Master Thesis

Master's Degree in Industrial Engineering

Development of a CAN-Wifi converter based on a ESP32

Report

Author: Eduard Valentino Birau
Supervisor: Manuel Moreno Eguílaz
Call: April 2018

Barcelona School of Industrial Engineering

Review

The main goal of this project is to prove that an open source microcontroller, such as the ESP32 Thing [3], can be used to implement a functional CAN-WiFi controller which can be configured by a device connected to the board over WiFi. The CAN-WiFi controller function is to read messages being sent over a CAN network, filter according user settings and send them to a device connected to the own ESP32 WiFi network.

This goal of the project has been met by developing an application based on components and libraries available in the open source community that has grown around the ESP32.

The application allows the user of the device connected to the ESP32 board, to send commands that allow full control of the CAN configuration and to start receiving the messages send by the nodes of the CAN network. The user is also capable of make use of the built-in filtering function of the CAN port, without the need of further data processing and the development of a specific application for this task.

This document explains the basic hardware and software tools used for the application and the development of the application itself, which has been wrote in C language according to the ESP-IDF framework. Finally, the tests carried out are showed and explained to better understand how the application works and how to send commands to the ESP32.

Summary

REVIEW 1

SUMMARY 3

1. INTRODUCTION 6

1.1. Objectives of the project 6

1.2. Scope of the project 6

2. HARDWARE 7

2.1. ESP32 7

2.1.1. ESP32 Forum and learning material 8

2.1.2. CAN Port 9

2.2. CAN Transceiver 9

2.3. Budget 11

3. CAN NETWORK 12

3.1. History 12

3.2. CAN Architecture 13

3.3. CAN Frames 14

4. SOFTWARE 16

4.1. CAN Driver 16

4.2. ESP WiFi Library 18

4.3. Development 21

4.3.1. Setting the Environment 21

4.3.2. Node to Node communication 24

4.3.3. Test Bench 26

4.3.4. First Test 27

4.3.5. WiFi communication 28

4.3.6. Second Test 33

4.3.7. Dual Filter Mode 36

4.3.8. Final Test 39

5. ENVIRONMENTAL IMPACT 42

CONCLUSIONS 43

ACKNOWLEDGMENTS 44

6. ANNEXES _____ 45

6.1. Demo Source Code 45

6.2. Task_STA_CMD() 48

1.1. Modified task_CAN() routine 50

1.1. Modified CAN_init() routine 51

8. REFERENCES _____ 53

1. Introduction

IoT devices are becoming more and more popular among the developer community, which has caused manufacturers from all other the world to present new affordable solutions into the market. One of these manufacturers is Espressif [1], a company that has launched to the market a very cheap and reliable IoT chip, the ESP32 chip.

On the other hand, CAN interfaces and loggers are quite an expensive product, which make them affordable only by professionals of the automotive industry. The CAN bus is often studied in engineering courses but only its theoretical basis and its message frame, usually student do not perform actual tests on a real CAN network to improve their understanding of the well establish standardized CAN protocol.

In an effort to make CAN practices affordable by engineering students and professors, the ESP32 will be used to connect to an actual CAN network and send messages to an external device connected to the microcontroller, making use of the WiFi connectivity and CAN port of the ESP32 Thing board [3].

1.1. Objectives of the project

The main objective of the project is to prove that a currently commercialized cheap microcontroller, such as the ESP32, can be used to obtain the streaming data from a CAN Bus Network. The ESP32 functioning as a CAN-Wifi controller must be able to collect all the messages in the network and send them to a third-party device, such as a computer, tablet or smartphone.

Communication parameters, such as Baud Rate or Acceptance Filters and Masks, will be controlled from the same third-party device.

1.2. Scope of the project

To achieve the objectives of this project, a CAN Driver will be used and will allow to interact directly with the ESP32 CAN port and obtain the messages streaming on the network.

Using the microcontroller WiFi functionality, the converter will be able to act as an Access Point, so creating its own WiFi network. All devices connecting to this network will be able to set the communication parameters and receive the messages from the CAN network.

2. Hardware

As mentioned before the main piece of hardware needed for the proposed CAN-Wifi converter is the well-known microcontroller ESP32. It is a very cheap microcontroller with a broad functionality. The device will be furtherly described in section 2.1.

Besides the ESP32 itself, the only hardware needed has been the SN65HVD230, which is a CAN transceiver, allowing the own ESP32 CAN controller to connect to the actual bus lines. Further description of the device can be found in section 2.2.

2.1. ESP32

The actual board used for this project is the ESP32 Thing, manufactured by *SparkFun Electronics*®, using the ESP32 chip manufactured by Espressif Systems [1].

The main features of the board are:

- Dual-core Tensilica LX6 microprocessor
- Up to 240MHz clock frequency
- WiFi connectivity
- Bluetooth connectivity
- Very Low consumption
- Integrated LiPo Battery Charger

The light-weight operating system FreeRTOS allow for multitasking, which is needed when multiple threads must be executing at the same time. The RTOS scheduler rapidly switches between tasks, making it appear as if each task had been executing simultaneously. This functionality will be used for this application to have the WiFi task reviewing the state of connections, while the CAN task continuously pools messages from the CAN network.

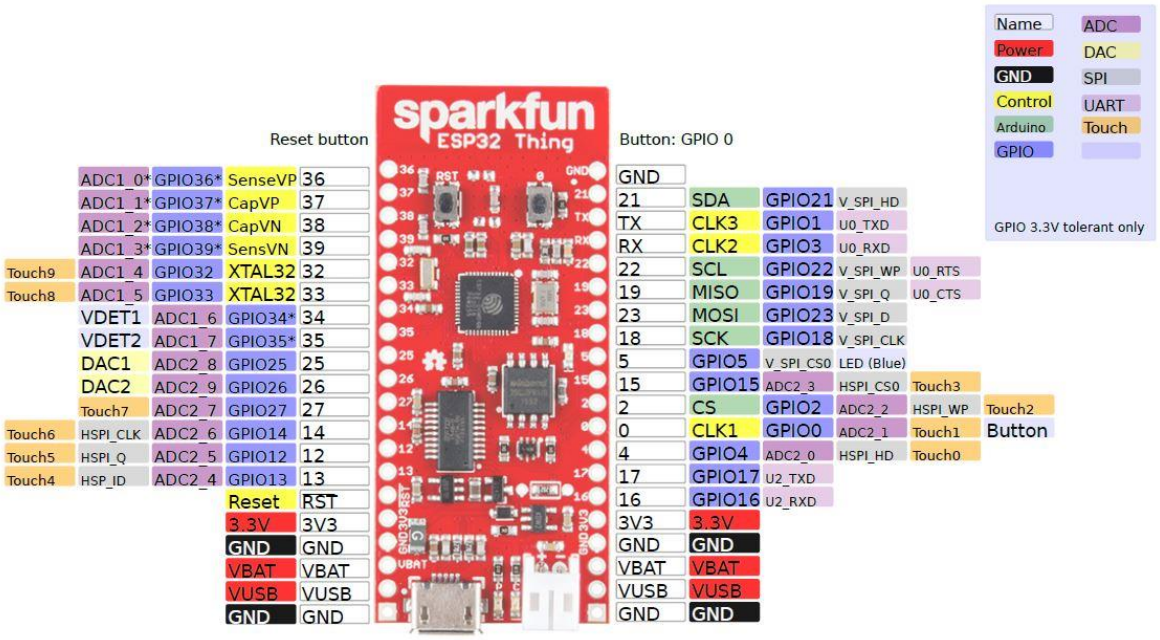

Fig. 1 ESP32 Thing pin-out. Source: [3].

The development of apps for this microcontroller is done thanks to the ESP-IDF development framework [6]. This framework is a set of open source libraries and tools needed to deploy apps on the ESP32 microcontroller and to run all its available hardware.

ESP-IDF and the toolchain to build and flash the app into the ESP32 can be installed on both Windows or Linux operating systems (link to the *Get-Started Guide* can be find in References). After experiencing several problems and long building and flashing times, it is highly recommended to use a Linux based platform, such as *Ubuntu*. More details can be found in section *Setting the Environment*.

2.1.1. ESP32 Forum and learning material

The ESP32 versatility, high processing power, low consumption, low price and most important of all high connectivity, has allowed it to be embraced and widely used by the IOT Developers Community. This has resulted in the creation of a very active and complete forum, which is the *Espressif ESP32 Forum [2]*.

The forum is the best way to look for solutions during the installation procedure of the ESP-IDF, while coding and when in search of new ideas that can speed up your project.

Besides the forum there is learning material all over the Internet, including a very complete ebook about the ESP32 entitled "*Kolban's book on ESP32*" [8], wrote by the software engineer and developer Neil Kolban. The author is a firm believer in sharing knowledge.

That is why the ebook can be downloaded for free from *Leanpub* website [4].

2.1.2. CAN Port

Along with the WiFi transceiver, the CAN port is the key feature for this project. This port is based on the SJA 1000 Stand-Alone CAN controller, so the library and the code development done in this project have been made consulting its datasheet [7].

The SJA 1000 is a widely use controller in the automotive and general industry. It supports both standard (11-bit) and extended (29-bit) identifiers and bit rates up to 1Mbits/s.

It has two modes of operation: BasicCAN and PeliCAN. The main difference that concerns this project is the Acceptance Filter Mask function. In BasicCAN the Acceptance Filter size is only 8-bit, which does not allow for a complete filter of the identifier, being it at least 11-bit long. In PeliCAN mode the Acceptance filter has two modes:

- Single Filter. The Acceptance registers are each 4 bytes long, one register is assigned to the acceptance code and the other to the acceptance mask.

- Dual Filter. This is a two-stage filter. Each stage has two 2 bytes registers assigned, one for the acceptance code and the other for the acceptance mask.

As it will be described in section 0, the acceptance code is used to define the actual value that we want to accept, and the acceptance mask to define which bits of said code has to be taken into account to filter.

Due to longer registers available in PeliCAN mode, filtering can be done over the entire identifier of the message, both in standard and extended frame, and include data in standard frame only.

2.2. CAN Transceiver

The CAN Transceiver allows the ESP32 CAN controller to access the CAN_H and CAN_L lines of the network. Therefore, its basic function is to adapt the CAN bus levels to levels compatibles with the CAN controller for incoming messages, but also very important to protect the controller by a possible power surge on the bus (ESD protection). It also allows the controller to transmit new messages over the bus by adapting the data to CAN bus levels.

The transceiver used in this project is the SN65HVD230 CAN Board, manufactured by Waveshare, which works with a voltage level of 3.3V, compatible with the ESP32.

Fig. 2 SN65HVD230 CAN Board. Source: [14].

The integrated circuit mounted on the board is the SN65HVD230, manufactured by Texas Instruments. It is intended for use in application employing the CAN serial communication physical interface according the ISO 11898, series of standards that defines the CAN. Working in High Speed mode, by connecting the pin Rs to ground (see Fig. 3), can allow up to 1 Mbps signal rates.

Because of the high input impedance state, that the IC assumes while no data is being transmitted on to the bus by the node, it is possible to have up to 120 nodes connected to it.

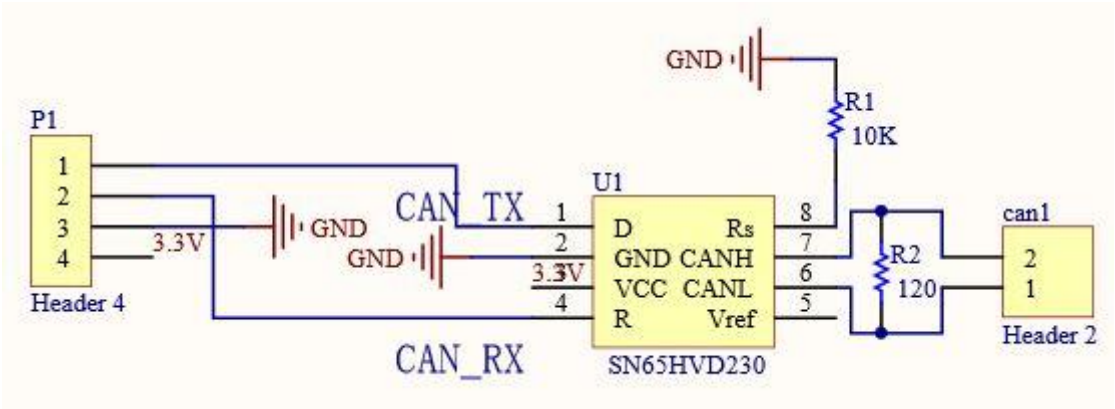

Fig. 3 SN65HVD230 CAN Board Schematics. Source: [14].

By inspecting *Fig. 3* it is possible to find the SN65HVD230 IC mentioned before. Header 4 of the schematic is connected to the CAN controller, being pin 1 and 2 for transmission and reception, respectively. Pin 4 and 3 power the transceiver board and consequently the IC. Header 2 has two pins that connect directly to the bus lines.

2.3. Budget

Components	**Unit**	**Price**	**Total**
Sparkfun ESP32-Thing DEV-13907	2	€ 28,00	€ 56,00
Waveshare SN65HVD230 CAN Board	2	€ 9,41	€ 18,82
ProtoBoard 1660 contacts	1	€ 20,00	€ 20,00
Development	**Hours**	**Price**	**Total**
Programming	180	€ 40,00	€ 7.200,00
Testing	80	€ 40,00	€ 3.200,00
Report	40	€ 25,00	€ 1.000,00
Amortization	**Months**	**Price**	**Total**
Laptop ASUS S400	6	€ 13,00	€ 78,00
		Total Budget	€ 11.572,82

Table 1 Budget. Source: own.

3. CAN Network

3.1. History

History of automobiles starts in 1769 with the invention of the first steam-powered automobile capable of transporting human by Nicolas-Joseph Cugnot. Following this and during almost the entire 19th century engineers and inventors make continuously attempts to find the best propulsion solution for the automobile.

Two of the nowadays most used solution are the four-stroke petrol internal combustion engine patented by Nikolaus Otto in 1877 and its Diesel counterpart invented by Rodolf Diesel.

Until the Post-war era, most efforts were putted in improving the propulsion system and in building cars only affordable by richest part of society. After the end of World War II and especially from the 60s, manufacturers started to change the paradigm and focused on mass-produced vehicles that could be afford by the working class.

With the development of electronics, the need to integrate electronic devices in automobiles and the use of ECUs, in 1983 Bosch starts the development of the Controlled Area Network (CAN). This was an effort to avoid the problems caused by having one physical communication channel for each signal to be transmitted, reduce the amount of wiring needed to connect each ECU and have a more reliable and fast way of communication.

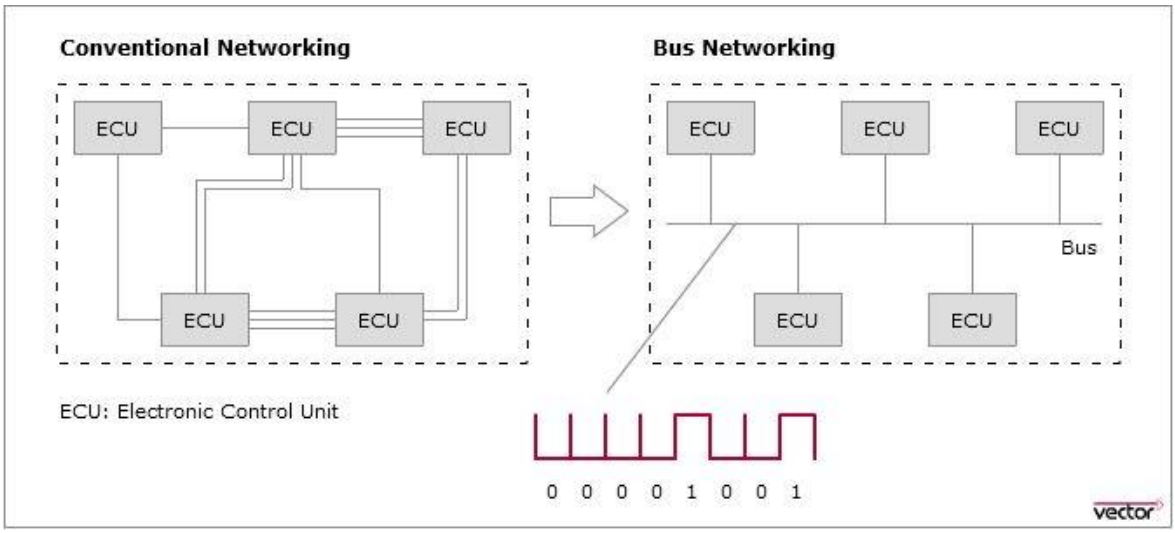

Fig. 4 Conventional Networking vs CAN Networking. Source: [12].

Finally, in 1987 the first CAN controller chip goes on the market, produced by Intel and Philips. Next in 1991 the first ever production-vehicle to use the CAN based wiring system is manufactured, the Mercedes Benz W140.

In 1993 starts the standardization of this protocol by the release of the CAN standard ISO 11898. From this moment on, the CAN protocol begins to be widely accepted by automotive manufacturers and it is subject to continuous improvement making it reliable and able to satisfy real-time requirements.

3.2. CAN Architecture

A CAN Network consists of a number of CAN Nodes, linked via a two-line physical transmission medium called CAN Bus. Each CAN Node is connected to this bus via a CAN interface, that allows the node to receive and send messages.

The two lines of the can bus, CANH and CANL, are two unshielded twisted wires over which a symmetrical signal runs. This is done to minimize electromagnetic disturbances over the signal induced by motors, ignition systems and switch contacts. It also improves electromagnetic compatibility by reducing the magnetic field generated by the signals running through the bus.

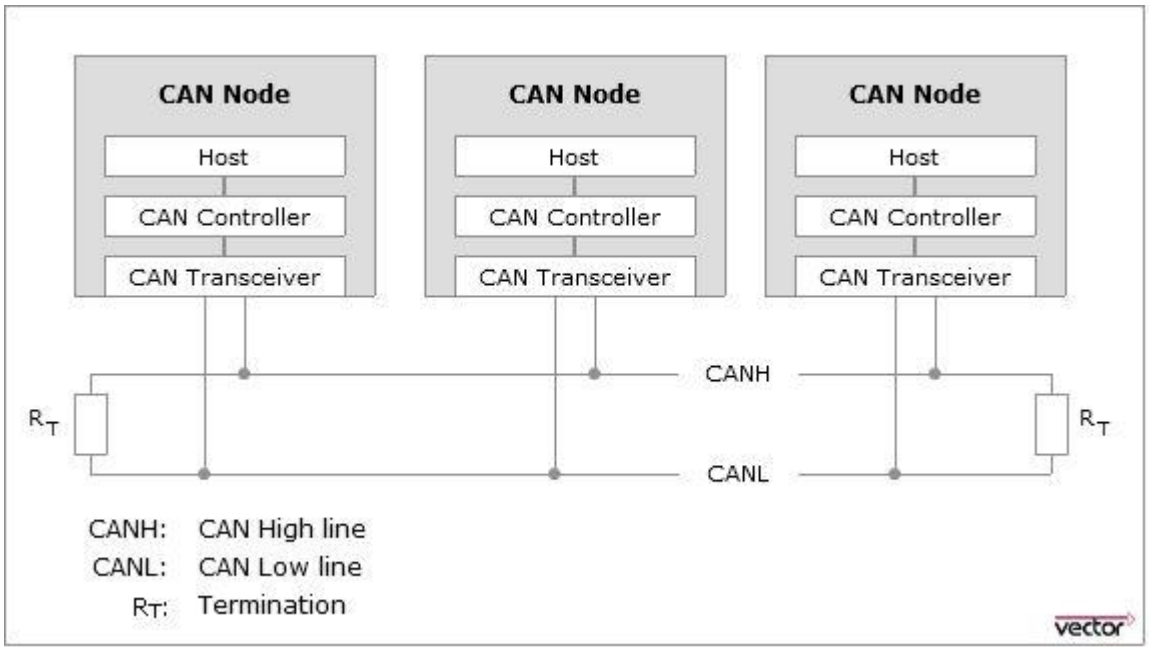

Fig. 5 CAN Architecture. Source: [12].

Maximum data rate is 1 Mbit/s and maximum network extension is up to 40 m. At each end

of the CAN Network there is a termination resistance of 120Ω to prevent transient phenomena such as reflections.

3.3. CAN Frames

CAN Protocol makes use of so-called frames to exchange data between nodes in the network. A frame is a composition of fields that allows each node to correctly receive the message, evaluate its integrity and decide to take it into account and pass it to the host of said node or not. This is necessary because each frame sent by a node is broadcasted to every node connected to the network and not only to its true target.

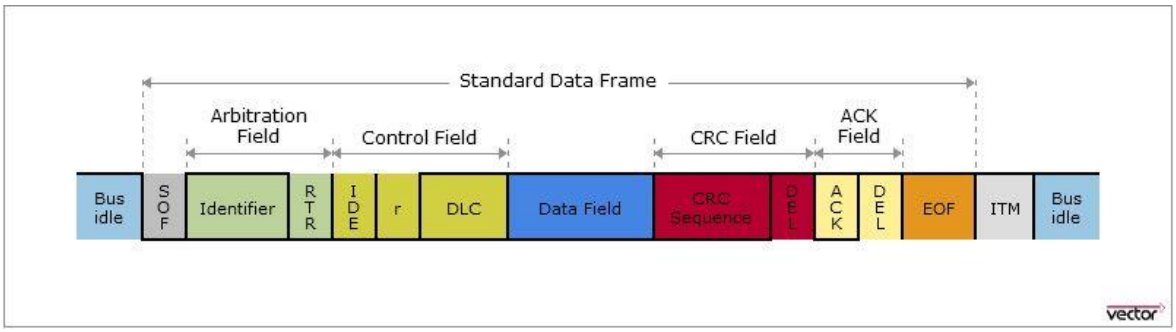

Fig. 6 CAN Standard Frame. Source: [12].

As can be seen in *Fig. 6* there are five main fields:

- Identifier field. It is composed by the identifier of the message, which sets its priority, and the RTR bit, used to request data.
- Control field. It is composed by IDE field which indicates if the identifier is standard, 11-bit long, or extended, 29-bit long. DLC field indicates the actual size of the data field.
- Data field. It contains the actual data to be transmitted, which can be up to 8 bytes.
- CRC field. Checksum used to evaluate the integrity of the payload.
- ACK field. This field is used to acknowledge by the receivers, the integrity of the message, having checked the CRC field.

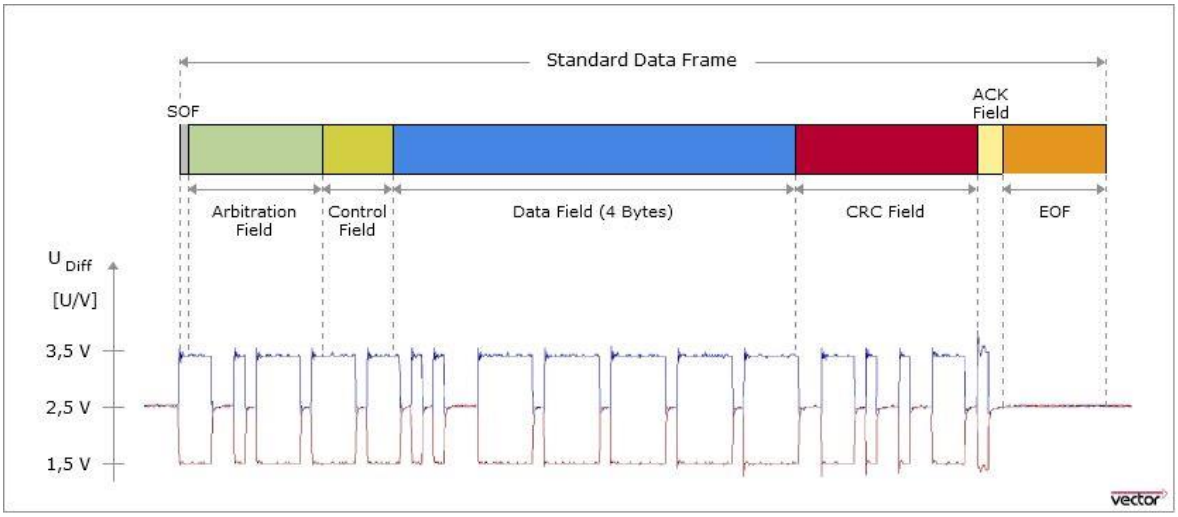

Fig. 7 Example of a standard frame transmission. Source: [12].

4. Software

In the following sections the library used to develop the CAN WiFi controller will be explained, as well as the actual development starting from the setting of the environment in the computer used and going through the phases of the development.

4.1. CAN Driver

The CAN Driver, which acts as the interface with the actual CAN Module installed on the ESP32 chip, has been developed by Thomas Barth, German PhD student, and published in his own technology blog *Barth Development* [15].

The library is divided into four main parts:

- API Configuration. The structure of the message frame is defined in this file (see section 3.3) and also the main routines are defined, such as *CAN_init()* necessary to initialize the CAN Driver, *CAN_write_frame(frame)* used to send message frames in to the bus and *CAN_stop()*, used set the ESP32 CAN Port in reset mode.

- Configuration. All the configurations needed for the correct initialization of the CAN Driver are defined: the available CAN speed values at which the Driver can be set, the ESP32 pins used for the connection to the bus lines, through the CAN transceiver (see section 2.2) and the FreeRTOS queue handler assigned [16]. This message handler acts as a FIFO buffer (First In First Out) and it will be set to be able to store up to 10 CAN frames.

- Register Definition. Registers of the actual CAN Port installed on the chip are defined. These registers are assigned using as starting point the ones defined in the SJA1000 datasheet. Key registers are the ones that allow to access the TX/RX interface of the port, the ones used to define the Acceptance Filter (including acceptance code and acceptance mask). Beside the Acceptance Filter registers the RM and AFM registers are needed to correctly configure the CAN Port.

  RM register is used to set the port in reset mode, mode in which the filter can be modified and set to the desired value. AFM register is used to set the filter in single or dual mode (see section 2.1.2).

- Routines. In this part all the routines necessary to initialize and communicate through the CAN bus are defined.

- CAN_init(). First the CAN module is enabled by activating the CAN clock bit and disactivating the reset bit of said module. Next the TX and RX pins are assigned and set to output and input respectively.

  Following, all the necessary registers of the CAN module are define and set to its correct value, for example register CDR.B.CAN_M is set to 0x01 to enable PelicanMode (see 2.1.2), register BTR1.B.TSEG1 is set depending on the Baud Rate defined in the configuration structure before initialization, acceptance code and acceptance mask are set in order to accept all messages and finally register MOD.B.RM is set to 0 to exit reset mode and start receiving/sending messages.

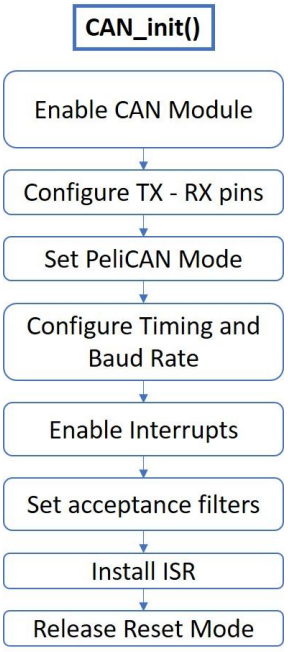

Fig. 8 CAN_init() diagram.

- CAN_isr(void *arg_p). The Interrupt Service Routine is defined and enabled. The only interrupt used for this application is the one associated to the reception flag, so each time a new message is received the CAN_read_frame() routine is executed.

- CAN_read_frame(). This routine copies the message received into the data frame defined in the API configuration, and then send it into the queue so it can be read by the corresponding task. Finally, the CMR.B.RRB register is set to 1 to let the hardware know that the message has been received and read correctly. This will cause the memory space of the RXFIFO, used by the received message, to be released so it can be occupied by the next incoming message activating the reception flag and repeating the read action.

- CAN_write_frame(const CAN_frame_t* p_frame). The p_frame parameter is the frame which needs to be sent over the CAN bus. Its structure complies with the frame structure defined in the API configuration. The routine differs between standard or extended frame and the copies the p_frame content into the Transmission Buffer Layout (see section 6.3.7 of [7]). The transmission action ends by setting the CMR.B.TR register to 1 to let the hardware know that a new frame is ready to be transmitted.

- CAN_stop(). To stop all actions of the CAN module, the MOD.B.RM register is set to 1 to force the CAN module to enter reset mode.

4.2. ESP WiFi Library

The ESP WiFi library is one of the standard components installed inside the ESP-IDF (see section 2.1). The most important functionalities of the ESP32 WiFi are:

- It can support Station-only mode, SoftAP-only mode, Station/SoftAP-coexistence mode. Station-only mode means that the ESP32 connects to an already existing WiFi network, SoftAP-only mode generates an Access Point allowing close by devices connect to it. Coexistence mode allows the microcontroller to be able to connect to a WiFi network and simultaneously to generate an Access Point.

- It supports WPA/WPA2/WPA2-Enterprise and WPS.

- It supports Modem-sleep, which allows for low power consumption periods.

- It can reach up to 20 Mbit/sec TCP throughput.

The WiFi Driver is designed to work independently of other tasks, such as the TCP-IP communication task or the CAN task, but it sends events to let know the running tasks when the WiFi is ready, when a new scan of the network has been made, when a new station or client has connected to the Access Point, etc.

The tasks running in the main program are able to interact with the driver by the so-called API calls:

```
1.  ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
2.  ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_AP, &ap_config));
3.  ESP_ERROR_CHECK(esp_wifi_start());
```

- esp_wifi_set_mode. Sets the driver working mode: Station-only (WIFI_MODE_STA) or AccesPoint-only (WIFI_MODE_AP).

- esp_wifi_set_config(). Points to the driver the structure where the Access Point configuration is defined. In this structure the following parameters can be found: SSID, Password, Channel, Authentication mode, Maximum connections and Beacon Inteval.

- esp_wifi_start(). The Driver is started with the pointed configurations and in the working mode previously defined.

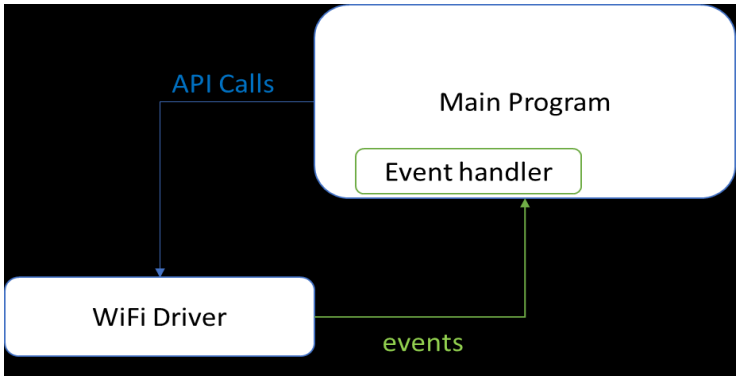

Fig. 9 WiFi Driver Diagram. Source: own.

An event handler is needed to handle a WiFi driven event and notify the running tasks. Notifying is done by a set of bits linked to each WiFi event via the handler. This action is done by the FreeRTOS function:

- EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup,

  const EventBits_t uxBitsToSet);

  Where xEventGroup is the EventGroup in which the bit must be set and that has been previously created. All other parameters are self-describing.

Tasks read or wait for said bits using the function:

- EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup,

  const EventBits_t uxBitsToWaitFor,

  const BaseType_t xClearOnExit,

  const BaseType_t xWaitForAllBits,

  TickType_t xTicksToWait);

Event handler and Bits definition code:

```
1.  // Event group
2.  static   EventGroupHandle_t wifi_event_group;
3.  const  int  STA_CONNECTED_BIT = BIT0;
4.  const  int  STA_DISCONNECTED_BIT = BIT1;
5.  const  int  AP_STARTED_BIT = BIT2;
```

Event handler routine:

```
esp_err_t event_handler(void  *ctx, system_event_t *event)
{
    switch (event->event_id)
    {
        case SYSTEM_EVENT_AP_START:
            printf("Access point started\n");
            xEventGroupSetBits(wifi_event_group, AP_STARTED_BIT);
            break ;
        case SYSTEM_EVENT_AP_STACONNECTED:
            xEventGroupSetBits(wifi_event_group, STA_CONNECTED_BIT);
            break ;
        case SYSTEM_EVENT_AP_STADISCONNECTED:
            xEventGroupSetBits(wifi_event_group, STA_DISCONNECTED_BIT);
            break ;
        default  :
            break ;
    }
    return  ESP_OK;
}
```

- STA_CONNECTED_BIT. It is set by the handler when a new station successfully connects to the access point.

- STA_CONNECTED_BIT. It is set by the handler when a previously connected station disconnects.

- AP_STARTED_BIT. It is set after correctly initializing and executing the Access Point.

For better understanding of the interaction between running tasks and the WiFi driver, see Fig. 10 Main Program - WiFi Driver interactions.

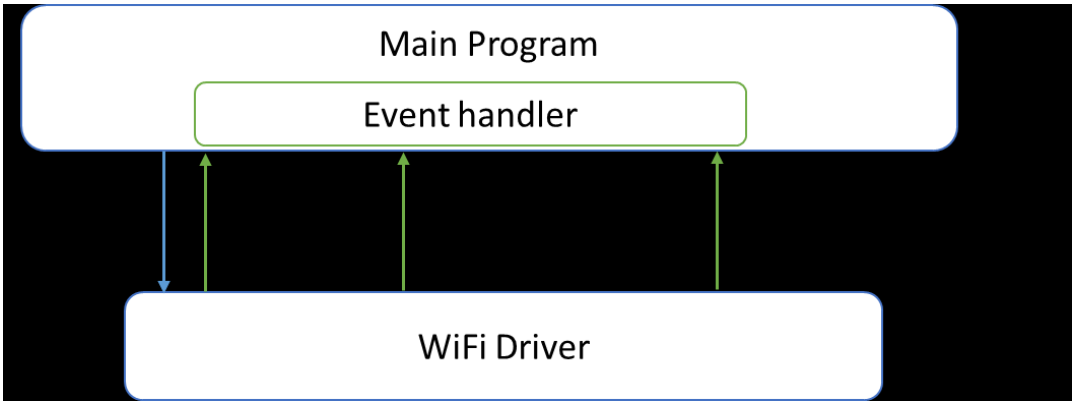

Fig. 10 Main Program - WiFi Driver interactions. Source: [17].

4.3. Development

Development is done by implementing step by step the functions that the microcontroller must be able to perform. Between each step, a round of tests has been performed to ensure the correct behaviour of the code.

4.3.1. Setting the Environment

First the environment needed to build and flash the code into the microcontroller must be set. ESP-IDF can be used either on a Linux based operating system or on Windows. Windows does not have a built-in make environment tool that automatically builds and transforms executable programs and libraries from a source code to a target result that can then be flashed into the microcontroller, so the *MSYS2* [18] environment must be installed to perform this function.

The Windows option has been tested and used in the first attempts of flashing the classical "Hello World" code into the ESP32 controller, but the current version of both ESP-IDF and MSYS2 causes the building and flashing of the code very slow, taking up to 10 minutes to build all the necessary directories and files each time or modification of the source code is made. This fact has a tremendous effect on the ability to make quick changes to the code and consequently test them. For this reason, the user has preferred to change to a Linux based operating system.

To avoid having to install the Linux OS, in this case *Ubuntu* [19], on the physical laptop, the user has preferred to install the *Oracle VM VirtualBox* [20], mounted a virtual machine with the Ubuntu OS installed on it.

The complete guide to set up an Ubuntu VM can be found at *Install Ubuntu on Oracle VirtualBox [21].* After completing the installation process and have the Virtual Machine running, from now on VM, it is time to set up the Toolchain, to be able to build the source code, and finally download the ESP-IDF API collection from GitHub. A guide for this step can be found at *ESP-IDF Programming Guide [6]*.

After the installation process and in order to be able to flash the code into the ESP32, the VM must be able to see the microcontroller connected via USB. Every USB connection is done to the physical laptop and can be seen by the OS installed in it. To transfer this USB connection to the VM, the connected device must be activated in the Oracle VM VirtualBox Toolbar: Devices→USB. Here a list of all the USB connected devices can be found. To make the transfer simply click on the device. The ESP32 will be found under the name "FTDI FT231X USB UART" or similar (see Fig. 11 Activating ESP32 Device in the VM.Fig. 11).

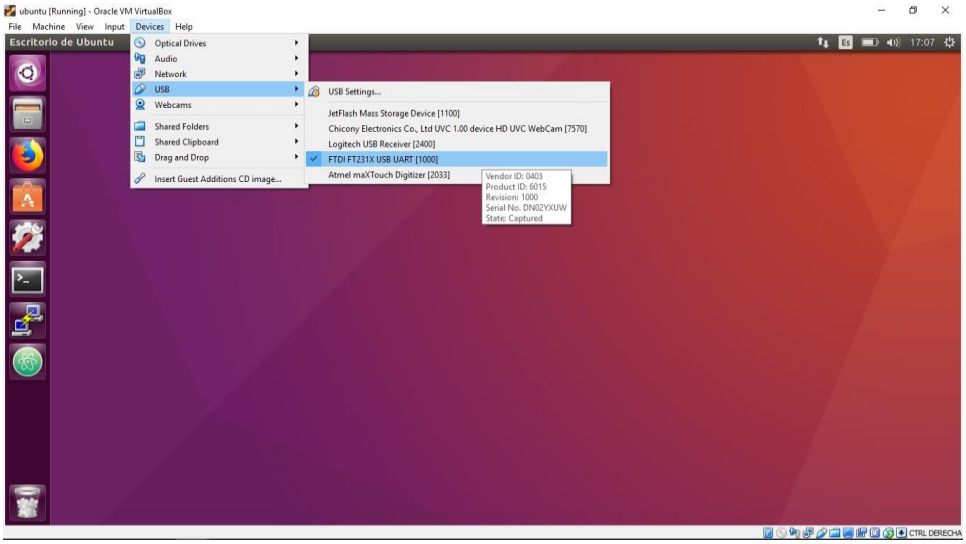

Fig. 11 Activating ESP32 Device in the VM. Source: own.

Having set up the environment the first test must be, of course, the classic "Hello World" code, which can be found at ESP-IDF Programming Guide [6]. After copying it in the *esp* directory, a Terminal must be opened and type in the following code:

```
1.  cd ~/esp/hello_world
2.  make menuconfig
```

This will allow to move where into the hello_world directory and then execute menuconfig, which is a special menu, which allows to graphically set ESP-IDF configurations. General configurations common to almost all applications are located in the *Serial flasher config*, where the Default serial port must be set to */dev/ttyS0*, the *Default baud rate* set to 115200 and *Flash SPI speed* to 40 MHz (see Fig. 12, Fig. 13, Fig. 14, Fig. 15), this last configuration is specific for the *ESP32 Thing board [3].*

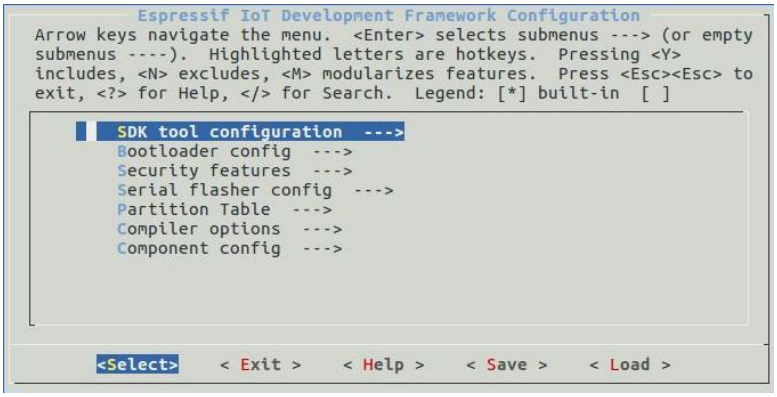

Fig. 12 ESP-IDF Menuconfig. Source: own.

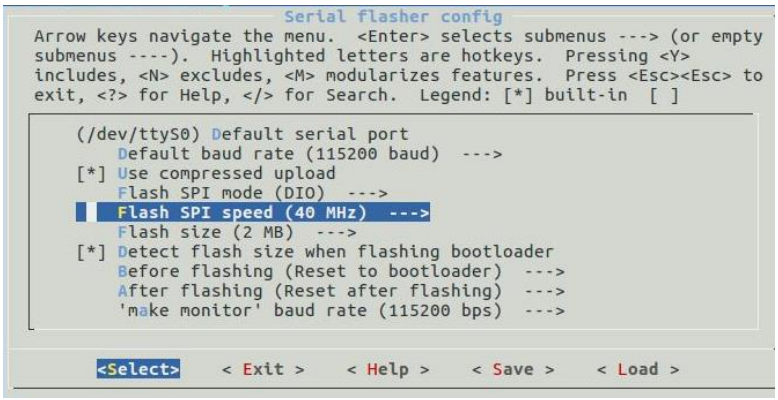

Fig. 13 Serial flasher config. Source: own.

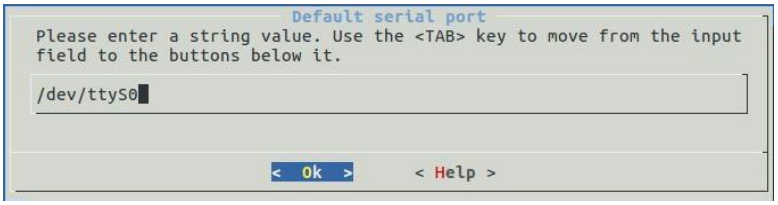

Fig. 14 Default serial port. Source: own.

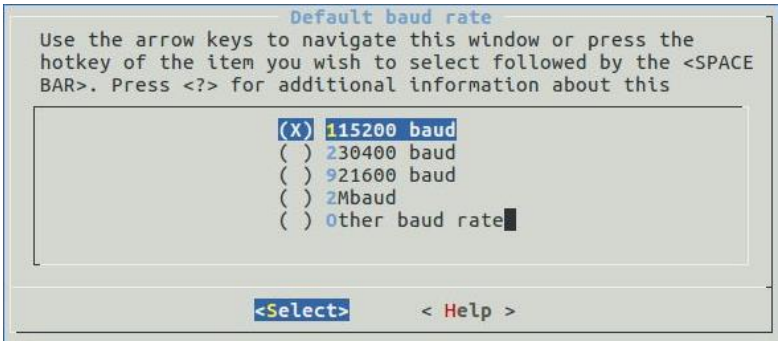

Fig. 15 Default baud rate. Source: own.

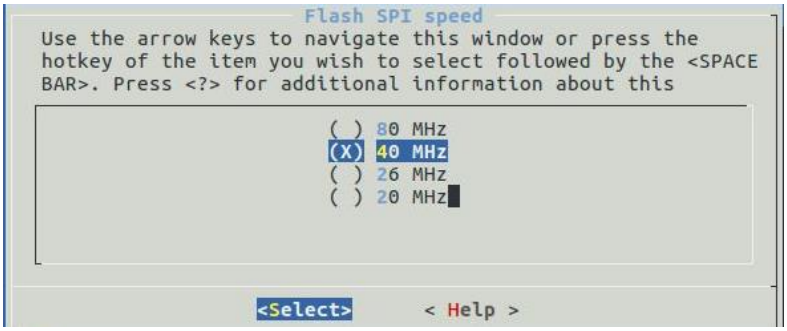

Fig. 16 Flash SPI speed. Source: own.

After exiting menuconfig and back to the terminal, type

```
1.  make flash
```

to build all the necessary directories and files, and then flash the application into the microcontroller. If the flashing has been done correctly, no error will be printed in the terminal and if a serial connection is done to the ESP32, with a software such as P*uTTY* which can be installed from Ubuntu's Software Center, at port /dev/ttyS0 and speed 115200, the output of the microcontroller will be:

```
1.  Hello world!
2.  Restarting in 10 seconds...
3.  Restarting in 9 seconds...
4.  Restarting in 8 seconds...
5.  Restarting in 7 seconds...
```

4.3.2. Node to Node communication

After setting the environment, next step is to test the demo applications of the CAN library. The simplest setup in which all the needed configurations and the code can be tested is by installing a two node CAN network, where one is the sender of the message and the other is the receiver.

The complete code for this test can be found in the section 6.1.

The part of code executed after the initialization of the microcontroller is the *app_main* routine, which acts as a setup before starting the actual task. In Fig. 17 its diagram is showed, which is very simple: after flash initialization, the CAN configuration parameters are printed to serial. These parameters can be changed by accessing *Component config* inside *menuconfig* (see section 4.3.1). Here configurations of all the ESP-IDF components can be found. By accessing the ESPCan component the following configurations can be made:

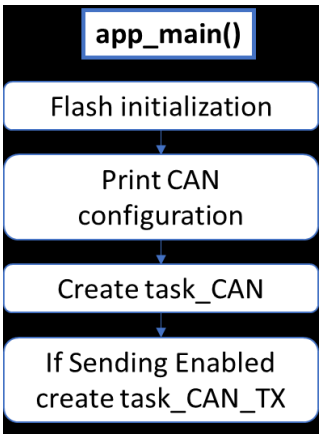

Fig. 17 app_main() diagram. Source: own.

- *ESP CAN Node itself*. Here the node ID can be defined. This will be set as the message ID in each transmission.

- *Pin Number of CAN RXD*. Pin used for reception is defined. The common used pin

is the 4th.

- *Pin Number of CAN TXD*. Pin used for transmission is defined. The common used pin is the 5th.
- *Select the ESP_CAN_SPEED*. The baud rate used for connection is set.
- *Enable/disable to send a test frame*. It must be enabled before flashing the application in the transmission node.

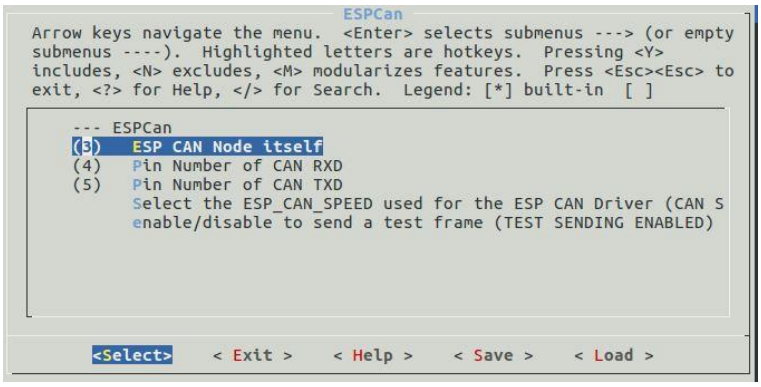

Fig. 18 ESPCan configuration. Source: own.

The task in charge to receive the messages from the CAN bus is the *task_CAN.* First, the reception frame is defined, next the queue where the received message will be copied is created. After this, the CAN may be initialized. As can be seen in the code, the *CAN_init* function needs four parameters. First two parameters will be used to set the Acceptance filter code and the last two to set the mask. For now, because all messages ID are to be received, these parameters are set to 0 and 0xff, respectively.

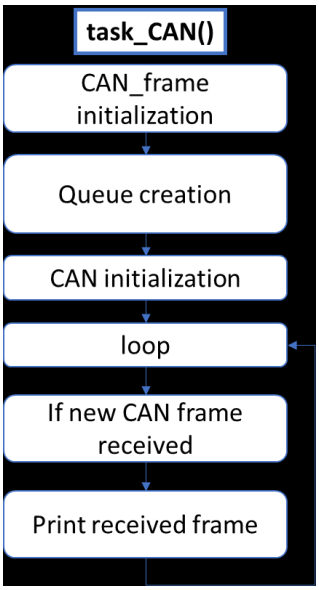

Fig. 19 task_CAN() diagram.

Having initialized the CAN module correctly, the main loop starts. In each iteration the pass condition is the reception of a new message in the queue, moment when the received frame is serial printed.

For the transmission node, where the transmission test has been enabled (see Fig. 18), the task_CAN_TX() will also be executed. As can be seen in the complete code, its start is delayed a few hundred milliseconds after the start of task_CAN() to ensure the complete initialization of the CAN module. Task_CAN_TX() is a very simple routine which has a loop, that in each iteration increases a

counter, whose is set as the 8th bit of data inside a predefined transmission frame. After the counter reaches a maximum value, this restarts from 0.

4.3.3. Test Bench

The hardware described in section 2 has been used to prove the correct functioning of the application and to look for possible bugs. The components have been mounted on the top of a protoboard to have easy access to both ESP32 and CAN transceiver pins.

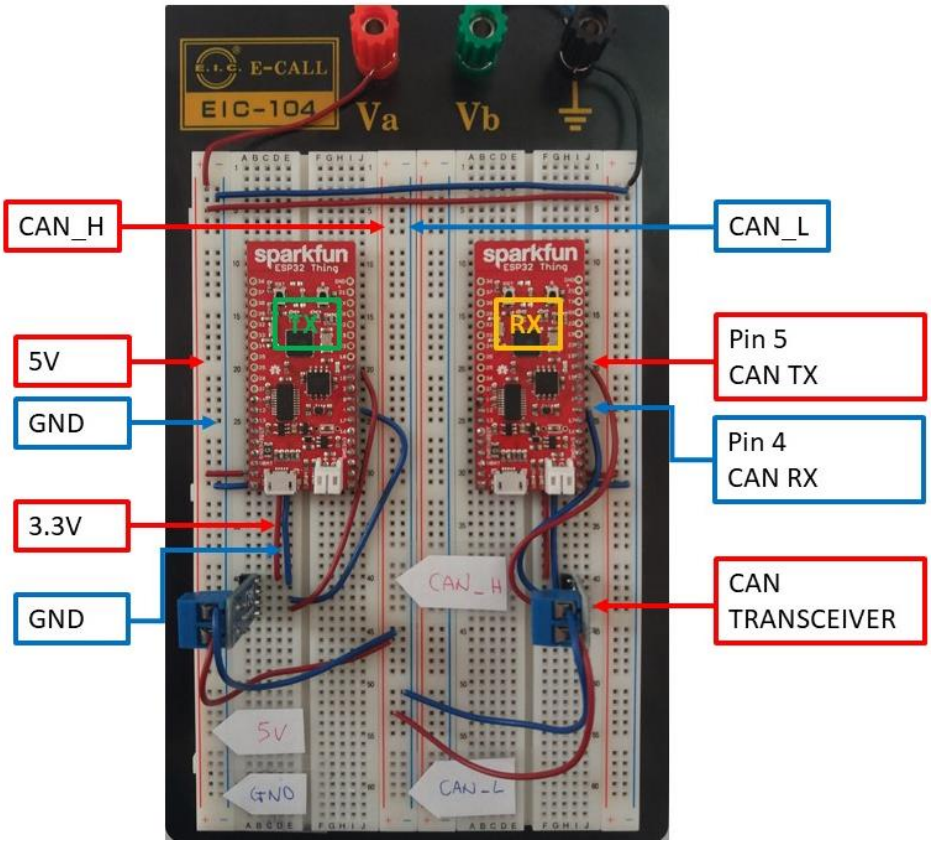

Fig. 20 Test Bench. Source: own.

As can be seen in *Fig. 20* there are two ESP32 boards, the one on the left being the transmission node and the one on the right the receiving node. Of course, both nodes are able to send and receive, but for testing purposes this configuration is the best. The receiving node for now only receives messages but it will also be the node that generates the access point for other devices to connect to (see section 4.3.5).

Both nodes share the same power supply, which can come from one of them being connected via the micro-USB connector to a laptop or a power source or from the top connectors *Va* and ground. When using these connectors, the voltage supply

must not exceed 5V.

Each one of the nodes needs a CAN transceiver to be able to connect to the CAN network. Four lines connect each ESP32 to its transceiver: CAN RX and CAN TX, pin 4 and 5 respectively, and 3.3V and GND. It is very important to connect the transceiver to a 3.3V power supply, otherwise it will be damaged because it does not have an internal voltage regulator and either an overvoltage protection.

Thanks to the CAN transceiver, the ESP32 nodes can now be connected to the CAN_H and CAN_L lines. Even though both nodes are terminal nodes, they "close" the CAN network, no 120Ω resistance needs to be installed because each transceiver has one already integrated between the CAN_H and CAN_L pins.

4.3.4. First Test

The first test done with the test bench and the demo source has been done to prove the correct functioning of the CAN library, of the CAN port and to detect problems not related to the code itself but to the flashing of the ESP32.

The program used for this test is PuTTY, mentioned in section 4.3.1, because of its simplicity.

The test code used is the demo that came with the library. After flashing both ESP32 boards and setting one as the transmission node, enabling the test frame as shown in Fig. 18, and the other as reception node, the test started by powering one of the ESP32 boards with an USB cable connected to the PC. Setting PuTTY to listen to the serial transmission of the reception node allowed to see the test messages sent over the CAN network.

This test allowed to detect that the multiple tasks created need a certain amount of stack memory, which is allocated when it is created, to run correctly and not to crash. The amount of stack needed is defined experimentally, because after powering the board the code will stop executing and send an error message over the serial output.

When a sufficient value is set in the task creation, by modifying the *xTaskCreate()* parameters in the source code (see 6.1, line 134), the application will run correctly.

Another bug detected is when flashing the source code on the ESP32. If an error is shown on the terminal after running the *make flash* command (see section 4.3.1), that the connection with the ESP32 has timed out, the board has to be disconnected from the protoboard and when the *make flash* command is attempting connection, the reset button on the board (see Fig. 1) must be pressed for a brief moment.

The procedure must be repeated until the *make flash* command ends successfully.

CAN library and CAN port worked without any bugs, even after 8 hours of continuous transmission.

4.3.5. WiFi communication

Having tested the node to node communication it is time to give the receiving node the ability to generate an Access Point.

The CAN messages will only be received by the receiving node once the WiFi has been correctly initialized, the Access Point created, and a client has connected to it. So the API calls described in section 4.2 must be made in the app_main() task (see 4.3.2) before creating the task_CAN.

Parallel to the Access Point initialization the tcp_server task starts. This task allows to install a TCP server and use TCP/IP protocol to communicate with a connected device. An external device connected to the Access Point will be able to connect to the created socket by requesting a connection to the ESP32 IP and port. IP is defined when initializing the Access Point and the port when socket binding is done.

```
1.  // assign a static IP to the network interface
2.  tcpip_adapter_ip_info_t info;
3.  memset(&info, 0, sizeof (info));
4.  IP4_ADDR(&info.ip, 192, 168, 10, 1);
5.  IP4_ADDR(&info.gw, 192, 168, 10, 1);
6.  IP4_ADDR(&info.netmask, 255, 255, 255, 0);
7.  ESP_ERROR_CHECK(tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_AP, &info));
```

The TCP/IP adapter is configurated by assigning the desired value to the *info* structure composed by ip, gateway and netmask.

```
1.  ESP_LOGI(TAG,"tcp_server task started \n");
2.  struct   sockaddr_in tcpServerAddr;
3.  tcpServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
4.  tcpServerAddr.sin_family = AF_INET;
5.  tcpServerAddr.sin_port = htons( 3000 );
```

The TCP server is configured by defining in the `tcpServerAddr` structure 3 parameters: *s_addr* is set so that the socket can listens to any address connection request, *sin_family* is set to AF_INET which defines that the socket can only listen to addresses that comply with *Internet Protocol v4* [22] and finally *sin_port* defines the port the socket is listening to.

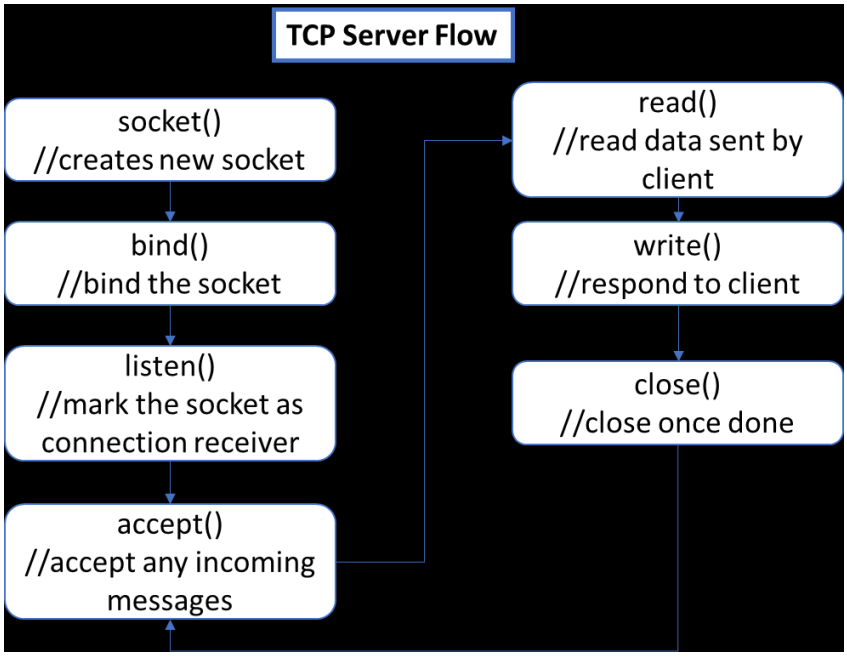

Fig. 21 TCP Server Flow. Source: [23].

The TCP Server Flow illustrated in Fig. 21 cannot start until the Access Point is correctly initialized. This is ensured by using the event handler described in section 4.2.

```
1.  EventBits_t evg_bits;
2.  xEventGroupWaitBits(wifi_event_group,AP_STARTED_BIT,false ,true ,portMAX_DELAY);
```

The tcp_server task will wait until the AP_STARTED_BIT is set to initiate the TCP Server flow. After initiation the socket is created and bonded to the IP address and port previously configured. Next the socket is set to listen to all incoming new connections; the *accept()* function will not return any value until a new connection is requested.

After the correct connection with an external device the task_STA_CMD() task is initialized, its function is to wait for incoming messages from the device. These messages, if sent with the correct format, will allow the task_CAN to execute and to send each received CAN message to the connected device.

Parallel to the task_STA_CMD() and task_CAN(), the tcp_server task is checking every 500 ms if the station, device connected to the established socket, is still connected. If not, the task will be stopped and the ESP32 will be forced to restart, terminating all executing tasks.

Two basic command structures have been defined for the task_STA_CMD:

- Start command. The station must send it if it wants the CAN communication to start. Besides the start command, it must also send the Baud Rate at which the CAN

connection must run. Optional parameters of the start commands are the one related to the Acceptance Filter. If not defined the Acceptance Filter will be set to accept all messages.

Command format:

<START;BAUD_RATE;ACR0;ACR1;AMR0;AMR1>

BAUD_RATE usual possible values: {100, 125, 250, 500, 800, 1000}.

ACRi: it is an 8-bit parameter, which defines the value against which the filter will compare the received message. At this point the application can be used to filter in single filter mode and with standard frames only. Also, only the 11-bit identifier and RTR bit are filtered by, so only two of the four 8-bit parameters are sent to the task_CAN. In addition to this only the four MSB of ACR1 are used.

AMRi: it is an 8-bit parameter that defines which bits of the ACRi are to be considered by the filter. A bit is considered if the correspondent bit in the mask is set to 0.

- Stop command. It is used to stop the communication between the ESP32 and the station. After receiving it, the ESP32 stops CAN and WiFi communication, restarts itself and waits for a new incoming connection.

Fig. 22 task_STA_CMD() diagram. Source: own.Fig. 22 shows the flow diagram of task_STA_CMD(), which is executed every 500 ms. Each loop checks if a new message from the station has been received, if so it analyses its contents and acts accordingly.

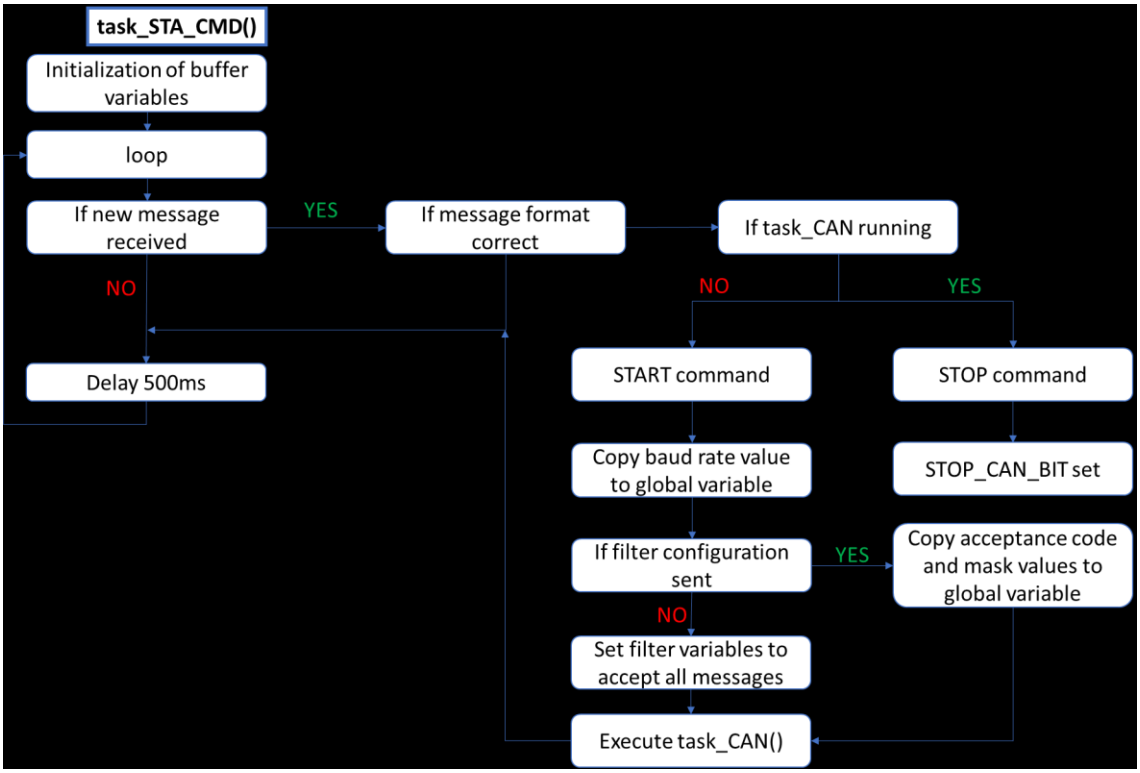

Fig. 22 task_STA_CMD() diagram. Source: own.

The start command previously explained integrates the values necessary to set the acceptance filter. These values are the acceptance code and the acceptance mask. All four values must be sent in hexadecimal format.

ACR0								ACR1							
ACRn7	ACRn6	ACRn5	ACRn4	ACRn3	ACRn2	ACRn1	ACRn0	ACRn7	ACRn6	ACRn5	ACRn4	X	X	X	X
IDn10	IDn9	IDn8	IDn7	IDn6	IDn5	IDn4	IDn3	IDn2	IDn1	IDn0	RTR	X	X	X	X
AMR0								AMR1							
AMRn7	AMRn6	AMRn5	AMRn4	AMRn3	AMRn2	AMRn1	AMRn0	AMRn7	AMRn6	AMRn5	AMRn4	X	X	X	X

Fig. 23 Filter and CAN message correlation. Source: own.

To better understand how to send commands from the station and how to set the acceptance filter an example is shown:

The ESP32 CAN-WiFi controller is connected to a CAN network running at 500 kbps. Message IDs sent by the active nodes in the network go from 0x50 to 0x69.

1. The user wants to receive all messages running throw the network so the command to be sent is: <START;500>.

2. The user wants to receive messages with ID between 0x50 and 0x59 without considering the RTR bit, so the filter must be set accordingly.

   From ACR0 all bits are relevant, while from ACR1 none is.

   Hexadecimal value: 0x50 → Binary value: 0101 0000

ACR0								ACR1							
IDn10	IDn9	IDn8	IDn7	IDn6	IDn5	IDn4	IDn3	IDn2	IDn1	IDn0	RTR	X	X	X	X
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
AMR0								AMR1							
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Fig. 24 Acceptance Code Register and Acceptance Mask Register, example 2. Source: own.

According to *Fig. 24* the command to be sent is: <START;500;A;0;0;FF>

3. The user only wants to receive request data messages with ID=0x63.

   Hexadecimal value: 0x63 → Binary value: 0110 0011

ACR0								ACR1							
IDn10	IDn9	IDn8	IDn7	IDn6	IDn5	IDn4	IDn3	IDn2	IDn1	IDn0	RTR	X	X	X	X
0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	0
AMR0								AMR1							
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Fig. 25 Acceptance Code Register and Acceptance Mask Register, example 3. Source: own.

According to *Fig. 25* the command to be sent is: <START;500;C;70;0;F>

After correctly receiving the command from the station the task_CAN() is executed. As in the demo code, the CAN_frame is initiated, and the queue created. Next the CAN speed is set to the value previously stored in the global variable BaudRate. Also the CAN_stop() routine is called to ensure that the CAN port is in reset mode. This is necessary to be able to set the filter to single mode, by setting register MOD.B.AFM to 1, and to store the acceptance filter values in their respective registers (see Annexe 6.2, code line 64).

Having initialized the CAN port, the loop begins. In each iteration the queue is checked and if there is a message, this is serial printed and next sent to the connected station. If the message is not delivered correctly the socket is closed, the CAN is stopped and the STA_DISCONNECTED_BIT is set. This causes the tcp_server task to force the ESP32 restart.

After successfully sending a message, the task checks the STOP_CAN_BIT's state and if it is set, the CAN_stop() routine is called and the ESP32 is restarted. STOP_CAN_BIT is set only if the task_STA_CMD() routine receives a stop command from the station.

4.3.6. Second Test

The second set of tests have been made to prove the correct initialization of the Access Point and the correct transmission of the CAN frames sent by the receiving node. Also, the single filter mode has been tested.

On the device connected to the ESP32 Access Point the program used is the *Real Term* [24], which interface allows to easily send commands to a specific port and IP.

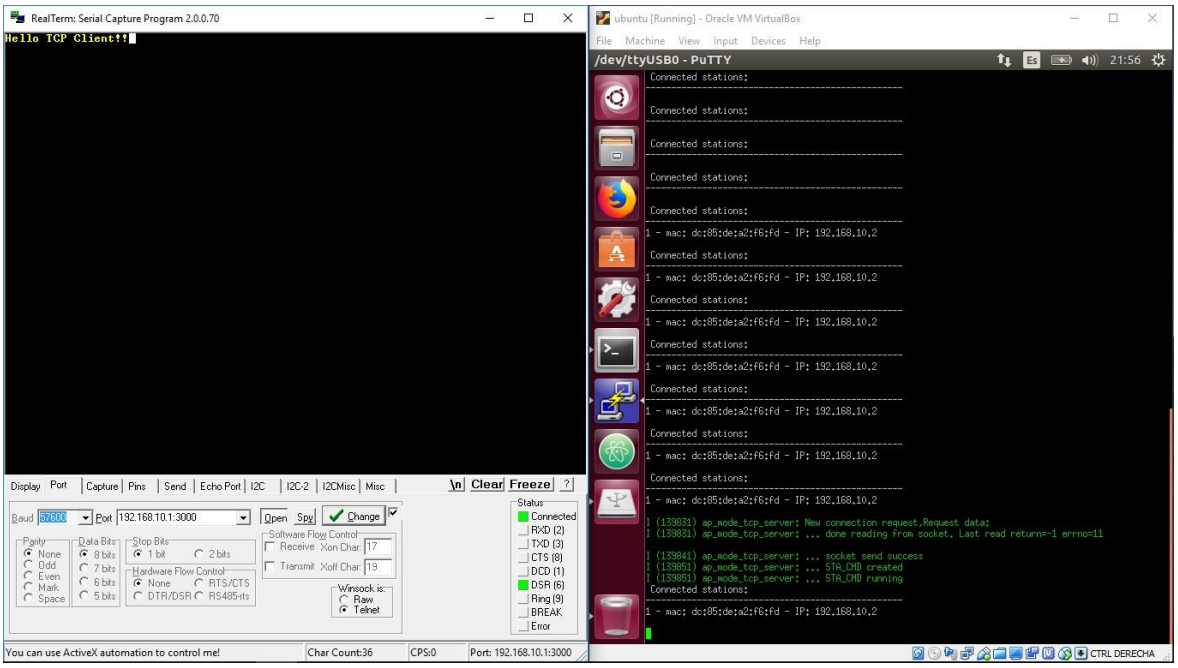

Fig. 26 External Device connection. Source: own.

On the left side of *Fig. 26* the *Real Term* program, executing on the external Device, attempts a connection at the following direction: 192.168.10.1:3000, which the IP of

the ESP32 board and the port the socket is listening to. *Real Term* receives the message *"Hello TCP Client!"* after successfully establishing communication with the ESP32. On the right side of *Fig. 26* the serial output of the ESP32 is showed.

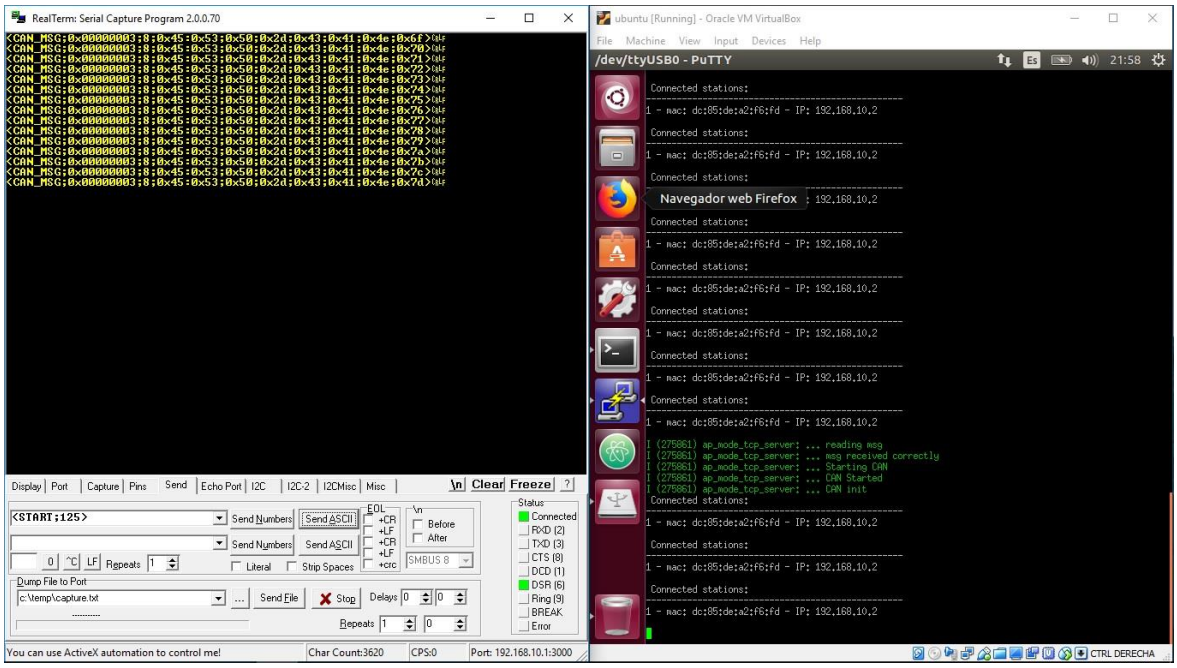

Fig. 27 Start command at 125 Kbps. Source: own.

After successfully connecting with the ESP32 board and sending the command <START;125> from the external device, see left side of *Fig. 27*, the CAN communication starts and all messages are sent to the device, as can be seen by the messages received by *Real Term*.

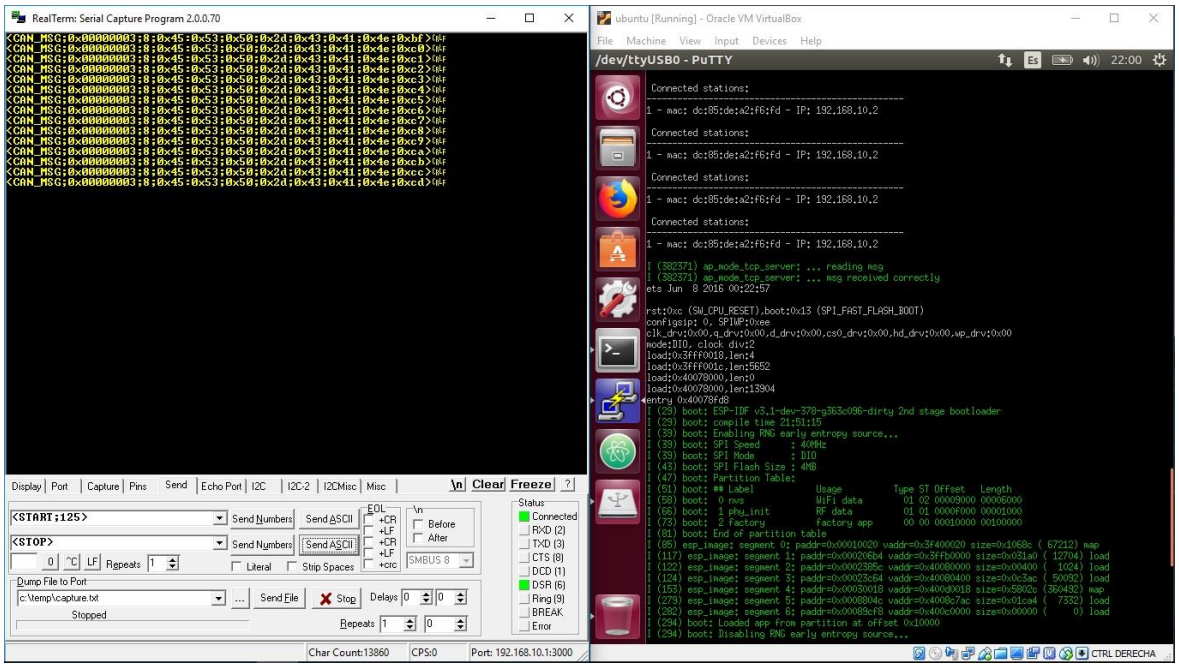

Fig. 28 Stop command. Source: own.

After sending the stop command to the ESP32, CAN communication is ended, and the board restarted. See *Fig. 28*.

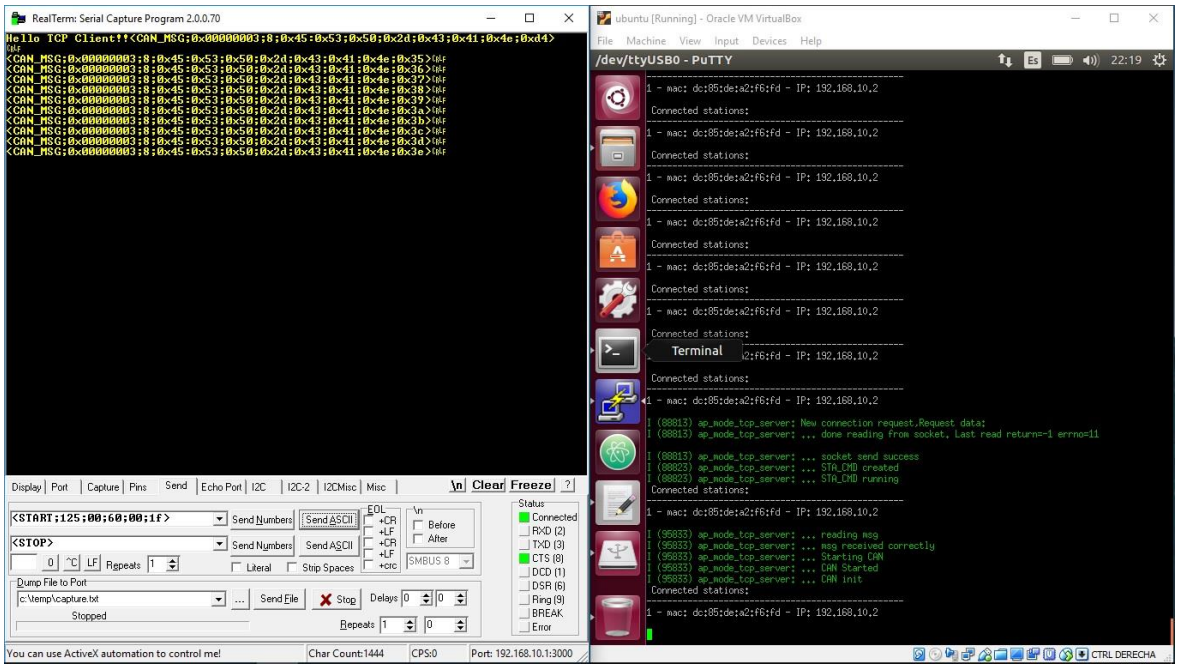

Fig. 29 Start command with filter setting [1]. Source: own.

After restart and successfully connect again to the ESP32, the command

<START;125;00;60;00;1f> is sent to start CAN communication and receive messages from node 0x03. Messages are being received because the transmission node has been set with ID 0x03. If the start command is changed to <START;125;00;40;00;1f>, to accept messages from ID 0x02, no more messages are received. (see *Fig. 30*).

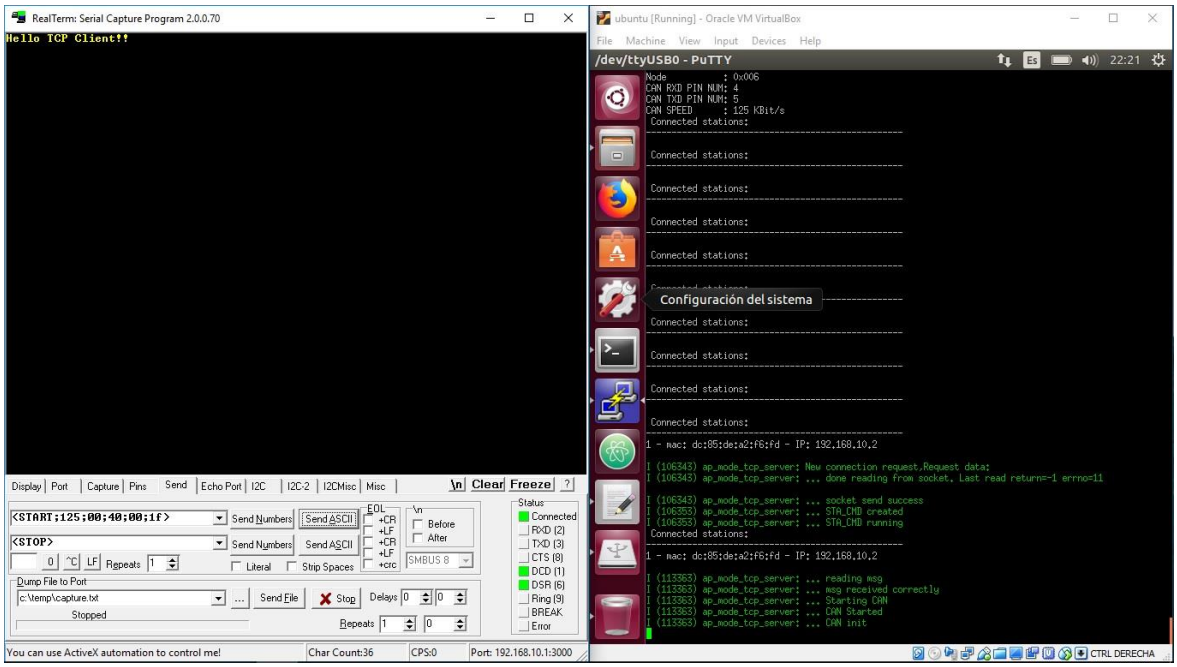

Fig. 30 Start command with filter setting [2]. Source: own.

After this set of tests more bugs have been detected:

- Having terminated the tcp connection with the ESP32, attempting again connection without restarting the ESP32, causes the socket not to initialize correctly and to crash short after. This is why when the tcp server task forces restart if the station connected to it is disconnected.

- After sending the stop command to terminate the CAN communication and right after sending a start command, the CAN messages are not received correctly. This has been a random problem and only effective solution to it has been forcing the ESP32 to restart.

4.3.7. Dual Filter Mode

After testing of the initial WiFi commands and the single filter mode, the final step is to allow

the user to enable the dual filter mode of the CAN port.

As pointed out in section 2.1.2 the CAN port can be used in single filter mode, where 4 of the available Acceptance Filter registers are used, or in dual filter mode where all 8 registers are used. This mode allows the user to filter the message IDs using a two-step filter. Each filter has its own acceptance code and mask defined and it is very useful when wanting to obtain messages with very different IDs.

Modifications to both the source code of the application and the CAN library have been made to correctly set up the filters:

- Application. Task_STA_CMD() has been adapted to receive both settings for the single filter and the dual filter mode. The user operating the external device connected to the ESP32 Access Point, will have to choose among the available starting commands:
    - <START;BAUD_RATE>. It starts CAN communication at BAUD_RATE speed with no filter, so all messages will be sent to the external device.
    - <START;BAUD_RATE;AFM;ACR0;ACR1;AMR0;AMR1>. This command format starts the CAN communication in single filter mode. AFM must be equal to 1 to correctly set the filter. The last four parameters are the same that have been explained in section 4.3.5.
    - <START;BAUD_RATE;AFM;ACR0;ACR1;ACR2;ACR3;AMR0;AMR1;AMR2; AMR3>. This command starts the CAN communication in dual filter mode. AFM must be set to 0 to correctly set both filters. Compared to the previous command 4 more parameters are available: ACR2, ACR3, AMR2 and AMR3. Their function is the same as for the parameters of the first filter.
- CAN library. CAN_init() has been modified to be able to receive and correctly set the AFM register, that defines if port is initiated in single filter mode (AFM =1) or in dual filter mode (AFM = 0), and also all the Acceptance Filter registers, both code and mask.

Next more examples will be illustrated to understand how to use the modified start command:

1. User wants to receive messages only from ID 0x02. Single filter mode will be sufficient for this request:

   Hexadecimal value: 0x02 →Binary value: 0010

ACR0								ACR1							
IDn10	IDn9	IDn8	IDn7	IDn6	IDn5	IDn4	IDn3	IDn2	IDn1	IDn0	RTR	X	X	X	X
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
AMR0								AMR1							
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Fig. 31 Acceptance Code Register and Acceptance Mask Register, example 1. Source: own.

According to *Fig. 31* the command to be sent is: <START;500;1;0;40;0;1F>.

2. User wants to receive messages from ID 0x243 and 0x25B. Dual filter mode will be necessary for this request:

   Hexadecimal value: 0x243 →Binary value: 0010 0100 0011

   Hexadecimal value: 0x25B →Binary value: 0010 0101 1011

ACR0								ACR1							
IDn10	IDn9	IDn8	IDn7	IDn6	IDn5	IDn4	IDn3	IDn2	IDn1	IDn0	RTR	X	X	X	X
0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	0
AMR0								AMR1							
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Fig. 32 Acceptance Code Register and Acceptance Mask Register, filter 1, example 2. Source: own.

ACR2								ACR3							
IDn10	IDn9	IDn8	IDn7	IDn6	IDn5	IDn4	IDn3	IDn2	IDn1	IDn0	RTR	X	X	X	X
0	1	0	0	1	0	1	1	0	1	1	0	0	0	0	0
AMR2								AMR3							
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Fig. 33 Acceptance Code Register and Acceptance Mask Register, filter 2, example 2. Source: own.

According to *Fig. 32* and *Fig. 1* the command to be sent is:
<START;500;0;48;60;4B;60;0;1F;0;1F>.

If single filter mode would have been used and bits corresponding to IDn4 and IDn3

(see Fig. 32) of the mask set to 1, as not to be considered in the filtering, the filter would have let message from ID 0x253 (0010 0101 0011) and ID 0x24B (0010 0100 1011) pass, which was not the goal of the user.

4.3.8. Final Test

Having allowed configuration of the dual filter mode from the starting commands, next and last step is to test its correct functioning.

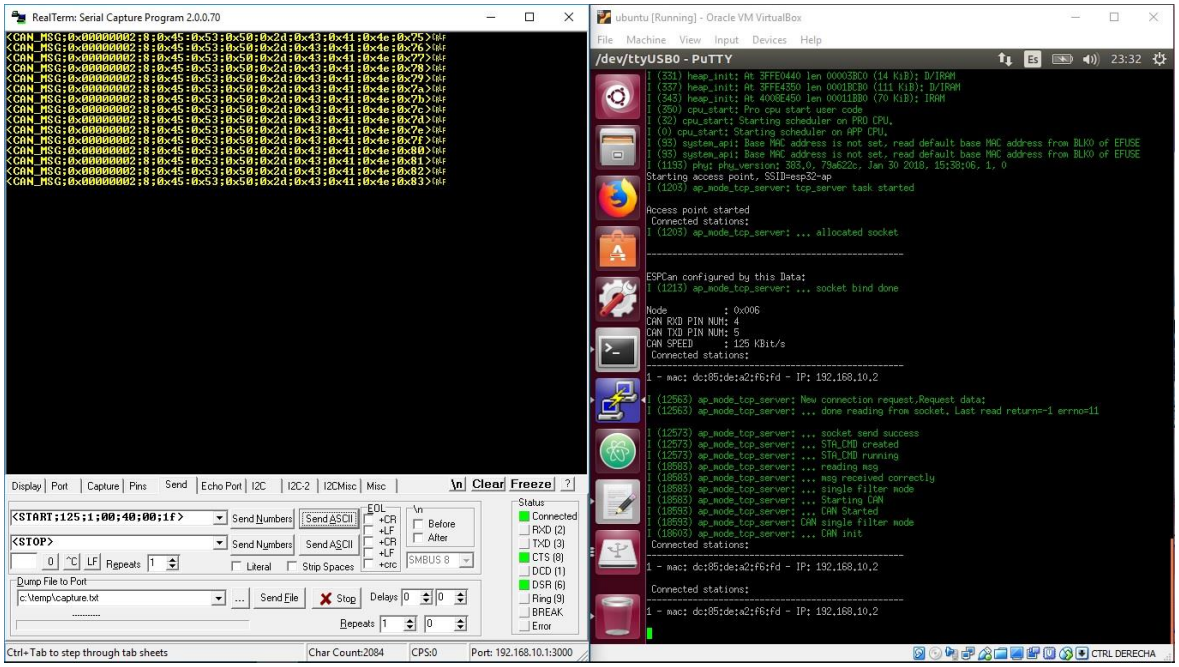

Fig. 34 Start command with single filter setting. Source: own.

Fig. 34 shows the latest start command format <START;125;1;00;40;00;1F> in single filter mode, that filters messages with ID 0x02.

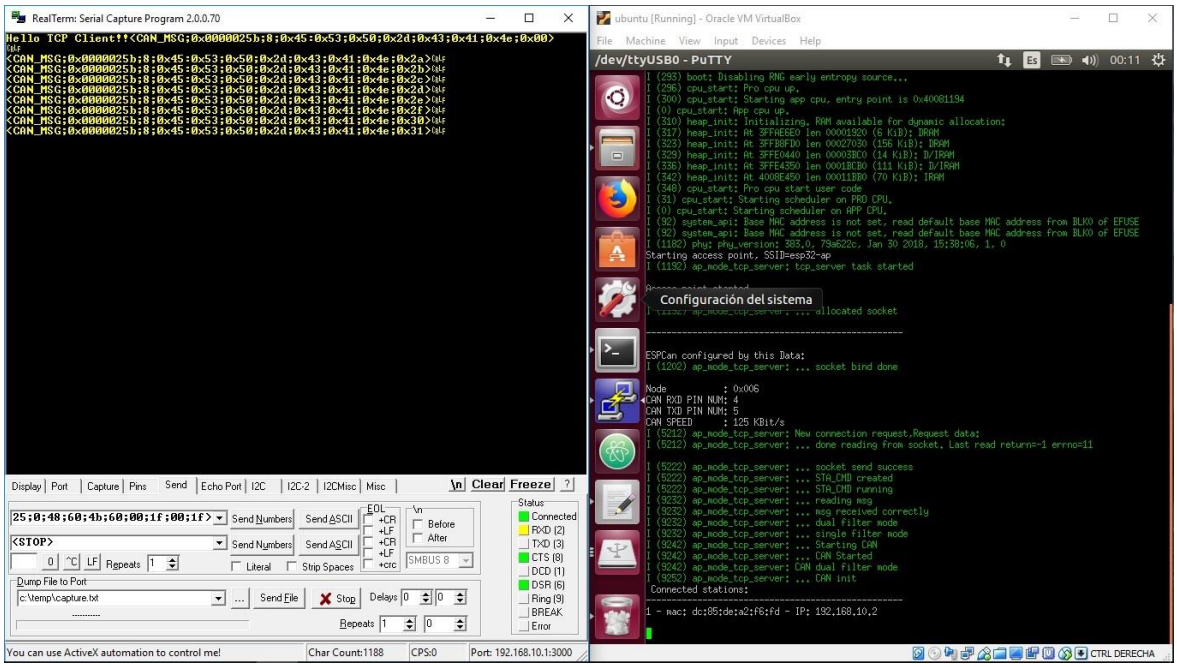

Fig. 35 Start command with dual filter setting [1]. Source: own.

The command to successfully start the CAN communication in dual filter mode is <START;125;0;48;60;4B;60;00;1F;00;1F>, this command is explained in more detail in example 2 of section 4.3.7. This command allows reception from ID 0x243 and 0x25B. In the test showed in Fig. 35 the transmission node was configured with ID 0x25B.

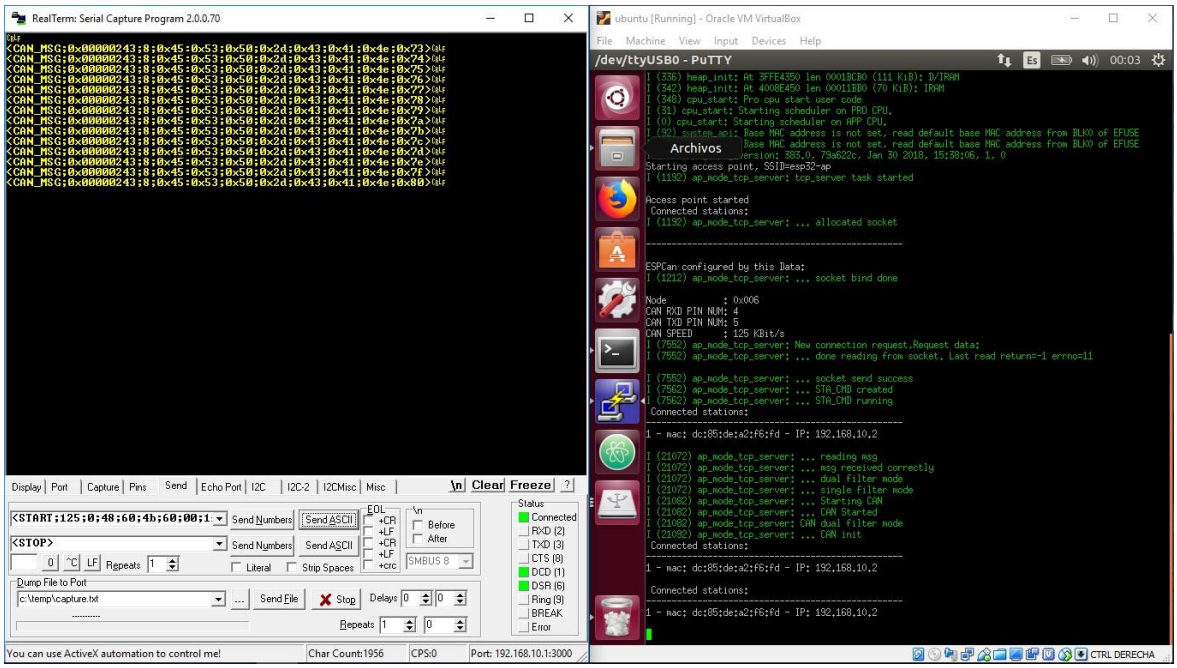

Fig. 36 Start command with dual filter setting [2]. Source: own.

ETSEIB

Fig. 36 shows the same command sent to the ESP32 and messages being received from ID 0x243.

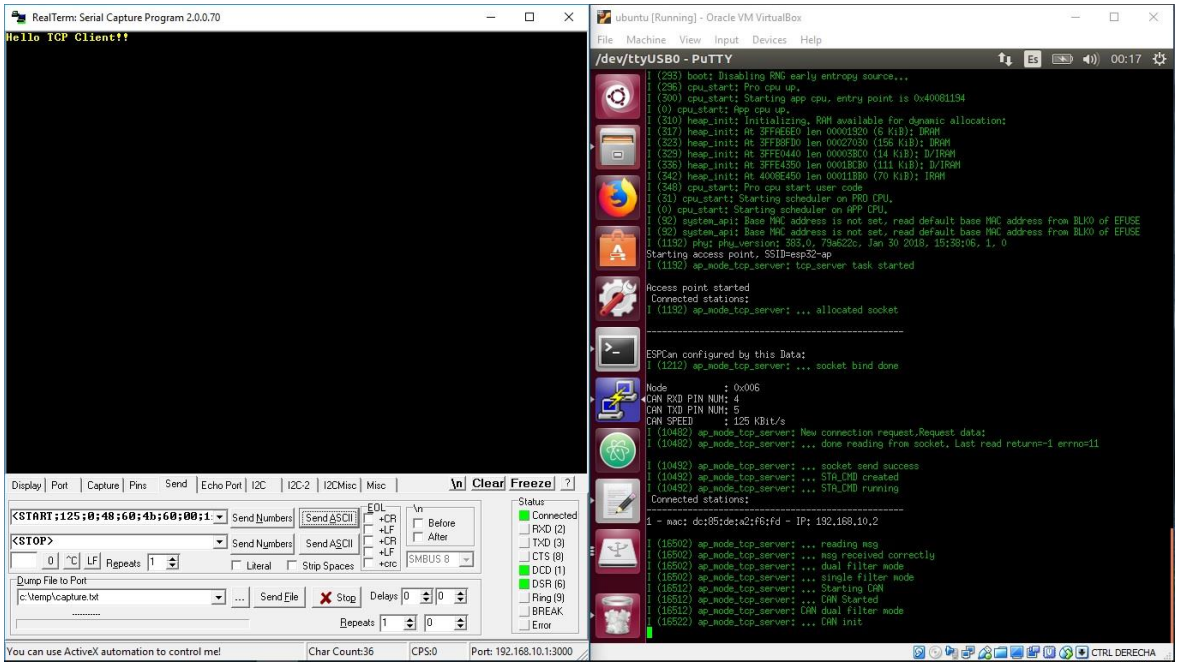

Fig. 37 Start command with dual filter setting [3]. Source: own.

Fig. 37 shows again the same command sent but no message being receive, because the ID of the transmission node has been set to 0x1D, so no message with this ID won't pass any of the two filters set.

5. Environmental Impact

The use of this application and the CAN-WiFi converter will allow faster fail detection in modern cars, which heavily depends on the CAN network to correctly work.

The diagnosis CAN network allows mechanics to obtain information of all the ECUs installed on a car and to detect what causes the failure. The CAN-WiFi converter can be connected to said network and receive information and messages from the ECUs to a laptop, where the mechanic can easily analyse it.

Direct consequences of this are faster and cheaper reparations, reduced contamination, because of the decrease prices charged to final clients.

The use of a CAN network itself considerably decreases the amount of wiring needed inside the car, which again reduces contamination due to less amount of plastic and copper used.

Along with the use of the CAN-WiFi converter comes the emission of electromagnetic waves generated by the ESP32 board, which levels are below the legal limits.

The ESP32 board [3] is ROHS compliant, which means that it is manufactured to limit its environmental footprint. Also it is lead and mercury free.

Conclusions

This project has proven that open source microcontrollers can be used in applications for well-established and standardized industrial protocols, such as the CAN bus protocol.

The project has covered implementation starting from the original CAN library, following to its improvement and adaptation to comply with the objectives set at the beginning and ending in testing the application and its capabilities with a self-made two node network.

Improvements can be done on the work realized, as for example to allow more than one external device to connect to the Access Point and also receive messages from the CAN-WiFi Converter, isolate the cause of desynchronization just after starting CAN communications having stopped it before and eliminate the need of the restart of the board.

Next step following this project is to develop an application for the external device that could help analyse the received data and maybe gain access to a manufacturer CAN library that would allow to understand the content of the messages received. These libraries are usually not published by car manufacturers, so it would not be simple to obtain one. Another solution would be to develop the previously mentioned application, focusing on easing the reverse engineering task of CAN messages.

Of course, the decodification task of the CAN messages should not be one if it concurs against the car manufacturer property and private know-how, and if it is meant to be used to as a hacking tool.

The final thought is that this project is a proof of how easy it is nowadays to get ahold of tools that allow to obtain the most important and critical information necessary for the correct functioning of modern cars. In the wrong hands this information and potential control over somebody else car, can have dangerous and unwanted consequences. This is an important design flaw that hopefully will be soon solved by car manufacturers.

Acknowledgments

The author was able to finish the project thanks to the invaluable help and patience of the Director of this Project, Mr. Juan Manuel Moreno Eguilaz, so he deserves a special thanks and great respect for the work he is doing at the University with all his students and their projects.

6. Annexes

6.1. Demo Source Code

```
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "esp_log.h"
#include "nvs_flash.h"

#include "CAN.h"
#include "CAN_config.h"

//set ESPCAN CONFIG values
 #ifndef CONFIG_ESPCAN
 #error for this demo you must enable and configure ESPCan in menuconfig
 #endif

#ifdef CONFIG_CAN_SPEED_100KBPS
#define CONFIG_SELECTED_CAN_SPEED CAN_SPEED_100KBPS
#endif

#ifdef CONFIG_CAN_SPEED_125KBPS
#define CONFIG_SELECTED_CAN_SPEED CAN_SPEED_125KBPS
#endif

#ifdef CONFIG_CAN_SPEED_250KBPS
#define CONFIG_SELECTED_CAN_SPEED CAN_SPEED_250KBPS
#endif

#ifdef CONFIG_CAN_SPEED_500KBPS
#define CONFIG_SELECTED_CAN_SPEED CAN_SPEED_500KBPS
#endif

#ifdef CONFIG_CAN_SPEED_800KBPS
#define CONFIG_SELECTED_CAN_SPEED CAN_SPEED_800KBPS
#endif

#ifdef CONFIG_CAN_SPEED_1000KBPS
#define CONFIG_SELECTED_CAN_SPEED CAN_SPEED_1000KBPS
#endif

#ifdef CONFIG_CAN_SPEED_USER_KBPS
#define CONFIG_SELECTED_CAN_SPEED CONFIG_CAN_SPEED_USER_KBPS_VAL
#endif


CAN_device_t CAN_cfg = {
   .speed      = CONFIG_SELECTED_CAN_SPEED,   // CAN Node baudrade
   .tx_pin_id    = CONFIG_ESP_CAN_TXD_PIN_NUM,   // CAN TX pin example menuconfif GPIO_NUM_5
```

```
52.       .rx_pin_id     = CONFIG_ESP_CAN_RXD_PIN_NUM,   // CAN RX pin example menuconfig GPIO_NUM_4
53.       .rx_queue    = NULL,                      // FreeRTOS queue for RX frames
54.    };
55.
56.    void  task_CAN( void  *pvParameters ){
57.        (void )pvParameters;
58.
59.        //frame buffer
60.        CAN_frame_t __RX_frame;
61.
62.        //create CAN RX Queue
63.        CAN_cfg.rx_queue = xQueueCreate(10,sizeof (CAN_frame_t));
64.
65.        //start CAN Module
66.        CAN_init(0,0,0xff,0xff);
67.
68.        while  (1){
69.            //receive next CAN frame from queue
70.            if (xQueueReceive(CAN_cfg.rx_queue,&__RX_frame, 3*portTICK_PERIOD_MS)==pdTRUE){
71.             printf("Frame from : 0x%08x, DLC %d \n", __RX_frame.MsgID, __RX_frame.FIR.B.DLC);
72.             printf("D0: 0x%02x, ", __RX_frame.data.u8[0]);
73.             printf("D1: 0x%02x, ", __RX_frame.data.u8[1]);
74.             printf("D2: 0x%02x, ", __RX_frame.data.u8[2]);
75.             printf("D3: 0x%02x, ", __RX_frame.data.u8[3]);
76.             printf("D4: 0x%02x, ", __RX_frame.data.u8[4]);
77.             printf("D5: 0x%02x, ", __RX_frame.data.u8[5]);
78.             printf("D6: 0x%02x, ", __RX_frame.data.u8[6]);
79.             printf("D7: 0x%02x\n", __RX_frame.data.u8[7]);
80.             printf("================================================================================
    ===\n");
81.               //loop back frame
82.               CAN_write_frame(&__RX_frame);
83.            }
84.        }
85.    }
86.
87.    void  task_CAN_TX(void * pvParameters) {
88.
89.       CAN_frame_t __TX_frame;
90.       uint32_t counter = 0;
91.
92.          __TX_frame.MsgID = CONFIG_ESP_CAN_NODE_ITSELF;
93.          __TX_frame.FIR.B.DLC   =  8;
94.          __TX_frame.data.u8[0] = 'E';
95.          __TX_frame.data.u8[1] = 'S';
96.          __TX_frame.data.u8[2] = 'P';
97.          __TX_frame.data.u8[3] = '-';
98.          __TX_frame.data.u8[4] = 'C';
99.          __TX_frame.data.u8[5] = 'A';
100.         __TX_frame.data.u8[6] = 'N';
101.         __TX_frame.data.u8[7] = counter;
102.
103.   while (1) {
104.         __TX_frame.data.u8[7] = counter;
105.         CAN_write_frame(&__TX_frame);
106.         vTaskDelay( 1000 / portTICK_PERIOD_MS);
107.         counter++;
108.         if  (counter >= 256) counter = 0;
109.     }
110.   }
111.
112.   void  app_main(void )
```

```
113.  {
114.
115.      esp_err_t ret = nvs_flash_init();
116.      if  (ret == ESP_ERR_NVS_NO_FREE_PAGES)
117.      {
118.          ESP_ERROR_CHECK(nvs_flash_erase());
119.          ret = nvs_flash_init();
120.      }
121.
122.      #ifdef CONFIG_ESPCAN
123.      printf("ESPCan configured by this Data:\n");
124.      printf("Node           : 0x%03x\n", CONFIG_ESP_CAN_NODE_ITSELF);
125.      printf("CAN RXD PIN NUM: %d\n", CONFIG_ESP_CAN_RXD_PIN_NUM);
126.      printf("CAN TXD PIN NUM: %d\n", CONFIG_ESP_CAN_TXD_PIN_NUM);
127.      printf("CAN SPEED      : %d KBit/s\n", CONFIG_SELECTED_CAN_SPEED);
128.
129.      #ifdef CONFIG_CAN_SPEED_USER_KBPS
130.      printf("kBit/s setting was done by User\n");
131.      #endif
132.
133.      //Create CAN receive task
134.      xTaskCreate(&task_CAN, "CAN", 2048, NULL, 5, NULL);
135.        #ifdef CONFIG_CAN_TEST_SENDING_ENABLED
136.        vTaskDelay( 1000 / portTICK_PERIOD_MS);
137.        xTaskCreate(&task_CAN_TX, "task_CAN_TX", 2048, NULL, 5, NULL);
138.        #endif
139.      #endif
140.
141.      #ifndef CONFIG_ESPCAN
142.      printf("Hello World without ESPCan ;-)\n");
143.      #endif
144.  }
```

6.2. Task_STA_CMD()

```
7.    void  task_STA_CMD( void  *pvParameters )
8.    {
9.    char  recv_buf[64];
10.   char  msg[64];
11.   char  start_char[14];
12.   char  end_char[14];
13.   char  cmd[14];
14.   char  strData[255];
15.   char  *strtokIndx;
16.   bool  taskCANStarted=false  ;
17.
18.      ESP_LOGI(TAG, "... STA_CMD running");
19.      while (1)
20.      {
21.         //wait for new message
22.         bzero(recv_buf, sizeof  (recv_buf));
23.         if (read(CS,recv_buf,sizeof  (recv_buf)-1)>0)
24.         {
25.            ESP_LOGI(TAG, "... reading msg");
26.            bzero(start_char, sizeof  (start_char));
27.            bzero(end_char, sizeof  (end_char));
28.            bzero(cmd,sizeof  (cmd));
29.            bzero(strData,sizeof  (strData));
30.            slice_str(recv_buf,start_char,0,0);
31.            slice_str(recv_buf,end_char,strlen(recv_buf)-1,strlen(recv_buf)-1);
32.            if ((strcmp(start_char,START_CHAR)==0)&(strcmp(end_char,END_CHAR)==0))
33.            {
34.               ESP_LOGI(TAG, "... msg received correctly");
35.               slice_str(recv_buf,msg,1,strlen(recv_buf)-2);
36.               strtokIndx=strtok(msg,";");
37.               strcpy(cmd,strtokIndx);
38.               if ((strcmp(cmd,START)==0)&!taskCANStarted)
39.               {
40.                  strtokIndx=strtok(NULL,";");
41.                  if (strcmp(strtokIndx,"NULL")==0) continue  ;
42.                  else
43.                  {
44.                     strcpy(strData,strtokIndx);
45.                     BaudRate=atoi(strData);
46.                     strtokIndx=strtok(NULL,";");
47.                     if (strtokIndx!=NULL)
48.                     {
49.                        strcpy(Hex1,strtokIndx);
50.                        strtokIndx=strtok(NULL,";");
51.                        strcpy(Hex2,strtokIndx);
52.                        strtokIndx=strtok(NULL,";");
53.                        strcpy(Hex3,strtokIndx);
54.                        strtokIndx=strtok(NULL,";");
55.                        strcpy(Hex4,strtokIndx);
56.                     }
57.                     else
58.                     {
59.                        strcpy(Hex1,"0");
60.                        strcpy(Hex2,"0");
61.                        strcpy(Hex3,"ff");
62.                        strcpy(Hex4,"ff");
```

```
63.                     }
64.                     ESP_LOGI(TAG, "... Starting CAN");
65.                     taskCANStarted=true ;
66.                     xTaskCreate(&task_CAN, "CAN", 4096, NULL, 5, &TaskHandle_CAN); /
    /START task_CAN
67.                     ESP_LOGI(TAG, "... CAN Started");
68.                 }
69.             }
70.             else  if ((strcmp(cmd,STOP)==0)&taskCANStarted)
71.             {
72.                 xEventGroupSetBits(wifi_event_group,STOP_CAN_BIT);
73.                 taskCANStarted=false ;
74.             }
75.         }
76.     }
77.     vTaskDelay(500 / portTICK_PERIOD_MS);
78.   }
79. }
```

1.1. Modified task_CAN() routine

```
1.  void  task_CAN(void  *parameter){
2.
3.  char  message[500];
4.  uint32_t code1;
5.  uint32_t code2;
6.  uint32_t mask1;
7.  uint32_t mask2;
8.
9.      CAN_stop();
10.     EventBits_t evg_bits;
11.     //frame buffer
12.     CAN_frame_t __TCP_frame;
13.     //create CAN RX Queue
14.     CAN_cfg.rx_queue = xQueueCreate(10,sizeof (CAN_frame_t));
15.     CAN_cfg.speed = BaudRate;
16.
17.     //start CAN Module
18.     code1=strtoul(Hex1,NULL,16);
19.     code2=strtoul(Hex2,NULL,16);
20.     mask1=strtoul(Hex3,NULL,16);
21.     mask2=strtoul(Hex4,NULL,16);
22.     CAN_init(code1,code2,mask1,mask2);
23.     ESP_LOGI(TAG, "... CAN init");
24.     while (1)
25.     {
26.         if (xQueueReceive(CAN_cfg.rx_queue,&__TCP_frame, 3*portTICK_PERIOD_MS)==pdT
    RUE)
27.         {
28.            bzero(message, sizeof (message));
29.            sprintf(message,"<CAN_MSG;0x%08x;%d;0x%02x:0x%02x;0x%02x;0x%02x;0x%02x;
    0x%02x;0x%02x;0x%02x;0x%02x>\r\n",
30.             __TCP_frame.MsgID,__TCP_frame.FIR.B.DLC,__TCP_frame.data.u8[0],__TCP_f
    rame.data.u8[1],__TCP_frame.data.u8[2],
31.             __TCP_frame.data.u8[3],__TCP_frame.data.u8[4],__TCP_frame.data.u8[5],_
    _TCP_frame.data.u8[6],__TCP_frame.data.u8[7]);
32.             if ( write(CS , message , strlen(message)) < 0)
33.             {
34.                 ESP_LOGE(TAG, "... Send failed \n");
35.                 close(S);
36.                 xEventGroupSetBits(wifi_event_group,STA_DISCONNECTED_BIT);
37.                 CAN_stop();
38.                 vTaskDelay(4000 / portTICK_PERIOD_MS);
39.                 break ;
40.             }
41.          }
42.          evg_bits=xEventGroupGetBits(wifi_event_group);
43.          if (evg_bits&STOP_CAN_BIT)
44.          {
45.             xEventGroupClearBits(wifi_event_group,STOP_CAN_BIT);
46.             CAN_stop();
47.             esp_restart();
48.             vTaskDelete(NULL);
49.          }
50.          vTaskDelay( 200 / portTICK_PERIOD_MS);
51.      }
52. }
```

1.1. Modified CAN_init() routine

```
1.    int  CAN_init(uint32_t code1,uint32_t code2, uint32_t mask1,uint32_t mask2){
2.
3.        //Time quantum
4.        double  __tq;
5.        //MODULE_CAN->MOD.B.RM = 1;
6.        //enable module
7.        DPORT_SET_PERI_REG_MASK(DPORT_PERIP_CLK_EN_REG, DPORT_CAN_CLK_EN);
8.        DPORT_CLEAR_PERI_REG_MASK(DPORT_PERIP_RST_EN_REG, DPORT_CAN_RST);
9.
10.       //configure TX pin
11.       gpio_set_level(CAN_cfg.tx_pin_id, 1);
12.       gpio_set_direction(CAN_cfg.tx_pin_id,GPIO_MODE_OUTPUT);
13.       gpio_matrix_out(CAN_cfg.tx_pin_id,CAN_TX_IDX,0,0);
14.       gpio_pad_select_gpio(CAN_cfg.tx_pin_id);
15.
16.       //configure RX pin
17.       gpio_set_direction(CAN_cfg.rx_pin_id,GPIO_MODE_INPUT);
18.       gpio_matrix_in(CAN_cfg.rx_pin_id,CAN_RX_IDX,0);
19.       gpio_pad_select_gpio(CAN_cfg.rx_pin_id);
20.
21.       //set to PELICAN mode
22.       MODULE_CAN->CDR.B.CAN_M=0x1;
23.
24.       //synchronization jump width is the same for all baud rates
25.       MODULE_CAN->BTR0.B.SJW       =0x1;
26.
27.       //TSEG2 is the same for all baud rates
28.       MODULE_CAN->BTR1.B.TSEG2 =0x1;
29.
30.       //select time quantum and set TSEG1
31.       switch (CAN_cfg.speed){
32.           case CAN_SPEED_1000KBPS:
33.               MODULE_CAN->BTR1.B.TSEG1 =0x4;
34.               __tq = 0.125;
35.               break;
36.
37.           case CAN_SPEED_800KBPS:
38.               MODULE_CAN->BTR1.B.TSEG1 =0x6;
39.               __tq = 0.125;
40.               break ;
41.           default  :
42.               MODULE_CAN->BTR1.B.TSEG1 =0xc;
43.               __tq = ((float  )1000/CAN_cfg.speed) / 16;
44.       }
45.
46.       //set baud rate prescaler
47.       MODULE_CAN->BTR0.B.BRP=(uint8_t)round((((APB_CLK_FREQ * __tq) / 2) -
      1)/1000000)-1;
48.
49.       /* Set sampling
50.        * 1 -
      > triple; the bus is sampled three times; recommended for low/medium speed buses
         (class A and B) where filtering spikes on the bus line is beneficial
51.        * 0 -
      > single; the bus is sampled once; recommended for high speed buses (SAE class C)*
      /
52.       MODULE_CAN->BTR1.B.SAM   =0x1;
53.
```

```
54.       //enable all interrupts
55.       MODULE_CAN->IER.U = 0xff;
56.
57.       //set acceptance filter
58.       MODULE_CAN->MOD.B.AFM = 1;
59.       MODULE_CAN->MBX_CTRL.ACC.CODE[0] = code1;
60.       MODULE_CAN->MBX_CTRL.ACC.CODE[1] = code2;
61.       MODULE_CAN->MBX_CTRL.ACC.CODE[2] = 0;
62.       MODULE_CAN->MBX_CTRL.ACC.CODE[3] = 0;
63.       MODULE_CAN->MBX_CTRL.ACC.MASK[0] = mask1;
64.       MODULE_CAN->MBX_CTRL.ACC.MASK[1] = mask2;
65.       MODULE_CAN->MBX_CTRL.ACC.MASK[2] = 0xff;
66.       MODULE_CAN->MBX_CTRL.ACC.MASK[3] = 0xff;
67.
68.       //set to normal mode
69.       MODULE_CAN->OCR.B.OCMODE=__CAN_OC_NOM;
70.
71.       //clear error counters
72.       MODULE_CAN->TXERR.U = 0;
73.       MODULE_CAN->RXERR.U = 0;
74.       (void )MODULE_CAN->ECC;
75.
76.       //clear interrupt flags
77.       (void )MODULE_CAN->IR.U;
78.
79.       //install CAN ISR
80.       esp_intr_alloc(ETS_CAN_INTR_SOURCE,0,CAN_isr,NULL,NULL);
81.
82.       //Showtime. Release Reset Mode.
83.       MODULE_CAN->MOD.B.RM = 0;
84.
85.       return  0;
86.   }
```

8. References

[1] Espressif Systems, *ESP32 Datasheet Version 2.1.* espressif.com. 2018.

[2] Espressif ESP32 Forum. esp32.com.

[3] *Sparkfun ESP32 Thing.* sparkfun.com.

[4] Weller, Tomer*. ESP32 - first steps.* blob.tomerweller.com. January 8th, 2017.

[5] *Scheduling.* Freertos.com.

[6] *ESP-IDF Programming Guide*. esp-idf.readthedocs.io.

[7] Philips*. SJA 1000 Stand-alone CAN Controller Datasheet.* nxp.com. January 4th, 2000.

[8] Kolban, Neil. *Kolban's Book on ESP32*. leanpub.com. April 1st, 2018.

[9] *History of the automobile.* en.wikipedia.org.

[10] *History of CAN technology*. can-cia.org

[11] *Mercedes-Benz S-Class W 140*. mercedes-benz.com. February 23rd, 2016.

[12] *Learning Module CAN*. elearning.vector.com

[13] Texas Instruments, *3.3-V CAN TRANSCEVEIRS*, revised June 2002.

[14] *SN65HVD230 CAN Board.* waveshare.com/wiki, revised December 20th, 2016.

[15] *A CAN driver for the ESP32. Barth Development.* barth-dev.de.

[16] *Queues, Mutexes, Semaphores... freertos.org.*

[17] *ESP32 (6) – How to connect to a wifi network.* Author: Luca Dentella, lucadentella.it, January 1st, 2017.

[18] *MSYS2* on Github. github.com/msys2.

[19] *Ubuntu – Open Source Software Operating System. ubuntu.com/*

[20] *Oracle VM VirtualBox. virtualbox.org.*

[21] *Install Ubuntu on Oracle VirtualBox. linus.nci.nih.gov/bdge/installUbuntu.html.*

[22] *IPv4.* Wikipedia Article. en.wikipedia.org/wiki/IPv4.

[23] *ESP-IDF: TCP Server on ESP32.* Author: Sankar Cheppali, September 20th, 2017. icircuit.net.

[24] *Real Term. realterm.sourceforge.io*