Master Thesis

Master's degree in Industrial Engineering

Development of a CAN-FD Sniffer using Python

REPORT

Author: Joaquín Cortés Fuentes
Director: Juan Manuel Moreno Eguilaz
Call: June 2019

ETSEIB

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

DEVELOPMENT OF A CAN-FD SNIFFER USING PYTHON

JOAQUÍN CORTÉS FUENTES

Master's degree in Industrial Engineering
Electronic Engineering
Universitat Politècnica de Catalunya
Escola Tècnica Superior d'Enginyeria Industrial de Barcelona
(ETSEIB)

June 2019

Abstract

Today, the CAN-FD protocol is taking over classical CAN, as it allows a faster transfer speed, and as technology is being developed, more and more information is needed to be sent and processed between several devices connected all together in order to perform. In this project, a CAN-FD sniffer[1] software has been developed in Python language, in order to be able to communicate through a CAN-FD network, using a CAN-FD controller and transceiver, a SPI interface, and a FTDI-USB cable to connect this device to a PC.

A Python library has been written in order to implement the communication of the PC and the CAN-FD. Currently, there are already libraries to do this functionality in other languages (such as C and C++), but Python was chosen because this language is lacking this kind of libraries that support CAN-FD (there are CAN libraries, but very few have CAN-FD support, and most of them have limited support), and it is being more used every day in data analysis and machine learning tasks, so it is very important to be able to capture all possible information about several devices in order to gather data and perform analysis.

Finally, a GUI is implemented, in order to have a basic layout for using the library functions and perform simple tests so as to verify the library.

1 A sniffer is a device (usually a software program, but can be also a combination of software and hardware) that is able to capture all the traffic between other devices connected to a network with a shared media. It is usually used in network tests and analysis, but also for malicious ends, such as personal data spying.

Contents

List of Figures

List of Tables

Listings

Glossary

.c, .h C file extensions.
.py Python file extension.

API Application Programming Interface.

bin Binary.

CAN Controller Area Network.
CAN-FD CAN with Flexible Data-Rate.
CLK Clock signal.
CS, nCS Chip Select signal.

FIFO First In First Out stack.

GND Ground.
GPIO General Purpose Input-Output.
GUI Graphical User Interface.

hex Hexadecimal.

I^2C Inter-Integrated Circuit.

JTAG Joint Test Action Group.

LSB Least Significant Bit.

MB MegaByte.
MB MegaBit.
MISO, SDI Master Input Slave Output.
MOSI, SDO Master Output Slave Input.
MPSEE Multi-Protocol Synchronous Serial Engine.
MSB Most Significant Bit.

OS Operating System.

R/W Read/Write.
RAM Random Access Memory.
Rx Reception.

SPI Serial Peripheral Interface.

Tx Transmission.

V_{CC} Input voltage.

1 Objectives

This Project was proposed by the Department of Electronic Engineering in order to develop the required software for the *MCP2157FD* CAN-FD receiver and transmitter to work in a PC with Windows, using the Python programming language. This language has been chosen as, for today, no public library available for Python was found to control this device (and there are very few libraries to work with a CAN-FD frame). The increasing use of Python, particularly in fields such as Data Analysis and Machine Learning, makes the necessity of being capable of gathering data from many sources, with can be working with different communication protocols, and as the CAN-FD protocol is also increasing in the industry (specially in the automotive sector), this kind of library is also needed in order to make possible the data collection and work with it in later analysis in the easier way possible.

In this project, a base library software will be developed, so it can be implemented in later applications. Also, a GUI interface application will be developed in order to test the base library and to have a basic graphical software that is capable of transmitting and reading messages from a CAN-FD device. External libraries will be used, as well as the necessary drivers and hardware to connect the MCP2157FD to a USB port. The objectives are, in this case:

- Develop a Python library to make possible the communication of the PC with the MCP2157FD device, using SPI protocol.
- Extend this library to be capable of reading and writing data to the device.
- Develop this software using as few external libraries as possible, so it can be more portable.
- Develop a Graphical User Interface (GUI) to test the library and to have a basic application for reading and writing messages in a CAN-FD frame.

2 Overview of concepts

In this chapter, some general concepts present in the project will be explained in order to understand the following chapters and the development of the software. Knowledge in digital electronics and programming is required to understand the explained concepts.

2.1 CAN-FD

CAN-FD protocol is a communication protocol conceived in 2012 by Bosch in order to have a superior version of the CAN protocol, able to transmit as much as eight times faster, as described in [9]. Before discussing the details of the CAN-FD protocol and its improvements, the CAN protocol should be explained before.

The CAN protocol is a communication protocol designed in the mid-80s and widely used in the industry since the 90s, particularly in the automotive sector. It allows to communicate several processing units (microcontrollers, sensors and actuators) without the need of a host computer, as described in [10]. The main characteristic of this protocol is how the data is transmitted across devices, with data packets (*frames*) containing information of the message (such as identication, length of message, etc.) and the actual data to be transmitted, between 0 and 8 bytes. It has a maximum speed of 1 Mb/s. Figure 1 details how a data frame is constructed. Here, the fields of the standard base format of the frame can be seen, and in Table 1, the detail of each one is described.

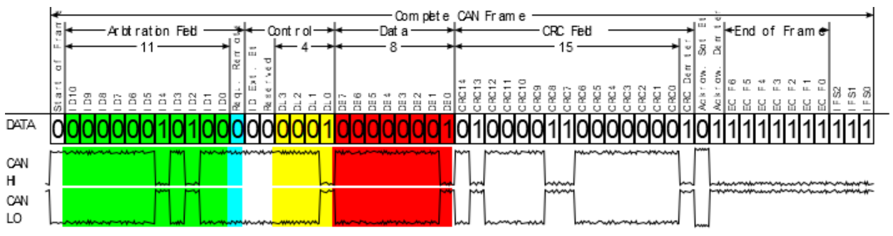


Figure 1: CAN data frame. Source: [1].

The most important fields are the identifier (which identifies the priority of the message) and the data field (which contains the actual data being transmitted). All the other fields perform auxiliary functions in order to properly transmit the message.

In addition to this standard frame, there is also the extended frame, which allows to have a 29-bit identifier, as well as two reserved bits. This extended fields are shown in Figure 2.

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier	11	A (unique) identifier which also represents the message priority
Remote transmission request (RTR)	1	0 for data frames and 1 for remote request frames
Identifier extension bit (IDE)	1	0 for base frame format with 11-bit identifiers
Reserved bit (r0)	1	Reserved bit. Must be 0, but accepted as either 0 or 1.
Data length code (DLC)	4	Number of bytes of data (0–8 bytes)
Data field	0-64	Reserved bit. Data to be transmitted (length in bytes dictated by DLC field)
CRC	15	Cyclic redundancy check. Used to detect errors in the data transmission
CRC delimiter	1	Must be 1
ACK slot	1	. Transmitter sends 1 and any receiver can assert a 0
ACK delimiter	1	. Must be 1
End-of-frame	7	Must be 1

Table 1: CAN data frame fields. Source: [1].

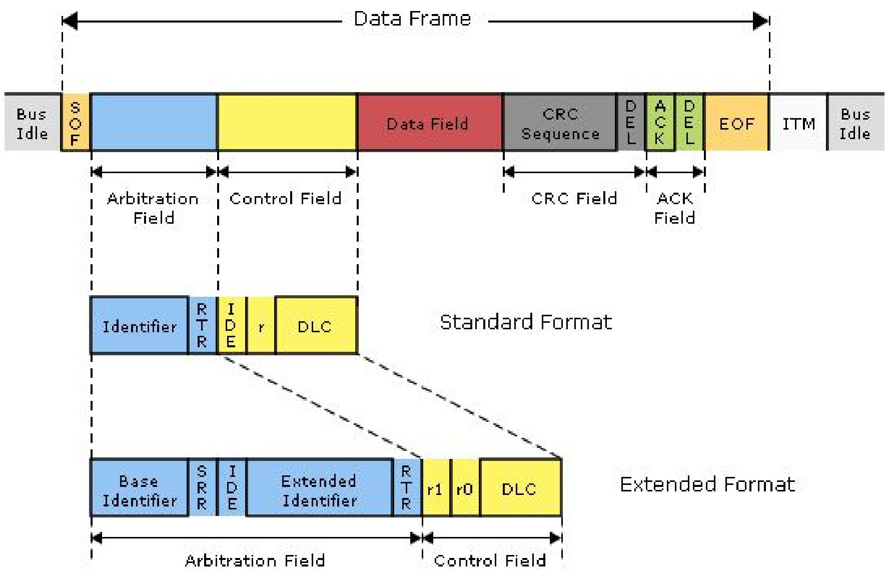


Figure 2: Extended CAN data frame. Source: [2]

As it can be seen, it has a 11-bit base identifier, followed by the SDR (Substitute remote request) bit, which must be always 1, the IDE bit, an extended identifier (of 18 bits), the RTR field, and an extra reserved bit. The main advantaged of this extended frame is being able to have a larger identifier, as well as having more reserved bits for future updates or additional features.

The ISO 11898 specification [10] has all of the details in architecture of the network, as well as electrical specifications, and will not be discussed here.

The main disadvantage of the CAN protocol is the maximum size of the the packets (8 bytes), as well as the speed of transmissions (1 Mb/s). The CAN-FD protocol does not have these limitations, as it allow speeds as high as 5 Mb/s and data messages of 64 bytes, as described in [11]. As it is an updated version of the CAN protocol, it inherits many of its characteristics, and the modifications needed to change a CAN network to a CAN-FD one are minimal. In Figure 3, the CAN-FD data frame is shown in comparison to the standard (non-extended) CAN frame.

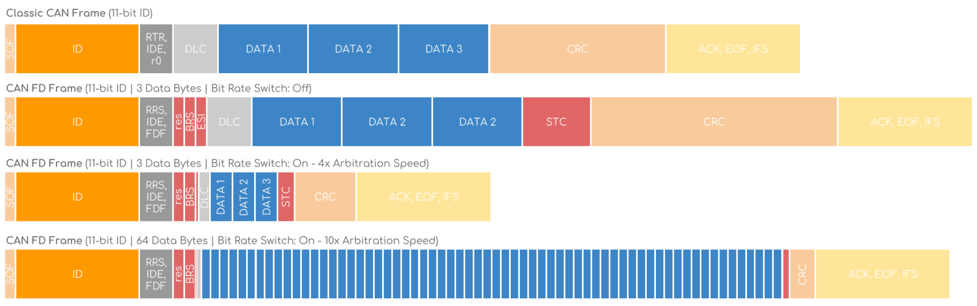

Classic CAN Frame (11-bit ID)
SOF
ID
RTR, IDE, r0
DLC
DATA 1
DATA 2
DATA 3
CRC
ACK, EOF, IFS
CAN FD Frame (11-bit ID | 3 Data Bytes | Bit Rate Switch: Off)
SOF
ID
RRS, IDE, FDF
res
BRS
ESI
DLC
DATA 1
DATA 2
DATA 2
STC
CRC
ACK, EOF, IFS
CAN FD Frame (11-bit ID | 3 Data Bytes | Bit Rate Switch: On - 4x Arbitration Speed)
SOF
ID
RRS, IDE, FDF
res
BRS
DLC
DATA 1
DATA 2
DATA 3
STC
CRC
ACK, EOF, IFS
CAN FD Frame (11-bit ID | 64 Data Bytes | Bit Rate Switch: On - 10x Arbitration Speed)
SOF
ID
RRS, IDE, FDF
res
BRS
CRC
ACK, EOF, IFS

Figure 3: CAN vs CAN-FD. Source: [3].

The main differences between the protocols are:

- The following fields are added to the data frame:
  - RRS (Remote Request Substitution): The CAN-FD does not allow for remote frames, and the bit is substituted with this one.
  - FDF (FD Format): Denotes that the frame is of CAN-FD type. In standard CAN, this bit is always 0, whereas in CAN-FD, is 1.
  - res: reserved bit. It has the same function as the CAN r0.
  - BRS (Bit Rate Switch): Indicates if the transmission speed is set at the arbitration rate (of maximum 1 Mb/s) if set to 0, or if the transfer speed is higher (up to 10 Mb/s) if set to 1.
  - ESI (Error Status Indicator): Indicates that there is a failure in the system if set to 1.
  - STC (Stuff Bit Content): It consists of three bit in Gray code and a parity bit. It helps to improve the reliability of the communication.
- The DLC field has the same 4 bits as in the CAN frame, but also allows to use up to 64 bytes, having consistency with the standard CAN frame up to 8 bytes. The Table 2 shows how both protocols use the DLC field consistently.
- The CRC field is expanded from 15 bits to 17 bits (if the data transmitted is between 0 and 16 bytes), or to 21 bits (if the number of data bytes is higher).

DLC (bin)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
DLC (dec)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Classic CAN	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8
CAN FD	0	1	2	3	4	5	6	7	8	12	16	20	24	32	48	64

Table 2: Corresponce of DLC bits between CAN and CAN-FD protocols. Source: [3].

The CAN-FD protocol has more bits than the classic CAN, so it may seem that it can not transfer as many frames as the standard CAN per unit of time. This is in fact true if the speed is set to the arbitration rate and the data sent is of the same size (maximum 8 bytes), but the CAN-FD allows for higher transmission speed using the BRS bit, so the actual data bytes can be sent or received in less amount of time.

Also, due to the greater length of the data bytes, as much as 8 times more data can be transmitted in a similar amount of time.

The CAN-FD protocol is actually used in the ECU systems in electrical vehicles and advanced driving systems [12], robotics [13], and has improved security [14].

2.2 SPI protocol

The SPI (Serial Peripheral Interface) protocol is a synchronous serial communication interface used in short distance communication (typically between 10 and 100 meters). The interface was developed by Motorola in the 80s and is widely used in today industry. One of its main characteristics is the master-slave architecture, where the master device reads data from the slaves and writes to them. It allows full-duplex communication[2]. It has a four-wire serial bus with the following signals:

- SCLK (Serial Clock, also called SCK): it synchronizes the transmission of data between the devices. This makes the interface a synchronous one.
- MOSI (Master Output Slave Input, also called SDO): it is the data output of the master device.
- MISO (Master Input Slave Output, also called SDI): data input of the master device.
- SS (Slave Select, also called CS): it allows the master device to select between slave devices to communicate.

Figure 4 shows a typical bus between devices. As it can be seen, when there is more than one slave, the master device needs additional SS channels. Some devices allow to have daisy chain connections between devices, or allow to multiplex the signal in time or frequency

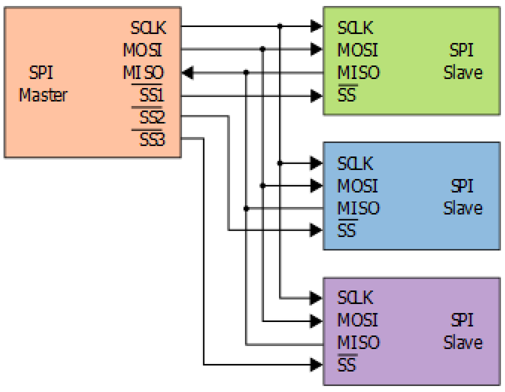


Figure 4: SPI interface with a master device and three slaves. Source: [4].

2 Full-duplex communication is the capacity to transmit data while simultaneously reading from the same device.

The SPI has four different modes, each one being a combination on the different values of the CPOL and CPHA, which determine the polarity of the clock and the timing of the data bits relative to the clock pulses. This configurations allow more flexibility between devices communicating. Figure 5 showcases the different configurations.

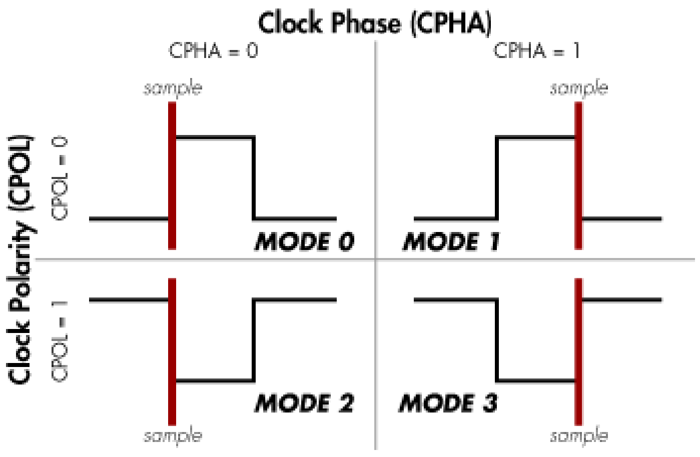


Figure 5: The four different configurations of the SPI. Source: [5].

The main advantages of the SPI protocol are the arbitrary size of messages and simple hardware interfacing. In the other hand, it only supports one master device (although certain hardware implementations may allow to have more than one master) and there is no defined protocol to check errors.

3 Description of Hardware

In this section, the hardware used in the project will be discussed. Details of each device and its technical implementations will not be discussed in the project, only its main aspects and characteristics.

3.1 MCP2157FD

The MCP2157FD chip is a CAN FD controller, designed by Microchip, with a SPI interface. It allows for communication with both CAN and CAN-FD networks. Its main characteristics are:

- Conforms to ISO11898-1:2015.
- Supports both CAN 2.0 and CAN FD.
- Arbitration Bit Rate up to 1 Mb/s.
- Data Bit Rate up to 8 Mb/s[3].
- Up to 20 MHz SPI Clock Speed.
- V_{DD}: 2.7 V-5.5 V.
- 2 kB RAM.
- Three interrupt pins.
- Up to 40 MHz internal clock.
- 20 MHz of SPI clock.
- Supports SPI modes 0 and 1.
- GPIO pins.
- 31 FIFOs for transmit and receiving of messages.

In Figure 6, the memory map of the controller is seen. A more detailed view of the memory map and the internal registers functions can be seen in [6].

3 Theoretical. In practice, it goes up to $\approx$ 5.5 Mb/s

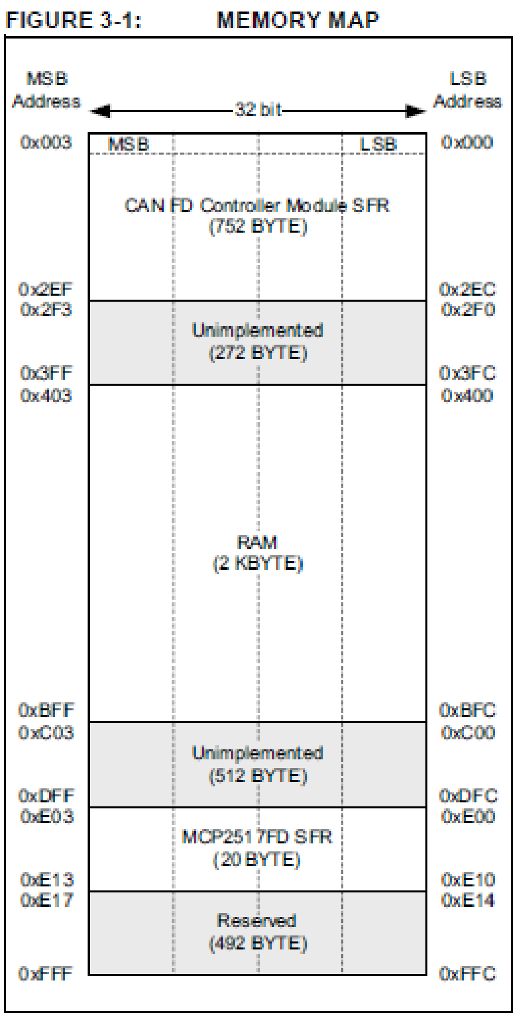


Figure 6: Memory map of the MCP2517FD. Source: [6].

In this project, the MCP2517FD CLICK device is used, which combines this controller chip with a ATA6563 high speed CAN transceiver (also from Microchip) and a standard DB9 pin male connector. This device needs both 5 V supply and 3.3 V. The MPSSE cable can supply 3.3 V, and an extra USB cable is used to power the 5 V needed (with no data transmission, only as power supply).

The ATA6563 is a CAN transceiver that interfaces between a CAN controller and the physical CAN bus. It allows for 5 Mb/s speed. It is integrated in the MCP2517FD CLICK device, and together with the MCP2517FD, it allows for transmittion and reception of messages from a physical CAN network, using the DB9 connector.

In Figure 7 the device is shown. The two main chips are seen, along with the DB9 connector and pin connections to the MCP2517FD

Figure 7: MCP2517FD Click. Source: own.

3.2 MPSSE

The MPSEE (Multi-Protocol Synchronous Serial Engine) cable, developed by FTDI, allows for communication between a USB port and several communication interfaces, such as SPI, JTAG and I^{2}C. It contains the FT232H chip, which handles all the USB communications and protocols, allowing to connect devices with several synchronous communication protocols to a computer or any other device with USB ports. The cable has at one end ten wires to be interfaced to a male header in the device of interest.

The main features of the cables are the following:

- USB protocol handled on the chip.
- USB powered.
- Data speed up to 30 Mb/s.
- 1 kB receive and transmit buffers.

In this project, the cable is used to interface the CAN-FD controller with a PC, using SPI communication. Figure 8 shows the cables on the device and how they relate to the signals on the device.

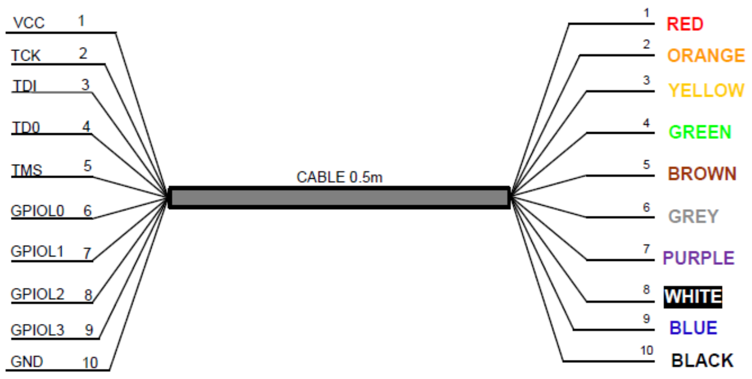


Figure 8: MPSSE signals and terminations. Source: [7].

Table 3 shows the description of the signals used in the SPI interface, and the relation between the device signals and the connections between the MCP2517FD pins. The MPSSE cable acts as the Master device in the SPI interface, and the MCP2517FD as the Slave.

Color	Pin Number	Name	MCP2517FD pin	MCP2517FD name	Description
Red	1	V_{CC}	+3.3V	7	Power supply output to target
Black	10	GND	8	GND	Device ground
Orange	2	SK	4	SCK	Serial Clock
Yellow	3	DO	6	MOSI	Serial Data Output (for Master device)
Green	4	DI	5	MISO	Serial Data Input (for Master device)
Brown	5	CS	3	CS	Serial Chip Select

Table 3: MPSSE connections. Source: own.

The MPSSE cable incorporates the FT232H chip, which allows for USB-SPI connnection.

4 Description of the proposed software and state of the art

4.1 Other libraries

Before explaining the proposed software developed in this project, the state of the art in CAN-FD libraries will be discussed. As the CAN-FD technology is relatively new (as 2019, it has been around 7 years), there is not a large number of available libraries. There is a popular Python library for CAN communication frame, the *python-can* library [15], which is easily available in Python environments with its *pip* package-management system . It has support for different device interfaces, and its main functionalities will be discussed in this section.

There is also an API to use with the controller proposed in this project, but it uses a library developed by Microchip written in C, and it is designed to be used with a Teensy 3.2 board[4]. This library will be also discussed, and it will be the starting point for developing the library for this project, as it incorporates the basic functions for using the controller.

4.1.1 *python-can*

This library allows to establish communication between a CAN controller and any kind of device capable of running Python. It has the architecture to allow for communication in a CAN frame, and has support for several devices and interfaces. In order to do this, it does an abstraction task in the operations performed in the CAN communication so it can read and transmit messages and all the necessary data in a CAN network.

This library has several available interfaces for different hardware devices, but only some of them support CAN-FD frames, which are the following:

- SocketCAN: This package is designed for implementation of CAN communication between devices in a Linux environment and implements the CAN device drivers as network interfaces.
- Kvaser's CANLIB: This interface allows for communication with a device of the Kvaser family.
- NEOVI Interface: This part of the library allows to communicate a device of the Intrepid Control Systems family.
- Vector: This part of the package allows for communication with devices of the Vector family.

4 The Teensy board is a embedded system for fast development that incorporates a 32-bit ARM Cortex processor.

As it can be seen, none of the available interfaces of this library have support for the device used in this project (the Microchip MCP2517FD) in a Windows OS. Additional interfaces can be added via plugins, but the process is complex and documentation of the process has not been found for a Windows machine and this specific device. Moreover, this project uses an intermediate device to communicate the CAN-FD controller and the USB port of the computer via SPI communication, and this library does not allow this kind of interface directly, and to do so, modifications of the library should be made, which is out of the scope of this project.

4.1.2 Microchip API

This API is developed to use the MCP2517FD device with the Teensy board. It has all the necessary packages to use the functionalities of the controller, as well as SPI communication and all the needed configurations, and a main function for demonstration purposes, which has the needed configurations for initializing the device, configuring the transmit and receiving FIFOs, transmittion and reception of messages with the FIFO and the interrupt pin (R/W of individual bytes, 32-bit words, byte arrays and word arrays, using CRC or not), and testing the RAM and registers read and write capabilities.

This library has all the needed capabilities for this project for using the controller. However, it has two main drawbacks:

- It is developed to be used with a Teensy board, which uses an specific processor, with its own instructions and interfaces.
- It is written in C.

Despite of this, this library will be used as a starting point for this project, as it has the necessary functions for communicating with the Microchip specific device (as it is developed by the same company). The CAN-FD functions will be translated from the C files to Python in order to be able to perform this functions in Python language.

As it can be seen, the available libraries have some mayor drawbacks: they do not support directly the device used in this project in a Windows OS and using Python, and modifications to do so are out of scope. Communication between this device and a PC may be easier in a Linux environment or substituting the PC with a embedded device (like the Teensy board), but the main objective of this project is to be able to communicate this specific device with a PC in a Windows environment using Python, an as it has been seen, there are no available solutions for this specific problem (at least in public domains). The official library by Microchip is a good starting point, as it incorporates the necessary functions to communicate with the MCP2517FD controller via SPI interface. The SPI protocol interface is not going to be developed, and instead, an already available Python library will be used, which will be discussed in the next chapter. This library allows for communication with a USB port using SPI and the MPSEE cable.

In the following section, the minimal specifications of the proposed library will be discussed, as well as the desired behaviour of its function.

4.2 Minimal functionalities

The proposed software needs to be able to do the following functions in order to perform communication in a CAN-FD network:

- SPI R/W: The library needs to be able to read and write bytes via SPI protocol between the PC and the device.
- R/W messages: Bytes, 32-bit words, byte arrays and word arrays must be able to be read and written.
- R/W CAN-FD messages: Full CAN-FD frames must be recognized and treated correctly, giving the resulting data in them.
- Configuration: FIFOs, channels, and internal configurations should be correctly set.
- Checking FIFO status: the software must be able to check the status of the FIFOs in order to know if messages can be read or written.
- Tests: several tests should be available so as to know if the device is performing correctly.
- GUI: a basic GUI should be also be developed, in order to make more accessible all of the functions of the library and to demonstrate the capabilities of the library.

The CAN-FD controller can work with an interruption system or with a polling one. This latter one will be the one used in this project, as the MPSEE cable used does not support interruptions. Also, half-duplex communication will be used for the same reasons, even in the controller supports full-duplex.

There are more options and functions available in the device, such as masks and filters, but they are not going to be discussed in this project.

5 Development of the library

After checking the minimal functionalities that the proposed library should have, it seems that translating the C code from the API is the best option. However, before doing that, the following tasks need to be done in order to properly set the device working:

- The MPSSE cable must be correctly installed and configured so as to be used in a Windows OS with Python.
- The SPI functions must be ported.
- The C library must be correctly understood in order to port it to Python.

5.1 Initial configurations

The correct drivers for the MPSEE cable must be installed in the computer in order to put the cable to work, as well as the necessary libraries to connect to the device. The Adafruit GPIO FT232H library has all the needed functions to easily connect any device to a computer via USB ports, using SPI protocol[5]. The next steps need to be done so as to install correctly the device and its associated libraries:

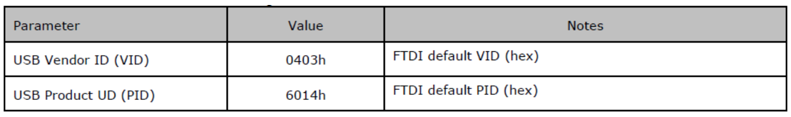
1. Install the drivers distributed by FTDI located at [16] so as the MPSEE cable appears as a COM port.
2. Substitute the VCP drivers installed by the libusB using the Zadig software located at [17].
3. Install the Python *libftdi* library, and the Adafruit library, both located at [18].

All this process is done accordingly to the instructions seen in [18]. After this process, the SPI communication can be now be controlled in Python.

5.2 MPSSE

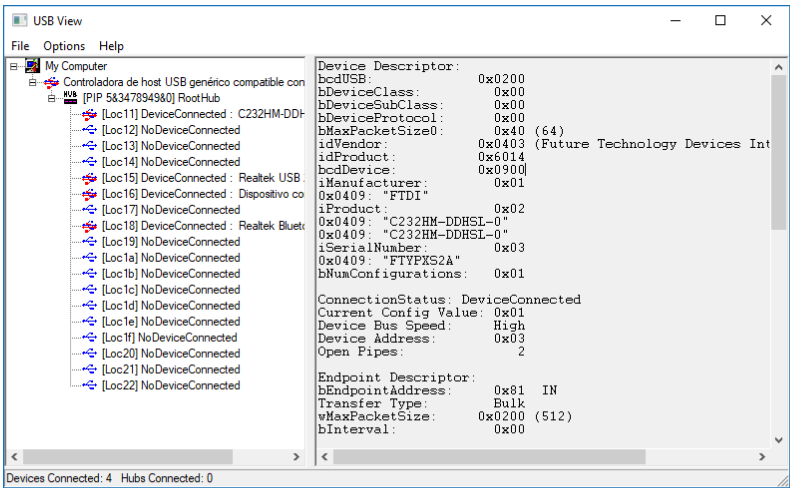
In order to test the correct installation, FTDI distributes a software to check the devices in the COM ports, and in the case of the MPSEE cable, see its internal values, like the serial number. Checking these values and comparing it to the ones in the manual [7], it can be seen that the device is correctly installed. In Figures 9 and 10 this values can be seen.

5 This library also supports other communication protocols, like I^2C, serial UART and JTAG, but as they are not needed in this project, they will not be discussed.



Parameter	Value	Notes
USB Vendor ID (VID)	0403h	FTDI default VID (hex)
USB Product UD (PID)	6014h	FTDI default PID (hex)

Figure 9: USB Vendor ID and Product ID values from the manual. Source: [7].



```
My Computer
  Controladora de host USB genérico compatible con
    [PIP 5&347894980] RootHub
      [Loc11] DeviceConnected : C232HM-DDH
      [Loc12] NoDeviceConnected
      [Loc13] NoDeviceConnected
      [Loc14] NoDeviceConnected
      [Loc15] DeviceConnected : Realtek USB
      [Loc16] DeviceConnected : Dispositivo co
      [Loc17] NoDeviceConnected
      [Loc18] DeviceConnected : Realtek Bluet
      [Loc19] NoDeviceConnected
      [Loc1a] NoDeviceConnected
      [Loc1b] NoDeviceConnected
      [Loc1c] NoDeviceConnected
      [Loc1d] NoDeviceConnected
      [Loc1e] NoDeviceConnected
      [Loc1f] NoDeviceConnected
      [Loc20] NoDeviceConnected
      [Loc21] NoDeviceConnected
      [Loc22] NoDeviceConnected
```

```
Device Descriptor:
bcdUSB:             0x0200
bDeviceClass:         0x00
bDeviceSubClass:      0x00
bDeviceProtocol:      0x00
bMaxPacketSize0:      0x40 (64)
idVendor:           0x0403 (Future Technology Devices Int
idProduct:          0x6014
bcdDevice:          0x0900
iManufacturer:        0x01
0x0409: "FTDI"
iProduct:             0x02
0x0409: "C232HM-DDHSL-0"
0x0409: "C232HM-DDHSL-0"
iSerialNumber:        0x03
0x0409: "FTYPXS2A"
bNumConfigurations:   0x01

ConnectionStatus: DeviceConnected
Current Config Value: 0x01
Device Bus Speed:     High
Device Address:       0x03
Open Pipes:              2

Endpoint Descriptor:
bEndpointAddress:     0x81  IN
Transfer Type:        Bulk
wMaxPacketSize:     0x0200 (512)
bInterval:            0x00
```

Devices Connected: 4 Hubs Connected: 0

Figure 10: USB Vendor ID and Product ID values read from software (among others). Source: own.

5.3 Adafruit library

In order to test the SPI communication, the MPSEE cable is connected to the MCP2517FD in the configuration seen in Table 3. The Adafruit library is then used to communicate the controller with the Python environment. To do so, Code 1 is used. Here, the register OSC, located at address **0xE00**, is read using the functions from the library. Comparing the values with the reset values referenced in the manual [6], the correct communication can be tested. The read process consists of writing the *read* instruction, followed by the address. Then, the data can be read at the MISO channel. The data must be a Python array consisting of 8-bit integers, and the returned data is a *bytearray* object. As the address has a length of 12 bits, it must be divided in two chunks.

```
# Test 1. SPI read
import Adafruit_GPIO.FT232H as ft

# Temporarily disable FTDI serial drivers.
ft.use_FT232H()
# Initialize a FT232H object
ft232h = ft.FT232H()
# Create a SPI object
spi = ft.SPI(ft232h, cs=3, max_speed_hz=20000000, mode=0, bitorder=
    ft.MSBFIRST)

# Instructions to read and write
cINSTRUCTION_READ = 0x03
cINSTRUCTION_WRITE = 0x02
```

```
# Prepare data to write
spiTransmitBuffer = []
address = 0xE00
spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >> 8)
    & 0xF))
spiTransmitBuffer.append(address & 0xFF)
spi.write(spiTransmitBuffer)
# Read data (4 bytes)
response = spi.read(4)

print ('Reading OSC register with SPI. Result:')
print ("32-bit word: {}".format(binascii.hexlify(response)))
print ("Bytes: {}".format(list(response)))
```

Listing 1: SPI Python example.

However, these instructions do not perform as required. This is because the library has a different instruction cycle that the needed in the controller. The library puts the CS signal at a high level before writing or reading, and then lowers it .In Figure 11 the controller instruction cycle is shown, and it is seen that it needs the CS signal to go to low level before writing the read instruction, and go high after reading.

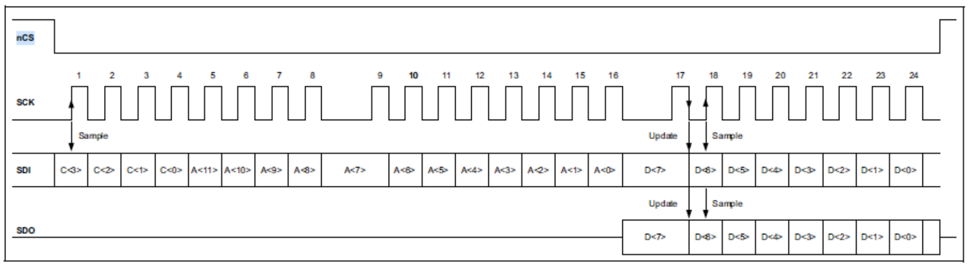

nCS
SCK
SDI
SDO
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Sample
Update
Sample
C<3> C<2> C<1> C<0> A<11> A<10> A<9> A<8> A<7> A<6> A<5> A<4> A<3> A<2> A<1> A<0> D<7> D<6> D<5> D<4> D<3> D<2> D<1> D<0>
Update
Sample
D<7> D<6> D<5> D<4> D<3> D<2> D<1> D<0>

Figure 11: Instruction Cycle for the MCP2517FD. Source: [6].

After inspecting the Adafruit library, the problem is solved by commenting the assert/deassert commands in the R/W functions and doing them manually according to the instruction cycle. This modified library (without changing CS signal in R/W functions) is saved as *ft.py* an used for the rest of the project. In Code 2 the modified test is seen, and the printed results are now correct.

```
# Test 2. SPI read with modified ft lib
import ft

# Temporarily disable FTDI serial drivers.
ft.use_FT232H()
# Initialize a FT232H object
ft232h = ft.FT232H()
# Create a SPI object
spi = ft.SPI(ft232h, cs=3, max_speed_hz=20000000, mode=0, bitorder=
    ft.MSBFIRST)

# Instructions to read and write
cINSTRUCTION_READ = 0x03
cINSTRUCTION_WRITE = 0x02

# Prepare data to write
spiTransmitBuffer = []
```

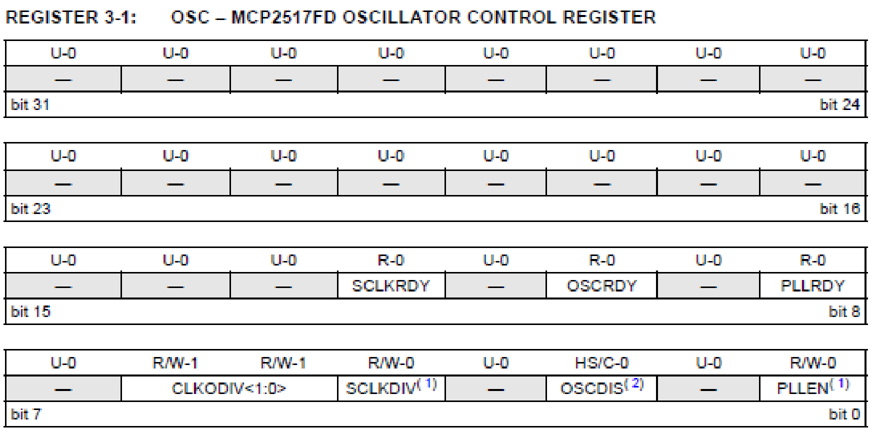
```
address = 0xE00
spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >> 8)
    & 0xF))
spiTransmitBuffer.append(address & 0xFF)
spi._assert_cs()
spi.write(spiTransmitBuffer)
response_0 = spi.read(4)
spi._deassert_cs()

print ('Reading OSC register with SPI. Result:')
print ("32-bit word: 0x{}".format(binascii.hexlify(response)))
print ("Bytes: {}".format(list(response)))

>> Reading OSC register with SPI. Result:
>> 32-bit word: 0x60040000
>> Bytes: [96, 4, 0, 0]
```

Listing 2: SPI Python example with modified ft library.

The *binascii* library is used to visualize better the results in a human-readable way. The result is printed in 32-bit word format (in hex) and as a byte array. As it can be seen, the results are the same as stated in the manual, but care must be taken in the bit and byte order: the first byte in the array corresponds to the leftmost byte in the word, and to the first byte in the word (bits 0 to 7). The LSB of the read word corresponds to the MSB of the value in memory (bits 24 to 31). In Figure 12, the values of the register are seen, as well as the location of its bits.



REGISTER 3-1: OSC – MCP2517FD OSCILLATOR CONTROL REGISTER

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31							bit 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23							bit 16

U-0	U-0	U-0	R-0	U-0	R-0	U-0	R-0
—	—	—	SCLKRDY	—	OSCRDY	—	PLLRDY
bit 15							bit 8

U-0	R/W-1	R/W-1	R/W-0	U-0	HS/C-0	U-0	R/W-0
—	CLKODIV<1:0>		SCLKDIV(1)	—	OSCDIS(2)	—	PLLEN(1)
bit 7							bit 0

Figure 12: OSC register values after reset from the manual. Source: [6].

This bit and byte order is used for all the values read and written in the device.

5.4 Microchip API

The C API developed by Microchip is very complete, as it has all the necessary functions to read and write messages in a CAN-FD frame, as well as subfunctions needed to do this. It has also more functions to operate the device in more advanced ways, such as using filters, masks, remote modes, and many more. Only the necessary functions of this library will be translated. The API is structured in the following way:

- System configuration: Clock and ports configurations are made. This part will be omitted as it does not apply to this project.
- SPI functions: The SPI protocol is configured and function to read and write are defined. This part will be also be omitted, as in this project, the SPI communication is made using the Adafruit library.
- CAN-FD functions: All functions needed to establish communication with the controller, including reading and writing of messages in RAM memory, use FIFOs for transmission and reception, interpreting the CAN-FD frames, configure all possible registers (at bit, byte or word level), resetting the device, etc.
- Main: A *main* loop with the logic of the program, showing the initialization and the different states of the API and which functions are executed in each one.
- Registers definition: The registers of the controller are defined in software, with its individual bits defined.
- Constants: Constants definition, such as reset values and several addresses are defined.

In the following sections, these files and its functions will be discussed, but without going into detail. These files can be seen at [19].

5.4.1 Constants

Constant values, such as memory addresses of certain registers, reset values, baudrate, clock settings, etc, are in this file.

5.4.2 Registers

Each register is defined in this file using the *struct* and *union* data types. In Code 3 the OSC register definition is seen. The other registers follow the same structure.

```
typedef union _REG_OSC {
    // Bits in register
    struct {
        uint32_t PllEnable : 1;
        uint32_t unimplemented1 : 1;
        uint32_t OscDisable : 1;
        uint32_t unimplemented2 : 1;
        uint32_t SCLKDIV : 1;
        uint32_t CLKODIV : 2;
        uint32_t unimplemented3 : 1;
        uint32_t PllReady : 1;
        uint32_t unimplemented4 : 1;
        uint32_t OscReady : 1;
        uint32_t unimplemented5 : 1;
        uint32_t SclkReady : 1;
        uint32_t unimplemented6 : 19;
    } bF;
    // 32-bit word
    uint32_t word;
    // Array of 4 bytes
    uint8_t byte[4];
} REG_OSC;
```

Listing 3: Register definition.

5.4.3 CAN-FD functions

These functions define the CAN-FD communication. The functions of interest are the following:

- *Reset*: Writes the reset command.
- *ReadByte*: Reads one byte from the memory address indicated.
- *ReadWord*: Reads a word from the memory address indicated.
- *ReadByteArray*: Reads an array of bytes, starting from the memory address indicated.
- *ReadWordArray*: Reads an array of words, starting from the memory address indicated.
- Write functions: Analogous to the read functions but adding an extra parameter with the data to write.
- *Configure*: Configures the *CiCON* (CAN Control Register) register.
- *OperationModeSelect*: Selects the operation mode of the controller.
- *OperationModeGet*: Gets the current operation mode of the controler.
- *TransmitChannelConfigure*: Configures the transmit FIFO CiFIFOCON register in the corresponding channel used for transmittion.
- *TransmitChannelLoad*: Checks that the message to transmit is put on a transmit buffer, is of correct length, updates status in the transmit FIFO CiFIFOSTA, gets RAM address to write message in from the CiFIFOUA register, and constructs the CAN-FD frame with the corresponding headers and fields.
- *TransmitChannelStatusGet*: Checks the transmit FIFO status.
- *ReceiveChannelConfigure*: Configures the receive FIFO CiFIFOCON register in the corresponding channel used for reception.

- *ReceiveChannelStatusGet*: Checks the receive FIFO status.
- *ReceiveMessageGet*: Checks that the FIFO selected is a receive buffer, is of correct length, updates status in the receive FIFO CiFIFOSTA, gets RAM address to read message from the CiFIFOUA register, and extracts data from the CAN-FD frame received.
- *EccEnable/Disable*: Enables/Disables the ECC (Error Correction Code) function of the RAM memory.
- *BitTimeConfigureNominal**XX**Hz*: Configures the CiNBTCFG register according to the desired bit time *XX*. (clock speed, in MHz).

This functions allow to communicate in a CAN-FD network.

5.4.4 *Main*

In the main file, the logic of the system is defined. There are two demos in the API: one is used to transmit a image file and receive messages using LEDs as indicators and buttons as command controls; the other is a simpler function, passing sequentially through the states, including tests, configuration, reception and transmittion. More functions are used in both demos of the API (like the use of TEF[6] and interrupt pins), but only the main ones (the necessary functions needed to establish communication in a CAN-FD network and configuring the device) are considered here. The basic state logic of this last demo is seen in Figure 13.

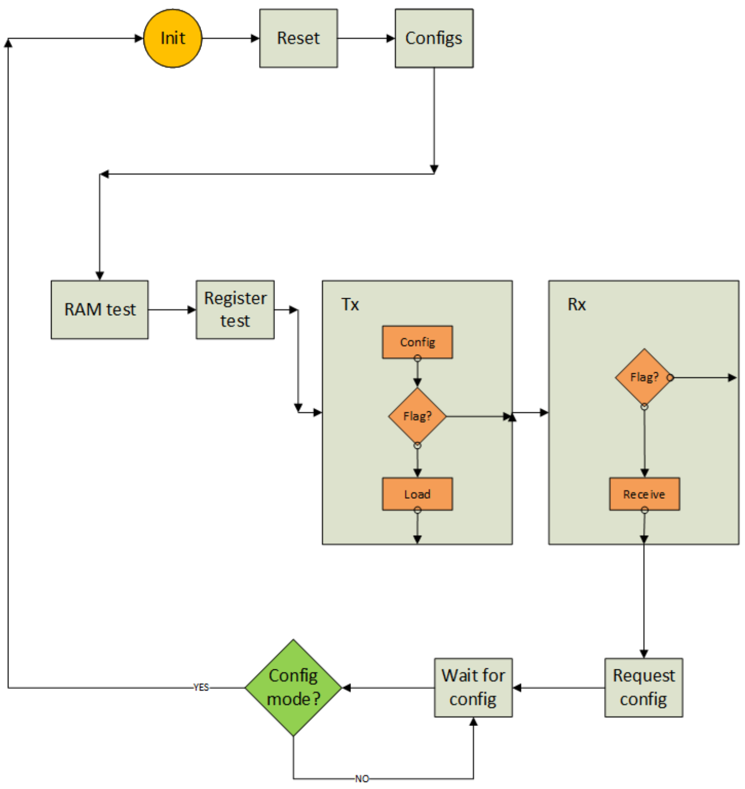


Figure 13: State machine of the *main* function in the API. Source: own.

6 The Transmit Event FIFO (TEF) stores the messages IDs of the transmitted messages.

The states are sequential, but a loop can be easily be implemented in order to check the state periodically and executing the functions needed. Also, there is no input in the simple demo: the message to transmit is hard-coded, and consists of 64 random bytes.
In each state, a series of functions are executed to perform the task. These are the following:

- *Init*: Initial state, variables are initialized.
- Reset: The device is reset.
- *Configs*: The controller is configured. This includes the CAN Control Register, Tx and Rx FIFOs, RAM, and Normal Mode is selected so the device can start operating. It should be noted that at least one filter object should be configured and linked to the Rx FIFO for the device to work.
- RAM test: Data is written in RAM memory and then read in order to verify it. The data consists of n random bytes, where n goes from 4 to the maximum length of messages (64 bytes) in steps of 4[7].
- Register test: Same as the RAM test, but now data is written in registers CiFLTOBJ.
- Tx: Transmission of message. This task consists of three subtasks:
  - *Config*: The header of the frame is prepared, as seen in Code 4.
  - Check flags: Check if the Tx buffer is not full. If not, load the message; else, it finishes the task.
  - Load: The message is put on a Tx buffer (a FIFO configured as a transmit one). The header of the frame is set with the Control Register (CiFIFOCON) corresponding to the Tx channel. The address in RAM memory where to write the message is read from the User Address Register (CiFIFOUA) and is written, and the status is updated and can be read from the Status Register (CiFIFOSTA).
- Rx: Reception of message. This task is also divided in separate subtasks:
  - Check flags: Check if the Rx buffer is not empty. If not, proceeds to receive task; else, skips it.
  - Receive: The status is read from the CiFIFOSTA register corresponding to the Rx channel; the memory address where to read the message is get from the CiFIFOUA register; the header of the message is set to the CiFIFOCON register, and the status is updated.
- *Request/Wait config*: Sets the operation mode of the controller to Configuration Mode, and waits until the operation is done. Then, it resets the device and the routine restarts from the *Init* state.

7 This is because RAM can only be accessed in multiples of 4 bytes

```
// Assemble transmit message: CAN FD Base frame with BRS, 64 data
    bytes
CAN_TX_MSGOBJ txObj;
uint8_t txd[MAX_DATA_BYTES];

// Initialize ID and Control bits
txObj.word[0] = 0;
txObj.word[1] = 0;

txObj.bF.id.SID = txCounter; // Standard or Base ID
txObj.bF.id.EID = 0;

txObj.bF.ctrl.FDF = 1; // CAN FD frame
txObj.bF.ctrl.BRS = 1; // Switch bit rate
txObj.bF.ctrl.IDE = 0; // Standard frame
txObj.bF.ctrl.RTR = 0; // Not a remote frame request
txObj.bF.ctrl.DLC = CAN_DLC_64; // 64 data bytes
// Sequence: doesn't get transmitted, but will be stored in TEF
txObj.bF.ctrl.SEQ = 1;
```

Listing 4: Tx object (representation of the frame) initialization.

5.5 Python library: *canfdlib*

In order to design the Python library, the functionalities of the Microchip library are taken and translated to Python code. Given the differences in both languages and how they work, this is not a trivial task.

First of all, the library needs the SPI functions in order to communicate. The best option for implementing the SPI library is creating a class, named *canfdlib*, that inherits all the SPI functions in the modified Adafruit library, and adding the functions described in Section 5.4.3. A separate file with constant values is made, as well as another for registers definitions. The main loop will be discussed, but it will not be implemented in the main library, but rather in the GUI application. Finally, this methodology allows to have several connections simultaneously, having them in seperate instances of the class.

5.5.1 *canfdlib*: Registers

The registers can not be defined in the same way as in the C library, as Python does not directly support the *struct* and *union* datatypes. At first, the *ctypes* library was used, which allows to create classes with the same behaviour as the C data types, allowing to create a variable with a fixed bit length and that can be accessed as a 32-bit integer, a 4 byte array, or bitwise, using the same structure as in section 5.4.2 (exactly in the same order as written in the C library).

This solution was tested, but suffered a problem: the bit and byte order did not correspond with the ones in the device, neither with the ones in the C library. There are two workarounds for this problem:

- Re-arrange the bit and byte order dynamically in runtime, preserving the same structure as in the C library.

- Re-arrange the bit and byte order manually during definition.

The first solution is not suitable as it would be called each time a register is read or written, and considering the number of operations that can be potentially be made in a unit of time, this would add a bottleneck and slow down the performance of the software. Giving that Python is already not as optimized as C for working directly with memory and low-level instructions, this solution is discarded.

The other solution, in the other hand, would not add any time constraint in the software, as it consists only in changing the order of each bit in each *struct*. However, as the registers were already defined in the initial order (the same as in the C library), re-arranging them would be very time consuming, and it would be confusing for other users, as the orders would not coincide with the ones in the manual.

Finally, the solution adopted was to use the *ctypes* library only for a few variables of internal use, in order to easily access the individual bits. As in these variables the bit fields are the only ones used, its order is irrelevant, and only the value is used. As for all other registers used, a 32-bit word is used as a base value (that depends on the register and the function, and can be a reset value, a defined value, or a read value), and then each necessary bit of this integer is changed according to the context. Although it may be a little cumbersome, this approach preserves the same bit and byte order as in the controller, and the time needed to perform the bit masking is very low.

Nevertheless, these three approaches were tested in order to check any significant differences in time execution. In Code 5 the time test performed is seen, as well as its results. Each of these methods is executed 100000 times and timed. The results show that the proposed approach is faster than the others, even faster that instantiating the class in the correct bit order. However, the results are very similar, and repeating the test, sometimes it had the opposite results. Nevertheless, the proposed approach is maintained as it requires less manual work in assembling the correct word.

```
def reverse(n):
    array = [int(hex(int(n) >> i & 0xff).replace('L', ''), 16) for i
     in (24, 16, 8, 0)]
    new = []
    for byte in array:
        aux = bin(byte).replace('0b', '')
        while len(aux) < 8:
            aux = '0' + aux
        new.append(int(aux[::-1], 2))
    return int(binascii.hexlify(bytearray(new)), 16)

def set_bit(v, index, x):
  mask = 1 << index
  v &= ~mask
  if x:
    v |= mask
  return v

def method1():
    # Struct register.
    # Reverse the bit order in each byte of the word
    reg = REG_CiCON() # register instance
    reg.IsoCrcEnable = 1 # bit is changed
    return reverse(reg.word) #result to write

def method2():
    # Change bit of word
    word = 0
    return set_bit(word, 5, 1) #result to write

def method3():
    # register instance with modified bit order
    reg = REG_CiCON()
    reg.IsoCrcEnable = 1
    # reg.word is result to write

if __name__ == '__main__':
    print(timeit.timeit("method1()", setup="from __main__ import
    method1", number=100000))
    print(timeit.timeit("method2()", setup="from __main__ import
    method2", number=100000))
    print(timeit.timeit("method3()", setup="from __main__ import
    method3", number=100000))

>> 1.44910316838
>> 0.0353271315938
>> 0.0401096362199
```

Listing 5: Time test performed. Result is in seconds.

5.5.2 *canfdlib*: CAN-FD functions

The functions of the C library are written in the class as methods. From these, the different tasks are encapsulated as functions as well and made methods of the class. Most of the variables have the same role and type, but registers have been changed and discussed in 5.5.1. Also, some values, such as length of messages and channels are now attributes of the class and can be modified in runtime.

In Code 6, the `__init__` method of the class is shown. Here, the `__init__` method of the SPI class defined in the Adafruit library is called. After that, some internal parameters are defined, such as states, configurable parameters and some register definitions. To initialize the *canfdlib* needs the following parameters:

- *ft232h*: An instance of a ft232h object from the Adafruit library. This object creates the connection with the MPSSE cable.
- *cs*: Chip Signal.
- *max_speed_hx*: Maximum SPI speed.
- *mode*: SPI mode.
- *bitorder*: Bit order (MSB or LSB).
- *SPI_DEFAULT_BUFFER_LENGTH*: Default length of the SPI buffer.
- *SPI_MAX _BUFFER_LENGTH*: Maximum length of the SPI buffer.
- *SPI_BAUDRATE*: Baud rate.

```
def __init__(self, ft232h, cs, max_speed_hz, mode, bitorder,
   SPI_DEFAULT_BUFFER_LENGTH, SPI_MAX_BUFFER_LENGTH, SPI_BAUDRATE):
    # Call __init__ from parent class
    super(CANFD_SPI, self).__init__(ft232h, cs, max_speed_hz, mode,
    bitorder)

    # Internal parameters
    self.SPI_DEFAULT_BUFFER_LENGTH = SPI_DEFAULT_BUFFER_LENGTH
    self.SPI_MAX_BUFFER_LENGTH = SPI_MAX_BUFFER_LENGTH
    self.SPI_BAUDRATE = SPI_BAUDRATE

    self.clk = CAN_SYSCLK_40M
    self.txFromFlash = True
    self.switchChanged = True
    self.ramInitialized = False

    # Config internal registers
    self.can_config = REG_CAN_CONFIG()

    self.rxFlags = CAN_RX_FIFO_NO_EVENT
    self.txFlags = CAN_RX_FIFO_NO_EVENT
    self.errorFlags = CAN_ERROR_FREE_STATE

    self.txConfig = CAN_TX_FIFO_CONFIG()
    self.rxConfig = CAN_RX_FIFO_CONFIG()

    # Message objects
    self.txObj = CAN_TX_MSGOBJ()
    self.rxObj = CAN_RX_MSGOBJ()

    self.txCounter = 0

    self.transmitBuffer = []
    self.receiveBuffer = []

    # Configurable parameters
    self.opMode = NORMAL_MODE
    self.selectedBitTime = CAN_500K_2M
    self.txchannel = CAN_FIFO_CH2
    self.rxchannel = CAN_FIFO_CH1
    self.txdlc = CAN_DLC_64

    # State of the program.
    self.state = "idle"  # APP_STATE_INIT
```

Listing 6: init method of *canfdlib*

The functions described in Section 5.4.3 are written in Python. Some are practically the same, and others have been written in a more *pythonic* way, (using lists comprehension for example), and others have been completely changed, as Python does not support directly the use of pointer variables and union data types like C. The main changes done between languages are the following:

- SPI communication is changed completely, and to full-duplex (original) to half-duplex.
- Returned data from SPI is transformed differently: in C, pointer variables are used to access memory directly and bitwise operations are

done to store the data in a variable as an integer or array of integers; in Python, the SPI returns a `bytearray` object of the length indicated.

- In Python, the returned value of the SPI is cast to an hex using the *binascii* and then to a integer or list of integers.

- In the original configuration functions, all registers were defined in software as a combination of *union* and *struct* data types, a default value was given to its word, then it was modified at bit level, and finally written as a 32-bit integer; in Python, the bits of interest are changed in a variable containing the default value as an integer, and then is written.

- In the *OperationModeSelect* function, in order to modify only the bits of interest, the C library does the following operations:

  ```
      d &= ~0x07;
      d  |= opMode;
  ```

  In Python, this is changed to gain clarity to the following form, which does the same operation:

  ```
      byte = (0xF8 & byte) + mode
  ```

- *switch* clauses are substituted by cascades of *if-elif-else* statements.

- Lists comprehension are used when possible to substitute *for* loops.

All other functions retain the same structure, with the obvious changes in syntax between languages (such as the declaration of variables, which in C are declared explicitly along with its type, and in Python this is not needed).
More methods are defined in this class, which perform all the operations needed in the tasks, and will be discussed in Section 5.5.3.

5.5.3 *canfdlib*: *Main* tasks

The tasks in the *main* function are practically the same as the ones discussed in 5.4.4, but a loop will be implemented in the GUI. Each task has been encapsulated into a single function. The tasks in the *canfdlib* library have the following changes with respect to the C library:

- *Init*: In the C library, the oscillator frequency is divided, and some objects used, such as TEF, interruptions and GPIOs are configured. None of these functions is used in the Python library (they could be added but they are not used in the scope of this project).

- Reset: This task does not change.

- *Configs*: Only the implementation of the *configure* function is changed, but the task is the same.

- RAM test: Does not change.

- Register test: Does not change.

- Tx: A new method is defined that accepts one parameter (the message to transmit, as an array of bytes) and calls the following methods:
  - *Config*: The header of the message is configured.
  - Check flags: Task does not change.
  - Load: The message is written in FIFO, and the status is updated.
- Rx: A new method is also defined for Rx. It returns the received message as an array of bytes.
  - Check flags: Does not change.
  - Receive: The message is read from FIFO and returned.
- *Request/Wait config*: Does not change.

The new state machine of the program (with the loop implemented for later use) is shown in Figure 14. The tasks are the same as before, but the logic of the system is changed so after each task finishes, it goes to a new task, *Change state*, where the state of the system is updated. The state can be also changed by user input, and the value of the state variable determines which task to execute next. As the main loop is not implemented directly in the library, but rather in the GUI application, it can be configured by the user as desired given the task functions (it can be modified to add more configurations, interruptions can be implemented by other means, etc).

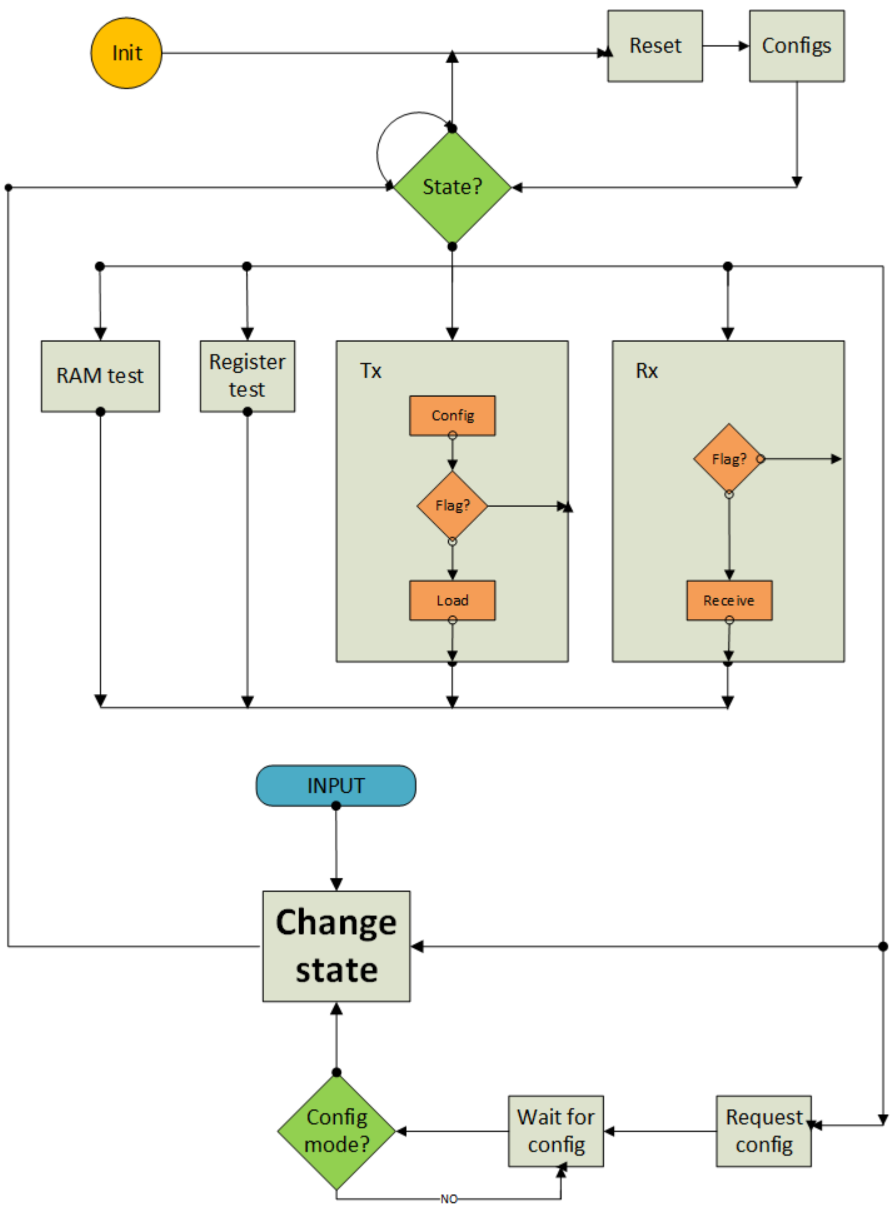


Figure 14: State machine of the *main* function in the *canfdlib* library. Source: own.

5.5.4 GUI

The GUI application is designed in order to test the library and to have a minimum, verifiable and complete example. It is a separate entity from the library, so it can used and modified without changing the base files.

The *main* loop is implemented, where the state is check periodically and the corresponding task is executed. Several elements are needed in the GUI so as to change the state of the program, as well as to configure some parameters. These elements are the following ones:

- Connect button: To establish connection with the device.
- Reset button: Button to reset the device when it is clicked.
- Stop button: Halts the program.

- Tx button and input box: A message that the user wants to transmit is written in the input box as an array of bytes.
- Rx box and button: A text box where the received message is shown, as well as program messages and other information.
- Clear button: Button to clear the Rx text box.
- Config droplist and button: To select a operation mode of the controller and configuring it.
- DLC droplist and button: To select the length of the messages.
- Tx and Rx channel input and button: To change the channels used for transmition and reception.

In Figure 15, a template of the position of this elements is shown.

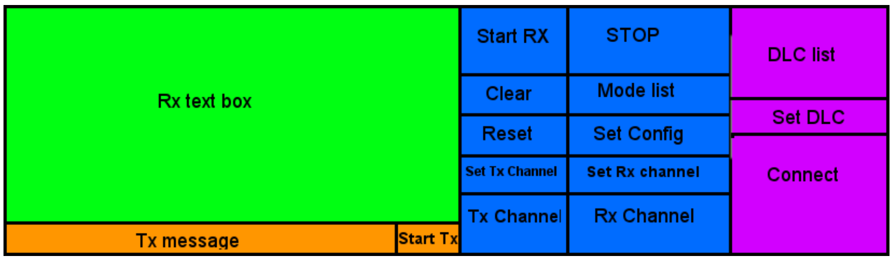


Figure 15: Template of the GUI. Source: own.

The *mttkinter* package is used, as it allows to develop graphical elements very easily and has support for parallel executions. A class is created to encapsulate the functionalities of the GUI. Inside this class, all its graphical elements will be attributes, and an instance of the *canfdlib* is made an attribute and used for accessing its methods.

In Code 7 the `__init__` method is shown. Here, all the elements are positioned according to the template. In order to do so, several frames are defined, corresponding to the colored zones in the template. Each frame can have its own number of rows and columns, so the distribution can be stylized. For each element, positioning and size are passed. Finally, if *debug* was set to `False`, the *main* loop function is attached to run after each loop of the mttkinter root function[8], and then tries to connect to the device. If not, it writes an error message and the controller should be manually connected. If *debug* was set to `True`, the device is not connected automatically, and no loop is attached.

```
    def __init__(self, inputDict=None, debug=False):
        self.inputDict = inputDict
        self.debug = debug

        self.rxd = []
        self.txd = []

        self.window = tk.Tk()
        self.window.title(window_title)
```

8 The mttkinter executes a loop so as to refresh the GUI elements.

```
        self.window.geometry(window_size)

        self.right_frame = tk.Frame(self.window, width=450, height
=100)
        self.left_frame = tk.Frame(self.window, width=250, height
=100)
        self.corner_frame = tk.Frame(self.window, width=100, height
=20)
        self.extra_frame = tk.Frame(self.window, width=30, height
=100)
        self.window.grid_columnconfigure(1, weight=1)
        self.right_frame.grid(row=0, column=1, sticky="nsew")
        self.left_frame.grid(row=0, column=0, sticky="nsew")
        self.corner_frame.grid(row=1,column=0,sticky="sw")
        self.extra_frame.grid(row=0, column=2)

        # textbot for rx
        self.rx_box_scrollbar = tk.Scrollbar(self.left_frame)
        self.rx_box_text = tk.Text(self.left_frame, height=
rx_textbox_height, width=rx_textbox_width)
        self.rx_box_scrollbar.grid(column=1, row=0,  sticky=tk.N+tk.
S+tk.W)
        self.rx_box_text.grid(column=0, row=0)
        self.rx_box_scrollbar.config(command=self.rx_box_text.yview)
        self.rx_box_text.config(yscrollcommand=self.rx_box_scrollbar
.set)

        # rx button
        self.receive_start_button = tk.Button(self.right_frame, text
="Start RX", command = self.receive)
        self.receive_start_button.grid(column=0,row=0, pady=5)

        # clear rx box windows button
        self.clear_button = tk.Button(self.right_frame, text="Clear"
, command=self.clear)
        self.clear_button.grid(column=0, row=1, pady=5)

        # tx button
        self.transmit_start_button = tk.Button(self.corner_frame,
text="Start TX", command=self.transmit)
        self.transmit_start_button.grid(column=2,row=0)

        # tx message
        self.tx_msg = tk.Entry(self.corner_frame, width=tx_msg_width
)
        self.tx_msg.grid(column=1,row=0)

        # tx label
        self.txlbl = tk.Label(self.corner_frame, text="TX Message:")
        self.txlbl.grid(column=0, row=0)

        # rx channel button
        self.rx_channel_button = tk.Button(self.right_frame, text="
Set RX channel", command=self.setRXchannel)
        self.rx_channel_button.grid(column=1, row=3, pady=5)

        # rx channel
        self.rx_channel = tk.Entry(self.right_frame, width=10)
        self.rx_channel.grid(column=1, row=4, pady=5)

        # tx channel button
        self.tx_channel_button = tk.Button(self.right_frame, text="
Set TX channel", command=self.setTXchannel)
        self.tx_channel_button.grid(column=0, row=3, pady=5)

        # tx channel
        self.tx_channel = tk.Entry(self.right_frame, width=10)
        self.tx_channel.grid(column=0, row=4, pady=5)

        # reset button
        self.reset_button = tk.Button(self.right_frame, text="Reset
Device", command=self.reset)
        self.reset_button.grid(column=0,row=2, pady=5)

        # opMode droplist and button
        OPTIONS = ["NORMAL_MODE","SLEEP_MODE","
INTERNAL_LOOPBACK_MODE","LISTEN_ONLY_MODE","CONFIGURATION_MODE",
"EXTERNAL_LOOPBACK_MODE","CLASSIC_MODE","RESTRICTED_MODE","
INVALID_MODE"]

        self.droplist = tk.StringVar(self.left_frame)
        self.droplist.set(OPTIONS[0])  # default value

        w = tk.OptionMenu(self.right_frame, self.droplist, *OPTIONS)
        w.grid(column=1,row=1, padx=5)

        self.opmode_button = tk.Button(self.right_frame, text="Set
config mode", command=self.changemode)
        self.opmode_button.grid(column=1,row=2, pady=5)

        # stop button
        self.stop_button = tk.Button(self.right_frame, text="STOP",
command=self.stop)
        self.stop_button.grid(column=1, row=0, pady=5)

        # connect button
        self.connect_button = tk.Button(self.extra_frame, text="
CONNECT", command=self.connect)
        self.connect_button.grid(column=0, row=2, pady=0) # enlarge
        self.connect_button.config(height=3, width=15)

        # dlc droplist and button
        OPTIONS_dlc = ["CAN_DLC_0","CAN_DLC_1","CAN_DLC_2","
CAN_DLC_3","CAN_DLC_4","CAN_DLC_5","CAN_DLC_6","CAN_DLC_7","
CAN_DLC_8","CAN_DLC_12","CAN_DLC_16","CAN_DLC_20","CAN_DLC_24","
CAN_DLC_32","CAN_DLC_48","CAN_DLC_64"]

        self.droplist_dlc = tk.StringVar(self.extra_frame)
        self.droplist_dlc.set(OPTIONS_dlc[-1])  # default value

        w_dlc = tk.OptionMenu(self.extra_frame, self.droplist_dlc, *
OPTIONS_dlc)
        w_dlc.grid(column=0, row=0, padx=5)

        self.dlc_button = tk.Button(self.extra_frame, text="Set DLC"
, command=self.changedlc)
        self.dlc_button.grid(column=0, row=1, pady=5)

        self.canfd = None

        if self.debug:
            # self.window.after(1000, self.dummy_main)
```

```
                pass
            else:
                try:
                    self.connect()
                except RuntimeError:
                    self.rx_box_text.insert(tk.END, "Device not ready.
        Connect manually." + '\n')
                self.window.after(0, self.main)
            self.window.mainloop()
```

Listing 7: init method of the GUI

This class accepts two parameters: a input dictionary containing all the necessary parameters needed to create an instance of the *canfdlib* class, and a boolean value that puts the GUI in *debug* mode. In this mode, the device does not execute the *main* loop, and the device must be manually initialized. Each button has a function associated, which is a method of the class. These functions have all a similar structure:

- If needed, a parameter is read from the input box and passed to a *canfdlib* method.
- The state is changed if the function uses the device (this does not happen in the *Clear* method, for example). The *main* loop then executes the necessary task.
- If a result is returned, this is printed in the Rx box.

The resulting GUI is shown in Figure 16.

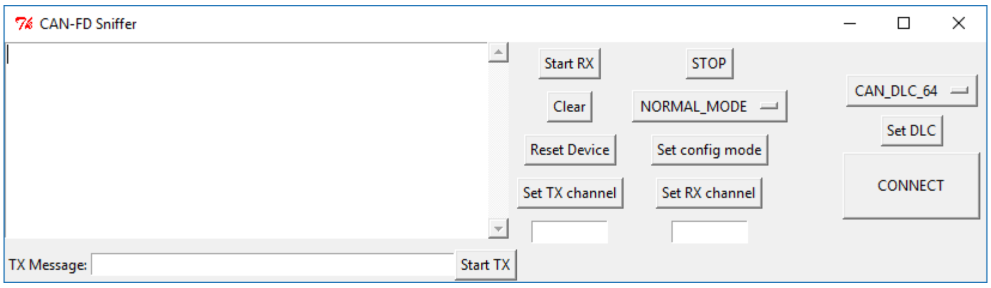


Figure 16: GUI. Source: own.

The tests explained in Chapter 6 are also implemented in the GUI, and can be easily accessible writing `testX` in the *Tx message* input box, where **X** is the number of the test (as there are several tests integrated).

6 Tests

In order to check the correct behaviour of the library and the GUI, several tests are defined. Several tests are done to check other functions, and will be discussed in this chapter. The tests are first written in a plan script, and when they successfully completed, they are encapsulated in a function and then grouped in a dictionary of functions so they can be easily accessible from external files. Also, this facilitates the task of implementing the tests in the GUI. In this Section, the shown tests are plain scripts, and in the Annex, the encapsulated version is present.

All numbers have a fixed width of 8 or 32 bits, so to compare them to other references (such as the values in manual), extra 0s should be added when necessary so as to have the correct bit length.

6.1 SPI read

This test has already been shown in Code 2. It uses the Adafruit library to read a register in the device. Comparing the values with the reset values of the manual, it is seen that the SPI reading operation works well.

6.2 SPI write

This test is shown in Code 8. The register CiCON, located at address 0x000 is read in order to check the initial value, and then some data is written on it. Finally, the register is read once again in order to check if the data was written.

```
spiTransmitBuffer = []
address = 0x000
spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >> 8)
   & 0xF))
spiTransmitBuffer.append(address & 0xFF)
spi._assert_cs()
spi.write(spiTransmitBuffer)
response_0 = spi.read(4)
spi._deassert_cs()
print ('CiCON register before writing:')
print (binascii.hexlify(response_0))
print(list(response_0))

spiTransmitBuffer = []
data = [0,0,0,0]
addressW = 0x000
spiTransmitBuffer.append((cINSTRUCTION_WRITE << 4) + ((addressW >>
   8) & 0xF))
spiTransmitBuffer.append(addressW & 0xFF)
spiTransmitBuffer = spiTransmitBuffer + data

spi._assert_cs()
spi.write(spiTransmitBuffer)
spi._deassert_cs()

spiTransmitBuffer = []
spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >> 8)
    & 0xF))
spiTransmitBuffer.append(address & 0xFF)

spi._assert_cs()
spi.write(spiTransmitBuffer)
response_1 = spi.read(4)
spi._deassert_cs()
print ('CiCON register modified:')
print (binascii.hexlify(response_1))
print(list(response_1))

# Reset
spiTransmitBuffer = []
spiTransmitBuffer.append(cINSTRUCTION_RESET << 4)
spiTransmitBuffer.append(0)
spi._assert_cs()
spi.write(spiTransmitBuffer)
spi._deassert_cs()

spiTransmitBuffer = []
spiTransmitBuffer.append((cINSTRUCTION_READ << 4) + ((address >> 8)
    & 0xF))
spiTransmitBuffer.append(address & 0xFF)

spi._assert_cs()
spi.write(spiTransmitBuffer)
response_1 = spi.read(4)
spi._deassert_cs()
print ('CiCON default:')
print (binascii.hexlify(response_1))
print(list(response_1))

>> CiCON register before writing:
>> 0x60079804
>> [96, 7, 152, 4]
>> Data to write (4 bytes):
>> [0, 0, 0, 0]
>> CiCON register modified:
>> 0x00000000
>> [0, 0, 0, 0]
>> CiCON default:
>> 0x60079804
>> [96, 7, 152, 4]
```

Listing 8: SPI write test

In order to change any value written to its default, the *reset* command is written before each test. In order to check the correct behaviour of the *reset*, it is written after this test and the register is read. As it is seen, the register has returned to its default value.

6.3 *canfdlib* read

In this test, the functions developed in the library for reading are tested. The register CiCON is read in several ways:

- As an array of bytes with *readByteArray*.
- As a 32-bit word with *readWord*.
- Its 1st byte only, with *readByte*.
- As a 32-bit word array, along with the next register CiNBTCFG with *readWordArray*.

The results are shown in Code 9. All the values read have the same values as the default ones.

```
address = 0x000
word = canfd.readWord(address)
byte = canfd.readByte(address)
byteArr = canfd.readByteArray(address, 4) # read 4 bytes
wordArr = canfd.readWordArray(address,2) # read 2 words
print ('Reading CiCON as 32-bit word with CAN FD lib, using readWord
    :')
print(word)
print(hex(word))

print ('Reading 1st byte of CiCON with CAN FD lib, using readByte:')
print(byte)
print(hex(byte))
print ('Reading CiCON as array of bytes with CAN FD lib, using
    readByteArray:')
print(list(byteArr))
print ('Reading CiCON and CiNBTCFG as array of two 32-bit words with
    CAN FD lib, using readWordArray:')
print(list(wordArr))

Reading CiCON as 32-bit word with CAN FD lib, using readWord:
>> 1611110404
>> 0x60079804
>> Reading 1st byte of CiCON with CAN FD lib, using readByte:
>> 96
>> 0x60
>> Reading CiCON as array of bytes with CAN FD lib, using
    readByteArray:
[96, 7, 152, 4]
>> Reading CiCON and CiNBTCFG as array of two 32-bit words with CAN
    FD lib, using readWordArray:
>> [1611110404, 252657152]
```

Listing 9: *canfdlib* read test.

6.4 *canfdlib* write

In this test, data is written in the register CiCON in several ways:

- An array of bytes with *writeByteArray*.
- A 32-bit word with *writeWord*.
- A single byte with *writeByte*.
- A word array with *writeWordArray*

The value is read after each writing, and the device is resetted so that the value of the register is set to default. Results are seen in Code 10. As it can be seen, the read values coincide with the written data.

```
address = 0x000
word = canfd.readWord(address)
print ('Reading CiCON:')
print(word)
write_word = 0x600798F4
print("Word to write: ")
print(write_word)
canfd.writeWord(address, write_word)
word = canfd.readWord(address)
print ('Reading CiCON with 0x600798F4 written on it:')
print(word)
canfd.reset()
write_byte = 0x6F
canfd.writeByte(address, write_byte)
word = canfd.readWord(address)
print ('Reading CiCON with 0x00 written on its 1st byte:')
print(word)
canfd.reset()
write_byte_array = [0x60, 0x07, 0x98, 0xF4]
canfd.writeByteArray(address, write_byte_array)
word = canfd.readWord(address)
print ('Reading CiCON with [0x60, 0x07, 0x98, 0xF4] array written on
     it (4 bytes):')
print(word)
canfd.reset()
write_word_array = [0x600798F4, 0x7f0f3eff]
canfd.writeWordArray(address, write_word_array)
word = canfd.readWordArray(address, 2)
print ('Reading CiCON and CiNBTCFG with [0x600798F4, 0x7f0f3eff]
    written on it:')
print(word)

>> Reading CiCON:
>> 1611110404
>> Word to write:
>> 1611110644
>> Reading CiCON with 0x600798F4 written on it:
>> 1611110644
>> Resetting...
>> Reading CiCON with 0x00 written on its 1st byte:
>> 1862768644
>> Resetting...
>> Reading CiCON with [0x60, 0x07, 0x98, 0xF4] array written on it
    (4 bytes):
>> 1611110644
>> Resetting...
>> Reading CiCON and CiNBTCFG with [0x600798F4, 0x7f0f3eff] written
    on it:
>> [1611110644, 2131705599]
```

Listing 10: *canfdlib* write test.

6.5 RAM test

The RAM test is executed, where random bytes are written in RAM with various lengths of messages. The test returns -1 if some value is mismatched, and stops the test. If nothing goes wrong, it returns 1. In Code 11 the test is shown. The results can be seen at the Annex.

```
def ramTest(self):
    # verify R/W
    for length in range(4, MAX_DATA_BYTES + 1, 4):
        txd = [(randint(0, RAND_MAX) & 0xFF) for e in range(0,
    length)]
        #print("Data written on RAM: {}".format(txd))
        self.writeByteArray(cRAMADDR_START, txd)
        rxd = self.readByteArray(cRAMADDR_START, length)
        #print("Data read on RAM: {}".format(rxd))
        for i in range(0, length):
            good = txd[i] == rxd[i]
            if not good:
                print("Data mismatch!")
                return -1
    return 1
result = canfd.ramTest()
if result == -1:
    print("RAM test failed!")
else:
    print ('RAM test succesful!')
```

Listing 11: RAM test

6.6 Register test

This test is very similar to the RAM test, so its detailed code is not presented. In Code 12 the test is shown. The results can be seen at the Annex.

```
result = canfd.registerTest()
if result == -1:
    print("Register test failed!")
else:
    print ('Register test succesful!:')
```

Listing 12: Register test

6.7 Operation mode test

In this test, the operation mode of the device is changed and then is read from the device in order to check if the change was successful. Code 13 shows the test and its results. The mode after reset is checked, then changed a few times, and finally, the mode after initialization is seen in order to check it is correct. In Table 4, the correspondence between the mode and its associated integer value is seen.

Mode	Value (int)	Value (bit)
NORMAL_MODE	0	000
SLEEP_MODE	1	001
INTERNAL_LOOPBACK_MODE	2	010
LISTEN_ONLY_MODE	3	011
CONFIGURATION_MODE	4	100
EXTERNAL_LOOPBACK_MODE	5	101
CLASSIC_MODE	6	110
RESTRICTED_MODE	7	111

Table 4: Mode values. Source: [6]

```
mode = canfd.operationModeGet()
print("After reset mode: {}".format(mode))
print("selecting NORMAL_MODE mode")
canfd.operationModeSelect(NORMAL_MODE)
mode = canfd.operationModeGet()
print("Device mode: {}".format(mode))
print("selecting INTERNAL_LOOPBACK_MODE mode")
canfd.operationModeSelect(INTERNAL_LOOPBACK_MODE)
mode = canfd.operationModeGet()
print("Device mode: {}".format(mode))
print("selecting CONFIGURATION_MODE mode")
canfd.operationModeSelect(CONFIGURATION_MODE)
mode = canfd.operationModeGet()
print("Device mode: {}".format(mode))
print("selecting mode after init")
canfd.initialize()
mode = canfd.operationModeGet()
print("Device mode: {}".format(mode))

After reset mode: 4
selecting NORMAL_MODE mode
Device mode: 0
selecting INTERNAL_LOOPBACK_MODE mode
Device mode: 2
selecting CONFIGURATION_MODE mode
Device mode: 4
selecting mode after init
ECC enabled
RAM initialized
Device mode: 0
```

Listing 13: Operation mode test

6.8 Tx-Rx tests

Several tests are made in order to check the transmission and reception. In order to do them, the following devices are used:

1. Host PC: The PC where this project has been developed and the library is running.
2. MCP2517FD click + MPSEE cable: The controller is connected to the host PC and to a CAN-FD model with the DB9 connector.
3. Oscilloscope: It is used to verify that signals are transmitted.
4. CAN-FD model: This model, developed by the Electronic Engineering department, is able to transmit data using CAN-FD frames in order to simulate the ECU system of a automobile, and has a sniffer system incorporated, so that data can be seen with a suitable device, in a similar fashion to the project developed here. It also works as an intermediate device between both computers so they can communicate.
5. Transmitter-receiver (TRx) PC: This PC is connected to the model so it can read and write messages to it, as well as to the host PC. It has a custom software to transmit and receive messages.

These devices are shown in Figure 17.

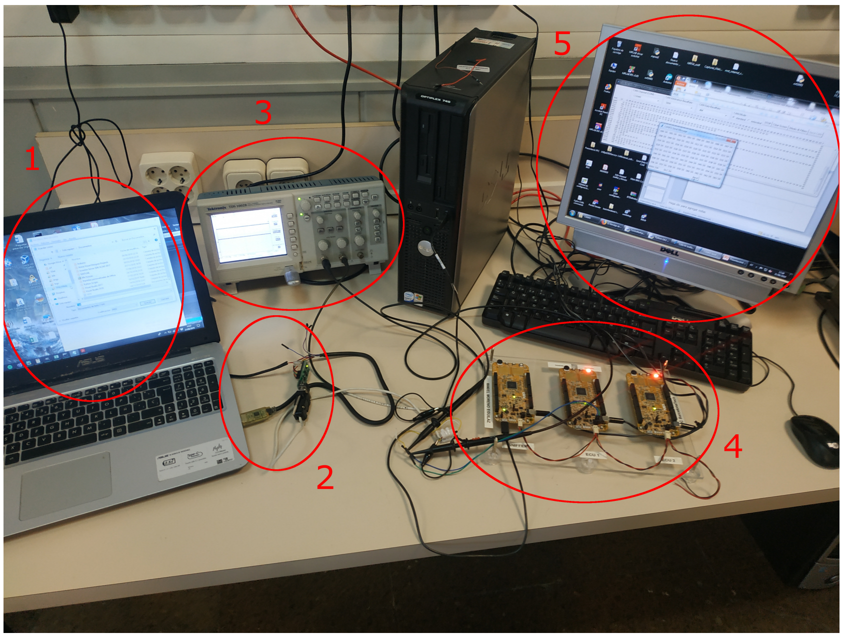


Figure 17: Devices used in testing. Source: own.

6.8.1 Test 1: Tx

In this test, a single message of 64 bytes is transmitted from the host to the TRx PC. The code for this test is shown in 14. The data received in the TRx PC is shown on screen, and with the oscilloscope, it can be verified that data is being transmitted. In Figure 18 the oscilloscope signal is seen, and in Figure 19 the data received is shown.

```
canfd.initialize()
txd = range(0, canfd.dlcToDataBytes(canfd.txdlc)) # 64 bytes of data
print("(TEST) message to transmit: {}".format([hex(a) for a in txd])
    )
canfd.transmitMessageTasks(txd)

>> (TEST) message to transmit: ['0x0', '0x1', '0x2', '0x3', '0x4', '
    0x5', '0x6', '0x7', '0x8', '0x9', '0xa', '0xb', '0xc', '0xd', '0
    xe', '0xf', '0x10', '0x11', '0x12', '0x13', '0x14', '0x15', '0
    x16', '0x17', '0x18', '0x19', '0x1a', '0x1b', '0x1c', '0x1d', '0
    x1e', '0x1f', '0x20', '0x21', '0x22', '0x23', '0x24', '0x25', '0
    x26', '0x27', '0x28', '0x29', '0x2a', '0x2b', '0x2c', '0x2d', '0
    x2e', '0x2f', '0x30', '0x31', '0x32', '0x33', '0x34', '0x35', '0
    x36', '0x37', '0x38', '0x39', '0x3a', '0x3b', '0x3c', '0x3d', '0
    x3e', '0x3f']
```

Listing 14: Test 1

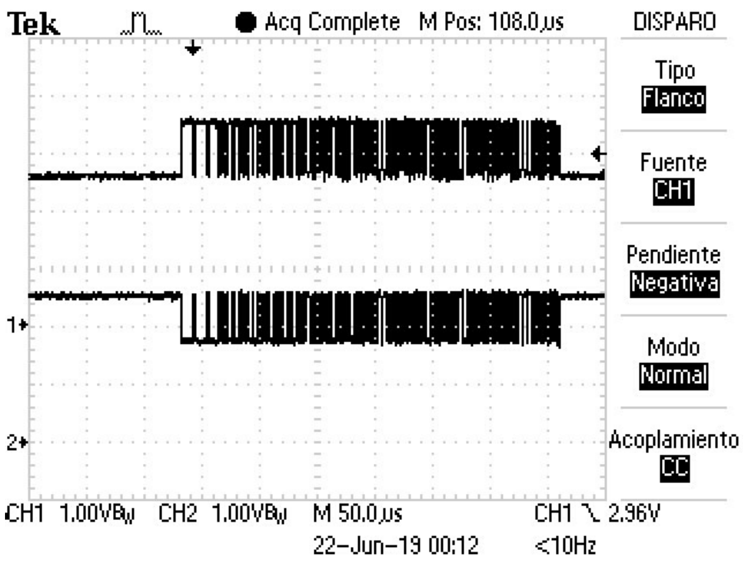
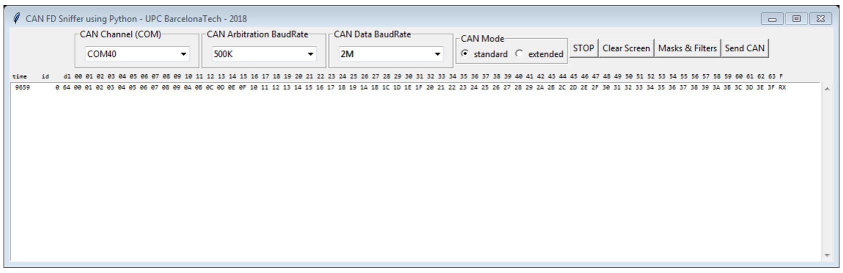


Figure 18: Oscilloscope signal for test 1. Source: own.



```
time    id    dl 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 F
 9659          0 64 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F RX
```

Figure 19: Data received for test 1. Source: own.

6.8.2 Test 2: Tx with different DLC

In this test, a single message of 32 bytes is transmitted from the host to the TRx PC. The code for this test is shown in 15. In Figure 20 the oscilloscope signal is seen, and in Figure 21 the data received is shown.

```
canfd.initialize()
canfd.txdlc = 13 # DLC value for payload of 32 bytes
txd = range(0, canfd.dlcToDataBytes(canfd.txdlc)) # 32 bytes of data
txd = txd[::-1]
print("(TEST) message to transmit: {}".format([hex(a) for a in txd])
    )
canfd.transmitMessageTasks(txd)

>> (TEST) message to transmit: ['0x1f', '0x1e', '0x1d', '0x1c', '0
    x1b', '0x1a', '0x19', '0x18', '0x17', '0x16', '0x15', '0x14', '0
    x13', '0x12', '0x11', '0x10', '0xf', '0xe', '0xd', '0xc', '0xb',
     '0xa', '0x9', '0x8', '0x7', '0x6', '0x5', '0x4', '0x3', '0x2',
    '0x1', '0x0']
```

Listing 15: Test 2

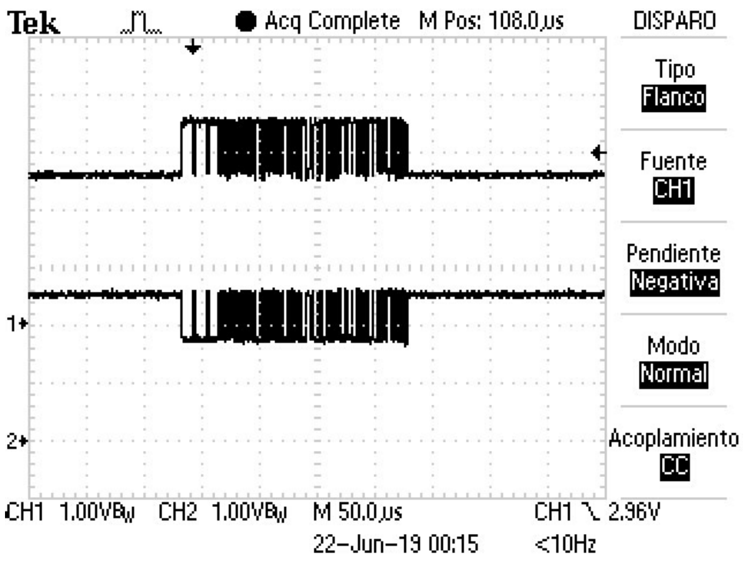
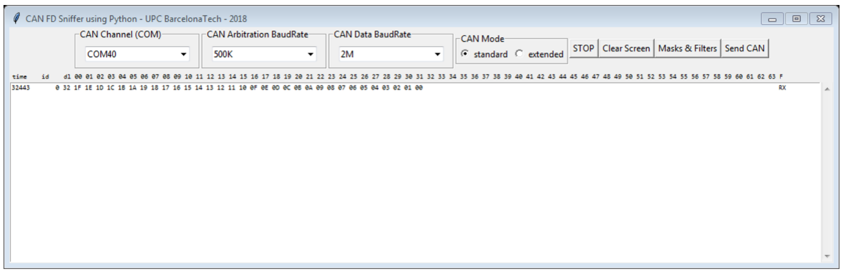


Figure 20: Oscilloscope signal for test 2. Source: own.

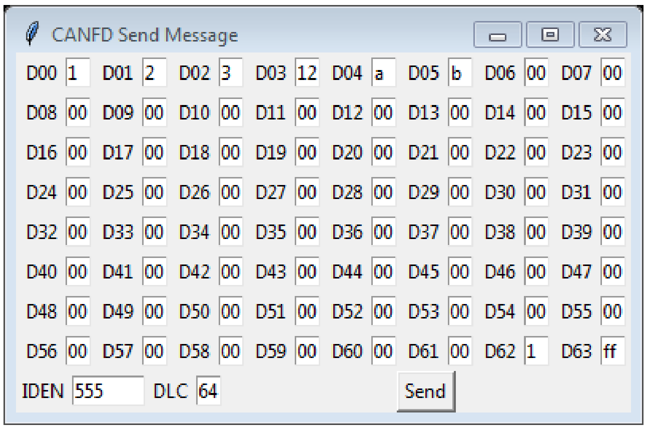


```
time    id   dl 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 F
32443        0 32 1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00                                                                                   RX
```

Figure 21: Data received for test 2. Source: own.

6.8.3 Test 3: Rx

In this test, a single message of 64 bytes is transmitted from the TRx PC to the host PC. In Figure 22 the data sent is shown, in Figure 23 the oscilloscope signal is seen, and in 16 the code for this test is shown, along with the data received.

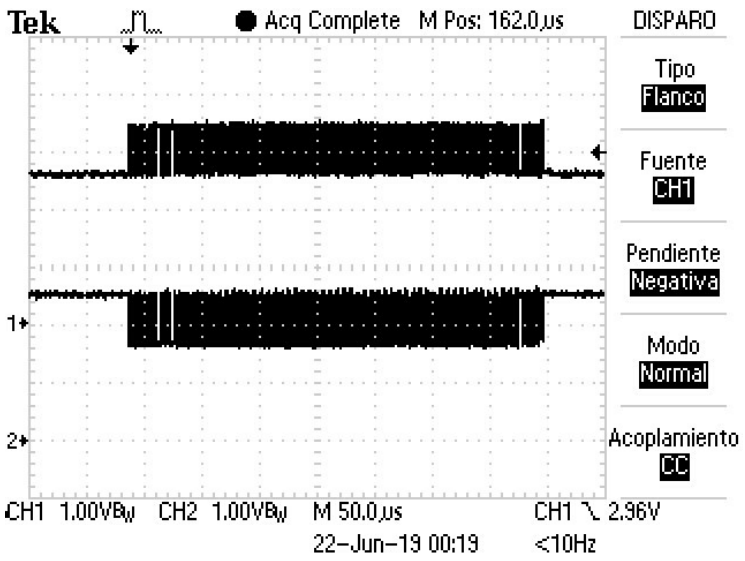


CANFD Send Message

D00	1	D01	2	D02	3	D03	12	D04	a	D05	b	D06	00	D07	00
D08	00	D09	00	D10	00	D11	00	D12	00	D13	00	D14	00	D15	00
D16	00	D17	00	D18	00	D19	00	D20	00	D21	00	D22	00	D23	00
D24	00	D25	00	D26	00	D27	00	D28	00	D29	00	D30	00	D31	00
D32	00	D33	00	D34	00	D35	00	D36	00	D37	00	D38	00	D39	00
D40	00	D41	00	D42	00	D43	00	D44	00	D45	00	D46	00	D47	00
D48	00	D49	00	D50	00	D51	00	D52	00	D53	00	D54	00	D55	00
D56	00	D57	00	D58	00	D59	00	D60	00	D61	00	D62	1	D63	ff

IDEN 555 DLC 64

Figure 22: Data sent for test 3. Source: own.



Tek ● Acq Complete M Pos: 162.0µs

DISPARO
Tipo Flanco
Fuente CH1
Pendiente Negativa
Modo Normal
Acoplamiento CC

CH1 1.00VBW CH2 1.00VBW M 50.0µs CH1 ↘ 2.96V
22-Jun-19 00:19 <10Hz

Figure 23: Oscilloscope signal for test 3. Source: own.

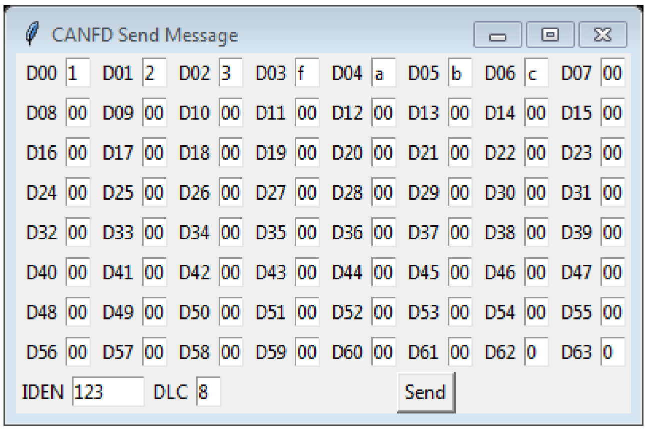
```
canfd.initialize()
while True:
    # receive message
    rxd = canfd.receiveMessageTasks()
    if rxd is not None: # print message when received
        print ("(TEST) received message: {}".format([hex(a) for a in
     rxd]))

>> (TEST) received message: ['0x1', '0x2', '0x3', '0x12', '0xa', '0
    xb', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0
    x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0
    x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0
    x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0
    x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0
    x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0x0', '0
    x0', '0x0', '0x0', '0x1', '0xff']
```

Listing 16: Test 3

6.8.4 Test 4: Rx with different DLC and ID

In this test, a single message of 8 bytes is transmitted from the TRx PC to the host PC, changing the ID from the previous test. In Figure 24 the data sent is shown, in Figure 25 the oscilloscope signal is seen, and in 17 the code for this test is shown, along with the data received.

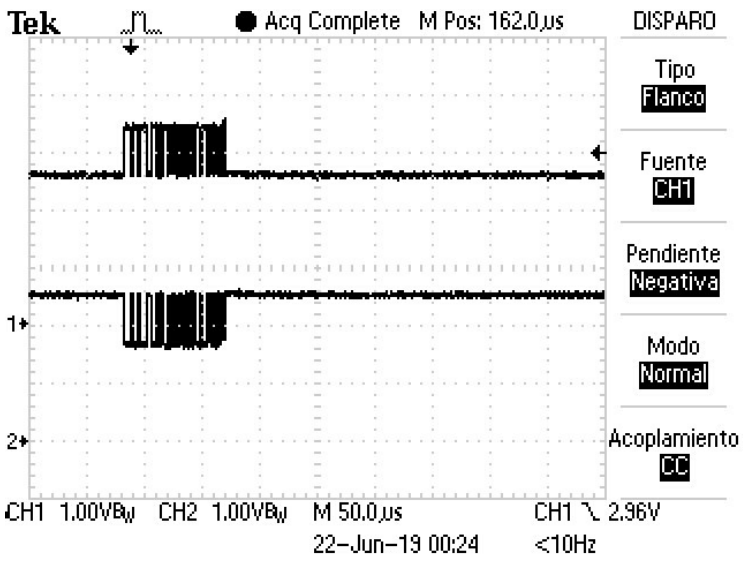


CANFD Send Message

D00	1	D01	2	D02	3	D03	f	D04	a	D05	b	D06	c	D07	00
D08	00	D09	00	D10	00	D11	00	D12	00	D13	00	D14	00	D15	00
D16	00	D17	00	D18	00	D19	00	D20	00	D21	00	D22	00	D23	00
D24	00	D25	00	D26	00	D27	00	D28	00	D29	00	D30	00	D31	00
D32	00	D33	00	D34	00	D35	00	D36	00	D37	00	D38	00	D39	00
D40	00	D41	00	D42	00	D43	00	D44	00	D45	00	D46	00	D47	00
D48	00	D49	00	D50	00	D51	00	D52	00	D53	00	D54	00	D55	00
D56	00	D57	00	D58	00	D59	00	D60	00	D61	00	D62	0	D63	0

IDEN 123 DLC 8 Send

Figure 24: Data sent for test 4. Source: own.



DISPARO
Tipo Flanco
Fuente CH1
Pendiente Negativa
Modo Normal
Acoplamiento CC

CH1 1.00V BW CH2 1.00V BW M 50.0µs CH1 ⌄ 2.96V
22-Jun-19 00:24 <10Hz

Figure 25: Oscilloscope signal for test 4. Source: own.

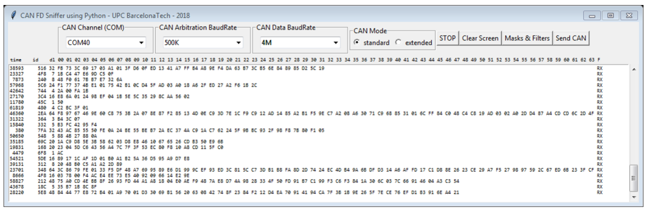
```
canfd.initialize()
while True:
    # receive message
    rxd = canfd.receiveMessageTasks()
    if rxd is not None: # print message when received
        print ("(TEST) received message: {}".format([hex(a) for a in
        rxd]))

>> (TEST) received message: ['0x1', '0x2', '0x3', '0xf', '0xa ', '0
    xb', 'Oxc', '0x0']
```

Listing 17: Test 4.

6.8.5 Test 5: Rx with random data

In this test, several messages transmitted by the model are received. The data has random elements, length and IDs. In Figure 26, part of the data sent is shown, and in 18 the code for this test is shown, along with part of the data received.



CAN Channel (COM): COM40
CAN Arbitration BaudRate: 500K
CAN Data BaudRate: 4M
CAN Mode: [✓ standard] extended

```
time   id  dl 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 F
38593  516 32 FB 73 3C 69 17 03 A1 01 3F D6 0F ED 13 41 A7 FF B4 A8 9E F4 DA 63 B7 3C B5 6E 84 89 85 D2 5C 19      RX
23327  4F8  7 1B C4 47 E6 9D C5 0F      RX
 7873  240  8 48 F0 61 7B B7 E7 32 6A      RX
57968  5C6 24 F1 77 37 4B E1 01 75 42 B1 0C D4 5F AD 03 A0 18 A6 2F ED 27 A2 F6 1B 2C      RX
42642  744  4 2A 00 FA 1B      RX
27170  3C4 16 E8 6A 01 24 98 EF 04 1B 5E 5C 35 29 8C AA 56 02      RX
11780  45C  1 50      RX
61819  480  4 C2 BC 3F 01      RX
46360  2EA 64 F8 97 67 46 9E 60 C8 75 3B 2A 07 8E 87 F2 85 13 4D 0E C9 3D 7E 1C F9 C9 12 AD 14 85 A2 B1 F5 9E C7 A2 08 A6 30 71 C9 68 85 31 01 6C FF 84 C0 48 C4 C8 19 AD 03 02 A0 2D D4 87 A4 CD CD 6C 2D 4F RX
31322  364  3 B4 3C 07      RX
15840  332  5 B3 FC 42 95 F4      RX
  380  7FA 32 43 AC 85 55 50 FE 0A 24 BE 55 BE B7 2A EC 37 4A C9 1A C7 62 24 5F 9B BC 93 2F 98 F8 7B 80 F1 05      RX
50650  548  5 88 4B 27 88 0A      RX
35185  69C 20 1A C9 D8 5E 3B 58 62 03 D8 E8 46 10 67 65 26 CD B3 50 E9 6B      RX
19831  168 20 23 04 5D C6 43 56 A4 7C 7F 3F 53 EC 80 F8 10 A8 CD 11 5F C0      RX
 4479  6F8  1 AC      RX
54521  5DE 16 89 17 1C AF 1D 01 B0 A1 B2 5A 36 D5 95 A9 D7 E8      RX
39131  312  8 20 4B B0 C5 A1 A2 2D 89      RX
23701  348 64 3C 86 79 FE 01 33 F5 DF 48 A7 69 95 89 E6 D1 99 9C EF 93 ED 3C 81 5C C7 3D B1 B8 FA 8D 2D 74 24 EC 4D B4 9A 6B DF D3 14 A6 AF FD 17 C1 D8 8E 26 23 CE 29 A7 F5 27 98 97 59 2C 67 ED 6B 23 3F CF RX
 8666  4F8 16 03 78 00 F4 AC E4 EE 73 E5 40 92 09 66 14 E2 9E      RX
58827  212 48 75 A0 CD 4E BB 8F 26 93 FD 44 A1 A8 18 04 E0 AE F9 48 7A E8 D7 4A 98 28 33 4F 50 FD 91 B7 C1 99 F3 C6 F3 84 1A 30 6C 03 7C 66 91 46 04 A3 C3 54      RX
43678  18C  5 35 B7 1B 8C 8F      RX
28220  5E8 48 84 44 77 E8 72 B4 01 A9 70 01 D3 30 69 B1 56 20 63 08 42 74 8F 23 84 F2 12 D4 EA 70 91 41 94 CA 7F 3B 1B 9E 26 5F 7E CE 76 EF D1 B3 91 6E A4 21      RX
```

Figure 26: Part of the data sent for test 5. Source: own.

```
canfd.initialize()
while True:
    # receive message
    rxd = canfd.receiveMessageTasks()
    if rxd is not None: # print message when received
        print ("(TEST) received message: {}".format([hex(a) for a in
     rxd]))

>> (TEST) received message: ['oxfb', 'ox73', 'ox3c', 'ox69', 'ox17',
     'ox3', 'oxal', 'oxl', 'ox3f', 'oxd6', 'oxf', 'oxed', 'ox13', 'o
    x41', 'oxa7', 'oxff', 'oxb4', 'oxa8', 'ox9e', 'oxf4', 'oxda', 'o
    x63', 'oxb7', 'ox3c', 'oxb5', 'ox6e', 'ox84', 'ox89', 'ox85', 'o
    xd2', 'ox5c', 'ox19']
>> (TEST) received message: ['oxlb', 'oxc4', 'ox47', 'oxe6', 'ox9d',
     'oxc5', 'oxf']
>> (TEST) received message: ['ox48', 'oxfo', 'ox61', 'ox7b', 'oxb7',
    'oxe7', 'ox32', 'ox6a']
>> (TEST) received message: ['oxfl', 'ox77', 'ox37', 'ox4b', 'oxel',
    'oxl', 'ox75', 'ox42', 'oxbl', 'oxc', 'oxd4', 'ox5f', 'oxad', 'o
    x3', 'oxao','ox18', 'oxa6', 'ox2f', 'oxed', 'ox27', 'oxa2', 'o
    xf6', 'oxlb','ox2c']
>> (TEST) received message: ['ox2a', 'oxo', 'oxfa', 'ox1b']
>> (TEST) received message: ['oxe8', 'ox6a', 'oxl', 'ox24
', 'ox98', 'oxef', 'ox4', 'oxlb', 'ox5e', 'ox5e', 'ox35', 'ox29', 'o
    x8c', 'oxaa', 'ox56', 'ox2']
>> (TEST) received message: ['ox50']
>> (TEST) received message: ['oxc2', 'oxbc', 'ox3f', 'ox1']
>> (TEST) received message: ['oxf8', 'ox97', 'ox67', 'ox46', 'ox9e',
     'ox60', 'oxc8', 'ox75', 'ox3b', 'ox2a', 'ox7', 'ox8e', 'ox87',
    'oxf2', 'ox85', 'ox13', 'ox4d', 'oxe', 'oxc9', 'ox3d', 'ox7e', '
    oxlc', 'oxf9', 'oxc9', 'ox12', 'oxad', 'ox14', 'ox85', 'oxa2', '
    oxbl', 'oxf5', 'ox9e
', 'oxc7', 'oxa2', 'ox8', 'oxa6', 'ox30', 'ox71', 'oxc9', 'ox68', 'o
    x85', 'ox31', 'oxl', 'ox6c', 'oxff', 'ox84', 'oxco', 'ox48', 'o
    xc4', 'oxc8', 'ox19', 'oxad', 'ox3', 'ox2', 'oan', 'ox2d', 'oxd4
    ', 'ox87', 'oxa4', 'oxcd
', 'oxcd', 'ox6c', 'ox2d', 'ox4f']
>> (TEST) received message: ['oxb4', 'ox3c', 'ox7']
>> (TEST) received message: ['oxb3', 'oxfc', 'ox42', 'ox95', 'oxf4']
>> (TEST) received message: ['ox43', 'oxac', 'ox85', 'ox55', 'ox50',
     'oxfe', 'oxa', 'ox24', 'oxbe', 'ox55', 'oxbe', 'oxb7', 'ox2a',
    'oxec', 'ox37', 'ox4a', 'oxc9', 'oxla', 'oxc7', 'ox62', 'ox24',
    '0x5f', '0x9b', '0xbc', '0x93', '0x2f', '0x98', '0xf8', '0x7b',
    '0x80', '0xfl', '0x5']
>> (TEST) received message: ['0x88', '0x4b', '0x27', '0x88', '0xa']
>> (TEST) received message: ['0x1a', '0xc9', '0xd8', '0x5e', '0x3b',
     '0x58', '0x62', '0x3', '0xd8', '0xe8', '0x46', '0x10', '0x67',
    '0x65', '0x26', '0xcd', '0xb3', '0x50', '0xe9', '0x6b']
>> (TEST) received message: ['0x23', '0x4', '0x5d', '0xc6', '0x43',
    '0x56', '0xa4', '0x7c', '0x7f', '0x3f', '0x53', '0xec', '0x80',
    '0xf8', '0x10', '0xa8', '0xcd', '0x11', '0x5f', '0c0']
>> (TEST) received message: ['0xac']
>> (TEST) received message: ['0x89', '0x17', '0xlc', '0xaf', '0xld',
     '0xl', '0xb0', '0xal', '0xb2', '0x5a', '0x36', '0xd5', '0x95',
    '0xa9', '0xd7', '0xe8']
>> (TEST) received message: ['0x20', '0x48', '0xb0', '0xC5', '0xa1',
     '0xa2', '0x2d', '0x89']
>> (TEST) received message: ['0x3c', '0x86', '0x79', '0xfe', '0xl',
    '0x33', '0xf5', '0xdf', '0x48', '0xa7', '0x69', '0x95', '0x89',
    '0xe6', '0xdl', '0x99', '0x9c', '0xef', '0x93', '0xed', '0x3c',
    '0x81', '0x5c', '0xc7', '0x3d', '0xbl', '0xb8', '0xfa', '0x8d',
    '0x2d', '0x74', '0x24', '0xec', '0x4d', '0xb4', '0x9a', '0x6b',
    '0xdf', '0xd3', '0x14', '0xa6', '0xaf', '0xfd', '0x17', '0xc1',
    '0xd8', '0x8e', '0x26', '0x23', '0xce', '0x29', '0xa7', '0xf5',
    '0x27', '0x98', '0x97', '0x59', '0x2a', '0x67', '0xed', '0x6b',
    '0x23', '0x3f', '0xcf']
>> (TEST) received message: ['0x3', '0x78', '0x0', '0xf4', '0xac', '
    0xe4', '0xee', '0x73', '0xe5', '0x40', '0x92', '0x9', '0x66', '0
    x14', '0xe2', '0x9e']
>> (TEST) received message: ['0x75', '0xa0', '0xcd', '0x4e', '0xbb',
     '0x8f', '0x26', '0x93', '0xfd', '0x44', '0xal', '0xa8', '0x18',
     '0x4', '0xe0', '0xae', '0xf9', '0x48', '0x7a', '0xe8', '0xd7',
    '0x4a', '0x98', '0x28', '0x33', '0x4f', '0x50', '0xfd', '0x91',
    '0xb7', '0xc1', '0x99', '0xf3', '0xc6', '0xf3', '0x84', '0xla',
    '0x30', '0x6a', '0x3', '0x7c', '0x66', '0x91', '0x46', '0x4', '0
    xa3', '0xc3', '0x54']
>> (TEST) received message: ['0x35', '0xb7', '0xlb', '0x8c', '0x8f']
>> (TEST) received message: ['0x84', '0x44', '0x77', '0xe8', '0x72',
     '0xb4', '0xl', '0xa9', '0x70', '0xl', '0xd3', '0x30', '0x69', '
    0xbl', '0x56', '0x20', '0x63', '0x8', '0x42', '0x74', '0x8f', '0
    x23', '0x84', '0xf2', '0x12', '0xd4', '0xea', '0x70', '0x91', '0
    x41', '0x94', '0xca', '0x7f', '0x3b', '0x1b', '0x9e', '0x26', '0
    x5f', '0x7e', '0xce', '0x76', '0xef', '0xdl', '0xb3', '0x91', '0
    x6e', '0xa4', '0x21']
```

Listing 18: Test 5.

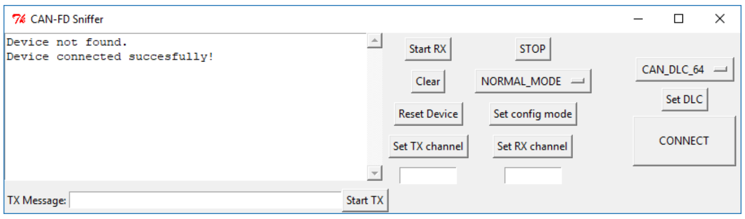
6.9 GUI test

In this final test, the GUI is tested to see if its functionalities work as expected. To do so, the different buttons and droplists are tested in order to check if they work properly, with the GUI not in debug mode. The following operations are made, in this order:

- The device is manually connected.
- DLC is set to a different value.
- Mode is set to External Loopback.

- A message is transmitted.
- The message is checked in the reception FIFO.
- The device is reset.
- Rx FIFO is checked again to see if the message is gone (and thus confirming the reset).

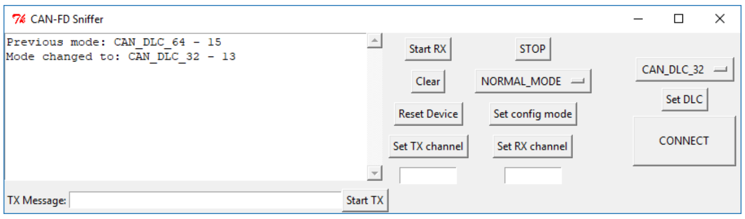
All the results are shown in Figures 27 to 30.



```
Device not found.
Device connected succesfully!
```

TX Message:

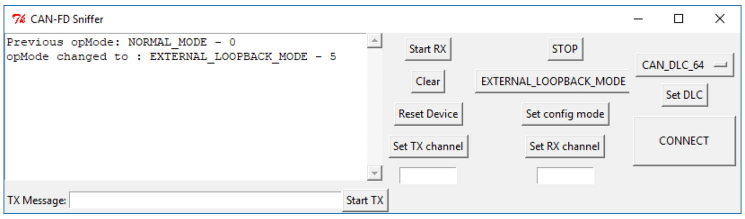
Figure 27: The device is manually connected. Source: own.



```
Previous mode: CAN_DLC_64 - 15
Mode changed to: CAN_DLC_32 - 13
```

TX Message:

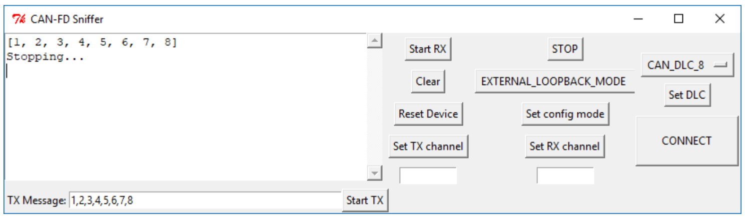
Figure 28: DLC is set to a different value. Source: own.



```
Previous opMode: NORMAL_MODE - 0
opMode changed to : EXTERNAL_LOOPBACK_MODE - 5
```

TX Message:

Figure 29: Mode is set to External Loopback. Source: own.



```
[1, 2, 3, 4, 5, 6, 7, 8]
Stopping...
```

TX Message: 1,2,3,4,5,6,7,8

Figure 30: Message is transmitted and received After reset, no message is received. Source: own.

7 Future work

Although the main objective of this project has been accomplished, there is still much more work to do in this library. All files are uploaded in GitHub [20], so anyone can access the library and improve it.

The main proposed improvements are the following:

- More functionality: This library only incorporates the basic functions of the Microchip API. Advanced functions, such as the use of the TEF, filters, masks, CRC use, and more, can be implemented.
- Further testing: Several tests have been done in this project in order to check the library and its functions, but other aspects, such as operation speed and additional bit time configurations, should be tested in more detail.
- Optimization: Some parts of the code could be improved in order to gain clarity and to improve performance. This includes the register problem discussed in 5.5.1. The access and modification of register variables should be unified and generalized in order to easily access and modify registers at bit, byte and word level, preserving the correct bit and byte order.
- The *ctypes* library should be totally implemented or discarded.
- Better GUI: The GUI developed in this project is made for demonstrations purposes of the library. It may have more advanced functions, such as configuration of additional parameters of the device, save log files, etc.
- Error handling: Most of the errors in the device operation go silently or stop the execution of the program. Exceptions and error handling should be added with *try-except* clauses, so that errors are noticed and treated correctly
- Updating: A major drawback of this library is the use of the Adafruit library, which forces the environment to be in Python 2.7, which is being deprecated (current version of Python is 3.7) and lacks some features, such as native support for *struct-like* data types. This library does not work in the newer versions of Python. Another library can be used to establish SPI communication, or the Adafruit library can be modified to work with newer versions of Python. Although the use of a virtual environment with Python 2.7 solves this problem when using a device with newer versions, a future goal could be the native support for Python 3.7
- Other OS support: Although the objective was to develop this library for use with Windows, it would be interesting to have support for Linux and Mac OS.

8 Temporal schedule

In Figure 31, a Gantt diagram of this project is seen. The *Research* task includes all the research done in other libraries, the devices used in this project, the Microchip library, as well as information on the CAN, CAN-FD and SPI protocols, and information about C and Python instructions. The *Setup* task includes the installation of all the necessary libraries and drivers. The rest of the tasks are self-explanatory.

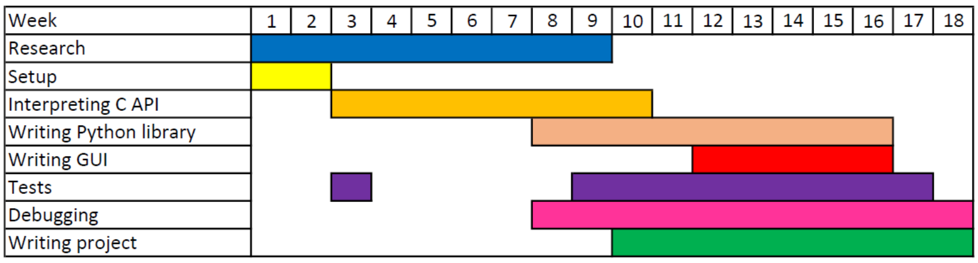


Figure 31: Gantt diagram. Source: own.

9 Budget and costs

The budget of this project is presented here, which can be seen in Table 5. The *activities* cost was calculated using the total time spent in the project (approximately 360h) with an hourly salary of 20 €. The hardware costs are very low, and all software and libraries used have no cost and license to be modified and distributed as desired, even for commercial use. Most of the cost comes from the human resources.

Product	**Price (€)**
MCP2517FD click	22.05
MPSEE cable C232HM-DDHSL-0	24.26
PC with Windows license	499.99
Activities (human resources)	7200
TOTAL	7746.3

Table 5: Costs. Source: own.

The lifetime of the software developed in this project is not expected to be very long, as it has some major drawbacks which need to be improved, as discussed in Section 7. The Python version needed is being deprecated and will not be updated any more in 2020; nevertheless, it is a good starting point for developing a more robust library with a longer life and support.

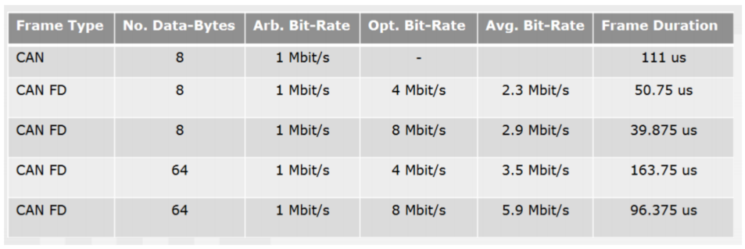
If the library was to be marketed and sold (which is not the case, as the library is designed to have a free license to use and modify), it would be interesting to know its value. The *python-can* library has around 299.985 downloads[9]. Supposing this library is as downloaded as the *python-can* one, each download of the library should cost at least 2.5 cents in order to amortize the investment, which is a very low price. In fact, it may be even distributed as free software and obtain income from other ways, such as maintenance or advertisement, thus obtaining better acceptance in the public.

Considering its actual state, its life would not be longer than 2 years. If its lifetime wants to be enlarged, more time would be required in order to update the library. This would take an estimated time of 140 h, which translates to an increase in 2800 € in the budget. This would increase the estimated value of the library to 3.5 cents/download.

9 This metric has been obtained from the Google Cloud Platform and only accounts the downloads from *pip*

10 Environmental impact

The use of CAN-FD protocol over classical CAN has the advantage of increasing the data transfered over unit of time, and thus requieres less time to transfer a certain amount of data. Consider the energy consumed by this device transfering 1 Mb (10^6 bits) of data at its maximum speed, in both CAN and CAN-FD modes. Figure 32 shows the duration of the frames in CAN protocol and some configurations of CAN-FD.



Frame Type	No. Data-Bytes	Arb. Bit-Rate	Opt. Bit-Rate	Avg. Bit-Rate	Frame Duration
CAN	8	1 Mbit/s	-		111 us
CAN FD	8	1 Mbit/s	4 Mbit/s	2.3 Mbit/s	50.75 us
CAN FD	8	1 Mbit/s	8 Mbit/s	2.9 Mbit/s	39.875 us
CAN FD	64	1 Mbit/s	4 Mbit/s	3.5 Mbit/s	163.75 us
CAN FD	64	1 Mbit/s	8 Mbit/s	5.9 Mbit/s	96.375 us

Figure 32: Frame duration of CAN and some configurations of CAN-FD. Source: [8]

From here, the required time to transfer 1 Mb of data at maximum speed is calculated, as shown in Equations 1 and 2 for the CAN frame, and in Equations 3 and 4 for the CAN-FD frame.

$$n = \frac{data}{2^6} = \frac{10^6}{64} = 15625 \text{ frames} \tag{1}$$

$$t = n \text{ x } 111 \ \mu s \approx 1.734s \tag{2}$$

$$n = \frac{data}{2^9} = \frac{10^6}{512} = 1954 \text{ frames} \tag{3}$$

$$t = n \text{ x } 96.375 \ \mu s \approx 0.188s \tag{4}$$

The operating current of the device at 5.5 V and 40 MHz is approximately 12 mA, so its power consumption is of 66 mW. The total energy consumption of the transmission is 114 mJ for the CAN frame and 12.4 mJ. The difference is astonishing, with the CAN-FD frame consuming only 10.8 % of the energy required for the classical CAN.

The CO_2 emissions for consuming electrical energy are, according to [21], about 0.27 tons of CO_2 per MWh consumed at the endpoint. Thus, the use of the CAN-FD protocol produces a saving of 7.62 µg of CO_2 per Mb of data transmitted. Taking an example, autonomous cars will treat as much as 4.000 GB of data per hour in the near future, as seen in [22]; the use of the CAN-FD protocol could save 243 g of CO_2 emissions per hour in a single car, which represents about a 1.95 % of saving in its hourly emissions[10].

10 Considering a mean speed of 70 km/h and emissions of 178 g/km, which is a the mean value of emissions of a gasoline car in Spain at November 2016 given by [23]

11 Conclusions

The objectives in this project have been accomplished. A basic library has been developed for Python to communicate a PC with Windows with the MCP2517FD click device through SPI communication, using a MPSEE cable with a USB port. This library is capable of reading data from a CAN-FD frame as well as writing to it. It can also access all internal registers of the device and configure it. It only uses four additional libraries (Adafruit for SPI, *ctypes* for some register-like objects, *binascii* for casting *bytearray* objects to integers, and *random* for generating some random data), and only one of these (Adafruit) is an external library, as all others are incorporated in Python. Finally, a basic GUI has also been developed to test the library, incorporating some tests. All of these files are located at [20], with GNU General Public License v3.0

This library is a starting point to develop more advanced libraries able to take advantage of all the functionalities of the controller. Although the lifetime of the library is somewhat limited, an update for working with newer versions of Python could extend it for several years. Nevertheless, the potential use of this library is very interesting, as it is now possible to connect the MCP2517FD with a computer, using Windows and Python 2.7, and several tasks, such as gathering data for analysis in real time, can be done easily in a single language. As all of the software has free license, any user can develop an application and contribute to this project, so both companies working with CAN-FD networks and home users developing its own projects.

12 Bibliography

[1] CAN bus. https://en.wikipedia.org/wiki/CAN_bus. Accessed: 2019-02-20.

[2] Introduction to CAN bus. https://learn.sparkfun.com/tutorials/ast-can485-hookup-guide/introduction-to-can-bus. Accessed: 2019-03-13.

[3] National Instruments. CAN FD EXPLAINED - A SIMPLE INTRO. https://www.csselectronics.com/screen/page/can-fd-flexible-data-rate-intro/language/en. Accessed: 2019-02-26.

[4] Serial Peripheral Interface. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface. Accessed: 2019-03-15.

[5] Total Phase. *Beagle Protocol Analyzers*, 1 2008. Rev. 3.02.

[6] Microchip Technology Inc. *External CAN FD Controller with SPI Interface*, 2017. Rev. 1.

[7] Future Technology Devices International Ltd. *USB 2.0 HI-SPEED TO MPSSE CABLE Datasheet*, 2016. Rev. 1.2.1.

[8] Universitatea Politehnica Timisoara. CAN with Flexible Data-Rate - CAN-FD. http://www.aut.upt.ro/~pal-stefan.murvay/teaching/nes/Lecture_05_CAN-FD.pdf. Accessed: 2019-02-26.

[9] G. Marcon Zago and E. Pignaton de Freitas. A Quantitative Performance Study on CAN and CAN FD Vehicular Networks. *IEEE Transactions on Industrial Electronics*, 65(5):4413–4422, May 2018.

[10] International Standards Organization. *Road vehicles - interchange of digital information - controller area network (CAN) for high-speed communication*, 11 1993. Rev. 1.

[11] Bosch. *CAN with Flexible Data-Rate*, 4 2012. Rev. 1.

[12] T. Nguyen, B. M. Cheon, and J. W. Jeon. CAN FD performance analysis for ECU re-programming using the CANoe. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, pages 1–4, June 2014.

[13] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, Apr 2007.

[14] S. Woo, H. J. Jo, I. S. Kim, and D. H. Lee. A Practical Security Architecture for In-Vehicle CAN-FD. *IEEE Transactions on Intelligent Transportation Systems*, 17(8):2248–2261, Aug 2016.

[15] Thorne B. and P. Ben. *python-can library*. https://python-can.readthedocs.io/en/stable/index.html. Accessed: 2019-04-1.

[16] Future Technology Devices International Ltd. Ftdi virtual com port drivers. https://www.ftdichip.com/Drivers/VCP.htm. Accessed: 2019-03-01.

[17] Batard P. Zadig. https://zadig.akeo.ie/. Accessed: 2019-03-01.

[18] DiCola T. Mpsee windows setup. https://learn.adafruit.com/adafruit-ft232h-breakout/windows-setup. Accessed: 2019-03-01.

[19] Microchip Technology Inc. Mcp2517fd c api. http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD_canfdspi_API_v1.0.zip. Accessed: 2019-02-20.

[20] J. Cortes. canfdlib. https://github.com/jcf9410/canfdlib.

[21] Departamento de Planificación y Estudios. FACTORES DE CONVERSIÓN ENERGÍA FINAL -ENERGÍA PRIMARIA y FACTORES DE EMISIÓN DE CO_2, 11 2011.

[22] Krzanich B. The coming flood of dta in autonomous vehicles. In *LA Auto Show's AutoMobility conference*, 2016.

[23] IDAE, ANFAC and ANIACAM. Guía de Vehículos Turismo de venta en España, con indicación de consumos y emisiones de CO_2, 3 2016.

[24] Microchip Technology Inc. Mc2517fd api. http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD_canfdspi_API_v1.0.zip. Accessed: 2019-03-01.

[25] Ràfols Bellés J. Development of a Python application for monitoring RF messages using one NRF24L01 board and a USB-MPSSE cable. Degree's final project, Universitat Politècnica de Catalunya, 2016.

[26] Fructuoso Keller D. LoRa sniffer using python and one MPSSE cable. Degree's final project, Universitat Politècnica de Catalunya, 2018.

[27] MicroControl GmbH & Co. KG. Can fd. an introduction. http://www.microcontrol.net/en/know-how/bus-systems/can-fd/. Accessed: 2019-02-25.

[28] National Instruments. Understanding can with flexible data-rate (can fd). http://www.microcontrol.net/en/know-how/bus-systems/can-fd/. Accessed: 2019-02-25.

[29] Motorola. *SPI Block Guide*, 1 2003. Rev. 3.06.

ETSEIB