Bachelor's Thesis

Bachelor's degree in Industrial Technology Engineering

CAN FD Node based on a PIC18 Microcontroller

REPORT

Author: Oriol Garrobé Guilera
Director: Manuel Moreno Eguílaz
Period: Spring semester

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Review

This document describes the process to implement a CAN node based on a PIC18 microcontroller and an MCP2517FD click. The system has been programmed using C language.

On the first hand, the architecture of the electronic system as well as the CAN FD protocol is explained. Briefly it is shown how the CAN FD frames are, in order to leave it clear for the reader, as the main object of the project is to transfer data through the bus CAN. Therefore, the structure of the data is of great relevance. Also, the architecture of the electronic system as well as its components, both hardware and software, is detailed.

On the other hand, it is explained step by step how to implement the system. The PIC18 microcontroller family from Microchip includes an 8 bit CPU, whereas the MCP2517FD CANFD controller is oriented to work with 4 byte word. From this regard, the main issue is to make these two devices compatible. It will be then, a good exercise to show how to link different devices with different bandwidths, with a method that is applicable to any device.

Finally, once the implementation is finished, several tests to prove that the system works properly are included. One of the main features of the CAN FD protocol is that it can transfer as well as receive data with different payloads and different baudrates, hence it will be demonstrated that the node is able to process any data frame in any rate set in the CAN FD protocol using a CAN FD sniffer.

Summary

1. Glossary

CAN: Controller Area Network.

CPU: Central Processing Unit.

CANFD: Controller Area Network with Flexible Data-rate.

IDE: Integrated Development Environment.

MCU: Microcontroller Unit.

2. Preface

2.1. Origin of the project

The CAN (Controller Area Network) protocol - designed by Robert Bosch GmbH in 1983 [1] - is a method of communication between various electronic devices. CAN provides a mechanism which is incorporated in the hardware and the software so different devices can communicate with each other using a common cable with two copper wires.

In 2011, Bosch invented the CAN FD (flexible data-rate), which improves the classical CAN [2]. It is possible to transmit data faster than with 1 Mbit/s and the payload (data field) is now up to 64 bytes long and not limited to 8 bytes anymore.

From this regard, it is of great relevance to understand how this new protocol works and be able to develop a low-cost node that eventually could be used in a real CAN FD network.

2.2. Motivation

This project was chosen because of the following reasons.

In the first place, I wanted to do a project where I could work with software. I want to improve my skills in this field, and this was a good opportunity to learn a new programming language.

From this regard, the subjects related to electrical and electronic engineering are the ones that I enjoyed the most. That is why I chose a project in electronics, so I could see how working in this field could be.

Finally, I would like to work in the automotive industry. The CAN bus is widely used in the automotive and aerospace industries and the CAN FD is a current extension to the original CAN. Therefore, this project gathered all my interests, and because of that I took it.

2.3. Previous requirements

In order to be able to develop the project it was necessary to have knowledge about the following topics.

On the one hand, knowing how a digital system works and its structure. The elements that compound the system and which their function is. Which the steps that any digital system follows to achieve their purpose are.

More specifically, it was necessary to understand the CAN FD protocol.

On the other hand, to be able to modify the functions of the system, it was necessary to program C language. That is why before starting the project I completed a course in *Tutorialspoint.com* [3] about C language.

3. Introduction

3.1. Objectives of the project

The main purpose of the project is to develop a CAN FD node using the MCP2517FD click and the PIC18F4520 microcontroller. It will be done by using the Microchip MPLAB IDE (Integrated Development Environment) and C language.

From this point, different objectives derived from the main one, such as getting to know the insights of the CAN protocol. Also, even though is not a main object of the project, it will be necessary to understand how the MPLAB IDE works, in order to maximize its features.

Once the project is done, it will be a very useful academic tool as it will receive and transmit both CAN and CAN FD messages. Combined with a CAN FD sniffer and an oscilloscope it will be instructive to understand the protocol and the CAN and CAN FD frames.

3.2. Scope of the project

This project is focused on developing a CAN FD node that can transmit and receive messages through a real CAN network. It is a project based on software, so no modifications on hardware will be effectuated.

A new driver will be developed so the CAN FD node works properly. The node will be configured so it can transmit and receive messages through FIFOs and apply Filters and Masks to each received message.

4. SYSTEM

4.1. CAN PROTOCOL

4.1.1. CONTROLLER AREA NETWORK

CAN is a communication protocol designed by the German company Robert Bosch GmbH [1]. It is based on a bus in order to allow devices - such as microcontrollers - to communicate with each other without a host computer. It was developed for the automotive market to reduce the weight and cost of wiring harnesses and add additional capabilities. Nowadays, it is also used in many industry applications.

CAN is a serial-based network. This means that the data is sent one bit at a time, sequentially, over a communication channel. Classic CAN allows simple messages of up to 8 bytes. There are no masters or slaves - all nodes see all messages.

It is not a complete network system. It consists of only the physical layer, the priority scheme and the error detection and handling circuitry.

4.1.1.1. Physical layer – Architecture

CAN usually consists of two wires - arranged as a differential pair for robust noise immunity, one is designated as CAN_HIGH and the other as CAN_LOW. The essence of the pair is the difference of voltage between the two lines. If the difference of voltage is 0, then the CAN delivers a “0”, being this the Recessive state. On the contrary, the Dominant state appears when there is a difference of voltage between lines and CAN delivers a “1” (see Fig.1).

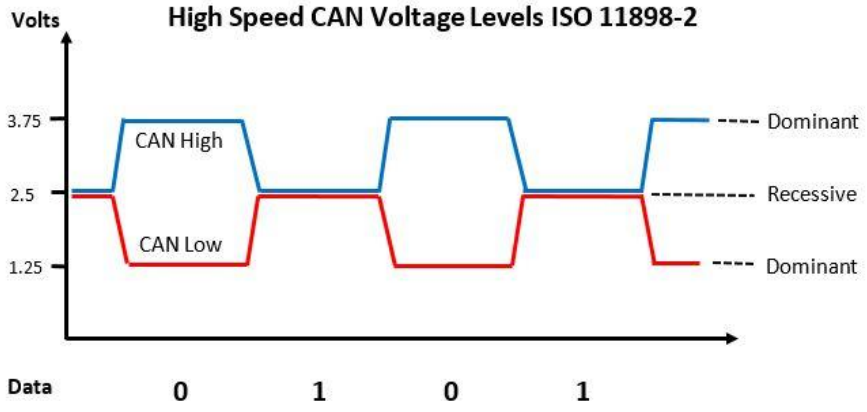


Fig. 1. CAN Voltage Levels. Source: [4].

A CAN data frame consists of a start bit, identifier, payload, CRC and some extra control bits. It can be a Standard message with a 11-bit identifier or Extended with a 29-bit identifier. According to its identifier, a CAN node can accept or ignore any message.

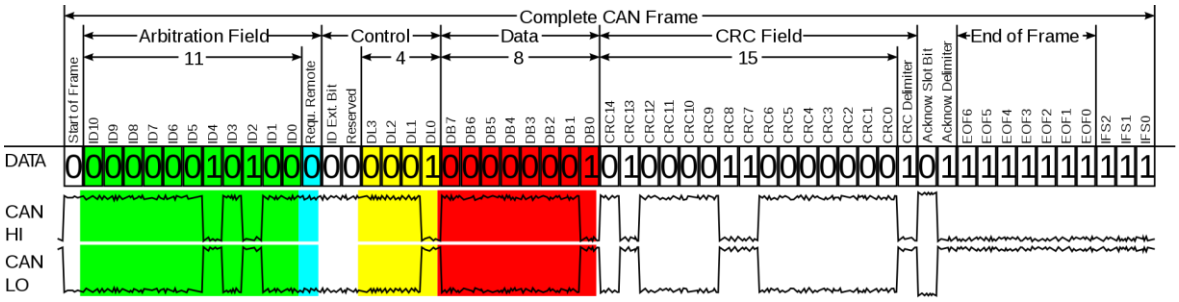


Fig. 2. CAN Frame. Source: [5].

A CAN node can put out either a “0” or a “1” in the bus at a certain speed or baudrate. It is not common to dynamically change speeds on a CAN bus - being the maximum speed 1 Mbits/s with 250 Kbit/s and 500 Kbit/s, common baudrates in commercial trucks and cars, respectively. In order to have all the CAN nodes clock synchronized, each CAN message must transmit at least one bit every 5-bit time.

The message with the highest priority always gets first. The priority is determined by the identifier - a lower value has a higher priority.

4.1.1.2. Error detection

Error detection in the CAN protocol is of great importance for the performance of a CAN system. The error handling aims at detecting errors in messages appearing on the CAN bus, so that the transmitter can retransmit an erroneous message. A CAN network can easily detect any errors. If a CAN node generates too many errors, it will take itself off the network (bus off condition).

Each node maintains two error counters: the Transmit Error Counter and the Receive Error Counter. If a CAN node detects too many errors, it will take itself off the network [1].

4.1.2. Control Area Network with Flexible Data-Rate

CAN FD [2] - just like classic CAN - is a protocol designed to transmit and record data. It is an extension of the CAN in response to the bandwidth requirements of the automotive industry - automakers needed more accurate “real-time” data. The CAN FD protocol support CAN FD and classical CAN frames.

This developed protocol can transmit data faster than 1 Mbits/s and the payload is longer - it can now transmit up to 64 bytes rather than the 8 bytes that classic CAN could. This is because when just one node is transmitting, the bitrate can be increased. After the transmission, the nodes need to be resynchronized.

The classical CAN and the CAN FD frames are both identical from the SOF throughout the 11 or 29 arbitration bits. After that, in the control and in the CRC field the CAN FD format use more bits. The main frame differences are the following:

- The FDF bit - in the control field - indicates whether the following bit sequence is interpreted as a CAN FD or a classical CAN frame.
- In the classical CAN control field, there is an RTR-bit while in the CAN FD there is an RRS-bit.
- In the CAN FD frame, there is a BRS-bit, that indicates the bit-rate switch.

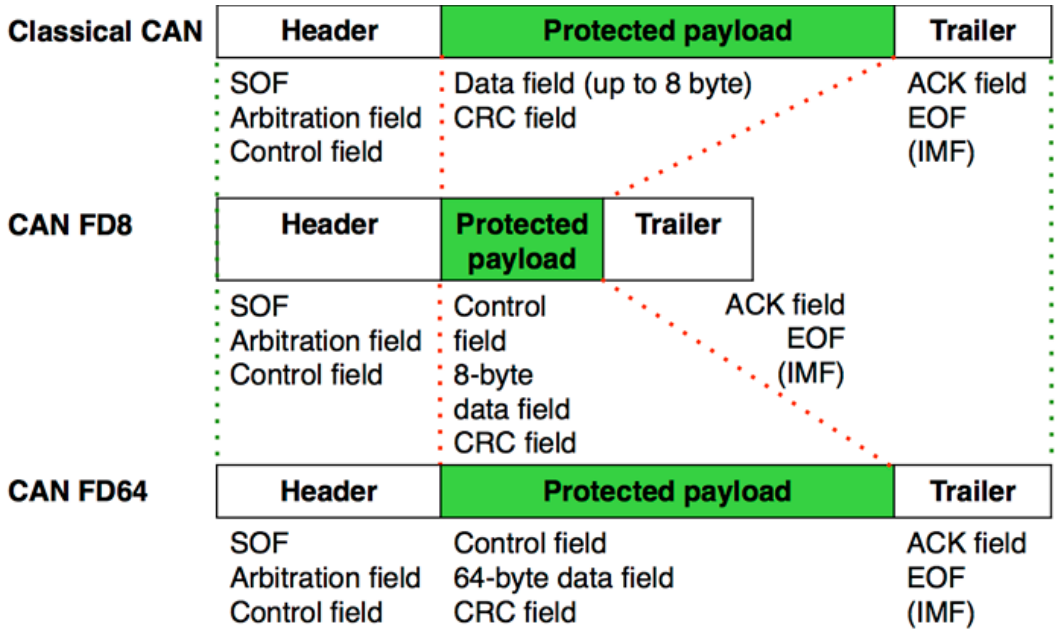


Fig. 3. CAN frame and CAN FD frame. Source: [6].

4.1.3. CAN Vs. CAN FD

Comparing CAN FD with classical CAN, the following reasons illustrate why to switch from classical CAN to CAN FD [7]:

- CAN FD provides shorter CAN frames, whilst increasing the bit rate - lower latency, better real-time performance and higher bandwidth.
- Option to switch to faster bit rate in the data phase. The arbitration bit rate is the same as in CAN.
- CAN FD can hold more data in the CAN frame, from 8 up to 64 bits. When sending large data objects, the software is simplified but it is necessary to increase the bitrate in order to reduce the CAN frame and the time that the frame occupies the communication line.
- CAN FD has a higher performance CRC algorithm - lowers the risk of undetected errors.
- All CAN FD controllers can handle both CAN FD and classical CAN frames. It is possible, then, to introduce a CAN FD controller in an existing system.
- Because of the higher bitrate the data throughput is increased, reducing the download time.

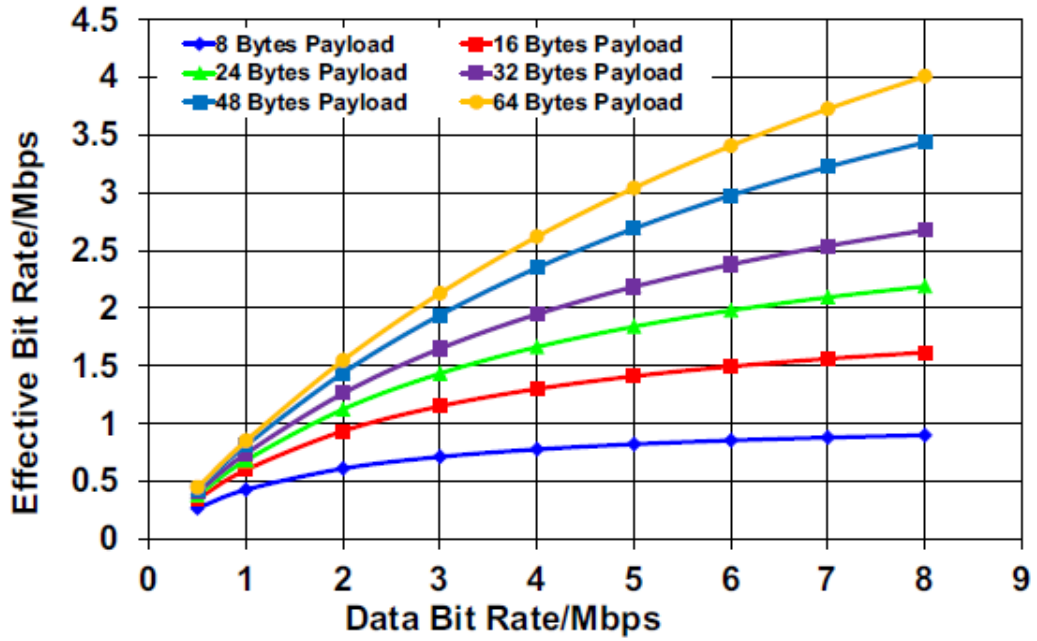


Fig. 4. CAN FD Performance. Source: [8].

4.2. MCP2517FD CLICK

MCP2517FD Click is a commercial board composed by the MCP2517FD CAN FD controller, the ATA6563 high speed transceiver and a DB9 9-pin connector [10].

Fig. 5. MCP2517FD Click. Source: [9].

4.2.1. External CAN FD Controller with SPI Interface.

The MCP2517FD is a cost-effective and small-footprint CAN FD controller that can be easily added to a microcontroller with an available SPI interface. Therefore, a CAN FD channel can be easily added to a microcontroller that is either lacking a CAN FD peripheral, or that does not have enough CAN FD channels.

The MCP2517FD supports both, CAN frames in the classical format and CAN FD format as specified in ISO 11898-1:2015 [10].

The MCP2517FD is composed by blocks, and contains the following ones:

- CAN FD Controller. It implements the CAN FD protocol and contains the FIFOs and the filters. It has different modes, among them: Normal CAN FD and normal CAN 2.0.
- SPI interface. It controls the device by accessing SFRs and RAM.
- RAM controller. It arbitrates the RAM accesses between the SPI and CAN FD controller module.
- Message RAM. It is used to store the data from the message objects.

- Oscillator. It generates the Clock.
- Internal LDO and POR circuit.
- The I/O control.

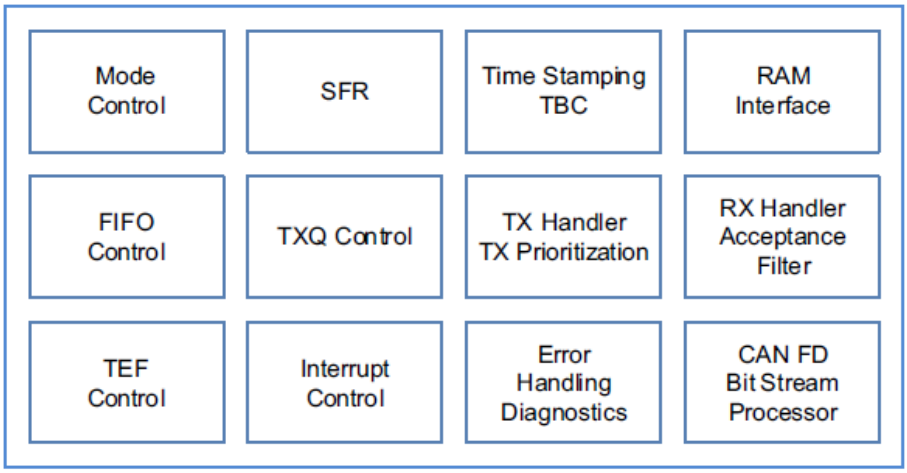


Fig. 6. Module block diagram. Source: [8].

4.2.2. ATA6563 CAN Transceiver

The ATA6563 [11] is a high-speed CAN transceiver that provides an interface between a controller area network (CAN) protocol controller and the physical two-wire CAN bus. It is designed for high-speed applications in the automotive industry, providing differential transmit and receive capability to a CAN protocol controller.

It offers improved electromagnetic compatibility (EMC) and electrostatic discharge (ESD) performance and has an ideal behaviour to the CAN when the supply voltage is off.

4.2.3. Used pins

For this project only the following pins of the MCP2517FD Click are used.

Pin name	Description
GND	Ground.
5V	Positive supply.

3V3	Positive supply.
SDI	SPI data input.
SDO	SPI data output.
SCK	Spi clock input.
nCS	SPI chip select input.

Table 1. Used pins of the MCP2517FD Click. Source: [12].

4.3. PIC18F4520

The PIC18F4520 microcontroller - in this case in a 40 pin PDIP package - is an 8-bit enhanced flash PIC microcontroller that comes with nanoWatt technology and is based on RISC architecture [13]. It comes with unbuilt peripheral with the ability to perform multiple functions.

The PIC18F4520 contains 256 bytes of EEPROM data memory, 1536 bytes of RAM, and program memory of 32K. It also incorporates 2 Comparators,10-bit Analog-to-Digital (A/D) converter with 13 channels and houses decent memory endurance around 1,000,000 for EEPROM and 100,000 for program memory.

Every pin on the module comes with a unique function, used as per the requirement of the project, and some pins incorporate multiple functions [13]. In this project the following pins are used:

PIN	**Features**
Pin 14: RC3/SCK/SCL.	RC3. Digital I/O. SCK. Synchronous serial clock Input/Output for SPI Mode. SCL. Synchronous serial clock Input/Output for IC Mode.
Pin 15: RC4/SDI/SDA.	RC4. Digital I/O. SDI. SPI data in. SDA. IC data I/O.

Pin 16: RC5/SDO.	RC5. Digital I/O. SDO. SPI data out.
Pin 21: RB0/INT0/FLT0/AN12.	RB0. Digital I/O. INT0. External interrupt 0. FLT0. PWM Fault input for CCP1. AN12. Analog input 12.

Table 2. PIN features. Source: [13].

It is important for the project to emphasize that it is a microcontroller with an 8 bit CPU.

The PIC18F4520 can perform many functions, among them, the In-circuit serial programming (ICSP), also called In-system programming (ISP), makes the device enable to be programmed in the required system after installation, setting it free from programming the device before making it compatible with the certain project.

Microchip provides for free its own standard compiler for the PIC controller family called MPLAB C compiler for PIC18 MCUs- known as C18 Compiler – which is explained in section 4.4.

The PIC18F4520 is a good choice for a university project due the following features:

- It has a user-friendly interface that requires no prior skills.
- It can perform many functions with the minimum circuitry.
- It is cheap.
- Minimum power consumption.

4.4. MPLAB C COMPILER FOR PIC18 MCUs

The MPLAB C18 is a C compiler used for the PIC18 family of PICMicro 8-bit MCUs [14].

It is a component of Microchip's MPLAB Integrated Development Environment (IDE) - providing a full graphical front end - for easy-to-use source level debugging with MPLAB's

software and hardware debug engines. Text errors in source code and breakpoints instantly switch to corresponding lines in the proper file, and watch windows show data structures with defined data types, including floating points, arrays and structures.

Among its features is important to highlight the following:

- It is compatible with ANSI ‘89.
- Compatibility with object modules generated by the MPASM assembler, allowing complete freedom in mixing assembly and C programming in a single project.
- Transparent read/write access to external memory.
- Strong support for inline assembly when total control is necessary.
- efficient code generator engine with multi-level optimization.
- Extensive library support, including PWM, SPI, I2C, UART, USART, string manipulation and math libraries.
- Full user-level control over data and code memory allocation.
- Supports both a small (16-bit pointers) and a large (24-bit pointers) memory model for efficient use of memory.
- MPLIB allows easy use of included libraries and for user created libraries.
- Extensive multi-pass optimizations.
- Supports new PIC18F extended mode instructions.

4.5. ARCHITECTURE OF THE SYSTEM

Using all hardware and software elements explained before, the architecture of the system is showed up next.

In the first place, the computer where the *Driver* is developed is connected to the PIC18 through the MPLAB ICD 3 [15] and the Wave Share PIC18 development board [16]. (see Fig.7).

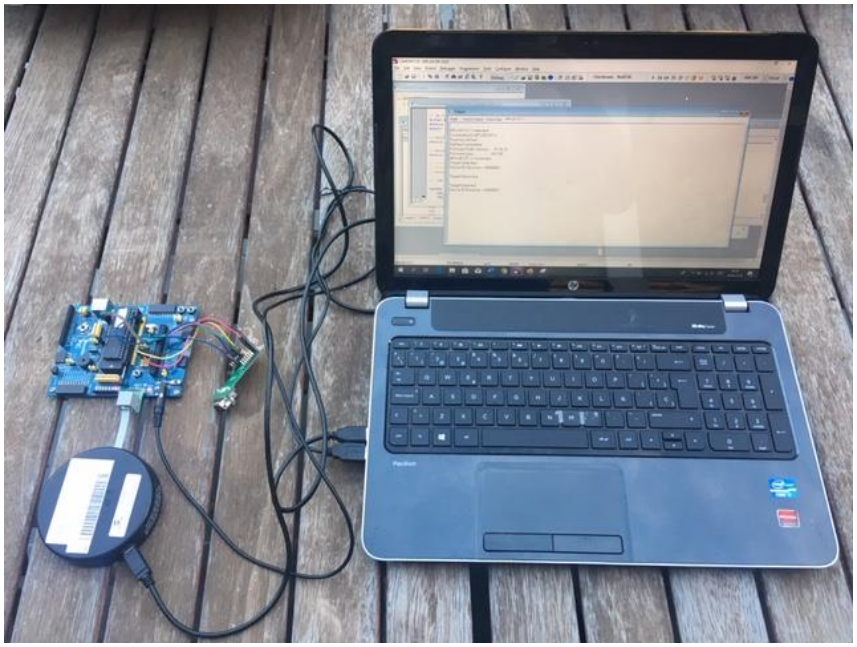


Figure 7. Computer connected to the node through the MPLAB ICD 3. Source: Own.

The next step is to connect the PIC18 to the MCP2517FD click using the PINs mentioned in section 4.2 and section 4.3. (see Fig.8).

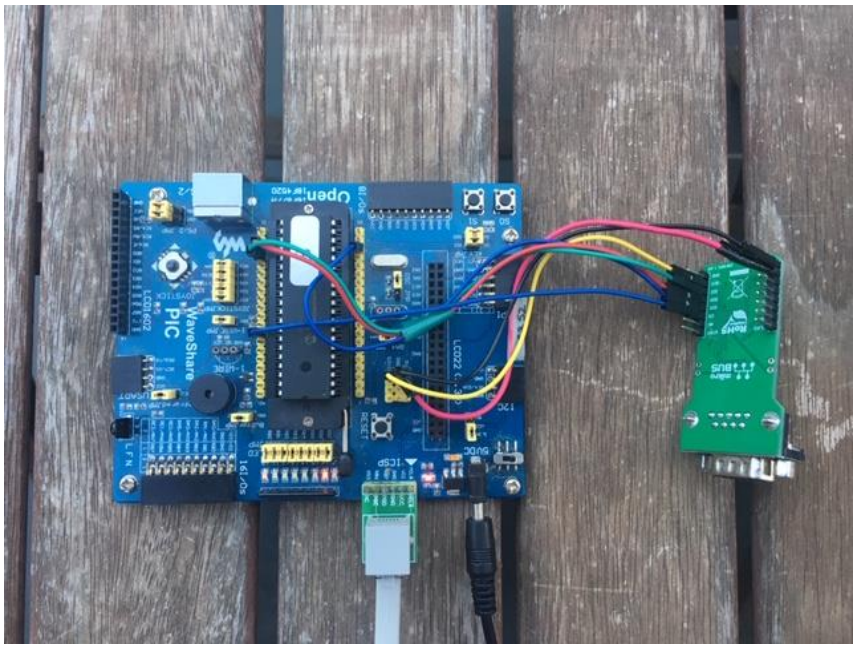


Figure 8. PIC18 connected to the MCP2517FD. Source: Own.

Finally, the MCP2517FD can be connected to a real CAN FD network. However, for the moment it will not - as the project can be developed and tested without the needs of a real network.

5. PROJECT

5.1. DRIVER DEVELOPMENT

5.1.1. Needed files

The development of the driver for this project is based on an existing solution from Microchip [16]. The source code that is used in this project is a variation of the MCP2517FD canfdspi API for a PIC32MX470 [17]. The PIC32MX470 is a 32-bit development platform, so the first step was to make possible that the PIC18 could process this code. In first place, it is necessary to import all the needed files, which are the following:

Header files	Source files
drv_canfdspi_api.h	drv_canfdspi_api.c
drv_canfdspi_defines.h	drv_spi.c
drv_canfdspi_register.h	main.c
drv_canfdspi_spi.h	
main.h	

Table 3. Used files. Source: [17].

Also, as the code is inherited, there are files that are not necessary for the project. These files, however, are imported in the original code. The next step is then to erase those files and comment the lines where they were imported in order to remove possible compiler errors.

5.1.2. Data types

Once the MPLAB IDE and the compiler C18 are installed, using the debugging tool MPLAB ICD3 it is proceeded to adapt the code.

The first step to adapt the code – using the debugging tool MPLAB ICD3 - is to define the data types - as the PIC18 defines them differently than the PIC32 - with the following order.

Original code	Modified code
None	#define uint32_t unsigned long #define uint16_t unsigned int #define uint8_t unsigned char #define int8_t char #define bool unsigned char

Table 4. Data types definition. Source: own.

These data types are defined in the “drv_canfdspi_defines.h” file, and then imported in the other files of the project with the following order:

#include “drv_canfdspi_defines.h”

5.1.3. Declaring local variables

Using this compiler, it is mandatory that all the local variables are defined at the very beginning of the function – to avoid a syntax error -. The next step is, then, to move relocate local variables definitions. One example of this procedure is shown next:

Original code	Modified code

int8_t
DRV_CANFDSPI_RamInit(CANFDSPI_MODULE_ID
index, uint8_t d)
{
uint8_t
txd[SPI_DEFAULT_BUFFER_LENGTH];
uint32_t k;
int8_t spiTransferError = 0;
// Prepare data
for (k = 0; k <
SPI_DEFAULT_BUFFER_LENGTH; k++) {
txd[k] = d;
}
uint16_t a = cRAMADDR_START;
for (k = 0; k < (cRAM_SIZE /
SPI_DEFAULT_BUFFER_LENGTH); k++) {
	int8_t
	DRV_CANFDSPI_RamInit(CANFDSPI_MODULE_ID
	index, uint8_t d)
	{
	uint8_t
	txd[SPI_DEFAULT_BUFFER_LENGTH];
	uint32_t k;
	int8_t spiTransferError = 0;
	uint16_t a = cRAMADDR_START;
	// Prepare data
	for (k = 0; k <
	SPI_DEFAULT_BUFFER_LENGTH; k++) {
	txd[k] = d;
	}
	for (k = 0; k < (cRAM_SIZE /
	SPI_DEFAULT_BUFFER_LENGTH); k++) {

```
    spiTransferError =
DRV_CANFDSPI_WriteByteArray(index, a, txd,
SPI_DEFAULT_BUFFER_LENGTH);
        if (spiTransferError) {
            return -1;
        }
        a += SPI_DEFAULT_BUFFER_LENGTH;
    }
    return spiTransferError;
}
```

```
    spiTransferError =
DRV_CANFDSPI_WriteByteArray(index, a, txd,
SPI_DEFAULT_BUFFER_LENGTH);
        if (spiTransferError) {
            return -1;
        }
        a += SPI_DEFAULT_BUFFER_LENGTH;
    }
    return spiTransferError;
}
```

Table 5. Local variables declaration. Source: own.

It can be seen that “uint16_t a = cRAMADDR_START;” is replaced at the top of the function.

5.1.4. Splitting the message

One of the biggest issues of the project is that the original code is written for a 32-bit microcontroller, while an 8-bit MCU is used in this project. Therefore, it is necessary first to split the structures in members no bigger than 1 byte, provided that the maximum length of the field is less than or equal to the integer word length of the microcontroller.

In the first place, in the “drv_canfdspi_defines.h” there are structure data types that contained bit fields with widths greater than 8 bits. Such bit fields must be split as explained before, so they can fit in the MCU memory. Following, an example of this procedure is shown:

Original code	**Modified code**
typedef struct _CAN_FILTEROBJ_ID { uint32_t SID : 11; uint32_t EID : 18; uint32_t SID1 : 1; uint32_t EXIDE : 1; uint32_t unimplemented1 : 1; } CAN_FILTEROBJ_ID;	typedef struct _CAN_FILTEROBJ_ID { uint32_t SIDA : 8; uint32_t SIDB : 3; uint32_t EIDA : 8; uint32_t EIDB : 8; uint32_t EIDC : 2; uint32_t SID1 : 1; uint32_t EXIDE : 1; uint32_t unimplemented1 : 1; } CAN_FILTEROBJ_ID;

Table 6. Structure type modification example. Source: own.

This problem is present also in the “drv_canfdspi_register.h” file. In this case it is in union data

types that define the architecture of the registers. The procedure used to split the register is the following:

Original code	Modified code

typedef union _REG_CiFIFOUA {
struct {
uint32_t UserAddress : 12;
uint32_t unimplemented1 : 20;
} bF;
uint32_t word;
uint8_t byte[4];
} REG_CiFIFOUA;
	typedef union _REG_CiFIFOUA {
	struct {
	uint32_t UserAddress1 : 8;
	uint32_t UserAddress2 : 4;
	uint32_t unimplemented11 : 8;
	uint32_t unimplemented12 : 8;
	uint32_t unimplemented13 : 4;
	} bF;
	uint32_t word;
	uint8_t byte[4];
	} REG_CiFIFOUA;

Table 7. Union type modification example. Source: Own.

In both cases now the message fits the Microchip PIC18.

Applying these changes to both the header code and the source code, the compiler does show no more errors. This does not mean that the function of the code is the proper one, this only means that the microcontroller understands the code.

5.1.5. New SPI

The SPI driver used for the PIC32 is indeed too advanced. Therefore, a simpler one is enough for the PIC18 – based on the C18 libraries [18].

Since there is only one SPI peripheral in the MCU there is no need to index which one the SPI is transferring to. From these regards, we must assign the values 0 or 1 – 1 meaning that we are accessing at the SPI - with the SPI function “SPI_CS” to access the RAM.

To initialize the SPI it is only necessary to use the function “OpenSPI” from the C18 libraries.

To transfer data, “WriteSPI” and “ReadSPI” functions are used and the code that assigns the slave index – as there is only one slave - is erased.

5.1.6. Memory usage

When trying to export the code to the PIC18, there was not enough ROM/RAM memory to fit it all. It was then necessary to take two actions to get enough space to implement the “main.c” afterwards.

The first step was to save some files at the ROM (Read Only Memory) in order to free space from the RAM (Random Access Memory).

The second step to solve the memory problem was to erase all the functions from the API that were not working. This means all the functions in the original code that were not necessary for the project were erased or commented.

The API functions needed for this project are enumerated and described in Annex 1.

The result was that enough space to develop the “main.c” was found.

5.1.7. SPI instructions

The SPI instructions are the ones that access the SFRs and the RAM. The SFRs are the Special Function Registers and are used to control and read the status of the CAN FD Controller module. The RAM – Random Access Memory – is used to store the data of the message objects.

To read and write through the SPI is necessary to use the transfer data functions set in the SPI. For this, it is necessary to modify the following functions. The “ReadSPI” and the “WriteSPI” are the functions from the SPI that are going to be called when using the SPI instructions.

To access the SFRs it is necessary that the device is in Configuration mode. On the other hand, the RAM is word oriented (4 bytes at a time), so any multiple of 4 data bytes can be read or written in one instruction.

Each SPI instruction starts by driving the nCS (Chip Select) low – from the value 1 to 0 -. The 4-bit command and the 12-bit address are shifted into the SDI. During a write instruction data-bits are shifted into the SDI on the rising edge of SCK. On the contrary, data bits are shifted out of the SDO on the falling edge.

Briefly, it is necessary to force the nCS low (0) at the beginning of every instruction and rising it back at the end of it. All the SPI instructions have been modified in order to fulfil this

requirement.

The SPI instructions used in this project are the following:

- DRV_CANFDSPI_Reset
- DRV_CANFDSPI_ReadByte
- DRV_CANFDSPI_WriteByte
- DRV_CANFDSPI_ReadWord
- DRV_CANFDSPI_WriteWord
- DRV_CANFDSPI_ReadHalfWord
- DRV_CANFDSPI_WriteHalfWord
- DRV_CANFDSPI_ReadByteArray
- DRV_CANFDSPI_WriteByteArray
- DRV_CANFDSPI_ReadWordArray
- DRV_CANFDSPI_WriteWordArray

Following there is one example of the modified code:

Original code

```
int8_t DRV_CANFDSPI_WriteByte(CANFDSPI_MODULE_ID index, uint16_t address, uint8_t txd)
{
    uint16_t spiTransferSize = 3;
    int8_t spiTransferError = 0;
    // Compose command
    spiTransmitBuffer[0] = (uint8_t) ((cINSTRUCTION_WRITE << 4) + ((address >> 8) & 0xF));
    spiTransmitBuffer[1] = (uint8_t) (address & 0xFF);
```

```
    spiTransmitBuffer[2] = txd;
    spiTransferError = DRV_SPI_TransferData(index, spiTransmitBuffer, spiReceiveBuffer, spiTransferSize);
    return spiTransferError;
}
```

Modified code

```
int8_t DRV_CANFDSPI_WriteByte(CANFDSPI_MODULE_ID index, uint16_t address, uint8_t txd)
{
    uint16_t spiTransferSize = 3;
    int8_t spiTransferError = 0;
    SPI_CS = 0;
    // Compose command
    spiTransmitBuffer[0] = (uint8_t) ((cINSTRUCTION_WRITE << 4) + ((address >> 8) & 0xF));
    spiTransmitBuffer[1] = (uint8_t) (address & 0xFF);
    spiTransmitBuffer[2] = txd;
    WriteSPI(spiTransmitBuffer[0]);
    WriteSPI(spiTransmitBuffer[1]);
   WriteSPI(spiTransmitBuffer[2]);
   SPI_CS = 1;
   return spiTransferError;
}
```

Table 8. SPI instruction modification example. Source: own.

5.1.8. Linking structs and unions

To fit the Microchip PIC18 with a bandwidth of 8 bits, it is necessary to split structures and unions. In order to get the original message and send it to the MCU properly, it is necessary to order it and put it together again.

5.1.8.1. Union data type

In the “drv_canfdspi_register.h” file, the modified data types are unions. The first step to take is then to make sure that the compiler packs the bit fields properly. From this point, it is necessary that the bit fields are multiple of 8 bits to avoid overlapping of the data. Following there is one illustrating example.

Original code	Modified code

typedef union _REG_CiFIFOUA {
struct {
uint32_t UserAddress : 12;
uint32_t unimplemented1 : 20;
} bF;
uint32_t word;
uint8_t byte[4];
} REG_CiFIFOUA;
	typedef union _REG_CiFIFOUA {
	struct {
	uint32_t UserAddress1 : 8;
	uint32_t UserAddress2 : 4;
	uint32_t unimplemented11 : 4;
	uint32_t unimplemented12 : 8;
	uint32_t unimplemented13 : 8;
	} bF;
	uint32_t word;
	uint8_t byte[4];
	} REG_CiFIFOUA;

Table 9. Union type modification. Source: own.

Since the bit fields modified are unimplemented, when packing them, there should not be any problem and the data will be properly packed as compactly as possible.

On the other hand, when the API calls the integer “UserAddress”, an error rises as after the modification there is no such bit field. In order to get the proper value then, it is necessary to take both “UserAddress1” and “UserAddress2” and logically add them. An example is shown following:

Original code
a = ciFifoUa.bF.UserAddress;
Modified code
a = ((uint16_t)ciFifoUa.bF.UserAddress2)<<8 | ciFifoUa.bF.UserAddress1;

Table 10. UserAddress call. Source: own.

5.1.8.2. Structure data type

In this file the modified data types are structures. With the same procedure followed in the registers, the bit fields must be rearranged. Following there is one example:

Original code	Modified code
typedef struct _CAN_FILTEROBJ_ID { uint32_t SID : 11; uint32_t EID : 18; uint32_t SID1 : 1; uint32_t EXIDE : 1; uint32_t unimplemented1 : 1; } CAN_FILTEROBJ_ID;	typedef struct _CAN_FILTEROBJ_ID { uint32_t SIDA : 8; uint32_t EIDA : 8; uint32_t EIDB : 8; uint32_t SIDB : 3; uint32_t EIDC : 2; uint32_t SID1 : 1; uint32_t EXIDE : 1; uint32_t unimplemented1 : 1; } CAN_FILTEROBJ_ID;

Table 11. Structure type modification. Source: own.

In this case, since the modified bitfields are implemented it is mandatory to order them when called in the API. This issue is only noted in the next three API functions:

- DRV_CANFDSPI_TransmitChannelLoad
- DRV_CANFDSPI_FilterObjectConfigure
- DRV_CANFDSPI_FilterMaskConfigure

Following there is one example:

Original code

int8_t DRV_CANFDSPI_FilterObjectConfigure(CANFDSPI_MODULE_ID index,
CAN_FILTER filter, CAN_FILTEROBJ_ID* id)
{
uint16_t a;
REG_CiFLTOBJ fObj;
int8_t spiTransferError = 0;
// Setup
fObj.word = 0;
fObj.bF = *id;
a = cREGADDR_CiFLTOBJ + (filter * CiFILTER_OFFSET);

```
    spiTransferError = DRV_CANFDSPI_WriteWord(index, a, fObj.word);
    return spiTransferError;
}
```

Modified code

```
int8_t DRV_CANFDSPI_FilterObjectConfigure(CANFDSPI_MODULE_ID index,
        CAN_FILTER filter, CAN_FILTEROBJ_ID* id)
{
    //uint16_t a = 0;
    REG_CiFLTOBJ fObj;
    int8_t spiTransferError = 0;
            uint32_t new_word;
            new_word = id->SIDA;
            new_word = new_word | (((uint32_t)id->SIDB)<<8);
            new_word = new_word | (((uint32_t)id->EIDA)<<11);
            new_word = new_word | (((uint32_t)id->EIDB)<<19);
            new_word = new_word | (((uint32_t)id->EIDC)<<27);
            new_word = new_word | (((uint32_t)id->SID1)<<29);
            new_word = new_word | (((uint32_t)id->EXIDE)<<30);
    // Setup
            fObj.word = new_word;
    a = cREGADDR_CiFLTOBJ + (filter * CiFILTER_OFFSET);
    spiTransferError = DRV_CANFDSPI_WriteWord(index, a, fObj.word);
    return spiTransferError;
}
```

Table 12. Structure bit fields put together in order. Source: own.

6. RESULTS

Once the system is ready it is mandatory to do several tests in order to proof that it works properly.

6.1. SPI INSTRUCTIONS TESTS

The first step is then to test the SPI instructions, as the whole project is based on these few functions. If they do not work properly, it will be impossible to get the expected results. These functions, as mentioned, are the ones that access the RAM and the SFRs, so there is no need to configurate the CAN in order to proof their reliability.

Following there are several tests of these functions.

6.1.1. Test WriteByte & ReadByte

Using the code shown in Figure 7, the following SPI instructions are tested:

- DRV_CANFDSPI_Reset
- DRV_CANFDSPI_WriteByte
- DRV_CANFDSPI_ReadByte

```
//Test 1

//Test WriteByte & ReadByte

address = 0x010;

DRV_CANFDSPI_Reset(0);
DRV_CANFDSPI_WriteByte(0, address, 0x5A);
DRV_CANFDSPI_ReadByte(0, address, &value);
Nop();
```

Figure 7. WriteByte & ReadByte test code. Source: Own.

Results:

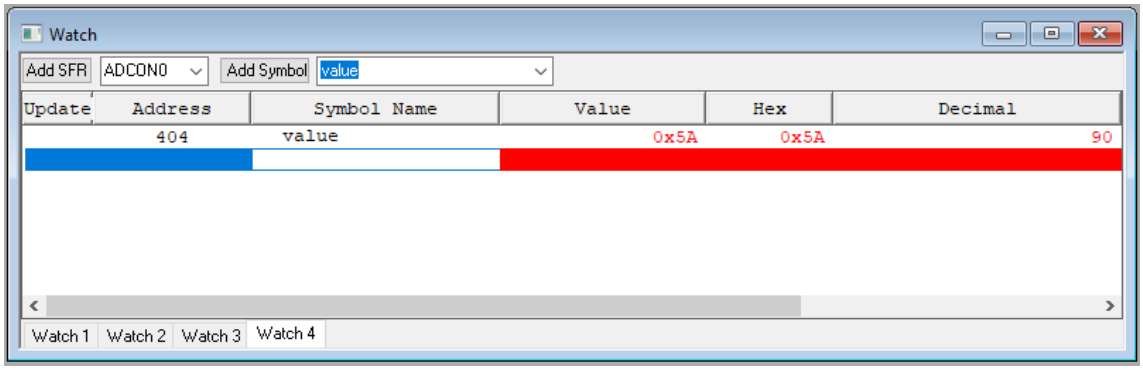


Figure 8. Results of the WriteByte & ReadByte test. Source: Own.

As expected, Figure 8 shows that the value of the variable "value" – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.1.2. Test WriteByte & ReadWord

Using the code shown in Figure 9, the following SPI instructions are tested:

- DRV_CANFDSPI_Reset
- DRV_CANFDSPI_WriteByte
- DRV_CANFDSPI_ReadWord

```
//Test 2

//Test WriteByte & ReadWord

address = 0x010;
DRV_CANFDSPI_Reset(0);
DRV_CANFDSPI_WriteByte(0, 0x010, 0x55);
DRV_CANFDSPI_WriteByte(0, 0x011, 0x55);
DRV_CANFDSPI_WriteByte(0, 0x012, 0x55);
DRV_CANFDSPI_WriteByte(0, 0x013, 0x55);
DRV_CANFDSPI_ReadWord(0, address, &value);
Nop();
```

Figure 9. WriteByte & ReadWord test code. Source: Own.

Results:

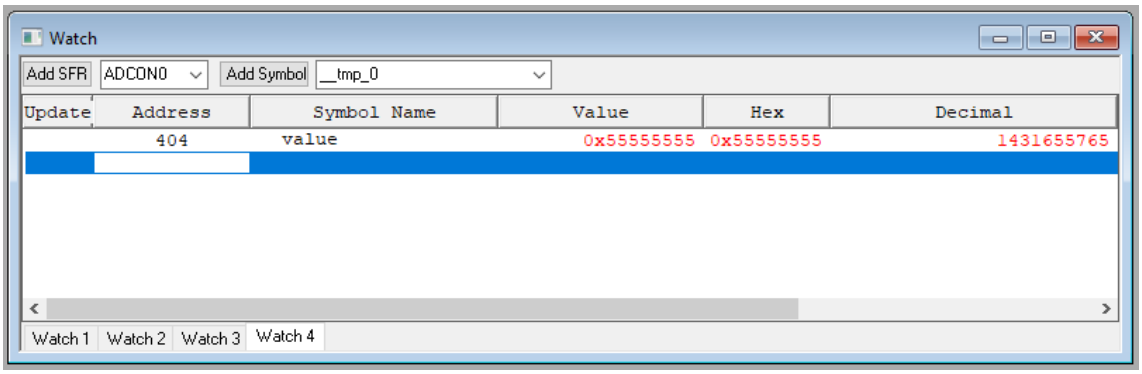


Figure 10. Results of the WriteByte & ReadWord test. Source: Own.

As expected, Figure 10 shows that the value of the variable “value” – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.1.3. Test WriteWord & ReadWord

Using the code shown in Figure 11, the following SPI instructions are tested:

- DRV_CANFDSPI_WriteWord
- DRV_CANFDSPI_ReadWord

```
//Test 3

//Test WriteWord & ReadWord

address = 0x400;
DRV_CANFDSPI_WriteWord(0, address, 0x11223344);
DRV_CANFDSPI_ReadWord(0, address, &value);
Nop();
```

Figure 11. WriteWord & ReadWord test code. Source: Own.

Results:

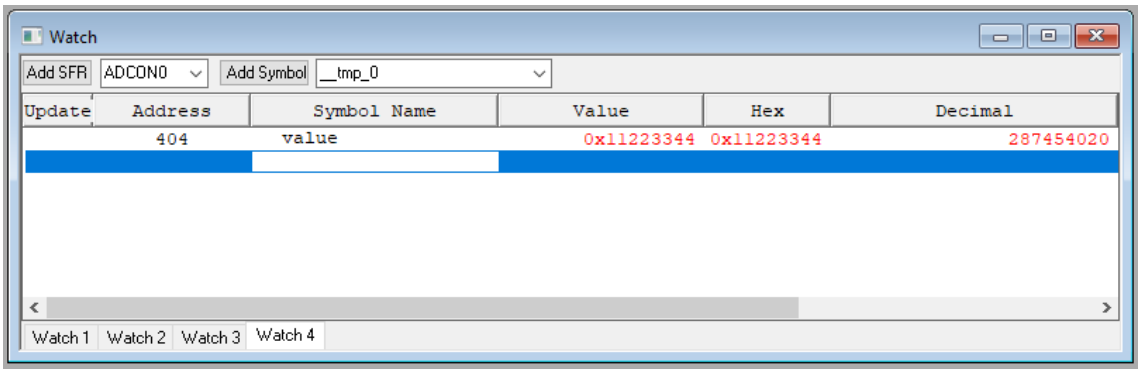


Figure 12. Results of the WriteWord & ReadWord test. Source: Own.

As expected, Figure 12 shows that the value of the variable “value” – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.1.4. Test WriteHalfWord & ReadHalfWord

Using the code shown in Figure 13, the following SPI instructions are tested:

- DRV_CANFDSPI_WriteHalfWord
- DRV_CANFDSPI_ReadHalfWord

```
//Test 4

//Test WriteHalfWord & ReadHalfWord

address = 0x010;
DRV_CANFDSPI_Reset(0);
DRV_CANFDSPI_WriteHalfWord(0, address, 0x5A5A);
DRV_CANFDSPI_ReadHalfWord(0, address, &value);
Nop();
```

Figure 13. WriteHalfWord & ReadHalfWord test code. Source: Own.

Results:

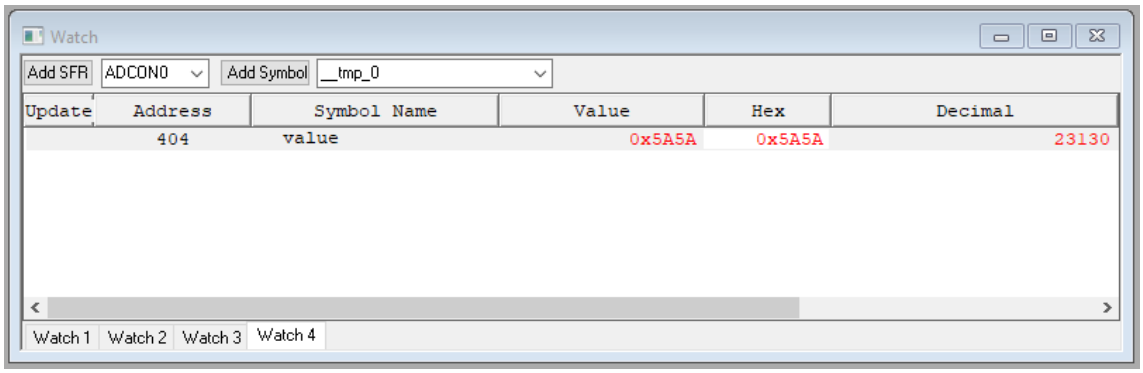


Figure 14. Results of the WriteHalfWord & ReadHalfWord test. Source: Own.

As expected, Figure 14 shows that the value of the variable “value” – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.1.5. Test WriteHalfWord & ReadWord

Using the code shown in Figure 15, the following SPI instructions are tested:

- DRV_CANFDSPI_WriteHalfWord
- DRV_CANFDSPI_ReadWord

```
//Test 5

//Test WriteHalfWord & ReadWord

address = 0x010;
DRV_CANFDSPI_Reset(0);
DRV_CANFDSPI_WriteHalfWord(0, address, 0x5555);
DRV_CANFDSPI_WriteHalfWord(0, 0x012, 0xAAAA);
DRV_CANFDSPI_ReadWord(0, address, &value);
Nop();
```

Figure 15. WriteHalfWord & ReadWord test code. Source: Own.

Results:

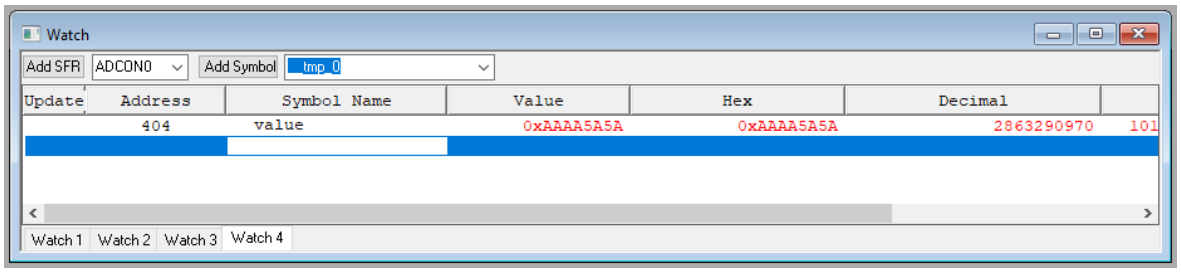


Figure 16. Results of the WriteHalfWord & ReadWord test. Source: Own.

As expected, Figure 16 shows that the value of the variable “value” – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.1.6. Test WriteByteArray & ReadByteArray

Using the code shown in Figure 17, the following SPI instructions are tested:

- DRV_CANFDSPI_WriteByteArray
- DRV_CANFDSPI_ReadByteArray

```
// Test 6

//Test WriteByteArray & ReadByteArray


for (length = 4; length <= 16; length++)
    {
    for (i = 0; i < 16; i++)
        {
          txd[i] = rand() & 0xff;
          rxd[i] = 0x00;
        }
    address = 0x400;

    DRV_CANFDSPI_Reset(0);
    DRV_CANFDSPI_WriteByteArray(0, address, txd, 16);
    DRV_CANFDSPI_ReadByteArray(0, address, rxd, 16);
    Nop();
    }
```

Figure 17. WriteByteArray & ReadByteArray test code. Source: Own.

Results:

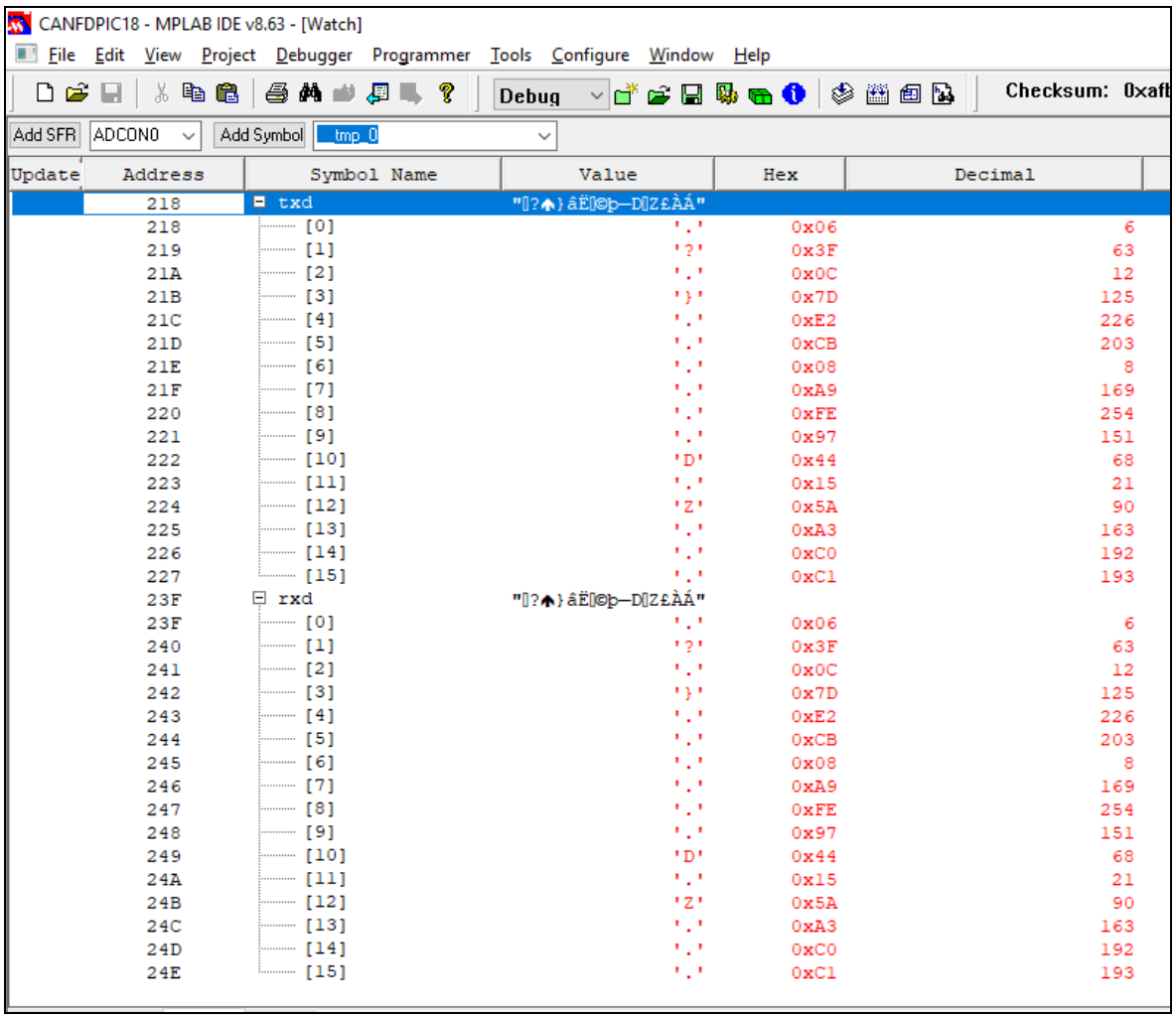


Figure 18. Results of the WriteByteArray & ReadByteArray test. Source: Own.

As expected, Figure 18 shows that the value of the variable “rxd” – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.1.7. Test WriteWordArray & ReadWordArray

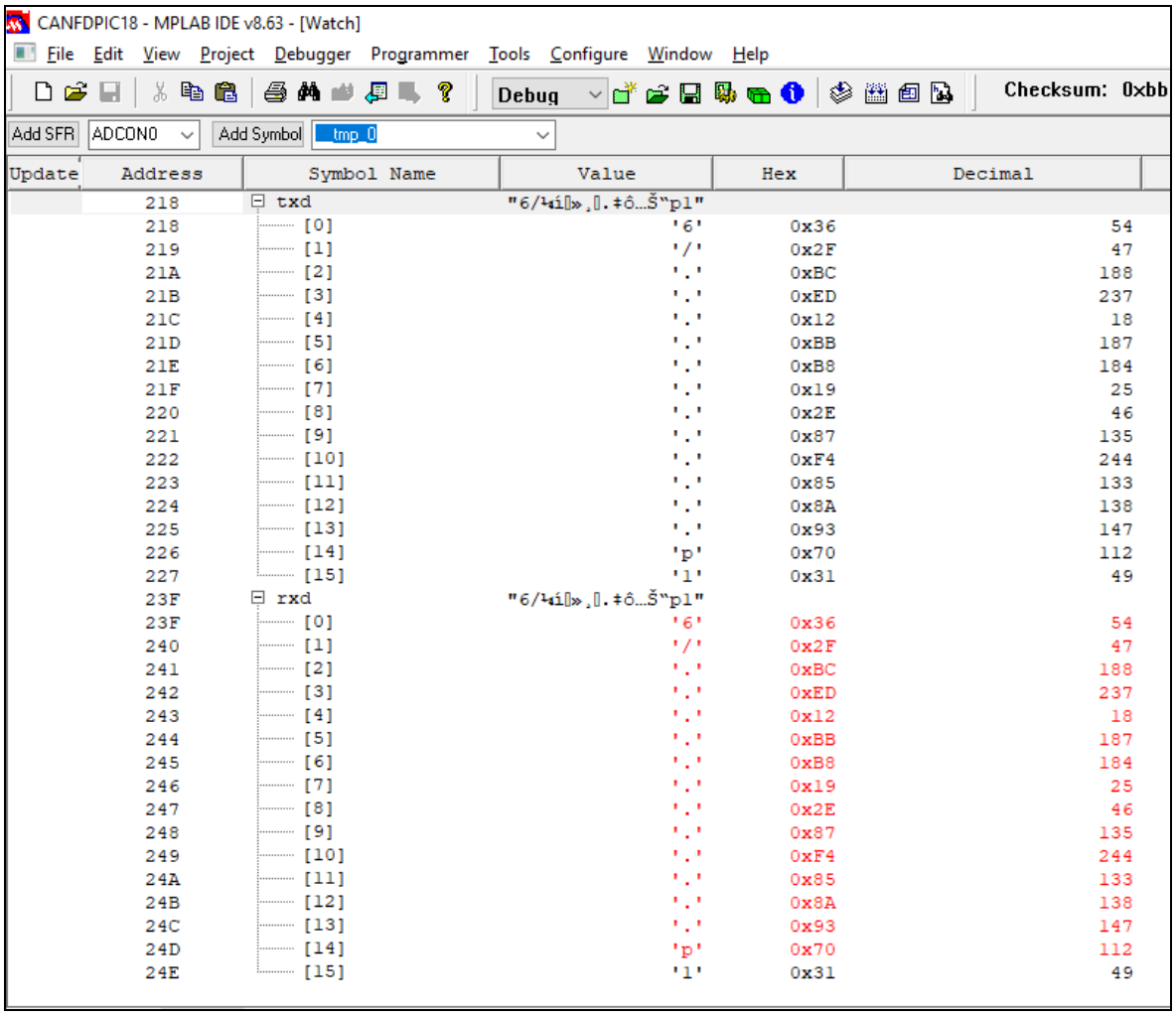
Using the code shown in Figure 19, the following SPI instructions are tested:

- DRV_CANFDSPI_WriteWordArray
- DRV_CANFDSPI_ReadWordArray

```
// Test 7

//Test WriteWordArray & ReadWordArray


    for (i = 0; i < 15; i++)
        {
          txd[i] = rand() & 0xff;
          rxd[i] = 0x00;
        }
    address = 0x400;

    DRV_CANFDSPI_Reset(0);
    DRV_CANFDSPI_WriteWordArray(0, address, txd, 4);
    DRV_CANFDSPI_ReadWordArray(0, address, rxd, 4);
    Nop();
```

Figure 19. WriteByteArray & ReadByteArray test code. Source: Own.

Results:



Update	Address	Symbol Name	Value	Hex	Decimal
	218	txd	"6/¼i▯»¸▯.‡ô…Š"p1"		
	218	[0]	'6'	0x36	54
	219	[1]	'/'	0x2F	47
	21A	[2]	'.'	0xBC	188
	21B	[3]	'.'	0xED	237
	21C	[4]	'.'	0x12	18
	21D	[5]	'.'	0xBB	187
	21E	[6]	'.'	0xB8	184
	21F	[7]	'.'	0x19	25
	220	[8]	'.'	0x2E	46
	221	[9]	'.'	0x87	135
	222	[10]	'.'	0xF4	244
	223	[11]	'.'	0x85	133
	224	[12]	'.'	0x8A	138
	225	[13]	'.'	0x93	147
	226	[14]	'p'	0x70	112
	227	[15]	'1'	0x31	49
	23F	rxd	"6/¼i▯»¸▯.‡ô…Š"p1"		
	23F	[0]	'6'	0x36	54
	240	[1]	'/'	0x2F	47
	241	[2]	'.'	0xBC	188
	242	[3]	'.'	0xED	237
	243	[4]	'.'	0x12	18
	244	[5]	'.'	0xBB	187
	245	[6]	'.'	0xB8	184
	246	[7]	'.'	0x19	25
	247	[8]	'.'	0x2E	46
	248	[9]	'.'	0x87	135
	249	[10]	'.'	0xF4	244
	24A	[11]	'.'	0x85	133
	24B	[12]	'.'	0x8A	138
	24C	[13]	'.'	0x93	147
	24D	[14]	'p'	0x70	112
	24E	[15]	'1'	0x31	49

Figure 20. Results of the WriteWordArray & ReadWordArray test.

As expected, Figure 20 shows that the value of the variable “rxd” – read by the reading function- is the written before in the same address. Therefore, these functions do work properly.

6.2. CAN TESTS

6.2.1. CONFIGURATION

The CAN FD controller module needs to be configured to run its functions properly. The following fields need to be set. In order to do so – aside from some mentioned occasions- it is necessary to be in configuration mode. The CANFD controller configuration is implemented in the “main.c” file.

6.2.1.1. Reset

The MCP2517FD should be reset to its initial values. This prevents the system to use any previously set value that makes the system work in a different way. By doing this, the MCP2517FD is set in Configuration mode. While in this mode, the system does not access the CAN, therefore it is not likely that the device disturbs the bus. Also, it is necessary that the system is in Configuration mode in order to configure some features of the system, such as the FIFOS or the oscillator.

6.2.1.2. Initialize RAM

It is necessary to initialize the RAM and fill it with any chosen value.

6.2.1.3. CAN configuration

The module supports ISO CRC and non-ISO CRC. It is necessary to enable or not this feature by setting the ISOCREN bit.

If preferred, RAM space must be reserved for the TEF and the TXQ by setting the STEF and TXQEN bits, respectively.

6.2.1.4. Set up TX and Rx FIFOs

The message objects of the Transmit and Receive FIFOs are in the RAM. Therefore, the application must configure the number and the payload size of the message objects inside each FIFO. The location of the objects in the RAM is determined by this configuration.

6.2.1.5. Filter & Mask

In order to set the Filter and Mask values, it is necessary to disable the filter first. It is not necessary though to be in configuration mode. When the object of the filter is set, it must be linked to the FIFO where the matching receive will be stored.

The filter and the mask are highly related – as the combination between them dictates which messages are received – but their function is different:

- Filter: The message objects that got through the mask must comply some conditions. The filter establishes which condition.

- Mask: Dictates which bits must be taking into account with the corresponding filter.

6.2.1.6. Bit time

In order to achieve higher bandwidth, bits inside a CAN FD frame can be transmitted in two different bit rates.

- Nominal Bit Rate (NBR). Used during the arbitration. It is the number of bits per second during this phase. It is the inverse of the Nominal Bit Time (NBT).

- Data Bit Rate (DBR). It is used during the data transferring.

These two values must be set up.

In order to enable a data phase bit time that is shorter than the transceiver loop delay, the Transmitter Delay Compensation (TDC) is implemented. To do so, a Secondary Sample Point (SSP) is calculated. This value can be automatically calculated by the system, if preferred, choosing the proper option.

Regarding the CAN clock frequency, it is advisable to choose the highest available – usually 20 or 40 MHz.

6.2.1.7. Select mode

Finally, it is necessary to choose the mode. In this project, aside from the Configuration mode, only two modes are used. These two modes are the following:

CAN FD Normal mode. In this mode, the device will be able to transmit and receive messages in CAN FD mode: bit rate switching can be enabled and up to 64 bytes can be transmitted and received.

CAN FD LoopBack mode. This mode is among the Debug modes and is a variant of the CAN Normal mode. This mode will allow internal transmission of messages from the transmit FIFOs to the receive FIFOs. More specifically it will be used the External LoopBack mode, where the transmit signal is internally connected to receive and transmit messages can be monitored on the TXCAN pin.

The next example shows how to configure the MCP2517FD.

Configuration code

```
// Reset
    DRV_CANFDSPI_Reset(0);

// Initialize RAM
    DRV_CANFDSPI_EccEnable(DRV_CANFDSPI_INDEX_0);
    if (!ramInitialized) {
        DRV_CANFDSPI_RamInit(DRV_CANFDSPI_INDEX_0, 0x00); //0xff);
        ramInitialized = true;
    }

// CAN configuration
    DRV_CANFDSPI_ConfigureObjectReset(&config);
    config.IsoCrcEnable = ISO_CRC;
    config.StoreInTEF = 0;
    config.TXQEnable = 0;
    DRV_CANFDSPI_Configure(0, &config);

//Setup Tx & Rx FIFOs
// Setup TX FIFO
    DRV_CANFDSPI_TransmitChannelConfigureObjectReset(&txConfig);
    txConfig.FifoSize = 0; //7;
    txConfig.PayLoadSize = CAN_PLSIZE_64;
    txConfig.TxPriority = 0;
    //txConfig.TxEnable = 1;
    DRV_CANFDSPI_TransmitChannelConfigure(0, APP_TX_FIFO, &txConfig);

    // Setup RX FIFO
    DRV_CANFDSPI_ReceiveChannelConfigureObjectReset(&rxConfig);
    rxConfig.FifoSize = 0;
    rxConfig.PayLoadSize = CAN_PLSIZE_64;
    rxConfig.RxTimeStampEnable = 1;
    DRV_CANFDSPI_ReceiveChannelConfigure(0, APP_RX_FIFO, &rxConfig);

//Filter & Mask
    DRV_CANFDSPI_FilterDisable(0, CAN_FILTER0);

// Setup RX Filter
```

```
    fObj.word = 0;
    fObj.bF.SIDA = 0x300>>3 ;
    fObj.bF.SIDB = 0x300 & (0x03) ;
    fObj.bF.EXIDE = 0;
    fObj.bF.EIDA = 0x00;
    fObj.bF.EIDB = 0x00;
    fObj.bF.EIDC = 0x00;
    DRV_CANFDSPI_FilterObjectConfigure(0, CAN_FILTER0, &fObj.bF);

// Setup RX Mask
    mObj.word = 0;
    mObj.bF.MSIDA = 0x0;
    mObj.bF.MSIDB = 0x0;
    mObj.bF.MIDE = 1; // Only allow standard IDs
    mObj.bF.MEIDA = 0x0;
    mObj.bF.MEIDB = 0x0;
    DRV_CANFDSPI_FilterMaskConfigure(0, CAN_FILTER0, &mObj.bF);
// Link FIFO and Filter
    DRV_CANFDSPI_FilterToFifoLink(0, CAN_FILTER0, APP_RX_FIFO, true);

//Bit time
// Setup Bit Time
    DRV_CANFDSPI_BitTimeConfigure(0, selectedBitTime, CAN_SSP_MODE_AUTO, CAN_SYSCLK_40M);

//Select mode
// Select Normal Mode
     DRV_CANFDSPI_OperationModeSelect(0, CAN_NORMAL_MODE);
```

Table 13. CAN configuration code example. Source: Own.

6.2.2. LoopBack mode tests

As mentioned before, the LoopBack mode allows the user to receive the message sent by the same CANFD controller. It is, then, a good exercise to make the first tests of the CAN without having a complete CAN network.

There are four variables that will be modified each test to make sure that the system works. These variables are the following:

- DLC. Refers to the Data Length Code.
- Txd. This is the variable used to store the data that will be transferred.
- FIFO size. Number of FIFOs.
- IDE. Refers to the standard identifier.
- Mask & Filter. They can be applied or not.

Also, masks and filters will be applied. Therefore, there are messages that will not be received, and messages that will be received partially.

Following there are several tests using the LoopBack mode.

6.2.2.1. Test 1

Table 14 shows the variable values that will be used for the Test 1.

Variable	Value
DLC	8
Txd[i]	0x0A + i
Tx FIFO size	1
Rx FIFO size	1
IDE	0x300
Mask	None
Filter	None

Table 14. Test 1 variable values. Source: Own.

Results:

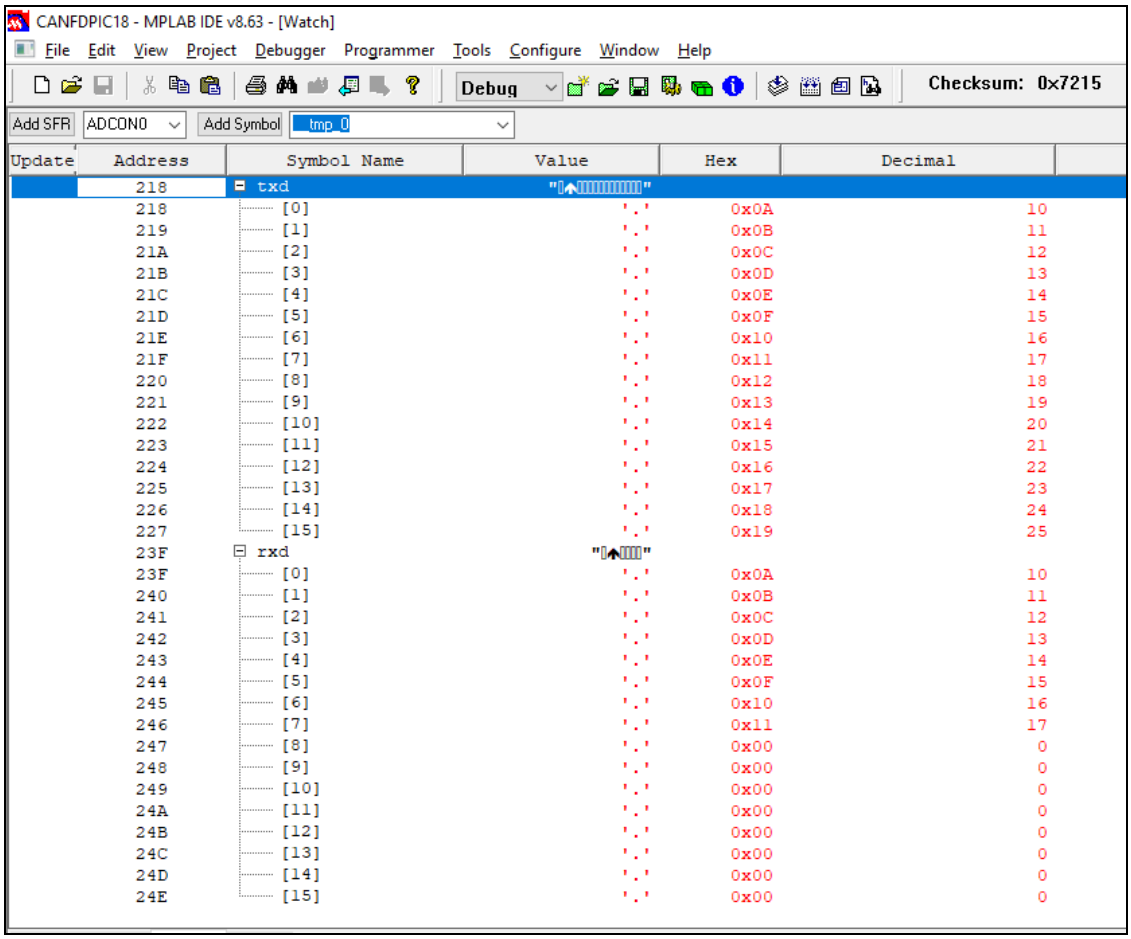


Figure 21. Test 1 results.

As expected, Figure 21 shows that the value of the variable “rxd” – received message- is the same as the “txd” – sent message-. Therefore, the system works properly.

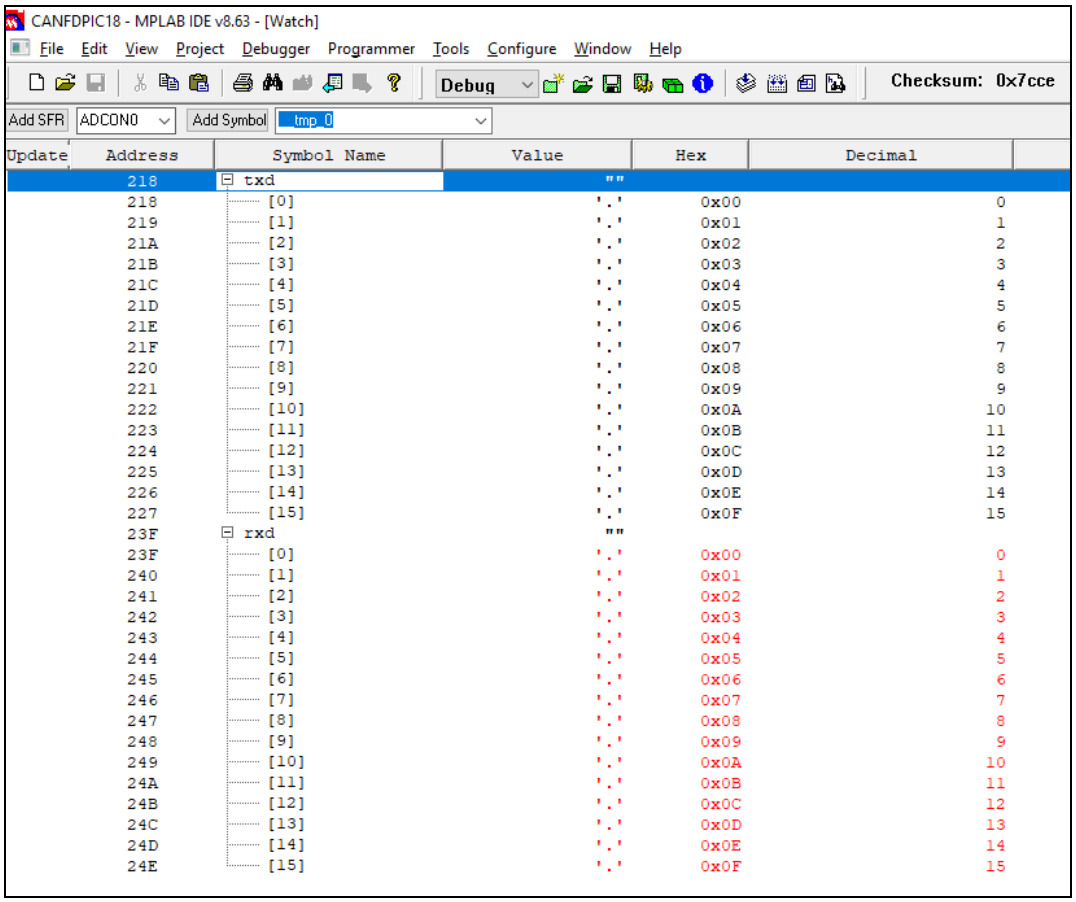
6.2.2.2. Test 2

Table 15 shows the variable values that will be used for the Test 2.

Variable	Value
DLC	16
Txd[i]	i
Tx FIFO size	16
Rx FIFO size	16
IDE	0x400
Mask	None
Filter	None

Table 15. Test 2 variable values. Source: Own.

Results:



Update	Address	Symbol Name	Value	Hex	Decimal
	218	txd	""		
	218	[0]	'.'	0x00	0
	219	[1]	'.'	0x01	1
	21A	[2]	'.'	0x02	2
	21B	[3]	'.'	0x03	3
	21C	[4]	'.'	0x04	4
	21D	[5]	'.'	0x05	5
	21E	[6]	'.'	0x06	6
	21F	[7]	'.'	0x07	7
	220	[8]	'.'	0x08	8
	221	[9]	'.'	0x09	9
	222	[10]	'.'	0x0A	10
	223	[11]	'.'	0x0B	11
	224	[12]	'.'	0x0C	12
	225	[13]	'.'	0x0D	13
	226	[14]	'.'	0x0E	14
	227	[15]	'.'	0x0F	15
	23F	rxd	""		
	23F	[0]	'.'	0x00	0
	240	[1]	'.'	0x01	1
	241	[2]	'.'	0x02	2
	242	[3]	'.'	0x03	3
	243	[4]	'.'	0x04	4
	244	[5]	'.'	0x05	5
	245	[6]	'.'	0x06	6
	246	[7]	'.'	0x07	7
	247	[8]	'.'	0x08	8
	248	[9]	'.'	0x09	9
	249	[10]	'.'	0x0A	10
	24A	[11]	'.'	0x0B	11
	24B	[12]	'.'	0x0C	12
	24C	[13]	'.'	0x0D	13
	24D	[14]	'.'	0x0E	14
	24E	[15]	'.'	0x0F	15

Figure 22. Test 2 results.

As expected, Figure 22 shows that the value of the variable “rxd” – received message- is the same as the “txd” – sent message-. Therefore, the system works properly.

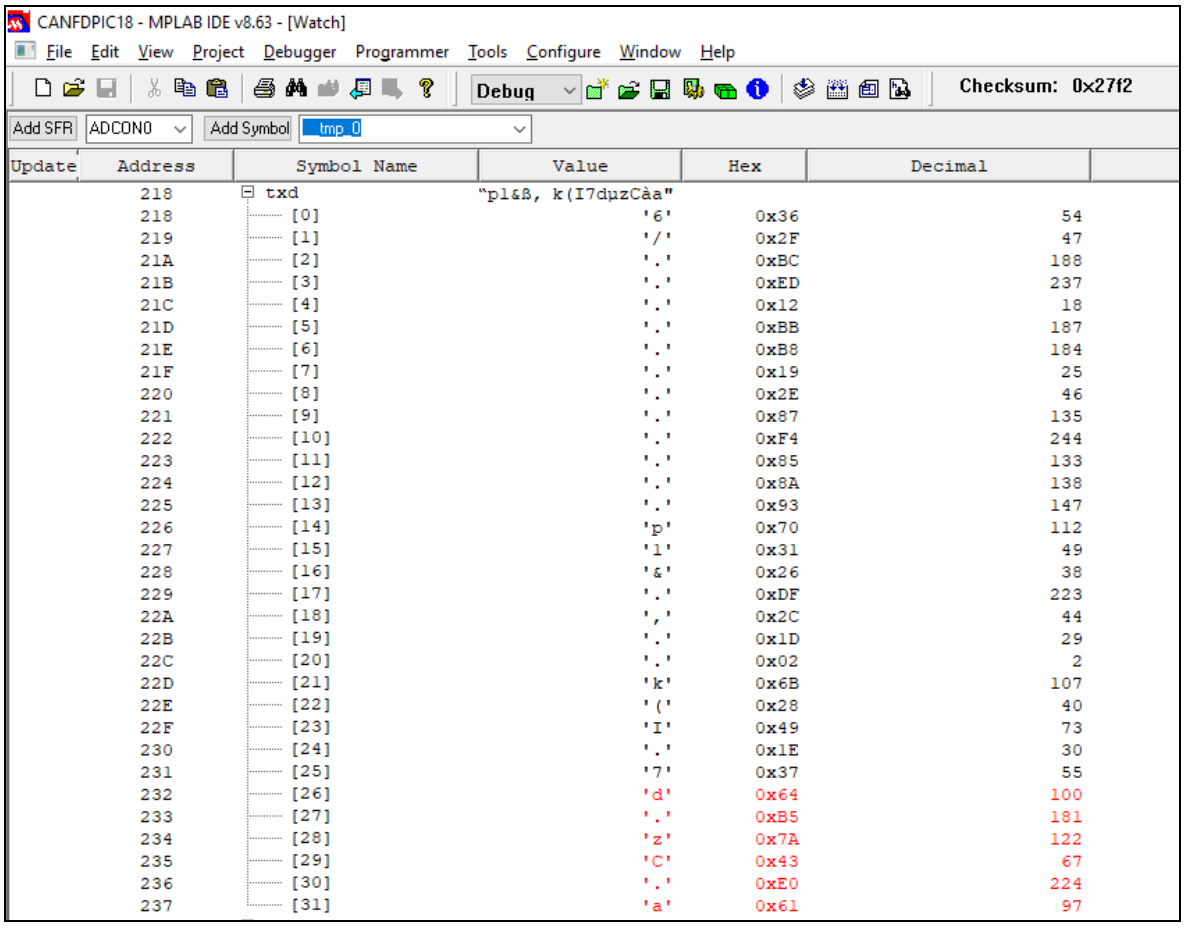
6.2.2.3. Test 3

Table 16 shows the variable values that will be used for the Test 3.

Variable	Value
DLC	32
Txd[i]	Random
Tx FIFO size	8
Rx FIFO size	1
IDE	0x400
Mask	None
Filter	None

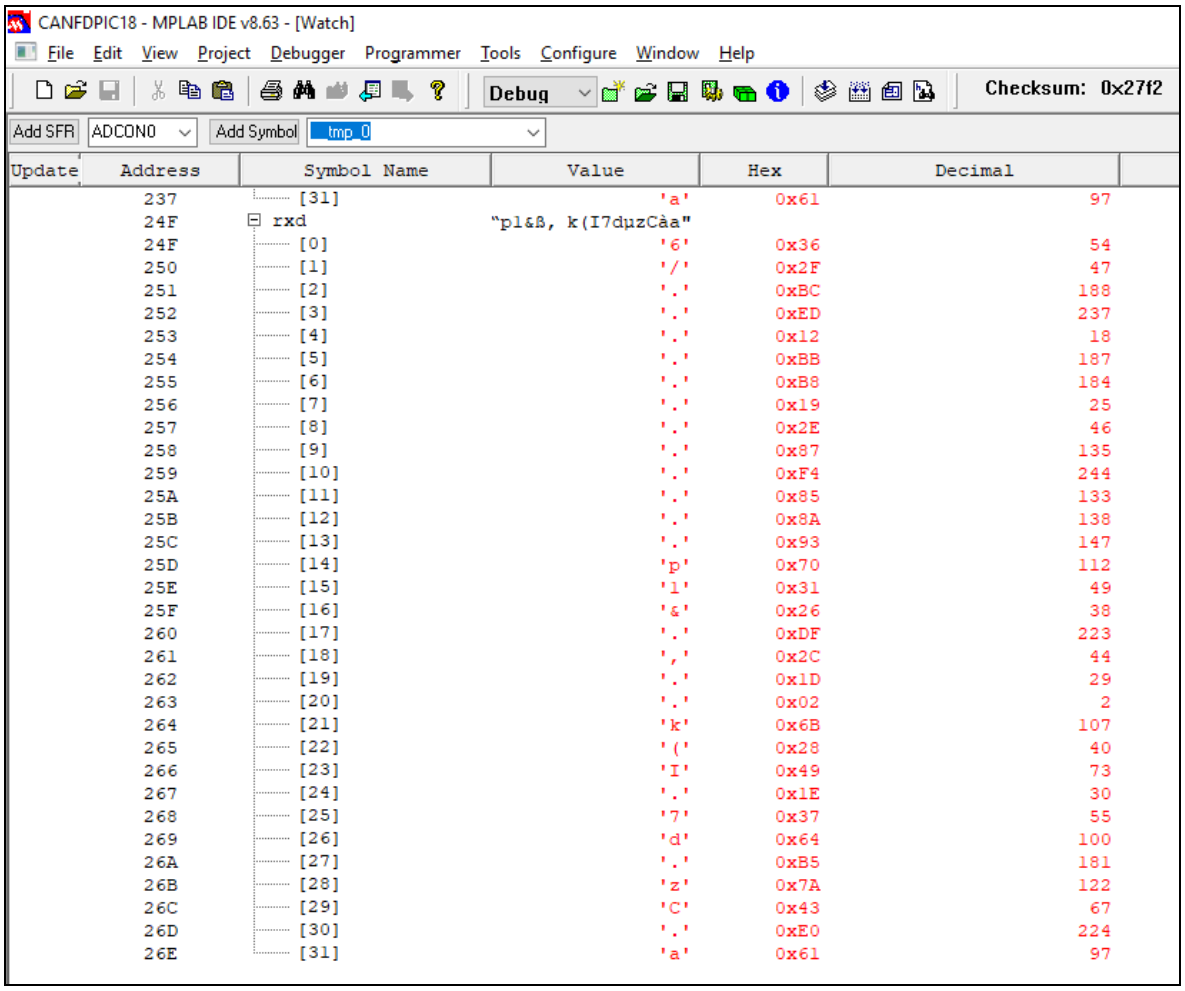
Table 16. Test 3 variable values. Source: Own.

Results:



Update	Address	Symbol Name	Value	Hex	Decimal
	218	txd	"pl&ß, k(I7dµzCàa"		
	218	[0]	'6'	0x36	54
	219	[1]	'/'	0x2F	47
	21A	[2]	'.'	0xBC	188
	21B	[3]	'.'	0xED	237
	21C	[4]	'.'	0x12	18
	21D	[5]	'.'	0xBB	187
	21E	[6]	'.'	0xB8	184
	21F	[7]	'.'	0x19	25
	220	[8]	'.'	0x2E	46
	221	[9]	'.'	0x87	135
	222	[10]	'.'	0xF4	244
	223	[11]	'.'	0x85	133
	224	[12]	'.'	0x8A	138
	225	[13]	'.'	0x93	147
	226	[14]	'p'	0x70	112
	227	[15]	'l'	0x31	49
	228	[16]	'&'	0x26	38
	229	[17]	'.'	0xDF	223
	22A	[18]	','	0x2C	44
	22B	[19]	'.'	0x1D	29
	22C	[20]	'.'	0x02	2
	22D	[21]	'k'	0x6B	107
	22E	[22]	'('	0x28	40
	22F	[23]	'I'	0x49	73
	230	[24]	'.'	0x1E	30
	231	[25]	'7'	0x37	55
	232	[26]	'd'	0x64	100
	233	[27]	'.'	0xB5	181
	234	[28]	'z'	0x7A	122
	235	[29]	'C'	0x43	67
	236	[30]	'.'	0xE0	224
	237	[31]	'a'	0x61	97

Figure 23. Test 3 results. Source: Own.



Update	Address	Symbol Name	Value	Hex	Decimal
	237	[31]	'a'	0x61	97
	24F	rxd	"pl&ß, k(I7dµzCàa"		
	24F	[0]	'6'	0x36	54
	250	[1]	'/'	0x2F	47
	251	[2]	'.'	0xBC	188
	252	[3]	'.'	0xED	237
	253	[4]	'.'	0x12	18
	254	[5]	'.'	0xBB	187
	255	[6]	'.'	0xB8	184
	256	[7]	'.'	0x19	25
	257	[8]	'.'	0x2E	46
	258	[9]	'.'	0x87	135
	259	[10]	'.'	0xF4	244
	25A	[11]	'.'	0x85	133
	25B	[12]	'.'	0x8A	138
	25C	[13]	'.'	0x93	147
	25D	[14]	'p'	0x70	112
	25E	[15]	'1'	0x31	49
	25F	[16]	'&'	0x26	38
	260	[17]	'.'	0xDF	223
	261	[18]	','	0x2C	44
	262	[19]	'.'	0x1D	29
	263	[20]	'.'	0x02	2
	264	[21]	'k'	0x6B	107
	265	[22]	'('	0x28	40
	266	[23]	'I'	0x49	73
	267	[24]	'.'	0x1E	30
	268	[25]	'7'	0x37	55
	269	[26]	'd'	0x64	100
	26A	[27]	'.'	0xB5	181
	26B	[28]	'z'	0x7A	122
	26C	[29]	'C'	0x43	67
	26D	[30]	'.'	0xE0	224
	26E	[31]	'a'	0x61	97

Figure 24. Test 3 results. Source: Own.

As expected, Figures 23 and 24 show that the value of the variable “rxd” – received message- is the same as the “txd” – sent message-. Therefore, the system works properly.

6.2.2.4. Test 4

Table 17 shows the variable values that will be used for the Test 4.

Variable	Value
DLC	64
Txd[i]	64-i
Tx FIFO size	1
Rx FIFO size	16
IDE	0x500
Mask	None
Filter	None

Table 17. Test 4 variable values. Source: Own.

Results:

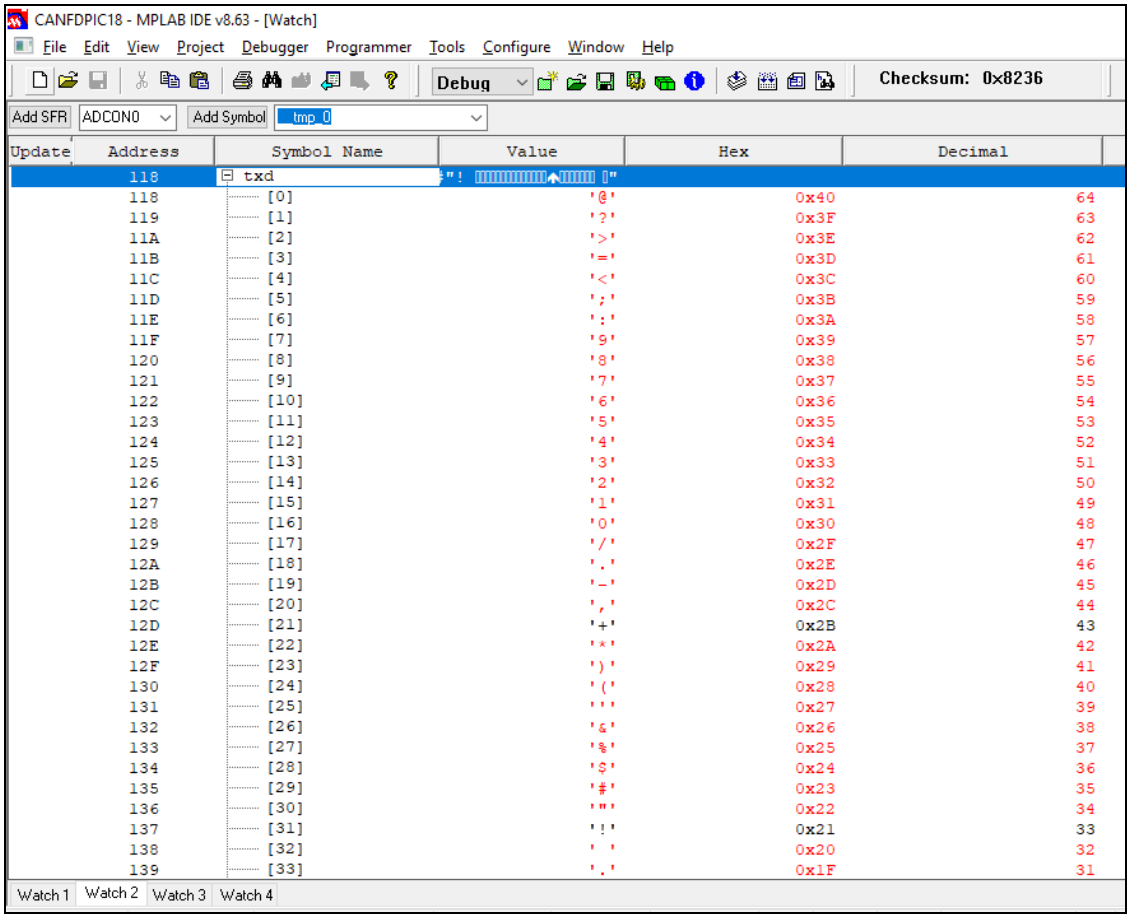
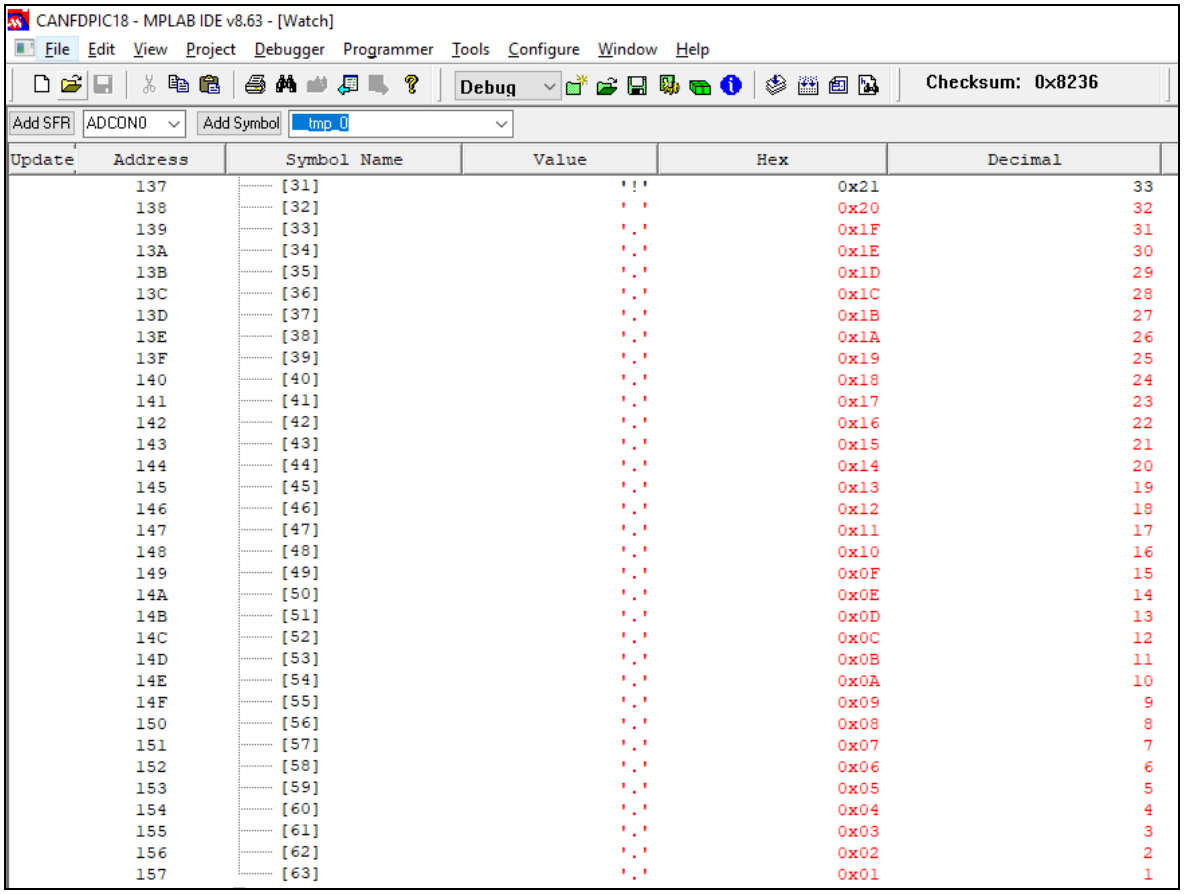


Figure 25. Test 4 results. Source: Own.



Update	Address	Symbol Name	Value	Hex	Decimal
	137	[31]	'!'	0x21	33
	138	[32]	' '	0x20	32
	139	[33]	'.'	0x1F	31
	13A	[34]	'.'	0x1E	30
	13B	[35]	'.'	0x1D	29
	13C	[36]	'.'	0x1C	28
	13D	[37]	'.'	0x1B	27
	13E	[38]	'.'	0x1A	26
	13F	[39]	'.'	0x19	25
	140	[40]	'.'	0x18	24
	141	[41]	'.'	0x17	23
	142	[42]	'.'	0x16	22
	143	[43]	'.'	0x15	21
	144	[44]	'.'	0x14	20
	145	[45]	'.'	0x13	19
	146	[46]	'.'	0x12	18
	147	[47]	'.'	0x11	17
	148	[48]	'.'	0x10	16
	149	[49]	'.'	0x0F	15
	14A	[50]	'.'	0x0E	14
	14B	[51]	'.'	0x0D	13
	14C	[52]	'.'	0x0C	12
	14D	[53]	'.'	0x0B	11
	14E	[54]	'.'	0x0A	10
	14F	[55]	'.'	0x09	9
	150	[56]	'.'	0x08	8
	151	[57]	'.'	0x07	7
	152	[58]	'.'	0x06	6
	153	[59]	'.'	0x05	5
	154	[60]	'.'	0x04	4
	155	[61]	'.'	0x03	3
	156	[62]	'.'	0x02	2
	157	[63]	'.'	0x01	1

Figure 26. Test 4 results. Source: Own.

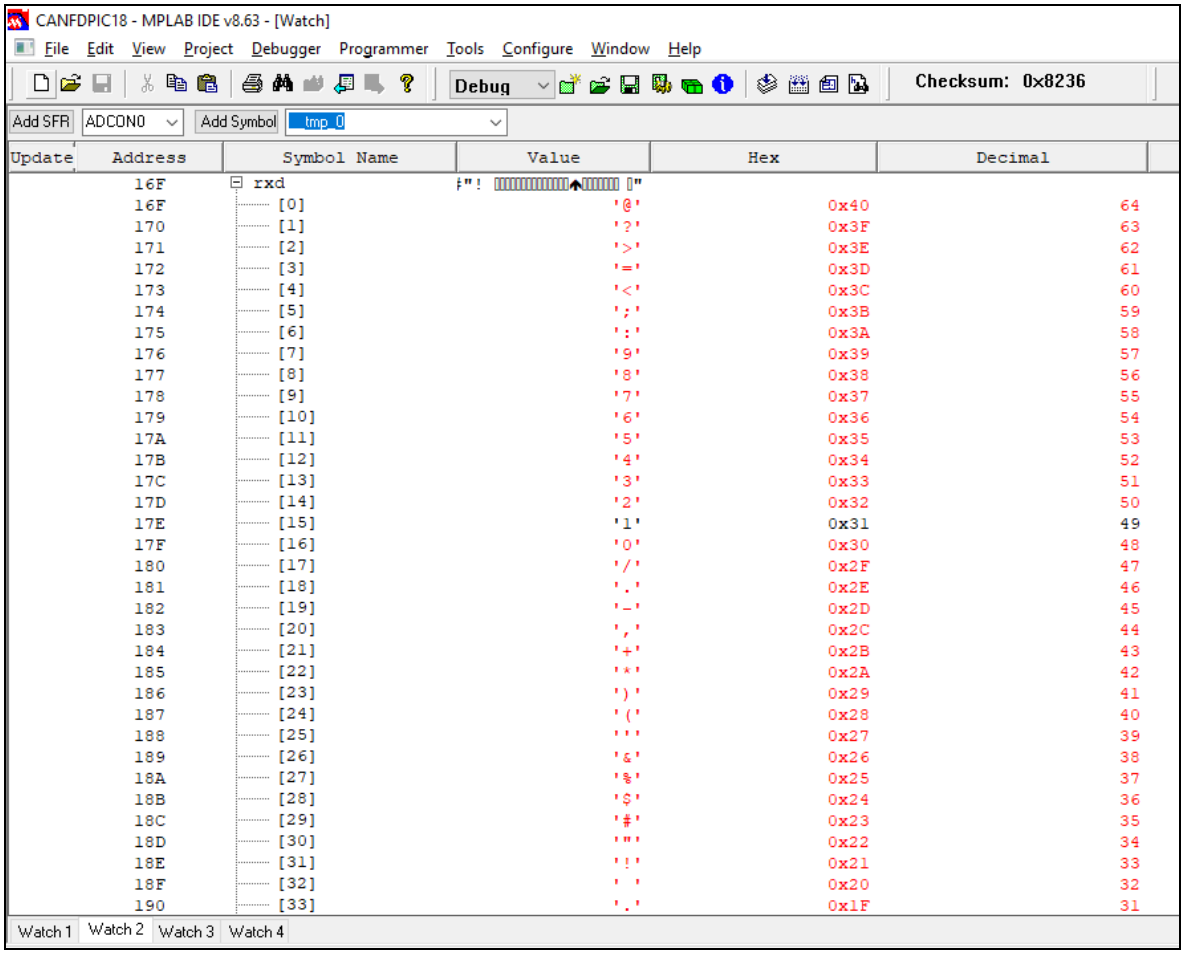


Figure 27. Test 4 results. Source: Own.

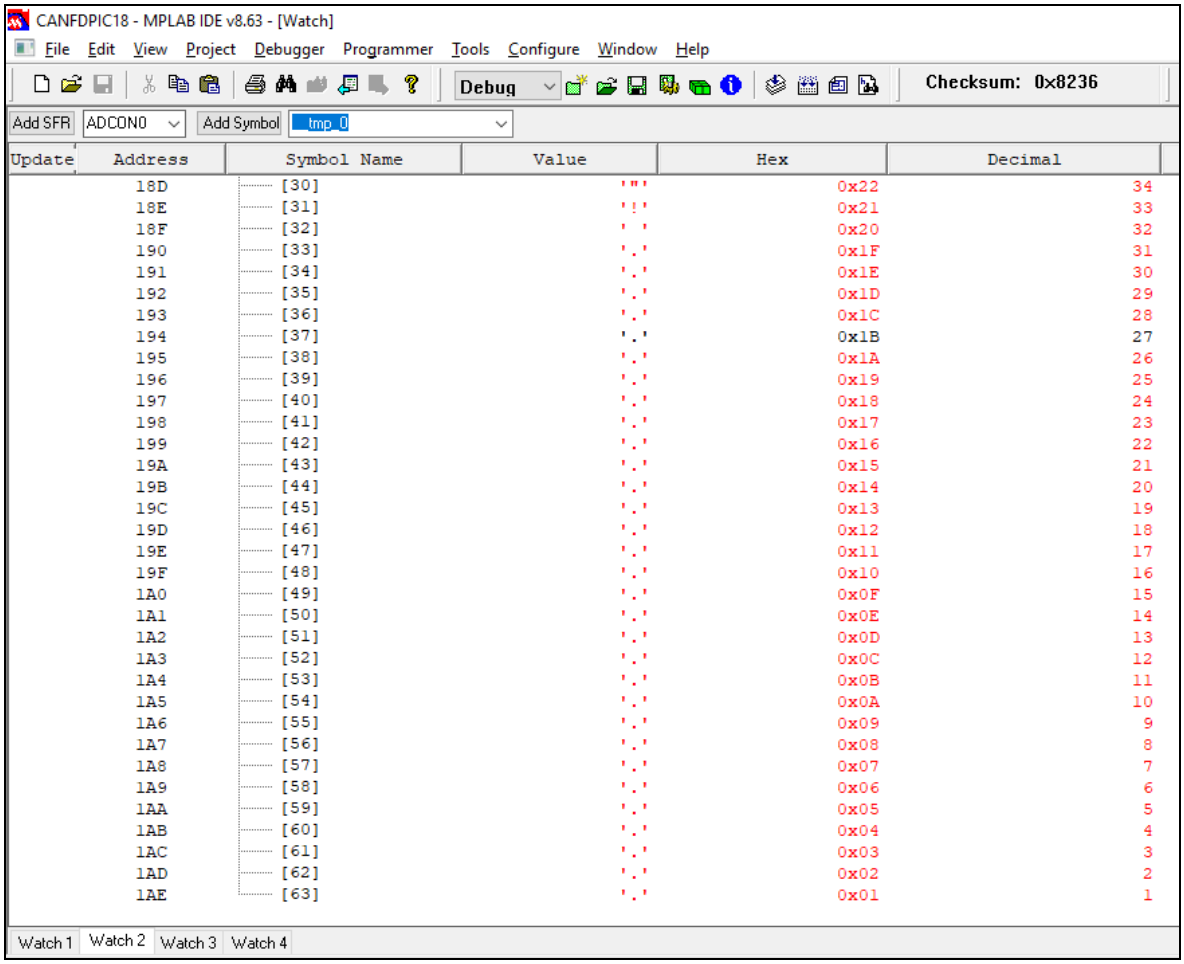


Figure 28. Test 4 results. Source: Own.

As expected, Figures 25, 26, 27 and 28 show that the value of the variable “rxd” – received message- is the same as the “txd” – sent message-. Therefore, the system works properly.

6.2.2.5. Test 5

Table 18 shows the variable values that will be used for the Test 5.

Variable	Value
DLC	32
Txd[i]	0x99
Tx FIFO size	8
Rx FIFO size	8
IDE	0x500
Mask	None
Filter	0x300

Table 18. Test 5 variable values. Source: Own.

Results:

As expected, as the filter only allow the messages with the 0x500 identifier, there is no message received. Therefore, the system works properly.

6.2.2.6. Test 6

Table 19 shows the variable values that will be used for the Test 6.

Variable	Value
DLC	32
Txd[i]	0x0A
Tx FIFO size	16
Rx FIFO size	1
IDE	0x300
Mask	0xff
Filter	None

Table 19. Test 6 variable values. Source: Own.

Results:

As expected, with the set mask, no messages get through, there are no messages received. Therefore, the system works properly.

6.2.3. Normal mode tests with CAN sniffer

Once it is proven that the system works properly using the LoopBack mode, it is possible to switch to the CANFD normal mode and send and receive messages through a real CANFD network.

In order to have evidence of the good functioning of the system, a CANFD sniffer is used. Through this sniffer, message objects will be sent and received, and therefore, if the results are the expected, it can be concluded that the system develops its function properly.

It will also be checked on the oscilloscope that the voltage levels are correct. On the oscilloscope it will be visible both the CAN HIGH and the CAN LOW. By doing this, the CAN FD frame will be clearly visible.

When using the sniffer, it will also be modified the arbitration and data bit time.

The tests will follow the same pattern used for the LoopBack mode tests.

In the Figure 29 it can be seen the CAN FD node connected to the CAN FD sniffer.

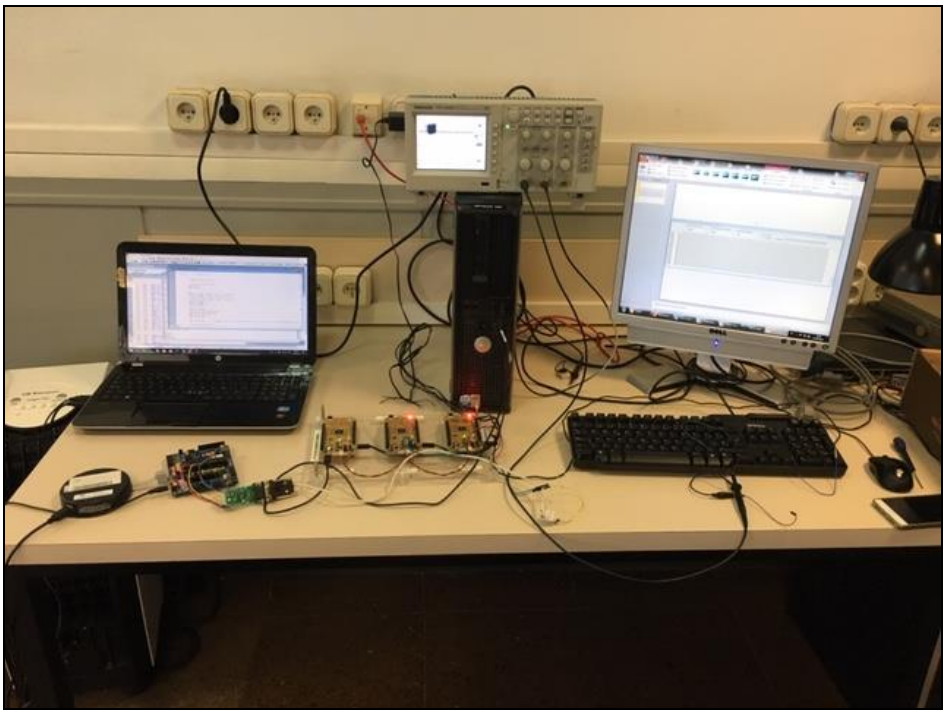


Figure 29. Assembling of the system. Source: Own.

Following there are several tests using the CANFD Normal mode and the CANFD sniffer.

6.2.3.1. Test 1

Table 20 shows the variable values that will be used for the Test 1.

Variable	Value
DLC	8
Txd[i]	i
Tx FIFO size	1
Rx FIFO size	1
IDE	0x300
Mask	None
Filter	None

Table 20. Test 1 variable values. Source: Own.

In the Figure 30 it can be clearly seen the CAN FD frame. Also, the differences of speeds between the arbitration zone and the payload zone. There is a payload of 8 bytes.

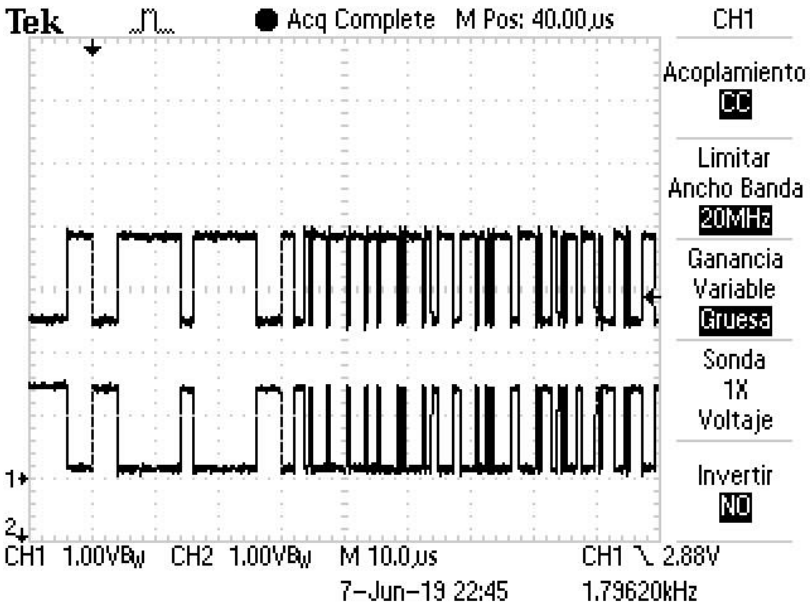


Figure 30. Test 1 voltage levels. Source: Own.

Results:

In Figure 31 it can be seen that the message sent through the real CAN FD network is the expected. Therefore, the system works properly.

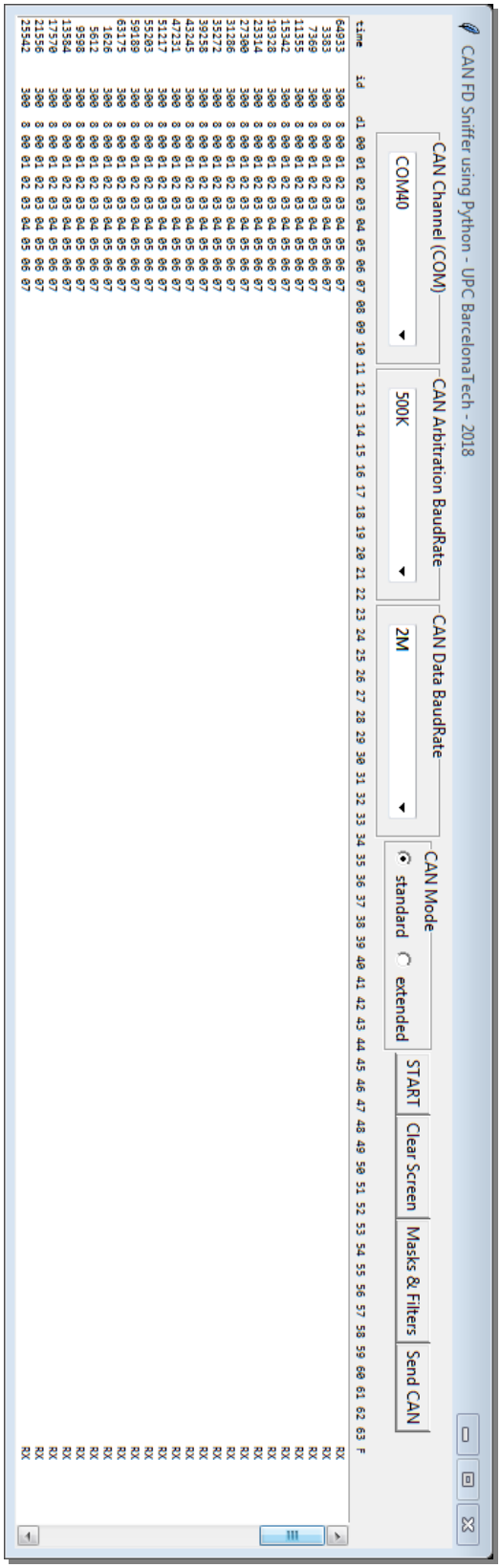


Figure 31. Test 1 results.

6.2.3.2. Test 2

Table 21 shows the variable values that will be used for the Test 2.

Variable	Value
DLC	12
Txd[i]	0xAA
Tx FIFO size	8
Rx FIFO size	8
IDE	0x500
Mask	None
Filter	None

Table 21. Test 2 variable values. Source: Own.

In Figure 32 it can be clearly seen the CAN FD frame. Also, the differences of speeds / baudrates between the arbitration zone and the payload zone. There is a payload of 12 bytes.

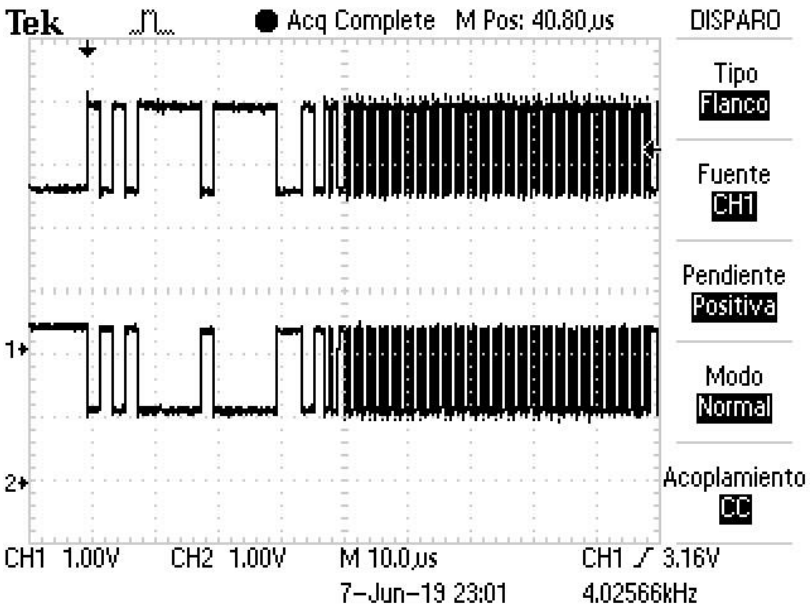


Figure 32. Test 2 voltage levels. Source: Own.

Results:

In Figure 33 it can be seen that the message sent through the real CAN FD network is the expected. Therefore, the system works properly.

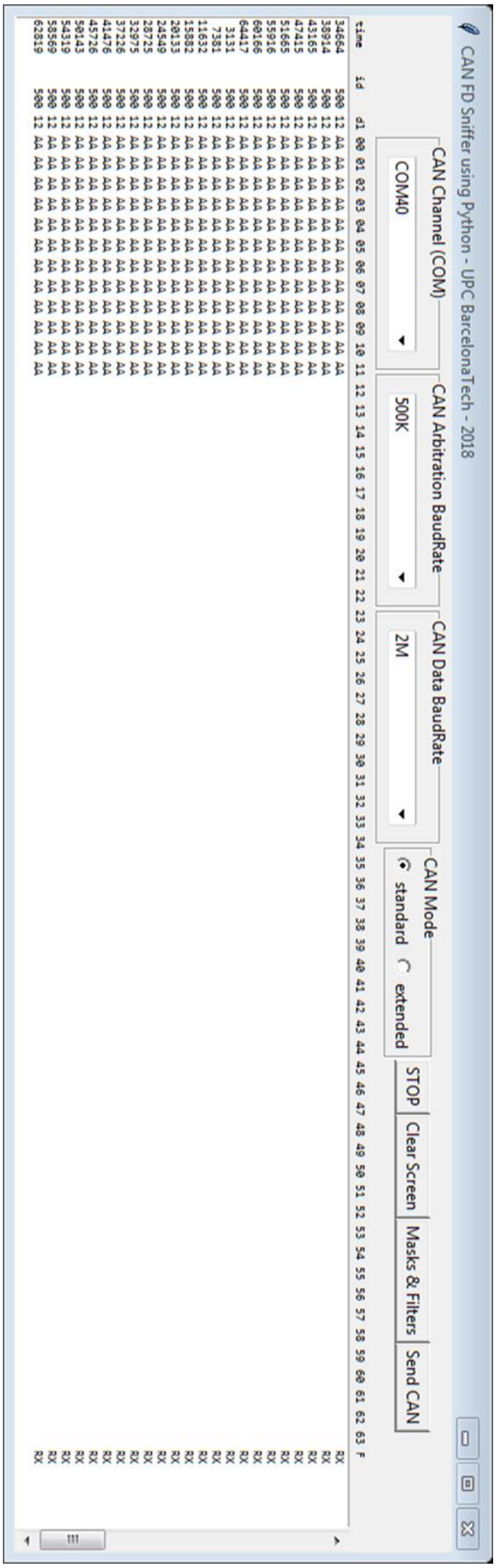


Figure 33. Test 2 results. Source: Own.

6.2.3.3. Test 3

Table 22 shows the variable values that will be used for the Test 3.

Variable	Value
DLC	12
Txd[i]	64 – i
Tx FIFO size	16
Rx FIFO size	1
IDE	0x500
Mask	None
Filter	None

Table 22. Test 3 variable values.

In Figure 34 it can be clearly seen the CAN FD frame. Also, the differences of speeds between the arbitration zone and the payload zone. There is a payload of 12 bytes.

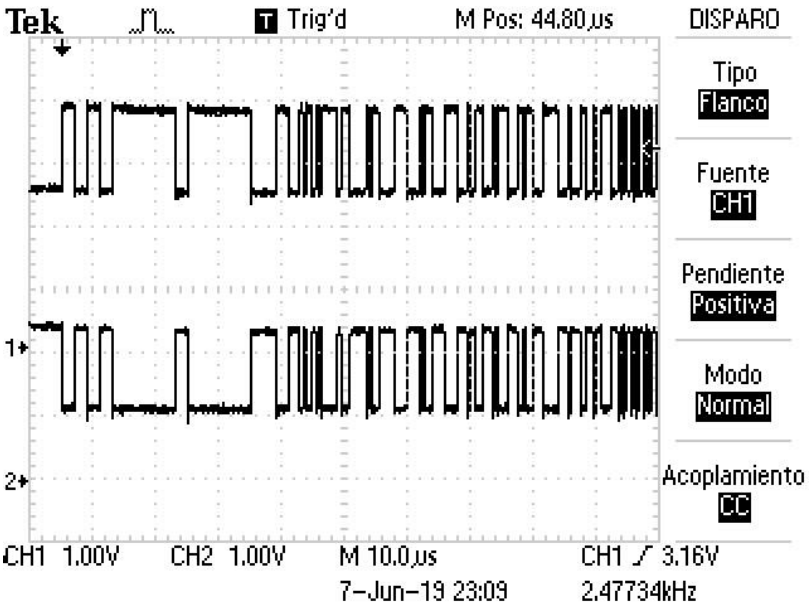


Figure 34. Test 3 voltage levels. Source: Own.

Results:

In Figure 35 it can be seen that the message sent through the real CAN FD network is the expected. Therefore, the system works properly.

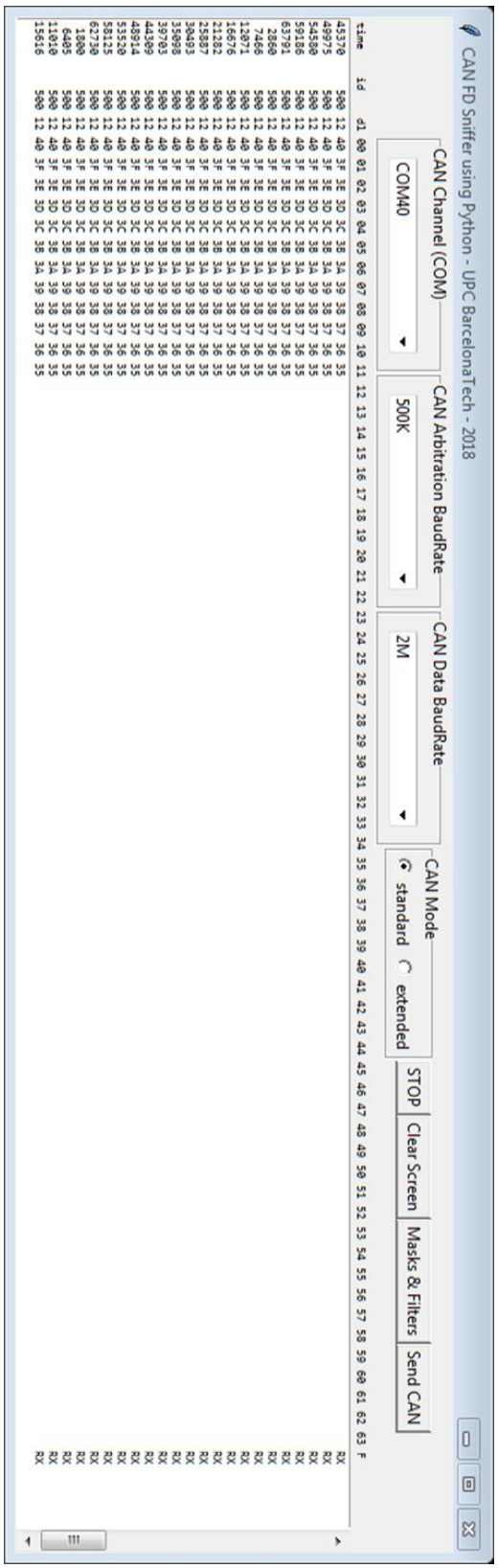


Figure 35. Test 3 results.

6.2.3.4. Test 4

Table 23 shows the variable values that will be used for the Test 4.

Variable	Value
DLC	64
Txd[i]	0x99
Tx FIFO size	1
Rx FIFO size	16
IDE	0x500
Mask	None
Filter	None

Table 23. Test 4 variable values. Source: Own.

In Figure 36 it can be clearly seen the CAN FD frame. Also, the differences of speeds between the arbitration zone and the payload zone. There is a payload of 64 bytes.

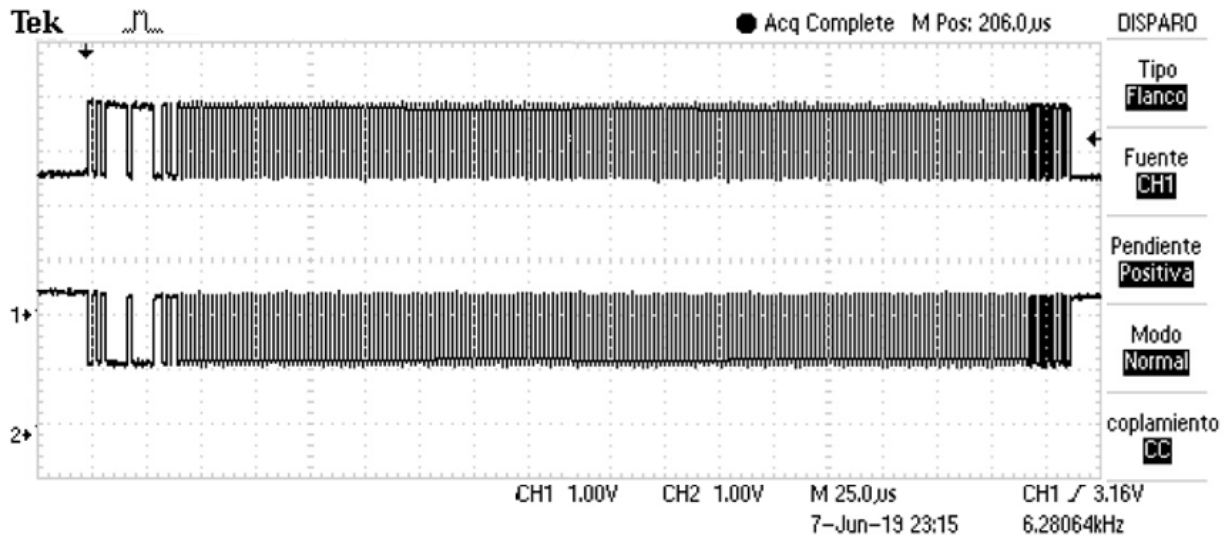


Figure 36. Test 4 voltage levels. Source: Own.

Results:

In Figure 37 it can be seen that the message sent through the real CAN FD network is the expected. Therefore, the system works properly.

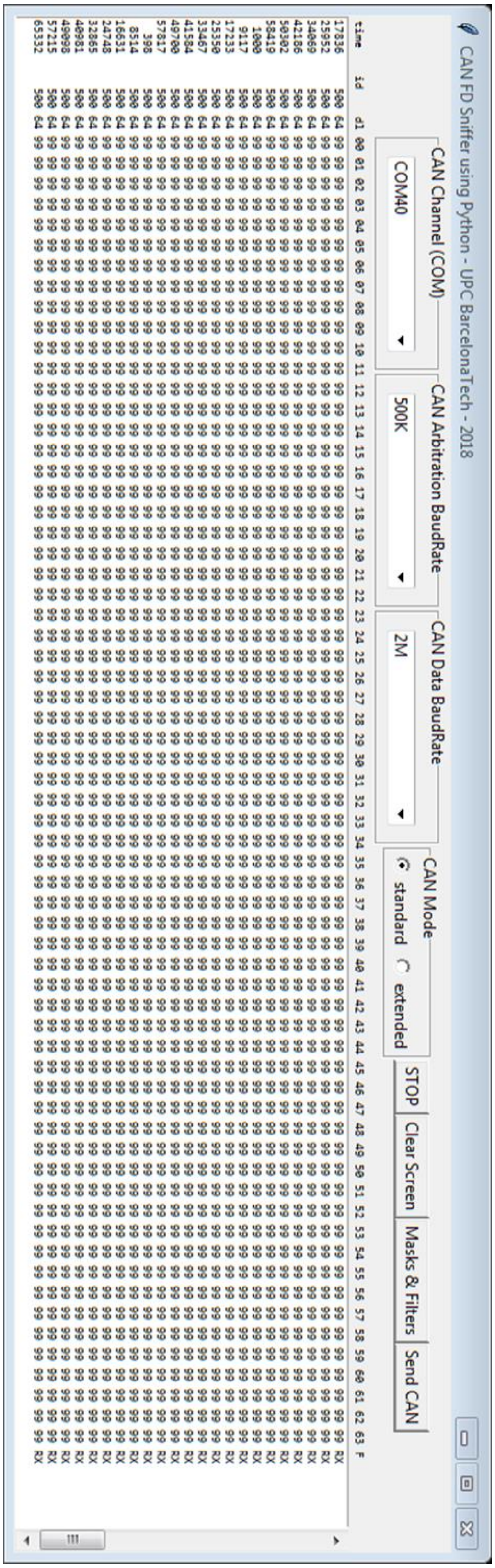


Figure 37. Test 4 results.

6.2.3.5. Test 5

Table 24 shows the variable values that will be used for the Test 5.

Variable	Value
DLC	32
Txd[i]	0xA0 + 2*i
Tx FIFO size	Random
Rx FIFO size	Random
IDE	Random
Mask	None
Filter	None
Arbitration bit time	250 kbit/s
Data bit time	1 Mbit/s

Table 24. Test 5 variable values. Source: Own.

Results:

In Figure 38 it can be seen that the message sent through the real CAN FD network is the expected. Therefore, the system works properly.

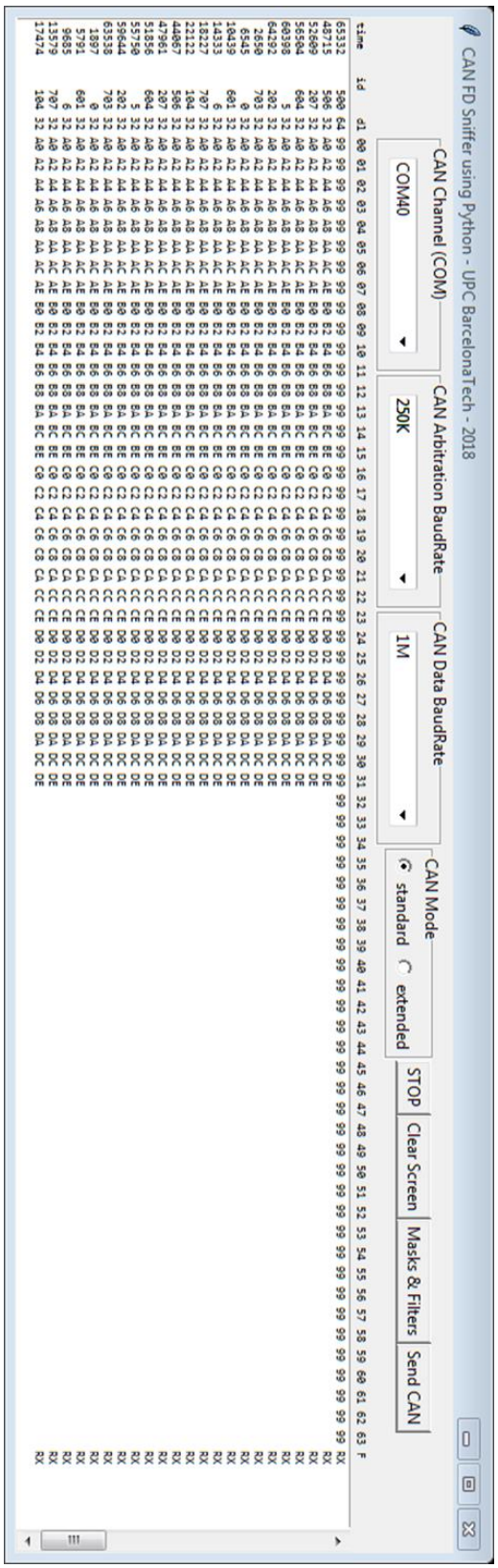


Figure 38. Test 5 results.

6.2.3.6. Test 6

Table 25 shows the variable values that will be used for the Test 6.

Variable	Value
DLC	48
Txd[i]	Random
Tx FIFO size	Random
Rx FIFO size	Random
IDE	0x402
Mask	None
Filter	None
Arbitration bit time	250 kbit/s
Data bit time	2 Mbit/s

Table 25. Test 6 variable values. Source: Own.

Results:

In Figure 39 it can be seen that the message sent through the real CAN FD network is the expected. Therefore, the system works properly.

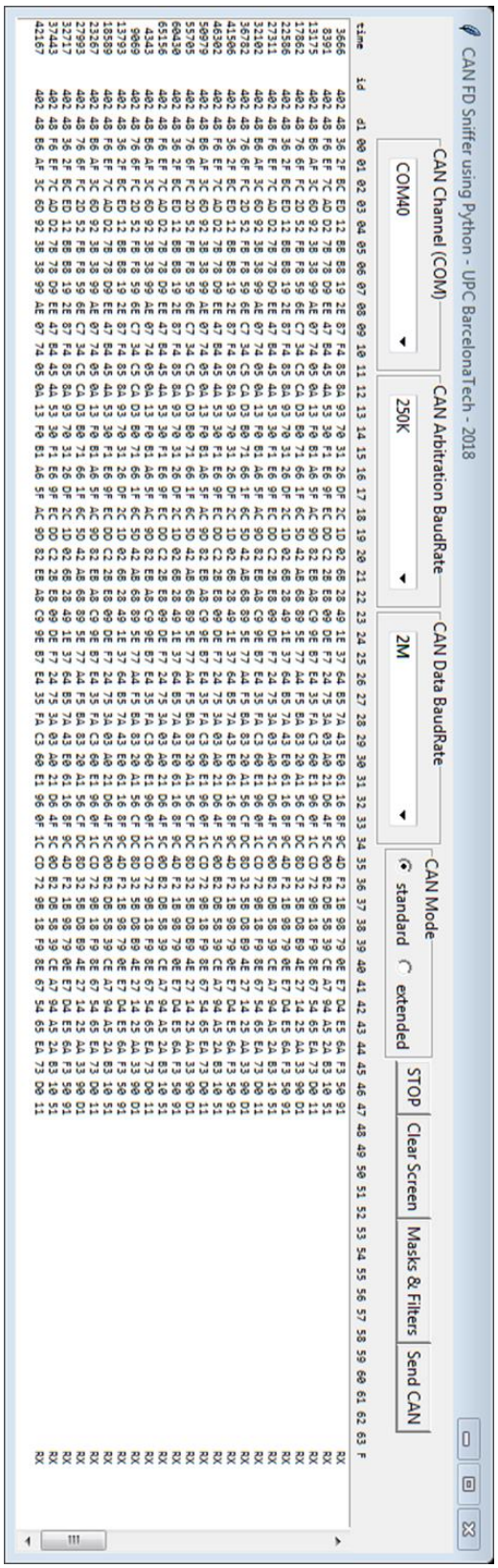


Figure 39. Test 6 results.

6.2.3.7. Test 7

Table 26 shows the variable values that will be used for the Test 7.

Variable	Value
DLC	64
Txd[i]	Figure 40
Tx FIFO size	8
Rx FIFO size	8
IDE	0x500
Mask	None
Filter	None
Arbitration bit time	250 kbit/s
Data bit time	4 Mbit/s

Table 26. Test 7 variable values. Source: Own.

Message sent:

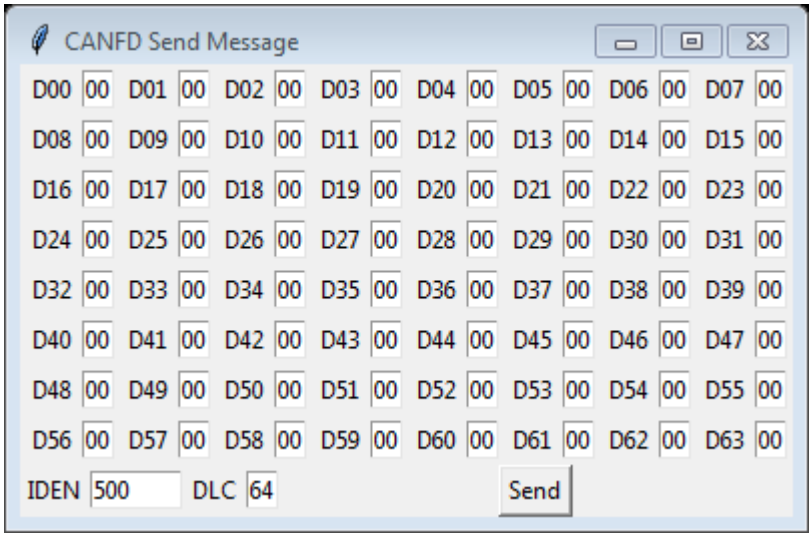


Figure 40. Message sent with the CAN FD sniffer. Source: Own.

Results:

In Figures 41 and 42 it can be seen that the message received through the real CAN FD network is the expected. Therefore, the system works properly.

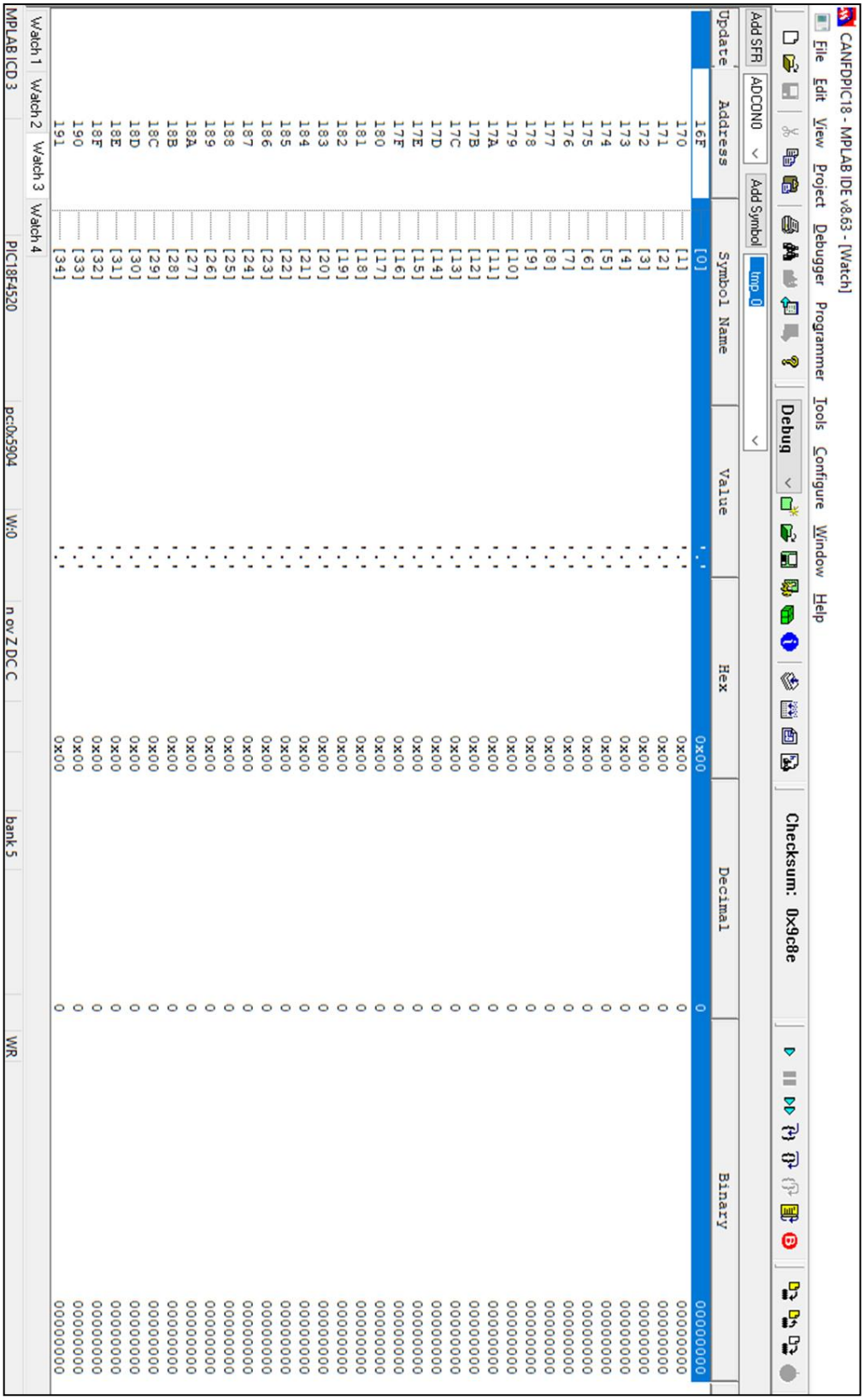


Figure 41. Test 7 results.

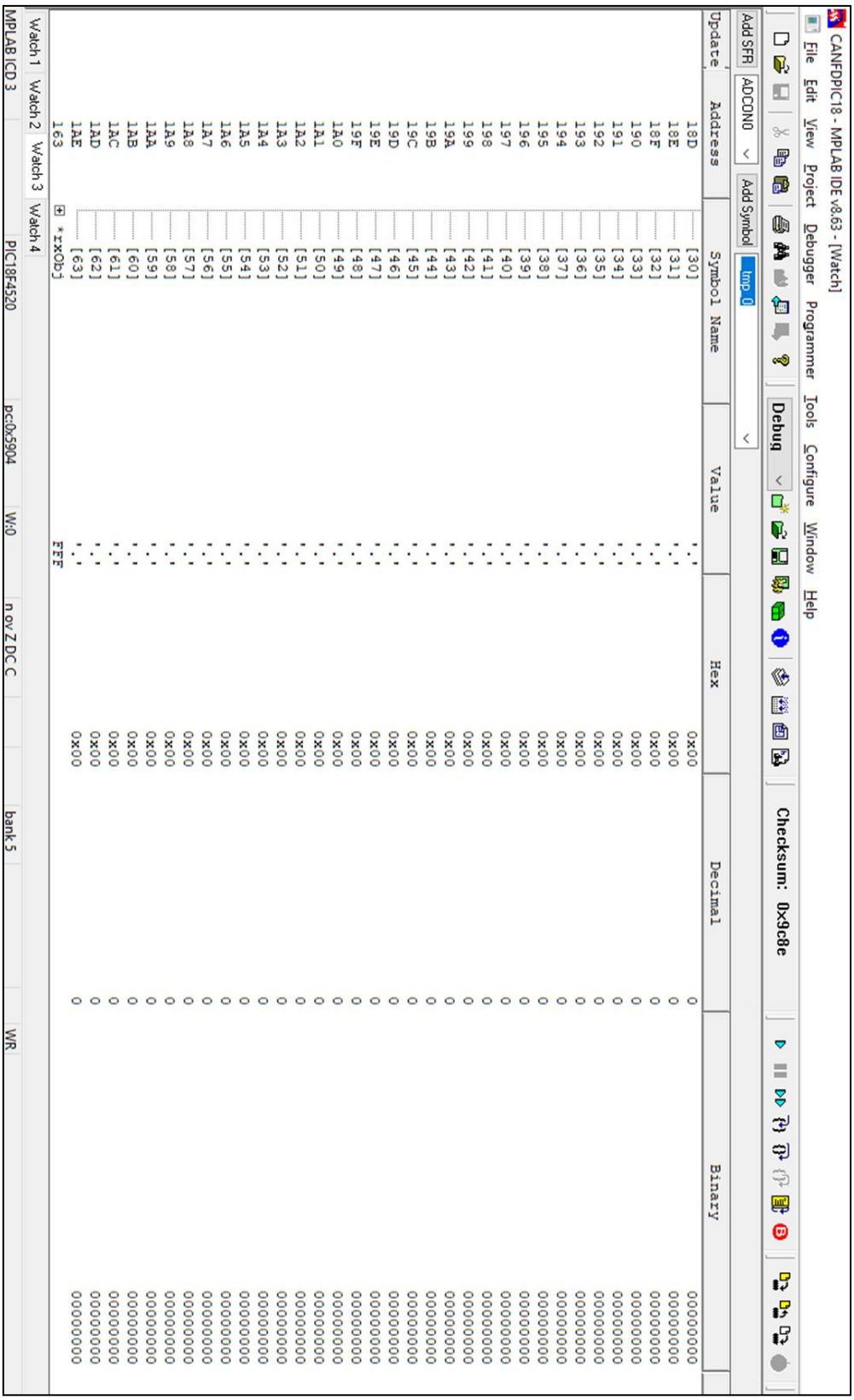


Figure 42. Test 7 results.

6.2.3.8. Test 8

Table 27 shows the variable values that will be used for the Test 8.

Variable	Value
DLC	16
Txd[i]	Figure 43
Tx FIFO size	1
Rx FIFO size	1
IDE	0x500
Mask	None
Filter	None
Arbitration bit time	250 kbit/s
Data bit time	2 Mbit/s

Table 27. Test 8 variable values.

Message sent:

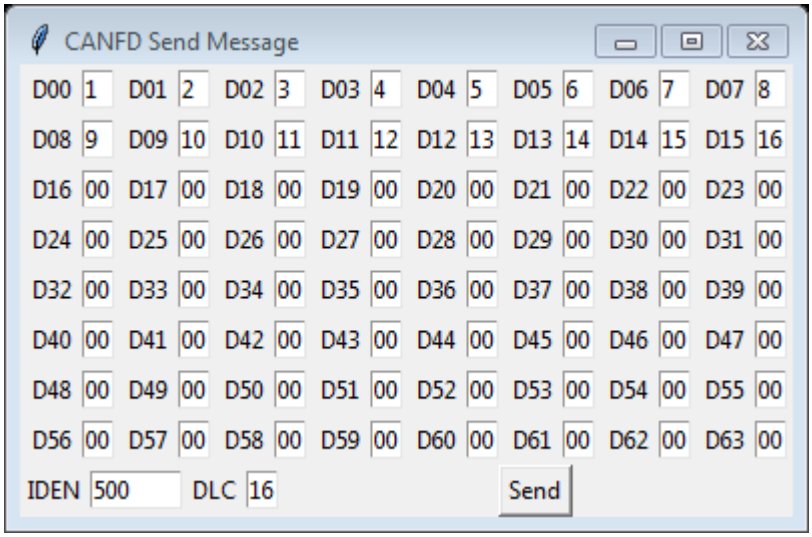


Figure 43. Message sent with the CAN FD sniffer. Source: Own.

Results:

In Figure 44 it can be seen that the message received through the real CAN FD network is the expected. Therefore, the system works properly.

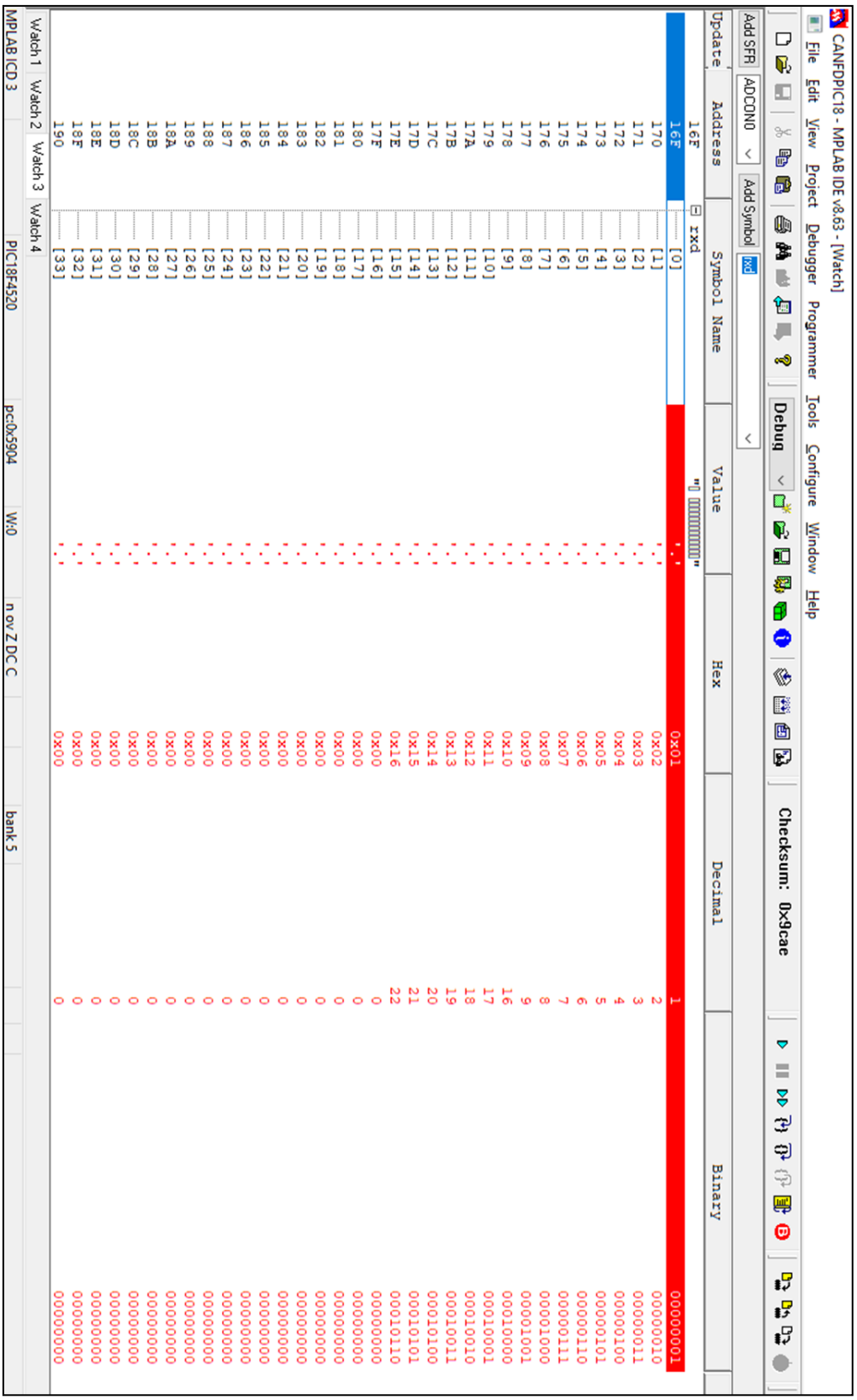


Figure 44. Test 8 results.

6.2.3.9. Test 9

Table 28 shows the variable values that will be used for the Test 9.

Variable	Value
DLC	32
Txd[i]	Figure 45
Tx FIFO size	Random
Rx FIFO size	Random
IDE	0x300
Mask	None
Filter	None
Arbitration bit time	250 kbit/s
Data bit time	4 Mbit/s

Table 28. Test 9 variable values. Source: Own.

Message sent:

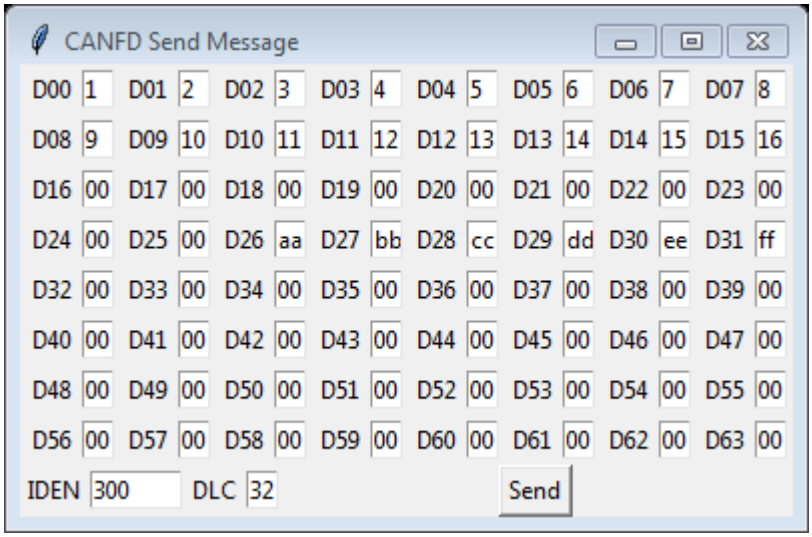


Figure 45. Message sent with the CAN FD sniffer. Source: Own.

Results:

In Figure 46 it can be seen that the message received through the real CAN FD network is the expected. Therefore, the system works properly.

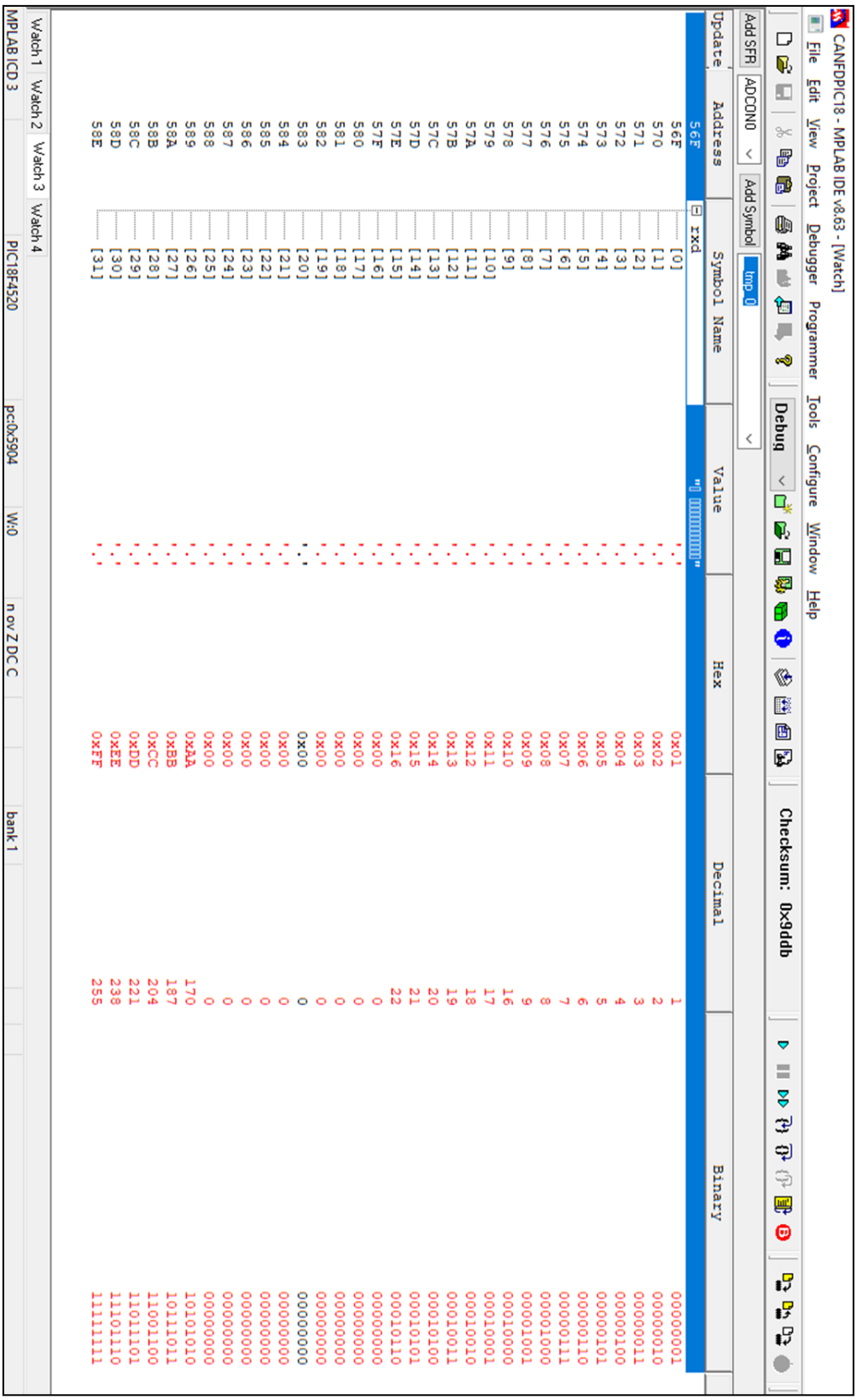


Figure 46. Test 9 results.

6.2.3.10. Test 10

Table 29 shows the variable values that will be used for the Test 10.

Variable	Value
DLC	32
Txd[i]	Figure 47
Tx FIFO size	Random
Rx FIFO size	Random
IDE	0x500
Mask	None
Filter	None
Arbitration bit time	500 kbit/s
Data bit time	2 Mbit/s

Table 29. Test 10 variable values. Source: Own.

Message sent:

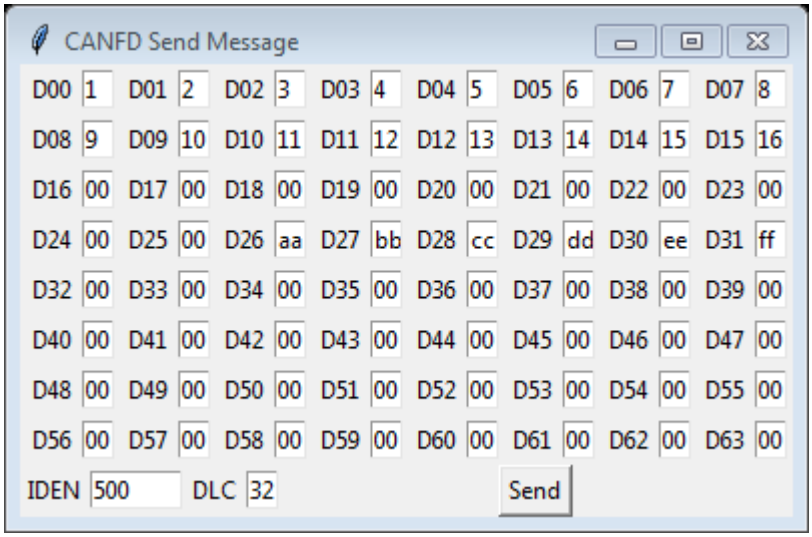


Figure 47. Message sent with the CAN FD sniffer. Source: Own.

Results:

In Figure 48 it can be seen that the message received through the real CAN FD network is the expected. Therefore, the system works properly.

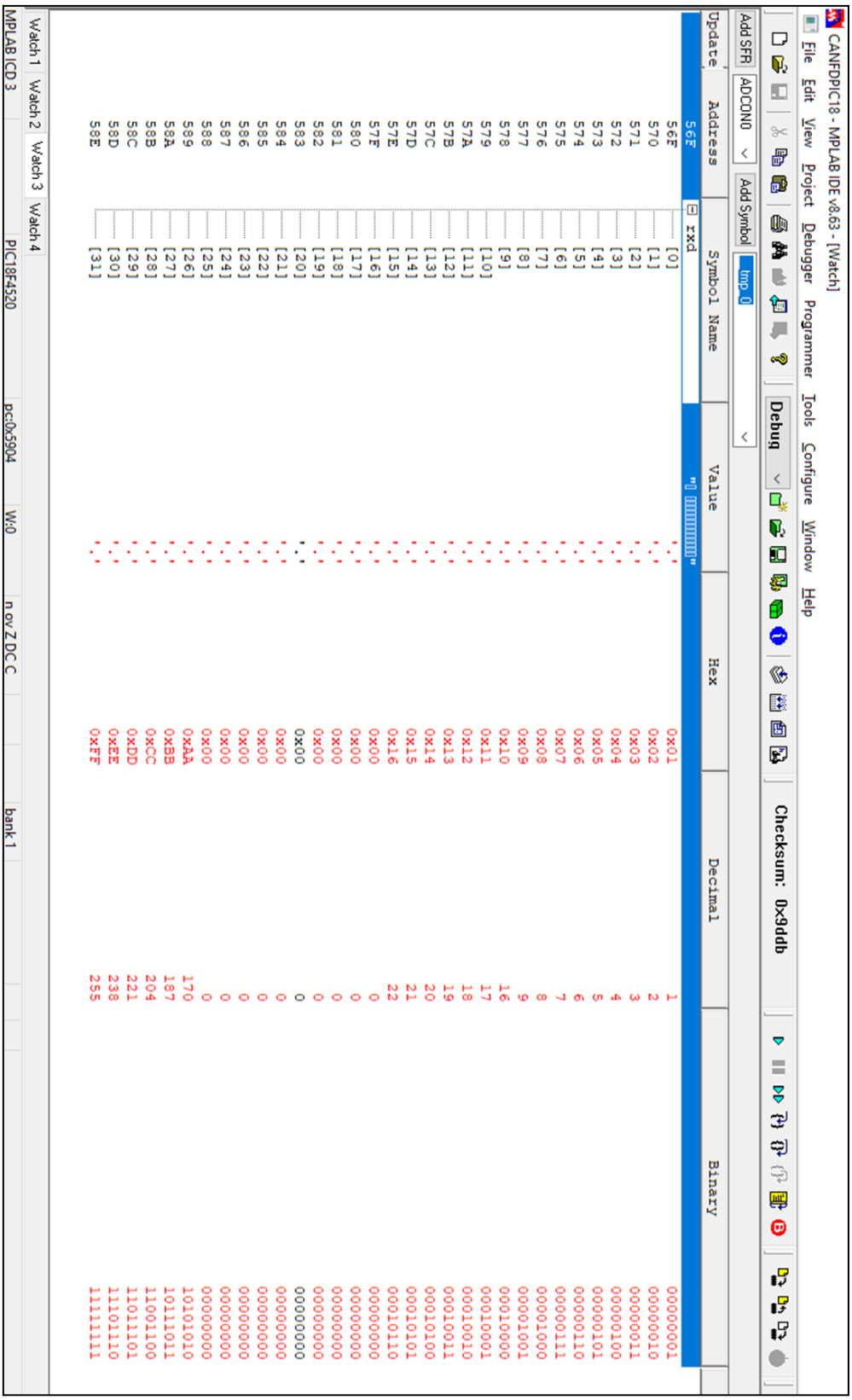


Figure 48. Test 10 results.

6.2.3.11. Test 11

Table 30 shows the variable values that will be used for the Test 11.

Variable	Value
DLC	64
Txd[i]	Figure 49
Tx FIFO size	16
Rx FIFO size	16
IDE	0x400
Mask	None
Filter	None
Arbitration bit time	250 kbit/s
Data bit time	2 Mbit/s

Table 30. Test 11 variable values. Source: Own.

Message sent:

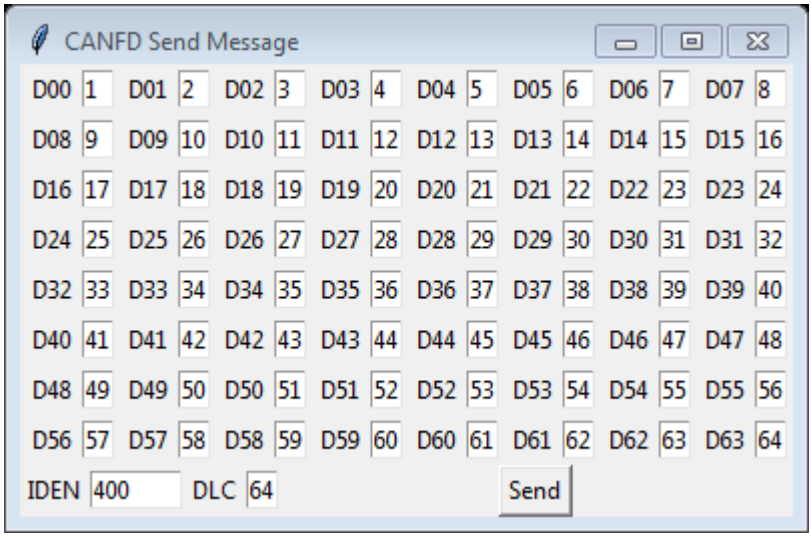


Figure 49. Message sent with the CAN FD sniffer. Source: Own.

Results:

In Figures 50 and 51 it can be seen that the message received through the real CAN FD network is the expected. Therefore, the system works properly.

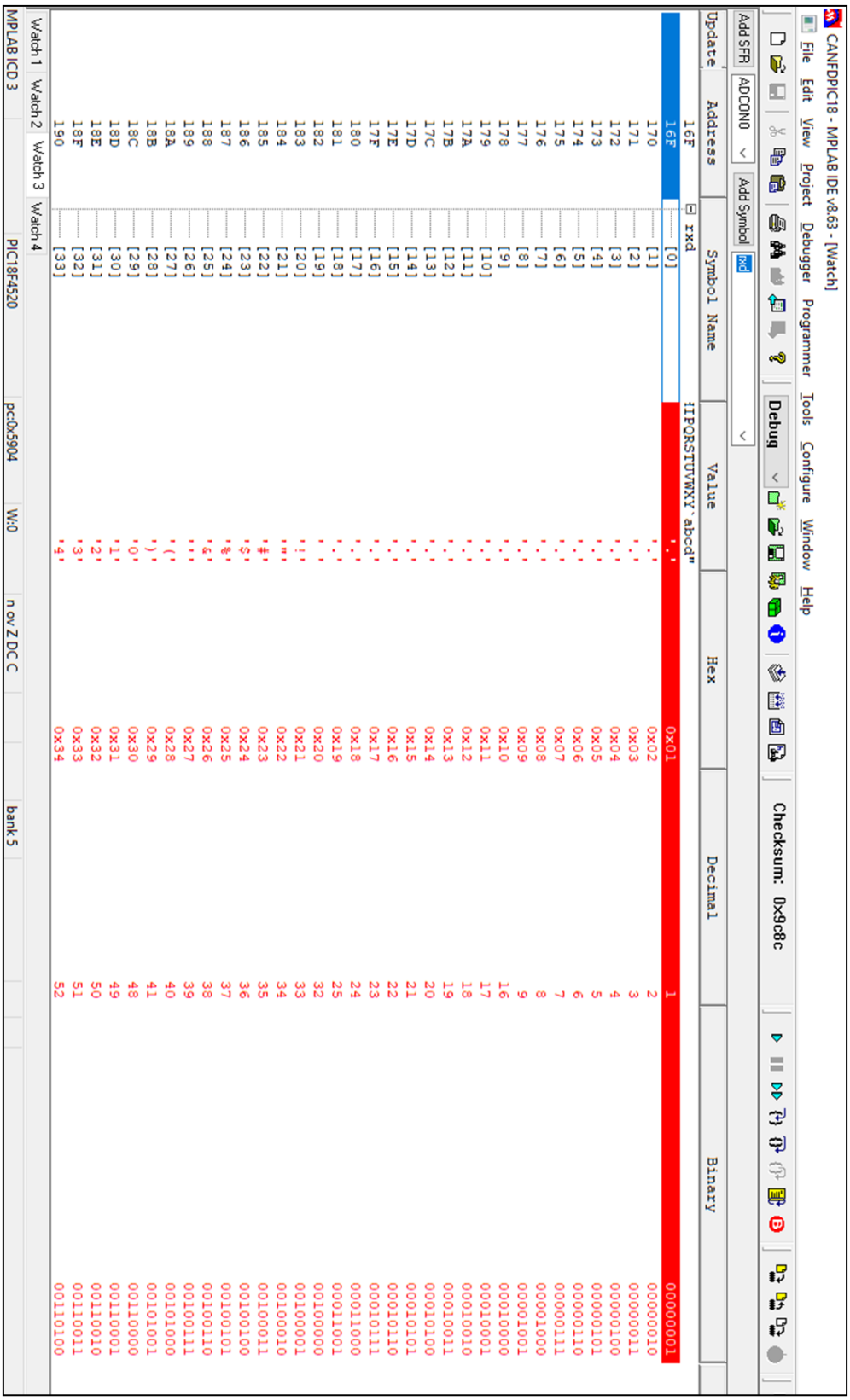


Figure 50. Test 11 results.

Figure 51. Test 11 results.

6.2.3.12. Test 12

Table 31 shows the variable values that will be used for the Test 12.

Variable	Value
DLC	64
Txd[i]	Figure 51
Tx FIFO size	16
Rx FIFO size	16
IDE	0x400
Mask	None
Filter	0x300
Arbitration bit time	250 kbit/s
Data bit time	2 Mbit/s

Table 31. Test 12 variable values. Source: Own.

Message sent:

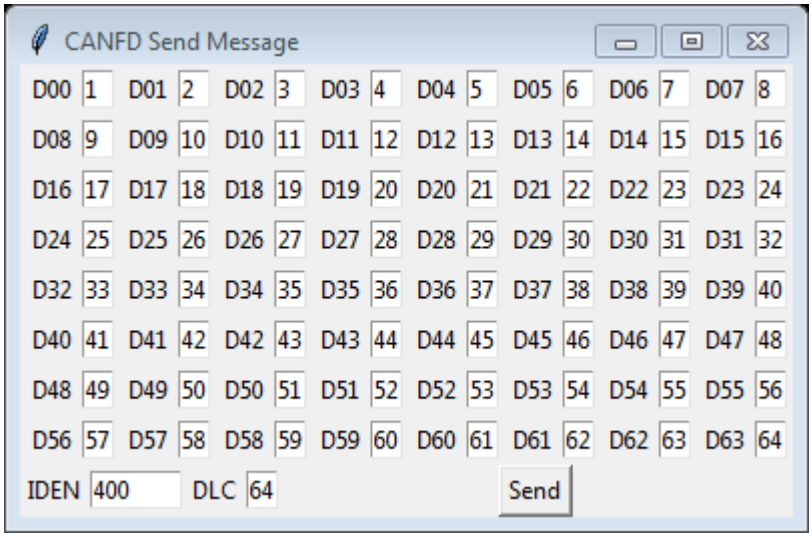


Figure 51. Message sent with the CAN FD sniffer. Source: Own.

Results:

As expected, as the filter only allow the messages with the 0x300 identifier, there are no messages received. Therefore, the system works properly.

6.2.3.13. Test 13

Table 31 shows the variable values that will be used for the Test 13.

Variable	Value
DLC	64
Txd[i]	Figure 52
Tx FIFO size	16
Rx FIFO size	16
IDE	0x400
Mask	0xff
Filter	Random
Arbitration bit time	250 kbit/s
Data bit time	2 Mbit/s

Table 31. Test 13 variable values. Source: Own.

Message sent:

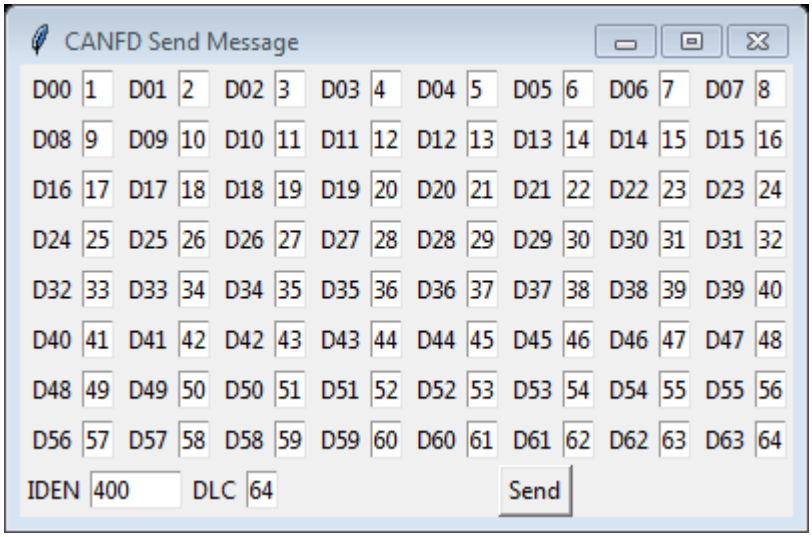


Figure 52. Message sent with the CAN FD sniffer. Source: Own.

Results:

As expected, with the set mask, no messages get through, there are no messages received. Therefore, the system works properly.

Conclusions and future work

As a conclusion it can be said that the main objective of the project has been achieved. It has been successfully developed a CAN FD node that can transmit - send and receive - data through a real CAN network.

Nevertheless, the node can be improved as this is the first version that only performs the most basic functions. In order to improve the project, and as a future work, some features of the CAN FD node based on a PIC18 microcontroller could be improved, which are the following:

- In the first place in this project - as commented in section 3.2 - it has only been used the data transmission through transmit and receive FIFOs. One point to improve, then, would be to use also the Transmit Event FIFOs (TEF) and the Transmit and Receive Queue (TxQ & RxQ), which would make the system more thorough [12].

- In the second place it could be interesting to add interrupts, as the CAN protocol is known by its error-detecting reliability and this would improve this feature.

- Also, it is possible with the used devices to introduce an error-detecting code called Cyclic Redundancy Check (CRC). This code reassures that the system functions properly and that there are no errors in the data transmission.

- Finally, in this project only two modes have been mentioned - CAN Normal mode and CAN LoopBack mode-. The MCP2517FD click allow several other modes, therefore implementing such modes could make the CAN FD node more efficient.

Briefly, the main objective has been successfully accomplished and those derived objectives that, even though are not that visible, have a huge impact to the project - such as understanding the CAN protocol, the insights of the MPLAB or coding in C - have also been surpassed.

Thanks

I would like to thank the great support and guidance that the supervisor of the project Manuel Moreno Eguílaz has given me. He has been always available to help me get through any issue that I found.

I would also like to thank all the professors that I have had during the degree, as they taught me the knowledge that I required to develop this project.

Finally, I also would like to thank my family and friends to offer me the support needed to get through the degree, without them it would have been way more difficult and less enjoyable.

Bibliography

Bibliographic references

[1] DEAR BORN GROUP, INC. CAN: Controller Area Network, Introduction and primer. 27077 Hills Tech Court, Farmington Hills, September 2004. [https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&ved=2ahUKEwjS3pW2ovHeAhVIAcAKHawaBkQQFjAHegQIBxAC&url=https%3A%2F%2Fspaces.usu.edu%2Fdownload%2Fattachments%2F53053449%2Fprimer.pdf%3Fversion%3D1%26modificationDate%3D1366417485000%26api%3Dv2&usg=AOvVaw3RQkvICuqs-0DTs59QCYdB, 13th February 2019].

[2] ROBERT BOSCH Gmbh: CAN FD Specification version 1.0. Gerlingen, Germany, April 2012. [file:///C:/Users/Oriol/Downloads/can_fd_spec%20(1).pdf, 23rd February 2019].

[3] TUTORIALS POINT, C programming tutorial [https://www.tutorialspoint.com/cprogramming/index.htm, 20th February 2019].

[4] PICOTECH, CAN and CAN FD bus decoding. [https://www.picotech.com/library/oscilloscopes/can-bus-serial-protocol-decoding, 23rd May 2019].

[5] WIKIPEDIA COMMONS, CAN-Frame mit Pegeln mit Stuffbits. [https://commons.wikimedia.org/wiki/File:CAN-Frame_mit_Pegeln_mit_Stuffbits.svg, 12th April 2019]

[6] CAN IN AUTOMATION, CAN FD, the basic idea. [https://www.can-cia.org/can-knowledge/can/can-fd/, 12th April 2019].

[7] KENT LENNARTSSON – KVASER, Comparing CAN FD with Classical CAN. [https://www.kvaser.com/wp-content/uploads/2016/10/comparing-can-fd-with-classical-can.pdf, 13th February 2019].

[8] MICROCHIP TECHNOLOGY INC, MCP25XXFD Reference Manual [http://ww1.microchip.com/downloads/en/DeviceDoc/MCP25XXFD-FRM,-CAN-FD-Controller-Module-DS20005678D.pdf, 17th April 2019].

[9] MIKROE EMBEDDED TOOLS, MCP2517FD Click. [https://www.mikroe.com/mcp2517fd-click, 10th June 2019]

[10] INTERNATIONAL STANDARD, ISO 11898-1 Road Vehicles [https://www.sis.se/api/document/preview/919965/, 10th June 2019].

[11] ATMEL, ATA6560/ATA6561 Datasheet. [https://download.mikroe.com/documents/datasheets/ata6563-datasheet.pdf, 10th June 2019].

[12] MICROCHIP, External CAN FD Controller with SPI Interface Datasheet.[http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD-External-CAN-FD-Controller-with-SPI-Interface-20005688B.pdf, 13th February 2019].

[13] MICROCHIP, PIC18F4520 Datasheet. [https://ww1.microchip.com/downloads/en/devicedoc/39631a.pdf. 13th February 2019].

[14] MICROCHIP, MPLAB X Integrated Development Environment. [https://www.microchip.com/mplab/mplab-x-ide, 20th March 2019].

[15] MICROCHIP, MPLAB ICD 3 In-Circuit Debugger. [https://www.microchip.com/developmenttools/ProductDetails/dv164035, 18th June 2019].

[16] WAVE SHARE, OPEN18F4520 Standard PIC Development Board Datasheet. [https://www.waveshare.com/wiki/Microchip_Datasheets#PIC18F4_Series, 18th June 2019].

[17] MICROCHIP, Firmware Drivers. [https://www.microchip.com/wwwproducts/en/MCP2517FD, 13th February 2019].

[18] MICROCHIP, MPLAB C18 C Compiler Libraries. [http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB_C18_Libraries_51297c.pdf, 28TH March 2019].

Annex

A.1. API FUNCTIONS

DRV_CANFDSPI_Reset

Resets internal registers to default state.

Sintax

int8_t DRV_CANFDSPI_Reset(CANFDSPI_MODULE_ID index)

Parameters

0

Return values

0

Precondition

None.

Side effects

Selects Configuration mode.

Exemple

DRV_CANFDSPI_Reset(0)

DRV_CANFDSPI_ReadByte

Reads one byte from SFR address.

Sintax

int8_t DRV_CANFDSPI_ReadByte(CANFDSPI_MODULE_ID index, uint16_t address, uint8_t *rxd)

Parameters

Index = 0

Address = Any readable/writable SFR. For example, 0x010

*rxd = Any variable. For example, &value

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReadByte(0, address, &value)

DRV_CANFDSPI_WriteByte

Writes one byte to SFR address.

Sintax

int8_t DRV_CANFDSPI_WriteByte(CANFDSPI_MODULE_ID index, uint16_t address, uint8_t txd)

Parameters

Index = 0

Address = Any readable/writable SFR. For example, 0x010

txd = Any byte. For example, 0x5A.

Return values

0

Precondition

Must be in Configuration mode.

Side effects

None.

Exemple

DRV_CANFDSPI_WriteByte(0, 0x010, 0x55)

DRV_CANFDSPI_ReadHalfWord

Reads two bytes from SFR address.

Sintax

int8_t DRV_CANFDSPI_ReadHalfWord(CANFDSPI_MODULE_ID index, uint16_t address, uint16_t *rxd)

Parameters

Index = 0

Address = Any readable/writable SFR. For example, 0x010

*rxd = Any variable. For example, &value

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReadHalfWord(0, address, &value)

DRV_CANFDSPI_WriteHalfWord

Writes two bytes to SFR address.

Sintax

int8_t DRV_CANFDSPI_WriteHalfWord(CANFDSPI_MODULE_ID index, uint16_t address, uint16_t txd)

Parameters

Index = 0

Address = Any readable/writable SFR. For example, 0x010

txd = Any pair of bytes. For example, 0x5A5A.

Return values

0

Precondition

Must be in Configuration mode.

Side effects

None.

Exemple

DRV_CANFDSPI_WriteHalfWord(0, address, 0x5A5A)

DRV_CANFDSPI_ReadWord

Reads four bytes from SFR/RAM address.

Sintax

int8_t DRV_CANFDSPI_ReadWord(CANFDSPI_MODULE_ID index, uint16_t address, uint32_t *rxd)

Parameters

Index = 0

Address = Any readable/writable SFR. For example, 0x010. Any RAM address. For example, 0x400.

*rxd = Any variable. For example, &value

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReadWord(0, address, &value)

DRV_CANFDSPI_WriteWord

Writes four bytes to RAM address.

Sintax

int8_t DRV_CANFDSPI_WriteWord(CANFDSPI_MODULE_ID index, uint16_t address, uint32_t txd)

Parameters

Index = 0

Address = Any readable/writable SFR. For example, 0x010. Any RAM address. For example, 0x400.

txd = Any group of four bytes. For example, 0x5A5A5A5A.

Return values

0

Precondition

Must be in Configuration mode.

Side effects

None.

Exemple

DRV_CANFDSPI_WriteHalfWord(0, address, 0x5A5A)

DRV_CANFDSPI_ReadByteArray

Reads an array of bytes from RAM address.

Sintax

int8_t DRV_CANFDSPI_ReadByteArray(CANFDSPI_MODULE_ID index, uint16_t address, uint8_t *rxd, uint16_t nBytes);

Parameters

Index = 0

Address = Any readable/writable RAM. For example, 0x400

*rxd = Any variable. For example, iprx.

nBytes = number of bytes to read.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReadByteArray(0, address, iprx, 4)

DRV_CANFDSPI_WriteByteArray

Writes an array of bytes to RAM address.

Sintax

int8_t DRV_CANFDSPI_WriteByteArray(CANFDSPI_MODULE_ID index, uint16_t address, uint8_t *txd, uint16_t nBytes);

Parameters

Index = 0

Address = Any readable/writable RAM. For example, 0x400

*txd = Array minimum of four bytes. For example ip[4] = {0x01,0x44,0x33,0x55}

nBytes = number of bytes to write. Must be multiple of four.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_WriteByteArray(0, address, ip, 4);

DRV_CANFDSPI_ReadWordArray

Reads an array of words (four bytes) from RAM address.

Sintax

int8_t DRV_CANFDSPI_ReadWordArray(CANFDSPI_MODULE_ID index, uint16_t address, uint32_t *rxd, uint16_t nWords);

Parameters

Index = 0

Address = Any readable/writable RAM. For example, 0x400

*rxd = Any variable. For example, iprx.

nWords = number of words to read.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReadWordArray(0, address, iprx, 1);

DRV_CANFDSPI_WriteWordArray

Writes an array of words to RAM address.

Sintax

int8_t DRV_CANFDSPI_WriteWordArray(CANFDSPI_MODULE_ID index, uint16_t address,

uint32_t *txd, uint16_t nWords);

Parameters

Index = 0

Address = Any readable/writable RAM. For example, 0x400

*rxd = Any array of words. For example ip[3] = {0x01010101,0x44444444,0x33333333}

nWords = number of words to write.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_WriteWordArray(0, address, ip, 1);

DRV_CANFDSPI_Configure

Can control register configuration.

Sintax

int8_t DRV_CANFDSPI_Configure(CANFDSPI_MODULE_ID index, CAN_CONFIG* config)

Parameters

index = 0

config = &CAN_CONFIG. Calls the values of CAN_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

DRV_CANFDSPI_Configure(0, &config);

DRV_CANFDSPI_ConfigureObjectReset

Resets Configure Object to reset values.

Sintax

int8_t DRV_CANFDSPI_ConfigureObjectReset(CAN_CONFIG* config);

Parameters

config = &CAN_CONFIG. Calls the values of CAN_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

CAN_CONFIG config;

DRV_CANFDSPI_ConfigureObjectReset(&config);

DRV_CANFDSPI_OperationModeSelect

Select Operation Mode.

Sintax

int8_t DRV_CANFDSPI_OperationModeSelect(CANFDSPI_MODULE_ID index, CAN_OPERATION_MODE opMode);

Parameters

index = 0

opMode = One of the following: CAN_NORMAL_MODE, CAN_SLEEP_MODE, CAN_INTERNAL_LOOPBACK_MODE, CAN_LISTEN_ONLY_MODE, CAN_CONFIGURATION_MODE, CAN_EXTERNAL_LOOPBACK_MODE, CAN_CLASSIC_MODE, CAN_RESTRICTED_MODE, CAN_INVALID_MODE.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

// Select Normal Mode

DRV_CANFDSPI_OperationModeSelect(0, CAN_NORMAL_MODE);

DRV_CANFDSPI_ModuleEventEnable

Enables interrupts

Sintax

int8_t DRV_CANFDSPI_ModuleEventDisable(CANFDSPI_MODULE_ID index, CAN_MODULE_EVENT flags);

Parameters

index = 0

flags = One, or more of the following: CAN_NO_EVENT, CAN_ALL_EVENTS, CAN_TX_EVENT, CAN_RX_EVENT, CAN_TIME_BASE_COUNTER_EVENT, CAN_OPERATION_MODE_CHANGE_EVENT, CAN_TEF_EVENT, CAN_RAM_ECC_EVENT, CAN_SPI_CRC_EVENT, CAN_TX_ATTEMPTS_EVENT, CAN_RX_OVERFLOW_EVENT, CAN_SYSTEM_ERROR_EVENT, CAN_BUS_ERROR_EVENT, CAN_BUS_WAKEUP_EVENT, CAN_RX_INVALID_MESSAGE_EVENT.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ModuleEventDisable(DRV_CANFDSPI_INDEX_0, CAN_TX_EVENT | CAN_RX_EVENT);

DRV_CANFDSPI_ModuleEventDisable

Disables interrupts.

Sintax

int8_t DRV_CANFDSPI_ModuleEventEnable(CANFDSPI_MODULE_ID index, CAN_MODULE_EVENT flags);

Parameters

index = 0

flags = One, or more of the following: CAN_NO_EVENT, CAN_ALL_EVENTS, CAN_TX_EVENT, CAN_RX_EVENT, CAN_TIME_BASE_COUNTER_EVENT, CAN_OPERATION_MODE_CHANGE_EVENT, CAN_TEF_EVENT, CAN_RAM_ECC_EVENT, CAN_SPI_CRC_EVENT, CAN_TX_ATTEMPTS_EVENT, CAN_RX_OVERFLOW_EVENT, CAN_SYSTEM_ERROR_EVENT, CAN_BUS_ERROR_EVENT, CAN_BUS_WAKEUP_EVENT, CAN_RX_INVALID_MESSAGE_EVENT.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ModuleEventEnable(DRV_CANFDSPI_INDEX_0, CAN_TX_EVENT | CAN_RX_EVENT);

DRV_CANFDSPI_GpioModeConfigure

Initialize GPIO Mode.

Sintax

int8_t DRV_CANFDSPI_GpioModeConfigure(CANFDSPI_MODULE_ID index,
GPIO_PIN_MODE gpio0, GPIO_PIN_MODE gpio1);

Parameters

Index = 0

gpio0 = One of the following: GPIO_MODE_INT, GPIO_MODE_GPIO

gpio1 = One of the following: GPIO_MODE_INT, GPIO_MODE_GPIO

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

//Input/Output configuration

DRV_CANFDSPI_GpioModeConfigure(0, GPIO_MODE_INT, GPIO_MODE_INT);

DRV_CANFDSPI_ModuleEventClear

Clears interrupt Flags.

Sintax

int8_t DRV_CANFDSPI_ModuleEventClear(CANFDSPI_MODULE_ID index, CAN_MODULE_EVENT flags);

Parameters

Index = 0

flags = One, or more of the following: CAN_NO_EVENT, CAN_ALL_EVENTS, CAN_TX_EVENT, CAN_RX_EVENT, CAN_TIME_BASE_COUNTER_EVENT, CAN_OPERATION_MODE_CHANGE_EVENT, CAN_TEF_EVENT, CAN_RAM_ECC_EVENT, CAN_SPI_CRC_EVENT, CAN_TX_ATTEMPTS_EVENT, CAN_RX_OVERFLOW_EVENT, CAN_SYSTEM_ERROR_EVENT, CAN_BUS_ERROR_EVENT, CAN_BUS_WAKEUP_EVENT, CAN_RX_INVALID_MESSAGE_EVENT.

Return values

0

Precondition

None.

Side effects

None.

Exemple

//Clear main interrupts

DRV_CANFDSPI_ModuleEventClear(0, CAN_ALL_EVENTS);

DRV_CANFDSPI_ModuleEventTxCodeGet

Get TX code.

Sintax

int8_t DRV_CANFDSPI_ModuleEventTxCodeGet(CANFDSPI_MODULE_ID index, CAN_TXCODE* txCode);

Parameters

Index = 0

txCode = &CAN_TXCODE. Calls the values of CAN_TXCODE previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_TXCODE txCode

DRV_CANFDSPI_ModuleEventTxCodeGet(0, &txCode);

DRV_CANFDSPI_ModuleEventRxCodeGet

Get RX code.

Sintax

int8_t DRV_CANFDSPI_ModuleEventRxCodeGet(CANFDSPI_MODULE_ID index, CAN_RXCODE* rxCode);

Parameters

Index = 0

rxCode = &CAN_RXCODE. Calls the values of CAN_RXCODE previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_RXCODE rxCode

DRV_CANFDSPI_ModuleEventRxCodeGet(0, &rxCode);

DRV_CANFDSPI_ModuleEventFilterHitGet

Get Filter Hit.

Sintax

int8_t DRV_CANFDSPI_ModuleEventFilterHitGet(CANFDSPI_MODULE_ID index, CAN_FILTER* filterHit);

Parameters

Index = 0

filterHit = &CAN_FILTER. Calls the values of CAN_FILTER previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_FILTER filterHit

DRV_CANFDSPI_ModuleEventFilterHitGet(0, &filterHit);

DRV_CANFDSPI_ModuleEventIcodeGet

Get ICODE.

Sintax

int8_t DRV_CANFDSPI_ModuleEventIcodeGet(CANFDSPI_MODULE_ID index, CAN_ICODE* icode);

Parameters

Index = 0

icode = &CAN_ICODE. Calls the values of CAN_ICODE previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ModuleEventIcodeGet(0, &icode);

DRV_CANFDSPI_ TransmitChannelLoad

TX Channel Load. Loads data into Transmit channel. Requests transmission, if flush==true.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelLoad(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_TX_MSGOBJ* txObj, uint8_t *txd, uint32_t txdNumBytes, bool flush);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

txObj = &CAN_TX_MSGOBJ. Calls the values of CAN_TX_MSGOBJ previously set.

txd = Any variable. For example, &txd.

txdNumBytes = n: Any given number between 0 and 64.

flush = True or False. True sets the flush that ensures that all messages from the FIFO in case a message is appended to a FIFO while it is already transmitting.

Return values

0

Precondition

None.

Side effects

None.

Exemple

txd[MAX_DATA_BYTES];

CAN_TX_MSGOBJ txObj

DRV_CANFDSPI_TransmitChannelLoad(0,APP_RX_FIFO,APP_TX_FIFO, &txObj, txd, n, 0);

DRV_CANFDSPI_ TransmitChannelUpdate

Transmit FIFO Update. Sets UINC of the transmit channel. Keeps TXREQ unchanged.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelUpdate(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, bool flush);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flush = True or False

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_TransmitChannelUpdate(0, CAN_FIFO_CH1, True);

DRV_CANFDSPI_ TransmitChannelEventEnable

Transmit FIFO Event Enable. Enables Transmit FIFO interrupts.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelEventEnable(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_TX_FIFO_EVENT flags);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flags = One of the following: CAN_TX_FIFO_NO_EVENT = 0, CAN_TX_FIFO_ALL_EVENTS = 0x17, CAN_TX_FIFO_NOT_FULL_EVENT = 0x01, CAN_TX_FIFO_HALF_FULL_EVENT = 0x02, CAN_TX_FIFO_EMPTY_EVENT = 0x04,CAN_TX_FIFO_ATTEMPTS_EXHAUSTED_EVENT = 0x10

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_TransmitChannelEventEnable(0, CAN_FIFO_CHN2, CAN_TX_FIFO_NOT_FULL_EVENT);

DRV_CANFDSPI_ TransmitChannelEventEnable

Transmit FIFO Event Enable. Enables Transmit FIFO interrupts.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelEventEnable(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_TX_FIFO_EVENT flags);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flags = One of the following: CAN_TX_FIFO_NO_EVENT = 0, CAN_TX_FIFO_ALL_EVENTS = 0x17, CAN_TX_FIFO_NOT_FULL_EVENT = 0x01, CAN_TX_FIFO_HALF_FULL_EVENT = 0x02, CAN_TX_FIFO_EMPTY_EVENT = 0x04,CAN_TX_FIFO_ATTEMPTS_EXHAUSTED_EVENT = 0x10

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_TransmitChannelEventEnable(0, CAN_FIFO_CHN2, CAN_TX_FIFO_NOT_FULL_EVENT);

DRV_CANFDSPI_ TransmitChannelEventGet

Transmit FIFO Event Get. Reads Transmit FIFO interrupt Flags.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelEventGet(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_TX_FIFO_EVENT* flags);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flags = &CAN_TX_FIFO_EVENT. Calls the values of CAN_TX_FIFO_EVENT previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_TX_FIFO_EVENT txFlags;

DRV_CANFDSPI_TransmitChannelEventGet(0, CAN_FIFO_CHN2, &txFlags);

DRV_CANFDSPI_ TransmitChannelEventAttemptClear

Transmit FIFO Event Clear. Clears Transmit FIFO Attempts Exhausted interrupt Flag.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelEventAttemptClear(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_TransmitChannelEventAttemptClear(0, CAN_FIFO_CHN2);

DRV_CANFDSPI_ TransmitChannelConfigure

Configure Transmit FIFO.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelConfigure(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_TX_FIFO_CONFIG* config);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

config = &CAN_TX_FIFO_CONFIG. Calls the values of CAN_TX_FIFO_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

CAN_TX_FIFO_CONFIG txConfig;

DRV_CANFDSPI_TransmitChannelConfigure(0, CAN_FIFO_CHN2, &txConfig);

DRV_CANFDSPI_ TransmitChannelConfigure

Configure Transmit FIFO.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelConfigure(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_TX_FIFO_CONFIG* config);

Parameters

Index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

config = &CAN_TX_FIFO_CONFIG. Calls the values of CAN_TX_FIFO_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

CAN_TX_FIFO_CONFIG txConfig;

DRV_CANFDSPI_TransmitChannelConfigure(0, CAN_FIFO_CHN2, &txConfig);

DRV_CANFDSPI_ TransmitChannelConfigureObjectReset

Reset TransmitChannelConfigure object to reset values.

Sintax

int8_t DRV_CANFDSPI_TransmitChannelConfigureObjectReset(CAN_TX_FIFO_CONFIG* config);

Parameters

config = &CAN_TX_FIFO_CONFIG. Calls the values of CAN_TX_FIFO_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

CAN_TX_FIFO_CONFIG txfConfig

DRV_CANFDSPI_TransmitChannelConfigure(&txfConfig);

DRV_CANFDSPI_ TefUpdate

Transmit Event FIFO Update. Sets UINC of the TEF.

Sintax

int8_t DRV_CANFDSPI_TefUpdate(CANFDSPI_MODULE_ID index);

Parameters

index = 0

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_TefUpdate(0);

DRV_CANFDSPI_ TefEventGet

Reads Transmit Event FIFO interrupt Flags.

Sintax

int8_t DRV_CANFDSPI_TefEventGet(CANFDSPI_MODULE_ID index,
CAN_TEF_FIFO_EVENT* flags);

Parameters

index = 0

flags = &CAN_TEF_FIFO_EVENT. Calls the values of CAN_TEF_FIFO_EVENT previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_TEF_FIFO_EVENT tefFlags;

DRV_CANFDSPI_TefEventGet(DRV_CANFDSPI_INDEX_0, &tefFlags);

DRV_CANFDSPI_ TefEventEnable

Transmit Event FIFO Event Enable. Enables Transmit Event FIFO interrupts.

Sintax

int8_t DRV_CANFDSPI_TefEventEnable(CANFDSPI_MODULE_ID index,
CAN_TEF_FIFO_EVENT flags);

Parameters

index = 0

flags = &CAN_TEF_FIFO_EVENT. Calls the values of CAN_TEF_FIFO_EVENT previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_TEF_FIFO_EVENT tefFlags;

DRV_CANFDSPI_TefEventEnable(0, &tefFlags);

DRV_CANFDSPI_ TefEventDisable

Transmit Event FIFO Event Disable. Disables Transmit Event FIFO interrupts.

Sintax

int8_t DRV_CANFDSPI_TefEventDisable(CANFDSPI_MODULE_ID index,

CAN_TEF_FIFO_EVENT flags);

Parameters

index = 0

flags = &CAN_TEF_FIFO_EVENT. Calls the values of CAN_TEF_FIFO_EVENT previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_TEF_FIFO_EVENT tefFlags;

DRV_CANFDSPI_TefEventDisable(0, &tefFlags);

DRV_CANFDSPI_ TefEventOverflowClear

Transmit Event FIFO Event Clear. Clears Transmit Event FIFO Overflow interrupt Flag.

Sintax

int8_t DRV_CANFDSPI_TefEventOverflowClear(CANFDSPI_MODULE_ID index);

Parameters

index = 0

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_TefEventOverflowClear(0);

DRV_CANFDSPI_ ReceiveMessageGet

Get Received Message. Reads Received message from channel.

Sintax

int8_t DRV_CANFDSPI_ReceiveMessageGet(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_RX_MSGOBJ* rxObj, uint8_t *rxd, uint8_t nBytes);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

rxObj = &CAN_RX_MSGOBJ. Calls the values of CAN_RX_MSGOBJ previously set.

*rxd = Any variable. For example, rxd.

nBytes = number of bytes to get. Usually ‘MAX_DATA_BYTES’.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_RX_MSGOBJ rxObj;

uint8_t rxd[MAX_DATA_BYTES];

DRV_CANFDSPI_ReceiveMessageGet(0, CAN_FIFO_CH1, &rxObj, rxd, MAX_DATA_BYTES);

DRV_CANFDSPI_ReceiveEventGet

Receive FIFO Event Get. Get pending interrupts of all receive FIFOs.

Sintax

int8_t DRV_CANFDSPI_ReceiveEventGet(CANFDSPI_MODULE_ID index, uint32_t* rxif);

Parameters

index = 0

rxif = Any variable. For example, rxif.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReceiveEventGet(0, rxif);

DRV_CANFDSPI_ ReceiveEventOverflowGet

Receive FIFO Event Get. Get pending RXOVIF of all receive FIFOs.

Sintax

int8_t DRV_CANFDSPI_ReceiveEventOverflowGet(CANFDSPI_MODULE_ID index,
uint32_t* rxovif);

Parameters

index = 0

rxovif = Any variable. For example, rxovif.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReceiveEventOverflowGet(0, rxovif);

DRV_CANFDSPI_ ReceiveChannelUpdate

Receive FIFO Update. Sets UINC of the receive channel.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelUpdate(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReceiveChannelUpdate(0, CAN_FIFO_CH1);

DRV_CANFDSPI_ ReceiveChannelEventGet

Receive FIFO Event Get. Reads Receive FIFO interrupt Flags.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelEventGet(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_RX_FIFO_EVENT* flags);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flags = &CAN_RX_FIFO_EVENT. Calls the values of CAN_RX_FIFO_EVENT previously set.

Return values

0

Precondition

None.

Side effects

None.

Exemple

CAN_RX_FIFO_EVENT rxFlags;

DRV_CANFDSPI_ReceiveChannelEventGet(0, CAN_FIFO_CH1, &rxFlags);

DRV_CANFDSPI_ ReceiveChannelEventEnable

Receive FIFO Event Enable. Enables Receive FIFO interrupts.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelEventEnable(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_RX_FIFO_EVENT flags);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flags = One, or more of the following: CAN_RX_FIFO_NO_EVENT, CAN_RX_FIFO_ALL_EVENTS, CAN_RX_FIFO_NOT_EMPTY_EVENT, CAN_RX_FIFO_HALF_FULL_EVENT, CAN_RX_FIFO_FULL_EVENT, CAN_RX_FIFO_OVERFLOW_EVENT.

Return values

0

Precondition

None.

Side effects

None.

Exemple

// Setup Transmit and Receive Interrupts

DRV_CANFDSPI_ReceiveChannelEventEnable(0, CAN_FIFO_CH1, CAN_RX_FIFO_NOT_EMPTY_EVENT);

DRV_CANFDSPI_ ReceiveChannelEventDisable

Receive FIFO Event Disable. Disables Receive FIFO interrupts.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelEventDisable(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_RX_FIFO_EVENT flags);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

flags = One, or more of the following: CAN_RX_FIFO_NO_EVENT, CAN_RX_FIFO_ALL_EVENTS, CAN_RX_FIFO_NOT_EMPTY_EVENT, CAN_RX_FIFO_HALF_FULL_EVENT, CAN_RX_FIFO_FULL_EVENT, CAN_RX_FIFO_OVERFLOW_EVENT.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReceiveChannelEventEnable(0, CAN_FIFO_CH1, CAN_RX_FIFO_NO_EVENT);

DRV_CANFDSPI_ReceiveChannelEventOverflowClear

Receive FIFO Event Clear. Clears Receive FIFO Overflow interrupt Flag.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelEventOverflowClear(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_ReceiveChannelEventOverflowClear(0, CAN_FIFO_CH1);

DRV_CANFDSPI_ ReceiveChannelConfigureObjectReset

Reset ReceiveChannelConfigure object to reset value.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelConfigureObjectReset(CAN_RX_FIFO_CONFIG* config);

Parameters

config = & CAN_RX_FIFO_CONFIG. Calls the values of CAN_RX_FIFO_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

// Setup RX FIFO

CAN_RX_FIFO_CONFIG rxConfig

DRV_CANFDSPI_ReceiveChannelConfigureObjectReset(&rxConfig);

DRV_CANFDSPI_ ReceiveChannelConfigure

Configure Receive FIFO.

Sintax

int8_t DRV_CANFDSPI_ReceiveChannelConfigure(CANFDSPI_MODULE_ID index, CAN_FIFO_CHANNEL channel, CAN_RX_FIFO_CONFIG* config);

Parameters

index = 0

channel = CAN_FIFO_CHN. Being N any number between 0 and 31.

config = & CAN_RX_FIFO_CONFIG. Calls the values of CAN_RX_FIFO_CONFIG previously set.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

// Setup RX FIFO

CAN_RX_FIFO_CONFIG rxConfig

DRV_CANFDSPI_ReceiveChannelConfigure(0, CAN_FIFO_CH1, &rxConfig);

DRV_CANFDSPI_BitTime

Configure Bit Time registers (based on CAN clock speed).

Sintax

int8_t DRV_CANFDSPI_BitTimeConfigure(CANFDSPI_MODULE_ID index, CAN_BITTIME_SETUP bitTime, CAN_SSP_MODE

sspMode, CAN_SYSCLK_SPEED clk);

Parameters

index = 0

bitTime = selectedBitTime. For example, CAN_500K_2M

sspMode = One of the following: CAN_SSP_MODE_OFF, CAN_SSP_MODE_MANUAL, CAN_SSP_MODE_AUTO.

clk = CAN_SYSCLK_40M or CAN_SYSCLK_20M

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

// Setup Bit Time

DRV_CANFDSPI_BitTimeConfigure(DRV_CANFDSPI_INDEX_0, selectedBitTime, CAN_SSP_MODE_AUTO, CAN_SYSCLK_40M);

DRV_CANFDSPI_BitTimeConfigureData10Mhz

Configure Bit Time registers (based on CAN clock speed).

Sintax

int8_t DRV_CANFDSPI_BitTimeConfigureData10MHz(CANFDSPI_MODULE_ID index, CAN_BITTIME_SETUP bitTime, CAN_SSP_MODE sspMode)

Parameters

index = 0

bitTime = selectedBitTime. For example, CAN_500K_2M

sspMode = One of the following: CAN_SSP_MODE_OFF, CAN_SSP_MODE_MANUAL, CAN_SSP_MODE_AUTO.

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

// Setup Bit Time

DRV_CANFDSPI_BitTimeConfigureData10MHz(0, selectedBitTime, CAN_SSP_MODE_AUTO);

DRV_CANFDSPI_BitTimeConfigureNominal20Mhz

Configure Bit Time registers (based on CAN clock speed).

Sintax

int8_t DRV_CANFDSPI_BitTimeConfigureNominal20MHz(CANFDSPI_MODULE_ID index,

CAN_BITTIME_SETUP bitTime)

Parameters

index = 0

bitTime = selectedBitTime. For example, CAN_500K_2M

Return values

0

Precondition

Must be in configuration mode.

Side effects

None.

Exemple

// Setup Bit Time

DRV_CANFDSPI_BitTimeConfigureNominal20MHz(0, selectedBitTime);

DRV_CANFDSPI_EccEnable

Enable ECC.

Sintax

int8_t DRV_CANFDSPI_EccEnable(CANFDSPI_MODULE_ID index);

Parameters

index = 0

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_EccEnable(0);

DRV_CANFDSPI_RamInit

Initialize RAM.

Sintax

int8_t DRV_CANFDSPI_RamInit(CANFDSPI_MODULE_ID index, uint8_t d);

Parameters

index = 0

d = Any 1-byte long value. For example, 0xff.

Return values

0

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_RamInit(0, 0xff);

DRV_CANFDSPI_DlcToDataBytes

DLC to number of actual data bytes conversion.

Sintax

uint32_t DRV_CANFDSPI_DlcToDataBytes(CAN_DLC dlc);

Parameters

dlc = One of the following: 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.

Return values

Given CAN_DLC_N, returns the numerical value N.

Precondition

None.

Side effects

None.

Exemple

DRV_CANFDSPI_DlcToDataBytes(CAN_DLC_64);

DRV_CANFDSPI_DataBytesToDlc

Data bytes to DLC conversion.

Sintax

CAN_DLC DRV_CANFDSPI_DataBytesToDlc(uint8_t n);

Parameters

n = Any number between 0 and 64.

Return values

CAN_DLC_N, being N the closest number to n multiple of four. Also, N must be bigger than n.

Precondition

None.

Side effects

None.

Exemple

CAN_DLC DRV_CANFDSPI_DataBytesToDlc(7);

DRV_CANFDSPI_FilterEnable

Sets the FLTEN bit after changing the filter or mask object.

Sintax

int8_t DRV_CANFDSPI_FilterEnable(CANFDSPI_MODULE_ID index, CAN_FILTER filter);

Parameters

Index = 0

filter = CAN_FILTERN, being N any number between 0 and 31. It usually works with N = 0.

Return values

0

Precondition

None.

Side effects

Initializes the buffer pointer that points to the FIFO where the matching receive message will be stored.

Exemple

DRV_CANFDSPI_FilterDisable(0, CAN_FILTER0);

DRV_CANFDSPI_FilterDisable

Clears the FLTEN bit before changing the filter or mask object.

Sintax

int8_t DRV_CANFDSPI_FilterDisable(CANFDSPI_MODULE_ID index, CAN_FILTER filter);

Parameters

Index = 0

filter = CAN_FILTERN, being N any number between 0 and 31. It usually works with N = 0.

Return values

0

Precondition

None.

Side effects

It allows changing the filter and mask object.

Exemple

DRV_CANFDSPI_FilterDisable(0, CAN_FILTER0);

DRV_CANFDSPI_FilterObjectConfigure

Filter Object Configuration. Configures ID of filter object.

Sintax

int8_t DRV_CANFDSPI_FilterObjectConfigure(CANFDSPI_MODULE_ID index,

CAN_FILTER filter, CAN_FILTEROBJ_ID* id);

Parameters

Index = 0

filter = CAN_FILTERN, being N any number between 0 and 31. It usually works with N = 0.

id = calls the values of the CAN_FILTEROBJ_ID, previously set.

Return values

0

Precondition

Must be in configuration mode.

The filter must be disabled before changing the filter object.

Side effects

None.

Exemple

fObj.word = 0;

fObj.bF.SIDA = 0x300>>3 ;

fObj.bF.SIDB = 0x300 & (0x03) ;

fObj.bF.EXIDE = 0;

DRV_CANFDSPI_FilterObjectConfigure(0, CAN_FILTER0, &fObj.bF);

DRV_CANFDSPI_FilterMaskConfigure

Filter Mask Configuration. Configures Mask of filter object.

Sintax

int8_t DRV_CANFDSPI_FilterMaskConfigure(CANFDSPI_MODULE_ID index, CAN_FILTER filter, CAN_MASKOBJ_ID* mask);

Parameters

Index = 0

filter = CAN_FILTERN, being N any number between 0 and 31. It usually works with N = 0.

mask = calls the values of the CAN_MASKOBJ_ID, previously set.

Return values

0

Precondition

Must be in configuration mode.

The filter must be disabled before changing the mask object.

Side effects

None.

Exemple

mObj.bF.MSIDA = 0x0;

mObj.bF.MSIDB = 0x0;

mObj.bF.MIDE = 1; // Only allow standard IDs

mObj.bF.MEIDA = 0x0;

mObj.bF.MEIDB = 0x0;

DRV_CANFDSPI_FilterMaskConfigure(0, CAN_FILTER0, &mObj.bF);

DRV_CANFDSPI_FilterToFifoLink

Link Filter to FIFO. Initializes the Pointer from Filter to FIFO. Enables or disables the Filter

Sintax

int8_t DRV_CANFDSPI_FilterToFifoLink(CANFDSPI_MODULE_ID index,

CAN_FILTER filter, CAN_FIFO_CHANNEL channel, bool enable);

Parameters

Index = 0

filter = CAN_FILTERN, being N any number between 0 and 31. It usually works with N = 0.

channel = CAN_CHANNEL, being N any number between 0 and 31. The RX channel is CAN_FIFO_CH1, whether the TX channel is CAN_FIFO_CH2.

bool = true or false. True, enables the filter, while False disables the filter.

Return values

0

Precondition

None.

Side effects

None.

Exemple

// Link FIFO and Filter

DRV_CANFDSPI_FilterToFifoLink(0, CAN_FILTER0, APP_RX_FIFO, true);

DRV_CANFDSPI_TefConfigure

Link Filter to FIFO. Initializes the Pointer from Filter to FIFO. Enables or disables the Filter

Sintax

int8_t DRV_CANFDSPI_FilterToFifoLink(CANFDSPI_MODULE_ID index,

CAN_FILTER filter, CAN_FIFO_CHANNEL channel, bool enable);

Parameters

Index = 0

filter = CAN_FILTERN, being N any number between 0 and 31. It usually works with N = 0.

channel = CAN_CHANNEL, being N any number between 0 and 31. The RX channel is CAN_FIFO_CH1, whether the TX channel is CAN_FIFO_CH2.

bool = true or false. True, enables the filter, while False disables the filter.

Return values

0

Precondition

None.

Side effects

None.

Exemple

// Link FIFO and Filter

DRV_CANFDSPI_FilterToFifoLink(0, CAN_FILTER0, APP_RX_FIFO, true);