AUTOMOTIVE CONNECTIVITY

Design of a CAN line for testing

Authors:
Rosario La Rocca
Jacopo Ferretti

Professor:
Giovanni Pau

December 13, 2021

MOTORVEHICLE UNIVERISITY OF EMILIA ROMAGNA

Automotive Connectivity M

CAN bus protocol project

Abstract

The following project will cover the CAN bus protocol implementation using some demoboards from STmicroelectronics and an Homemade CAN sniffer designed and provided from a friend of us.

The aim of the project is to train ourself in a practical work, and this report is a sort of step by step guide for CAN beginners.

A big thank you to the professor Pau that gave us the opportunity to develop this project for his exam.

Contents

Chapter 1

CAN Bus

1.1 Introduction

Controller Area Network, better knows as "CAN", is a widely used communication protocol in vehicle application. The development started in 1983 at BOSCH with the goal to replace the complex wiring harness of a vehicle with a two-wire bus.
The huge success mainly comes from these advantages:

- **Low cost**: reduce the number of wires used inside the vehicle.
- **Robust**: in case of failure of subsystems and electromagnetic interface.
- **Efficient**: CAN messages are prioritized via IDs.
- **Flexible**: possibility to modify and include additional nodes.

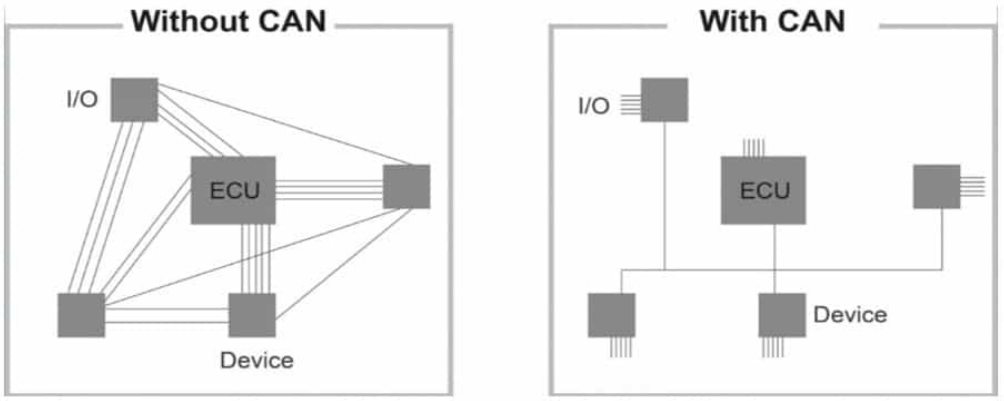


FIGURE 1.1: *Wire Harness with and without CAN*

1.2 Basics

A CAN messages can be transmitted by any node when the bus is free. Each message is received by all the nodes (including the node that sent the message, allowing errors monitoring).
If two or more nodes attempt to transmit simultaneously, the process of arbitration take place and the message with the higher priority will be send to the line in advance (priority is set with the ID of the arbitration field inside the CAN message).

Each node on the network reads the identifier of a message and determines if the message needs to be ignored or processed.

A CAN node is composed of three basic parts: a processor, a network controller and a transceiver. In some hardware (like the one that we used for the project) the controller is integrated in the processor, reducing the read/write time from the processor to the controller, further decreasing the load on the processor and increasing overall system performance. Integrating the CAN controller into the processor also saves board space, simplifies board layout, and reduces total chip cost.

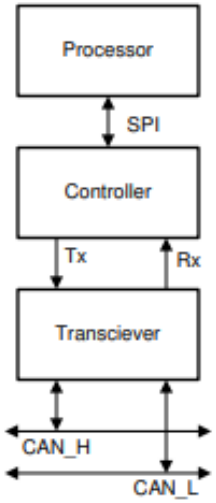


FIGURE 1.2: *Structure of a CAN Node*

CAN communication protocol use only three level of the ISO/OSI layers.
Level 1 (physical layer) comprise the bus line and the transceiver, Level 2 (Data link) the CAN controller and level 7 (application) is the layer of the application software implemented in a microcontroller or a DSP.

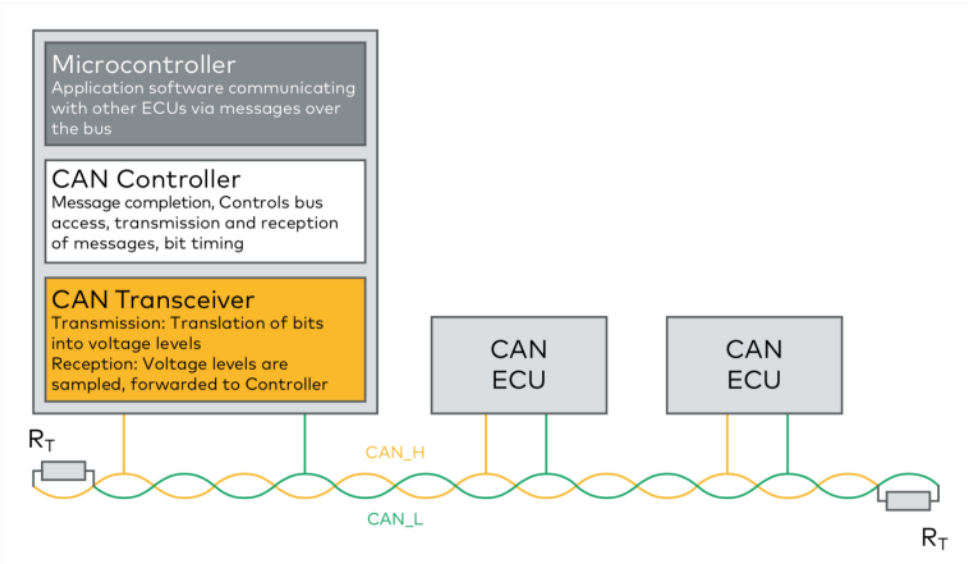


FIGURE 1.3: *CAN line example*

1.2.1 CAN Physical layer

From the physical point of view the CAN bus is simply composed by two twisted wires (CAN-H and CAN-L) with network terminations of 120Ω. Each node is electrically connected in parallel and must follow the CAN interface,that consist in a CAN controller and a CAN transceiver.

In our project the line is simply implemented with jumper wires, a breadboard and for the terminations are used two through hole resistances of 120Ω each.

CAN transceiver

The role of the transceiver is to drive and detect data to and from the line bus. This type of devices are the interface between the CAN controller and the physical bus.
From the controller to the bus convert the logic signal to the differential one transmitted over the line.
In the opposite direction take the differential voltage signal, rejects the common-mode noise, and outputs a logic signal for the CAN controller.

FIGURE 1.4: *TJA1050 CAN transceiver*

The transceiver distinguishes between two bus logic states, **dominant** and **recessive**. In case of 5V bus lines the voltage level for the recessive state is 2.5V for each CAN-H and CAN-L (in bit logic is 1), while for the dominant state CAN-H is equal to 3.5V and CAN-L to 1.5V (in bit logic is 0).

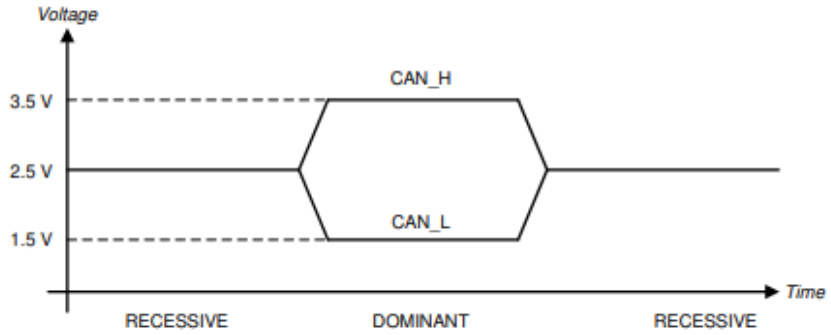


FIGURE 1.5: *CAN bus states*

From the TJA1050 datasheet is possible to obtain information about the internal block diagram and the pin configuration (fig. 1.6).

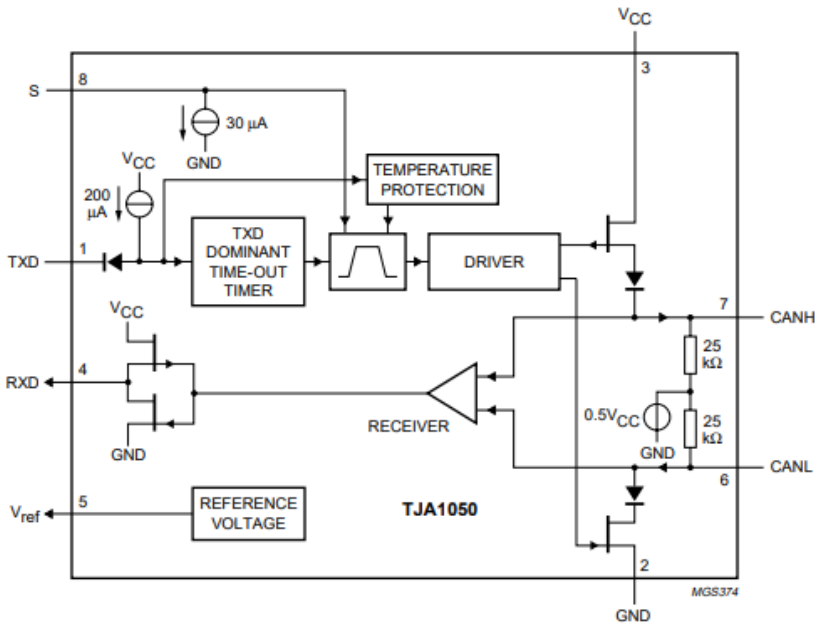


FIGURE 1.6: *TJA1050 block diagram*

This specific transceiver is primarily intended for high-speed automotive applications with a max baud rate of 1 Mbaud.
Inside the device also some protection are present, a current-limiting circuit protection protects the transmitter from accidental short circuit in the supply voltage. Another protection is related to the max temperature, switching off the transmitter if the junction temperature exceeds a value of 165 °C.

SYMBOL	PIN	DESCRIPTION
TXD	1	transmit data input; reads in data from the CAN controller to the bus line drivers
GND	2	ground
V_{CC}	3	supply voltage
RXD	4	receive data output; reads out data from the bus lines to the CAN controller
V_{ref}	5	reference voltage output
CANL	6	LOW-level CAN bus line
CANH	7	HIGH-level CAN bus line
S	8	select input for high-speed mode or silent mode

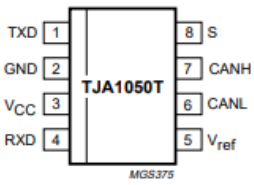


Fig.2 Pin configuration.

FIGURE 1.7: *TJA1050 pin configuration*

1.2.2 CAN controller

The CAN controller is present in each node and it is part of the Data link layer in the ISO/OSI stack.
The main job of this component is the message completion, control bus access, transmission of messages and bit timing. This component can be thought as the part of

the CAN node that processes all the information to and from the CAN bus.

As previously told, the chosen project hardware (ST nucle F302R8) has a CAN controller built on it.

The CAN controller must be configured to reconcile the data rate and timing on the bus with the hardware oscillator used for the controller.

Controller configuration: Bit timing

To achieve a robust network with a reliable timing and synchronization between nodes, the system must be able to tolerate the propagation delay with the chosen data rate and CAN-controller clock.
To compensate the propagation delays, specific parameters related to timing and synchronization must be set up for CAN controller. Configuring the controller is preferable to set the variables that dictate the bit time used by the controller.
The hardware selection of the oscillator, and the software configuration of the BRP (baud rate prescaler) and number of TQ (time quanta) per bit time set the data rate.

Each bit of a CAN frame is composed by four segments, each segment is composed by an integer number of Time Quanta, that are the smallest discrete timing resolution used by a CAN node.

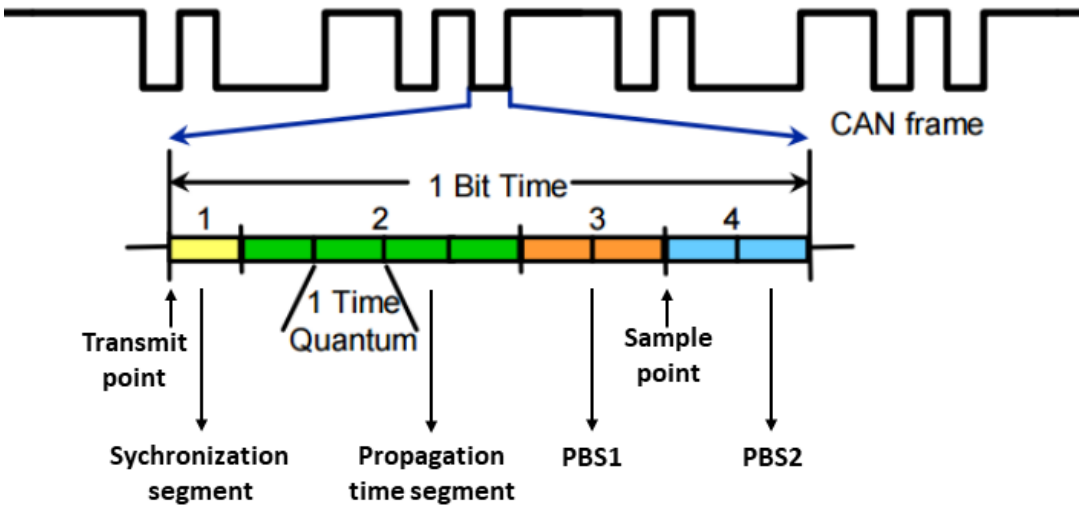


FIGURE 1.8: *Bit time segments*

Each segment has a specified function:

- **Synchronization segment**: the length is always 1 TQ and it is used to synchronize the various bus nodes
- **Propagation time segment**: the length is programmable from 1 to 8 TQ and it allow signal propagation. It is necessary to compensate for signal propagation delays on the bus line.

- **Phase buffer segment 1**: the length is programmable (from 1 to 8 TQ) and it is the only segment that can be lengthened during resynchronization. At the end of this segment the bus state is sampled (the sample time can be tuned).
- **Phase buffer segment 2**:the length is programmable (from 1 to 8 TQ) and can be shortened during resynchronization.

It is important to specify that some CAN modules combine the propagation time segment and the PSB1.

Usually it is not possible to define just the bit rate because CAN bus is asynchronous, what we can do is to play with the number of segments and time quanta to achieve the wanted bit rate. Instead of doing it randomly, we can use an online tool from *can-wiki*[1] that does all the calculation for us.
This website is what we are using for our project, as can be seen in section 3.3.2

1.2.3 CAN message

Each bit field has its own function and some of the info must be filled manually in the software used for our project. For this reason a short overview of a CAN frame is necessary.

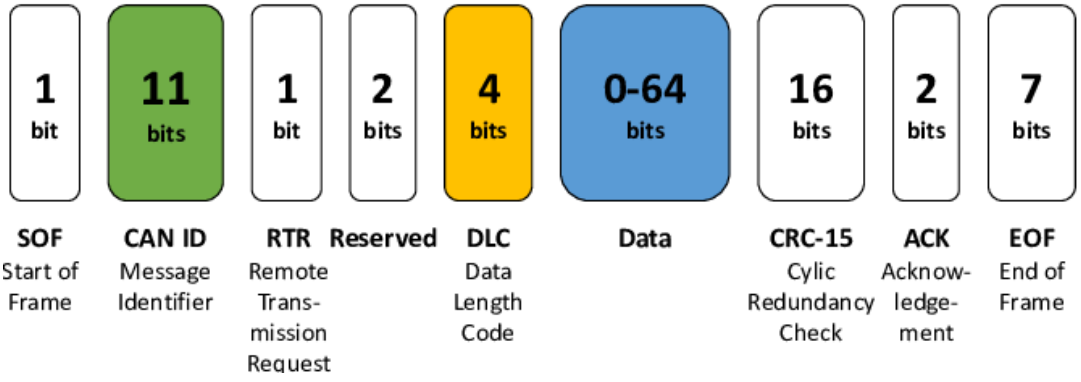


FIGURE 1.9: *CAN data frame*

The data frame could be standard or extended, the first one is composed by the following bits:

- **SOF**: A single dominant bit marks the start of the message.
- **Identifier**: 11bit ID establish the priority of the message, the lower the value, the higher the priority.
- **RTR**:(Remote Transmission Request) the bit is dominant when the information is required from another node.
- **Reserved**
- **DLC**: 4 bit length contains the number of bytes of data transmitted.
- **Data**: up to 64 bits may be transmitted.
- **CRC**: used for error detection.

- **ACK**: check the integrity of the data.
- **EOF**: 7 bit field that mark the end of a CAN frame.

Chapter 2

Hardware

In this chapter a description of the main Hardware components used in the project is delivered. Then, the Electrical schematic is reported to give a better comprehension of how the used hardware is connected to each other.

2.1 HWtronics CAN sniffer

The CAN sniffer is one the most important device of our project, with that it is possible to sense the line. connecting the sniffer to a PC, via USB port, we are able to read the CAN message over the bus line.
The sniffer used in this project is made by HWtronics[2], a small company created by a very talented electronic engineer. Reporting some details from the manufacture, the sniffer is based on STM32, it features a software selectable 120Ω CAN-Termination, it is self-powered from USB and communicate to the pc trough the Virtual COM Port interface. Moreover it has two 10-pin headers that can be used in order to connect external sensors.
The used device, not only sense the line but it is able to send CAN message, acting like a node of the CAN line. To sense and send messages we use an open source software called BusMaster, that will be explained in details in the next chapter.

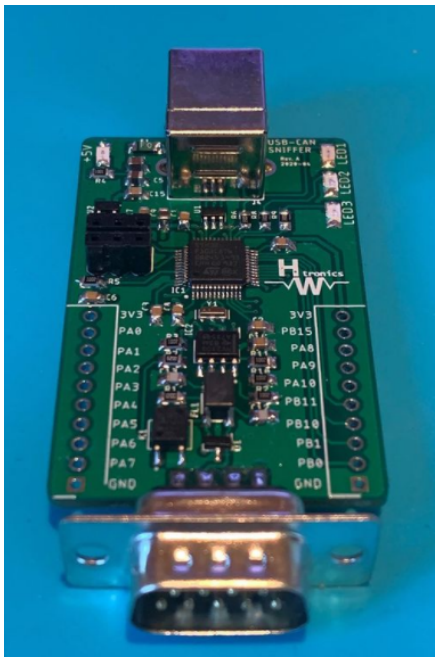
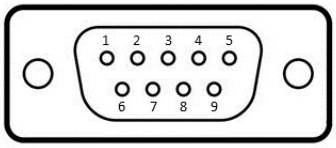


FIGURE 2.1: *HWtronics CAN sniffer*

Regarding the project, it is important to highlight the pin configuration of the DB9 connector that represent the interface between the sniffer and the bus line.



CAN-L	4
CAN-H	8
GND	3

FIGURE 2.2: *DB9 pin configuration*

2.2 ST nucleo F302R8

To implement the nodes of our CAN line we use two ST nucleo F302R8[3]. For the aim of our project the chosen demoboard has a built-in CAN controller than can be connected directly to the TX-RX pin of the CAN transceiver. Then, connecting the power pin (5V and GND) to the transceiver, the necessary connection are done.
We suggest to refer to the electrical schematic shown fig. 2.4. In fig. 2.3, a picture of the ST demoboard.

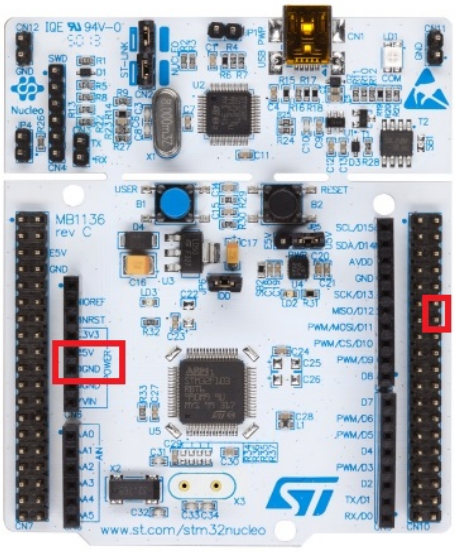


FIGURE 2.3: *ST nucle F302R8*

The used pin in our project are the one highlighted in red in the above picture, those are:

- 5V
- GND
- CAN1_RD CN10 PA_11
- CAN1_TD CN10 PA_12

2.3 TJA1050 CAN transceiver

The microcontroller we decided to use is not able to communicate via CAN by themselves, so it was necessary to use a *CAN transceiver* (as already discussed in section 1.2.1).
The transceiver we used is a board based on the TJA1050 (fig. 1.4), because it was easy to use and just working out of the box (no pin configuration needed).
The only concern was that the chip needed 5V to work, but our microcontroller works with 3.3V logic.
After a research on the microcontroller datasheet [4], we were able to verify that the two pins used (PA11, PA12) were 5V tolerant, so the hardware should be compatible.

2.4 Electrical connection schematic

Using the free software KiCAD, an electric schematic reporting the hardware connection was designed.

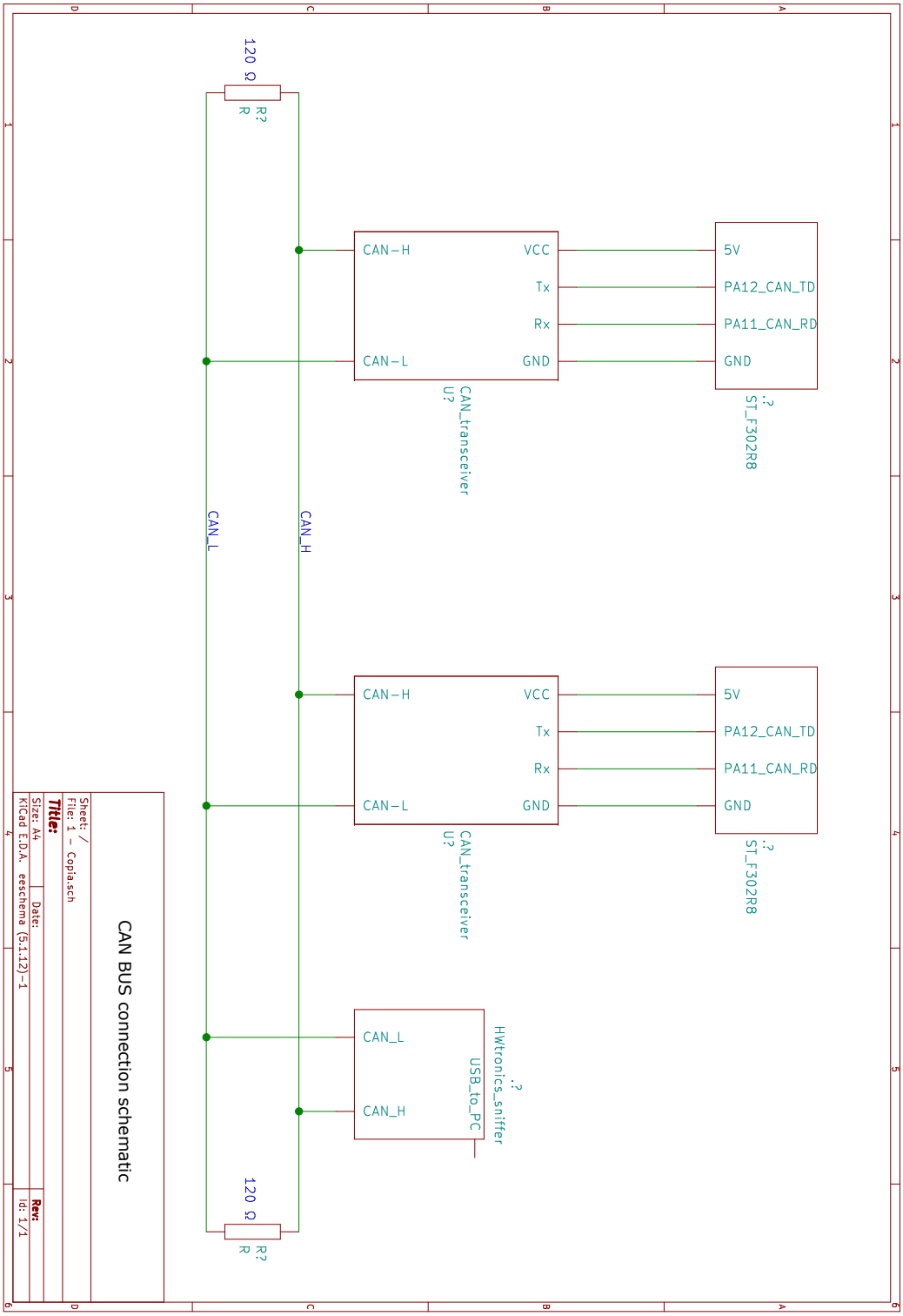


FIGURE 2.4: *CAN line: Electrical schematic*

Chapter 3

Software implementation

3.1 BusMaster

BusMaster[5] is an open source software developed by ETAS and it is useful to design, monitor, analyze and simulate a CAN network.
In our project we use this software connecting the CAN sniffer to it, allowing us to sense and send CAN messages.
In the following subsections a description of the busmaster fuction that we use during the project development.

Sniffer connection

In this section is described the steps to connect the used sniffer to BusMaster. Once opened the software, in the top left there is the hardware configuration menu.
Below, the steps useful to connect the HWtronic sniffer:

1. Open the driver selection menu
2. Select VScom CAN-API
3. Go to advanced section to set other parameter
4. Click to "search for Devices on COM-Ports"
5. Set the Baudrate data
6. Click to the Connect button to start the connection

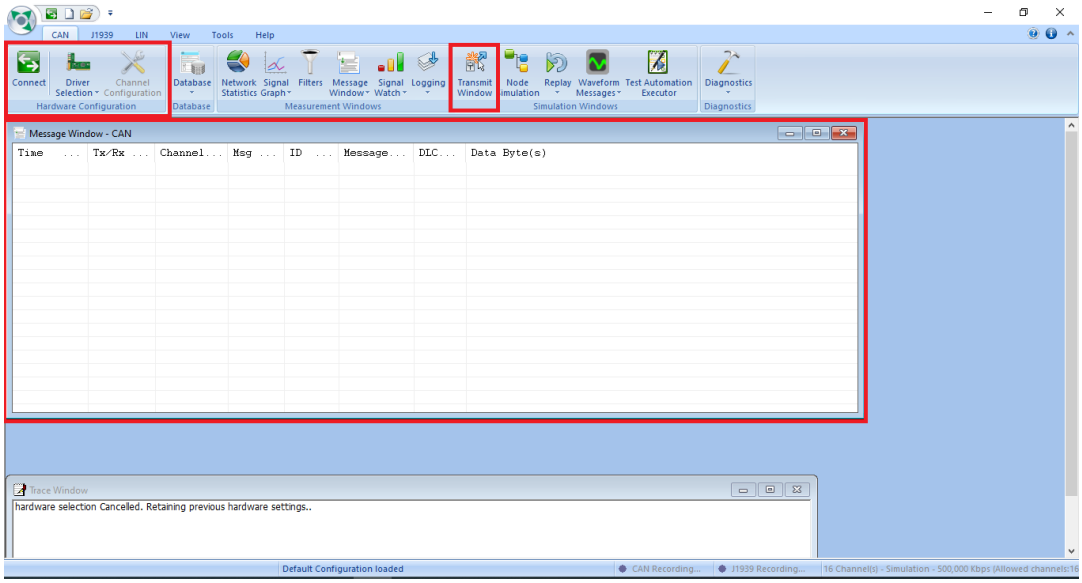


FIGURE 3.1: *BusMaster interface*

Message and Transmit window

In the message window are displayed the transmitted and received (Tx,Rx) CAN messages with all the information related to each packet. This section is very useful to understand what is happening inside the bus line.

Another useful tool is the "transmit window" where is possible to build in few second a simple message that can be transmitted to test the line.

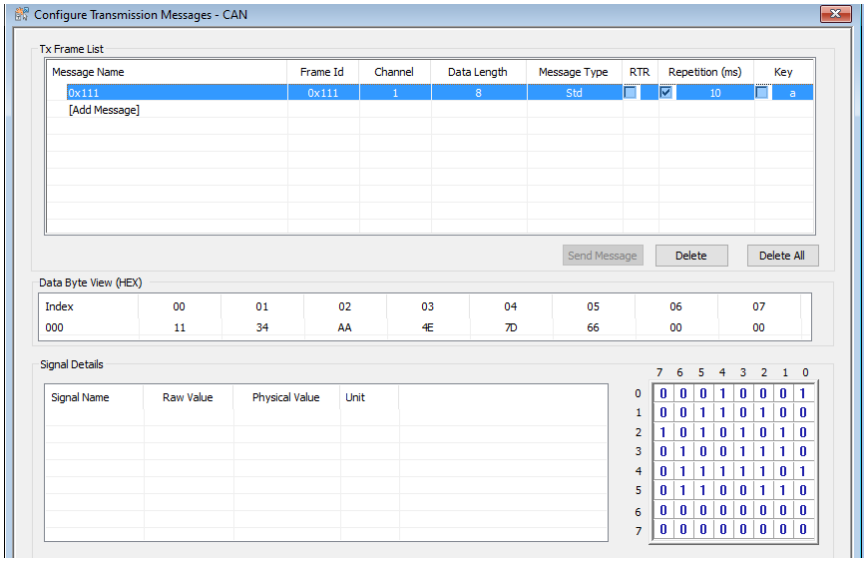


FIGURE 3.2: *BusMaster transmit window*

Network statistics

The network statistics, in fig. 3.3, shows some parameter useful to monitor the bus line. It is possible to see the number of messages (standard or extended), both Tx and Rx, the load percentage of the line (peak and average) and the number of errors.

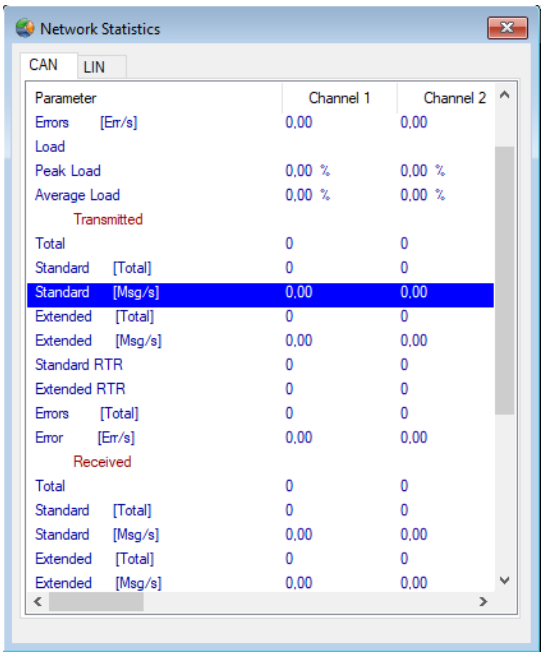


FIGURE 3.3: *BusMaster Network statistics*

Database editor

Busmaster has an inside tool that creates a database of messages using the "CAN DBF Editor" inside the Tools Menu. This function give us the possibility to set a list of messages that can be transmitted from the transmit window.
To create a database the following steps are needed:

1. Open the tool menu.
2. Create a new database clicking the CAN DBF editor and the new button.
3. From the database editor is possible to create a new message, setting the name, the ID, the message type and length.
4. From the message and signal information menu of each messages is possible to set new signals and insert details of them.

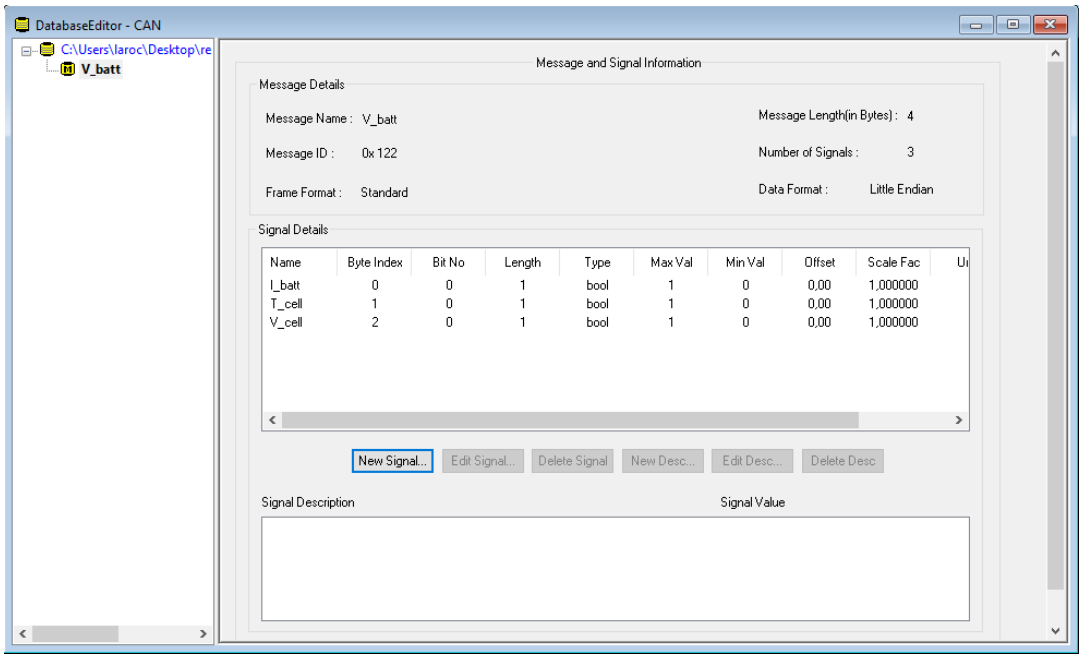


FIGURE 3.4: *Database editor page*

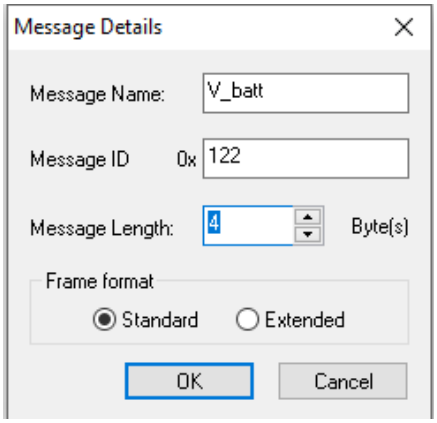


FIGURE 3.5: *Database message details*

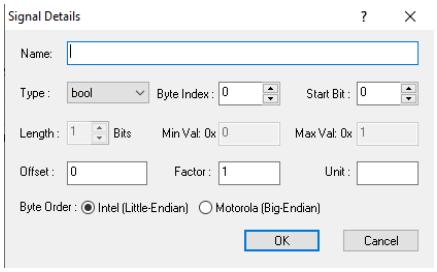


FIGURE 3.6: *Database signal editor page*

Once a database is created, in order to use it and transmit the contained information, it is necessary to associate the database in Busmaster by clicking:

CAN menu → Database → Associate

After this operation all the messages inside the database will be available inside the transmit window.

3.1.1 BMS node database

For a testing purpose we develop a database that simulate a BMS (Battery Management System) node of a battery pack. This is done to show the potentiality of Busmaster during the design and test phases of a CAN line.

A battery management system (BMS) manages a battery pack, by protecting the battery from operating outside its safe operating area avoiding faster ageing and risk of

fire and explosion.
A BMS must be able to measure:

- Voltage [V]
- Current [A]
- Temperature [°C]

Other functions of a BMS are the Balancing, SoC and SoH estimation.

After this short description, the following table report the data inserted in a Database to simulate a BMS. In the last chapter some test using them are reported.

Message Details				**Signal Details**					
Name	**ID**	**Length (Bytes)**	**Type**	**Name**	**Type**	**Byte index**	**Length (Bits)**	**Factor**	**Physical value**
Vcells	0x111	8	std	V1,V2, V3,V4	uint	0,2,4,6	16	0.001	[0 65.535]
Tcells	0x113	2	std	T	int	0	16	0.01	[-327.68 327.67]
Current	0x112	2	std	I	int	0	16	0.1	[-3276.8 3276.7]

3.2 Coding enviroment

The decision of using a ST NUCLEO board came both for an economical reason, and for it's widespread use trough the internet community (a lot of resources and guides are available).
Another strong reason was that a mature and easy to use softwares are available to be used, like *STM32CubeIDE*[6]: a complete and free to use development tool that makes the programming and deployment on the board incredibly easy.

3.2.1 Used language

The most widely used languages in embedded programming are `C` and `C++`, and our MCU is not an exception.
Before to start we tried to use `C++` with *Mbed OS*[7], but the approach with that Real time OS was too simplistic, hiding too many information, so it was not very useful for our purpose.
Therefore, we decided to stick with the more classic `C` language and the Hardware Abstraction Layer provided by the chip manufacturer (ST), by doing so we were able to control every aspect of our board.

3.2.2 Development environment

The development environment used was STM32CubeIDE[6]: it integrates the IDE where we can develop the code, deploy it on the hardware, use the debugger, and set any microcontroller option trough an easy to use GUI.
Another added value is that this software officially supports any boards manufactured by STMicroelectronics, making the initialization process straight forward.

3.3 Preliminary configuration with STM32CubeMX GUI

Opening the IDE, we need to create a new STM32 project by clicking:

> *File → New → STM32 Project*

A new window will welcome us (fig. 3.7), here we are able to search our board and initialize the pins with the integrated GUI.

> *board selector → search for "NUCLEO-F302R8" → Click on the board → next*

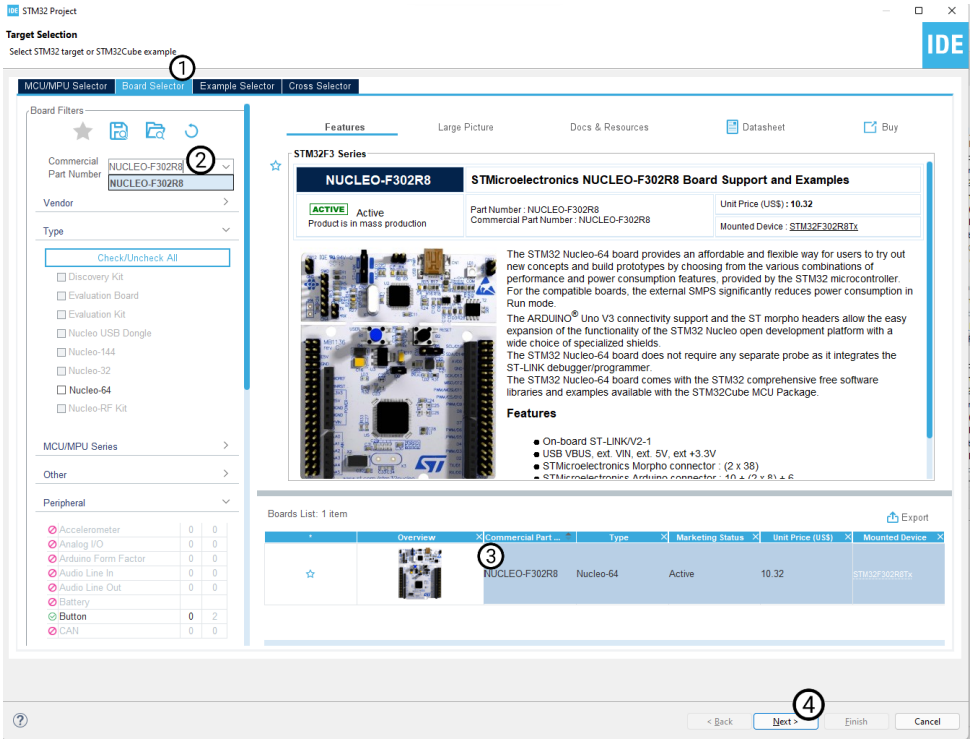


FIGURE 3.7: *New STM32 project*

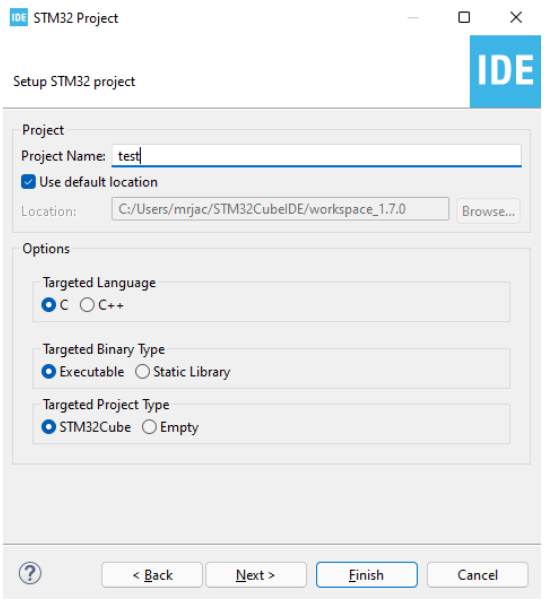


FIGURE 3.8: *Insert project name*

After inserting the project name (fig. 3.8), you will be asked to initialize all peripherals with default mode: just click yes and a new environment will be shown.

3.3.1 Pin Configuration with STM32CubeMX

What you can see now is the pin configuration GUI (previously it was called STM32CubeMX, now embedded in STM32CubeIDE).

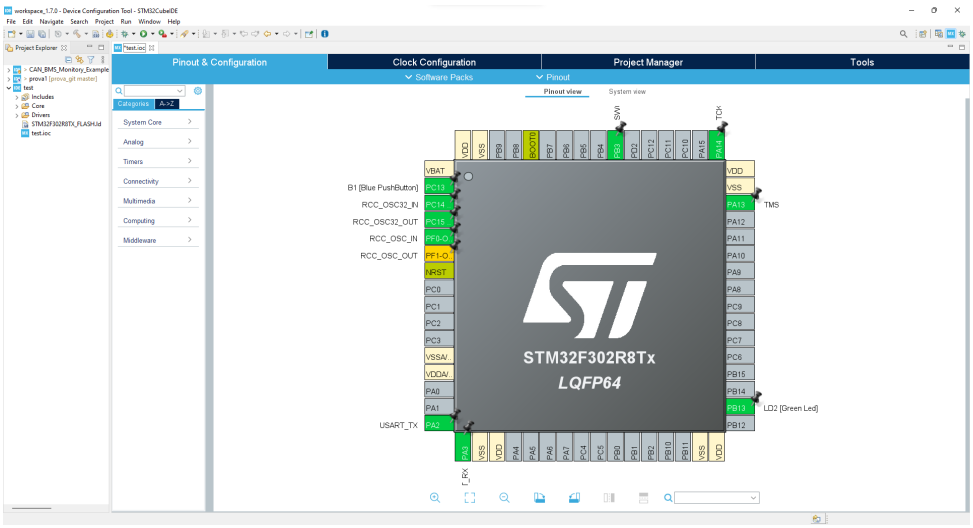


FIGURE 3.9: *MXCube default configuration*

In fig. 3.9 it is the default initial configuration page, from here you are able to modify many parameters using a nice GUI, this allow us to avoid wasting time in reading the MCU documentation and writing every option manually in the code.
What we want to do now is to initialize the can bus peripheral:

connectivity → CAN → check the box "activate"

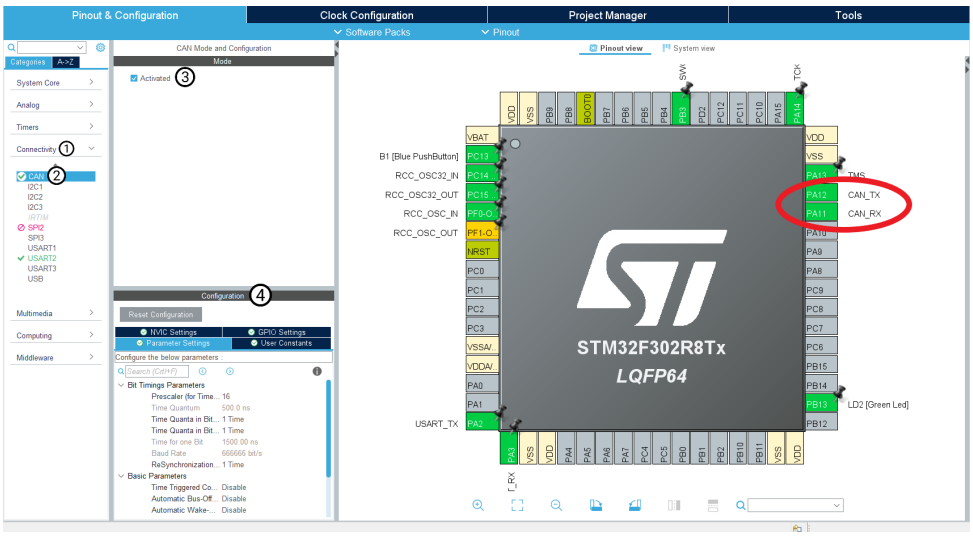


FIGURE 3.10: *MXCube activation of can bus*

As you can see in fig. 3.10, now the two pin `PA11` and `PA12` are coloured in green, `CAN_TX` and `CAN_RX` appeared on the right. This mean that those two pin are enabled and are assigned to the CAN BUS Peripheral.
The next step is to configure it the way we want, it can be obviously done from the configuration panel paired on the bottom left.

3.3.2 CAN Timing

What we want now is to configure our CAN BUS to communicate at 1000 kbit/s.
We decided this high bit-rate to be as compatible as possible with the can sniffer used (it is possible to change it obviously, but with our tests everything works fine, no reason to change it).
As we have already explained in section 1.2.2, CAN protocol is an asynchronous protocol, so we need to specify the number of different segments and time quanta.
From the configuration panel we are able to put those number, and the bit rate is automatically calculated... but we want to do the opposite!
Starting from the bit rate instead, we need to insert the following parameters inside the *can-wiki*[1] website:

1. Microcontroller used (ST Microelectronics bxCAN).
2. Clock rate of the can peripheral (32 MHz).
3. Sample point (keep standard 87.5 %).
4. SJW (keep standard 1).
5. bit rate (1000 kbit/s).

In fig. 3.11 you can see how the website should look like.
At the bottom it is possible to see the configurations to be used in order to obtain a 1000 kbit/s bit rate.

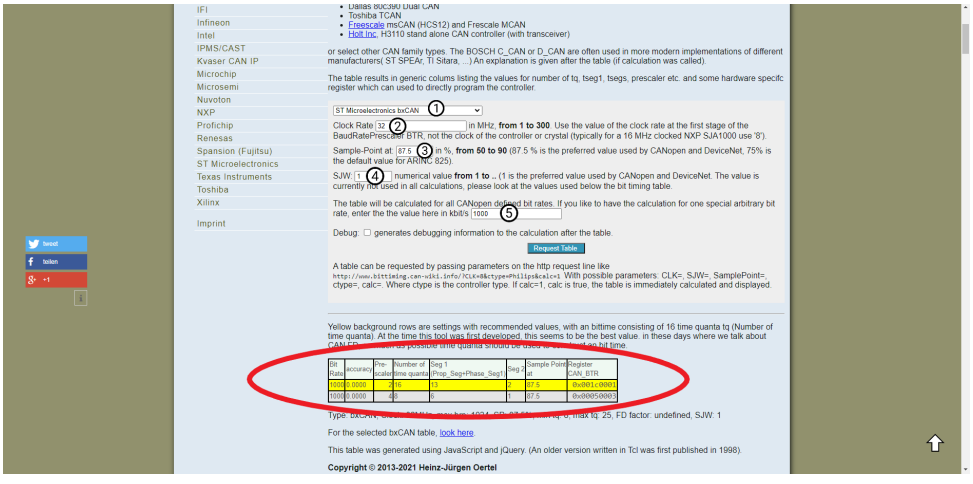


FIGURE 3.11: *CAN-wiki[1] configuration*

The Clock rate of the can peripheral can be found in the STM32CubeMX GUI by clicking:

Clock Configuration → *APB1 peripheral clocks* (MHz)

It is possible to change many different clock speed as you can see in fig. 3.12, but we don't want to go too deep in that field, so keep everything standard, or just copy our configuration in fig. 3.12.

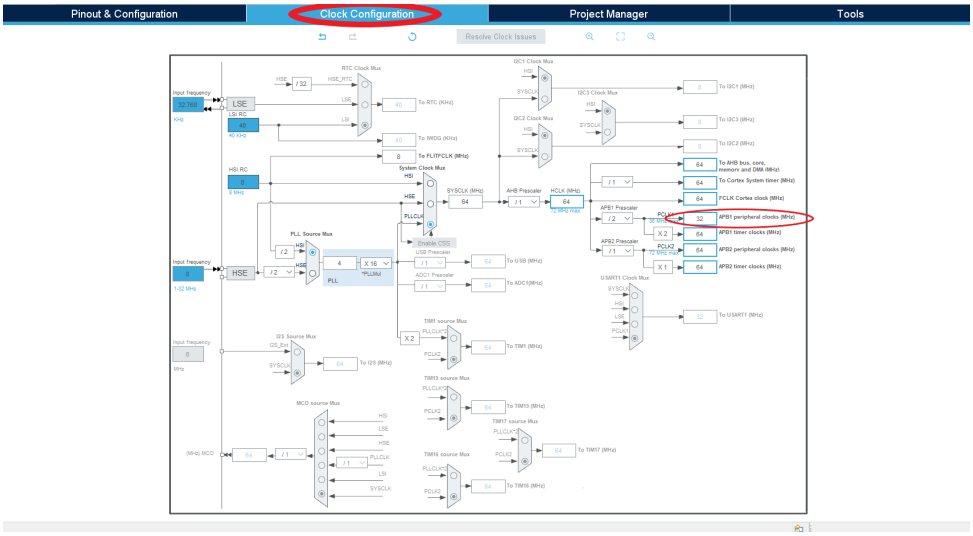


FIGURE 3.12: *System clock configuration*

3.3.3 CAN Peripheral Configuration

Now it is possible to properly configure the CAN peripheral.
Starting from the Timing configuration found trough can-wiki (fig. 3.11), go back to STM32CubeMX GUI, go back to *Pinout & configuration* tab and:

connectivity → *CAN* → *Parameter Settings*

Here we need to set the settings with recommended values from can-wiki, that are the Yellow background rows you can see in fig. 3.11.

1. `Prescaler (for Time Quantum) = 2`
2. `Time Quanta in Bit Segment 1 = 13`
3. `Time Quanta in Bit Segment 2 = 2`
4. `Operating Mode = Normal`

Everything should look like fig. 3.13.
The operating mode is set to normal to communicate properly with the other nodes of the bus, for test purpose it is possible to use *loopback* mode: it allows to send packets over the can line, and "auto-trigger" the interrupt reading same packets sent.

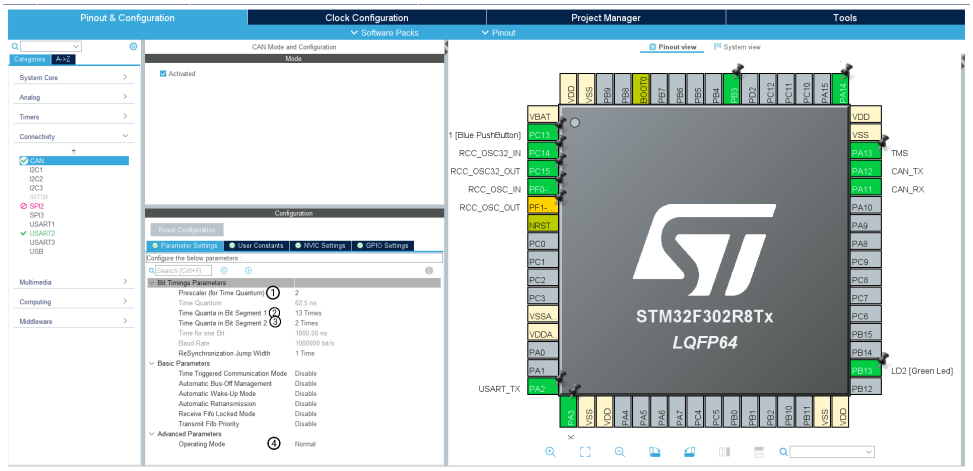


FIGURE 3.13: *CAN Peripheral configuration*

3.3.4 CAN Interrupts Configuration

Another thing we want to set up are the interrupts: we won't go much in details here, but it is important to set them correctly. First of all we need to enable them:

connectivity → CAN → NVIC Settings

Here we need to turn on the following tick, as can also seen in fig. 3.14:

1. `CAN RX0 and USB low priority interrupts`
2. `CAN RX1 interrupt`

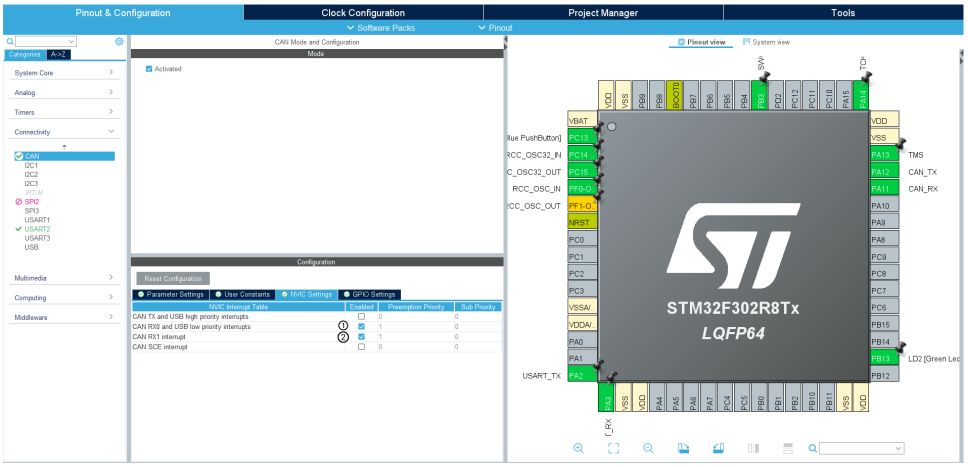


FIGURE 3.14: *CAN interrupt configuration*

Then, it is also important to set the priority of those interrupt: this is mandatory to prioritize timing interrupts (otherwise we are going to experience an infinite loop when using `HAL_Delay`). to do this, we need to go to set more bits to be used for priority groups:

system core → NVIC → set Priority Group to "4 bits..."

now you need to set to 1 (or higher) any enabled interrupts (blue tick). Note that the CAN interrupts are automatically enabled (after the previous passage), and it is possible to see more tick than fig. 3.15: it is normal, and I suggest to set priority 1 (or higher) on any other enabled interrupts.
Low value set → high priority (0 is the maximum priority)

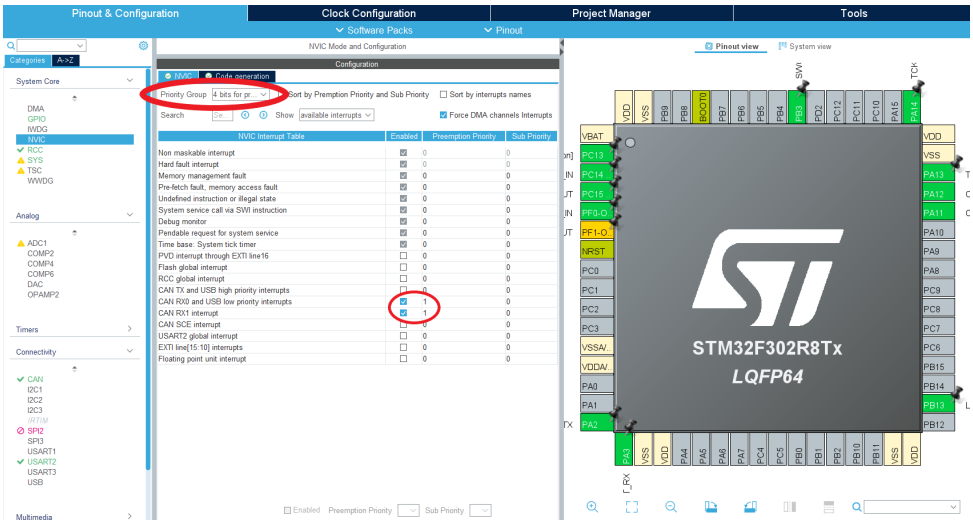


FIGURE 3.15: *CAN interrupt priority configuration*

User button configurations

The CAN configuration over this GUI is finished, now we are going to configure the user button (blue button on the board) to be used with our code.
The process is very similar to what we have done before, so we will go a bit faster here: First step is to turn enable the user button pin:

> *press "PC13" pin of the mcu from the image → press "GPIO_EXTI13"*

Then we need to enable the interrupt:

> *system core → GPIO → Press on "PC13" → select "External Interrupt Mode with Falling edge trigger detection"*

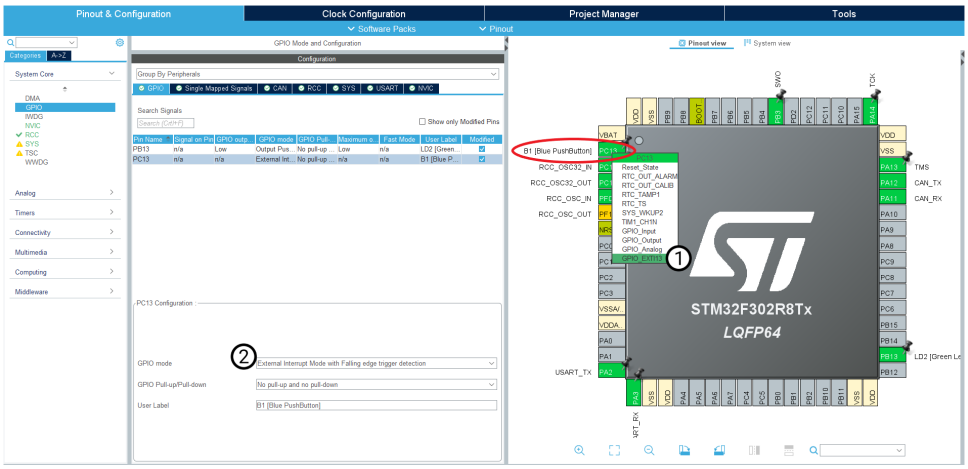


FIGURE 3.16: *User button configuration*

Make sure that the following interrupts are enabled in the NVIC settings, as can be seen in fig. 3.17, and set all their preemption priority to 1 or higher:

- RCC global interrupt
- CAN RX0 and USB high priority interrupts
- CAN RX1 interrupt
- EXTI line [15:10]

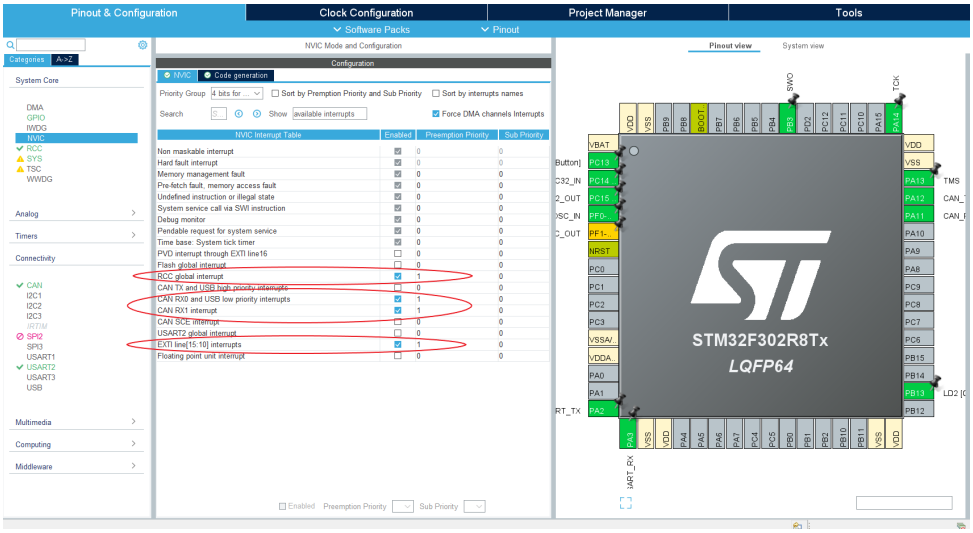


FIGURE 3.17: *NVIC settings for can and user button*

Now the configuration trough the GUI is done, we can save the file from the symbol on the top left. Then you are asked to generate generate the code: press yes and open the c perspective, now we are ready to write some `C` code.

3.4 Preliminary settings on workspace

Now we can see our workspace and some auto generated code. We remember to those who are not familiar with STM32CubeMX, that every code you write needs to be inserted inside the user code space, otherwise the lines are going to be overwritten after the coding generation.

```
   // this line will be overwritten after code generation
/* USER CODE BEGIN ... */
   // here i can write code
/* USER CODE END ... */
```

That being said, let's set some preliminary things, like the external libraries we are going to use:

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
```

Then another important step we need to configure, is the capability to print floating numbers. You can configure it by following those steps:

Press the project name on the workspace → then press file → properties

A new window will be opened, like you can see in fig. 3.18.

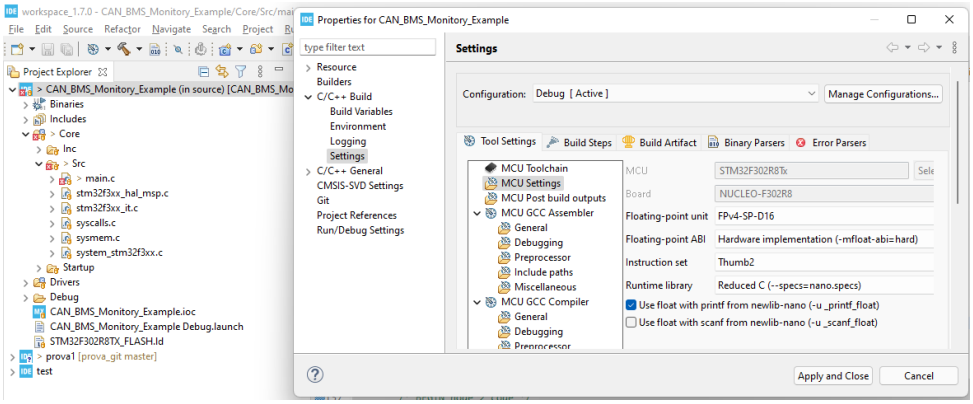


FIGURE 3.18: *Solve the print float number problem*

Then go to mcu settings:

C/C++ Build → Settings → MCU Settings

Here you need to turn on the check on *Use float with printf from newlib-nano* and Apply the settings.
Once we have done this, you are ready to set some more CAN options.

3.5 CAN Structure used

In order to use the CAN bus, we need to set few more things.
Those configurations are found under:

workspace → Core → Src → `main.c`

We need to define some structures to store the wanted configuration (that will be shown and explained later on):

```
CAN_TxHeaderTypeDef  TxHeader;   // CAN Tx message header structure
CAN_RxHeaderTypeDef  RxHeader;   // CAN Rx message header structure

uint32_t  TxMailbox;             // Mailbox where Tx message is stored

uint8_t  TxData[8];              // Vector that will contain data to be transmitted
uint8_t  RxData[8];              // Vector that will contain data received
uint8_t  count = 1;              // Simple counter used for debugging
```

3.5.1 `TxHeader`

`TxHeader` is the structure used to store the information about the header when sending a packet from our node.

3.5.2 `RxHeader`

`RxHeader` is a structure that will be used in incoming packets to know their header (so know the data lenght, the ID...)

3.5.3 TxMailbox

The CAN peripheral has 3 different mailboxes that can be used, those are like 3 buffers for outgoing messages. We will be using one mailbox, and this will be enough.

3.5.4 TxData[8]

`TxData[8]` is an 8 byte array where we will store the outgoing data to be transmitted (it is used like a temporary vector)

3.5.5 RxData[8]

`TxData[8]` is another 8 byte array where we will store the incoming payload from a received packet (again, it is used like a temporary vector)

3.6 CAN Filter

If we don't want to call an interrupt at any CAN package arriving at our MCU, it is possible to implement a filter: in this way only the wanted CAN IDs are able to trigger an interrupt.
In our application it is not something mandatory, but it is a nice addition as it can became quite handy for more crowded buses.
This filter is set up inside the CAN initialization function `MX_CAN_Init(void)`, here you can find some lines auto generated (don't touch them), and some space for the user code.
Inside the user code the mask configuration should look like this:

```
CAN_FilterTypeDef canfilterconfig;

canfilterconfig.FilterActivation = CAN_FILTER_ENABLE;// Enable can filtering
canfilterconfig.FilterBank = 0;                      // Anything between 0 to ...
    SlaveStartFilterBank
canfilterconfig.FilterFIFOAssignment = CAN_RX_FIFO0; // Filter bank to be initialized
canfilterconfig.FilterIdHigh = FILTERID <<5;         // Standard ID accepted (#1)
canfilterconfig.FilterIdLow = 0x0;                   // Extended part of ID (not used)
canfilterconfig.FilterMaskIdHigh = FILTERMASK <<5;   // Standard ID accepted (#2)
canfilterconfig.FilterMaskIdLow = 0x0;               // Extended part of ID (not used)
canfilterconfig.FilterMode = CAN_FILTERMODE_IDLIST;  // Specify filter mode
canfilterconfig.FilterScale = CAN_FILTERSCALE_32BIT; // Filter dimension
canfilterconfig.SlaveStartFilterBank = 0  ;          // Not important

HAL_CAN_ConfigFilter(&hcan, &canfilterconfig);       // Initialize the filter
```

`canfilterconfig` is structure containing the CAN filter configuration, used for the initializzation.
We are now going to explain the most intresting filter parameters:

- `FilterMode = CAN_FILTERMODE_IDLIST`: This is the CAN filter mode, we are using now the IDLIST mode. With that mode we can define two IDs that can trigger an interrupt.
- `FilterIdHigh = FILTERID <<5`: it is the standard id that can pass trough my filter, it is shifted by 5 because the can ID is 11 bit long, but the varavle we are writing on is 16 bit (`uint16_t`).

- `FilterIdLow = 0x0`: it is used if we deal with extended ID, so we are not using it.
- `FilterMaskIdHigh = FILTERMASK <<5`: in IDLIST mode it works as an additional standard ID that can trigger the interrupt, in MASK it is the mask correspondent to `FilterIdHigh`.
- `FilterMaskIdLow = 0x0`: same as above, but used for extended IDs

What we have used here is the simpler ID list mode, but it is nice to spend more words about the MASK mode:
In MASK mode, we can define an ID and a mask: the filter will compare bitwise the filter id and the received id, ignoring the position where we set `0` in the mask. In this way we could let pass multiple IDs by creating rules in the mask (for example, a mask `0b00000001` will let pass every odd numbers).
In other words, the incoming id will be compared with the filter only if the mask bit is 1, following the truth table in table 3.1. More info at [8]

Mask Bit n	Filter Bit n	Message Identifier bit	Accepted or Rejected bit n
0	X	X	Accepted
1	0	0	Accepted
1	0	1	Rejected
1	1	0	Rejected
1	1	1	Accepted

TABLE 3.1: Filter/mask truth table

3.7 CAN Header

The CAN header describes how the packets are made, and it is set in the `main()` function, after the CAN pheriperal initialization (`MX_CAN_Init()`) and start (`HAL_CAN_Start(&hcan)`). The header structure has been already definied, and filled with:

```
TxHeader.DLC = 8;                          // Can data lenght (8 byte)
TxHeader.IDE = CAN_ID_STD;                 // Id lenght       (standard 11 bit)
TxHeader.RTR = CAN_RTR_DATA;               // Type of frame   (Data frame)
TxHeader.StdId = NODEID;                   // Can id          (0x123 or 0x124)
TxHeader.ExtId = 0;                        // Extended id     (not used)
TxHeader.TransmitGlobalTime = DISABLE;     //                 (disabled)
```

As you can see from the code, we are using a standard ID, and the data length is 8 byte.

3.8 CAN packet Transmission

We want to transmit CAN packets whenever the blue button is pressed, so we are going to manage the transmission inside the user button callback function.

```
// User button callback
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
```

Here the the function to transmit CAN packets is called:

```
HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox)
```

- `&hcan`: It is the CAN peripheral that we will be using, it is auto generated and set by the code generator.
- `&TxHeader`: The header information, that we have already set.
- `TxData`: Data payload, it can be whatever 8 byte message, in our case an `uint8_t` vector with lenght 8.
- `&TxMailbox` It is the malibox used, where transmission message is stored (not that important).

The data stored in the vector `TxData` is transmitted over the CAN line, and the header of the packet is defined in `TxHeader`.
In our code, everytime that the blue button is pressed, a vector of 4 `uint8_t` numbers is printed on screen and transmitted over the CAN bus.

3.9 CAN packet Reception

Whenever a non masked ID is transmitting some data over the CAN line, an interrupt is triggered, and we can modify the behavior with its callback function.

```
// CAN Rx callback
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
```

Here we can get the received packet thanks to it's relative function:

```
HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RxData);
```

- `hcan`: It is the CAN peripheral that we will be using, it is auto generated and set by the code generator.
- `CAN_RX_FIFO0`: FiFo number of the received message to be read.
- `&RxHeader`: Header of the received packet, it is important to check the id of the transmitter and the data lenght.
- `RxData`: Received data payload, in our example it will be an `uint8_t` vector with lenght 8.

To elaborate or print data accordingly to the sender ID, an `if` statement is used.
For example: we know that the sender with ID `0x111` is the ADC, and it's payload are 4 `uint16_t` voltages with $1e^{-3}$ as scale factor: when this type of message is received the data is converted in 4 voltages, then it is printed, and undervoltages are checked.

3.10 Other functions

To debug and demonstrate the functionality of our tests, some print function has been implemented: those are an handy way to print data over the serial

```
//normal print
void debugPrint(UART_HandleTypeDef *huart, char _out[])

//print with newline
void debugPrintln(UART_HandleTypeDef *huart, char _out[])

//print a vector of 8 uint8_t numbers
void PrintlnEightBit(UART_HandleTypeDef *huart,uint8_t TxData[])
```

An undervoltage error handler has been implemented for test purpose, it prints some warnings and starts to blink a led:

```
void UnderVoltageError()
```

3.11 Run the code

Before running the code, make sure we have selected the right board: in our example we have used 2 different boards with different ID and CAN MASK.
The following line should be commented if we are programming BOARD 2, or not commented if we are programming BOARD 1:

```
37 #define BOARD1 1  // comment out this line when compiling for board #2
```

Then just press the green run button as shown in fig. 3.19, or it is also possible to run in debug mode (with the green bug button).

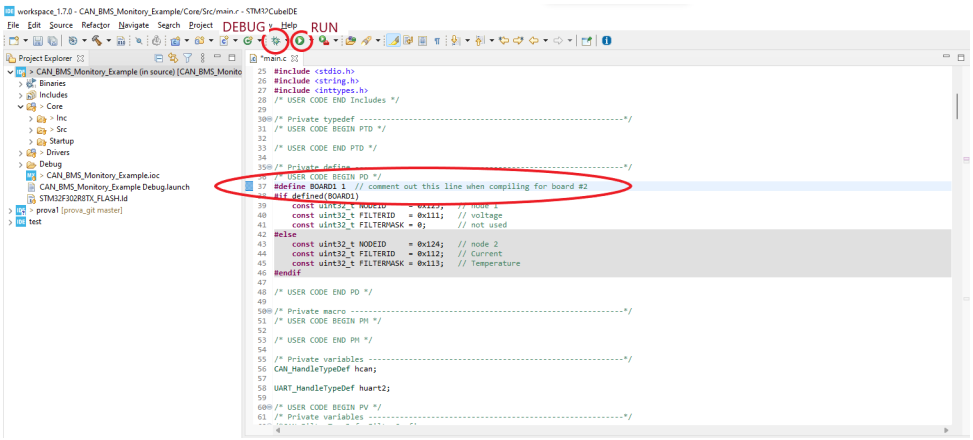


FIGURE 3.19: *Run the code on hardware*

Chapter 4

Implementation and test

In this last chapter of the project report are shown the various test that we did to validate our system. The following activities can be also used for other type of test in real or simulated environments.

Cable twisting

Despite we will not use this technique during our test, due to the short length of the cables, it is good to explain the reasons why the CAN-H and CAN-L wires are twisted. This to have a complete view of the CAN line installation in a real environments, like the automotive one.

FIGURE 4.1: *CAN twisted wires*

A twisted pair of wires reduces electromagnetic radiation from the pair and crosstalk between near cables and increase the rejection of external electromagnetic interference. The twisting ensures that both signal and return wires are subjected to the same interfering field and therefore cancel out its effect.

4.1 Reading and transmitting packets between nodes

The first test done is a simple experiment useful to understand if our CAN line works without any issues. The setup is composed by two ST demoboard that will act as two nodes of the line.
In the fig. 4.2 is possible to notice the transceivers and the entire CAN line (created with jumpers) disposed in a breadbord. This ensure fast and flexible connections, very useful in case of testing environments.

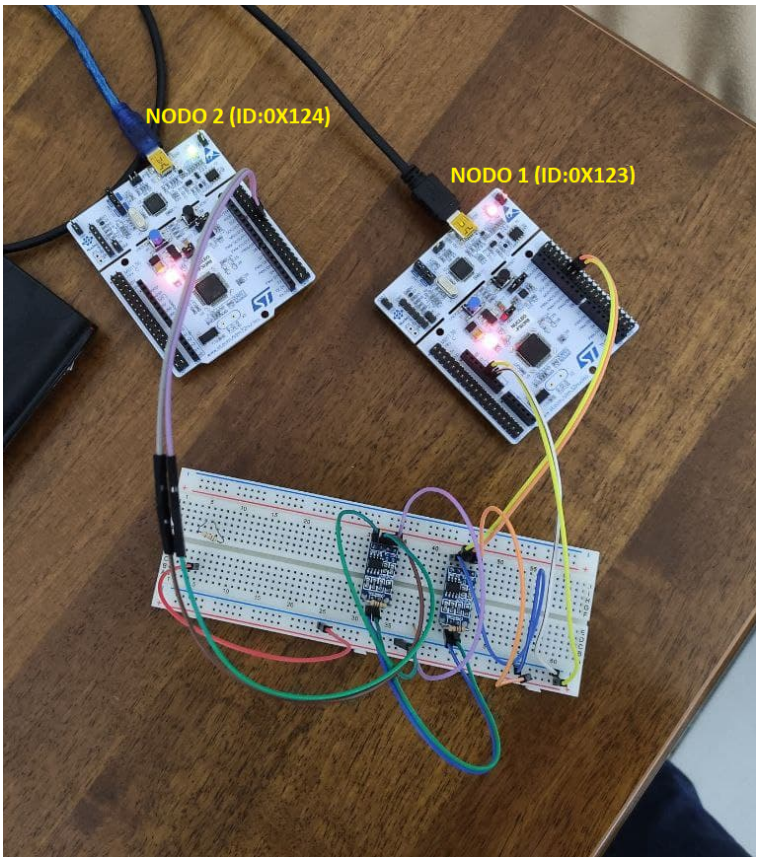


FIGURE 4.2: *Nodes test setup*

By clicking the user button (the blue one) of each node, a CAN message is sent to the CAN line. Using Coolterm, that is a serial visualizer, it is possible to observe the ID and the Data frame of the messages sent or received by the nodes.

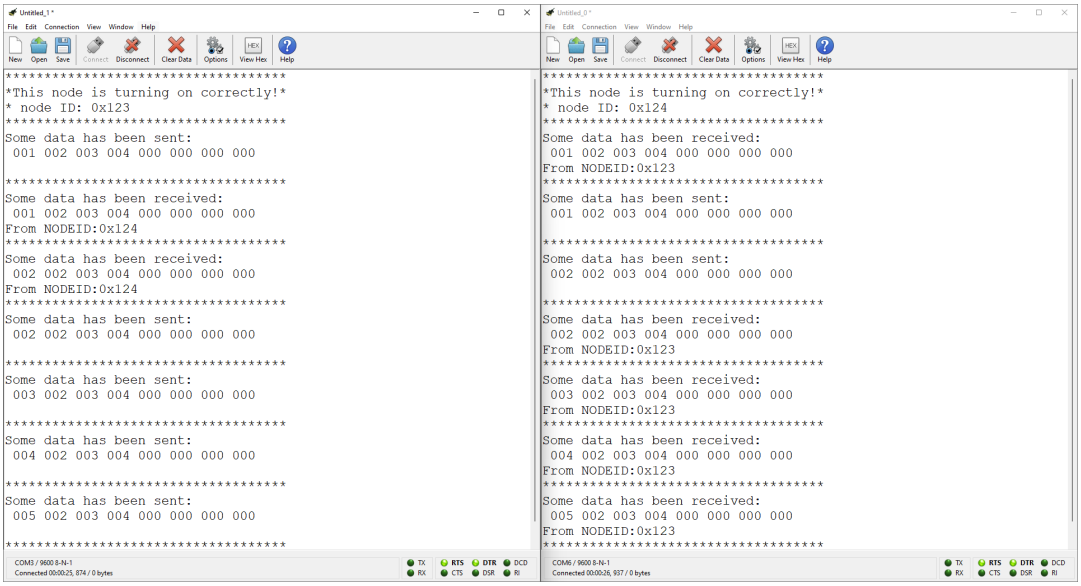


FIGURE 4.3: *Coolterm serial visualizer*

4.2 CAN sniffing using Busmaster

In the second test, the CAN sniffer is introduced in parallel to the Hardware setup.

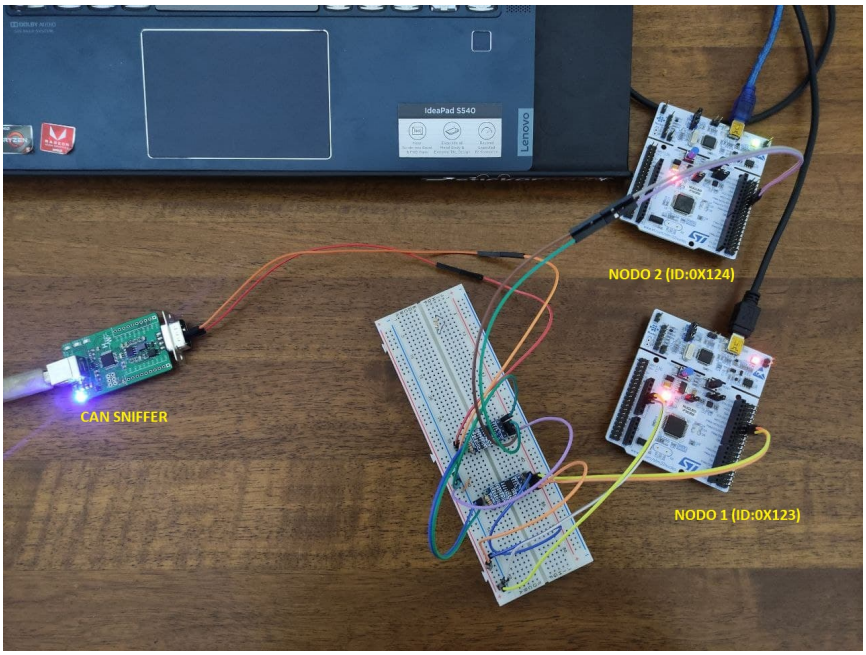


FIGURE 4.4: *Sniffing test setup*

This particular CAN sniffer device has a LED on the top that blink when a message is sensed on the line. Using the software Busmaster is possible to sniff the line, acquiring the different messages trasmitted by the nodes.

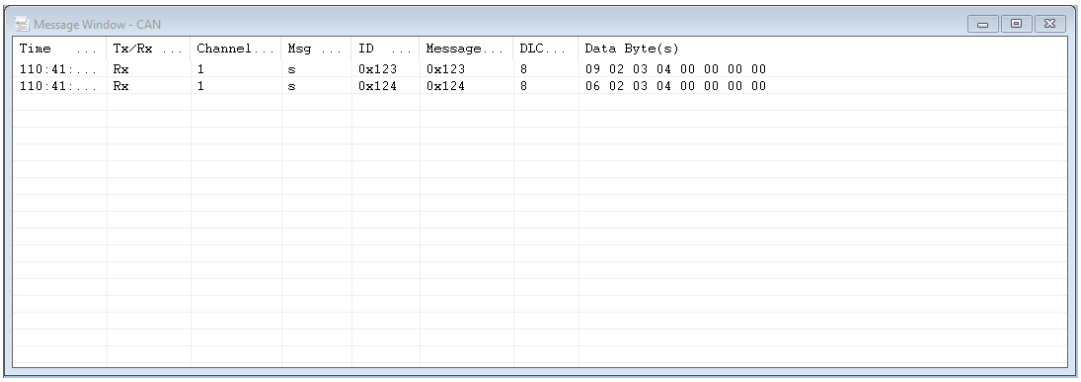


FIGURE 4.5: *Sniffing results on Busmasster*

4.3 Node simulator using Database

The last test performed present the same setup of the second one, with the two nodes and the CAN sniffer. In this case the HWtronics device is implemented as a node simulator able to send data stored in a database created in Busmaster.

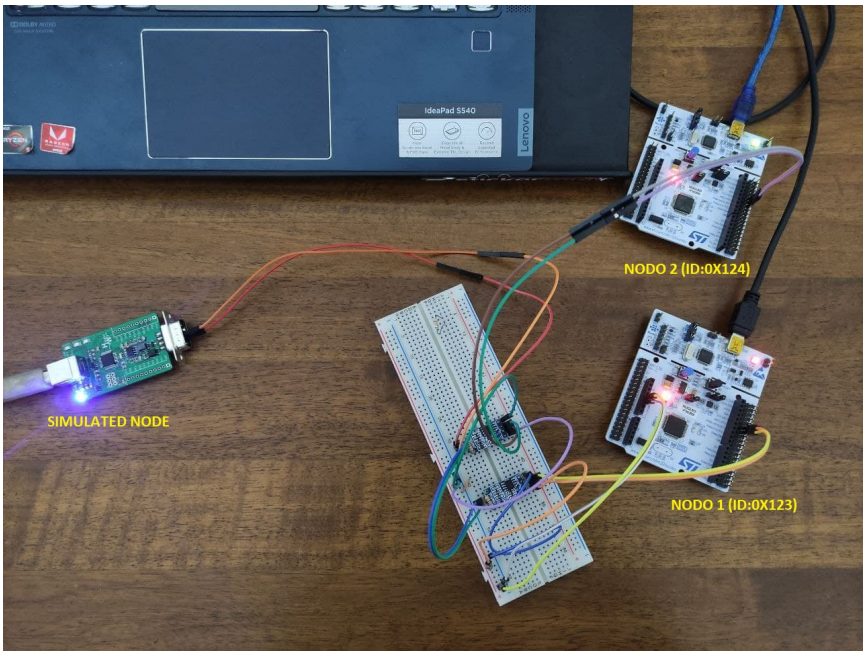


FIGURE 4.6: *Node simulation test setup*

In this configuration the simulated node can be "designed" to be whatever the user want, allowing different type of test, both on the CAN line and on the nodes software algorithms.
In the following subsections some example of what we did using the database presented in the previous chapter.

4.3.1 Voltage,current and temperature receiving

In fig. 4.7 the Busmaster window where is possible to see the Tx messages send from the Transmit window. Three different packets are send with different ID (and so with different priorities) and each contains different information.

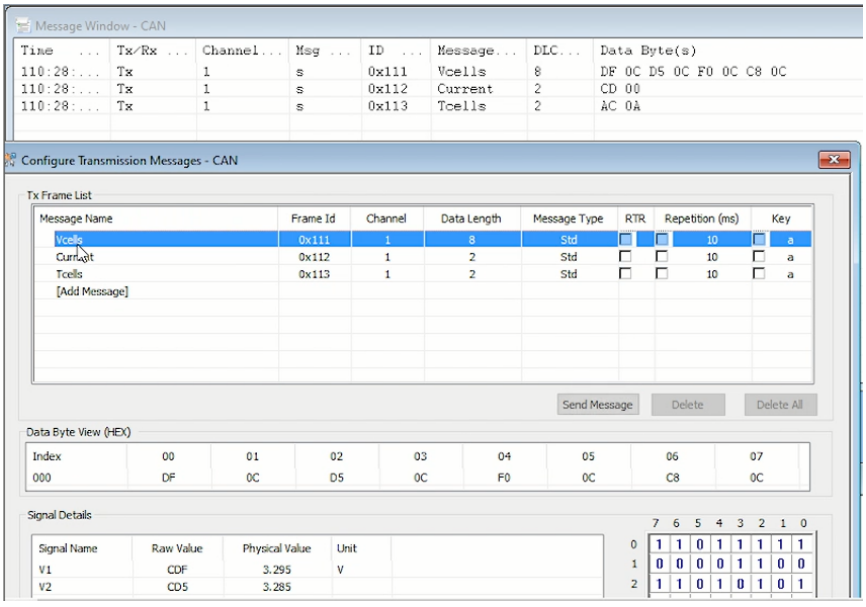


FIGURE 4.7: *Busmaster window with Rx messages*

The two nodes inside the line implement a mask that allow to take in consideration only the messages with a certain ID, while the others are discarded.
In fig. 4.8 two Coolterm windows are open and it is possible to notice that the packet containing the voltage is only seen by node 1, while node 2 only sees current and temperature packets (thanks to the filter).

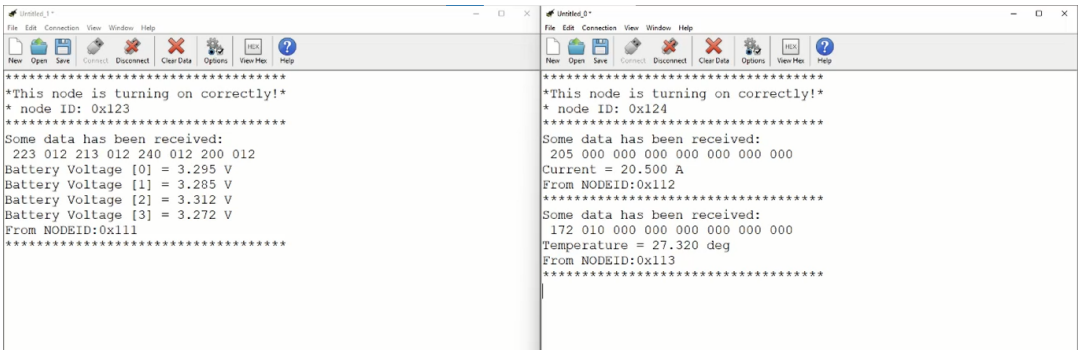


FIGURE 4.8: *Coolterm window with Rx messages*

4.3.2 Voltage in unsafe operating range

As previously explained, one of the measured quantity in the BMS is the voltage of the cell in series. This quantity must stay always in a safe range, with a precise max and min limits.
To simulate a possible unsafe simulation, the algorithm of the node 1 present an if statement that work in the following way:

- If the voltage of one of the four cells is lower than 2 (the value is just to test the logic) we are in an unsafe operation range.
  The node 1 send a new packet containing the message "error" informing about the unsafe state of one of the cells.

In fig. 4.9 the message window of Busmaster display the Rx message, containing "error", sent by the node 1 to inform about the unsafe state of the voltage quantity.

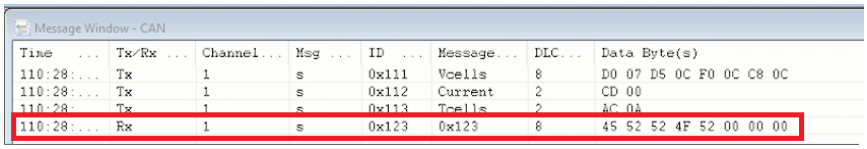


FIGURE 4.9: *Error message received by Busmaster*

In the same situation, the coolterm visualizer show the cell with the unsafe voltage and report a warning message notifying that an error has occurred and the execution has stopped.

```
**********
Some data has been received:
 208 007 213 012 240 012 200 012
Battery Voltage [0] = 2.000 V

**************************************
* An error has occurred!: UNDERVOLTAGE *
* execution has stopped, shooting down *
**************************************
```

FIGURE 4.10: *Unsafe cell voltage displayed in Coolterm*

4.4 Video demonstration

A video demonstration has been made to shows how everything is working. It is available on youtube on the following link[9]

4.5 Github repository

The final goal of this project was to learn and enjoy studying this subject, but what we care the most is to share our knowledge with people facing our same problem. For this reason (and for coding convenience of course) we decided to make our project open source sharing it on Github[10].
In that repository it is possible to take a look at the entire code, read this report, and for any suggestion or bug found feel free to reach us privately or just open an issue, we will be glad to give any help.

Bibliography

[1] can wiki, *Can bit time calculation website*, Last accessed December 13, 2021. [Online]. Available: http://www.bittiming.can-wiki.info.

[2] HWtronics, *Hwtronics can sniffer*, Last accessed December 13, 2021. [Online]. Available: https://www.hwtronics.com/2020/04/16/hw-mods-a-prototype-board-for-the-stlink-v3-mods/.

[3] STMicroelectronics, *Nucleo-f302r8*, Last accessed December 13, 2021. [Online]. Available: https://os.mbed.com/platforms/ST-Nucleo-F302R8/.

[4] ——, *Stm32f302x6 stm32f302x8 datasheet*, Last accessed December 13, 2021. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f302r8.pdf.

[5] ETAS, *Busmaster homepage*, Last accessed December 13, 2021. [Online]. Available: https://www.etas.com/en/applications/applications_busmaster.php.

[6] STMicroelectronics, *Stm32cubeide main page*, Last accessed December 13, 2021. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html.

[7] ARM, *Mbed os main page*, Last accessed December 13, 2021. [Online]. Available: https://os.mbed.com/mbed-os/.

[8] ImProgrammer, *Canbus id filter and mask*, Last accessed December 13, 2021. [Online]. Available: https://www.cnblogs.com/shangdawei/p/4716860.html.

[9] La Rocca Ferretti, *Design of a can line for testing video demonstration*, Last accessed December 13, 2021. [Online]. Available: https://youtu.be/j-Kal4BvnG0.

[10] La Rocca, Ferretti, *Can_bms_monitory_example github repository*, Last accessed December 13, 2021. [Online]. Available: https://github.com/mrjacopong/CAN_BMS_Monitory_Example.