SPC58EC CAN bus Configuration

Application note

Application Note for the basic configuration on SPC58EC platform.

INDEX

1 Introduction

This manual is intended for developers of system software wishing to develop an application with CAN communication.

The aim of this application example is to transmit messages through a CAN controller, from a Node 1, implemented with ST's SPC58ECx (Chorus 4M automotive microcontroller), 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.

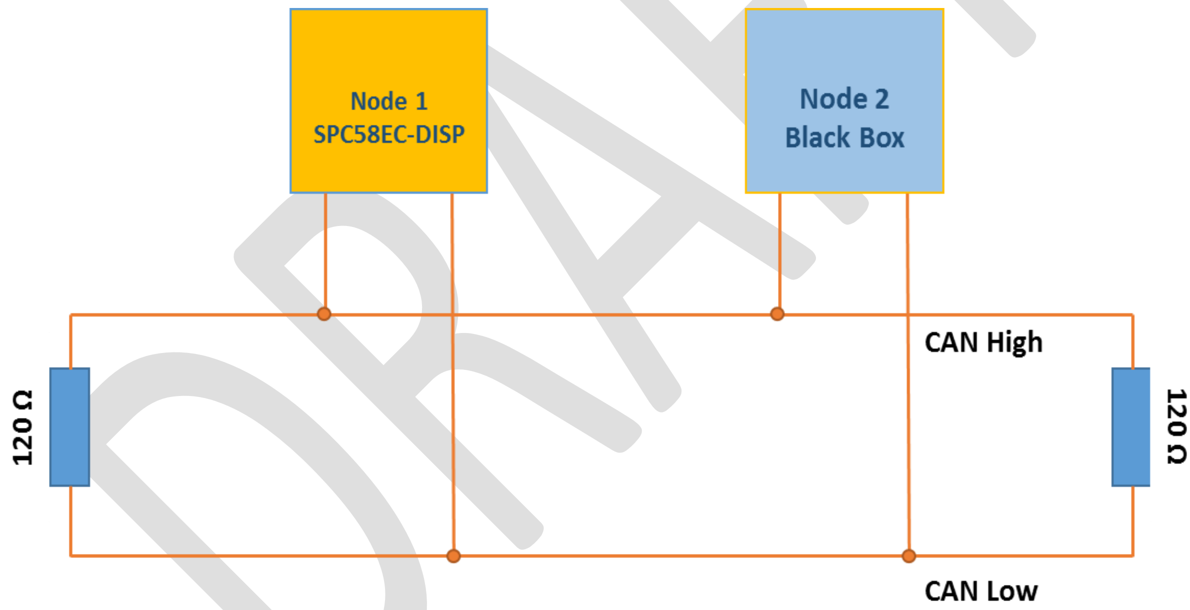


Figure 1: Basic application principle

2 Hardware Overview

To implement an example application, it has been used:

- A SPC58EC-DISP discovery board hosting a 4MB flash memory 32-bit SPC58 automotive microcontroller (Chorus 4M). SPC58EC-DISP represents the NODE 1 of our example.
- One 1-Amp 12-Volt 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.

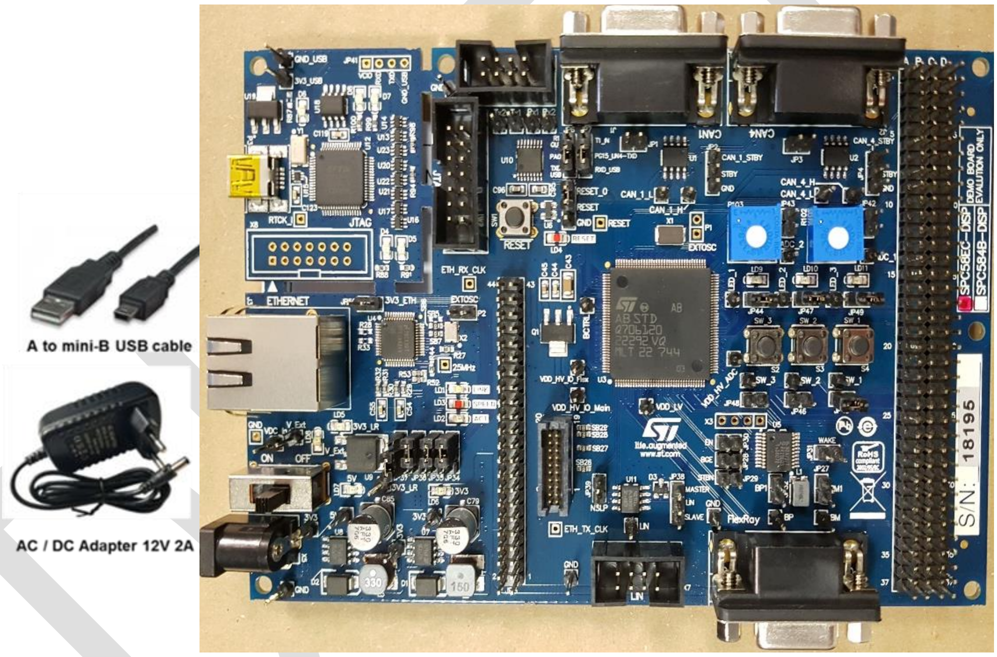


Figure 2: SPC58EC-DISP

3 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 into 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 the RAM access by the various CAN controllers.
- **SRAM:** The CAN subsystem will interface with an external RAM using this interface, it is the SRAM.
- **ECC Controller**: it contains the logic to compute and validate the correction code on the SRAM memory cells.

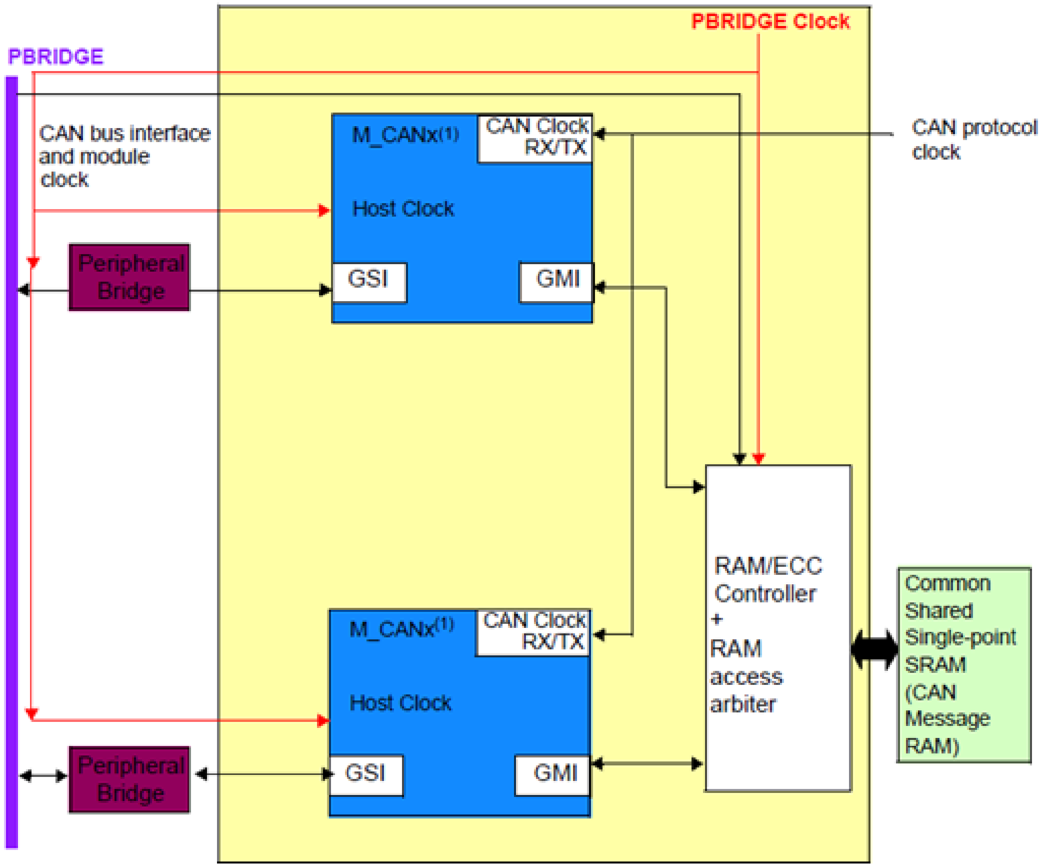


Figure 3: CAN subsystem generic block diagram

Each subsystem requires a specific configuration valid for all CAN controller embedded into this subsystem. As shown in the table 0 below, the first four CAN controllers belong to subsystem0 and the remaining controllers belong to subsystem1.

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

TABLE-1 subsystems

For own application example, 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 Chorus 4M documentation.

Finally 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 chosen 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 as well as providing receive message status information.

The Tx Handler is responsible for the transfer of transmit messages from the Message RAM to the CAN core as well as providing transmit status information.

Because the Message RAM is equipped with 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).

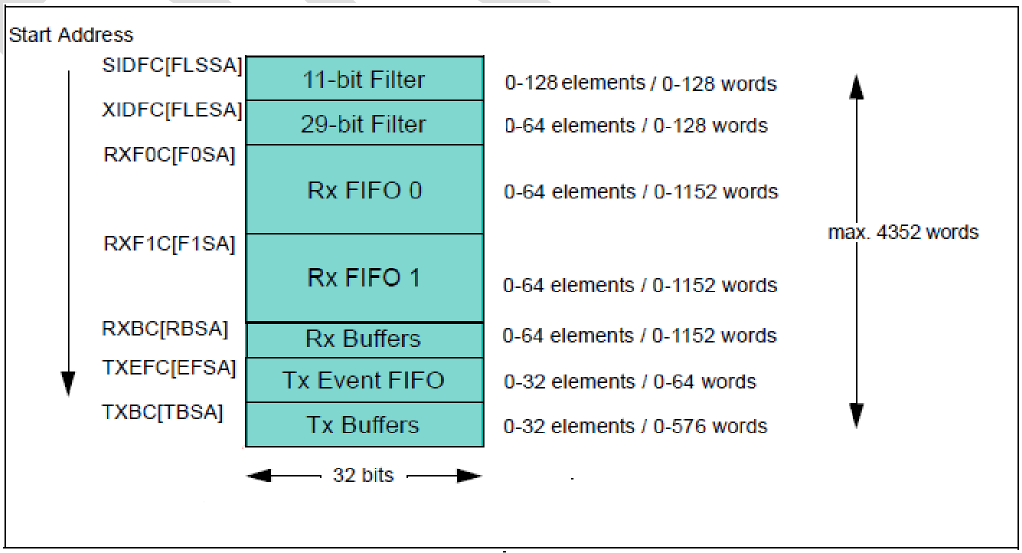
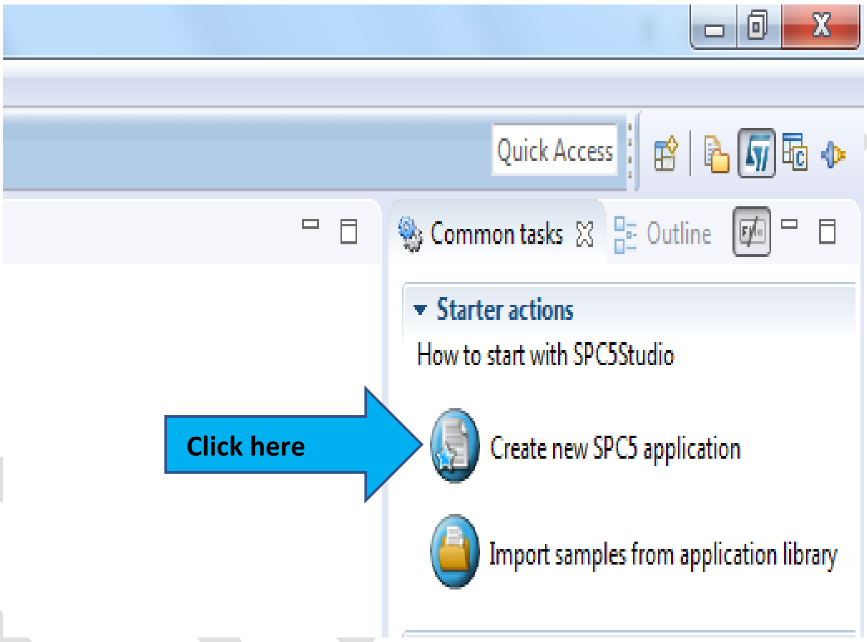


Figure 4: Message RAM Configuration

4 Create a NEW-PROJECT with SPC5-STUDIO, using the application wizard

In the previous chapters, the CAN protocol has been described briefly, the CAN controllers and the purpose of the application example.
The description of the methods and steps to follow to create a new project with SPC5Studio are described in this chapter.SPC5Studio is based on Eclipse Engine. SPC5-Studio allows user to entire 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:



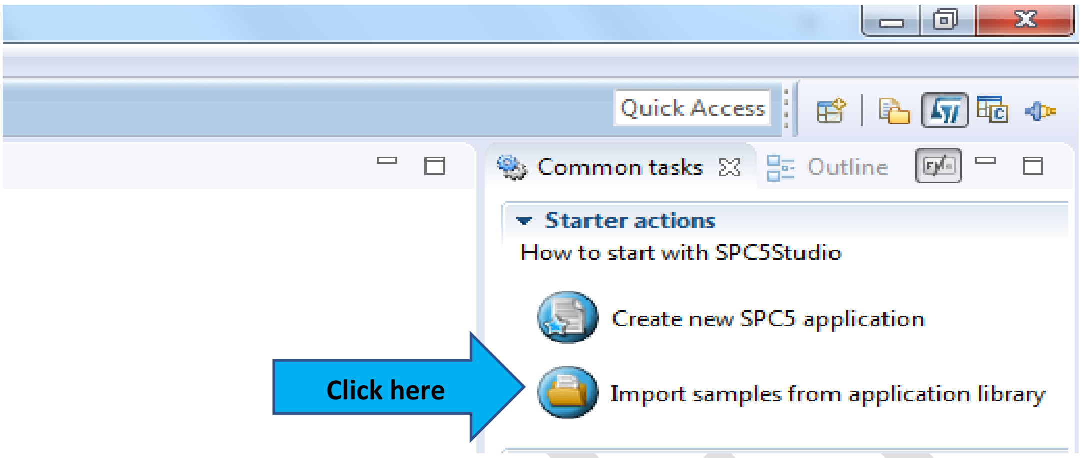
This example relies on the CAN driver demo available for the SPC58EC microntroller. The demo will import and it use as starting point for this application. Then, the demo will be fine tuned

by changing the fields and source code to adapt it to this application example.

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

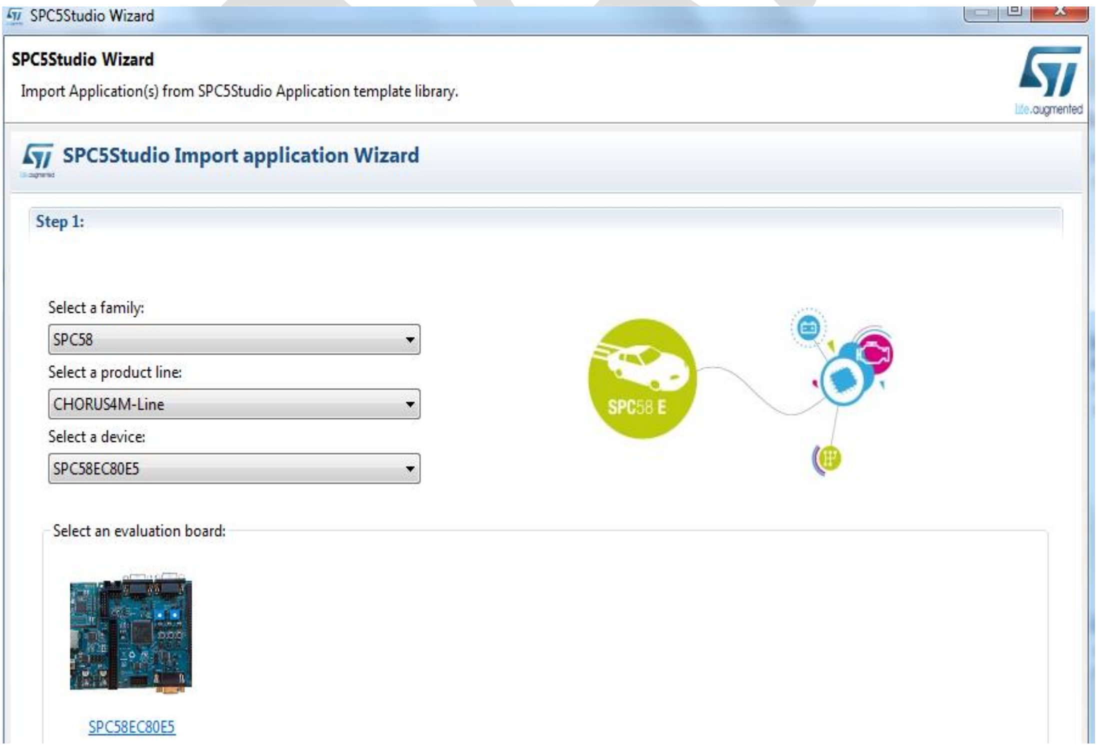
Step1: import a sample Application

User must click on Import Samples form application library in order to import a sample application, click the highlighted icon and go to step2.



Step2: select the Platform through the Application Wizard

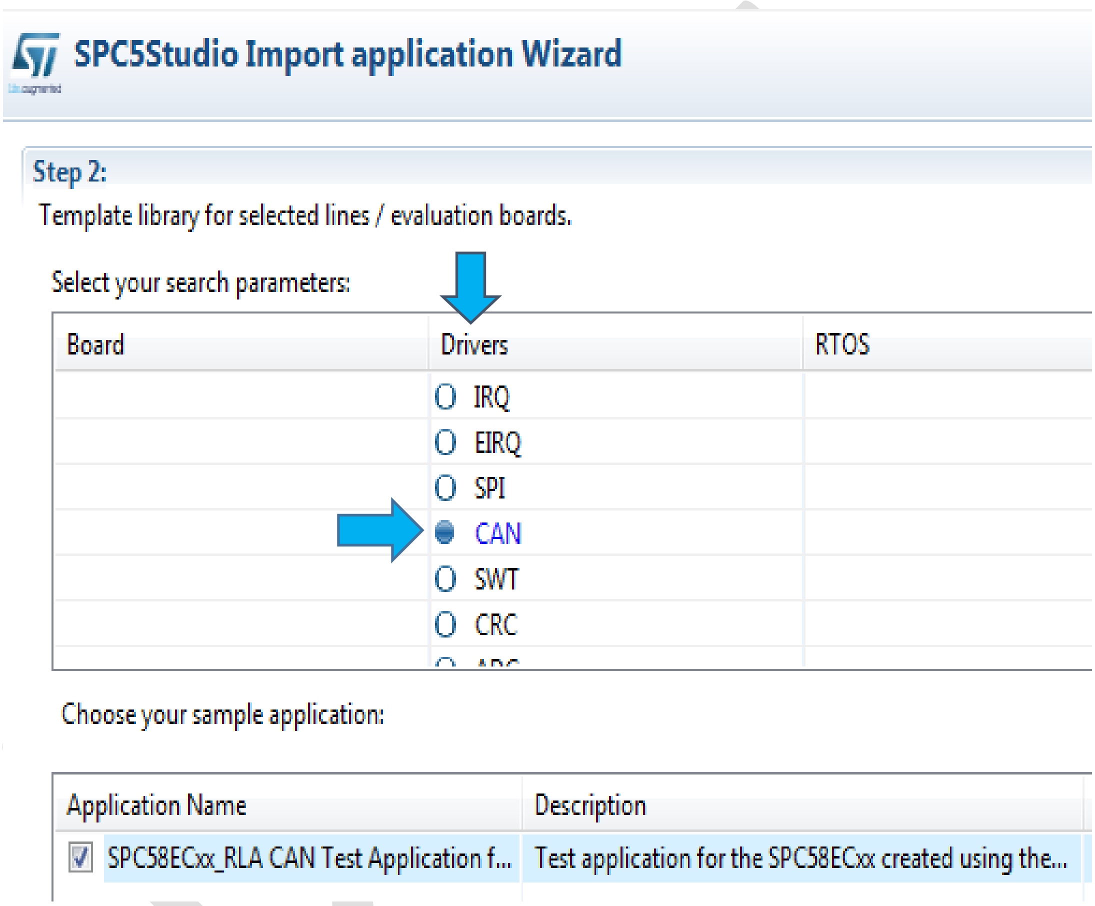
Select a family, a product line and a device as shown.



For our application, the family is SPC58, the product line is Chorus 4M and the device is SPC58C80ES

Step3: select the driver

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



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

5 PIN Configuration

This paragraph will explain how to select and configure the microcontroller pins needed for our application example.

Some I/O PINs of Chorus 4M can be identified as Ports. The PINs are the physical connection, one and an only 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 inside SPC5Studio: this is an intuitive graphical interface dedicated to microcontroller pin configuration.

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

A brief summary of the pin configuration procedure follows:

1. Select the Driver, e.g. CAN1
2. Select the signal by clicking on the driver. You will be presented with the list of signals to be configured, 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.

Automatic generation of code based on the pin configuration is one of the major advantages of using SPC5-Studio.

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

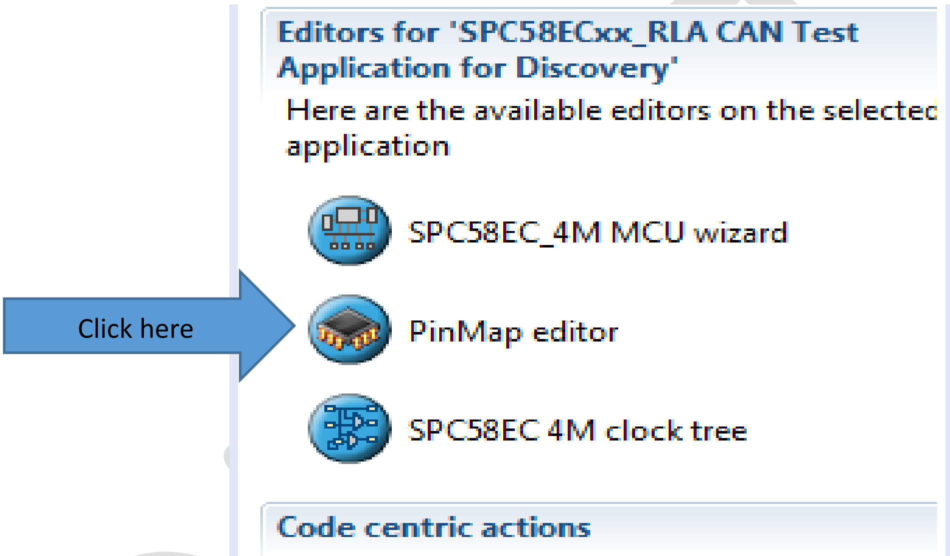
1. Pin selection from schematic;
2. Pin selection from function.

Case 1. Pin selection from schematic:

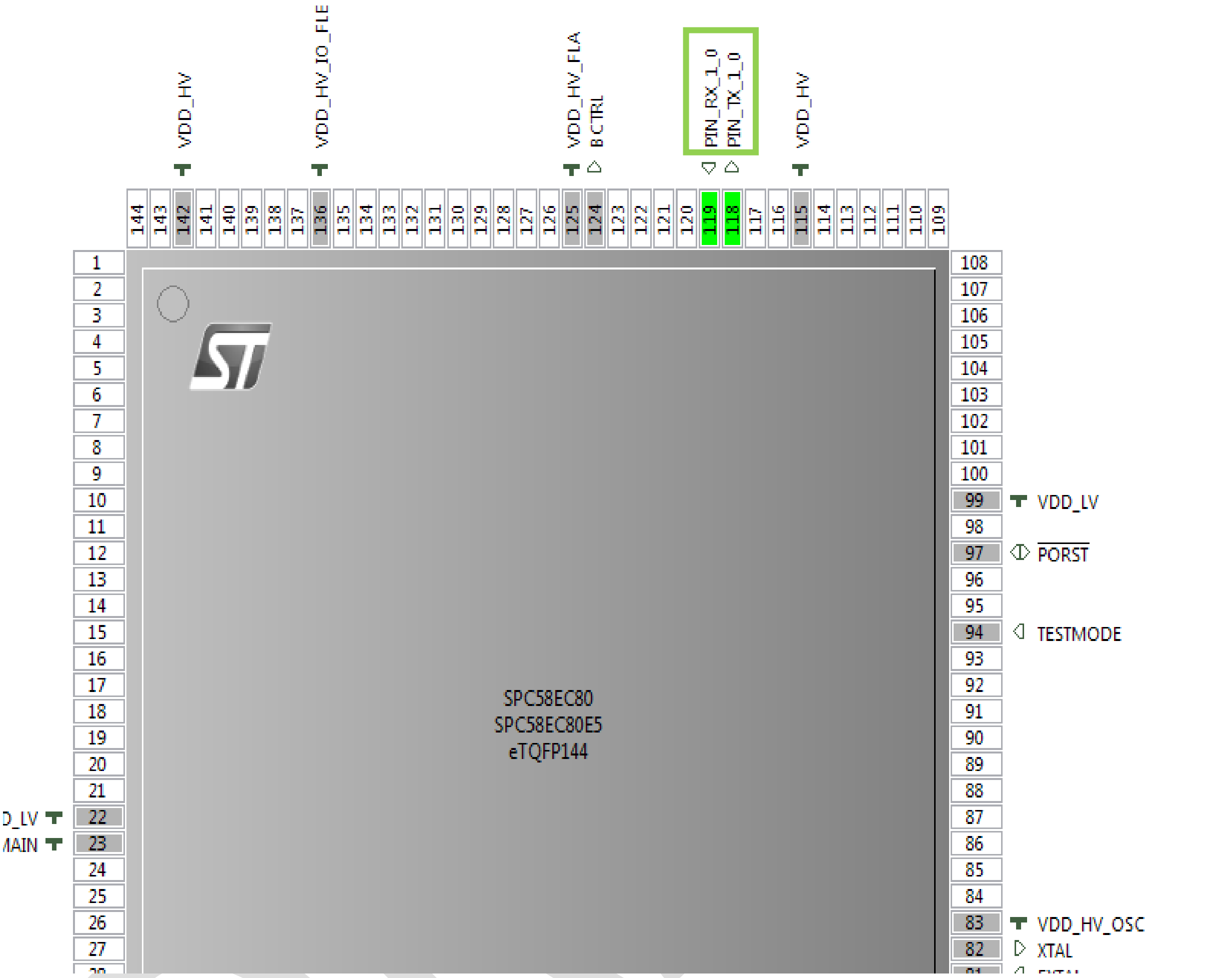
In this case, it is assumed that the user knows which pins to activate, which signal to associate for each of them and the direction of these signals. It is sufficient to click on the chosen pin on the PinMap and proceed with the desired configuration setting.

Step1. Open PinMap editor:

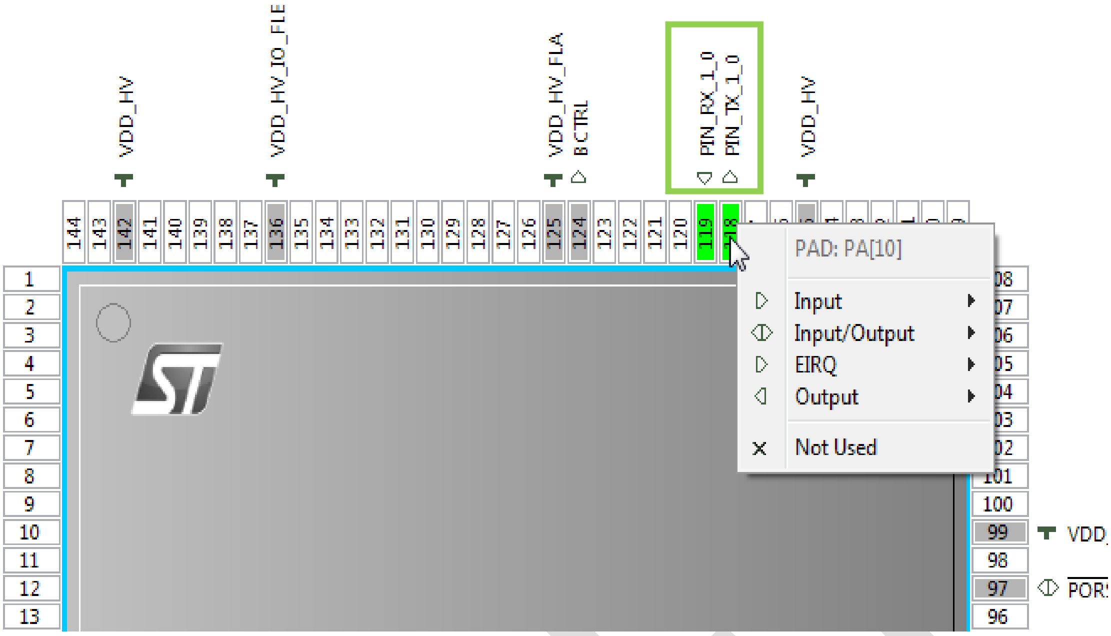
The user can use PinMap by clicking the icon shown in the figure below:



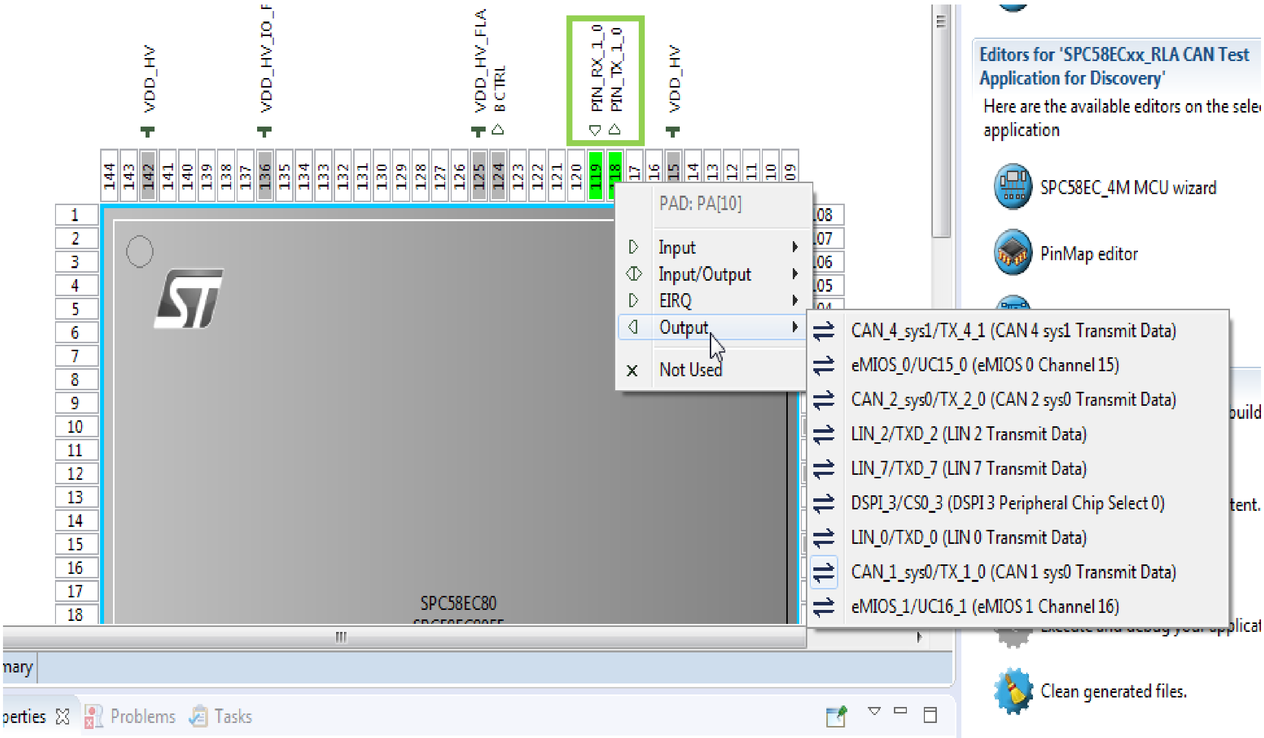
Step2. Pin selection:



Select the pin and click the right button to select the signal direction: Input, Output or I/O.



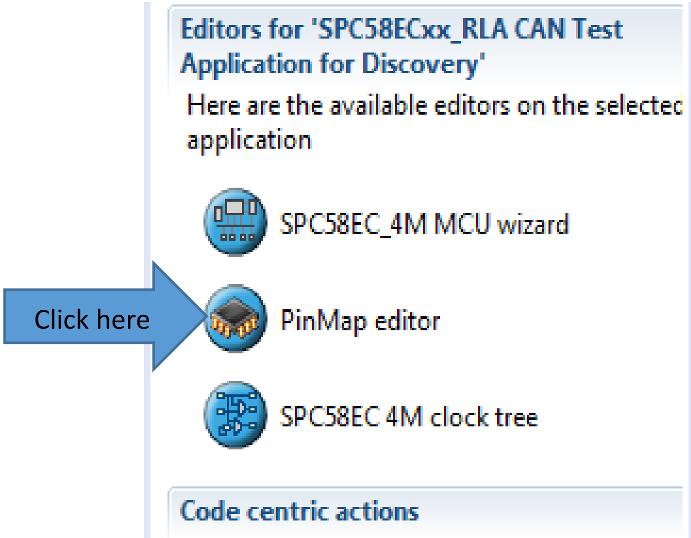
After selecting the signal direction, select the pre-loaded configuration on SPC5-Studio related to the signal that the user needs for its application. If the user commits an error in choosing the pin direction, then he/she will not find the pre-loaded configuration related to the signal of the chosen driver.



Case 2. Pin selection from function:

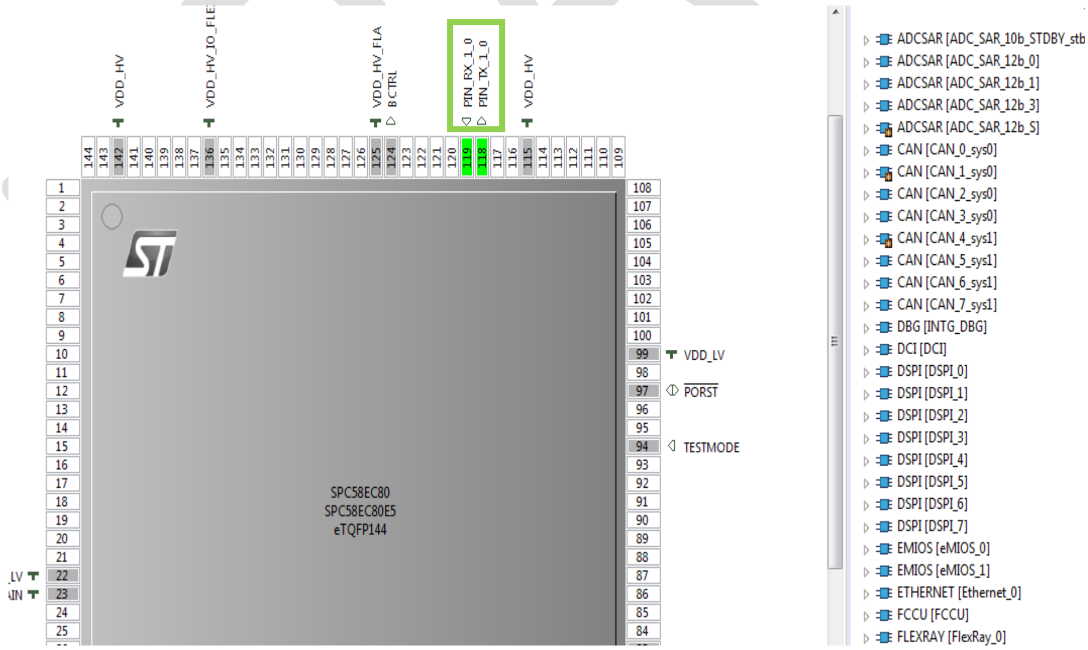
In this case, the pin is selected based on its driver. By clicking on the driver, the user will see a list of signals to be configured. For each signal, a list of all available pins will be shown. Next step for the user is to associate the chosen pin to the related signal and driver (please see above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

Step1. Open PinMap editor:

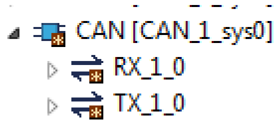


Step2. Pin selection:

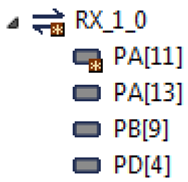
In this case the user will choose the pin according to the driver



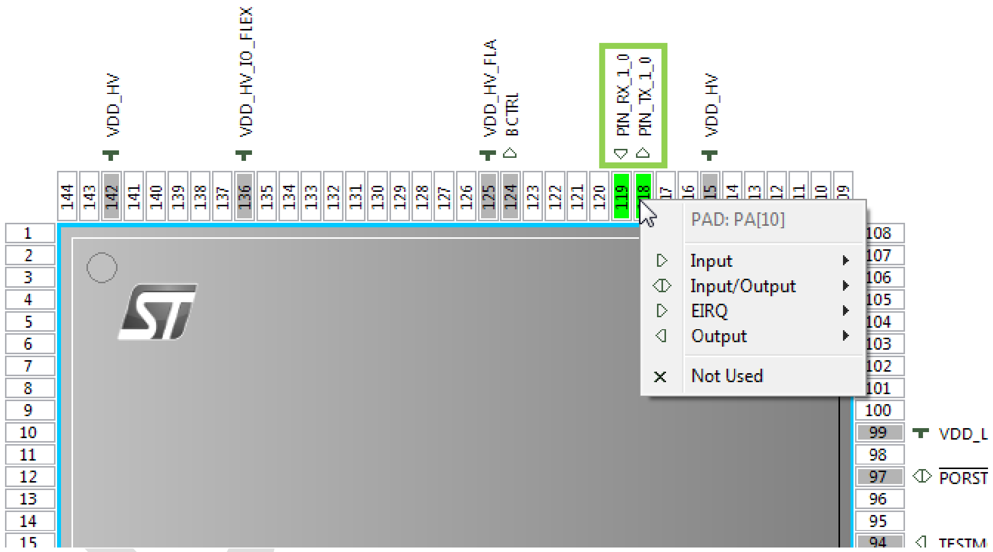
After selecting the driver it is possible to select the signal, from the signal list, necessary to configure the pin.



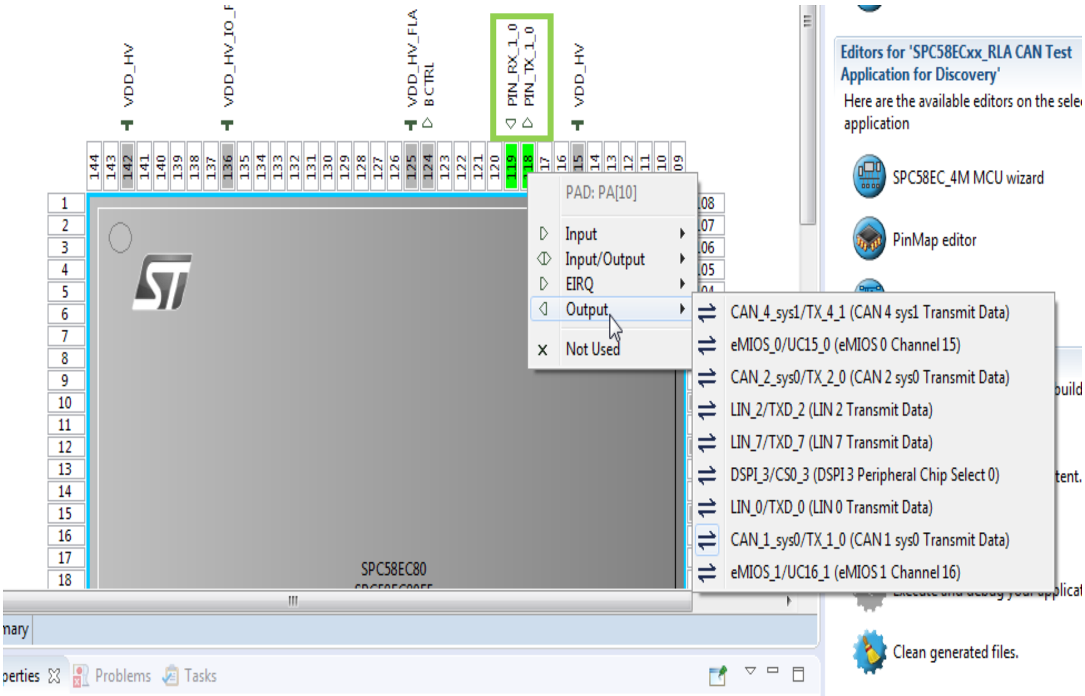
For each signal related to the specific driver, all the eligible pins are shown with reference to the microcontroller ports:



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

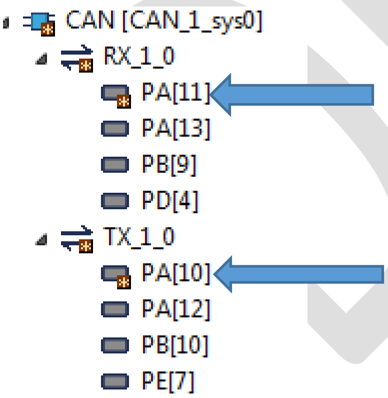


After selecting the signal direction, select the preloaded configuration.



For CAN2 (refer to TABLE-1), there are two signals to set, Rx and Tx. **Rx is an Input signal and Tx is a output signal.**

A possible pin configuration for our application will be shown. To use the controller CAN2 (CAN1 subsystem0).



NOTE: by default each SPC5Studio application provides a pin configuration suitable to run fine on the target. Anyway, as detailed in this document, user can modify and tune the setup according our needed by using the PinMap wizard.

6 CAN bus general description

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 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. The wires are implemented with a twisted pair with a 120 Ω (nominal) characteristic impedance.

Each node is able to send and receive messages, but not simultaneously. A message (or frame) consists primarily of the ID (identifier), which is the message priority.

The standard CAN message has at most length of 8 bytes of data. The CAN-FD extends the data frame length to 64 bytes per frame. What was said as shown on the following Figure 5.

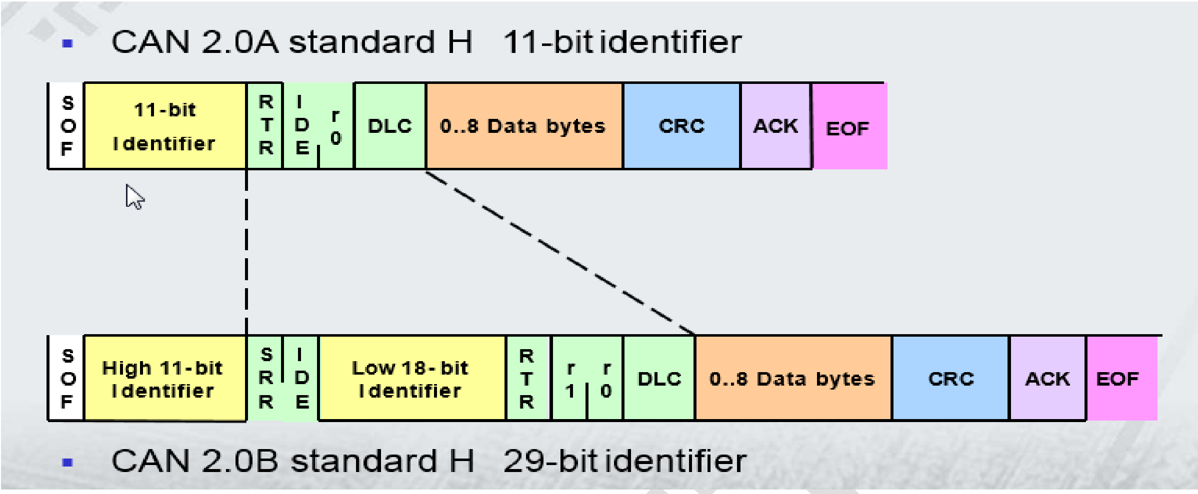


Figure 5: CAN Frame STANDARD vs EXTENDED DATA FRAMES

The nodes are always listening and will receive all 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" messages.

The dominant messages are those that will have the first bit equal to 0, while the recessive messages are those that will have the first bit equal to 1.

In the case of contention of a bus it is necessary to determine the priority of each message.

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

- the Arbitration Field,
- the Data Field,
- the CRC Field,
- the Acknowledgment Field.

The Arbitration Field is the part of the Data Frame that has the task of determining the priority of the message.

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

Thus, the lower the identification number of the binary message, the higher is his 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 be attributed to its many methods of error checking.
If a message is not received correctly, 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 the message level and 2 a single bit level:
Bit Stuffing Error: a node during the transmission inserts a bit of opposite polarity after 5 of consecutive bits of the same polarity; this is called a bit of stuffing. Therefore a node that receives more than 5 consecutive bits of equal sign will detect an error of this type.

What was said as shown on the following Figure 6.

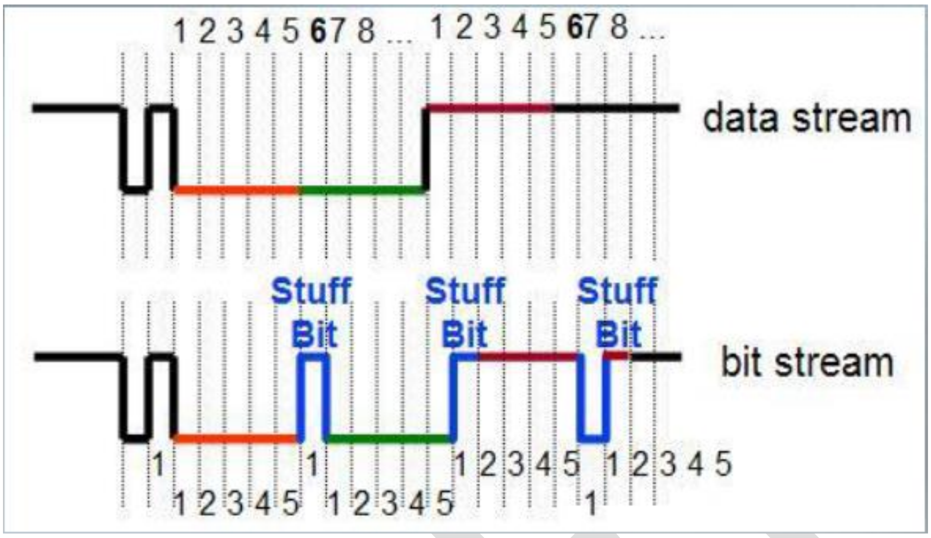


Figure 6: Bit Stuffing

- **Bit Error**: a transmitting node always listens to the bus to check the correspondence with what it is transmitting.
  If a bit other than the one sent is received, a bit error will be signaled.
- **Checksum Error**: The checksum is a sequence of bits, which, associated with the transmitted packet, 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 are obtained by processing the input data which are cyclically run through a logical network.
- **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 again to send the message.

- **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 overwrite 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 the transmitting node does not receive this message, it will send an acknowledgment error message.

7 Low Level Driver setup

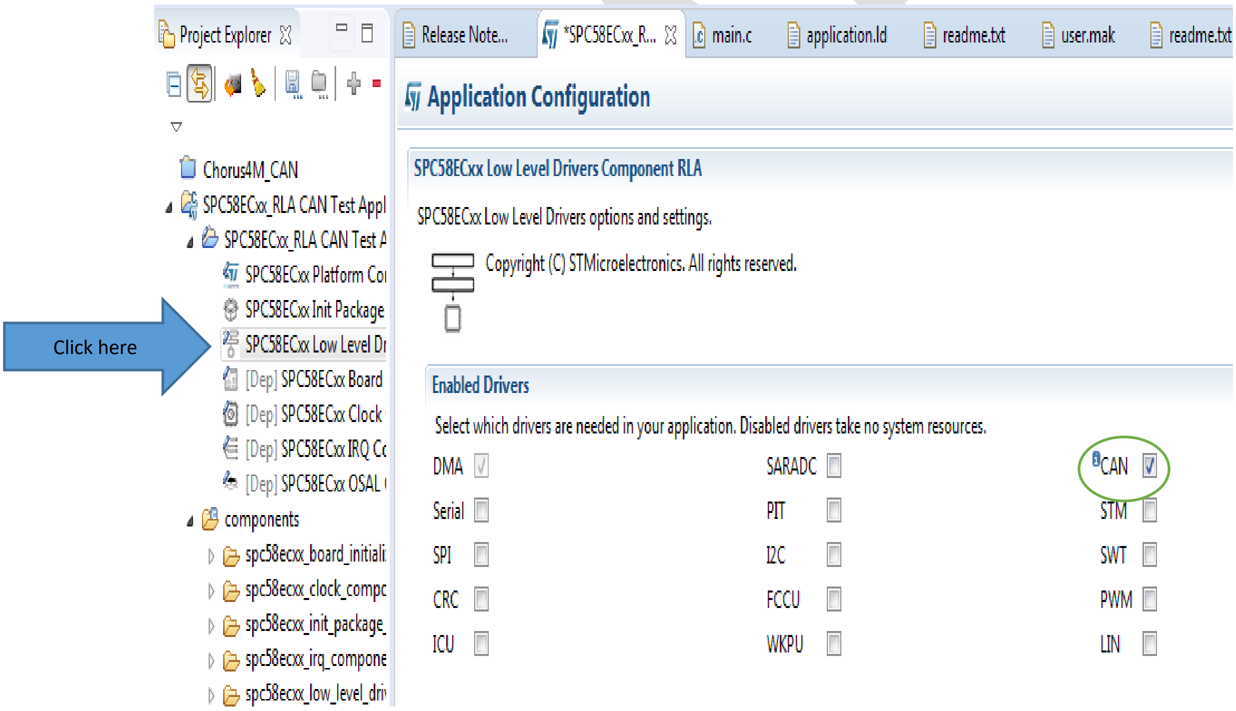
Low Level Drivers Component in SPC5-Studio, contains a collection of low level drivers for Register Level Access (RLA) components. RLA is a feature available in SPC5-Studio to allow easy and direct access to MCU and peripheral registers. The RLA component can be added and configured via the Application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

The Low lever driver concerns with 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.

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.



7.1 CAN Component

SPC58EC-DISP has two CAN subsystems implemented, as mentioned in the chapter 2:

- CAN Subsystem 0
- CAN Subsystem 1.

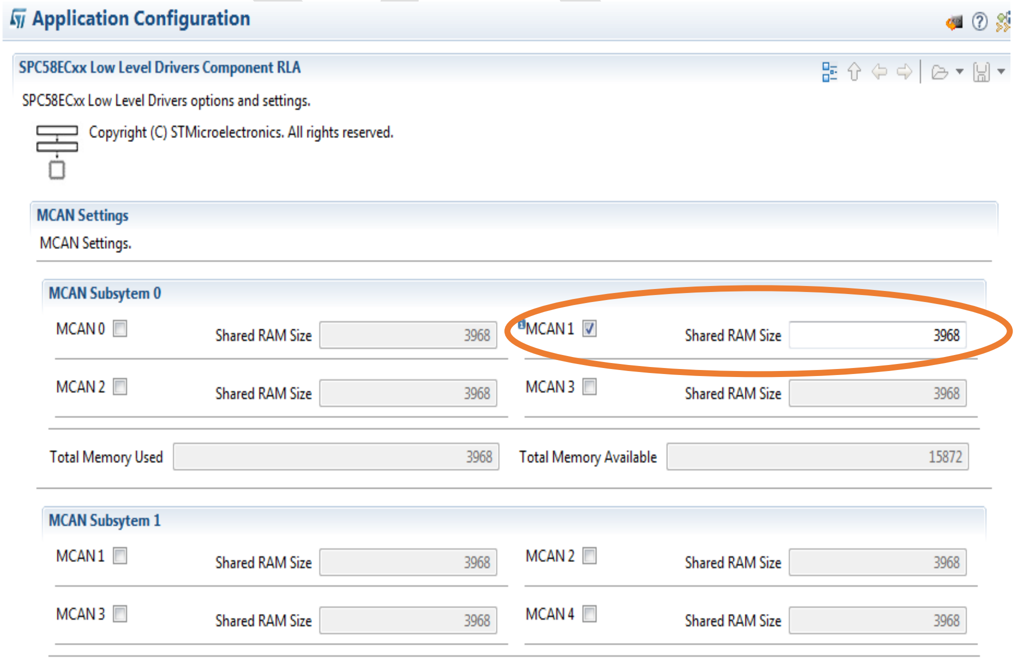
Table-1 lists the different CAN controllers inside each subsystem

Note: All CAN instances are supporting ISO CAN FD mode, the CAN-FD will be explored in paragraph 7.5.

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.



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

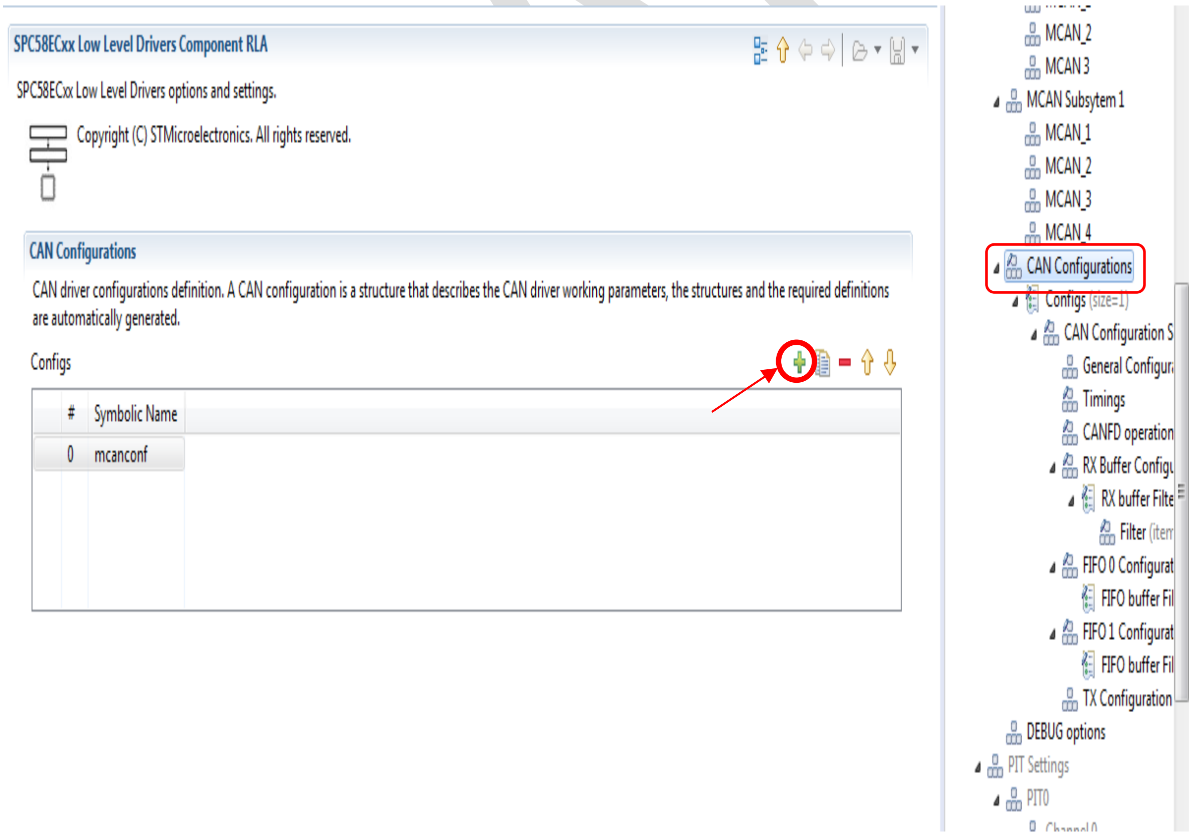
2) MCAN configuration:

Selecting on CAN Configuration a window will be opened allows user to add and set up 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. A new configuration will be created with the following symbolic name: *mcanconf*.



In detail: click in Outline and a window will open in which you can create a new configuration by clicking the green icon”   ”.

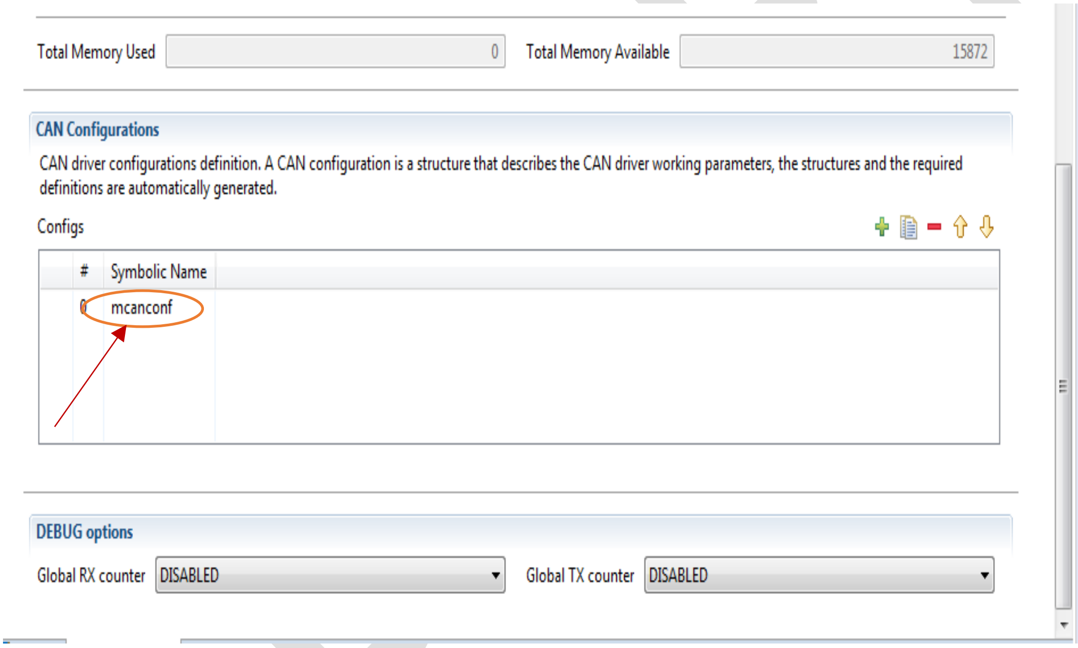
Obviously to remove the configuration created you have to select it and then click the red icon “   ”.

In the next paragraphs it will be shown what you need to do to rename the created configuration.

7.2 CAN Configuration Setting

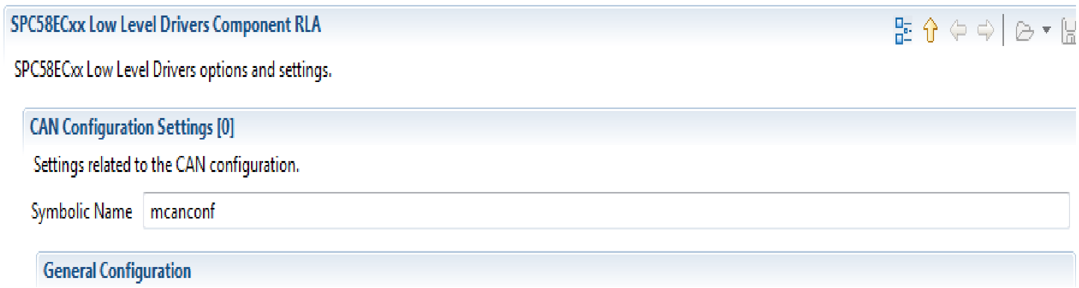
In this short paragraph it will be explained how the user can rename the configuration created.

After creating a new configuration, click on mcanconf.



Clicking on the configuration will open the window that contains all the fields that the user needs to configure his MCAN.

In this paragraph the field "CAN Configuration Setting [0]" will be only analyzed, useful for modifying the Symbolic Name.

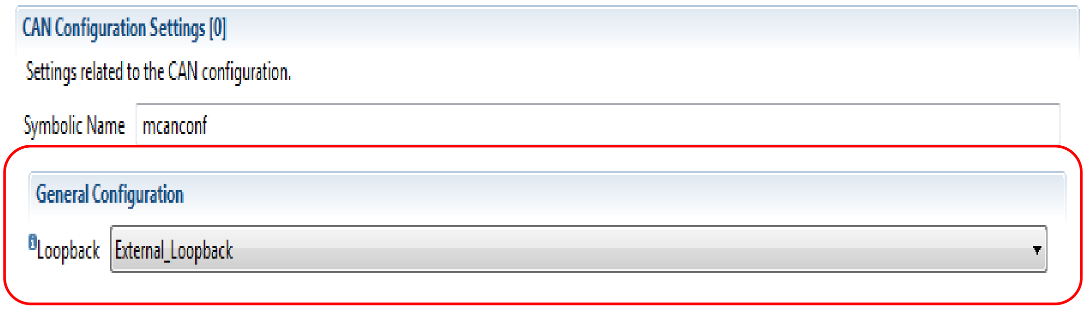


The following fields will require certain theoretical concepts that will be investigated and explained in the following paragraphs, also inherent with the MCAN configuration.

In this application uses only the MCAN1 of subsystem0.

7.3 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.



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 mean 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. The Figure 3 shows the connection of the Tx and Rx signals to the MCAN in External Loop Back mode. This mode is provided for the hardware self-test.
   To be independent from external stimulation, the M_CAN ignores acknowledge errors in Loop Back 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.

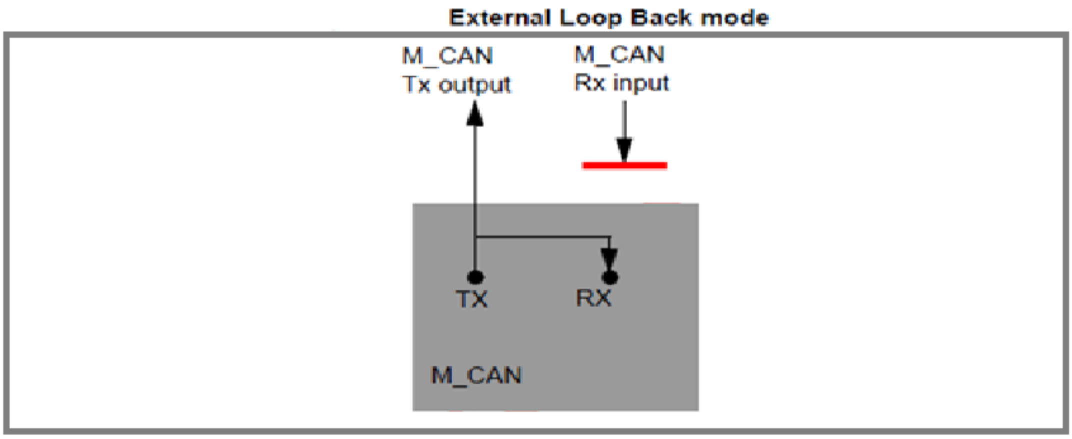


Figure 7: External Loopback on SPC584Cx/SPC58ECx

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. The Figure 4 shows the connection of MCAN Tx pin and Rx to the M_CAN in case of Internal LoopBack mode.

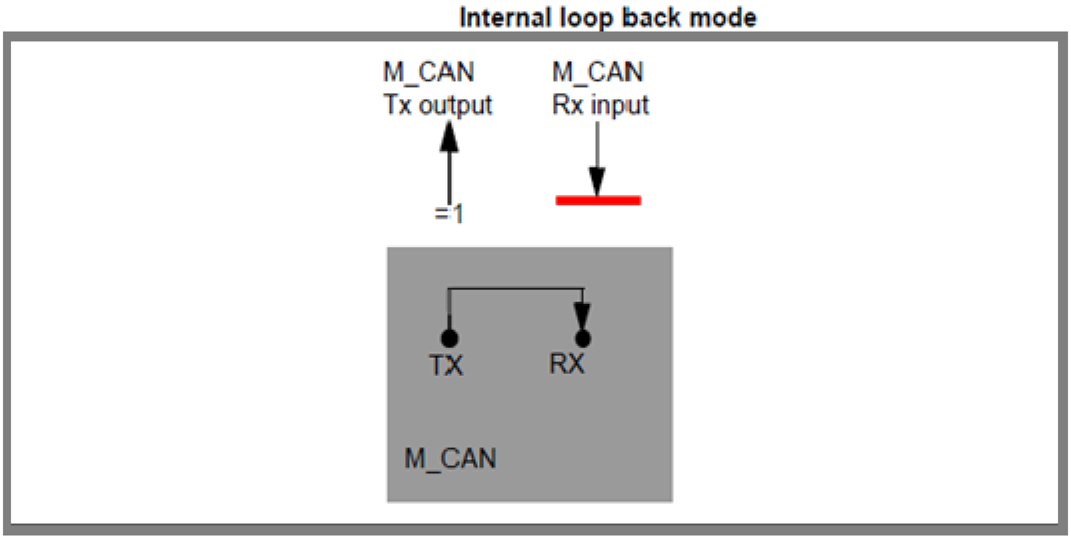


Figure 8: Internal Loopback on SPC584Cx/SPC58ECx

3. No Loopback: The real transmission mode is used. The figure shows the 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).

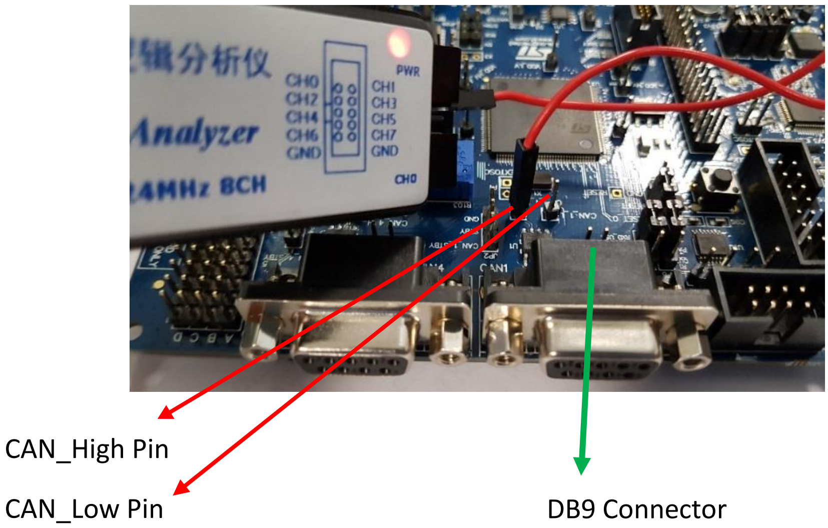


Figure 9: The ways to perform a real CAN transmission

7.4 Clock and timing

7.4.1 Protocol timing setting

The next step for the configuration of the Low Level Drivers is the setting of the clock and timing parameters.

In this field it is possible to set the clock values to obtain a bit rate value for the CAN subsystem.

It is important to underline that in the CAN protocol there is the possibility to set the bit time of the network. 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 of the node.

The Clock Prescaler is a frequency regulator. 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 in Figure 10.

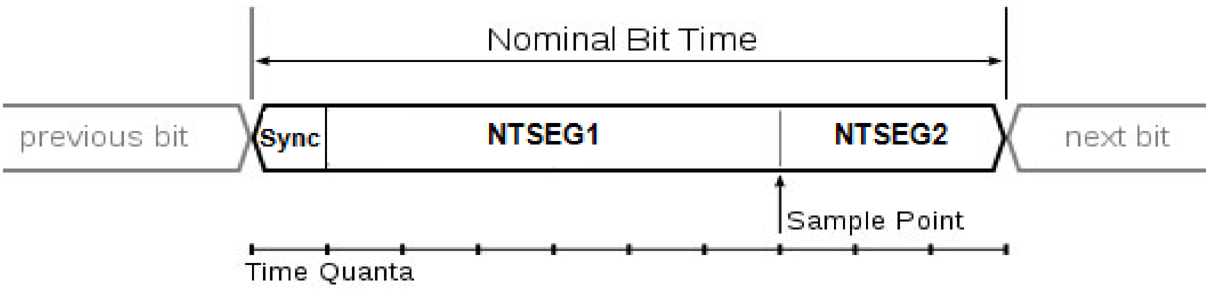
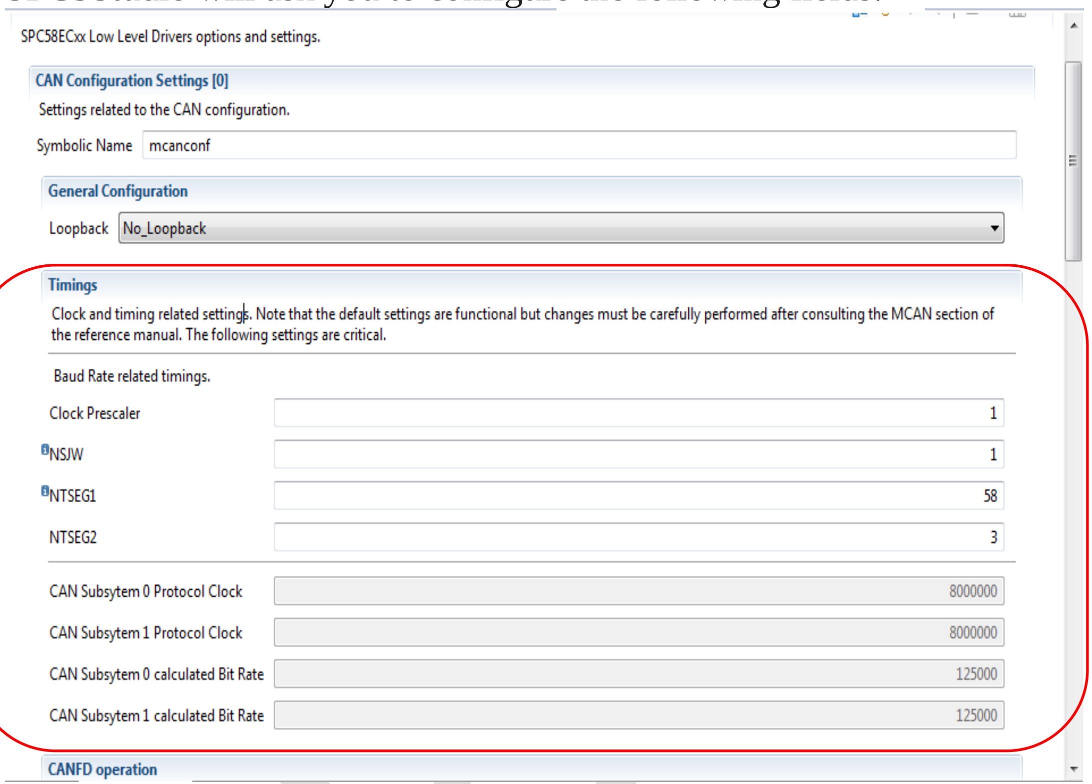


Figure10: Bit Rate

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 RM0407 Reference manual)

SPC5Studio will ask you to configure the following fields:



NSJW: As a result of Resynchronization, the NTSEG1 may be lengthened or 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.

All these 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 following figure.

7.4.2 Divide Clock System Configuration

CAN Protocol Clock is the clock that controller uses to define the Time Quanta period and configure the bit time.

To properly configuration the CAN controller two clocks must be provided.

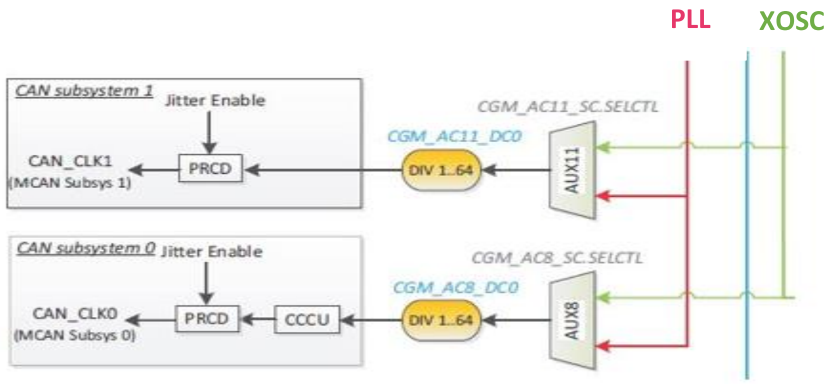


Figure 11: CAN Protocol Clock

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

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.

7.5 CAN-Flexible Data Rate Overview

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

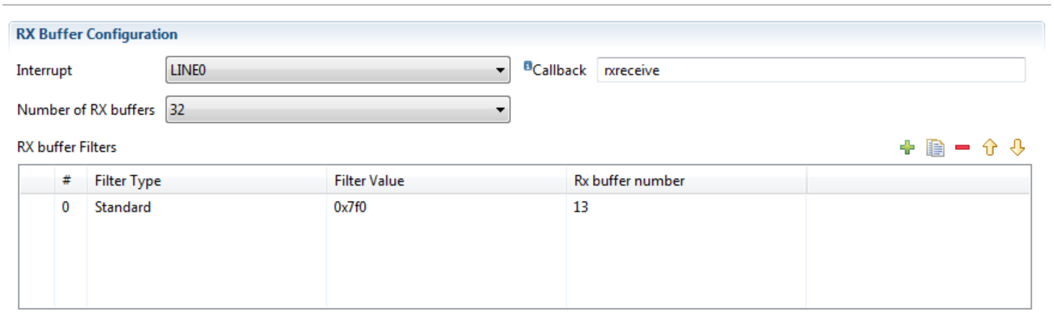
All the CAN controllers available of the SPC58EC-MCU are CAN-FD. In this application the Extended messages are not used, which are those used by the CAN-FD, but only Standard messages typical of the traditional CAN bus.

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.
The CAN-FD protocol has lower latency and has greater latency and higher bandwidth.
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.

7.6 CAN Receive Buffer (RX Buffer) Configuration

The Rx Buffer is a memory area used by messages received.
To store the incoming message buffer show below the respective configuration of Rx buffer for subsystem 0:



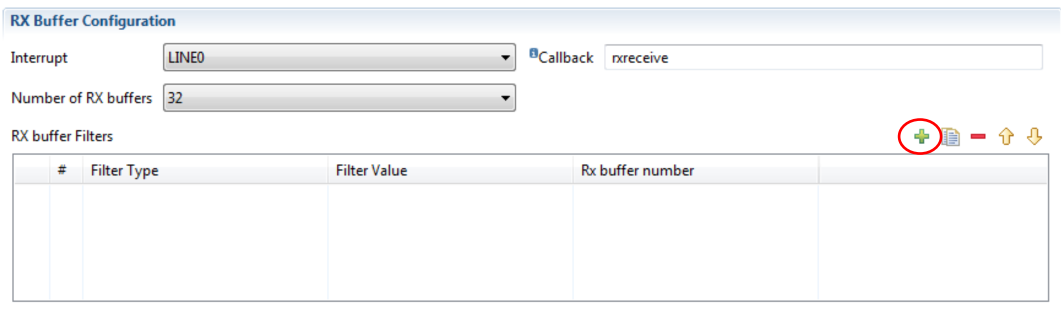
The user can decide to disable or enable, for the subsystem0 or for the subsystem1, the interrupt to which a specific callback function will be associated, it is possible to fix the number of Rx buffers, between 0 and 64.

The callback function allows the developer to add own application a specific entry point at design time in order to invoke inside the ISR, own specialized routine.

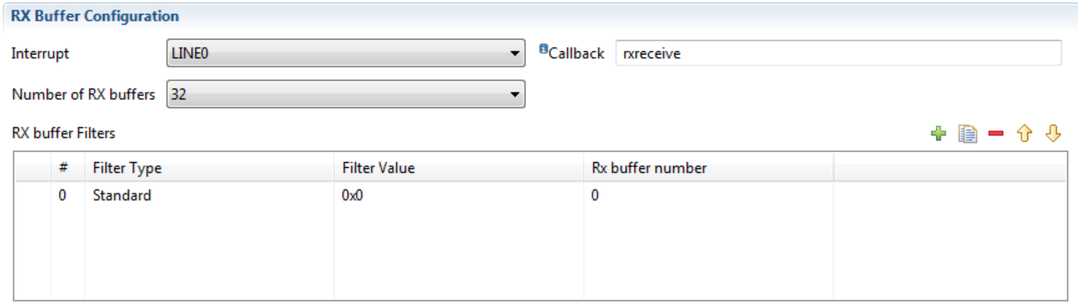
Interrupts are requests that arrive at the MCU while it is working and executing the instructions contained in a program. IRQs typically arrive from peripherals and ask the microcontroller to interrupt their work to respond to peripheral needs. In some cases, the MCU can reject the request and refuse 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, to avoid overloading the computational load of the network.

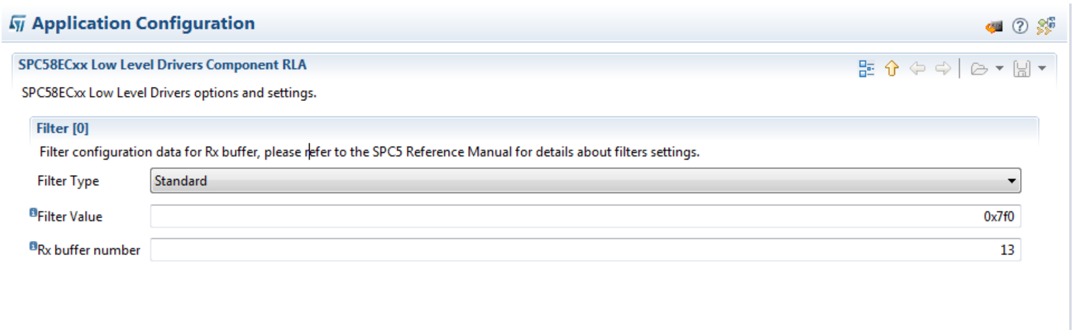
How add and configure a filter for the Rx buffer has shown under:



Click the green icon “ ” and so a filter will be added, which obviously will have to be set:



To configure the user will have to double click on the generated filter and choose the type of filter (Standard or Extended), the value and the Rx buffer number. Obviously the latter must be less than or equal to the chosen Number of Rx buffers:



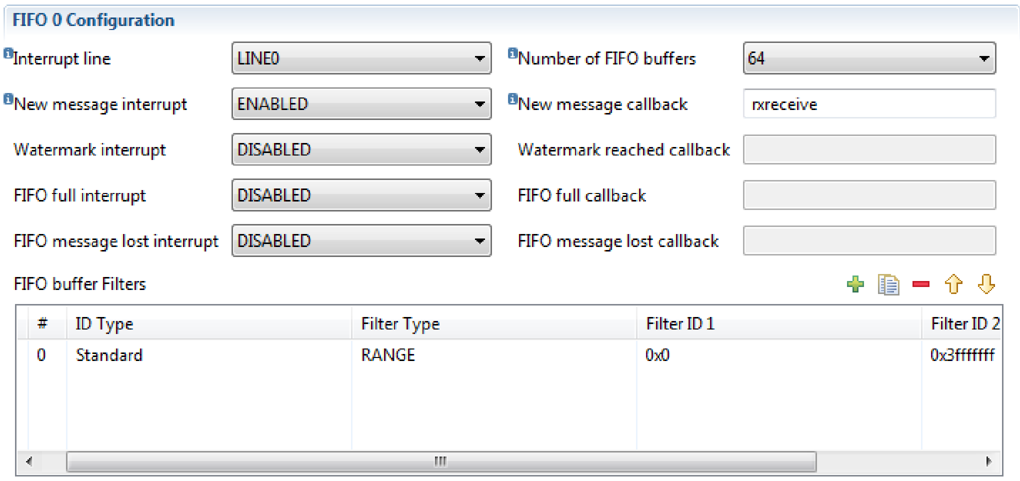
The Filter Value is set by the user and is used to filter the values that the Rx buffer, relative to a given CAN instance, must receive.

7.7 FIFO Configuration

The FIFO configuration is used when the volume of messages is relevant. For the proper managementeach for each messager,the CAN messager shall include a ID.

To use the Rx buffer, the ID filters of the messages should be inserted one to the time.

To simplify this process, the FIFO allows us to insert a RANGE of filters ID. An example of FIFO Configuration is shown below:



7.8 Tx Configuration

There are several Modes to manage Tx Buffers:

1. Dedicated,
2. FIFO,
3. QUEUE,
4. MIXED FIFO or QUEUE.

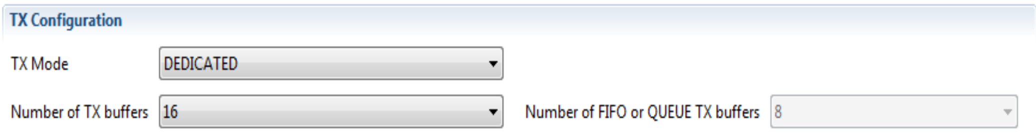
The Dedicated Tx Buffers are intended for message transmission. Each Dedicated Tx Buffer is configured with a specific Message ID.

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 his 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.



Note that for the DEDICATED Mode SPC5Studio gives the possibility to set only the TX buffer number.

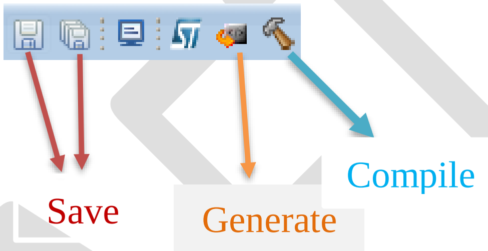
For QUEUE and FIFO Mode you can set the number of FIFO or QUEUE Buffers and instead for the MIXED Mode you can set both.

8 Generate Application Code and Compile Application

Next step is to 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:



9 The Code Overview

In this chapter will be presented the code used in our application and will be illustrated some meaningful parts of 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.`

9.1 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 above low-level driver set-up (see 7.2). The configuration used is mcanconf has been set 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.

.

9.2 Prepare frame and transmit a message

After the driver CAN has been initialized, the transmitting structure message is define 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: it indicates the type of message. There are two standard options, CAN_ID_STD or extender for CANFD (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, in this step is shown how to send it.

To send the message is used the second key function:

```
can_lld_transmit(&CAND2, CAN_DEDICATED_TXBUFFER, &txf);
```

This function (*can_lld_trasmit*) is handling the transmission of messages, it needs three 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 was already explained in 7.8. 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:

TX_Mode	Name_SPC5Studio
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

TABLE-2 Values Tx_Mode

- &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.

9.3 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 Chorus 4M 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: It indicates on which controller to receive.
- RX_Buffer: The concept was already explained in 7.6, 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.

RX_Buffer	Name_SPC5Studio
NOT DEFINITED	CAN_ANY_RXBUFFER
DEDICATED	CAN_DEDICATED_RXBUFFER
FIFO	CAN_FIFO_RXBUFFER

TABLE-3 Values Rx_Mode

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

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

- TIME: it indicates the timestamp. This is a number that expresses a temporal magnitude; more precisely, it corresponds to the number of seconds that have 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: it indicates the type of message. There are two types: CAN_ID_STD or CAN_ID_XTD (for CAN-FD).
- ID: it is the priority of message.
- DLC: it indicates the Data Length Code.
- Data32: it is the data Frame.

9.4 Message Status

In the protocol there are not many message status, 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.

9.5 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()
```

Often to synchronize the transmission time with the interrupt it is 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);
```

9.6 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. Inside these libraries are 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 recall 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: `can_lld_start`

```
  /* 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 are not loss of packets then the LED 2, in the MCU discovery board, will blink.

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

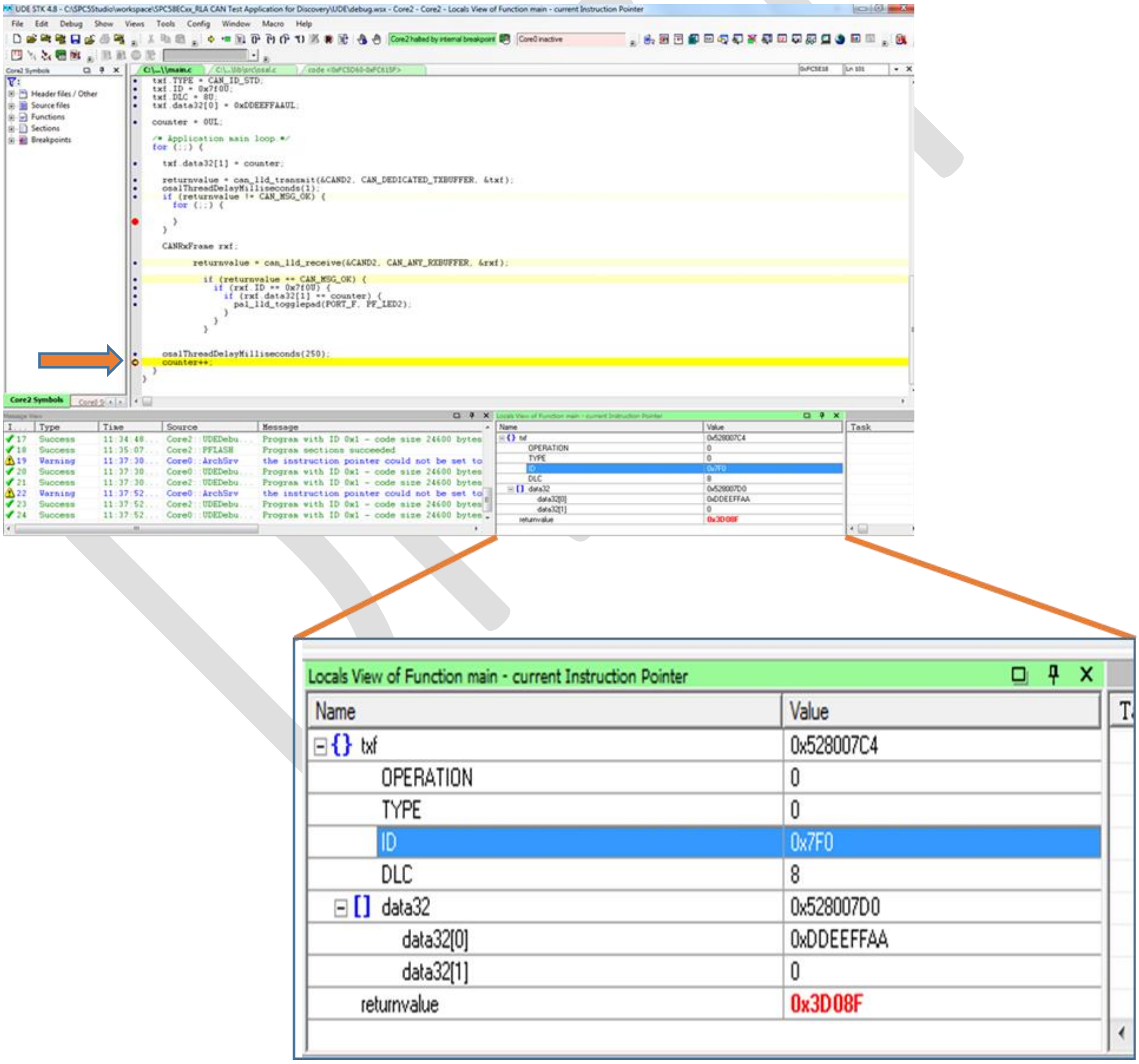
10 “No LoopBack” Example

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 will be to verify the correct transmission of the message through the use of UDE STK 4.8.

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

Below are shown the results obtained in the respective break points.



The reception phase will be verified with the use a" CAN Analyzer".

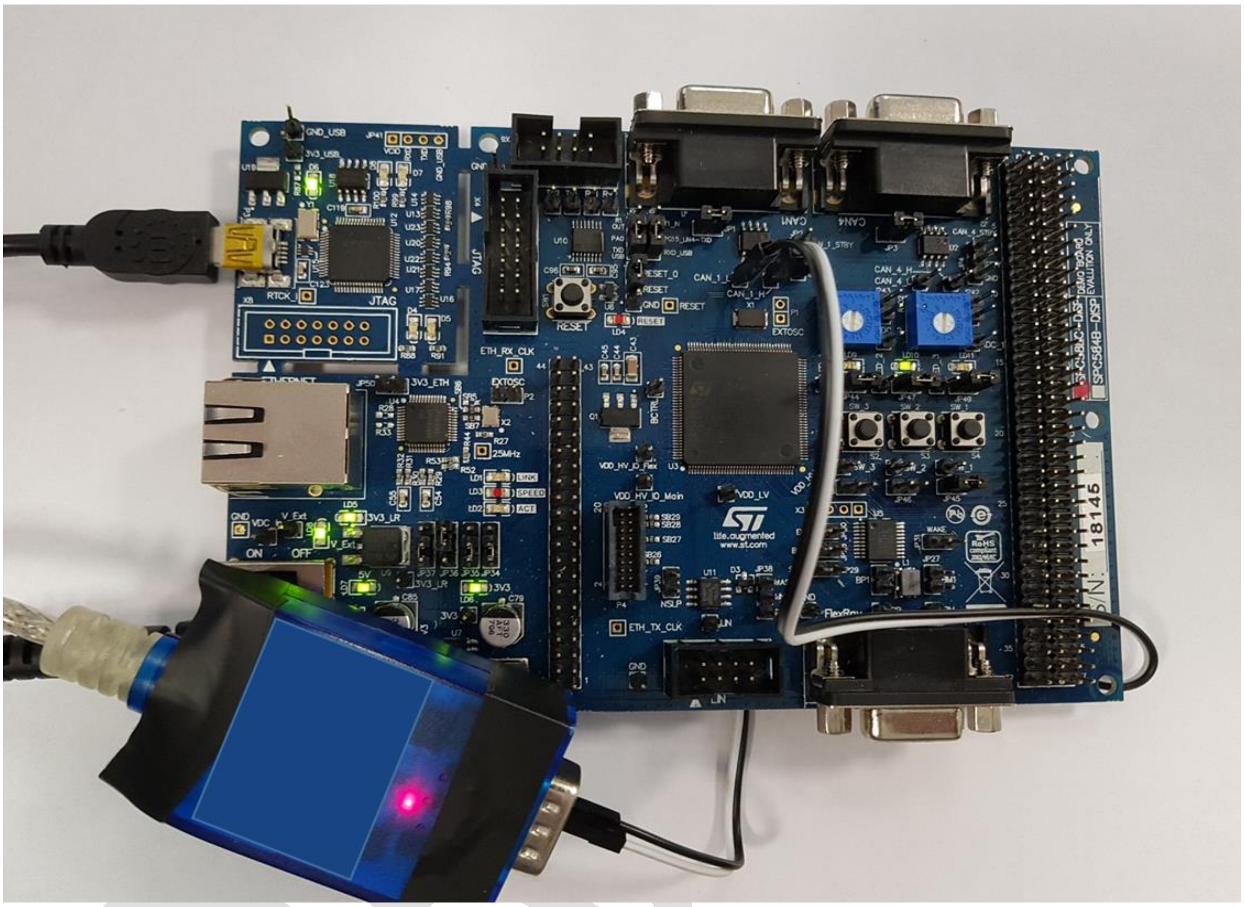
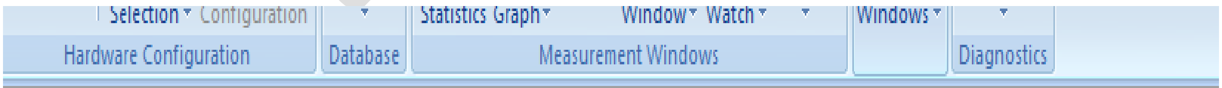


Figure 12: CAN Connection

This CAN Analyzer be equipped with a special tool that allows to see on a screen the data analyzed.

The message received in this example is shown below.



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.

Without any ID filters applied no message will be received.

A.1 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
tq	Time quanta

A.2 Reference documents

- RM0407 Reference manual

Revision history

Date	Revision	Changes
09/13/2018	V1.0	