<64;i++){
    if (i==63) tfp_printf("%x]\r\n  ",iprx[i]);
    else tfp_printf("%x, ",iprx[i]);
}
```

Figure 6.6. Write and read a byte array. Source: Own.

Test 6: Write and read a word array

```
uint8_t j;
uint32_t iprw[64];
uint32_t ipw[64];
for (j=0;j<64;j++){
    ipw[j]=0+j;
    iprw[j]=0;
}
DRV_CAN-FDSPI_WriteWordArray(0x400,ipw,64);
tfp_printf("\r\n TEST 6: Word array written in address 0x500 has the
following values: 0x3F...0x00");
DRV_CAN-FDSPI_ReadWordArray(0x400,iprw,64);
tfp_printf("\r\n Word Array read are: [ ");
for (j=0;j<64;j++){
    if (j==63) tfp_printf("%08x] ",iprw[j]);
    else tfp_printf("%08x, ",iprw[j]);;
}
```

Figure 6.7. Write and read a word array. Source: Own.

Fig. 6.8 shows the results of every test done to check the SPI instructions. As you can see, the results were the desired ones except for test 6, as its message exceeds the maximum data that can be sent using the implemented SPI protocol, which is 255 bytes.

```
TEST 1: the register is: 1120

TEST 2: Half word has been written in address 0x010 with a value of 0x5a5a
The half word read is: 5a5a

TEST 3: The word has been written in address 0x400 with a value of 0x11443322
Word read is: 11443322

TEST 4: The byte has been written in address 0x010: 0x98
Byte read is: 98

TEST 5: The byte array written in address 0x400 has the following values: 0x3F.
..0x00
Byte array read are: [ 3f, 3e, 3d, 3c, 3b, 3a, 39, 38, 37, 36, 35, 34, 33, 32,
31, 30, 2f, 2e, 2d, 2c, 2b, 2a, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 1f, 1e,
1d, 1c, 1b, 1a, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, f, e, d, c, b, a, 9, 8,
7, 6, 5, 4, 3, 2, 1, 0]

transmitted information has more than 255 bytes, which is the maximum: your mes
sage has 64 words, which is the same as 258 bytes
TEST 6: Word array written in address 0x500 has the following values: 0x3F...0x
00
Word Array read are: [ 00000000, 00000001, 00000002, 00000003, 00000004, 000000
05, 00000006, 00000007, 00000008, 00000009, 0000000a, 0000000b, 0000000c, 000000
0d, 0000000e, 0000000f, 00000010, 00000011, 00000012, 00000013, 00000014, 000000
15, 00000016, 00000017, 00000018, 00000019, 0000001a, 0000001b, 0000001c, 000000
1d, 0000001e, 0000001f, 00000020, 00000021, 00000022, 00000023, 00000024, 000000
25, 00000026, 00000027, 00000028, 00000029, 0000002a, 0000002b, 0000002c, 000000
2d, 0000002e, 0000002f, 00000030, 00000031, 00000032, 00000033, 00000034, 000000
35, 00000036, 00000037, 00000038, 00000039, 0000003a, 0000003b, 0000003c, 000000
3d, 0000003e, 00000000]
```

Figure 6.8. Results of all tests. Source: Own.

The equation that relates the bytes used in the functions is represented below. As you can see for 64 words, the maximum data that can be sent is surpassed and because of that there were only read the first 63 words in test 6.

$$64\ words * 4\ \frac{bytes}{word} + 2\ bytes\ (from\ 4\ bit\ SPI\ command\ +\ 12\ bit\ address) = 258\ bytes > 255\ bytes$$

Equation 6.1. The number of bytes calculation for SPI instructions.

6.2. Message Checking

Once probed that the SPI instructions work correctly, the message transmission can be checked and validated too. To do that, the code shown in both parts of chapter 7.2, *TD_USER_Setup* and *TD_USER_Loop*, were flashed inside the microprocessor.

Here, the loopback mode takes importance because it allows checking if the transmitted message were the same as the one received for the same device.

The parameters that have been used to check the message were the ID, DLC and data. Additionally, masks and filters have been also proved for different configurations.

Test 1: When Filter is set to 0x121, the mask to 0x7FF and the ID=0x121.

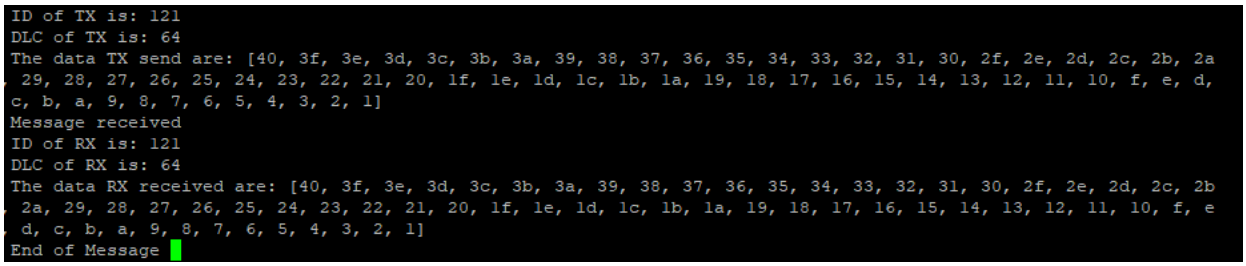


Figure 6.9. Results of test 1. Source: Own.

With this configuration the filter and mask allow to receive the transmitted message and, as you can see, every of the compared parameters matches. Notice that in DLC a value of 64 corresponds to a 0xF.

Test 2: When Filter is set to 0x121, the mask to 0x7FE and the ID=0x120.

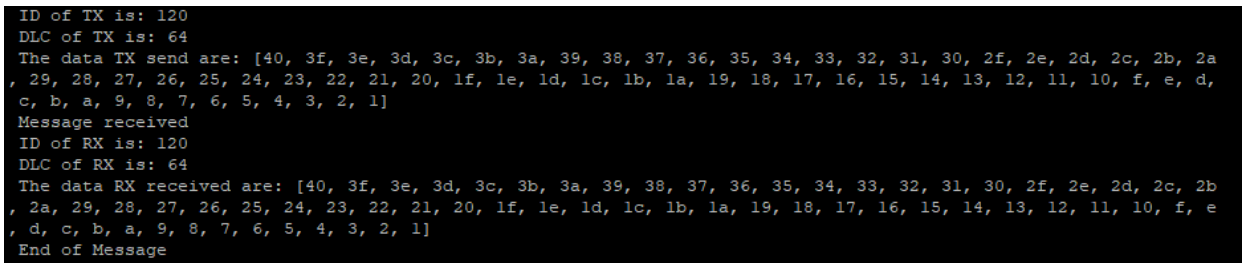


Figure 6.10. Results of test 2. Source: Own.

In this case, the message is accepted by the filter because the mask is not comparing the least significant bit (LSB), which is the only one that differs between filter and ID.

Test 3: When Filter is set to 0x121, the mask to 0x7FF and the ID=0x120.

```
 ID of TX is: 120
 DLC of TX is: 64
 The data TX send are: [40, 3f, 3e, 3d, 3c, 3b, 3a, 39, 38, 37, 36, 35, 34, 33,
32, 31, 30, 2f, 2e, 2d, 2c, 2b, 2a, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 1f,
1e, 1d, 1c, 1b, 1a, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, f, e, d, c, b, a, 9,
 8, 7, 6, 5, 4, 3, 2, 1]
 ID do not match with filter and Mask
```

Figure 6.11. Results of test 3: Own.

The last test is the one where the ID and filter identifications differ in 1 bit and the mask compares all bits. In this case, the message could not be received as it sees that the bits that have to be compared between filter and ID do not match at all.

6.3. Message checking using random parameters

It is worth validating if the program works correctly when random parameters are used, as in this way we can check how the Black-Box will act. The parameters that have been randomized are the ID, DLC and data because they are the ones that change more in a real communication using CAN-FD.

The first premise is that the ID will be standard (11 bits) but its value will be completely random. The reason is that using an extended format or not, there will not cause any significant change in the results and conclusions.

The second premise is that DLC will be random for any integer between 0 and 0xf (0-15).

The third premise is that the data will be random for both data values and the number of bytes sent for any integer between 0 and 63.

The last premise is that the mask only would take into account the least significant bit to not extend too much the time to receive a message allowed to read.

The next code shows the part that had been changed from the base Sigfox file, the other parts remain the same, as they are shown in chapter 7.2.

```
txObj.bF.id.SID = rand()%2047;//0x121;  //id between 2^11 and 0
txObj.bF.ctrl.DLC = rand()%15;
int maxrand=rand()%64;
tfp_printf("\n\r The number of bytes of Data is: %d ",maxrand);
tfp_printf("\n\r The data TX send are: [");
```

```
for (i = 0; i < maxrand; i++)
    {
    txd[i] = rand()%63;
    rxd[i] = 0;
    if (i!=maxrand-1){
        tfp_printf("%x, ",txd[i]);
    }
    else tfp_printf("%x] ",txd[i]);;
}
```

Figure 6.12. Changes in message configuration to put random. Source: Own.

The utility of doing that is to check if that program works with any conditions and what happens when it is tried to send more bytes than the DLC allows or vice versa.

6.3.1. Trying to send more data bytes than DLC

Using the random seed of 12, *srad (12)*, the program gets an example of what happens when the device is trying to send more data than the DLC could provide.

```
ID of TX is: 4d
DLC of TX is: 3
The number of bytes of Data is: 24
The data TX send are: [9, 2e, 5, 20, 1, b, 6, 2e, 1b, 3, 18, 3, 1c, 39, 39, 11, 30, 34, c, 3b, 3b, 36, 3a, b]
Message received
ID of RX is: 4d
DLC of RX is: 3
The data RX received are: [9, 2e, 5]
End of Message
```

Figure 6.13. Results when sending more data bytes than DLC. Source: Own.

As shown in Fig. 6.13, the bytes of received data are reduced to fit with the DLC.

6.3.2. Trying to send fewer data bytes than DLC

The other case is what happens when it sent fewer data bytes than the ones that have the DLC. The seed used in the *srand* function for this test had been the 10th.

```
ID of TX is: 47
DLC of TX is: 12
The number of bytes of Data is: 8
The data TX send are: [17, 1a, 1a, 2c, 33, 8, 9, 1b]
Message received
ID of RX is: 47
DLC of RX is: 12
The data RX received are: [17, 1a, 1a, 2c, 33, 8, 9, 1b, 0, 0, 0, 0]
End of Message
```

Figure 6.14. Results when sending fewer bytes than DLC. Source: Own.

As shown in Fig. 6.14, the remaining bytes are read as 0 in the output of the microprocessor and as expected it does not produce any error in the communication.

6.4. CAN-FD signal

Last but not least, is important to check if there exists a real electrical signal when a CAN-FD frame is transmitted. Because of that, an oscilloscope has been used as it permits sampling both high and medium frequency signals, which are the ones that influence this communication protocol.

The done tests were very simple as they only consist of sending CAN-FD frames in different bit rates using the BRS (bit rate switch) and sampling the results in the oscilloscope. However, even if this test is simple, it is extremely valuable to look now if an electrical signal exists because having bad wiring or a broken component is more common than it seems. Additionally, detecting where the problem is later would be more problematic as more ECUs would be connected to the system.

For these reasons, it has been done 2 tests using different bit rates.

- Arbitration rate of 250 Kbit/s and data rate of 2 Mbits/s:

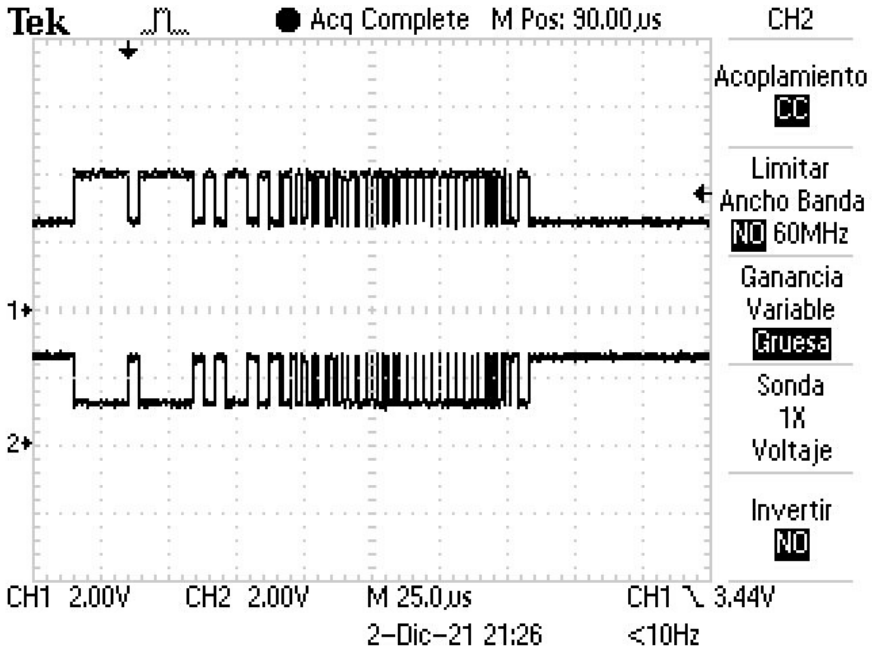


Figure 6.15. CAN FD signal for an arbitration rate of 250Kbit/s and a data rate of 2 Mbit/s. Source: Own.

As you can see both cables are differential wired-AND signals that allow only 2 states, the dominant and recessive. Moreover, both frequencies and voltage match for both bit rates.

- Arbitration rate of 1 Mbit/s and data rate of 8 Mbits/s:

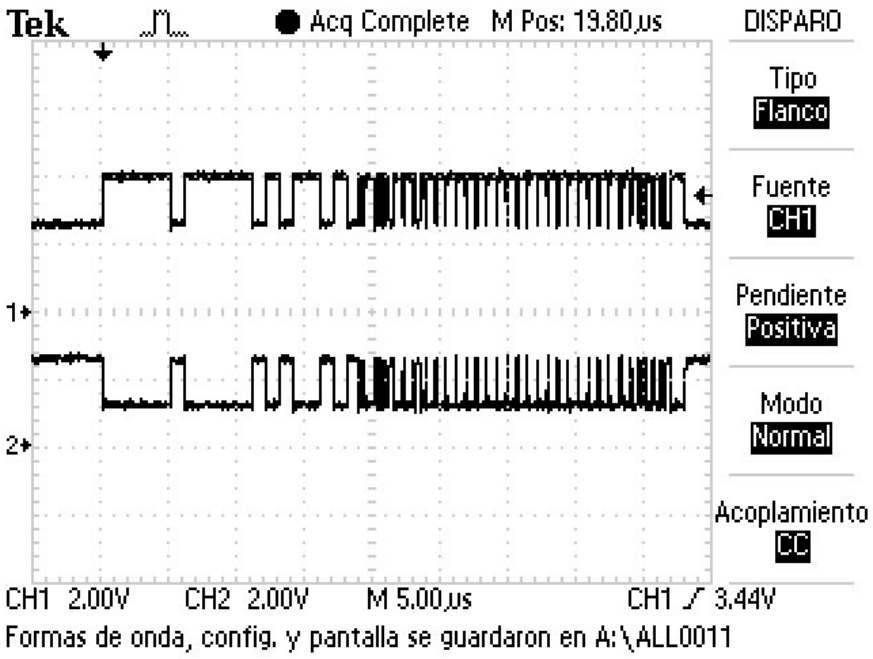


Figure 6.16. CAN FD signal for an arbitration rate of 1Mbit/s and a data rate of 8 Mbit/s. Source: Own.

This last test has been gone correctly even when the data rate was bigger than the maximum allowed for CAN-FD protocol. And because of that, we can validate that the messages are being sent correctly in terms of electrical signals.

7. Improving the main file

There is a need to improve the base main file to make the sum of components act as a real Black-Box. With the aim of that one of the main functions that the device should have is to receive external messages and send warmings to the Sigfox back-end according to the content of that message.

7.1. Modifications of the main file to receive external messages

It is important to modify the program to accept external messages because if not the Black-Box could not communicate with other devices, and as you might imagine, this is not acceptable.

The changes that had been applied to do that are:

- Removing the part of transmitting message and message data configuration.
- Removing the part of sending the message.
- Providing an infinity loop that checks if the FIFO channel is not empty.

Now *TD_USER_Loop* looks like:

```
uint32_t coutner;
while(1){
    ConfigMessage();
    //RX
    TD_RTC_Delay(T500MS);
    CAN_RX_FIFO_EVENT rxFlags;
    int8_t code = DRV_CAN-FDSPI_ReceiveChannelEventGet(APP_RX_FIFO,
&rxFlags);
    if (rxFlags & CAN_RX_FIFO_NOT_EMPTY_EVENT)
    {
    CAN_RX_MSGOBJ rxObj;
    uint8_t rxd[63]={0};
    code = DRV_CAN-FDSPI_ReceiveMessageGet(APP_RX_FIFO, &rxObj, rxd,
MAX_DATA_BYTES);
```

```
    int
DLC_RX = DRV_CAN-FDSPI_DlcToDataBytes(rxObj.bF.ctrl.DLC);
    uint32_t ID =rxObj.bF.id.SID;
    tfp_printf("\n\r Message received");
    tfp_printf("\n\r ID of RX is: %x",rxObj.bF.id.SID);//ID
    tfp_printf("\n\r DLC of RX is: %d",DLC_RX);
    tfp_printf("\n\r The data RX received are: [");
    int8_t i;
    for (i = 0; i < DLC_RX; i++)
        {
        if (i!=DLC_RX-1){
            tfp_printf("%X, ",rxd[i]);
            }
        else tfp_printf("%X] ",rxd[i]);;
        }
    counter+=1;
    tfp_printf("\n\r End of Message and counter is %d\n",counter);
    }
}
```

Figure 7.1. TD_USER_Loop modification. Source: Own.

However, this code cannot be assumed that work as it should, for intake it has been checked using the external loopback mode. So, to validate that it can receive messages concurrently a call-back has been implemented inside of *TD_USER_Setup*, which calls periodically a function that generates a random message.

Call-back instructions look like this:

```
TD_SCHEDULER_AppendIrq(10,0,0, 5, ConfigMessage,0);
```

Figure 7.2. Call-back instruction. Source: Own.

Arguments of Call-back instruction are the interval time, tick, delay, number of repetitions, the function that has to call and the arguments of the called function. In this case, the function is calling the function *ConfigMessage* every 10 seconds 5 times.

Using *TD_SCHEDULER_AppendIqr* sometimes appeared errors when sending messages to the FIFO channel, even so as this function is not needed in a real case it can be used anyway.

The next images show how the Black-Box can receive data concurrently, making valid the code to get CAN-FD messages from other devices.

```
ID of TX is: 2d
DLC of TX is: 20
The number of bytes of Data is: 6
The data TX send are: [1d, 0, 39, 25, 22, 3a]
Message send
Message received
ID of RX is: 2d
DLC of RX is: 20
The data RX received are: [1d, 0, 39, 25, 22, 3a, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0]
End of Message and counter is 1

ID of TX is: 794
DLC of TX is: 7
The number of bytes of Data is: 29
The data TX send are: [34, 27, 17, 0, e, 1, 2b, 2c, e, 22, 11, 30, 1b, 4, 28, 1
e, 26, f, 6, f, 25, 10, 6, 3, 3, 1e, 5, 15, 1c]
Message send
Message received
ID of RX is: 794
DLC of RX is: 7
The data RX received are: [34, 27, 17, 0, e, 1, 2b]
End of Message and counter is 2

ID of TX is: 72b
DLC of TX is: 4
The number of bytes of Data is: 28
The data TX send are: [20, 1d, 35, 6, 1f, 11, 33, 36, 36, 19, 3e, 14, f, 16, 33
, 2, 3c, 6, 1, 3d, 25, e, 34, 35, 25, 31, 1a, 36]
Message send
Message received
ID of RX is: 72b
DLC of RX is: 4
The data RX received are: [20, 1d, 35, 6]
End of Message and counter is 3
```

```
ID of TX is: 1e6
DLC of TX is: 3
The number of bytes of Data is: 9
The data TX send are: [10, 31, 2a, 8, 3e, 1f, 2d, 35, 18]
Message send
Message received
ID of RX is: 1e6
DLC of RX is: 3
The data RX received are: [10, 31, 2a]
End of Message and counter is 4

ID of TX is: 19a
DLC of TX is: 12
The number of bytes of Data is: 25
The data TX send are: [3e, 10, 0, 3d, 7, 18, 2e, 1e, 9, 32, 6, 12, 1d, 3c, 1b,
18, f, 35, 0, 8, 10, 20, a, 12, 3b]
Message send
Message received
ID of RX is: 19a
DLC of RX is: 12
The data RX received are: [3e, 10, 0, 3d, 7, 18, 2e, 1e, 9, 32, 6, 12]
End of Message and counter is 5
```

Figure 7.3. Result of sending 5 consecutive bytes using the call-back instruction. Source: Own.

7.2. Warmings using SIGFOX communication

The other feature that the Back-Box must implement is to send warnings/alarms to the Sigfox backend depending on the ID and the data values of the CAN-FD frame. As mentioned in the introduction, this feature is a must for the new Black-Box that include a communication system to alert both malfunctioning and crashes of a vehicle.

The management of the received messages has been done by creating a list that classy IDs, the maximum value of data that they could have, and the message that has to be sent via Sigfox. In this project, the implemented list only defines 10 different identifiers and maximum values because the aim of that is to build a functional prototype of a Black-Box.

```
uint8_t N_warnings=10;
ID_VALUES warning [N_warnings];
warning[0]=(ID_VALUES){10,310,"Fatal err"};
warning[1]=(ID_VALUES){1,1000,"T > lim"};
warning[2]=(ID_VALUES){2,2000,"RPM>lim"};
warning[3]=(ID_VALUES){3,30,"Low Tank"};
warning[4]=(ID_VALUES){4,400,"Seat belt"};
warning[6]=(ID_VALUES){6,95,"Open door"};
warning[7]=(ID_VALUES){7,310,"Open boot"};
warning[8]=(ID_VALUES){8,62,"No conect"};
warning[9]=(ID_VALUES){9,100,"Change oil"};
```

Figure 7.4. List of the warnings implemented. Source: Own.

Warmings are sent to Sigfox Backend using a proper function of TD1204 EVB called *TD_SIGFOX_Send*. This message could contain up to 12 bytes and it is received in the backend with a Hex format.

```
TD_SIGFOX_Send(warning[0].Message,12,1)
```

Figure 7.5. Instruction to send messages via Sigfox: Own.

The arguments of *TD_SIGFOX_Send* are the message to send, the payload size and the last is the number of retries in the sending.

Once the microprocessor runs the last instruction, it sends a message to the Sigfox backend.

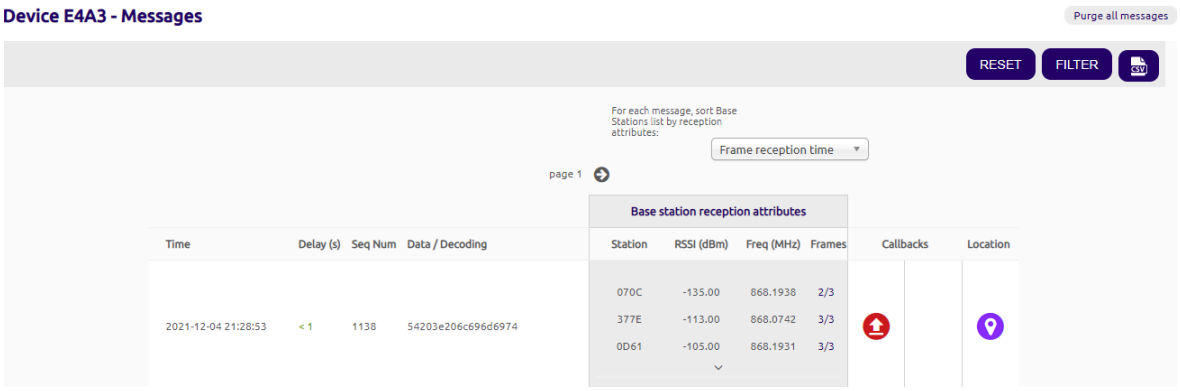


Figure 7.6. Sigfox backend received messages. Source: Own.

As mentioned previously, to understand the received data has to be converted to ASCII.

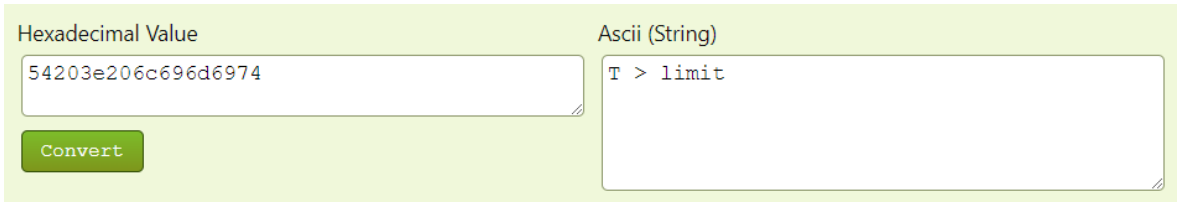


Figure 7.7. Example of converting the hexadecimal value to ASCII. Source: Own.

8. Proving the Black-Box with real CAN-FD nodes

8.1. Real CAN FD bus demonstrator

A demonstrator has been used to validate the functionality of the Black-Box. The demonstrator incorporates three boards, 2 ECUs and one Sniffer. Thanks to that, the demonstrator can emulate a real CAN FD system, such as the one that a vehicle or production line could have.

ECUs are nothing more than CAN FD nodes that send messages with random data and identifiers. The only difference between the two implemented ECUs are that the random identifiers of one are odd while the other is even.

The other board, called Sniffer, has the main functionality of capturing the CAN FD data frames sent by the ECUs and displaying them if a computer is connected. Additionally, it can send customized CAN FD messages.

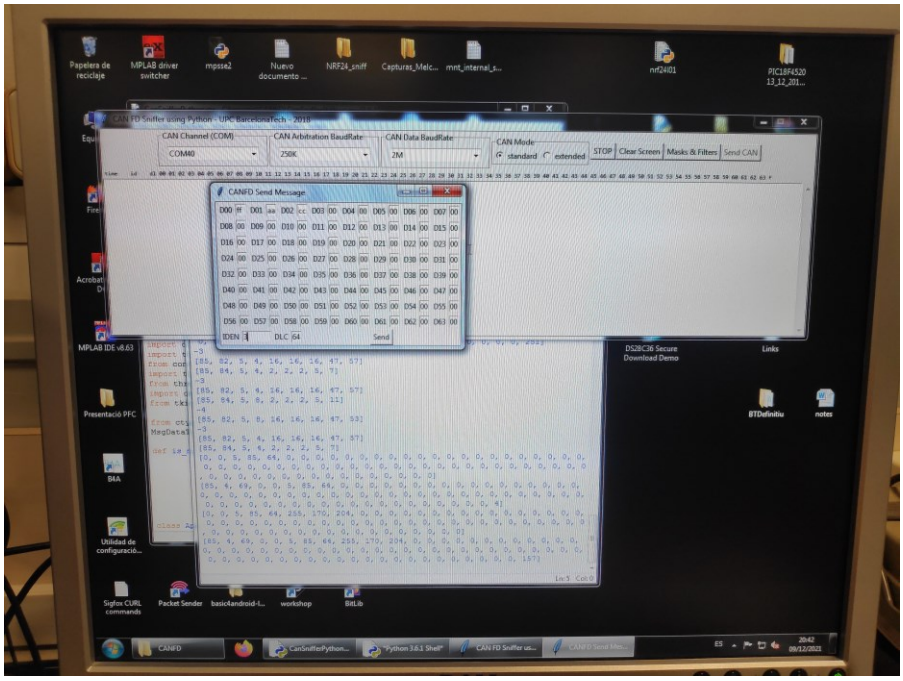


Figure 8.1. CAN FD program to send customized messages. Source: Own.

Talking deeper about these boards, all of them have three different buttons. The first, located in the top left, is called Baud rate and it is in charge of modifying the velocity of the node, it is

important to know that the baud rate of the node has to be the same as the one in the gateway to send messages. The second button, located at the right of the baud rate, is the Start/Stop and it has the purpose of initializing or stopping the message sending. The last button is the RESET, located in the lower-left corner, which has the function of initializing the board, it has to be always pressed when a device is connected to the demonstrator.

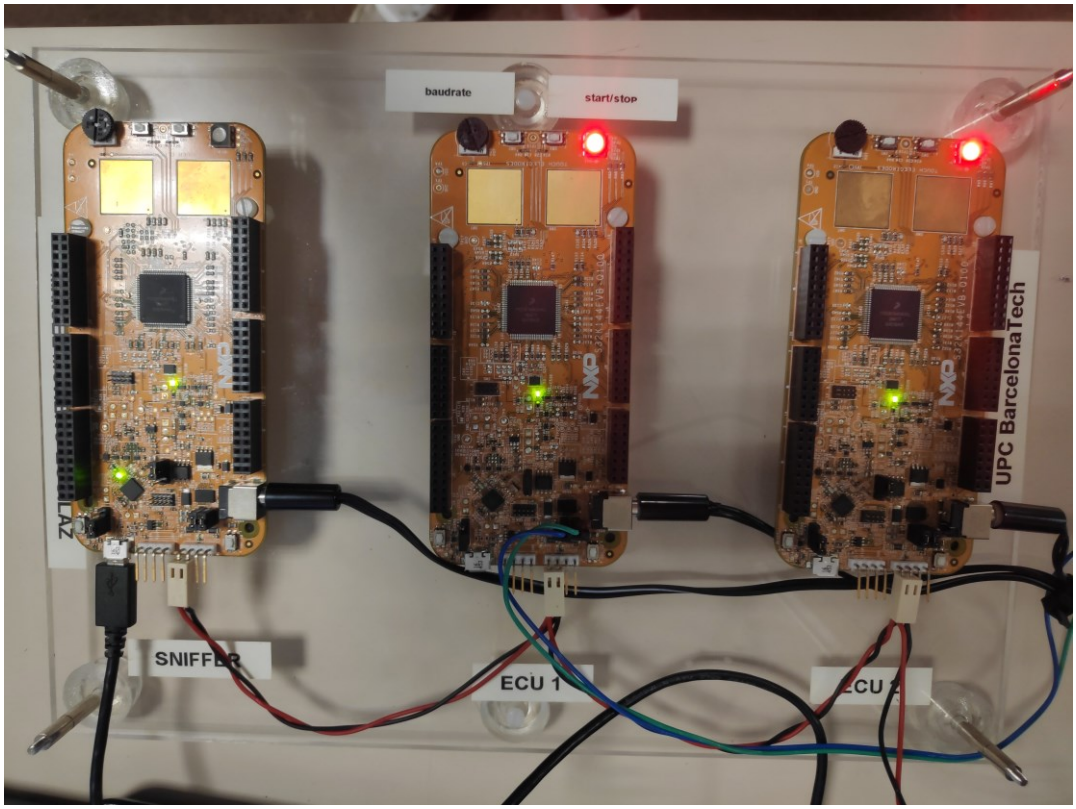


Figure 8.2. CAN FD demonstrator. Source: Own.

Finally, only last to talk about how the demonstrator is linked with our device. The connection is done using the D-sub type male connector, the de facto standard for the CAN FD physical layer, which incorporates the MCP2517 FD. The D-Sub pins that have been used are the 2, 3 and 7, which correspond to the CAN-Low, GND and CAN-High signals, respectively.

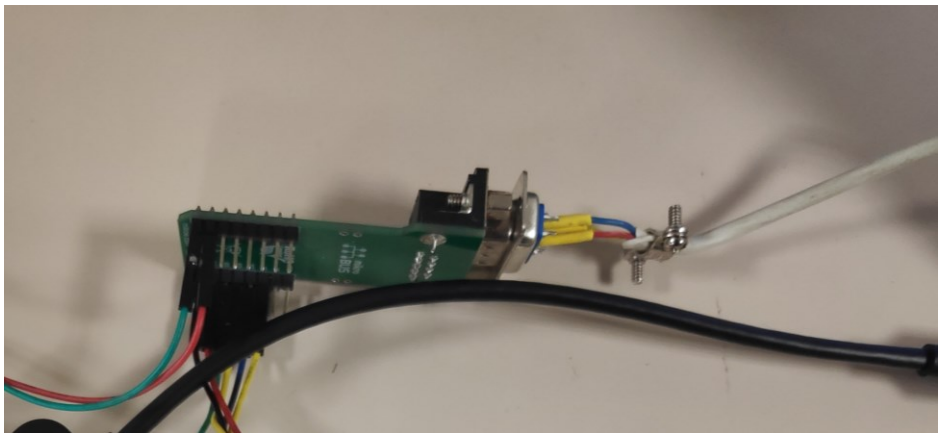


Figure 8.3. The connection between MCP2517 FD and the demonstrator. Source: Own.

8.2. Results using the demonstrator

This chapter shows the results using the demonstrator for two different tests. One consists of pressing the Start/Stop button of ECU 1 and the other sending a customized message using the Sniffer. In both tests, the operational mode of the CAN-FD controller has been changed to CAN_NORMAL_MODE.

Figure 8.4. Test setup. Source: Own.

8.2.1. Test 1

The test consists of pressing the Start/Stop of the ECU 1 or ECU 2 and seeing if the Black-Box could receive correctly all of them.

The results can be checked by clicking on the next link:

https://youtu.be/LwAauRLsbgY

As shown in the video, the Black-Box receives correctly all the messages sent by ECU 1.

8.2.2. Test 2

Test 2 consist of sending customized messages that produce warming in the Black-Box.

To see the results, you have to click on the next link:

https://youtu.be/TqowsLSdvcI

In the video, it is shown how the customized message produces the warming “Tank low”, which will be automatically sent to the Sigfox Backend.

9. Environmental impact

It is increasingly common for vehicles to have a Black-Box because communications between components have increased at outrageous speeds. With this line, the project develops a high efficient Black-Box not only because it uses CAN-FD but also for including Sigfox communication.

Starting talking about CAN-FD is a protocol much better than its predecessor CAN, as it improves both data rate and efficiency. Consequently, the protocol accepts a wider range of devices interconnected, increasing the overall efficiency of the whole system.

On the other hand, using Sigfox communication to send the warmings contribute to a huge savings of both energy and cost. The reduced price of Sigfox Backend and the low consumption of the protocol makes this technology a good sustainable option.

To sum up, it can assert that this project has an overall positive impact by enhancing the utilization of both efficient communications such as are CAN-FD and Sigfox. Additionally, the negative impact regarding the usage of materials is not important at all, as it only uses a few devices.

10. Planning

The final master project of the MUEI master degree consists of 12 ECT credits, where one ECT corresponds approximately to 25-30 hours of work. In my case, the development of the project has required 320 hours distributed among different processes.

Starting chronologically, the first work done in the project was setting the Eclipse IDE with the drivers, launchers and libraries and configuring the debug out mode of the Starter kit EFM32TG-STK3300. The total time invested in this part reaches 20 hours, due to some problems involving getting the launchers of Telecom Design, it seems that the company no longer works.

The next step in the project was implementing SPI protocol via software, which took like 40 hours. The time to do that includes testing the pins that could be used, getting and modifying an SPI file to fit with the project and modifying some functions of the API file. Additionally, there was like 5 hours more in terms of documentation about the SPI.

After that, the following procedure was creating the main source file, which required a total of 120 hours. The work of creating the main file includes reading the documentation about CAN-FD and the controller, coding the file, testing the SPI instructions and messages and correcting all the errors that appeared.

Once the main file was done, the next task involved modifying the file to accept external messages, manage data and send warnings to Sigfox. The time used in this part, including the tests done to check that it works correctly, has lasted up to 50 hours.

Finally, for the documentation and presentation, 85 hours have been inverted, which give the total 320 hours done for doing the project.

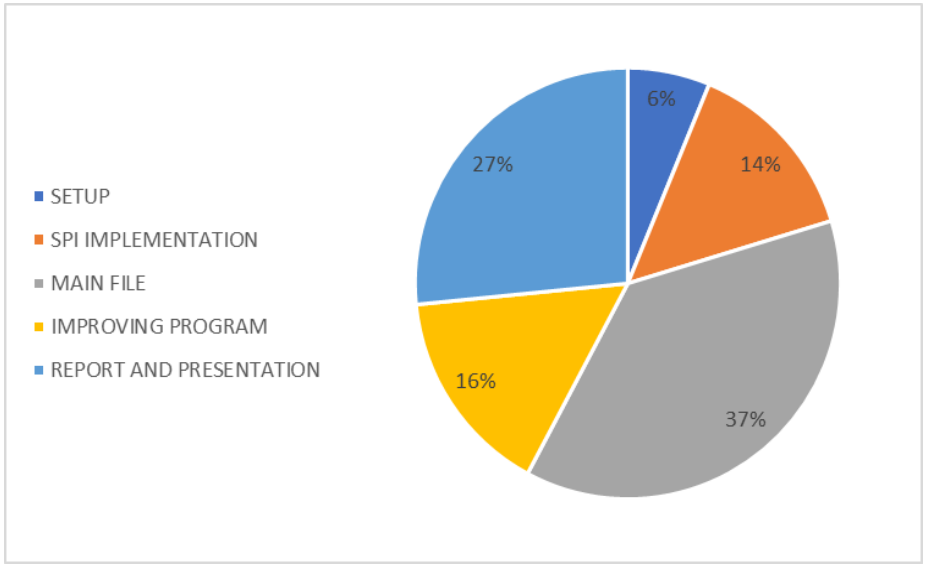


Figure 10.1. Time distribution. Source: Own.

11. Budget

The budget has been broken down into two parts, one that includes the pertinent cost of devices and software, and the other that relates to the hours of engineering.

At first, the devices and software costs are presented as well as the supplier. This includes the TD1204 EVB, Starter Kit EFM32TG-STK3300, MCP2517 FD Click, laptop and a hub USB. Regarding the laptop, it has been considered that has an amortization of 5 years and because of that, the price shown represents only 10% of the total value.

The next table shows the prices and the supplier for each device:

Device	Supplier	Cost (€)
TD1204 EVB	Avnet	236,75
Starter kit EFM32TG-STK3300	Farnell	98,64
MCP2517 FD CLICK	Mouser	26,47
USB HUB	Amazon	10,99
Laptop-HP envy 13	Amazon	80
Sigfox Backend 1 year subscription	Sigfox	18,14
		470,99

Table 11.1. Device cost

Secondly, regarding the engineering cost, it has been invested a total of 320 hours with an associated price of 30 €/hour. With that, the engineering cost reach a total of 9,600 €.

Finally, adding both costs give an approximate budget of 10,071 € for the 300 hours invested in developing the Sigfox CAN-FD Black-Box.

Conclusions

Once done the last validations, I can affirm that the main objective of the project has been satisfactorily met, which is developing a CAN-FD Black-Box that allows sending messages using Sigfox communication. Even though the device was only tested in a laboratory demonstrator, it can be ensured that it could work correctly in a real system, as long as the pertinent modifications were made.

In the same way, I could say that the second objective of creating a code clear and easy-to-understand has been achieved too. Additionally, the explanations shown in this report helps to improve its comprehension.

Furthermore, note that the implementation of an SPI pin via software was successfully done, providing a solution of the connection between the TD1204 EVB and the MCP2517 FD Click.

Finally, I want to say that the development of the Black-Box does not end here, existing a large number of possible improvements. One of these might be creating a Black-Box that could receive a Sigfox message that modifies the behaviour of some specific nodes.

Acknowledgements

I do not want to finish without thanking professor Manuel Moreno Eguílaz for his help throughout the development of the project. Without any doubt, he has been more than a supervisor, being always available to give advice, answer questions and even help me to solve any unexpected problem that arose.

Bibliography

Bibliographic references

[1] Www.Ni.Com. (2014). *Understanding CAN with Flexible Data-Rate (CAN FD).* [online] Available at: https://www.ni.com/es-es/innovations/white-papers/14/understanding-can-with-flexible-data-rate--can-fd-.html [Accessed 4 Oct. 2021]

[2] EN.wikipedia.org. (2021). *Event data recorded.* [online]. Available at: https://en.wikipedia.org/wiki/Event_data_recorder#Regulatory_framework [Accessed 9 OCT 2021]

[3] TELECOM DESING (2015). *TD1204 Datasheet.* Available at: https://github.com/Telecom-Design/Documentation_TD_RF_Module/blob/master/TD1204%20Datasheet.pdf [Accessed 8 AUG. 2021]

[4] TELECOM DESING (2015). *TD1204 EVB User's Guide.* Available at: https://github.com/Telecom-Design/Documentation_TD_RF_Module/blob/master/TD1204%20EVB%20User's%20Guide.pdf [Accessed 8 AUG. 2021]

[5] SILICOM LABS (2011). *USER MANUAL Starter Kit EFM32TG-STK3300. Available at: https://www.silabs.com/development-tools/mcu/32-bit/efm32tg-starter-kit* [Accessed AUG 10]

[6] MikroElektronika. (2020). *MCP2517FD datasheet*. Available at: https://www.mikroe.com/mcp2517fd-click [Accessed 10 AUG. 2021].

[7] MikroElektronika. (2020). *ATA6563 datasheet*. Available at: https://www.mikroe.com/mcp2517fd-click [Accessed 10 AUG. 2021].

[8] Campbell, S. (2016). *Basics of the SPI communication protocol. Circuit Basics*. [online]

Available at: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/

[Accessed 12 OCT 2021]

[9] CSS Electronics. (2021*). CAN FD explained - A simple intro.* [online] Available at: https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro [Accessed 6 NOV 2021]

[10] EN.wikipedia.org. (2021*). CAN bus.* [online]. Available at: https://en.wikipedia.org/w/index.php?title=CAN_bus&oldid=1065503892 [Accessed 6 NOV 2021]

[11] CAN in Automation (CiA) (2015). *Basic information on the CAN physical and data link layer.* Available at: https://www.can-cia.org/can-knowledge/can/systemdesign-can-physicallayer/ [Accessed 7 NOV 2021]

[12] VECTOR. *CAN FD An introduction*. Available at https://can-newsletter.org/assets/files/ttmedia/raw/778ff25c6a48bbd0d67b4eee94ea11ea.pdf [Accessed 13 NOV 2021]

[13] KENT LENNARTSSON. *Comparing CAN FD with Classical CAN.* Available at: https://www.kvaser.com/wp-content/uploads/2016/10/comparing-can-fd-with-classical-can.pdf [Accessed 13 NOV 2021]

[14] BOSCH (2012). *CAN with Flexible Data-Rate.* Available at: https://can-newsletter.org/assets/files/ttmedia/raw/e5740b7b5781b8960f55efcc2b93edf8.pdf. [Accessed 13 NOV 2021]

[15] JAVIER A. De Esteban Garau (2020). *Registrador de CANFD utilizando lenguaje C y una Raspberry PI. Available at: https://upcommons.upc.edu/handle/2117/334112 [Accessed 21 NOV 2021]*

[16] Cnblogs.com. Canbus ID filter and mask. [online] Available at: https://www.cnblogs.com/shangdawei/p/4716860.html [Accessed 22 NOV 2021]