SPC58EC CAN bus configuration

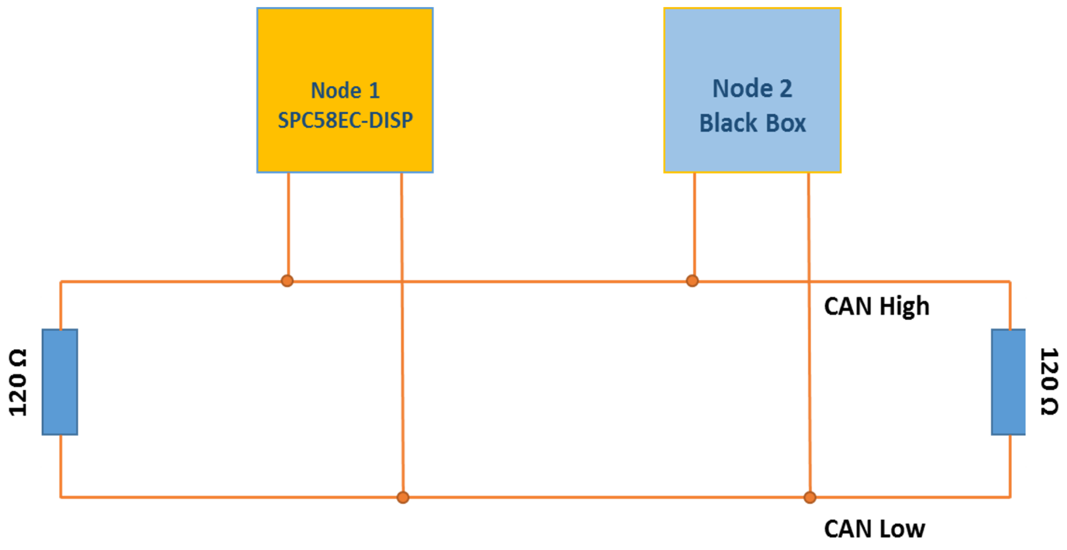
Introduction

This application note is intended for software developers wishing to develop an application with CAN communication.

The aim of this application example is to transmit messages through a CAN controller from Node 1, implemented with STMicroelectronics SPC58ECx, to another Node 2 exposing a CAN controller. The assumption is that Node 2 implementation is unknown so it will be treated as a black box.

It is assumed that the reader has knowledge of embedded MCU architecture and basic principles of embedded software development with C language.

Figure 1. **Basic application principle**



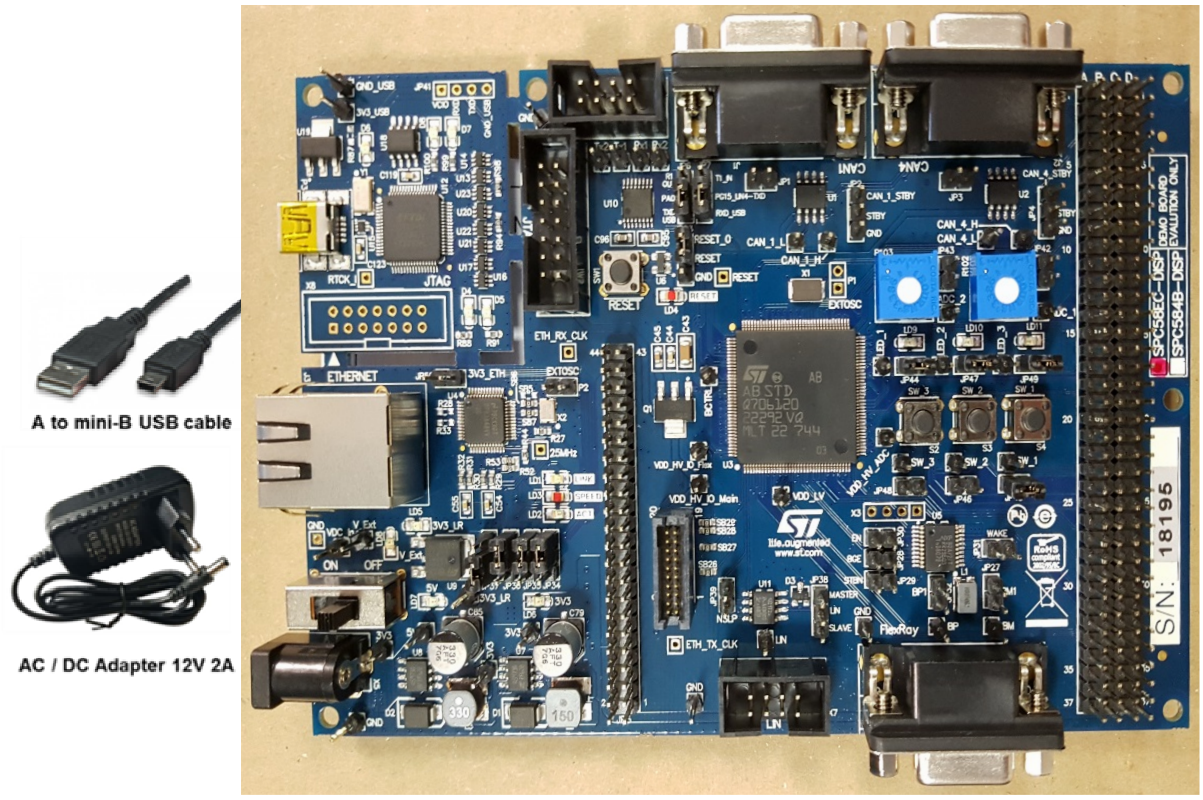
1 Hardware overview

To implement the example application, the following components have been used:

- SPC58EC-DISP discovery board hosting a 4MB Flash memory 32-bit SPC58 automotive microcontroller. SPC58EC-DISP represents the NODE 1 of our example.
- One 1 A / 12 V power supply for SPC58EC-DISP.
- One mini-B USB to USB Type-A cable to connect the MCU programmer.

For information on the software package, see dedicated Section 3 Create a new project with SPC5-Studio, using the application wizard.

Figure 2. SPC58EC-DISP



2 CAN subsystem overview

The SPC58, used in the above board, hosts 8 CAN controllers.

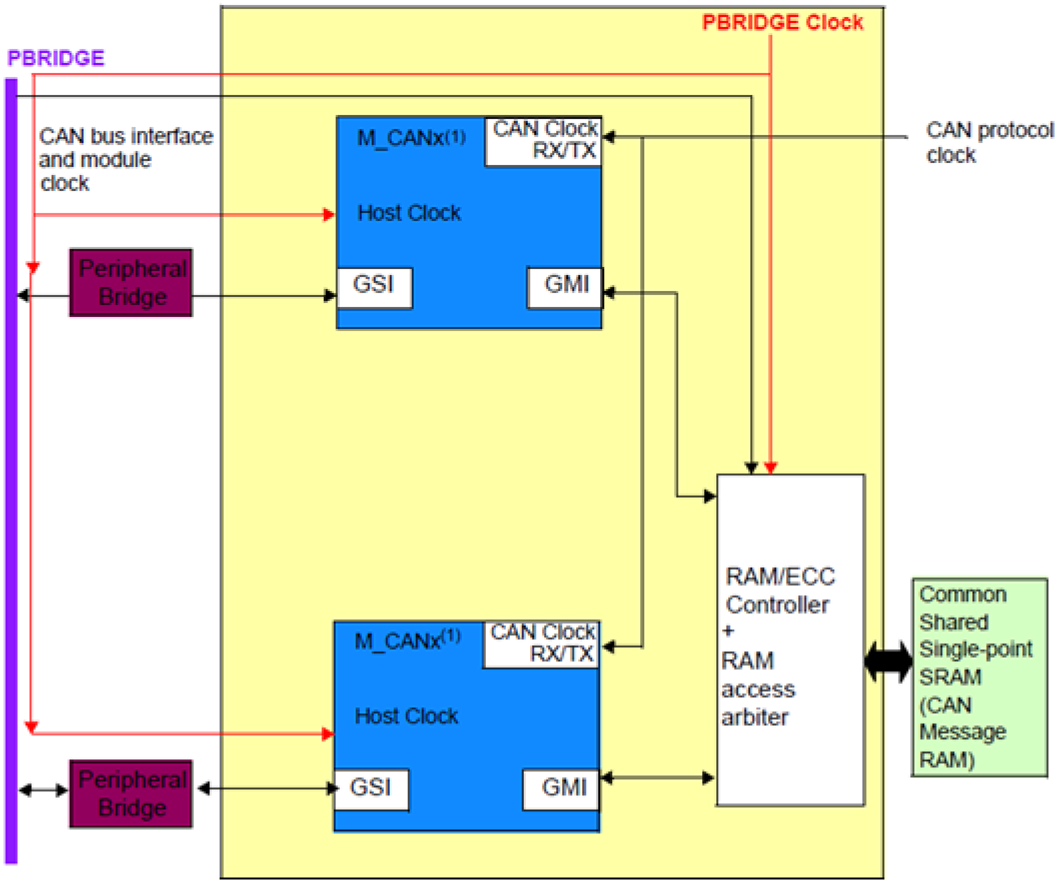
These controllers are organized into two subsystems to optimize hardware system resources.

All the embedded CAN controllers present in the same subsystem will share resources like RAM memory, clock, etc.

The CAN subsystem consists of the following major blocks:

- **Modular CAN cores**: the registers of the CAN module can be accessed using the generic slave interface (GSI). The peripheral GSI module enable acts as a request from each master.
- **CAN-RAM arbiter**: is an additional logic for arbitration between the requests for RAM access from the various CAN controllers.
- **SRAM**: the CAN subsystem will interface with an external RAM using this interface.
- **ECC Controller**: it contains the logic to compute and validate the correction code on the SRAM memory cells.

Figure 3. **CAN subsystem generic block diagram**



Each controller needs to be configured choosing parameters values base on the network where the controller is connected. As shown in Table 1. Subsystems below, the first four CAN controllers belong to Subsystem0 and the remaining controllers belong to Subsystem1.

Table 1. **Subsystems**

Subsystem0	
CAN1	MCAN0
CAN2	MCAN1
CAN3	MCAN2
CAN4	MCAN3
Subsystem1	
CAN7	MCAN1
CAN8	MCAN2
CAN9	MCAN3
CAN10	MCAN4

In the application example described in this document, the transmission and reception of a message via CAN2 (CAN2 supports CANFD) will be demonstrated. If you want to use other controllers, please refer to the SPC58ECx documentation.

At this point, it is also important to give the user an overview of the Message RAM.

The message storage is intended to be a single-ported Message RAM outside of the module. Depending on the selected device, multiple M_CAN controllers can share the same Message RAM.

All functions concerning the handling of messages are implemented by the RX Handler and the TX Handler.

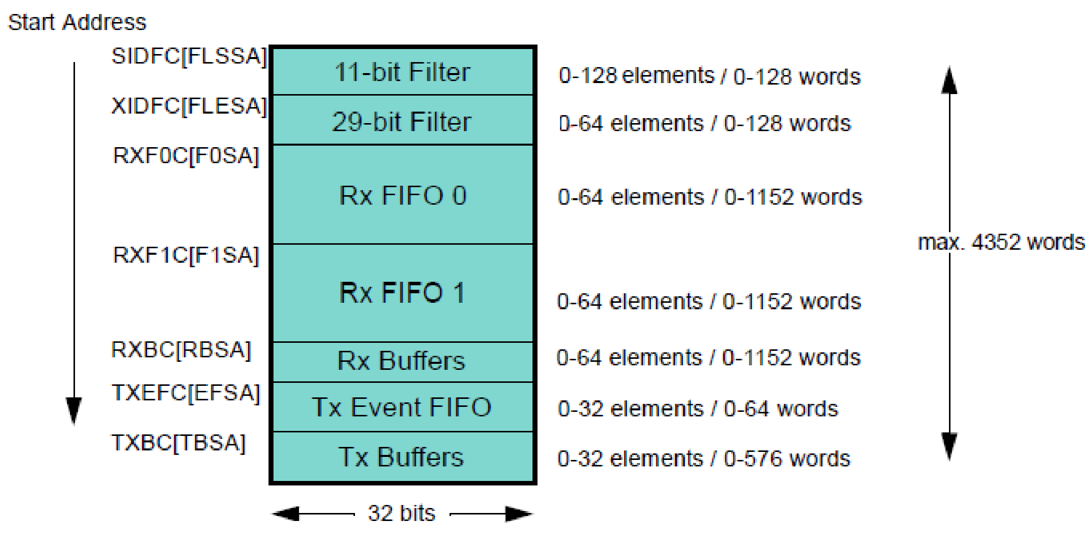
The RX Handler manages message acceptance filtering, the transfer of received messages from the CAN core to the Message RAM and it also provides receive message status information.

The TX Handler is responsible for the transfer of transmit messages from the Message RAM to the CAN core and it also provides transmit status information.

Since the Message RAM is equipped with the ECC functionality, it is recommended to initialize the Message RAM after hardware reset by writing 0x0.

This avoids that reading from uninitialized Message RAM sections will activate interrupt IR.BEC (Bit Error Corrected) or IR.BEU (Bit Error Uncorrected).

Figure 4. **Message RAM configuration**


Start Address
SIDFC[FLSSA]
XIDFC[FLESA]
RXF0C[F0SA]
RXF1C[F1SA]
RXBC[RBSA]
TXEFC[EFSA]
TXBC[TBSA]
11-bit Filter
29-bit Filter
Rx FIFO 0
Rx FIFO 1
Rx Buffers
Tx Event FIFO
Tx Buffers
0-128 elements / 0-128 words
0-64 elements / 0-128 words
0-64 elements / 0-1152 words
0-64 elements / 0-1152 words
0-64 elements / 0-1152 words
0-32 elements / 0-64 words
0-32 elements / 0-576 words
max. 4352 words
32 bits

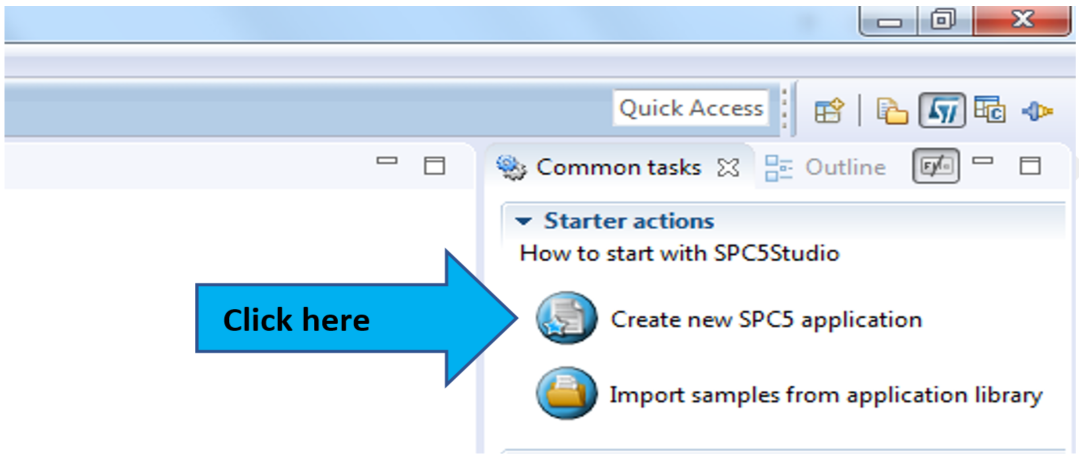
3 Create a new project with SPC5-Studio, using the application wizard

The previous chapters have briefly described the CAN protocol, the CAN controllers and the purpose of the present application example.

This chapter describes the methods and steps to follow to create a new project with SPC5-Studio. SPC5-Studio is based on Eclipse Engine. SPC5-Studio allows the user to create an entirely new project or rely on the pre-developed demo related to the specific MCU supported and the reference driver.

The creation of a new project with SPC5-Studio can be summarized in the following figure:

Figure 5. **SPC5-Studio: new application wizard tab**


Quick Access
Common tasks
Outline
Starter actions
How to start with SPC5Studio
Click here
Create new SPC5 application
Import samples from application library

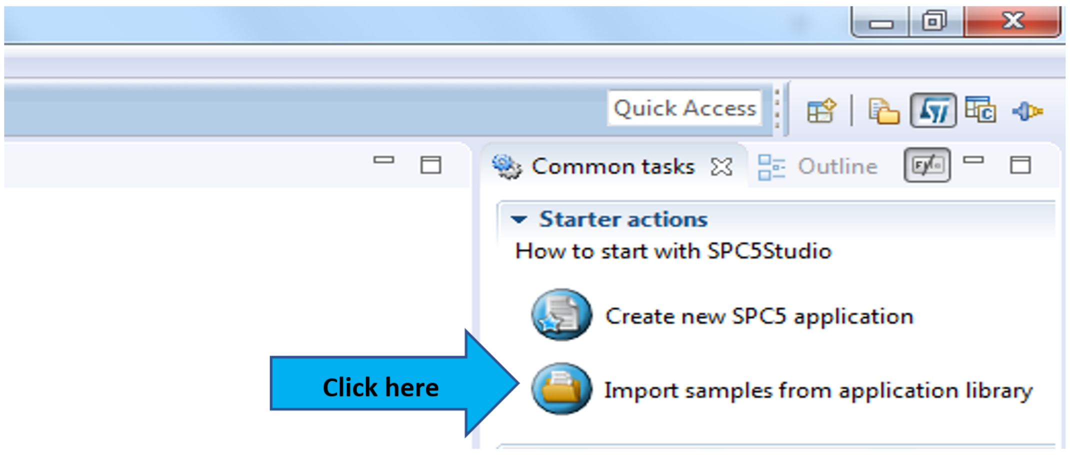
This example relies on the CAN driver demo available for the SPC58EC microntroller. The demo is used as a starting point for this application. Then, the demo will be fine tuned by changing the fields and source code to adapt it to this application example.

The import through the wizard procedure in SPC5-Studio can be summarized in three steps:

Step1. Import a sample application

To import a sample application, click on "Import samples from application library" as highlighted below and then go to step 2.

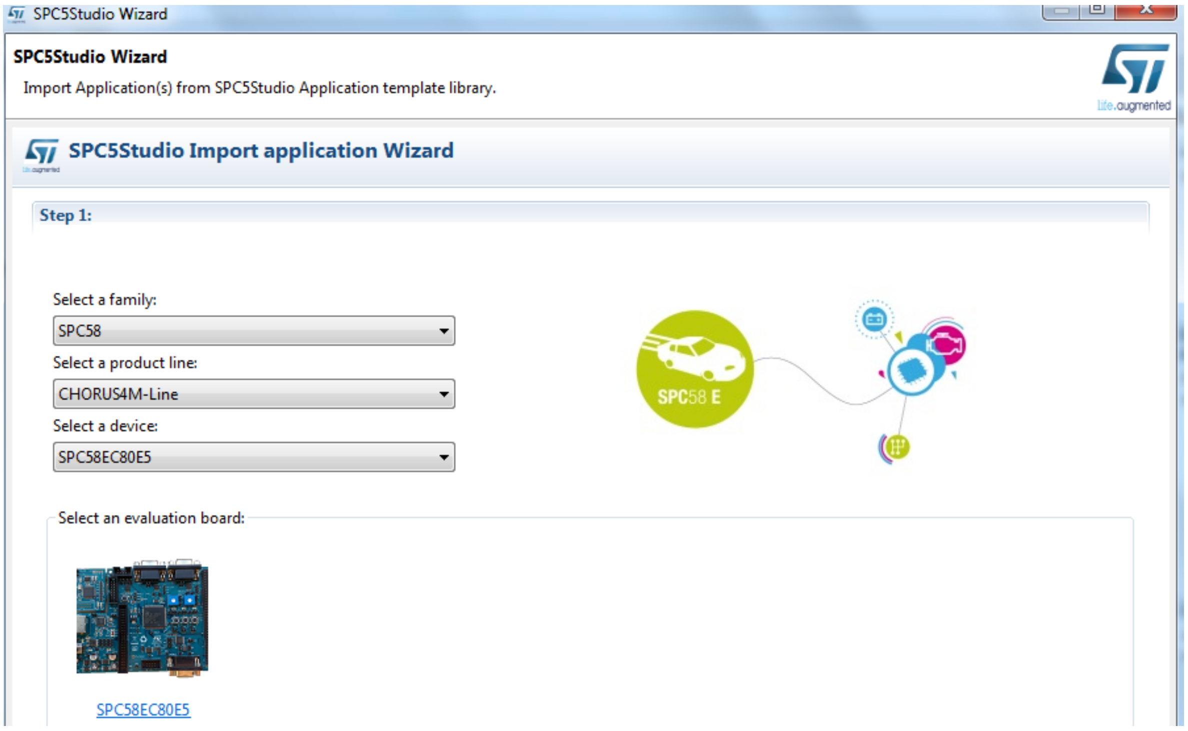
Figure 6. **SPC5-Studio: new application import from library tab**


Quick Access
Common tasks
Outline
Starter actions
How to start with SPC5Studio
Create new SPC5 application
Click here
Import samples from application library

Step2. Select the platform through the application wizard

Select a family, a product line and a device as shown in the below Figure 7:

Figure 7. **SPC5-Studio: device selection menu**



SPC5Studio Wizard

Import Application(s) from SPC5Studio Application template library.

SPC5Studio Import application Wizard

Step 1:

Select a family: SPC58

Select a product line: CHORUS4M-Line

Select a device: SPC58EC80E5

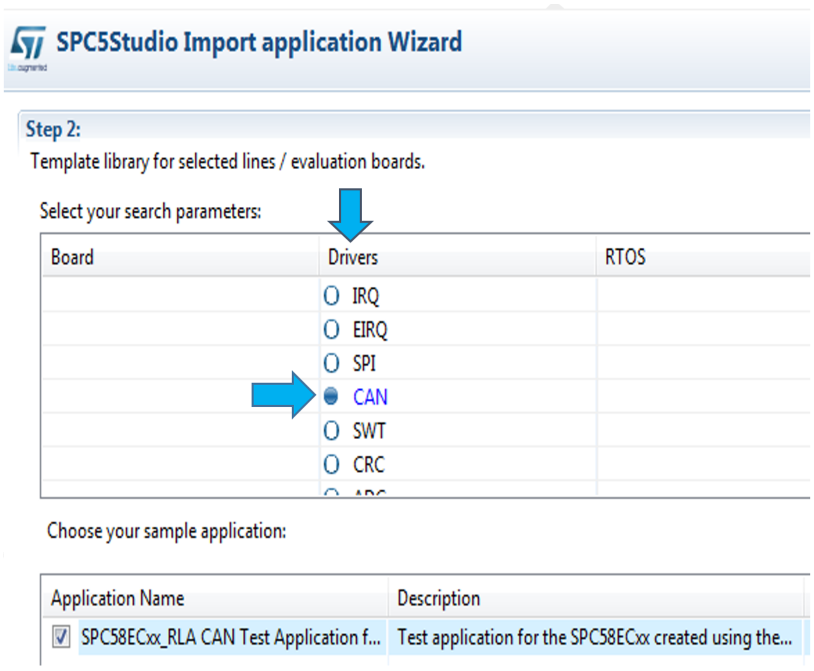
Select an evaluation board: SPC58EC80E5

For our application, the device is SPC58C80ES

Step3. Select the driver

In this step you can select the driver family and the specific application example. For this example application, the CAN driver and the CAN Test Application will be selected.

Figure 8. **SPC5-Studio: application drivers selection tab**



SPC5Studio Import application Wizard

Step 2:

Template library for selected lines / evaluation boards.

Select your search parameters:

Board	Drivers	RTOS
	IRQ	
	EIRQ	
	SPI	
	● CAN	
	SWT	
	CRC	

Choose your sample application:

Application Name	Description
✓ SPC58ECxx_RLA CAN Test Application f...	Test application for the SPC58ECxx created using the...

The selection of the demo application is now completed and the code in this project can be imported.

4 PIN configuration

This chapter explains how to select and configure the microcontroller pins needed for our application example.

Some I/O pins of SPC58ECx can be identified as ports. The pins are the physical connections, only one specific function may be associated to a specific pin.

The port is a logical abstraction of the pin in the sense that it is possible to associate one at the time a list of functions to the PIN through the internal configuration of the MCU registers.

A specific plug-in called the PinMap Editor is available in the SPC5-Studio: this is an intuitive graphical interface dedicated to the microcontroller's pin configuration.

The PinMap Editor monitors the configuration procedure step by step. In particular, it supports the user by disabling the options not compatible with the selected driver.

Here is a brief summary of the pin configuration procedure:

1. Select the driver, e.g. CAN1;
2. Select the signal by clicking on the driver. A list of signals to be configured will appear, e.g. in our example RX and TX for the CAN;
3. For each signal select the pin to configure. Once selected, the pin is highlighted in the PinMap GUI;
4. Right-click on the highlighted pin to select the signal direction: input, output or I/O;
5. If the above configuration is properly set, it will be possible to select the pre-loaded configuration related to the specific signal.

One of the main advantages of using SPC5-Studio is the automatic code generation based on the pin configuration settings.

Pins are grouped by function or by driver, so two pin selection modes can be distinguished:

1. Pin selection from schematic;
2. Pin selection from function.

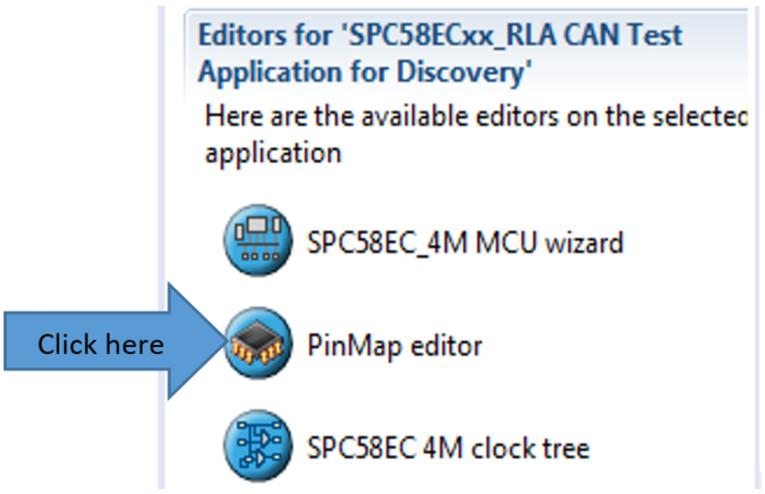
4.1 Case 1. Pin selection from schematic

In this case it is assumed that the user knows which pins to activate, which signal to associate for each of them and the direction of these signals. You can just click on the chosen pin on the PinMap and proceed with the desired configuration settings.

Step1. Open the PinMap editor:

You can use the PinMap by clicking the icon shown in the below figure:

Figure 9. SPC5-Studio: open PinMap editor

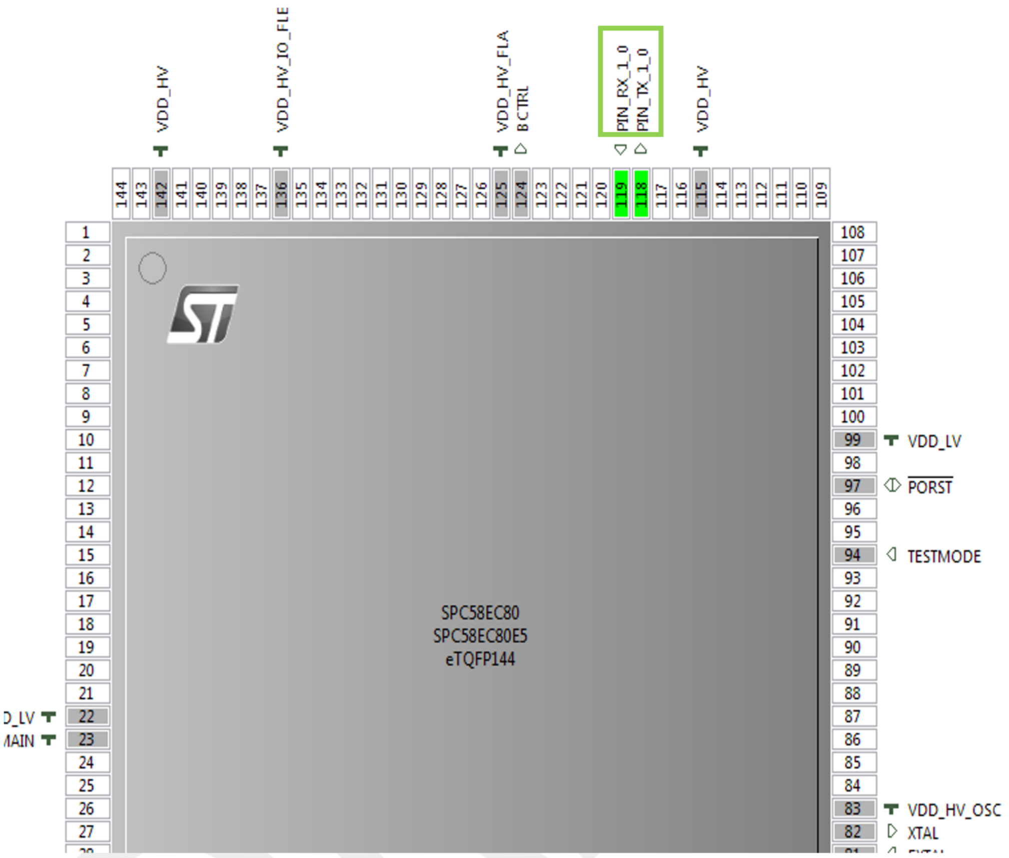


Editors for 'SPC58ECxx_RLA CAN Test Application for Discovery'

Here are the available editors on the selectec application

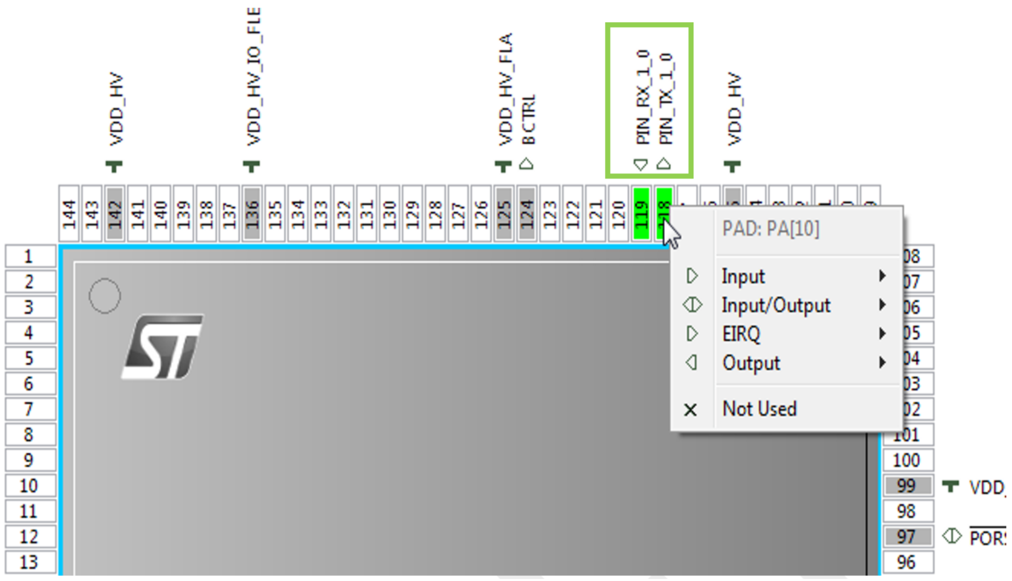
Step2. Pin selection:

Figure 10. **SPC5-Studio: pin map wizard**



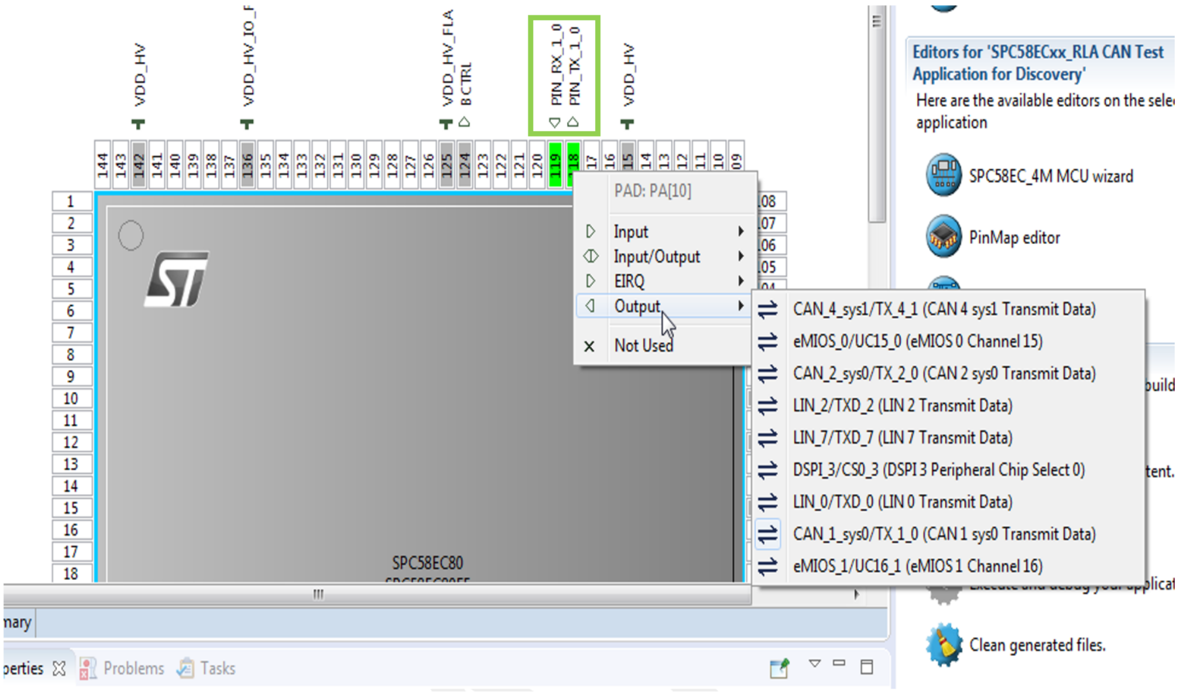
Select the pin and click on the right option to select the signal direction: input, output or I/O.

Figure 11. **SPC5-Studio: graphical pin configuration menu**



After selecting the signal direction, select the pre-loaded configuration on SPC5-Studio related to the signal that you need for your application. If you commit an error in choosing the pin direction, then you will not find the pre-loaded configuration related to the signal of the chosen driver.

Figure 12. **SPC5-Studio: graphical pin functionality selection**

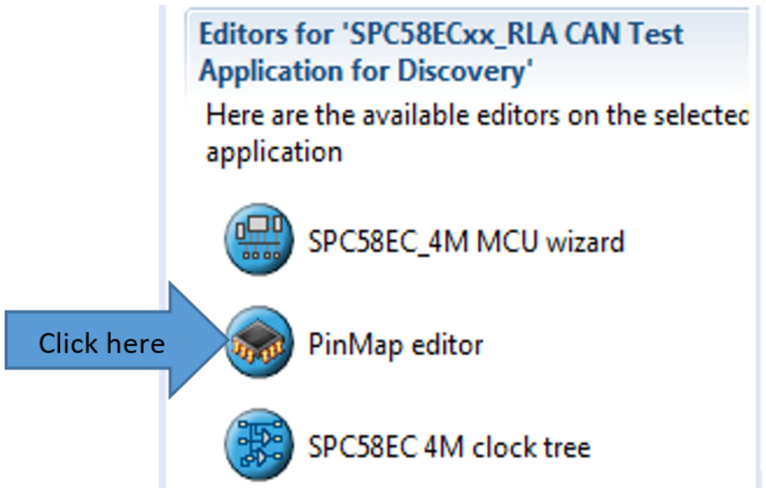

VDD_HV
VDD_HV_IO_F
VDD_HV_FLA
BCTRL
PIN_RX_1_0
PIN_TX_1_0
VDD_HV
Editors for 'SPC58ECxx_RLA CAN Test Application for Discovery'
Here are the available editors on the sele
application
SPC58EC_4M MCU wizard
PinMap editor
PAD: PA[10]
Input
Input/Output
EIRQ
Output
Not Used
CAN_4_sys1/TX_4_1 (CAN 4 sys1 Transmit Data)
eMIOS_0/UC15_0 (eMIOS 0 Channel 15)
CAN_2_sys0/TX_2_0 (CAN 2 sys0 Transmit Data)
LIN_2/TXD_2 (LIN 2 Transmit Data)
LIN_7/TXD_7 (LIN 7 Transmit Data)
DSPI_3/CS0_3 (DSPI 3 Peripheral Chip Select 0)
LIN_0/TXD_0 (LIN 0 Transmit Data)
CAN_1_sys0/TX_1_0 (CAN 1 sys0 Transmit Data)
eMIOS_1/UC16_1 (eMIOS 1 Channel 16)
SPC58EC80
Clean generated files.
Problems
Tasks

4.2 Case 2. Pin selection from function

In this case the pin is selected based on its driver. By clicking on the driver, you will see a list of signals to be configured. For each signal, a list of all available pins will be shown. Next step for the user is to associate the chosen pin to the related signal and driver (please see description above).

Step1. Open PinMap editor:

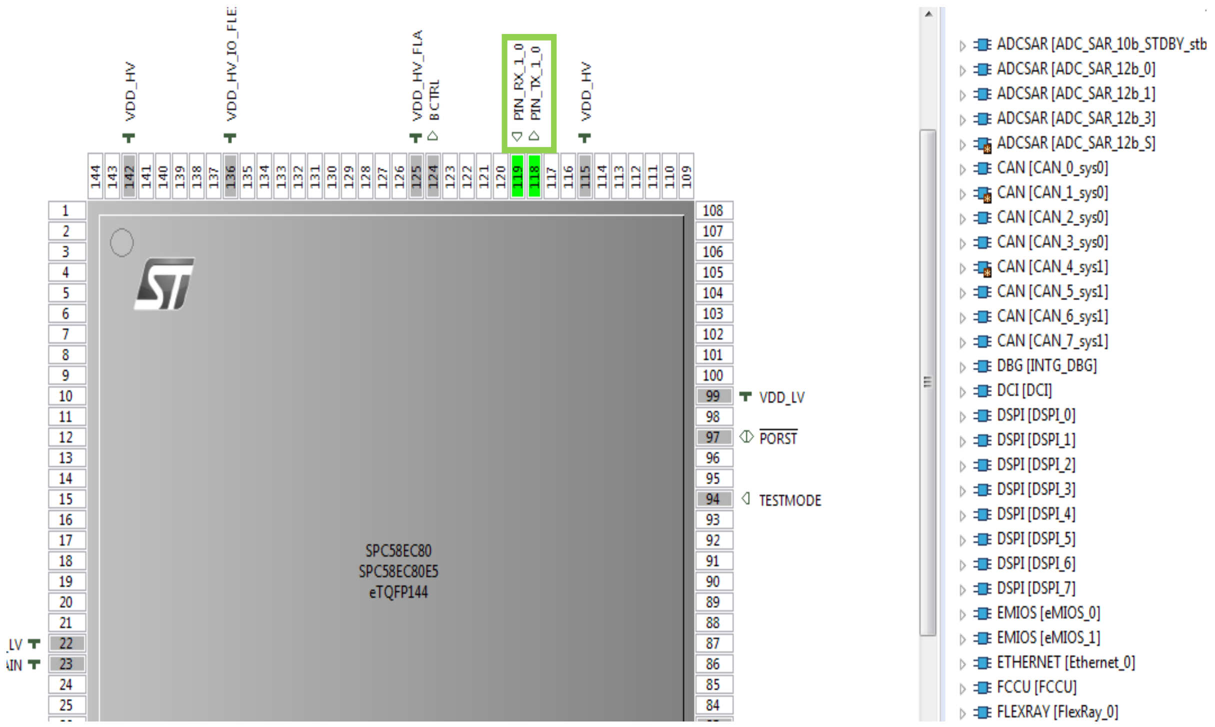
Figure 13. **SPC5-Studio: open pin map editor**


Editors for 'SPC58ECxx_RLA CAN Test Application for Discovery'
Here are the available editors on the selectec application
SPC58EC_4M MCU wizard
Click here
PinMap editor
SPC58EC 4M clock tree

Step2. Pin selection:

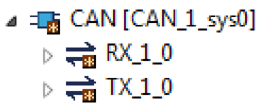
In this case you can choose the pin according to the driver:

Figure 14. **SPC5-Studio: driver pin selection**



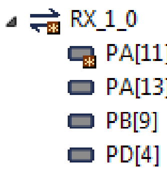
After selecting the driver, you can use the list of signals to choose the signal required to configure the pin:

Figure 15. **SPC5-Studio: driver pin list**



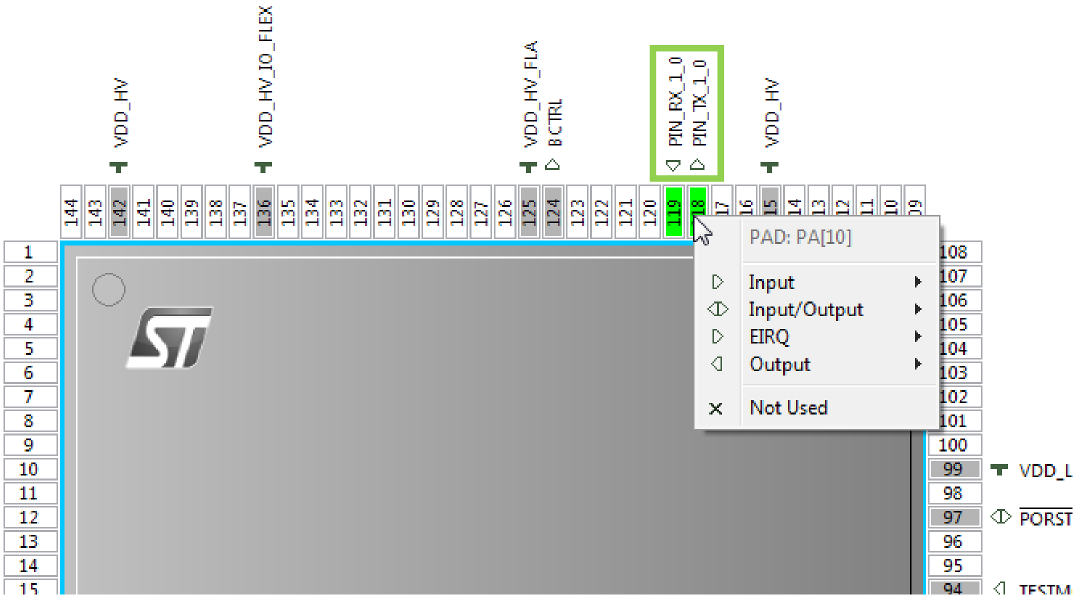
For each signal related to the specific driver all the eligible pins are shown with reference to the microcontroller ports:

Figure 16. **SPC5-Studio: driver pin ports**



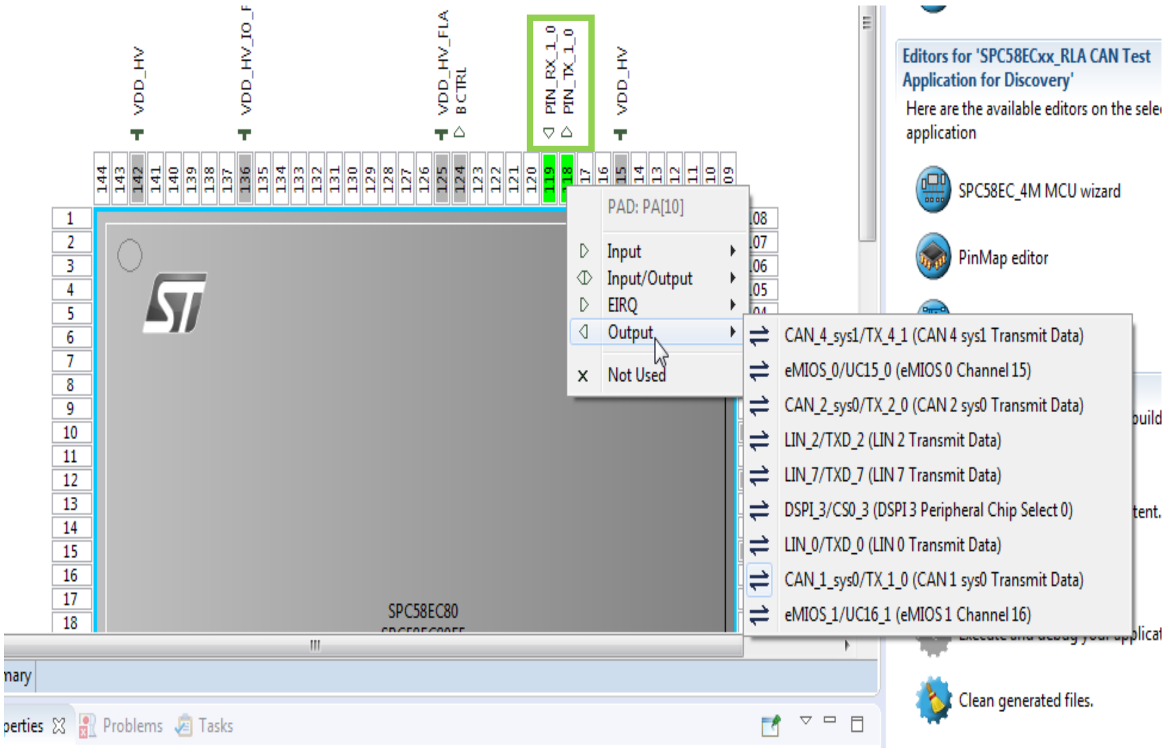
Select the pin and click the right button, then select the signal direction: Input in this specific case.

Figure 17. **SPC5-Studio: pin direction menu**



After selecting the signal direction, select the preloaded configuration.

Figure 18. **SPC5-Studio: pin configuration selection**

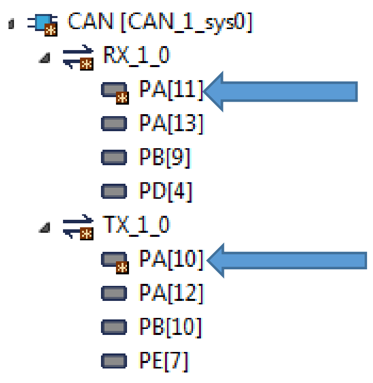


CAN_4_sys1/TX_4_1 (CAN 4 sys1 Transmit Data)
eMIOS_0/UC15_0 (eMIOS 0 Channel 15)
CAN_2_sys0/TX_2_0 (CAN 2 sys0 Transmit Data)
LIN_2/TXD_2 (LIN 2 Transmit Data)
LIN_7/TXD_7 (LIN 7 Transmit Data)
DSPI_3/CS0_3 (DSPI 3 Peripheral Chip Select 0)
LIN_0/TXD_0 (LIN 0 Transmit Data)
CAN_1_sys0/TX_1_0 (CAN 1 sys0 Transmit Data)
eMIOS_1/UC16_1 (eMIOS 1 Channel 16)

For CAN2 (refer to Table 1. Subsystems), there are two signals to set, RX and TX. RX is an input signal and TX is an output signal.

A possible pin configuration for our application is showed in the below Figure 19:

Figure 19. SPC5-Studio: driver pin ports


CAN [CAN_1_sys0]
RX_1_0
PA[11]
PA[13]
PB[9]
PD[4]
TX_1_0
PA[10]
PA[12]
PB[10]
PE[7]

Note: *By default each SPC5-Studio application provides a pin configuration suitable to run fine on the target. In any case, as detailed in this document, you can modify and tune the setup according to your needs using the PinMap wizard.*

5 CAN bus general description

The controller area network (CAN bus) is a robust vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer.

It is a message-based protocol, designed originally for multiplexing electrical wires within automobiles to save on copper, but it is also used in many other contexts.

CAN is a multi-master serial bus standard for connecting electronic control units also known as nodes.

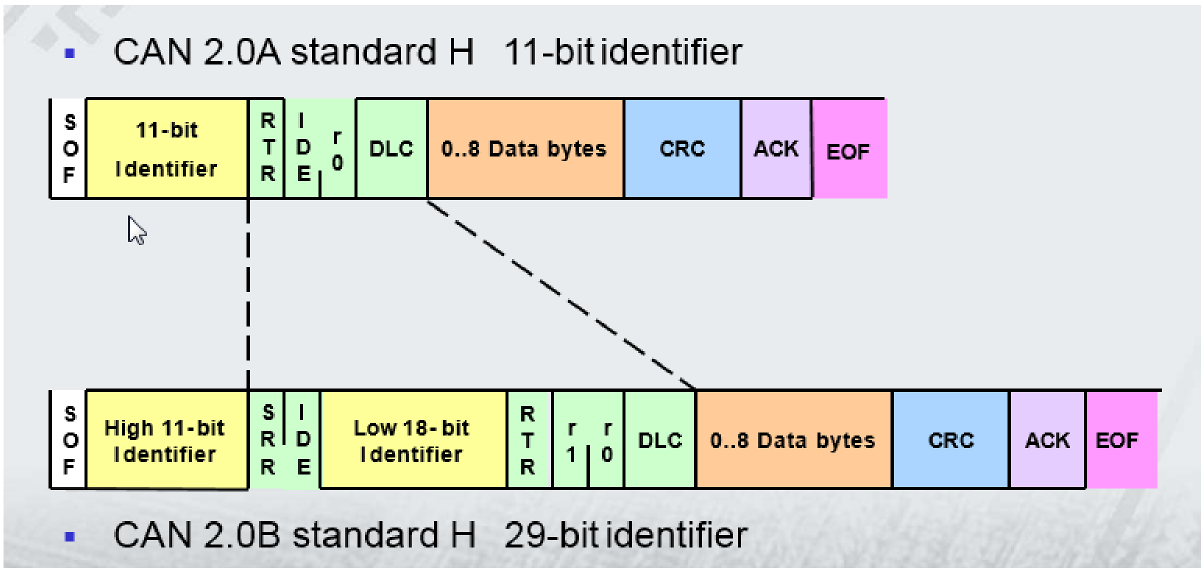
Two or more nodes are required on the CAN network to communicate. The complexity of the node can range from a simple I/O device up to an embedded computer with a CAN interface and sophisticated software.

All nodes are connected to each other through a two-wire bus. A network consists of a twisted pair terminated on both sides with a 120 Ω resistor (see Figure 1).

Each node is able to send and receive messages, but not simultaneously. A message (or frame) consists mainly of three parts: arbitration, control and data fields. The arbitration field contains the message identifier that represents the message priority (the lowest identifiers values have the highest priority).

The standard CAN message has a maximum of 8-byte data length. The CAN-FD extends the data frame length to 64 bytes per frame. The following figure shows the CAN standard and the extended frames:

Figure 20. **CAN frame standard vs extended data frames**



The nodes will always listen and receive all of the messages, therefore in the CAN protocol it is not possible to send messages to a specific node.

The CAN transmits data according to a model based on "dominant" and "recessive" bits. A zero bit is called dominant while an 1 bit is called recessive. Dominant bits prevail on recessive bits during the arbitration phase, where two or more nodes start to send frames at the same time.

In the case of contention of a bus the priority of each message needs to be defined.

The data frame is the most common message type, and comprises:

- Arbitration field;
- Control field;
- Data field;
- CRC field;
- Acknowledgment field.

The Arbitration field is the part of the CAN frame that defines the priority of the message.

The priority assignment to messages in the identifier is a CAN feature that makes it particularly attractive for use in a real-time control environment.

Thus, the lower is the message identifier, the higher is its priority.

An identifier consisting entirely of zeros is the message with the highest priority on a network, therefore it is the message on the bus with the longest dominant portion compared to the portion in all other messages.

The CRC field and the ACK field have the task of detecting any errors in the transmission of messages.

The robustness of CAN protocol depends on a different error detection mechanism.

If a sent message is not acknowledged by any other node, it will be transmitted again until it is received correctly.

The error reporting process consists of the following phases:

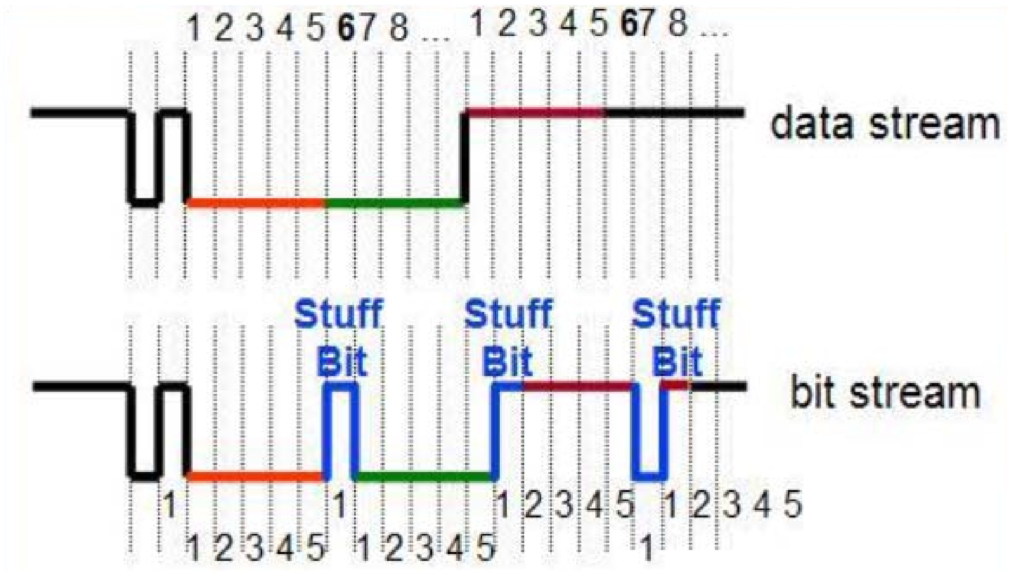
- A CAN controller detects an error (while sending or receiving);
- An error frame is immediately transmitted;
- The message is ignored by all nodes;
- The status of the CAN controller is updated;
- The message is retransmitted.

An error can be detected in 5 ways, 3 of which at message level and 2 at single bit level:

1. **Bit Stuffing Error**: a node during the transmission inserts a bit of opposite polarity after 5 consecutive bits of the same polarity; this is called bit stuffing. Therefore a node that receives more than 5 consecutive bits of equal sign will detect an error of this type.

   This concept is shown in the following Figure 21:

Figure 21. Bit stuffing



2. **Bit Error**: a transmitting node always listens to the bus to check the correspondence with what it is transmitting. If a different bit than the one sent is received, a bit error will be signaled.
3. **Checksum Error**: The checksum is a sequence of bits associated with the transmitted packet, it is used to verify the integrity of a message that can be altered during transmission on the communication channel.

   The simplest type of checksum consists in summing all the bits of the message in transmission and storing the resulting value in the sent frame.

   To verify the integrity of the message, it will be sufficient to do the same summing operation and compare it with the checksum stored in the frame.

   If the two values coincide, the data can be considered complete, otherwise an error message will be signaled.

   Cyclic Redundancy Check (CRC) is a checksum method. The name derives from the output data obtained by processing the input data which are cyclically run through a logical network.

4. **Frame Error**: An error frame is generated by any node that detects a bus error.

   These error frames "destroy" the current data or the remote frame on the bus.

   The transmitting node will know that the message has not been correctly received by all the nodes and will try to send the message again.
5. **Acknowledgment Error**: The ACK field is used to confirm receipt of a valid CAN frame.

   Each node that receives the frame without an error transmits a dominant level in the ACK field and then overwrites the recessive level of the transmitter.

   If a transmitter detects a recessive level in the ACK field, it knows that no receiver has found a valid frame.

   So, if a transmitter does not read back a dominant ACK bit, it will send an acknowledgment error message.

6 Low-level driver setup

Low-level drivers component in SPC5-Studio contains a collection of low-level drivers for Register Level Access (RLA) components. RLA is a feature available in SPC5-Studio to allow easy and direct access to MCU and peripheral registers. The RLA component can be added and configured via the Application wizard.

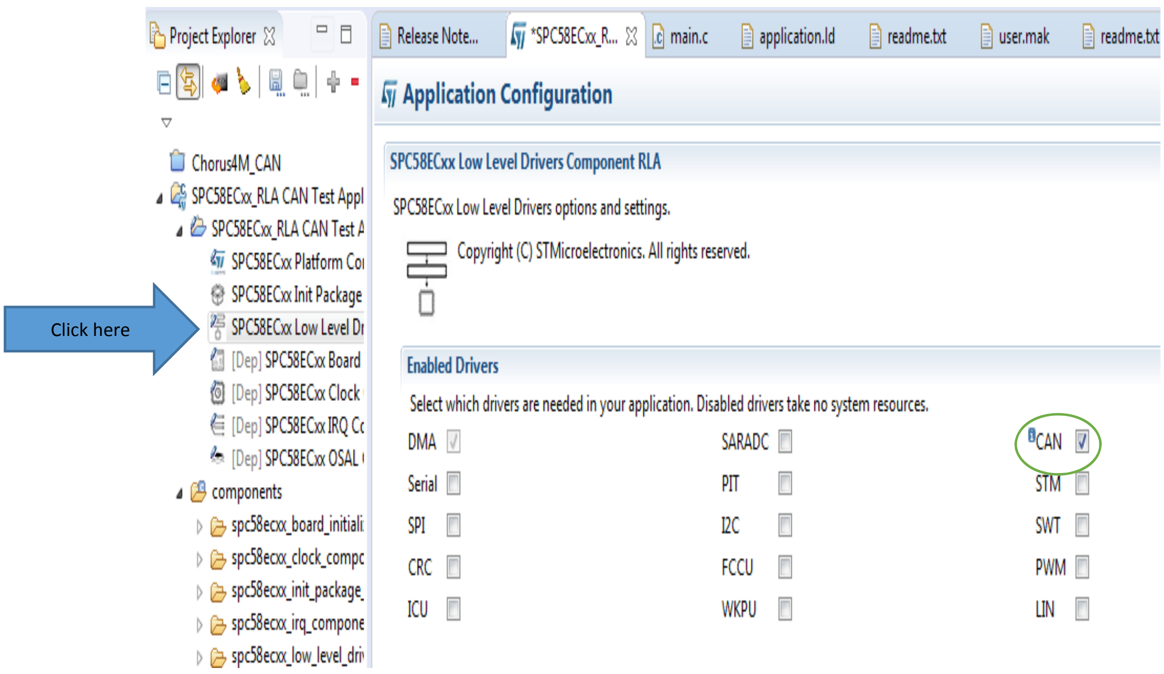
The low-level driver is related of the configuration and choice of the components to add. In this application the chosen component is the CAN controller. The configuration of each component will be explained in detail in the following sections.

Step1. Driver selection:

First you must choose to activate the CAN driver in this application.

Double-click on low-level driver and select the CAN driver.

Figure 22. **SPC5-Studio: project driver selection**



SPC58ECxx Low Level Drivers Component RLA

SPC58ECxx Low Level Drivers options and settings.

Copyright (C) STMicroelectronics. All rights reserved.

Enabled Drivers

Select which drivers are needed in your application. Disabled drivers take no system resources.

DMA	✓	SARADC	☐	CAN	✓
Serial	☐	PIT	☐	STM	☐
SPI	☐	I2C	☐	SWT	☐
CRC	☐	FCCU	☐	PWM	☐
ICU	☐	WKPU	☐	LIN	☐

6.1 CAN component

SPC58EC-DISP has two CAN subsystems implemented, as mentioned in Section 2 CAN subsystem overview:

- CAN Subsystem0
- CAN Subsystem1

Table 1. Subsystems lists the different CAN controllers inside each subsystem.

Note: *All CAN instances support ISO CAN FD mode, the CAN FD will be explored in Section 6.5 CAN-Flexible Data Rate overview.*

The CAN subsystem consists of the modular CAN (M_CAN) modules along with an integrated intelligent CAN RAM controller. The CAN RAM controller consists of additional logic for arbitration between the requests for the RAM access by the various CAN controllers.

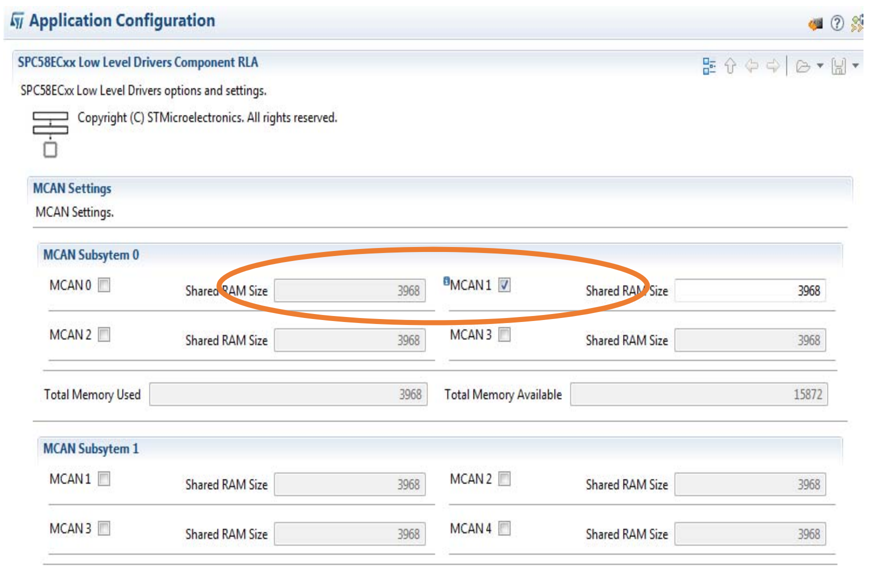
Each subsystem has a shared memory between the four M_CAN controllers in the subsystem.

The configuration of this driver is done as follows:

1. MCAN setting:

   In this step the MCAN is activated as shown:

Figure 23. **SPC5-Studio: controller driver enable tab**



Application Configuration

SPC58ECxx Low Level Drivers Component RLA

SPC58ECxx Low Level Drivers options and settings.

Copyright (C) STMicroelectronics. All rights reserved.

MCAN Settings

MCAN Settings.

MCAN Subsytem 0

MCAN 0	☐	Shared RAM Size	3968	MCAN 1 ☑	Shared RAM Size	3968
MCAN 2	☐	Shared RAM Size	3968	MCAN 3 ☐	Shared RAM Size	3968
Total Memory Used			3968	Total Memory Available		15872

MCAN Subsytem 1

MCAN 1	☐	Shared RAM Size	3968	MCAN 2 ☐	Shared RAM Size	3968
MCAN 3	☐	Shared RAM Size	3968	MCAN 4 ☐	Shared RAM Size	3968

   Double-click on MCAN setting and choose the CAN controllers for at least one subsystem, the CAN2 will be selected in this case.

2. MCAN configuration:

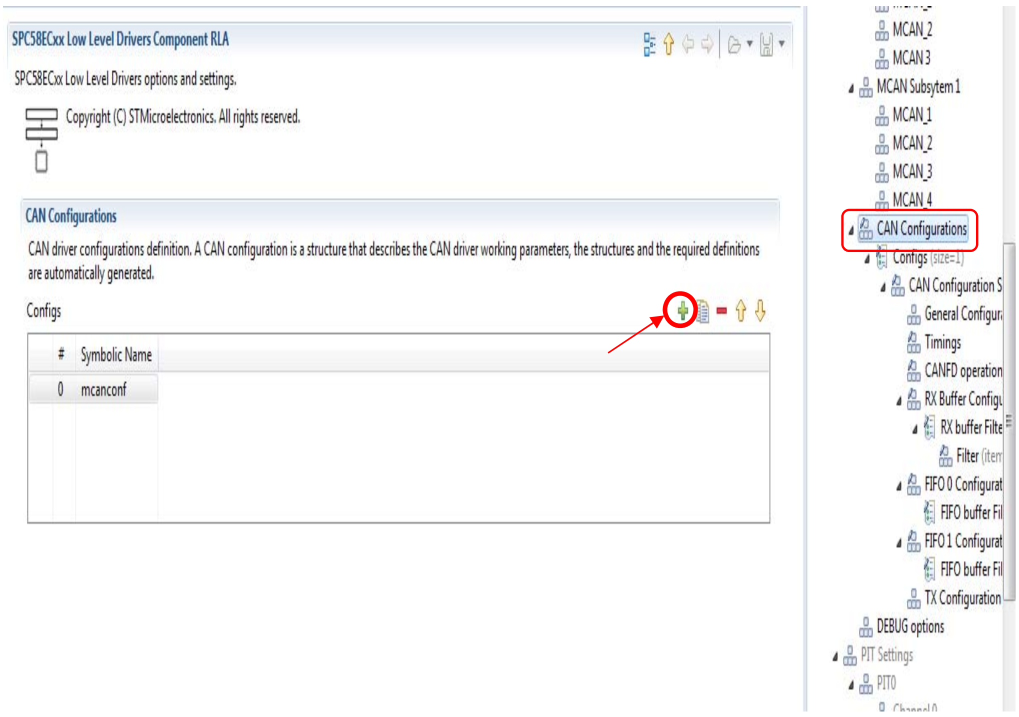
   Clicking on CAN configuration opens a window which allows the user to add and setup a MCAN configuration of that particular subsystem.

   To configure the parameters of the structures that will be necessary for a MCAN configuration, a graphical interface is available.

   These configurations will be converted into codes only after generation. When the code generation command is clicked, all the structures and definitions will be generated automatically.

   For each subsystem there could be a different configuration, which the user can create with SPC5-Studio by clicking on the icon highlighted in the following Figure 24. A new configuration will be created with the following symbolic name: *mcanconf*.

Figure 24. **SPC5-Studio: low-level driver configuration tab**



SPC58ECxx Low Level Drivers Component RLA

SPC58ECxx Low Level Drivers options and settings.

Copyright (C) STMicroelectronics. All rights reserved.

CAN Configurations

CAN driver configurations definition. A CAN configuration is a structure that describes the CAN driver working parameters, the structures and the required definitions are automatically generated.

Configs

#	Symbolic Name
0	mcanconf

In detail by clicking the green icon " ✚ " a window will open in which you can create a new configuration.

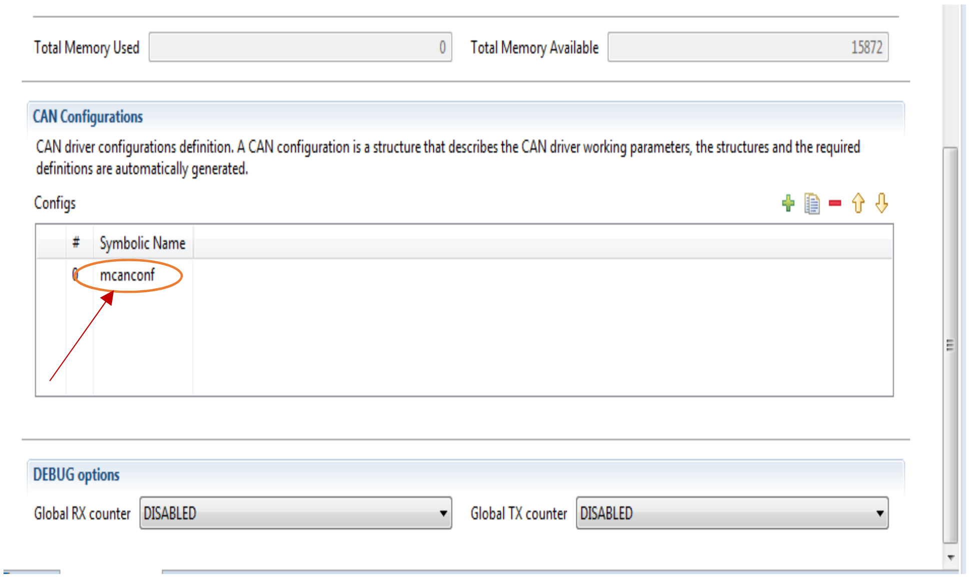
To remove the configuration created you have to select it and then click the red icon "━".

The next sections will describe how you can rename the created configuration.

6.2 CAN configuration setting

This section explains how the user can rename the configuration created.
After creating a new configuration, click on *mcanconf*.

Figure 25. **SPC5-Studio: configurations list**



Total Memory Used 0 Total Memory Available 15872

CAN Configurations

CAN driver configurations definition. A CAN configuration is a structure that describes the CAN driver working parameters, the structures and the required definitions are automatically generated.

Configs

#	Symbolic Name
0	mcanconf

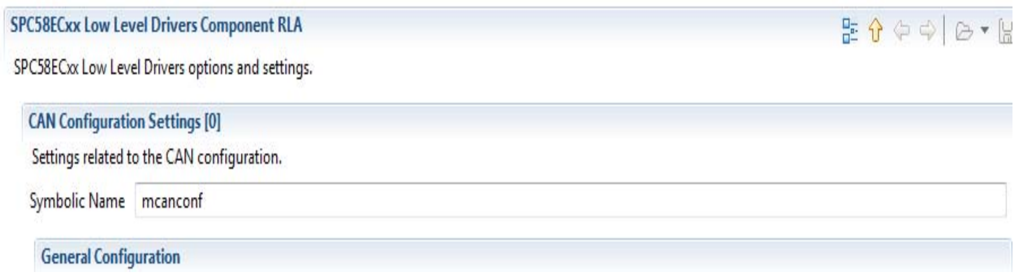
DEBUG options

Global RX counter DISABLED Global TX counter DISABLED

Clicking on the configuration will open the window that contains all the fields needed to configure the MCAN.

This section will analyze only the "CAN Configuration Setting [0]" field, which is useful for modifying the Symbolic Name.

Figure 26. **SPC5-Studio: rename configuration tab**



SPC58ECxx Low Level Drivers Component RLA

SPC58ECxx Low Level Drivers options and settings.

CAN Configuration Settings [0]

Settings related to the CAN configuration.

Symbolic Name: mcanconf

General Configuration

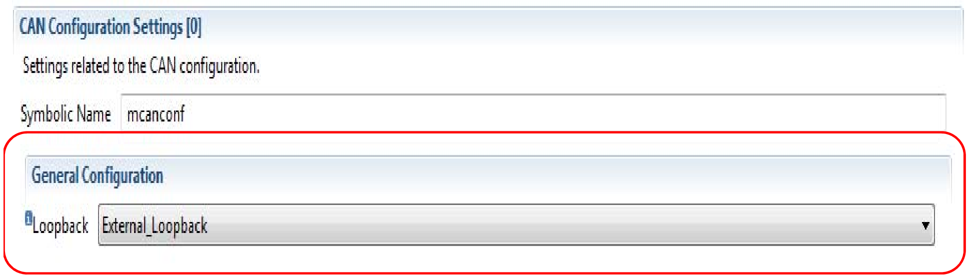
The following fields will require certain theoretical concepts that will be investigated and explained in the following paragraphs, also inherent with the MCAN configuration.

This application uses only the MCAN1 of Subsystem0.

6.3 Loopback

A fundamental debugging mode of CAN controllers is the Loopback mode. Immediately after "CAN Configuration Setting [0]" the user is asked to configure the type of General Configuration to be used.

Figure 27. **SPC5-Studio: CAN driver Loopback mode selection menu**



CAN Configuration Settings [0]

Settings related to the CAN configuration.

Symbolic Name: mcanconf

General Configuration

Loopback: External_Loopback

The Loopback is a feature that allows the CAN controller to talk to itself without sending anything on the Bus. Loopback refers to the routing of electronic signals, to their origin without intentional processing or modification. This is mainly a means of testing the transmission infrastructure.

Loopback is used only for troubleshooting and as a preliminary simulation.

There are three different types of configuration:

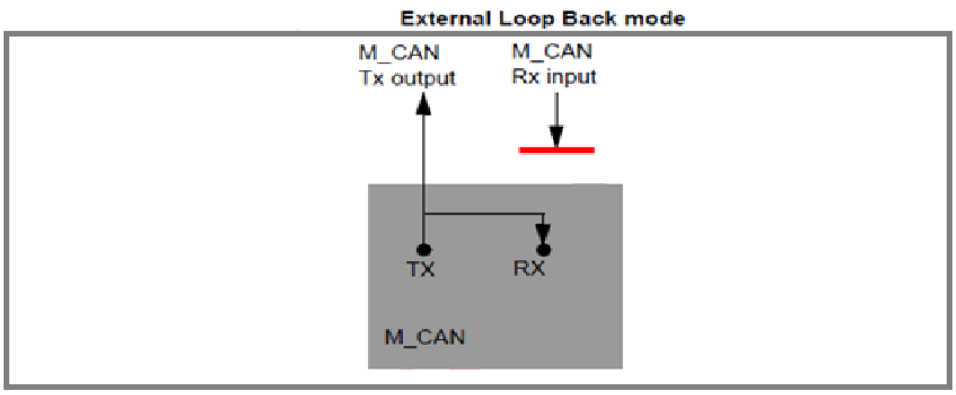
1. **External Loopback**

   In this mode, the CAN module treats its messages as received messages and stores them in FIFO RX, i.e. a set of buffers where the received messages are saved. Figure 28 shows the connection of the TX and RX signals to the MCAN in External Loopback mode. This mode is provided for the hardware self-test.

   To be independent from external stimulation, the M_CAN ignores acknowledge errors in Loopback mode.

   In this mode the actual value of the CAN RX input pin is ignored by the M_CAN. The message is copied in RX and also the transmitted messages can be monitored on the CAN TX output pin.

Figure 28. **External Loopback on SPC584Cx/SPC58ECx**

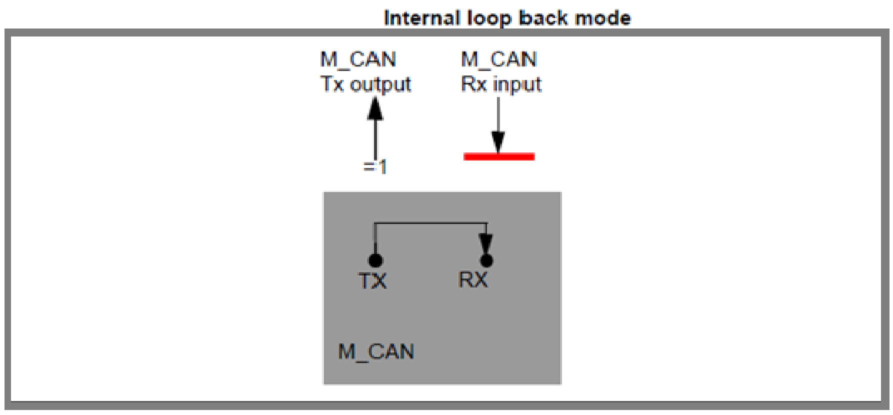


2. **Internal Loopback**

   In this mode the MCAN can be tested without affecting a running CAN system connected to the CAN TX output and RX input pins.

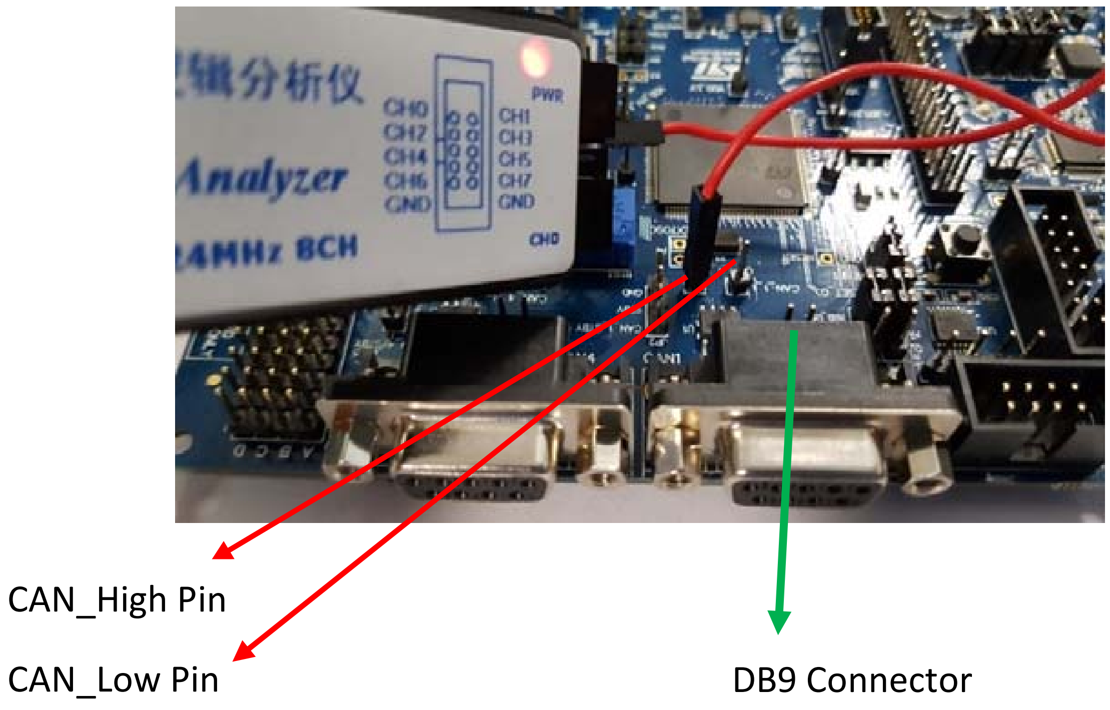
   In this mode MCAN TX pin is disconnected from the MCAN, and MCAN TX pin is held recessive. Figure 29 shows the connection of MCAN TX pin and RX to the M_CAN in case of Internal Loopback mode.

Figure 29. **Internal Loopback on SPC584Cx/SPC58ECx**



3. **No Loopback:**
The real transmission mode is used. Figure 30 shows two different ways to perform a real CAN transmission, via CAN-H and CAN-L pins or via DB9 Connector (also connected to CAN-H and CAN-L).

Figure 30. **How to perform a real CAN transmission**



6.4 Clock and timing

6.4.1 Protocol timing setting

The next step for the configuration of the low-level drivers is the setting of the clock and timing parameters.

This set of parameters allows the baudrate and bit sampling point configuration.

The bit time corresponds to the time that a transmitting node must keep the bus at a constant logical level so that it is recognized by any receiving node.

The CAN protocol divides each bit time into four segments, each of which consists of an integer multiple of the Time Quantum. The Time Quantum is a fixed time reference derived from the period of oscillation of the protocol clock.

The Clock Prescaler is a frequency regulator. The user can tune the bit rate by using a proper divisor value (stored in the Clock Prescaler register).

The three segments that make up the bit rate are shown below:

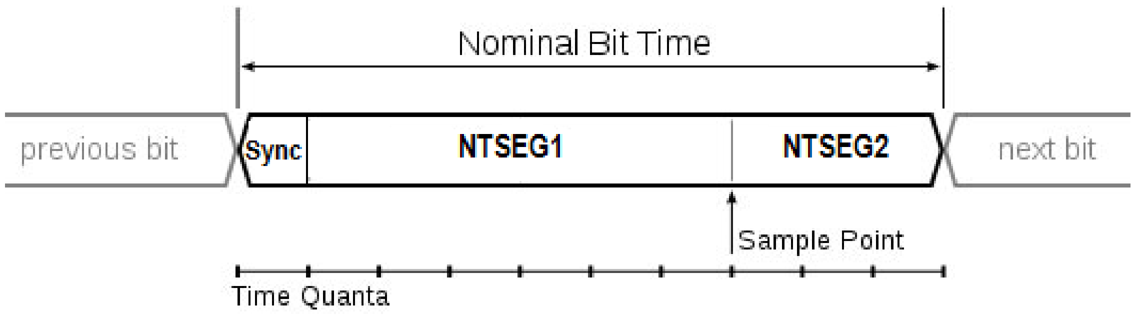


Figure 31. **Bit timing configuration fields**

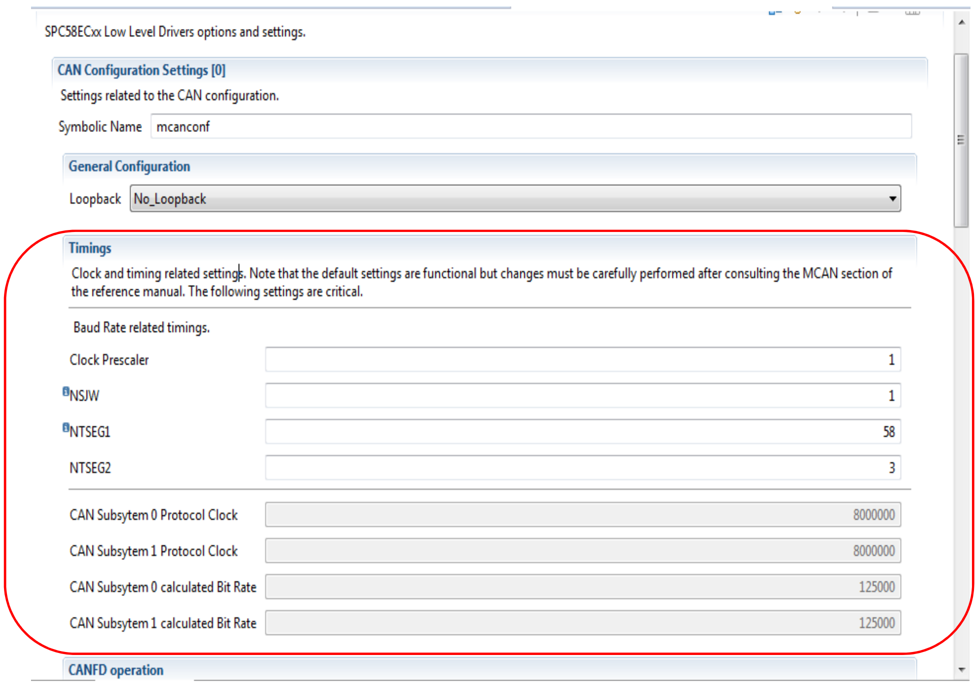
Note: *Sync=1 time quanta and NTSEG1 is the sum of Prop_Seg and Phase_Seg1.*

Prop_Seg compensates the delay in the propagation of the signal among two nodes.

Phase_Seg1 is the segment employee to compensate the error introduced in the bus on the sample point (refer to Table 5. RM0407 Reference manual).

SPC5-Studio will ask you to configure the following fields:

Figure 32. **SPC5-Studio: bit timing configuration tab**



SPC58ECxx Low Level Drivers options and settings.

CAN Configuration Settings [0]

Settings related to the CAN configuration.

Symbolic Name	mcanconf

General Configuration

Loopback	No_Loopback

Timings

Clock and timing related settings. Note that the default settings are functional but changes must be carefully performed after consulting the MCAN section of the reference manual. The following settings are critical.

Baud Rate related timings.

Clock Prescaler	1
NSJW	1
NTSEG1	58
NTSEG2	3
CAN Subsytem 0 Protocol Clock	8000000
CAN Subsytem 1 Protocol Clock	8000000
CAN Subsytem 0 calculated Bit Rate	125000
CAN Subsytem 1 calculated Bit Rate	125000

CANFD operation

NSJW: As a result of Resynchronization, the NTSEG1 may be lengthened or the NTSEG2 may be shortened. The amount of lengthening or shortening of these segments has an upper bound given by the NSJW.

NTSEG1-2: Both segments, NTSEG1 and NTSEG2, are used to define the sampling point of a bit. The length of NTSEG1 and NTSEG2, within certain limits set by the value of Resynchronization Jumper Width (NSJW), can respectively be lengthened or shortened to recover the lost synchronization. NTSEG1 and NTSEG2 can assume values between 1 and 8 time quanta (t_q).

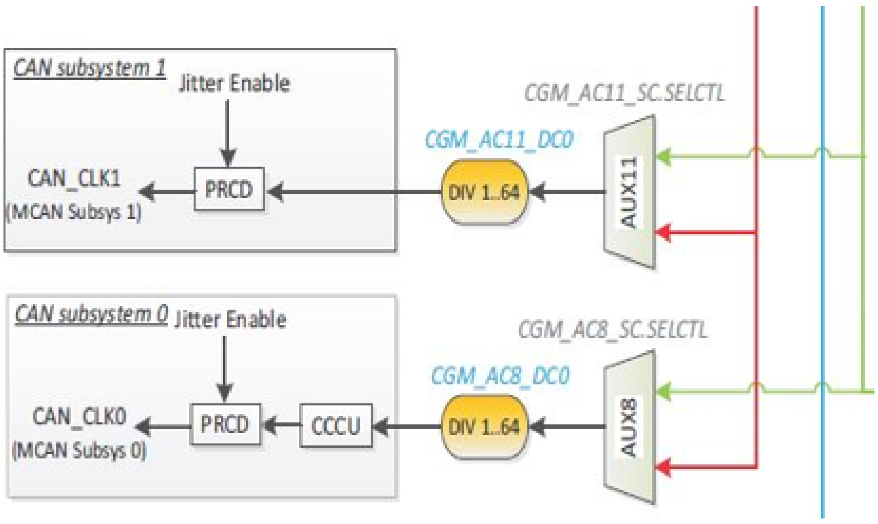
All concepts introduced in this paragraph are useful to understand how the timing parameters were set to calculate the bit rate. In our case bit rate = 125000, as shown in the Figure 33.

6.4.2 Divide clock system configuration

CAN protocol clock is the clock that the controller uses to define the time quanta (t_q) period and configure the bit time.

For a proper configuration of the CAN controller two clocks must be provided.

Figure 33. CAN protocol source clock



Protocol clock is provided by programming the CGM module as shown in the above Figure 33.

Another clock is the host clock that is the PFBridge_Clock. This is for the host controller.

The following formula must be respected for having a proper MCAN setup:

$$HOST_{CLK} \geq Protocol\ clock$$

Clock Jitter Enable is a software-accessible one-time writable register. This register stores the jitter enable status for the Pseudo Random Clock Divider Module (PRCD).

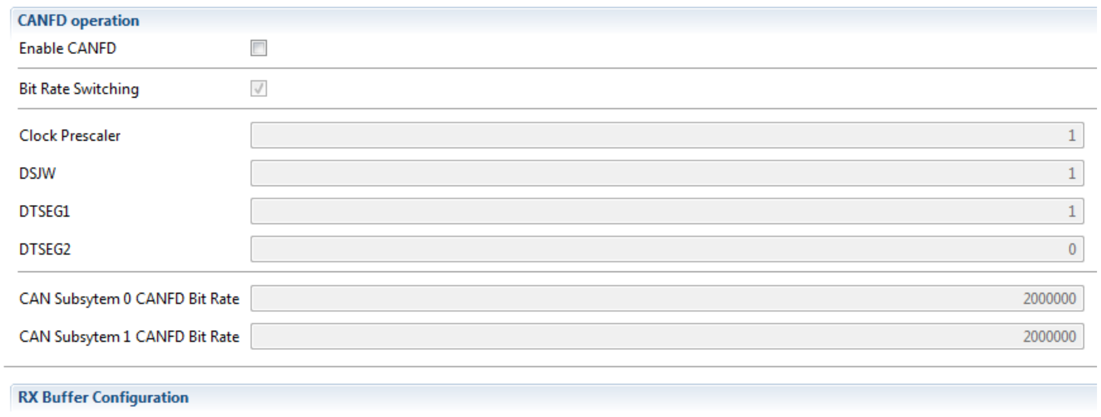
The clock jitter is activated in case of tampering. It represents in all respects a "counter-piracy" filter. If it is triggered, it will no longer be possible to unlock the microcontroller.

6.5 CAN-Flexible Data Rate overview

SPC5-Studio software allows the user to activate the CAN-FD configuration.

All the CAN controllers available in the SPC58EC-MCU are CAN-FD. In this application the extended messages are not used, but only Standard messages typical of the traditional CAN bus.

Figure 34. SPC5-Studio: CAN-FD bit timing configuration tab



CANFD operation

Enable CANFD	☐
Bit Rate Switching	✓
Clock Prescaler	1
DSJW	1
DTSEG1	1
DTSEG2	0
CAN Subsytem 0 CANFD Bit Rate	2000000
CAN Subsytem 1 CANFD Bit Rate	2000000

RX Buffer Configuration

CAN-FD (CAN with Flexible Data Rate) is an extension to the original CAN bus protocol specified.

CAN-FD was created to accommodate increases in bandwidth requirements within automotive networks.

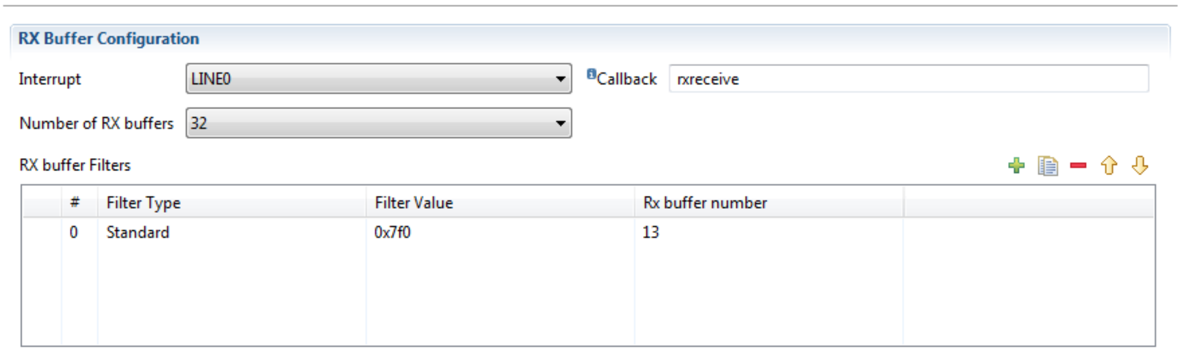
CAN-FD also allows for more storage capacity in the CAN Frame. While classic CAN has the capacity to hold 8 bytes of data within the CAN frame, CAN-FD can hold up to 64.

6.6 CAN receive buffer (RX buffer) configuration

The RX Buffer is a memory area used by received messages.

Figure 35 shows the configuration panel which allows the configuration of RX buffers and filters.

Figure 35. SPC5-Studio: CAN RX buffers and filters configuration tab



RX Buffer Configuration

Interrupt: LINE0
Callback: rxreceive
Number of RX buffers: 32
RX buffer Filters

#	Filter Type	Filter Value	Rx buffer number
0	Standard	0x7f0	13

The user can disable or enable the interrupt to which a specific callback function will be associated; the number of RX buffers can be defined between 0 and 64.

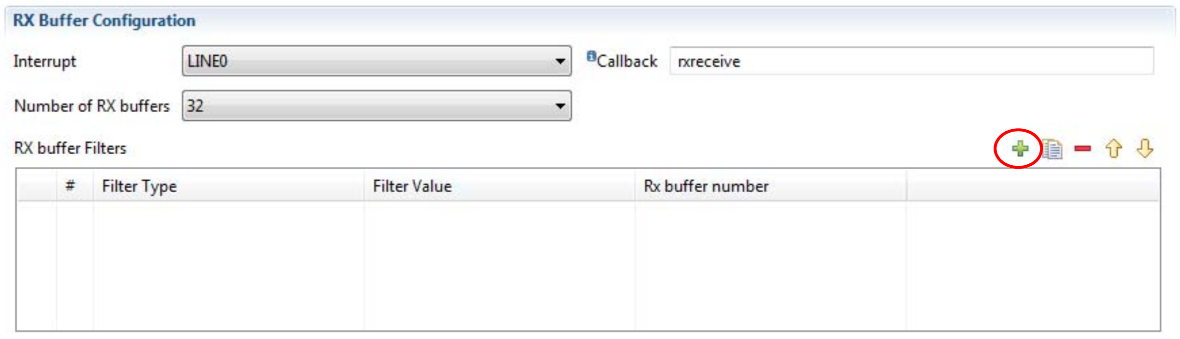
The callback function allows the developer to add a specific entry point for a specialized routine that will be called inside the ISR.

Interrupts are requests to MCU triggered by different configurable events. IRQs typically come from peripherals and ask the microcontroller to interrupt their work to respond to peripheral needs. In some cases the MCU can reject the request and refuse to stop working. If instead it accepts, it will take care of the needs of the device, then it will resume the execution of the program.

In the CAN protocol the messages will be received by all the nodes, but they are stored in memory only if they pass the CAN controller acceptance filters.

How to add and configure a filter element for a dedicated RX buffer is described below:

Figure 36. SPC5-Studio: adding CAN filters



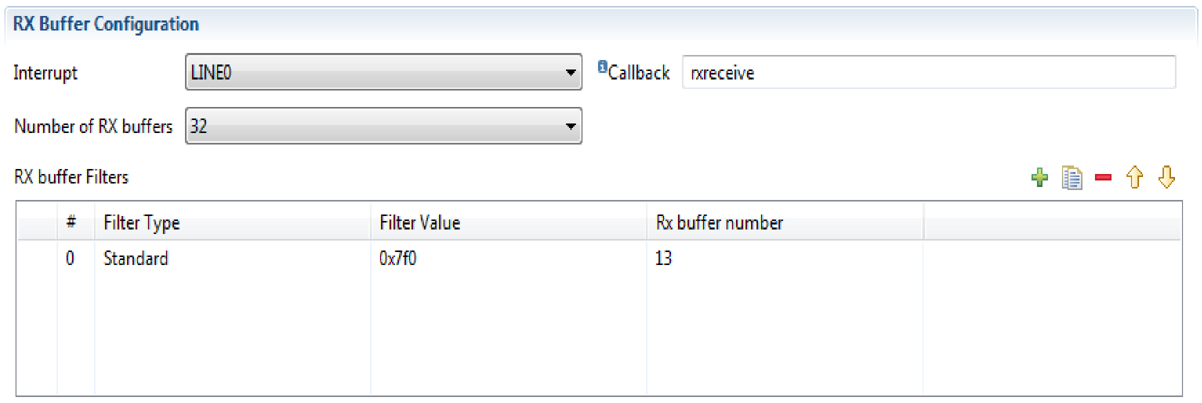
RX Buffer Configuration

Interrupt: LINE0
Callback: rxreceive
Number of RX buffers: 32
RX buffer Filters

#	Filter Type	Filter Value	Rx buffer number

Click the green icon " ✚ " to add a filter, which will have to be set:

Figure 37. SPC5-Studio: CAN filter example



RX Buffer Configuration

Interrupt: LINE0

Callback: rxreceive

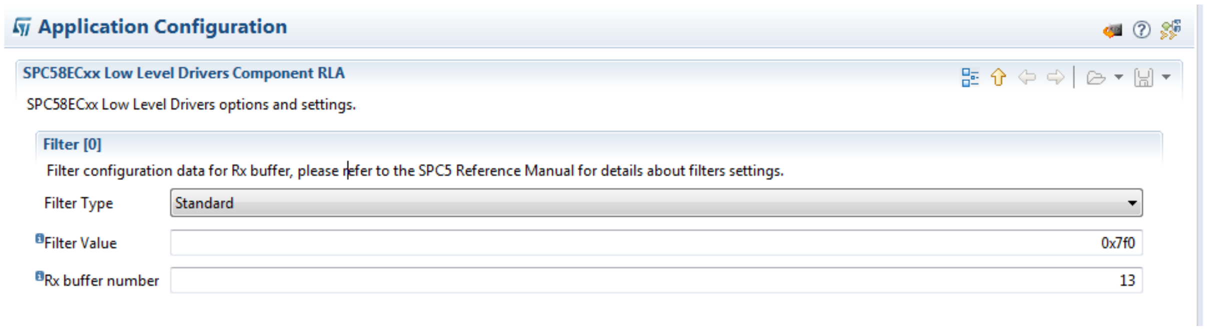
Number of RX buffers: 32

RX buffer Filters

#	Filter Type	Filter Value	Rx buffer number
0	Standard	0x7f0	13

To configure the filter, double click on the generated filter and choose the type of filter (Standard or Extended), the value and the RX buffer number. The latter must be less than or equal to the chosen number of dedicated RX buffers:

Figure 38. SPC5-Studio: CAN dedicated RX buffer filter configuration tab



Application Configuration

SPC58ECxx Low Level Drivers Component RLA

SPC58ECxx Low Level Drivers options and settings.

Filter [0]

Filter configuration data for Rx buffer, please refer to the SPC5 Reference Manual for details about filters settings.

Filter Type: Standard

Filter Value: 0x7f0

Rx buffer number: 13

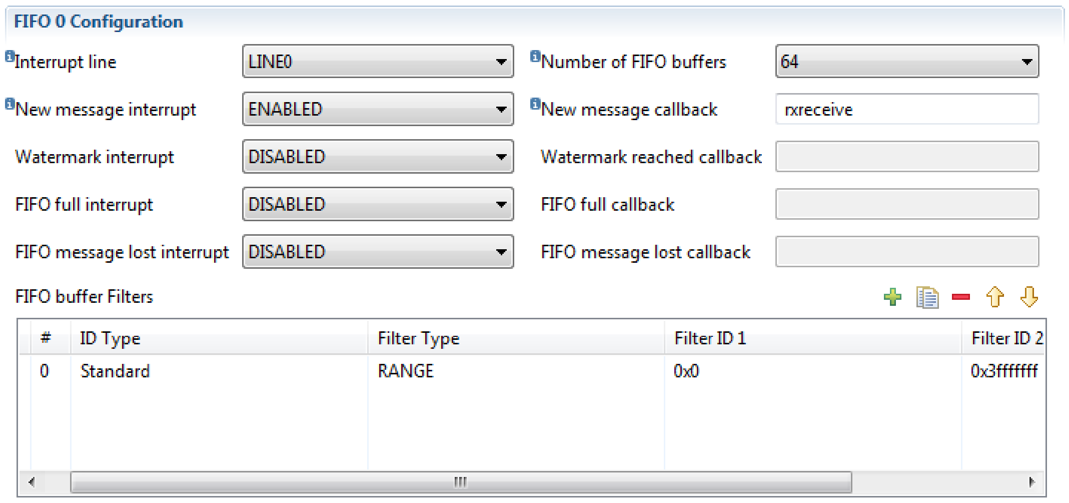
The filter value is set by the user and is used to filter the values that the RX buffer, relative to a given CAN instance, must receive.

6.7 FIFO configuration

The FIFO configuration is used when the volume of messages is relevant.

Three types of filters can be configured in MCAN controllers: range filters, dual ID filters and standard filters. Figure 39 shows a range filter where all messages having ID in the range [0, 0x3FFFFFFF] will be accepted by the configured controller:

Figure 39. SPC5-Studio: CAN FIFO and FIFO filters configuration tab



FIFO 0 Configuration

Interrupt line	LINE0	Number of FIFO buffers	64
New message interrupt	ENABLED	New message callback	rxreceive
Watermark interrupt	DISABLED	Watermark reached callback	
FIFO full interrupt	DISABLED	FIFO full callback	
FIFO message lost interrupt	DISABLED	FIFO message lost callback	

FIFO buffer Filters

#	ID Type	Filter Type	Filter ID 1	Filter ID 2
0	Standard	RANGE	0x0	0x3fffffff

6.8 TX configuration

There are several modes to manage TX buffers, such as::

- Dedicated;
- FIFO;
- QUEUE;
- MIXED FIFO or QUEUE.

The Dedicated TX Buffers are intended for message transmission. Each Dedicated TX Buffer is configured with a specific Message ID.

Behind the concept of FIFO Buffer: the data are managed and organized through the logic that the first one that comes in is the first one that comes out.

QUEUE is intended as a data structure in which messages are assigned with a priority or a different weight.

The Arbiter is able to select the message to be sent according to its priority, starting from the higher priority message to the lower one.

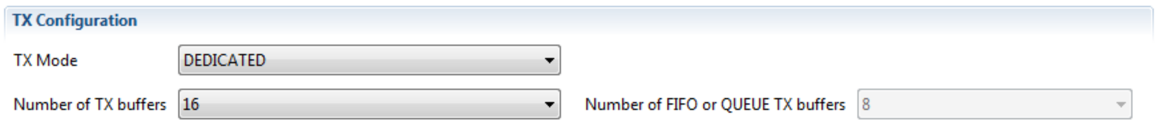
Consequently, the transmission does not rely on the message arrival order but on the priority assigned.

The MIXED mode is a combination of two modes. There are two types of MIXED modes:

- DEDICATED and FIFO;
- DEDICATED and QUEUE.

The two logics cannot coexist. The MIXED Mode is suitable for complex applications where the user needs to regulate the transmission of some messages via FIFO or QUEUE and others through DEDICATED buffer.

Figure 40. SPC5-Studio: CAN dedicated TX buffer configuration tab



TX Configuration

TX Mode	DEDICATED		
Number of TX buffers	16	Number of FIFO or QUEUE TX buffers	8

Note: For the DEDICATED Mode SPC5-Studio gives the possibility to set only the TX buffer number.

For QUEUE and FIFO Mode you can set the number of FIFO or QUEUE Buffers and instead for the MIXED Mode you can set both.

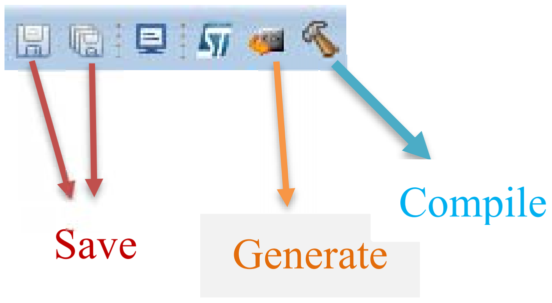
7 Generate application code and compile application

The next three phases are: save the project, generate the code and compile it.

Through code generation, all settings done by the user are generated in specific libraries. These libraries are related to the component drivers employed in the application, i.e. the CAN component in our case.

The image below shows how to execute the above three phases:

Figure 41. SPC5-Studio: generation and compilation buttons



8 Code overview

This chapter describes the code used in our application and illustrates some meaningful parts of the code.

The three key functions used in this application are:

- *can_lld_start*: Initialize the drivers of the subsystems;
- *can_lld_transmit*: Transmit a message;
- *can_lld_receive*: receive a message.

8.1 Initialize the drivers of the subsystems

The first step is to initialize the CAN bus device:

can_lld_start(&CAND2, &can_config_mcanconf);

the function *can_lld_start* has the task of initializing a specific CAN controller.

The configuration used for the selected CAN controller is specified in *can_config_mcanconf* created during the previous low-level driver setup (see Section 6.2 CAN configuration setting).

The configuration used is *mcanconf* has been setup in low-level driver. After the code has been generated, SPC5-Studio adds the prefix "*can_config_*" to the chosen symbolic name, e.g. *mcanconf*.

8.2 Prepare frame and transmit a message

After the driver CAN has been initialized, the transmitting structure message is defined with the type CANTxFrame.

CANTxFrame is the type assigned to the variable txf: *CANTxFrame txf.*

The CANTxFrame has the following structure:

```
typedef struct {
uint8_t OPERATION; /**< NORMAL or CANFD */
uint8_t TYPE; /**< Id type. STD or XTD, Standard or Extender */
uint32_t ID; /**< Standard identifier, message's priority.*/
uint8_t DLC; /**< Data Length Code. */
uint32_t data32[SPC5_CAN_MAX_DATA_LENGHT/4U]; /**<Frame data.*/
```

The message is prepared by compiling the fields of the CANTxFrame structure:

```
/* prepare frame to be transmitted on MCAN SUB 0 CAN 1*/
txf.TYPE = CAN_ID_STD;
txf.ID = 0x7f0U;
txf.DLC = 8U;
txf.data32[0] = 0xDDEEFFAAUL;
```

The fields of the above mentioned structure are:

- **TYPE**: indicates the type of message. There are two standard options, CAN_ID_STD or extended (CAN_ID_XTD);
- **ID**: is chosen by the user and indicates the priority of the message;
- **DLC**: is chosen by the user and indicates the Data Length Code;
- **Data32**: is chosen by the user and indicates the Frame data.

After preparing the message, use the following steps to send it.

To send the message the second key function is used:

can_lld_transmit(&CAND2, CAN_DEDICATED_TXBUFFER, &txf);

This function (*can_lld_trasmit*) handles the transmission of messages and needs three types of information:

- CAN Controller;
- type of TX Buffer;
- CANTxFrame.

The three variables used are explained in detail by *can_lld_trasmit*:

- **&CAN-Controller**: it identifies the controller used to transmit
- **TX MODE**: the concept has been already explained in Section 6.8 TX configuration. The TX variable depends on the selected configuration of the TX Buffer. This variable is automatically generated by SPC5-Studio. All the possible values will be listed in the following table:

Table 2. **Values TX_Mode**

TX_Mode	Name_SPC5-Studio
NOT DEFINITED	CAN_ANY_TXBUFFER
DEDICATED	CAN_DEDICATED_TXBUFFER
FIFO	CAN_FIFO_TXBUFFER
QUEUE	CAN_QUEUE_TXBUFFER
MIXED FIFO	CAN_MIXED_FIFO_TXBUFFER
MIXED QUEUE	CAN_MIXED_QUEUE_TXBUFFER

- **&message_Trasmitt (txf)**: it represents the actual message to be transmitted.

The message will be read by all the nodes, even from the node that sent it.

8.3 Receive message

The last key function is *can_lld_receive*:

can_lld_receive(&CAND2, CAN_DEDICATED_RXBUFFER, &rxf);

In our example, this function is used by SPC58ECx to receive the messages from Black Box node.

When a message reaches SPC58EC-DISP, the MCU will receive an interrupt. If interrupts are not masked, the MCU will finish executing the current instruction and will jump to the initial instruction of the Callback function.

The CANRxFrame type is assigned to *rxf* variable: *CANTxFrame txf*.

The function used to receive is similar to the transmission one, it uses the following variables:

- **&CAN-Controller**: indicates on which controller to receive;
- **RX_Buffer**: the concept has been already explained in Section 6.6 CAN receive buffer (RX buffer) configuration, the RX variable depends on the selected configuration of the RX Buffer. This variable is automatically generated by SPC5-Studio. All the possible values of this type are listed in the following table:

Table 3. **Values RX_Mode**

RX_Buffer	Name_SPC5-Studio
NOT DEFINITED	CAN_ANY_RXBUFFER
DEDICATED	CAN_DEDICATED_RXBUFFER
FIFO	CAN_FIFO_RXBUFFER

- **&message_receive (rxf)**: represents the location where to store the received message.

Now the structure of the message received, CANRxFrame, will be described:

- **TIME**: indicates the timestamp. This is a number that expresses a temporal magnitude; more precisely, it corresponds to the number of bit time that has elapsed since the moment the message was sent;
- **OPERATION**: indicates if you are using CAN Normal (CAN_OP_NORMAL) or CANFD (CAN_OP_CANFD);
- **TYPE**: indicates the type of message. There are two types: CAN_ID_STD or CAN_ID_XTD;
- **ID**: is the priority of message;
- **DLC**: indicates the Data Length Code;
- **Data32**: is the data Frame.

8.4 Message status

The protocol does not include many message status alerts, because, as already said, the management of most of the errors is handled by the CRC.

The message can be found in one of the following two status:

CAN_MSG_WAIT = the message is waiting for the free channel

CAN_MSG_OK = the message is transmitted/received

The only error that is visible during debugging is the BUS_ERROR. The Bus Error occurs only in the presence of hardware component failures or timing problems.

8.5 Interrupt enable

Another important concept is the activation of interrupts. This is valid also for the interrupts used by the CAN protocol. The function used is the following:

irqIsrEnable()

To synchronize the transmission time with the interrupt it is often necessary to introduce delays, these are managed in SPC5-Studio through the following function. The number of milliseconds [ms] of delay is inserted within the brackets.

osalThreadDelayMilliseconds(1);

8.6 Main code

Now the user has the tools to understand the syntax of the code used in this demo.

Please find below the example code:

```
/* Inclusion of the main header files of all the imported components in the
order specified in the application wizard. The file is generated
automatically.*/
#include "components.h"
#include "can_lld_cfg.h"
```

This header files, *"components.h"* and *"can_lld_cfg.h"*, include the code generated automatically. These libraries contain all the files resulting from the configuration in PinMap Editor and in low-level drivers.

```
static uint32_t counter;
/* using receive mode with callback */
void mcanconf_rxreceive(uint32_t msgbuf, CANRxFrame crfp) {
if (crfp.ID == 0x7f0U) {
if (crfp.data32[1] == counter) {
pal_lld_togglepad(PORT_F, PF_LED1);
}
}
(void)msgbuf;
}
```

Here the code recalls the callback configuration. Its structure is within the library *"can_lld_cfg.h"*:

```
/*
* Application entry point.
*/
int main(void) {
CANTxFrame txf;
uint32_t returnvalue;
/* Initialization of all the imported components in the order specified in
the application wizard. The function is generated automatically.*/
componentsInit();
/* Enable Interrupts */
irqIsrEnable();
can_lld_start(&CAND2, &can_config_mcanconf);/*MCAN SUB 0 CAN 1*/
```

Here this part of code contains the first key function:

```
/* prepare frame to be transmitted on MCAN SUB 0 CAN 1*/
txf.TYPE = CAN_ID_STD;
txf.ID = 0x7f0U;
txf.DLC = 8U;
txf.data32[0] = 0xDDEEFFAAUL;
counter = 0UL;
```

In this portion of code the frame will be prepared to be transmitted and initialize the variable counter used as message body. The structure of CANTxFrame contained in header file *"can_lld_cfg.h"*.

```
/* Application main loop.*/
for (;;) {
txf.data32[1] = counter;
returnvalue = can_lld_transmit(&CAND2, CAN_DEDICATED_TXBUFFER, &txf);
osalThreadDelayMilliseconds(1);
if (returnvalue != CAN_MSG_OK) {
for (;;) {
}
}
```

Here the message is sent and this part of code contains the second key function: *can_lld_transmit*.

If the transmission of message fails, the code enters an endless loop, this it is a feature of used test. The structure of CANRxFrame is contained in header file *"can_lld_cfg.h"*.

```
CANRxFrame rxf;
returnvalue = can_lld_receive(&CAND2, CAN_ANY_RXBUFFER, &rxf);
if (returnvalue == CAN_MSG_OK) {
if (rxf.ID == 0x7f0U) {
if (rxf.data32[1] == counter) {
pal_lld_togglepad(PORT_F, PF_LED2);
}
}
}
```

Here the message is received and this part of code contains the last key function: *can_lld_receive*.

If the transmission of message doesn't fail, the ID frame is correct one and there is not loss of packets then the LED 2, in the MCU discovery board, will blink.

```
osalThreadDelayMilliseconds(250);*/ delay 250ms*/
counter++; /* increase the counter */
}
}
```

9 “No Loopback” example

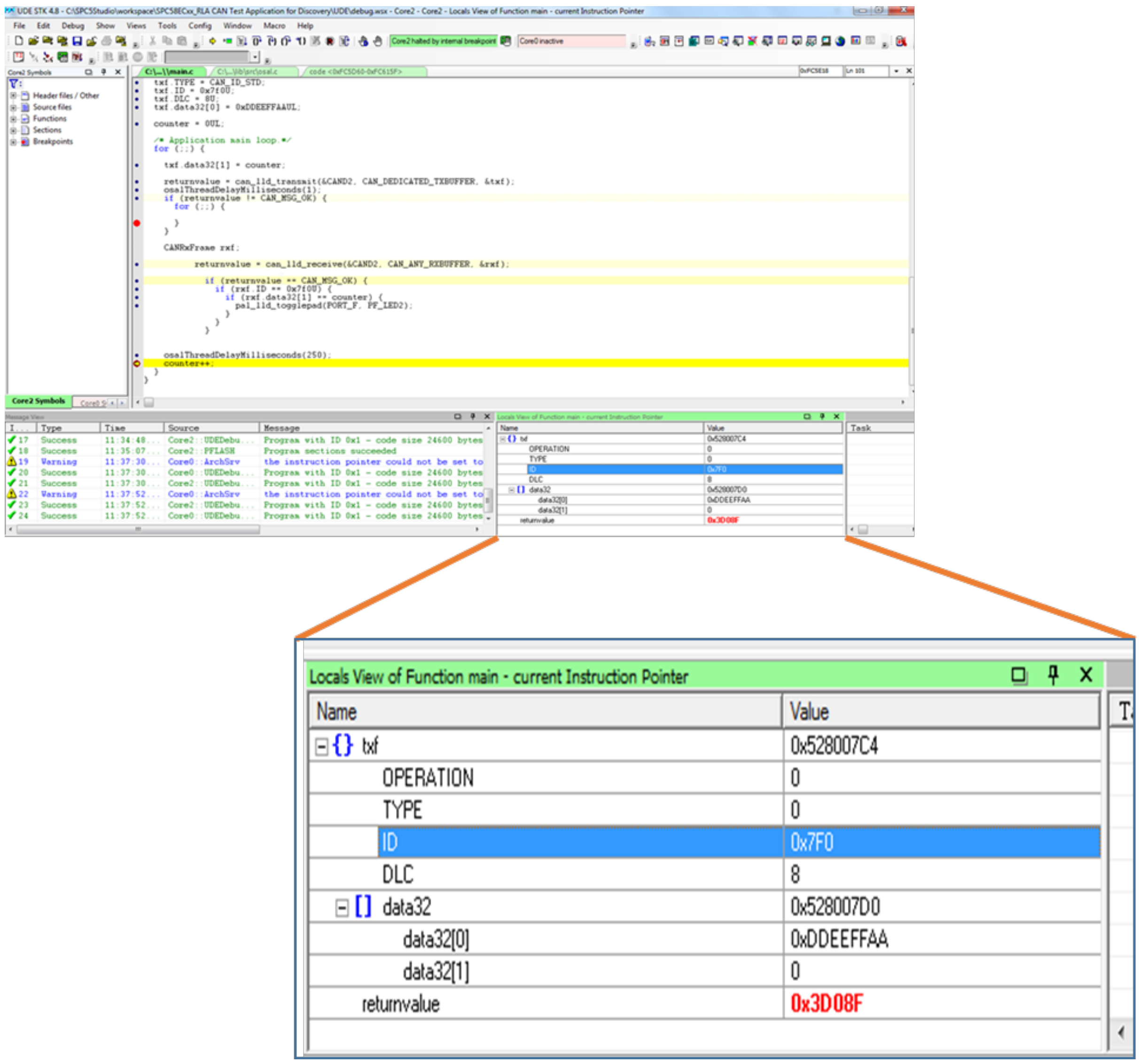
This chapter will show how to check the correct transmission and reception of messages. This test will be performed in No Loopback mode.

The first step is to verify the correct transmission of the message through the use of UDE STK 4.8.

UDE STK 4.8 is a Debugging Software for Windows developed by PLS Development Tools.

Figure 42 shows the results obtained in the respective break points:

Figure 42. SPC5-Studio: variable view during debug


UDE STK 4.8 - C:\SPC5Studio\workspace\SPC58ECxx_RLA CAN Test Application for Discovery\UDE\debug.wsx - Core2 - Core2 - Locals View of Function main - current Instruction Pointer
File
Edit
Debug
Show
Views
Tools
Config
Window
Macro
Help
Core2 halted by internal breakpoint
Core0 inactive
Core2 Symbols
Header files / Other
Source files
Functions
Sections
Breakpoints
Core2 Symbols
C:\...\\main.c
C:\...\lib\src\osal.c
code <0xFC5D60-0xFC615F>
Message View
Locals View of Function main - current Instruction Pointer
Task

```
txf.TYPE = CAN_ID_STD;
txf.ID = 0x7f0U;
txf.DLC = 8U;
txf.data32[0] = 0xDDEEFFAAUL;

counter = 0UL;

/* Application main loop.*/
for (;;) {

  txf.data32[1] = counter;

  returnvalue = can_lld_transmit(&CAND2, CAN_DEDICATED_TXBUFFER, &txf);
  osalThreadDelayMilliseconds(1);
  if (returnvalue != CAN_MSG_OK) {
    for (;;) {

    }
  }

  CANRxFrame rxf;

        returnvalue = can_lld_receive(&CAND2, CAN_ANY_RXBUFFER, &rxf);

          if (returnvalue == CAN_MSG_OK) {
            if (rxf.ID == 0x7f0U) {
              if (rxf.data32[1] == counter) {
                pal_lld_togglepad(PORT_F, PF_LED2);
              }
            }
          }


  osalThreadDelayMilliseconds(250);
  counter++;
 }
}
```

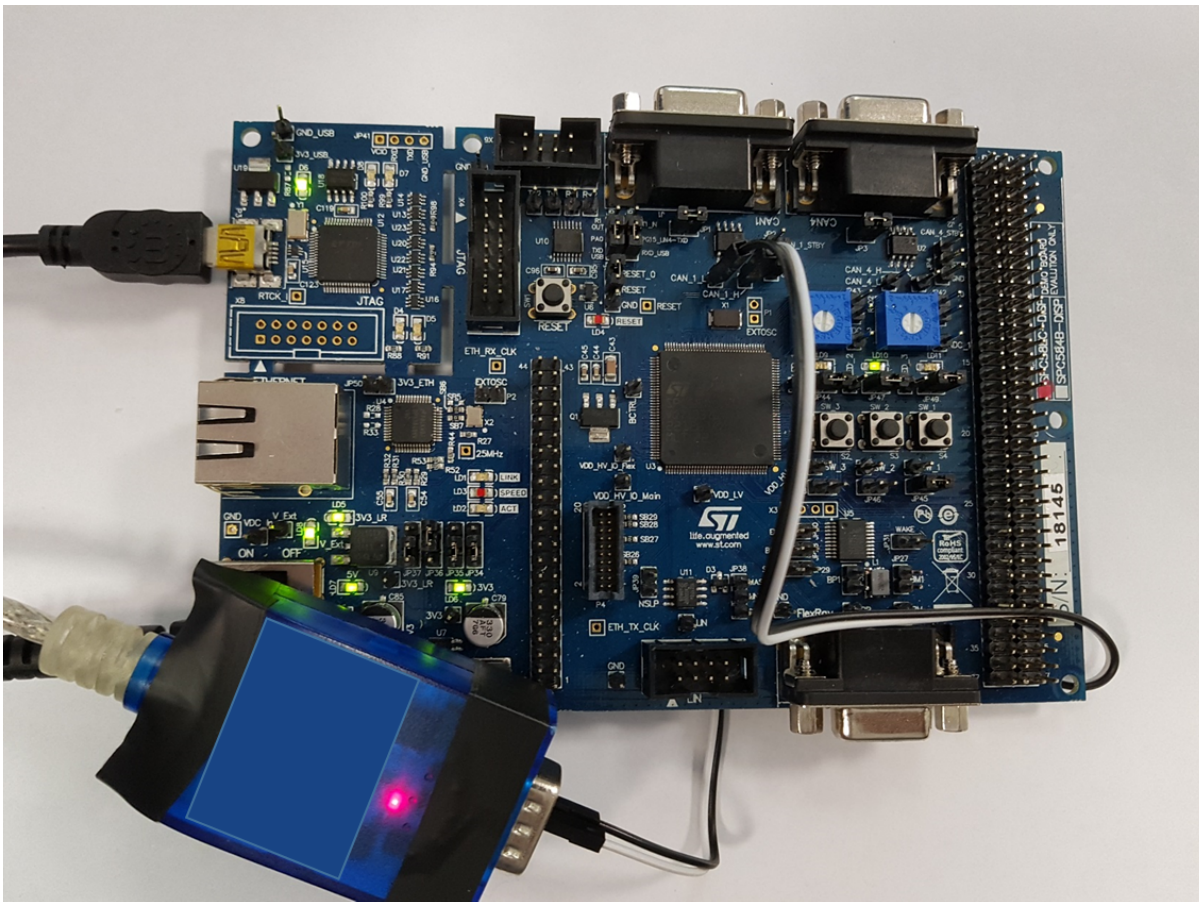
I...	Type	Time	Source	Message
✓ 17	Success	11:34:48...	Core2::UDEDebu...	Program with ID 0x1 - code size 24600 bytes
✓ 18	Success	11:35:07...	Core2::PFLASH	Program sections succeeded
⚠ 19	Warning	11:37:30...	Core0::ArchSrv	the instruction pointer could not be set to
✓ 20	Success	11:37:30...	Core0::UDEDebu...	Program with ID 0x1 - code size 24600 bytes
✓ 21	Success	11:37:30...	Core2::UDEDebu...	Program with ID 0x1 - code size 24600 bytes
⚠ 22	Warning	11:37:52...	Core0::ArchSrv	the instruction pointer could not be set to
✓ 23	Success	11:37:52...	Core2::UDEDebu...	Program with ID 0x1 - code size 24600 bytes
✓ 24	Success	11:37:52...	Core0::UDEDebu...	Program with ID 0x1 - code size 24600 bytes

Locals View of Function main - current Instruction Pointer

Name	Value
⊟ {} txf	0x528007C4
OPERATION	0
TYPE	0
ID	0x7F0
DLC	8
⊟ [] data32	0x528007D0
data32[0]	0xDDEEFFAA
data32[1]	0
returnvalue	0x3D08F

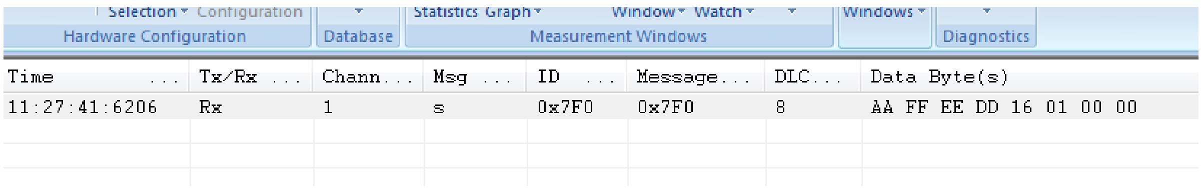
The reception phase will be verified using a "CAN bus sniffer".

Figure 43. **CAN connection**



The CAN bus sniffer GUI allows the visualization of received messages.
The message received in this example is shown below:

Figure 44. **CAN bus sniffer GUI: RX message view**



Time ...	Tx/Rx ...	Chann...	Msg ...	ID ...	Message...	DLC...	Data Byte(s)
11:27:41:6206	Rx	1	s	0x7F0	0x7F0	8	AA FF EE DD 16 01 00 00

The last step is to verify that what has been transmitted has been received.

Attention: *Without any ID filters applied no message will be received.*

Appendix A Acronyms, abbreviations and reference documents

Table 4. **Acronyms and abbreviations**

Terms	Meaning
CAN-FD	CAN-Flexible Data Rate
CAN-H	CAN-High
CAN-L	CAN-Low
M_CAN	CAN Module
NSJW	Resynchronization Jumper Width Sync
t_q	Time quanta
RLA	Register Level Access

Table 5. **Reference documents**

Document name	Document type
RM0407	Reference manual

Appendix B MCAN naming conventions

While using or porting own code for the CAN peripherals embedded in SPC5x micro-controllers, the user can find different names adopted for the instances available inside the subsystems.

Although the Reference Manual clearly reports the number of instances for each subsystem, and tools like SPC5-Studio actually helps on configuring in an easy way all of them (through wizard etc.), this section aims to help on summarizing and finding easily a match among those conventions.

The name of the CAN instances, in each subsystem documented in the reference manual and the I/O pin table document, are summarized in the table below.

Table 6. **MCAN naming conventions**

SPC58x MCU	CAN Subsystem	CAN instance in a subsystem	I/O pin table
SPC582Bx	0	CAN_SUB_0_M_CAN_0	CAN 0 sys0
		CAN_SUB_0_M_CAN_1	CAN 1 sys0
		CAN_SUB_0_M_CAN_2	CAN 2 sys0
		CAN_SUB_0_M_CAN_3	CAN 3 sys0
	1	CAN_SUB_1_M_CAN_1	CAN 4 sys1
		CAN_SUB_1_M_CAN_2	CAN 5 sys1
		CAN_SUB_1_M_CAN_3	CAN 6 sys1
SPC584Bx SPC584Cx/SPC58ECx	0	CAN_SUB_0_M_CAN_0	CAN 0 sys0
		CAN_SUB_0_M_CAN_1	CAN 1 sys0
		CAN_SUB_0_M_CAN_2	CAN 2 sys0
		CAN_SUB_0_M_CAN_3	CAN 3 sys0
	1	CAN_SUB_1_M_CAN_1	CAN 4 sys1
		CAN_SUB_1_M_CAN_2	CAN 5 sys1
		CAN_SUB_1_M_CAN_3	CAN 6 sys1
		CAN_SUB_1_M_CAN_4	CAN 7 sys1
SPC58xEx/SPC58xGx	0	CAN_SUB_0_M_TTCAN_0[1]	CAN 0 TT sys0
		CAN_SUB_0_M_CAN_1	CAN 1 sys 0
		CAN_SUB_0_M_CAN_2	CAN 2 sys 0
		CAN 1+2 sys0 Combined CAN 1 and CAN 2[2]	CAN 1+2 sys0
		CAN_SUB_0_M_CAN_3	CAN 3 sys 0
	1	CAN_SUB_1_M_CAN_1	CAN 4 sys 1
		CAN_SUB_1_M_CAN_2	CAN 5 sys 1
		CAN_SUB_1_M_CAN_3	CAN 6 sys 1
		CAN_SUB_1_M_CAN_4	CAN 7 sys 1
SPC58EHx/SPC58NHx[3]	0	CAN_SUB_0_M_CAN_0	CAN 0 sys0
		CAN_SUB_0_M_CAN_1	CAN 1 sys0
		CAN_SUB_0_M_CAN_2	CAN 2 sys0
		CAN_SUB_0_M_CAN_3	CAN 3 sys0
	1	CAN_SUB_1_M_CAN_1	CAN 4 sys1
		CAN_SUB_1_M_CAN_2	CAN 5 sys1

SPC58x MCU	CAN Subsystem	CAN instance in a subsystem	I/O pin table
SPC58EHx/SPC58NHx(3)	1	CAN_SUB_1_M_CAN_3	CAN 6 sys1
		CAN_SUB_1_M_CAN_4	CAN 7 sys1
	2	CAN_SUB_2_M_CAN_1	CAN 8 sys2
		CAN_SUB_2_M_CAN_2	CAN 9 sys2
		CAN_SUB_2_M_CAN_3	CAN 10 sys2
		CAN_SUB_2_M_CAN_4	CAN 11 sys2
	3	CAN_SUB_3_M_CAN_1	CAN 12 sys3
		CAN_SUB_3_M_CAN_2	CAN 13 sys3
		CAN_SUB_3_M_CAN_3	CAN 14 sys3
		CAN_SUB_3_M_CAN_4	CAN 15 sys3

1. *Time Triggered Modular CAN (M_TTCAN) core.*
2. *M_CAN1 and M_MCAN2 IO sharing implementation.*
3. *Referred to cut 2.*

The user could also meet, while porting or debugging software code, some other different names, for example: **CANSUB_<x>_MCAN_<y>**. The above table can help on matching the instance inside the related subsystem.

In SPC5-Studio please refer to Table 1. Subsystems that describes the MCAN software instances inside each subsystem.

Revision history

Table 7. **Document revision history**

Date	Revision	Changes
16-Jan-2020	1	First release.

Contents

List of tables

List of figures