**THE UNIVERSITY OF DANANG**
**UNIVERSITY OF SCIENCE AND TECHNOLOGY**
**Faculty of Advanced Science and Technology**

**LABORATORY REPORT**

# DEVICE NETWORKS

**Instructor** : Nguyen Huynh Nhat Thuong

**Class** : 21ECE

**Group members:** Luong Nhu Quynh

Vo Van Buu

Nguyen Thi Tam

Tran Hoang Minh

Da Nang, 6th May 2025

Table of Contents

## 1. Requirements

This report details a laboratory exercise conducted as part of the Device Network course, focusing on the practical implementation of the Controller Area Network (CAN) protocol. The primary objective was to establish and demonstrate a reliable two-way communication link between two distinct nodes. This was accomplished using two STM32F103C8T6 boards as the core processing units for each node. Essential to bridging the microcontroller logic with the physical network layer, dedicated CAN transceiver modules were employed on each node to handle the differential signaling required by the CAN bus standard. Furthermore, USB-to-TTL serial converters were connected to each node’s UART interface to provide real-time visibility into the network traffic, allowing transmitted and received messages to be easily monitored on a computer terminal. This report walks through the system's schematic design, the physical wiring connections, the configuration process for the STM32 microcontrollers, and the firmware code developed for both nodes. It concludes with a demonstration of the working bidirectional CAN communication.

## 2. Hardware

### 2.1 Block diagram

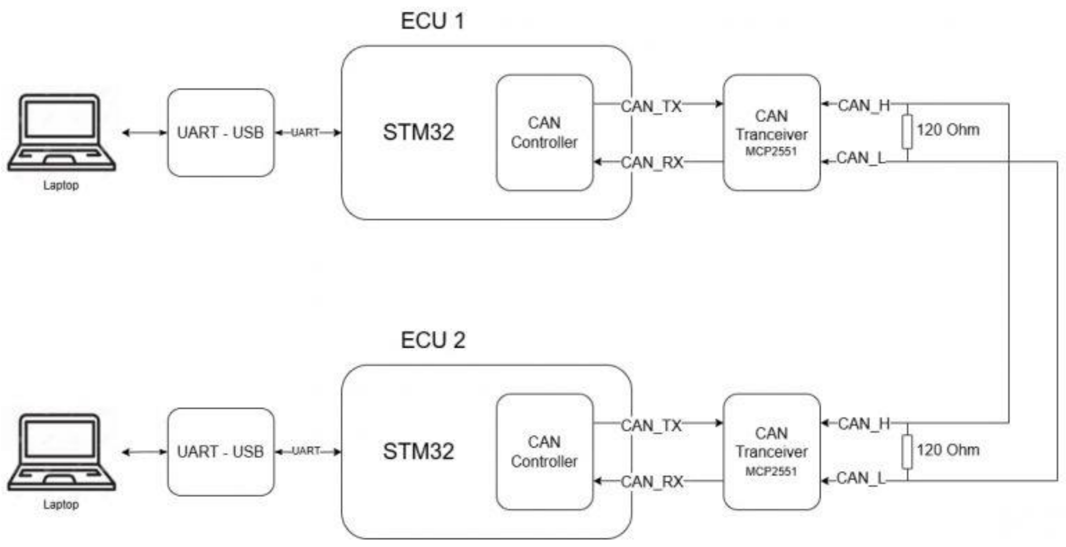


*Figure 1a. Block diagram*

### 2.2 Wiring diagram

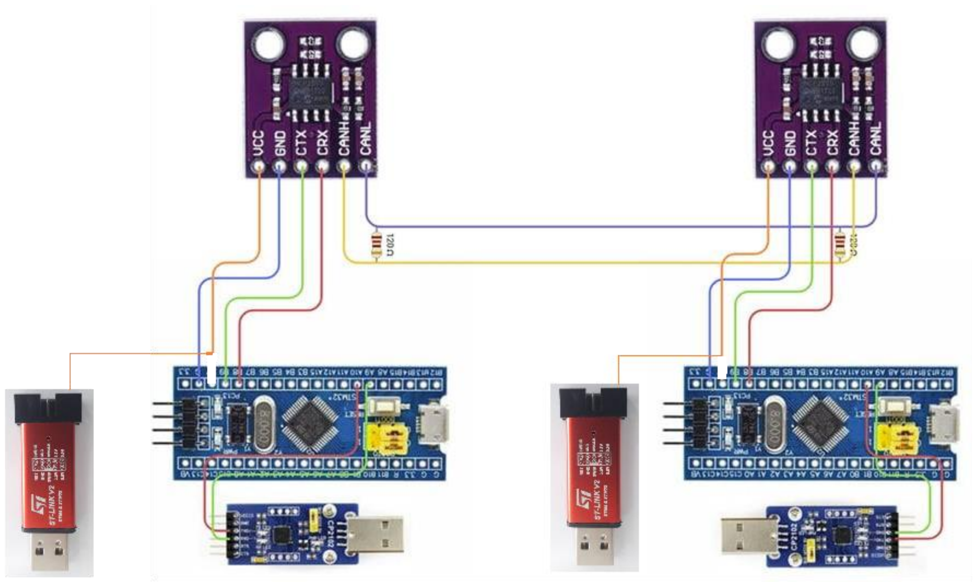
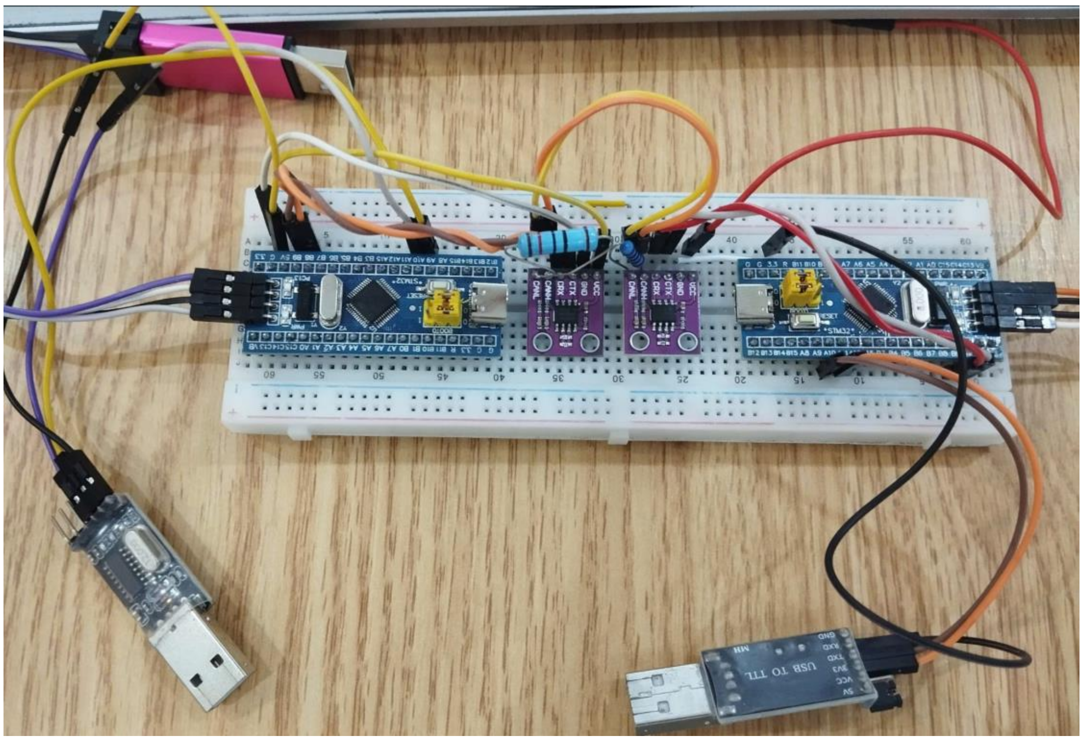


*Figure 1b. Wiring diagram*

### 2.3 Real circuit photo



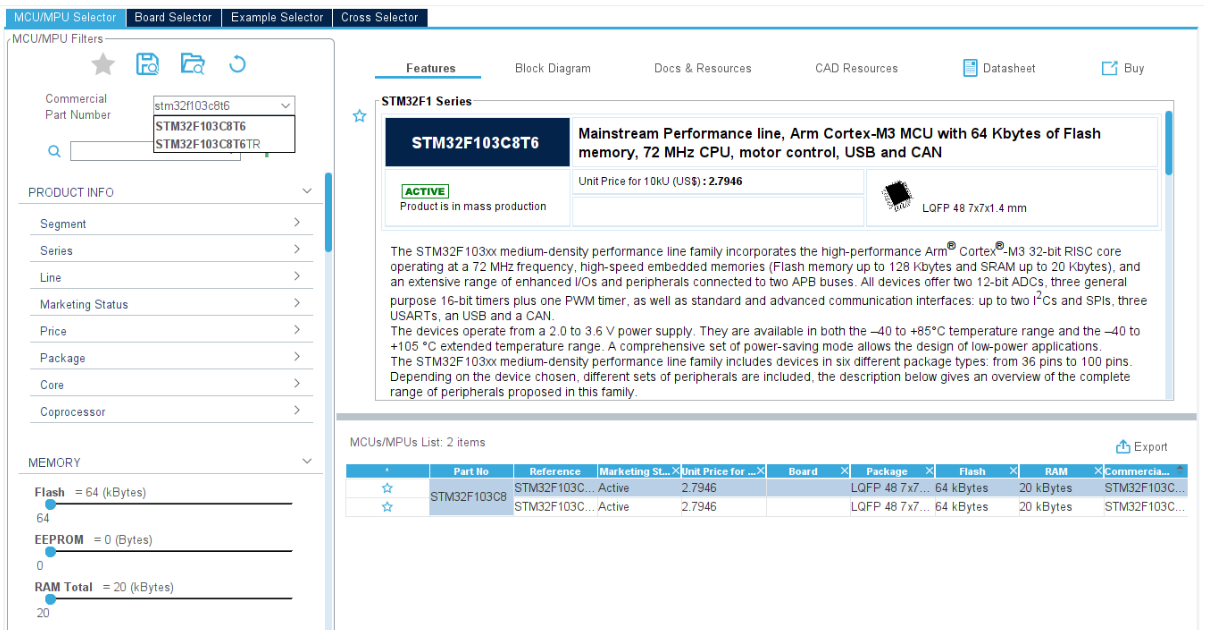
## 3. Software

### 3.1. Microcontroller configuration

The example is based on the STM32F103C8T6 board. The first thing to think about is the pins to be used by the CAN interface. So, you need to find which pins are free on the board.

Although an example is using Normal mode, you need to select CAN_Tx and CAN_RX with an externally unconnected I/O. For that, we need to refer to the board's schematic. On the STM32F103C8T6 MCU, there is only one CAN instance, which is CAN1. We need to start a project based on the board in STM32CubeMX:

1. Create a new project starting from the "ST board"
2. Select the **[Access to MCU selector]**
3. Find the board "STM32F103C8T6"
4. In the bottom right of the window, double-click on the board image.



Commercial Part Number: stm32f103c8t6

STM32F103C8T6
STM32F103C8T6TR

Flash = 64 (kBytes)
64
EEPROM = 0 (Bytes)
0
RAM Total = 20 (kBytes)
20

STM32F1 Series

STM32F103C8T6

Mainstream Performance line, Arm Cortex-M3 MCU with 64 Kbytes of Flash memory, 72 MHz CPU, motor control, USB and CAN

ACTIVE
Product is in mass production

Unit Price for 10kU (US$) : 2.7946

LQFP 48 7x7x1.4 mm

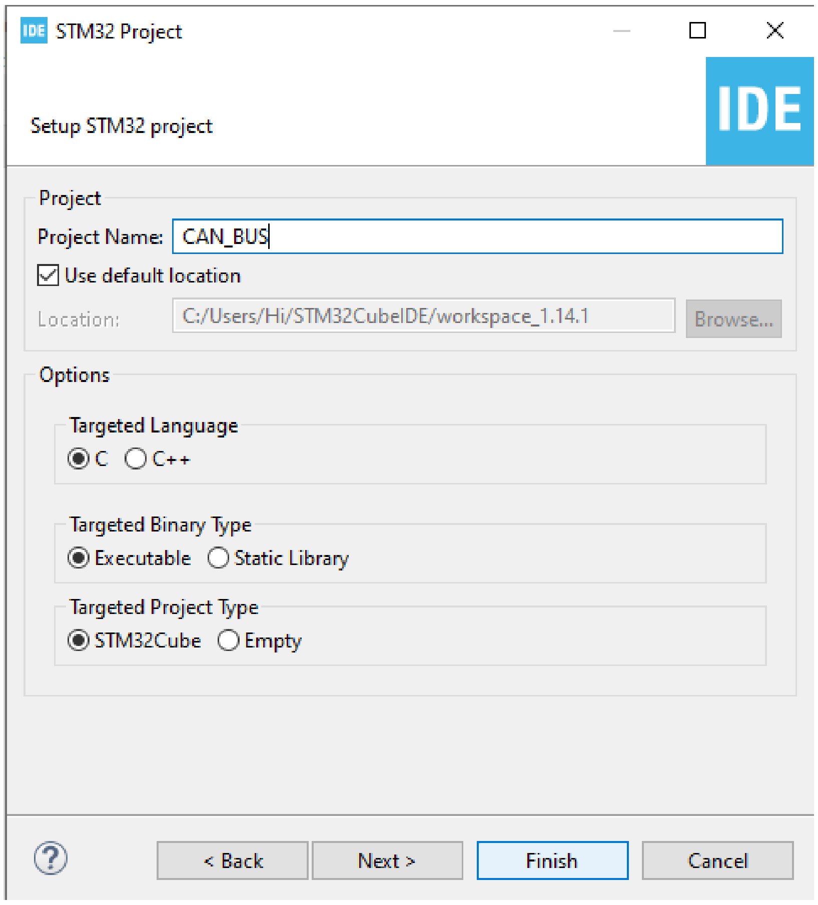
The STM32F103xx medium-density performance line family incorporates the high-performance Arm® Cortex®-M3 32-bit RISC core operating at a 72 MHz frequency, high-speed embedded memories (Flash memory up to 128 Kbytes and SRAM up to 20 Kbytes), and an extensive range of enhanced I/Os and peripherals connected to two APB buses. All devices offer two 12-bit ADCs, three general purpose 16-bit timers plus one PWM timer, as well as standard and advanced communication interfaces: up to two I$^2$Cs and SPIs, three USARTs, an USB and a CAN.
The devices operate from a 2.0 to 3.6 V power supply. They are available in both the –40 to +85°C temperature range and the –40 to +105 °C extended temperature range. A comprehensive set of power-saving mode allows the design of low-power applications.
The STM32F103xx medium-density performance line family includes devices in six different package types: from 36 pins to 100 pins. Depending on the device chosen, different sets of peripherals are included, the description below gives an overview of the complete range of peripherals proposed in this family.

MCUs/MPUs List: 2 items

| * | Part No | Reference | Marketing St... | Unit Price for ... | Board | Package | Flash | RAM | Commercia... |
|---|---|---|---|---|---|---|---|---|---|
| ☆ | STM32F103C8 | STM32F103C... | Active | 2.7946 | | LQFP 48 7x7... | 64 kBytes | 20 kBytes | STM32F103C... |
| ☆ | | STM32F103C... | Active | 2.7946 | | LQFP 48 7x7... | 64 kBytes | 20 kBytes | STM32F103C... |

*Figure 2. Select STM32F103C8T6*

STM32CubeMX asks you to initialize the peripherals with their default values. Click **[Yes]**:



STM32 Project

Setup STM32 project

Project

Project Name: CAN_BUS

[✓] Use default location

Location: C:/Users/Hi/STM32CubeIDE/workspace_1.14.1

Options

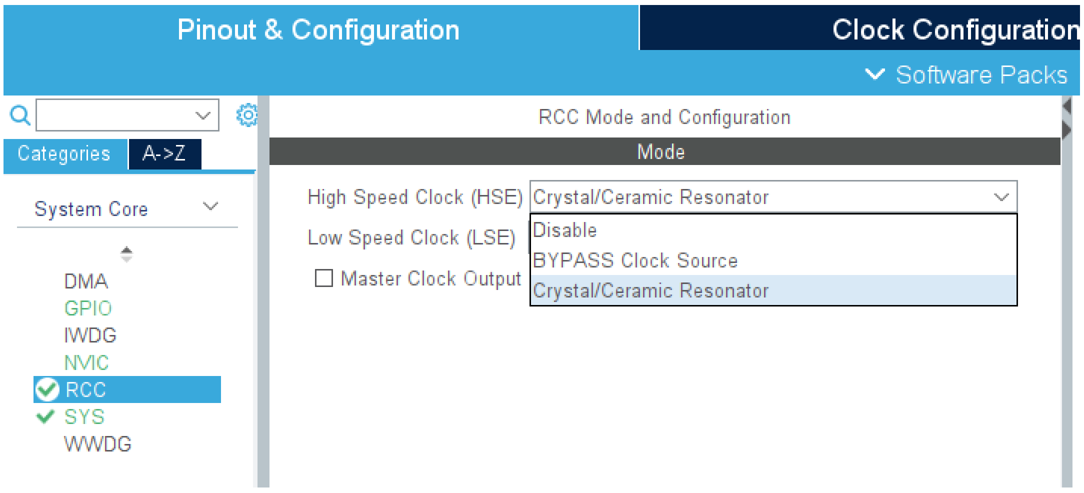
Targeted Language: [✓ C] C++

Targeted Binary Type: [✓ Executable] Static Library

Targeted Project Type: [✓ STM32Cube] Empty

*Figure 3. Click finish to create the Project*

At this stage, click RCC. Enable Crystal/Ceramic Resonator from the "High Speed Clock" menu.



RCC Mode and Configuration

Mode

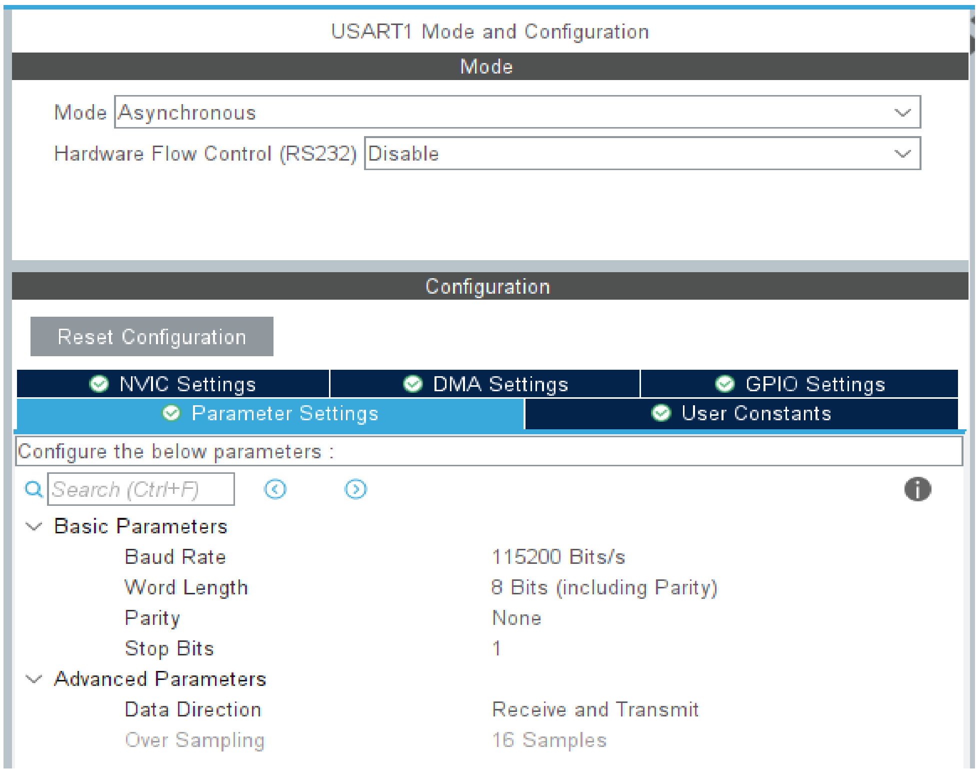
High Speed Clock (HSE): Crystal/Ceramic Resonator

Low Speed Clock (LSE)

[ ] Master Clock Output

*Figure 4. Enable Crystal/Ceramic Resonator*

The second thing to do is to enable USART1 from the "Connectivity" menu. Choose Asynchronous



USART1 Mode and Configuration

Mode

Mode: Asynchronous

Hardware Flow Control (RS232): Disable

Configuration

Reset Configuration

NVIC Settings | DMA Settings | GPIO Settings | Parameter Settings | User Constants

Configure the below parameters :

| | |
|---|---|
| **Basic Parameters** | |
| Baud Rate | 115200 Bits/s |
| Word Length | 8 Bits (including Parity) |
| Parity | None |
| Stop Bits | 1 |
| **Advanced Parameters** | |
| Data Direction | Receive and Transmit |
| Over Sampling | 16 Samples |

*Figure 5. USART activation*

Then enable CAN1 from the “Connectivity” menu:

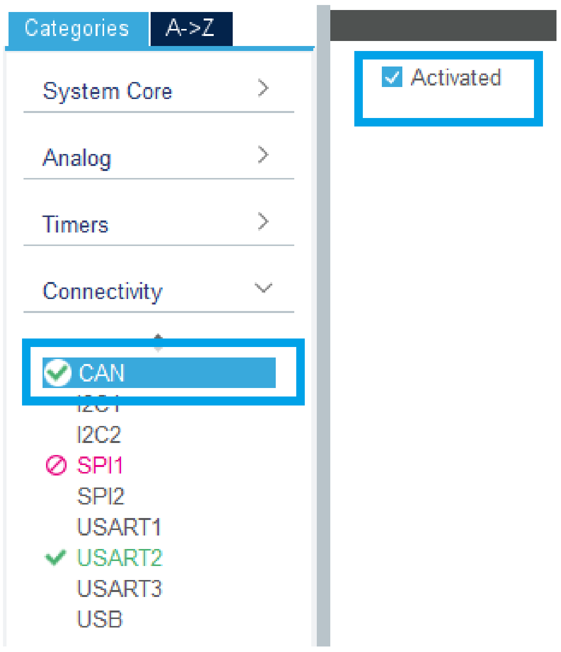


*Figure 6. CAN pin activation*

After enabling CAN1, PA11, and PA12 GPIOs are enabled by default for this product. Or we can choose PB8 and PB9. There were no conflicts between the already enabled GPIOs on the board and the CAN I/Os. This is confirmed by looking at the board’s schematic, PB8 and PB9 are free, and nothing is connected to them. So, these GPIOs can be used for the CAN application.

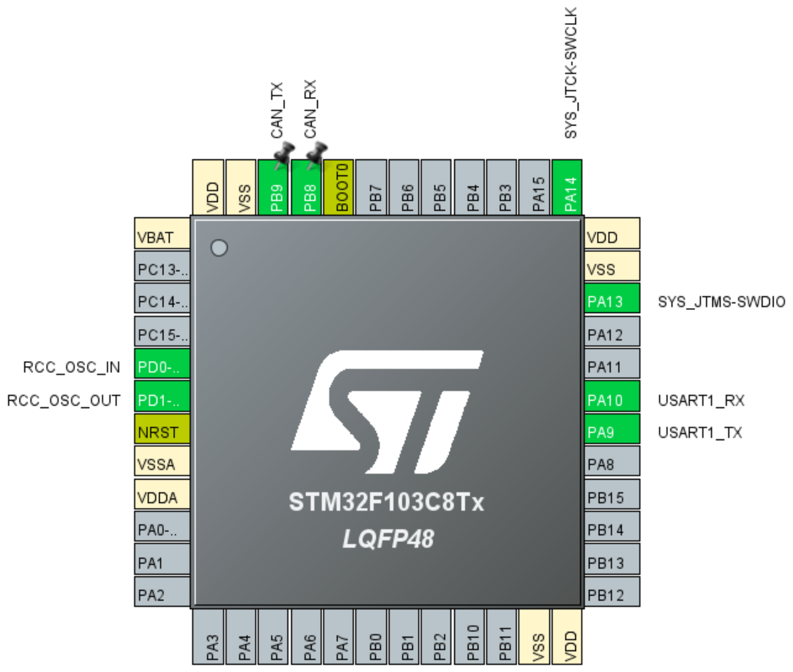


*Figure 7. CAN activated*

After the CAN interface activation, there are three things to configure: the clock configuration, the CAN timing configuration, and the GPIO configuration.

## CAN mode configuration

Once the initialization is complete, the software must request the hardware to enter Normal mode to be able to synchronize on the CAN bus and start reception and transmission.

**Figure 336. bxCAN operating modes**

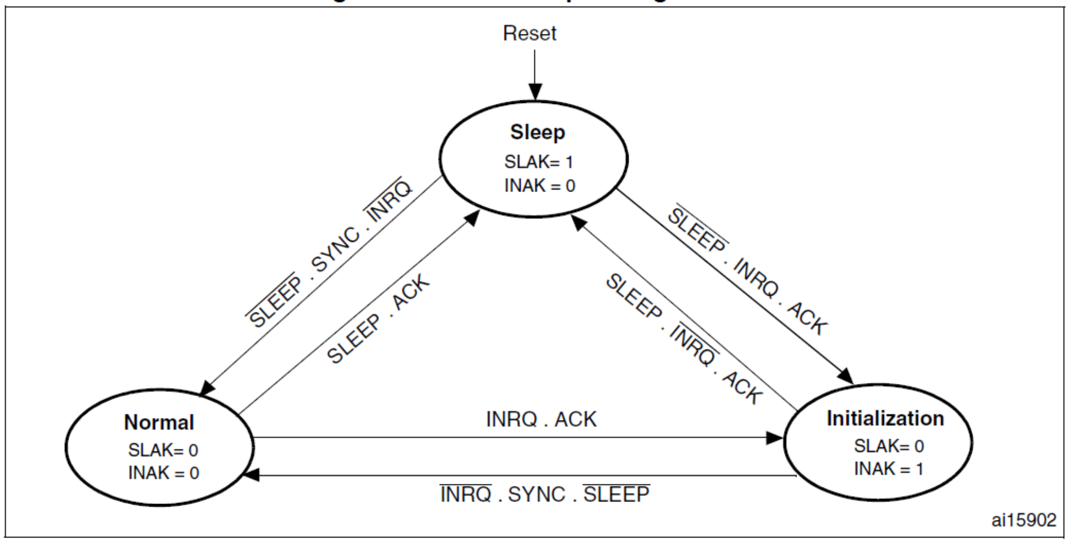
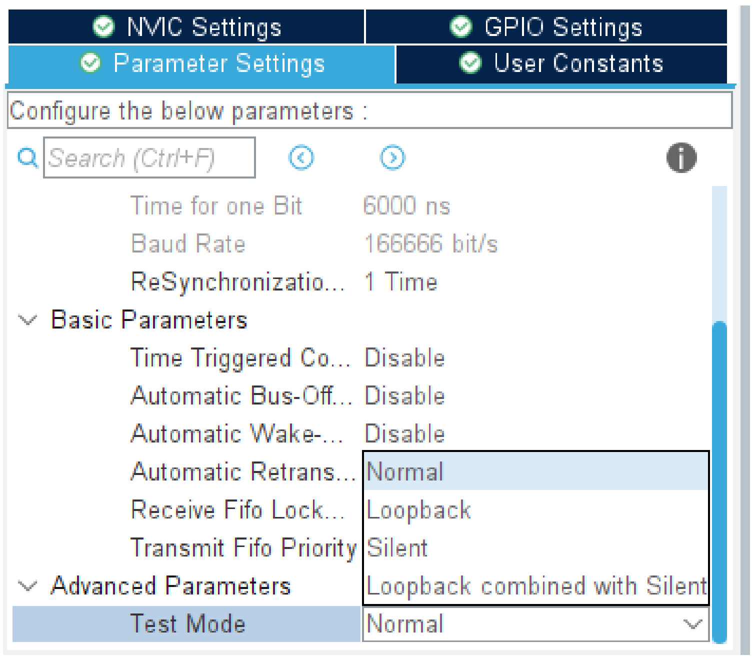


*Figure 8. CAN in Normal mode*

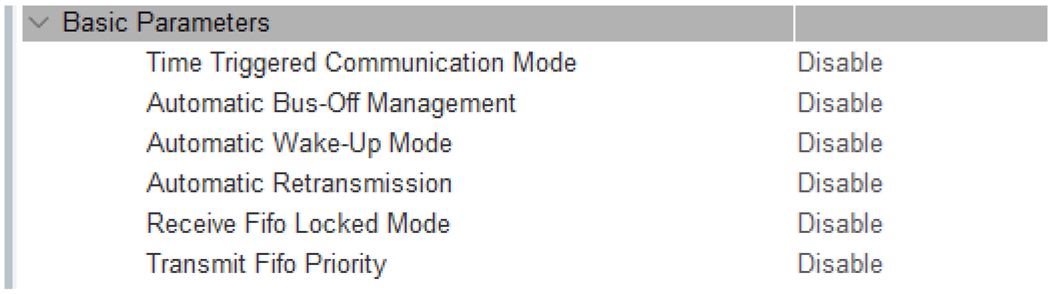
In STM32CubeMX, go to the "Parameter Settings" tab and set the Normal mode in the "Advanced Parameters" menu:



| | |
|---|---|
| Time for one Bit | 6000 ns |
| Baud Rate | 166666 bit/s |
| ReSynchronizatio... | 1 Time |
| Basic Parameters | |
| Time Triggered Co... | Disable |
| Automatic Bus-Off... | Disable |
| Automatic Wake-... | Disable |
| Automatic Retrans... | Normal |
| Receive Fifo Lock... | Loopback |
| Transmit Fifo Priority | Silent |
| Advanced Parameters | Loopback combined with Silent |
| Test Mode | Normal |

*Figure 9. Configuring CAN in Normal mode*

We keep the basic CAN parameters in their default settings. Not needed in our basic example:



| Basic Parameters | |
|---|---|
| Time Triggered Communication Mode | Disable |
| Automatic Bus-Off Management | Disable |
| Automatic Wake-Up Mode | Disable |
| Automatic Retransmission | Disable |
| Receive Fifo Locked Mode | Disable |
| Transmit Fifo Priority | Disable |

*Figure 10. Basic CAN parameter settings*

## Clock configuration

As the Normal mode is used, there is no need for a precise clock source because the bit sampler and the bit generator have the same clock source. So, HSI is used as the clock source in the example. In the case of STM32F103, the CAN peripheral is clocked from the APB1 domain as stated in the reference manual:

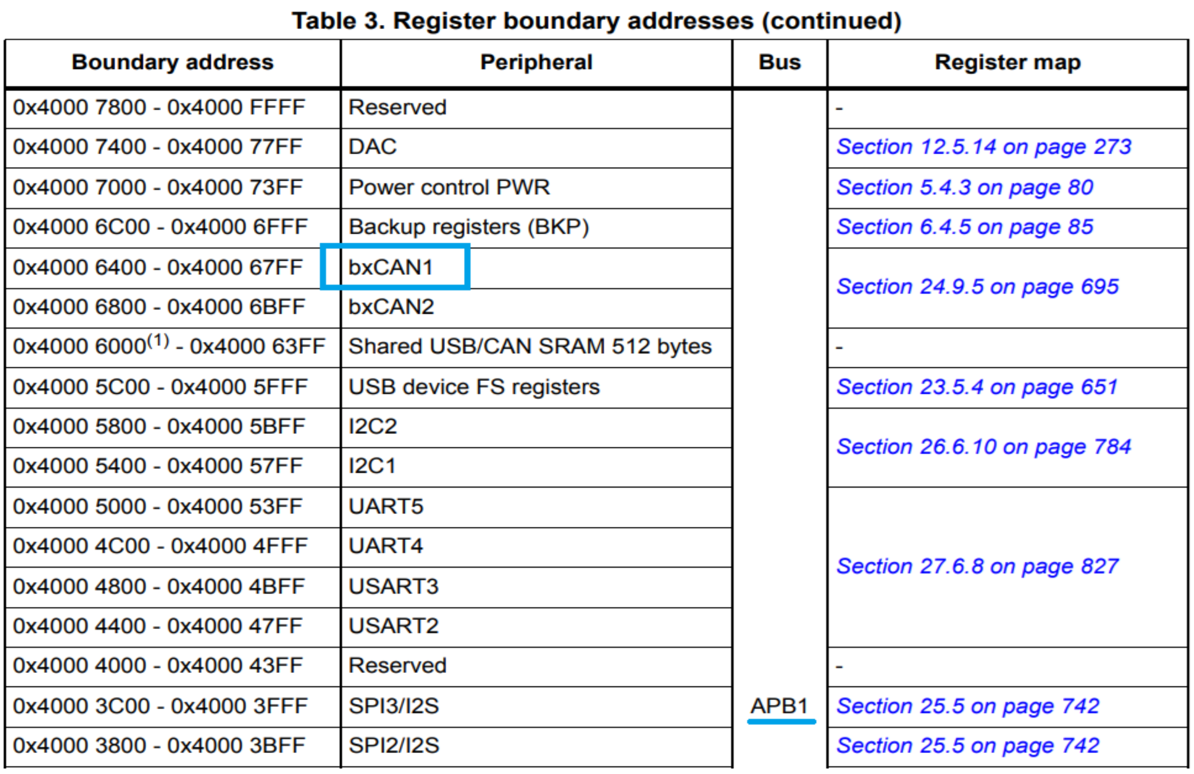


**Table 3. Register boundary addresses (continued)**

| Boundary address | Peripheral | Bus | Register map |
|---|---|---|---|
| 0x4000 7800 - 0x4000 FFFF | Reserved | APB1 | - |
| 0x4000 7400 - 0x4000 77FF | DAC | | *Section 12.5.14 on page 273* |
| 0x4000 7000 - 0x4000 73FF | Power control PWR | | *Section 5.4.3 on page 80* |
| 0x4000 6C00 - 0x4000 6FFF | Backup registers (BKP) | | *Section 6.4.5 on page 85* |
| 0x4000 6400 - 0x4000 67FF | bxCAN1 | | *Section 24.9.5 on page 695* |
| 0x4000 6800 - 0x4000 6BFF | bxCAN2 | | |
| 0x4000 6000(1) - 0x4000 63FF | Shared USB/CAN SRAM 512 bytes | | - |
| 0x4000 5C00 - 0x4000 5FFF | USB device FS registers | | *Section 23.5.4 on page 651* |
| 0x4000 5800 - 0x4000 5BFF | I2C2 | | *Section 26.6.10 on page 784* |
| 0x4000 5400 - 0x4000 57FF | I2C1 | | |
| 0x4000 5000 - 0x4000 53FF | UART5 | | *Section 27.6.8 on page 827* |
| 0x4000 4C00 - 0x4000 4FFF | UART4 | | |
| 0x4000 4800 - 0x4000 4BFF | USART3 | | |
| 0x4000 4400 - 0x4000 47FF | USART2 | | |
| 0x4000 4000 - 0x4000 43FF | Reserved | | - |
| 0x4000 3C00 - 0x4000 3FFF | SPI3/I2S | | *Section 25.5 on page 742* |
| 0x4000 3800 - 0x4000 3BFF | SPI2/I2S | | *Section 25.5 on page 742* |

*Figure 11. The CAN clock is on APB1 domain*

So, the CAN bit time is generated and sampled based on this clock source.

In STM32CubeMX, we select an HSI clock source over a PLL. The system clock source is 64 MHz, and the APB1 clock is 32 MHz (the maximum).

We also use the HSE clock source from the Crystal/Ceramic Resonator is maximum 72 MHz, and the APB1 clock is 36 MHz.

In this project, we chose the first option, in STM32CubeMX, in “Clock Configuration view”:

1. Select HSI clock source (8MHz/2)

2. Select PLLCLK to enable the PLL

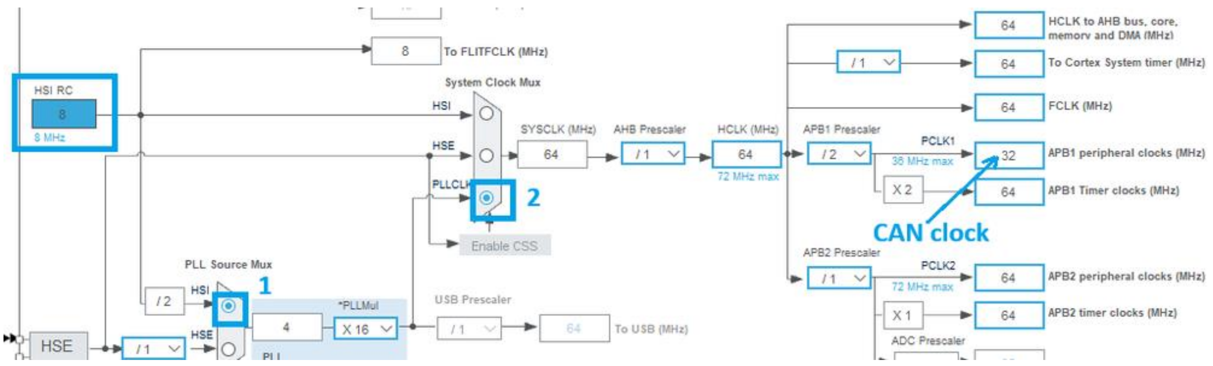


*Figure 12. System clock configuration*

## CAN timing configuration

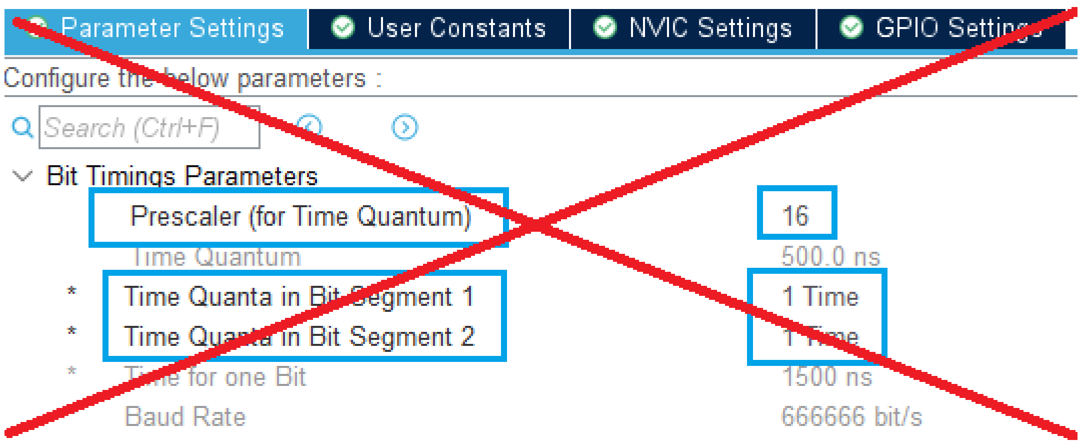
In this example, we set the CAN bitrate at 500 KB/s. Knowing that in the Normal mode, the bitrate configuration is not a must. The CAN bit time configuration applies to both Tx and Rx, and the transmitted messages are received at the same CAN bitrate.

Take precautions. Setting BS1 and BS2 to very low values, such as BS1=1 and BS2=1, leads to a communication issue. Better to decrease the CAN clock prescaler and increase BS1 and BS2 as much as possible, fitting the CAN bitrate.

In STM32CubeMX, in "Parameter Settings", the default timing config values set are:

- Prescaler = 16
- Time Quanta in Bit Segment 1 = 1
- Time Quanta in Bit Segment 2 = 1

These are not ideal settings.



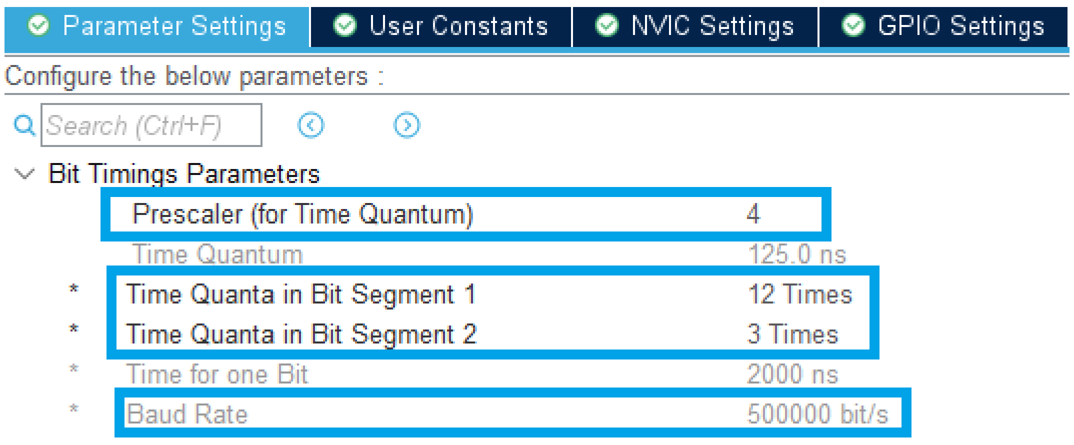
| | |
|---|---|
| Prescaler (for Time Quantum) | 16 |
| Time Quantum | 500.0 ns |
| Time Quanta in Bit Segment 1 | 1 Time |
| Time Quanta in Bit Segment 2 | 1 Time |
| Time for one Bit | 1500 ns |
| Baud Rate | 666666 bit/s |

*Figure 13. The default CAN timing configuration in STM32CubeMX*

The first thing to do is to increase BS1 and BS2 to their maximum values (BS1=16, BS2=8) in STM32CubeMX. Decrease the prescaler step by step until the CAN bitrate target value is found. If the operation does not allow it to converge to the wished exact bitrate, try to play with the prescaler.

As said, in our example, the CAN bitrate is set to 500 KB/s. The sample point needs to be selected between 75% and 87% of the bit time.

In STM32CubeMX, we choose these settings:


Parameter Settings | User Constants | NVIC Settings | GPIO Settings

Configure the below parameters :

Bit Timings Parameters

|  |  |  |
|---|---|---|
|  | Prescaler (for Time Quantum) | 4 |
|  | Time Quantum | 125.0 ns |
| * | Time Quanta in Bit Segment 1 | 12 Times |
| * | Time Quanta in Bit Segment 2 | 3 Times |
| * | Time for one Bit | 2000 ns |
| * | Baud Rate | 500000 bit/s |

*Figure 14. CAN timing configuration*

The bit time = SyncSeg + BS1 + BS2. In our case, the bit time = 1 + 12 + 3 = 16

Where SyncSeg is always = 1.

The sample point = (SyncSeg + BS1) / bit time = (1 + 12) /16 = 0,8125 à Sample point at: 0,8125%

In this configuration, the sample point is set at ~81% of the bit time.

You can use the attached Excel file calculator (CAN bitrate calculation.xlsx) to find the suitable timing configurations for the CAN bitrate, including the sample point.

## CAN GPIO configuration

Even if the CAN signals are internally connected by hardware, it is mandatory to configure the I/Os. This is already done when CAN is activated in STM32CubeMX as shown in Figures 5 and 6. To avoid issues, we need to activate the internal pull-up resistor manually on the CAN_Rx pin.

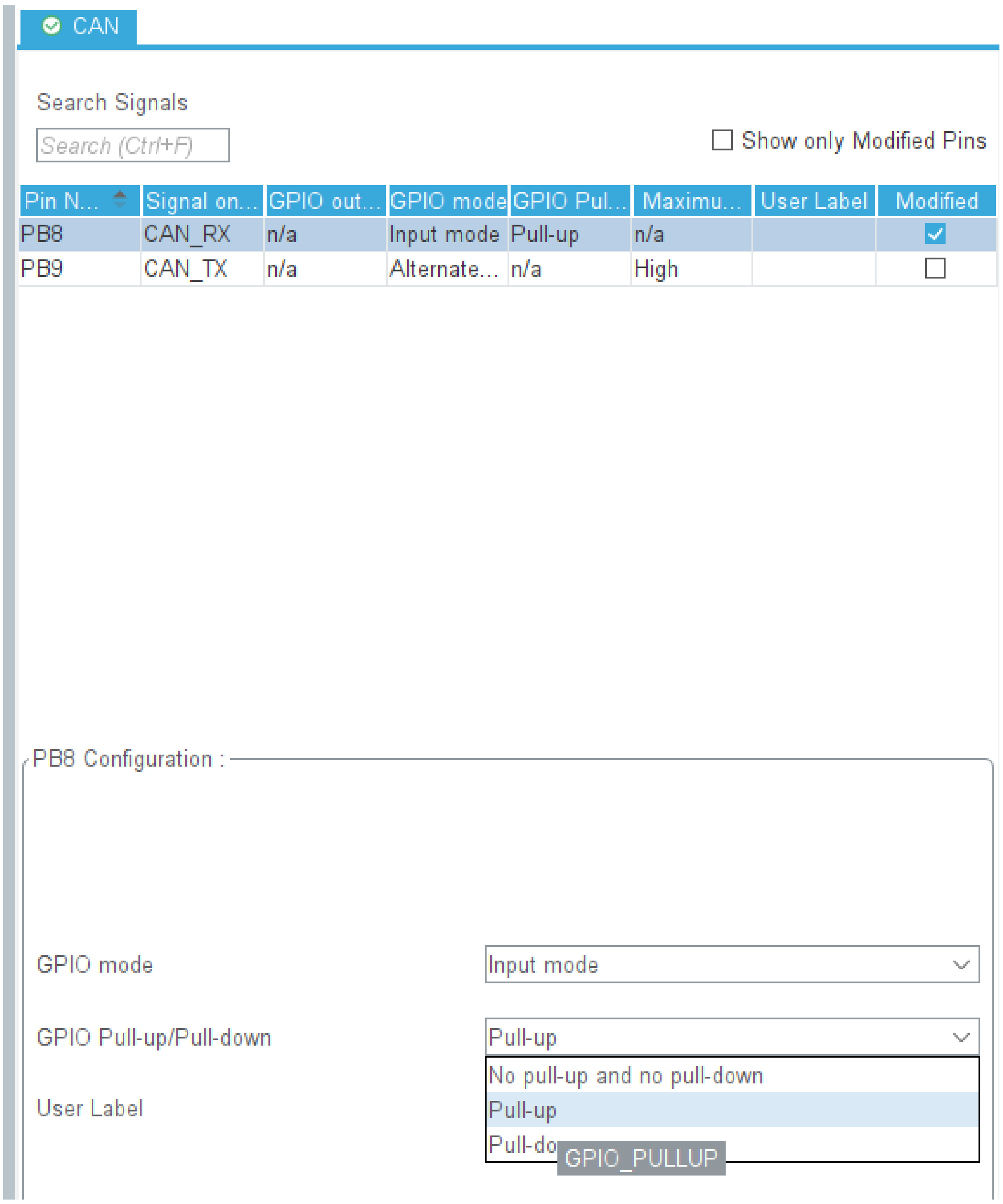
In that case, we need to go to the “GPIO settings” tab and set the pull-up on the CAN_Rx pin, which is PB8 in this case.

In STM32CubeMX, go to the “GPIO Settings” tab, and:

1. Click on the CAN_RX pin

2. Go to the PB8 configuration, then to “GPIO Pull-up / Pull-down” and select **[Pull-up]** to set the internal resistor.

***In CAN communication, the RX line often requires a pull-up resistor to maintain a stable logic level when the bus is in the idle state. Without a pull-up resistor, the pin may become noisy, leading to data reception errors.***



| Pin N... | Signal on... | GPIO out... | GPIO mode | GPIO Pul... | Maximu... | User Label | Modified |
|---|---|---|---|---|---|---|---|
| PB8 | CAN_RX | n/a | Input mode | Pull-up | n/a | | ✓ |
| PB9 | CAN_TX | n/a | Alternate... | n/a | High | | ☐ |

PB8 Configuration :

GPIO mode: Input mode

GPIO Pull-up/Pull-down: Pull-up

User Label

*Figure 15. Setting the pull-up on the CAN_Rx pin*

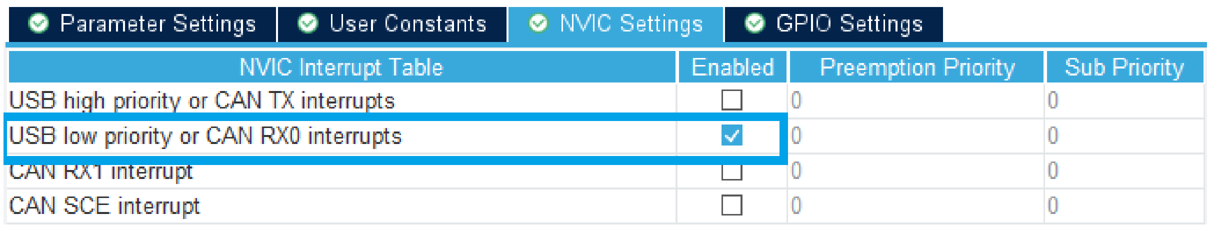
## CAN interrupt settings

In the example, we use the CAN message reception in the interrupt mode. There are three NVIC interrupts available for CAN: two for the FIFOs' reception (FIFO0 and FIFO1) and one for the CAN error management.

If your system needs to process received data immediately (e.g. real-time or high priority), it is useful to enable interrupts for the FIFO (such as CAN RX0 or CAN RX1 in your configuration table) to help the system respond quickly to new data. The interrupt will trigger a processing function when the FIFO receives data, helping you not to miss information. However, if the application allows polling, you may not need to enable interrupts, but only read the FIFO periodically. In this case, the interrupt priority and configuration (Preemption Priority, Sub Priority) can be left at default 0 as shown in the figure.

On the STM32F1 family, the CAN Rx FIF0 NVIC interrupt line is shared with a USB low-priority interrupt line. In our case, we activate the interrupt only on FIFO0 as we do not use FIFO1 or the CAN error.

In STM32CubeMX, go to the "NVIC Settings" tab, and check the **[USB low priority or CAN RX0 interrupts]** option:



| NVIC Interrupt Table | Enabled | Preemption Priority | Sub Priority |
|---|---|---|---|
| USB high priority or CAN TX interrupts | ☐ | 0 | 0 |
| USB low priority or CAN RX0 interrupts | ✓ | 0 | 0 |
| CAN RX1 interrupt | ☐ | 0 | 0 |
| CAN SCE interrupt | ☐ | 0 | 0 |

*Figure 16. Enable CAN RX0 NVIC interrupt channel*

We can keep the "Preemption Priority" and "Sub Priorities" set to their default values 0.

At this stage, all is set in the STM32CubeMX. It is time to generate the code. See section 3.

Note that it is not possible to use USB and CAN at the same time on the STM32F103. So, it is not possible to create a USB to CAN converter with it, while some other products and families do.

## 3.2 Flowchart

## 3.3. Code

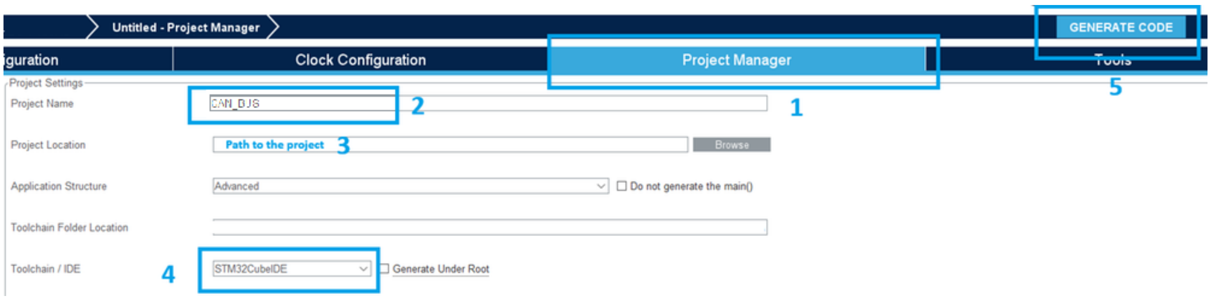
In STM32CubeMX, after setting all the parameters of the RCC and the CAN peripheral, we need to name the project. Select where the project will be saved and set the toolchain to use. In our case, it is STM32CubeIDE.

In STM32CubeMX,

1. Go to the Project Manager view

2. Name the project. Example: CAN BUS

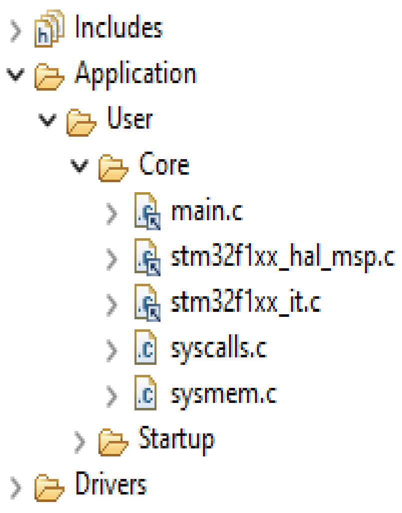
3. Browse to the location where the project will be saved
4. Select your preferred toolchain. In our case, it is STM32CubeIDE
5. Click on “Generate code.”



Project Settings
Project Name: CAN_DJG
Project Location: Path to the project | Browse
Application Structure: Advanced | ☐ Do not generate the main()
Toolchain Folder Location:
Toolchain / IDE: STM32CubeIDE | ☐ Generate Under Root

*Figure 17. Project code generation*

After generating the code and opening STM32CubeIDE, this is what you need to see in the “Project Explorer”:



```
> Includes
v Application
    v User
        v Core
            > main.c
            > stm32f1xx_hal_msp.c
            > stm32f1xx_it.c
            > syscalls.c
            > sysmem.c
        > Startup
> Drivers
```

*Figure 18. Project Explorer view after generating the code*

## Adding the CAN code to the application

We focus on the main.C file where we will add the CAN application.

The purpose of the application is very simple: Send 8 bytes of CAN data frame and receive them in Rx FIFO0 using an interrupt.

Variables to declare/add between */* USER CODE BEGIN PV */* and */* USER CODE END PV */*. Insert this code in between:

```
CAN_TxHeaderTypeDef TxHeader;

CAN_RxHeaderTypeDef RxHeader; ;

uint8_t TxData[8] = “TXsend_\n”;

uint8_t RxData[8];

uint32_t TxMailbox;
```

*We create a variable TxHeader and RxHeader of type CAN header struct that includes parts of the CAN message frame, and later, the Contents TxHeader will be declared and then added in the TX message. While RxHeader is used to store the header of the received message.

In MX_CAN_Init(), we add additional code: the CAN filter configuration, the CAN start preparation, and the CAN interrupt. For filters, we use a very simple configuration, i.e., passing all the IDs.

The code is added in between */* USER CODE BEGIN CAN_Init 2 */* and */* USER CODE END CAN_Init 2 */*. Insert this code in between:

```
/* The CAN filter configuration */

sFilterConfig.FilterBank = 0;

sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK; sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT; sFilterConfig.FilterIdHigh = 0x0000;

sFilterConfig.FilterIdLow = 0x0000;

sFilterConfig.FilterMaskIdHigh = 0x0000;

sFilterConfig.FilterMaskIdLow = 0x0000; sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;

sFilterConfig.FilterActivation = ENABLE; sFilterConfig.SlaveStartFilterBank = 14;

if (HAL_CAN_ConfigFilter(&hcan, &sFilterConfig) != HAL_OK)

{ Error_Handler(); }

if (HAL_CAN_Start(&hcan) != HAL_OK)

{  Error_Handler();}

if (HAL_CAN_ActivateNotification(&hcan, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK)
```

```
{ Error_Handler(); }
```

Do not forget to add the declaration of the filter structure in between */* USER CODE BEGIN CAN_Init 0 */* and */* USER CODE END CAN_Init 0 */*.

Insert this code in between: CAN_FilterTypeDef sFilterConfig;

Now, need to declare the CAN FIFO0 interrupt callback. You can add it in between

*/* USER CODE BEGIN 4 */* and */* USER CODE END 4 */*.

Insert this code in between:

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *CanHandle) { /* Get RX message
*/

if (HAL_CAN_GetRxMessage(CanHandle, CAN_RX_FIFO0, &RxHeader, RxData) != HAL_OK) {

/* Reception Error */ Error_Handler(); }

/* Transmit received data over UART */

if (HAL_UART_Transmit(&huart1, RxData, 8, 100) != HAL_OK)

{ /* UART Transmission Error */ Error_Handler(); }

}
```

Getting the message with *HAL_CAN_GetRxMessage()* fills the data buffer RxData with the received data and the RxHeader with the frame header that contains the information about the received frame like the ID, the data length etc. Then the message will be transmitted to PC through UART by *HAL_UART_Transmit()*

Finally, add the transmit operation in main(). The code to insert is between */* USER CODE BEGIN 2 */* and */* USER CODE END 2 */*:

The transmitted frame CAN ID is in a standard format with a value of 0x123 and acts as a data frame.

The data length is 8 bytes. The first frame that is transmitted.

```
TxHeader.StdId = 0x123;  // ID of CAN device

TxHeader.RTR = CAN_RTR_DATA; // This is a Data frame, not requesting or error message.
```

TxHeader.IDE = CAN_ID_STD;  // This is standard ID

TxHeader.DLC = 8; // The data length is 8 bytes.

TxHeader.TransmitGlobalTime = DISABLE;

Now, it is time to add the code responsible for the CAN transmission in the while loop. Note that we need to check first if there is one Tx mail box available to send the current CAN frame. Otherwise, HAL_CAN_AddTxMessage() returns an error if you do not wait until the previous frame is sent.

Some users put HAL_Delay(). It works but for efficiency reason, it is preferable to poll on the Tx free mail box.

The application transmits 8 bytes where the sixth byte (byte 6) is incremented.

Insert this code between *while (1) {* and */* USER CODE END WHILE */*

```
        TxData[6] ++;

        HAL_Delay(1000);

        while(HAL_CAN_GetTxMailboxesFreeLevel(&hcan) == 0);

        if (HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox) != HAL_OK) {
Error_Handler(); }
```

Now, compile the project by clicking on the hammer button:

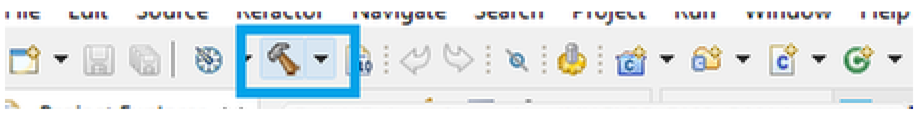


*Figure 19. Compile the project*

## Configure the second ECU

We can set up the same with the first ECU, or add some following code

Variables to declare/add between */* USER CODE BEGIN PV */* and */* USER CODE END PV */.*

Insert this code in between:

```
CAN_TxHeaderTypeDef TxHeader;
```

```
CAN_RxHeaderTypeDef RxHeader; ;

uint8_t TxData[8] = “RXsend_\n”;

uint8_t RxData[8];

uint32_t TxMailbox;
```

Now, need to declare the CAN FIFO0 interrupt callback. You can add it in between */* USER CODE BEGIN 4 */* and */* USER CODE END 4 */.*

Insert this code in between:

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *CanHandle)

{

 /* Get RX message */

 if (HAL_CAN_GetRxMessage(CanHandle, CAN_RX_FIFO0, &RxHeader, RxData) != HAL_OK)

 {

   /* Reception Error */

   Error_Handler();

 }

 /* Transmit received data over UART */

 if (HAL_UART_Transmit(&huart1, RxData, 8, 100) != HAL_OK)

 {

   /* UART Transmission Error */

   Error_Handler();

 }

 if (HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox) != HAL_OK)

 {

   /* Transmission request Error */
```

```
  Error_Handler();
}}
```

# 4. Results

## 4.1. Testing plan

In this section, we show how to test the CAN in Normal mode. For that we use the Saleae logic analyzer in CAN mode. We probe CAN_Tx (PB9).

Connect the GND of the logic analyzer to the GND of the board and connect the data 0 (D0) of the logic analyzer input to CAN_Tx (PB9).

In the Saleae logic analyzer software interface, enable the digital channel 0. Go to the analyzer menu “Add Analyzer” and search for “CAN” and select it:

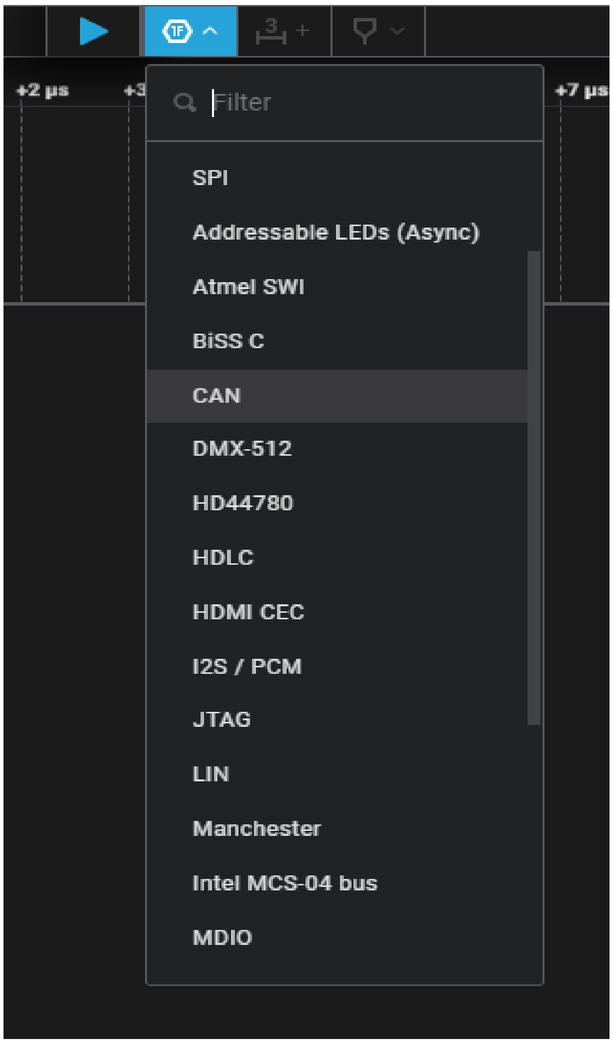
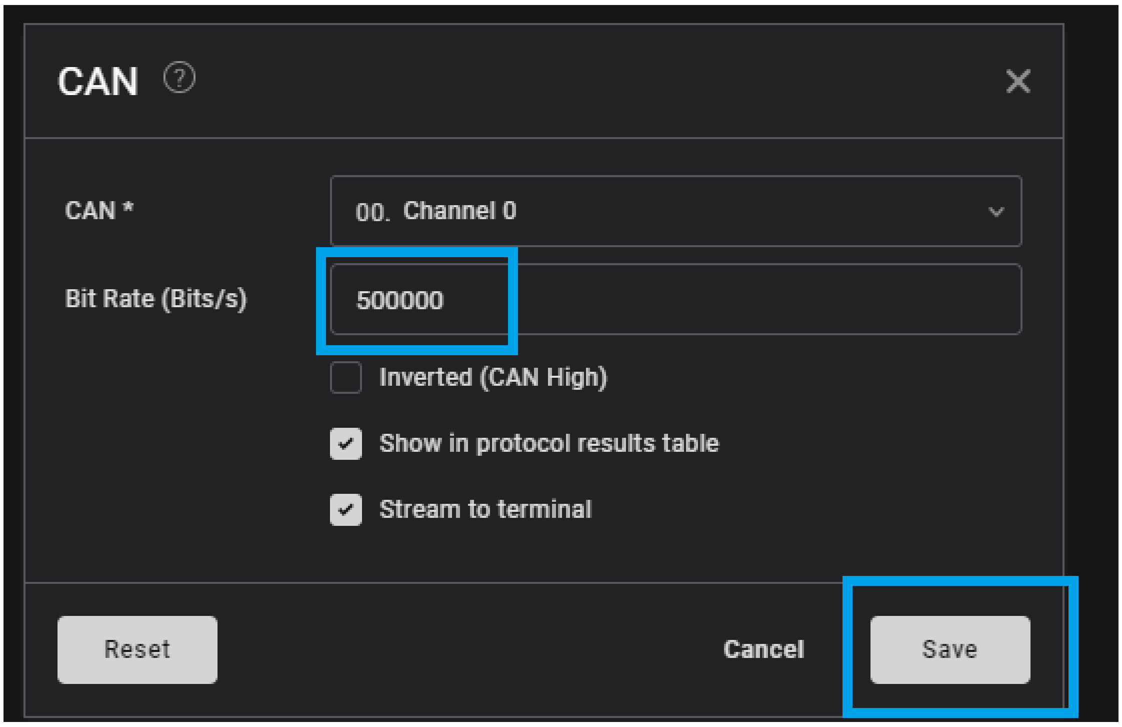


*Figure 20. Configure the Saleae logic analyzer digital channel 0 input in CAN mode*

A window is shown to configure the CAN bitrate. In our case 500 KB/s is used. Set the value to 500000 (in Bits/s) and click **[Save]**. Do the same with the UART in channel 1 and choose Async serial



CAN

CAN * 00. Channel 0

Bit Rate (Bits/s) 500000

- [ ] Inverted (CAN High)
- [x] Show in protocol results table
- [x] Stream to terminal

*Figure 21. CAN configuration with Saleae logic analyzer*

Now, connect the board to your PC/laptop with a USB data cable over STLINK USB connector and in STM32CubeIDE, start a debug session:

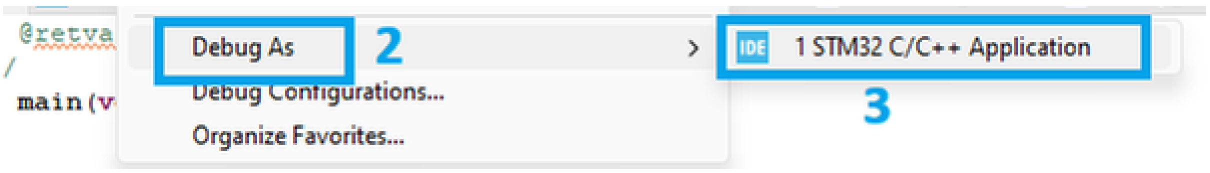
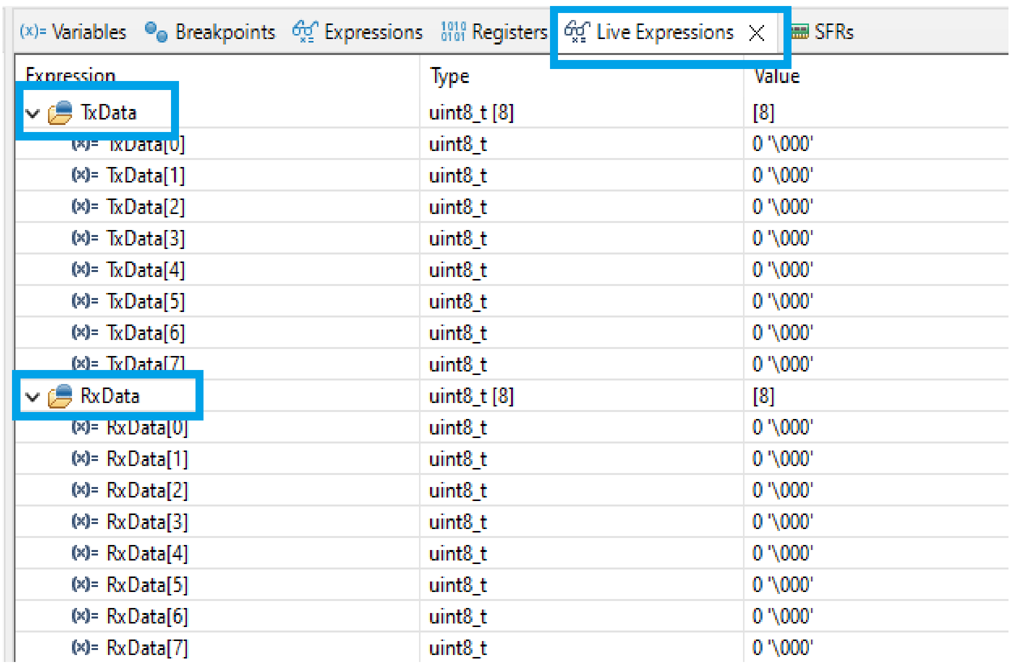


*Figure 22. Start a debug session of the application in STM32CubeIDE*

Wait until the debug to start.

You can add TxData and RxData to the "Live Expressions" view, to check how they update while the application runs:

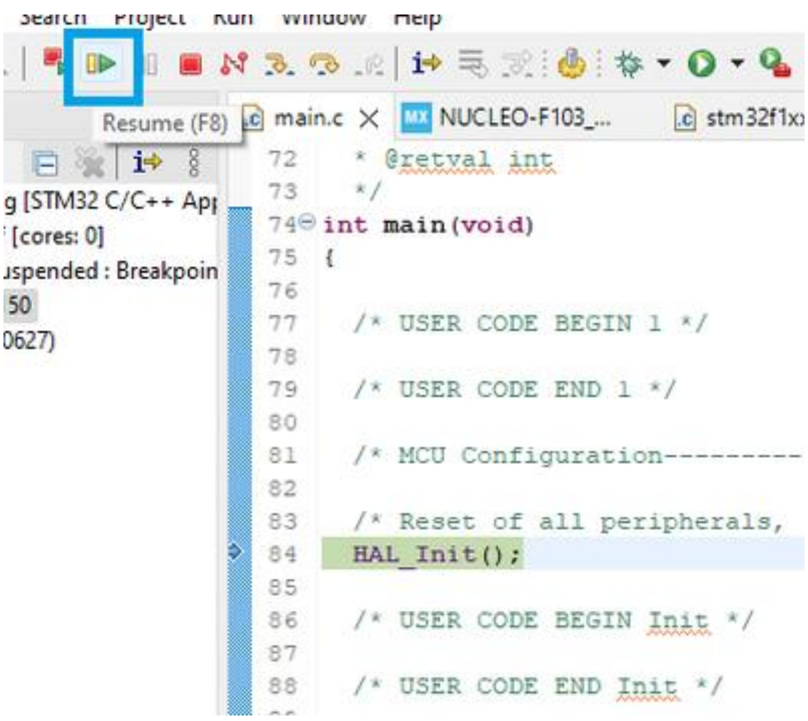


| Expression | Type | Value |
|---|---|---|
| TxData | uint8_t [8] | [8] |
| TxData[0] | uint8_t | 0 '\000' |
| TxData[1] | uint8_t | 0 '\000' |
| TxData[2] | uint8_t | 0 '\000' |
| TxData[3] | uint8_t | 0 '\000' |
| TxData[4] | uint8_t | 0 '\000' |
| TxData[5] | uint8_t | 0 '\000' |
| TxData[6] | uint8_t | 0 '\000' |
| TxData[7] | uint8_t | 0 '\000' |
| RxData | uint8_t [8] | [8] |
| RxData[0] | uint8_t | 0 '\000' |
| RxData[1] | uint8_t | 0 '\000' |
| RxData[2] | uint8_t | 0 '\000' |
| RxData[3] | uint8_t | 0 '\000' |
| RxData[4] | uint8_t | 0 '\000' |
| RxData[5] | uint8_t | 0 '\000' |
| RxData[6] | uint8_t | 0 '\000' |
| RxData[7] | uint8_t | 0 '\000' |

*Figure 23. Adding TxData and RxData to the Live Expressions in STM32CubeIDE*

In the logic analyzer, click the **[Start]** button to start the frame capture.

Now, run the application from STM32CubeIDE by clicking the green triangle:



```
   * @retval int
   */
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration---------

  /* Reset of all peripherals,
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
```

*Figure 24. Running the application on STM32CubeIDE*

The logic analyzer starts capturing the CAN frames sent on CAN_Tx pin (PB9).

These are the frames captured by the logic analyzer:

Zoom out of the capture:

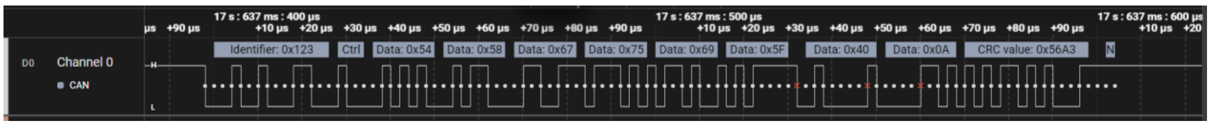


*Figure 25. Zoom-out of the CAN frames captured by the logic analyzer*

In the first frame sent by the application, the analyzer indicated that the frame ID is 0x123. The first byte sent in the frame is 0x54 and the last data sent is 0x0A. Which corresponds well to what the application is doing.

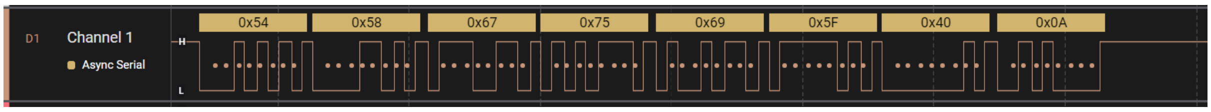
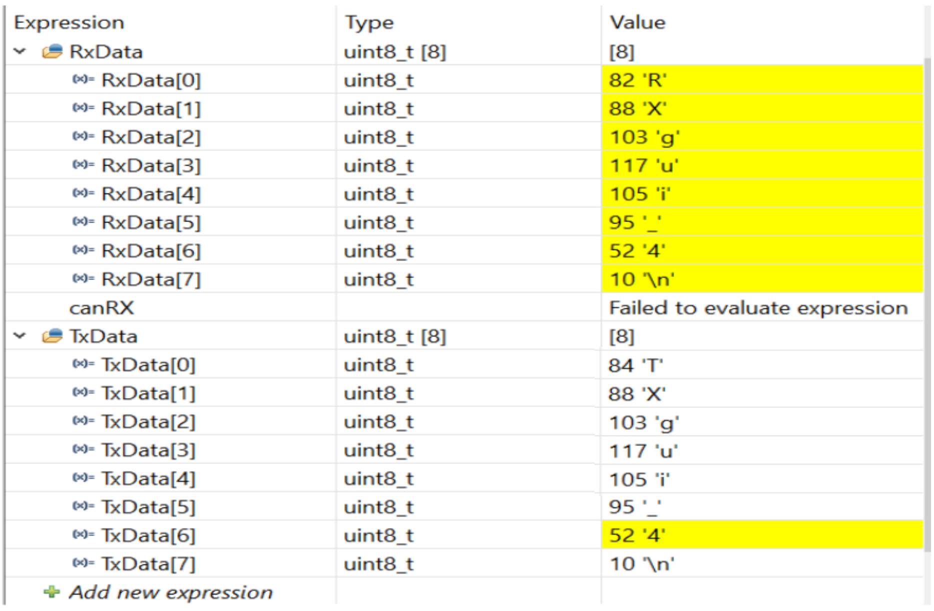


*Figure 26. Zoom-out of the UART frames captured by the logic analyzer*

Note that the pulses related to the zeroed data correspond to the bit stuffing.

In the debugger we can see that RxData is changing in live in the “Live Expressions”:

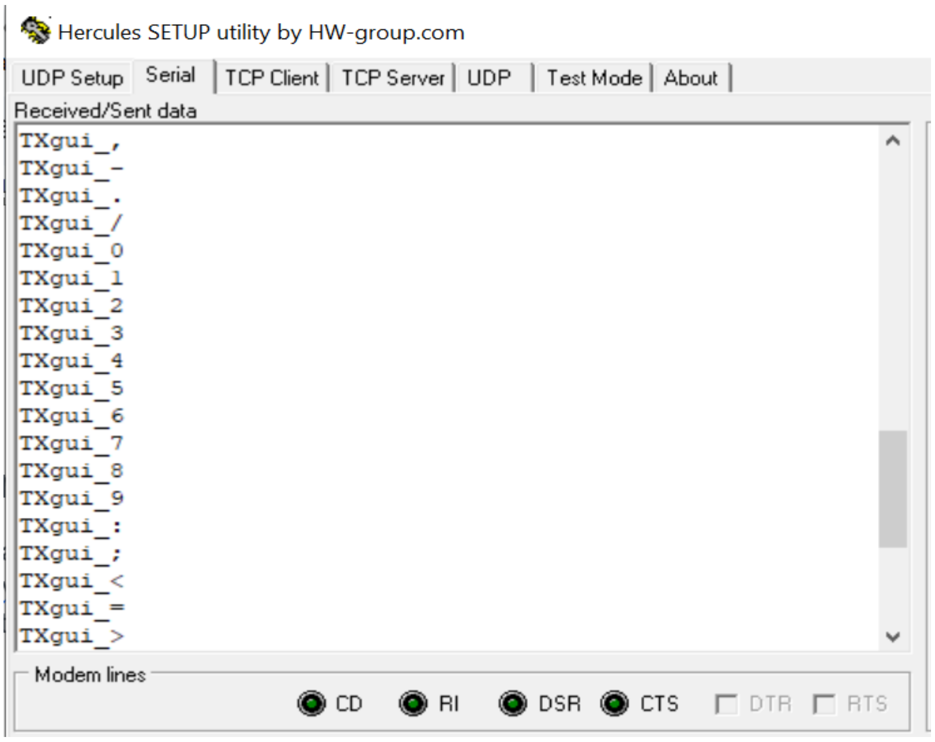
## 4.2. Results



| Expression | Type | Value |
|---|---|---|
| RxData | uint8_t [8] | [8] |
| RxData[0] | uint8_t | 82 'R' |
| RxData[1] | uint8_t | 88 'X' |
| RxData[2] | uint8_t | 103 'g' |
| RxData[3] | uint8_t | 117 'u' |
| RxData[4] | uint8_t | 105 'i' |
| RxData[5] | uint8_t | 95 '_' |
| RxData[6] | uint8_t | 52 '4' |
| RxData[7] | uint8_t | 10 '\n' |
| canRX | | Failed to evaluate expression |
| TxData | uint8_t [8] | [8] |
| TxData[0] | uint8_t | 84 'T' |
| TxData[1] | uint8_t | 88 'X' |
| TxData[2] | uint8_t | 103 'g' |
| TxData[3] | uint8_t | 117 'u' |
| TxData[4] | uint8_t | 105 'i' |
| TxData[5] | uint8_t | 95 '_' |
| TxData[6] | uint8_t | 52 '4' |
| TxData[7] | uint8_t | 10 '\n' |
| Add new expression | | |

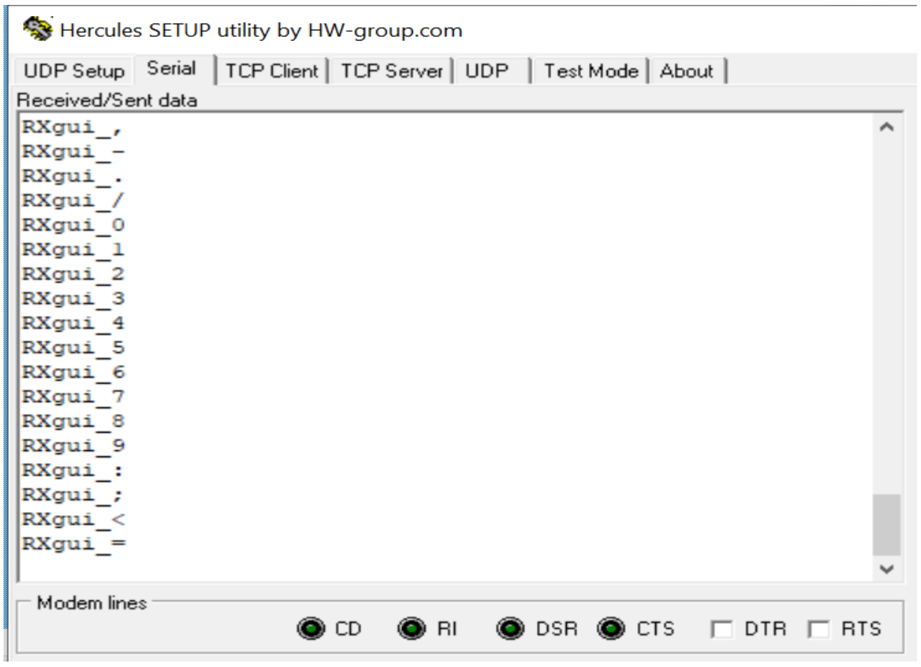
*Figure 27. Tx and Rx buffers are changing in live in the STM32CubeIDE debugger*

Turn on Hercules, choose”Serial” and set the appropriate COM and set baudrate is 115200 bit/s.



```
TXgui_,
TXgui_-
TXgui_.
TXgui_/
TXgui_0
TXgui_1
TXgui_2
TXgui_3
TXgui_4
TXgui_5
TXgui_6
TXgui_7
TXgui_8
TXgui_9
TXgui_:
TXgui_;
TXgui_<
TXgui_=
TXgui_>
```

*Figure 28. Rx buffers are sending in live in the Hercules terminal (second ECU)*



```
RXgui_,
RXgui_-
RXgui_.
RXgui_/
RXgui_0
RXgui_1
RXgui_2
RXgui_3
RXgui_4
RXgui_5
RXgui_6
RXgui_7
RXgui_8
RXgui_9
RXgui_:
RXgui_;
RXgui_<
RXgui_=
```

*Figure 29. Rx buffers are sending in live in the Hercules terminal (first ECU)*

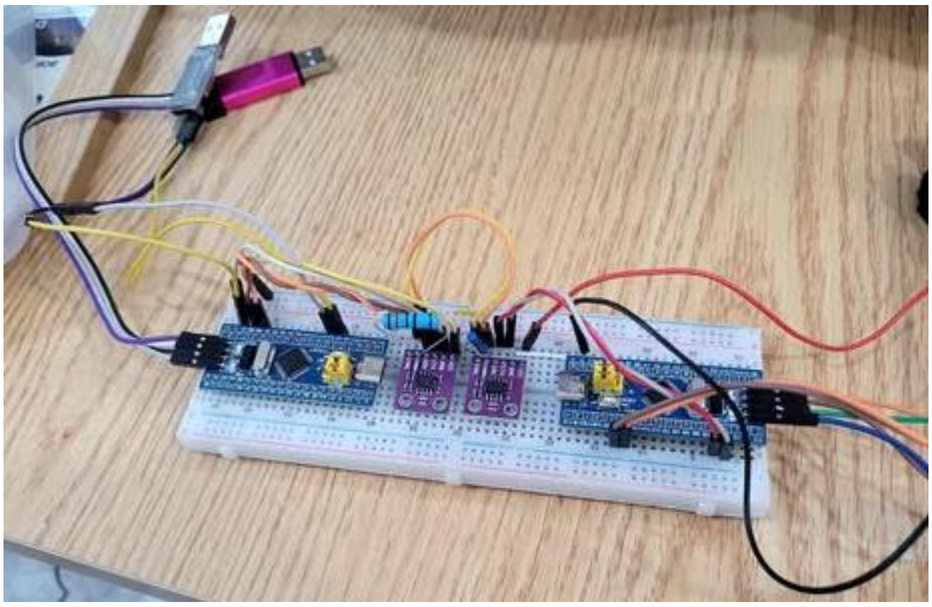


*Figure 30. Circuit*

## Conclusion

This article described step by step how to configure the CAN in Normal mode to receive a CAN frame in interrupt mode between 2 ECU and display on PC through UART. It showed how to run the CAN application and how to test with a logic analyzer.

## Reference

Guide to CAN (bxCAN/CAN2.0) configuration in Loop ... - STMicroelectronics Community

Giao tiếp CAN STM32 - Thực hành cơ bản - TAPIT