MULTICAN_GW_TX_FIFO_1 for KIT_AURIX_TC297_TFT

MULTICAN in GATEWAY mode using TX FIFO

AURIX™ TC2xx Microcontroller Training
V1.0.0

Scope of work

MULTICAN in Gateway mode is used to exchange data using a gateway with a TX FIFO structure between multiple nodes, implemented in the same device using Loop-Back mode.

The CAN messages are sent from CAN node 2 over CAN bus (in case of Loop-Back mode all nodes can access the internal bus). CAN node 0 receives the transmitted messages but also immediately gateways the received data to CAN node 1. Three message objects allocated to CAN node 1 define a TX FIFO buffer structure. Immediately, upon the reception of the data via gateway, CAN node 1 transmits the received data. The data transmitted by the CAN node 1 is received by the CAN node 3. The content of the received data will be compared to the content of the transmitted CAN messages together with the FIFO status check and in case of success, a LED is turned on to confirm successful message reception.

Introduction

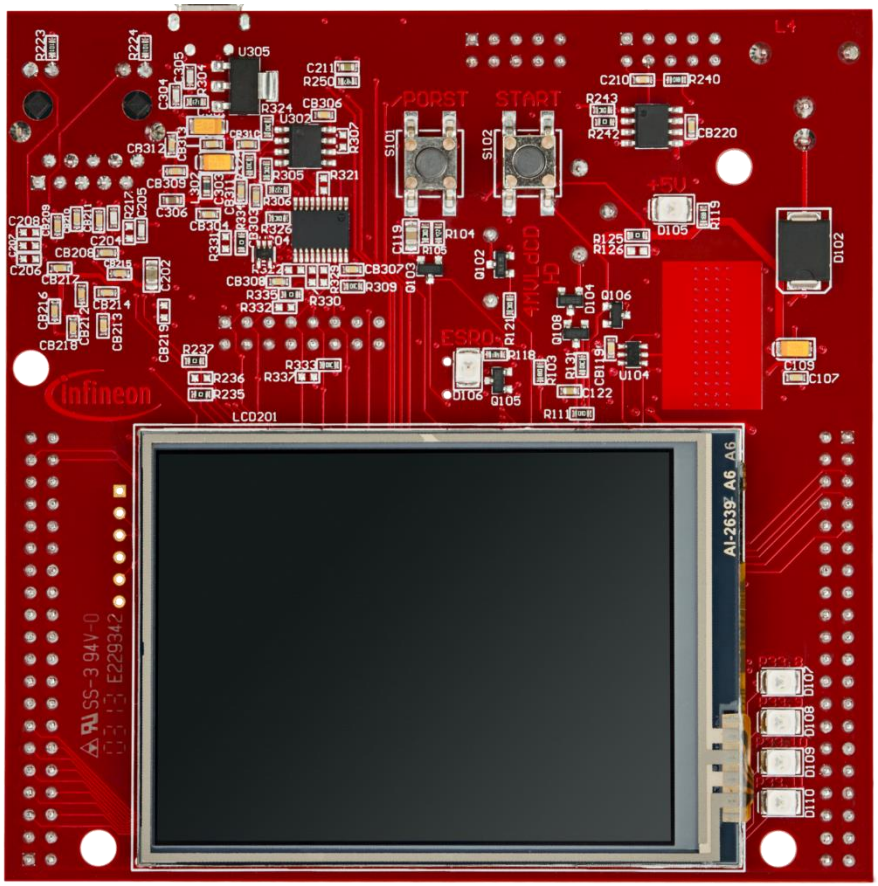
- The MultiCAN+ module provides a communication interface which is **fully compliant with CAN specification V2.0B (active)** and to **CAN FD ISO11898-1 DIS version 2014**, providing communication up to **1 Mbit/s in Classical CAN** (ISO 11898-1:2003(E)mode) and/or **CAN FD up to 5 Mbit/s** (dependent on frequency and nodes).

- The MultiCAN+ module consists of several **CAN nodes** (in case of AURIX™ TC29x device, 4 nodes) which are **CAN FD capable**. Each CAN node communicates over two pins (TXD and RXD). Additionally, there is an internal **Loop-Back Mode** functionality available for test purposes.

- All CAN nodes share a common set of 256 **message objects**. Each message object can be individually allocated to one of the CAN nodes. Besides serving as a **storage container for incoming and outgoing frames**, message objects can be combined to build **gateways** between the CAN nodes or to setup a **FIFO buffer**.

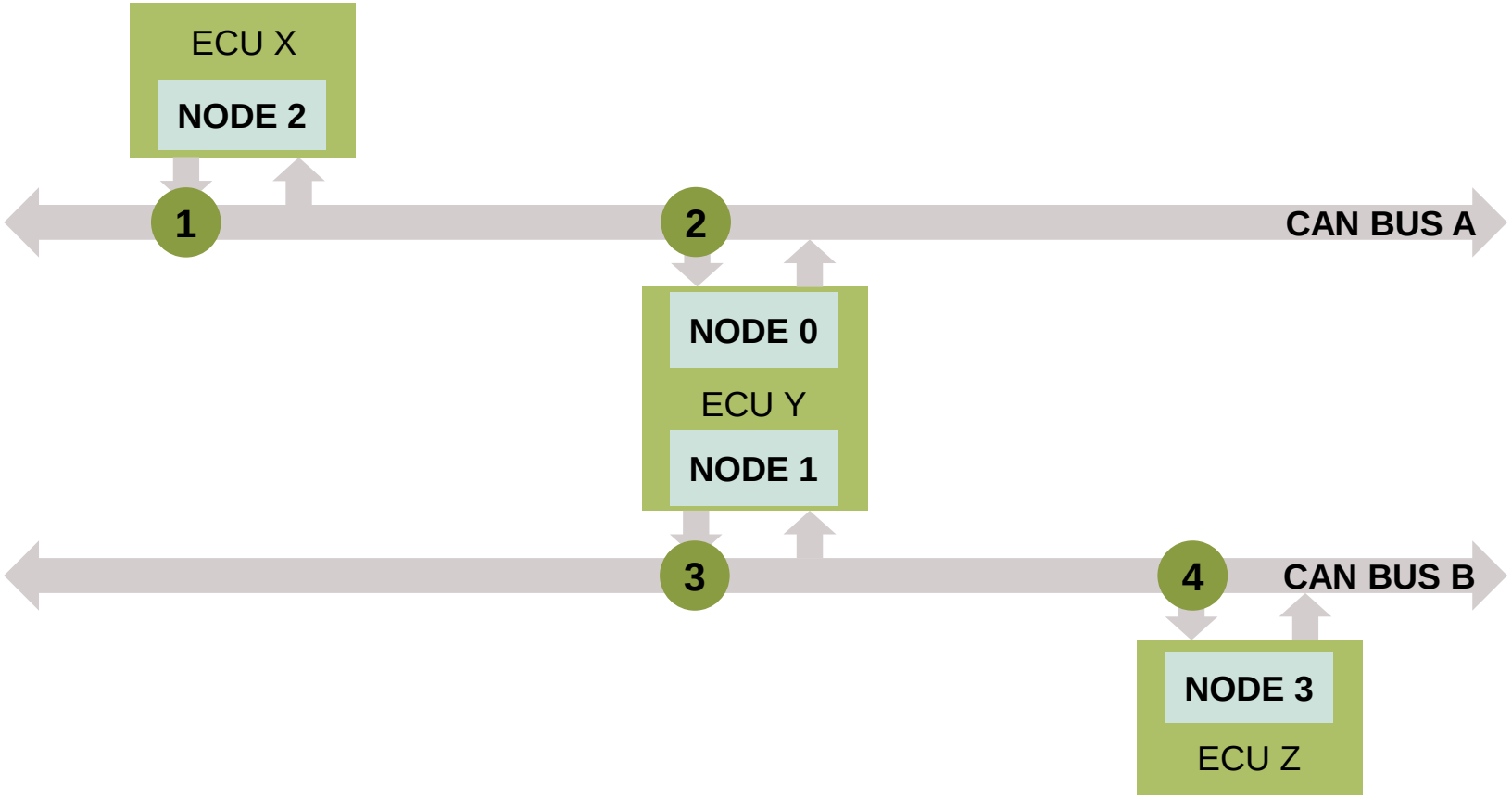
Hardware setup

This code example has been developed for the board KIT_AURIX_TC297_TFT_BC-Step.



Implementation

Application use case that is covered with this example:



Implementation

Application use case short description:

- The internal Loop-Back mode allows to implement the previously shown application use case by using just one AURIX™ TC29x device with 4 CAN nodes.
- In this example the Gateway mode functionality is used. The basic idea is to transfer data between two independent CAN buses without any CPU intervention.
- Additionally, in case of high CPU load, it might be difficult to process a series of CAN frames in time. This might happen if multiple messages are received or must be transmitted in short time. For this reason, a FIFO buffer structure usage is shown in this example as well.

1. The CAN node 2 sends a CAN message over the CAN bus (in case of Loop-Back mode all nodes can access the internal bus).
2. CAN node 0 receives the message and immediately gateways the received data to CAN node 1. Three message objects (MO1 as FIFO base object and MO2-MO3 as FIFO slave objects), allocated to CAN node 1, define a TX FIFO buffer structure.
3. Immediately upon the reception of the data via gateway, CAN node 1 transmits the received data.
4. The data transmitted by the CAN node 1 is received by the CAN node 3.

Implementation

Application code can be separated into four segments:

- Initialization of the MultiCAN+ module with the accompanying node and message objects initialization, implemented in the ***initMultican()*** function.
- Initialization of the port pin connected to the LED (D107 on the board). The LED is used to verify the success of a CAN message reception. This is done inside the ***initLed()*** function.
- Transmission of the configured CAN messages, implemented in the ***transmitCanMessages()*** function.
- Verification of the received CAN messages, implemented in the ***verifyCanMessages()*** function.

An additional interrupt service routine (ISR) is implemented:

- On RX interrupt, the ISR reads the received CAN message and, in case of no errors, increments the counter to indicate the number of successfully received CAN messages (realized by ***canIsrRxHandler()*** function).

Implementation

MultiCAN+ module initialization

Initialization is performed in three phases:

- A default CAN module configuration is loaded into the configuration structure by using the function ***IfxMultican_Can_initModuleConfig()***.
  Afterwards, the initialization of the CAN module with the user configuration is done with the function ***IfxMultican_Can_initModule()***.

- A default CAN node configuration is loaded into the configuration structure by using the function ***IfxMultican_Can_Node_initConfig()***. Initialization of the CAN nodes (0, 1, 2, and 3) with the different CAN node ID values and definition of Loop-Back Mode usage for all nodes is done with the function ***IfxMultican_Can_Node_init()***.

Implementation

MultiCAN+ module initialization

- A default CAN message object configuration is loaded into the configuration structure by using the function ***IfxMultican_Can_MsgObj_initConfig()***.
  The initialization of the CAN message objects with different configurations is done by the ***IfxMultican_Can_MsgObj_init()*** function.

All functions used for the MultiCAN+ module initialization are declared in the iLLD header ***IfxMultican_Can.h***.

Due to the multiple message objects used in this application use case and the complexity of their configuration, following slides cover the configuration of each message object.

Implementation

Message objects configuration

- Gateway source object (**MO0**):
  - ***canMsgObjConfig.msgObjId = 0*** – defines the message object ID
  - ***canMsgObjConfig.messageId = 0x444*** – defines the CAN message ID used during arbitration phase (shares the same ID with message object 10 (**MO10**))
  - ***canMsgObjConfig.msgObjCount = 2*** – defines the number of FIFO slave objects that is used as gateway DESTINATION object
  - ***canMsgObjConfig.frame = IfxMultican_Frame_receive*** – defines the message object as a receive message object
  - ***canMsgObjConfig.firstSlaveObjId = 2*** – defines the first slave object of the FIFO to be the first message object after TX FIFO base object
  - ***canMsgObjConfig.gatewayTransfers = TRUE*** – enables gateway transfers (defines this message object as gateway source object)
  - ***canMsgObjConfig.gatewayConfig.copyDataLengthCode = TRUE*** – copies data length code of the gateway source object to a gateway destination object
  - ***canMsgObjConfig.gatewayConfig.copyData = TRUE*** – copies data content of the gateway source object to a gateway destination object
  - ***canMsgObjConfig.gatewayConfig.copyId = FALSE*** – does NOT copy identifier (ID) of the gateway source object to a gateway destination object
  - ***canMsgObjConfig.gatewayConfig.enableTransmit = TRUE*** – enable setting TXRQ bit in the gateway destination object
  - ***canMsgObjConfig.gatewayConfig.gatewayDstObjId = 2*** – defines the first slave object of the FIFO as the gateway destination object

  Gateway source object (**MO0**) is assigned to **CAN Node 0**.

Implementation

Message objects configuration

- Gateway destination object (implemented as TX FIFO object)
  (**MO1** as FIFO base object and **MO2-MO3** as the FIFO slave objects):
  - ***canMsgObjConfig.msgObjId = 1*** – defines the message object ID
  - ***canMsgObjConfig.messageId = 0x777*** – defines the CAN message ID used during arbitration phase (shares the same ID with message object 20 (**MO20**))
  - ***canMsgObjConfig.msgObjCount = 2*** – defines the size of the structure (more than 1 message object specifies FIFO structure: **MO2 and MO3**)
  - ***canMsgObjConfig.frame = IfxMultican_Frame_transmit*** – defines the message object as a transmit message object (TX FIFO in this case)
  - ***canMsgObjConfig.firstSlaveObjId = 2*** – defines the first slave object of the FIFO to be the first message object after TX FIFO base object

  Gateway destination object (**MO1 and MO2-MO3**) is assigned to **CAN Node 1**.

Implementation

Message objects configuration

- Source standard message object (**MO10**):
  - ***canMsgObjConfig.msgObjId = 10*** – defines the message object ID
  - ***canMsgObjConfig.messageId = 0x444*** – defines the CAN message ID used during arbitration phase (shares the same ID with message object 0 (**MO0**))
  - ***canMsgObjConfig.frame = IfxMultican_Frame_transmit*** – defines the message object as a transmit message object

  Source standard message object (**MO10**) is assigned to **CAN Node 2**.

- Destination standard message object (**MO20**):
  - ***canMsgObjConfig.msgObjId = 20*** – defines the message object ID
  - ***canMsgObjConfig.messageId = 0x777*** – defines the CAN message ID used during arbitration phase (shares the same ID with message object 1 (**MO1**))
  - ***canMsgObjConfig.frame = IfxMultican_Frame_receive*** – defines the message object as a receive message object
  - ***canMsgObjConfig.rxInterrupt.enabled = TRUE*** – enables interrupt generation in case of CAN message reception
  - ***canMsgObjConfig.rxInterrupt.srcId = IfxMultican_SrcId_1*** – defines the interrupt node pointer to be used in case of an interrupt

  Destination standard message object (**MO20**) is assigned to **CAN Node 3**.

Implementation

Initialization of a pin connected to the LED

An LED is used to verify the success of a CAN message reception. Before using the LED, a port pin to which the LED is connected must be configured.

- First step is to set the port pin to level “HIGH”; this keeps the LED turned off as a default state (***IfxPort_setPinHigh()*** function).
- Second step is to set the port pin to push-pull output mode with the ***IfxPort_setPinModeOutput()*** function.
- Finally, the pad driver strength is defined through the function ***IfxPort_setPinPadDriver()***.

All functions are declared in the iLLD header ***IfxPort.h***.

Implementation

Transmission of CAN messages

Before the CAN messages are transmitted, a number of CAN messages need to be initialized. The user can change the number of CAN messages by modifying ***NUMBER_OF_CAN_MESSAGES*** macro value. The TX messages (messages that will be transmitted) are initialized with the combination of predefined content and current CAN message value. The RX messages (messages where the received CAN message will be stored) are initialized with invalid ID, data, and length value. After successful CAN transmission the values are replaced with the valid content. Following each CAN message transmission, a code execution waits until the received data has been read by the interrupt service routine.

- Initialization of both TX and RX messages is done by using ***IfxMultican_Message_init()***
- A CAN message is transmitted by using ***IfxMultican_Can_MsgObj_sendMessage()***. A CAN message is continuously transmitted as long as the returned status is ***IfxMultican_Status_notSentBusy*** (this status occurs if there is a pending transmit request).

The function ***IfxMultican_Message_init()*** is declared in the iLLD header ***IfxMultican.h*** while the ***IfxMultican_Can_MsgObj_sendMessage()*** function is declared in the iLLD header ***IfxMultican_Can.h***.

Implementation

Verification of CAN messages

After successful reception of all expected CAN messages, the Current Object Pointer (CUR) values of the gateway source and of the destination message objects are checked.
This check is performed by comparing the CUR pointer value given in the Message Object FIFO/Gateway Pointer Register **(FGPR)** of the related message object to the expected value.
The function ***IfxMultican_MsgObj_getPointer()*** returns the pointer to the related message object.
Finally, the check is performed to verify the success of CAN message transmission and reception by comparing the received ID, data, and length value with the transmitted ones. In case of success, the LED is turned on (***IfxPort_setPinLow()***) to indicate the correctness of the received messages and consequently the correctness of the CAN transmission.
The function ***IfxMultican_MsgObj_getPointer()*** is declared in iLLD header ***IfxMultican.h*** while the ***IfxPort_setPinLow()*** function is declared in iLLD header ***IfxPort.h***.

Implementation

Interrupt Service Routine (ISR)

An ISR is triggered by the successful CAN message reception.

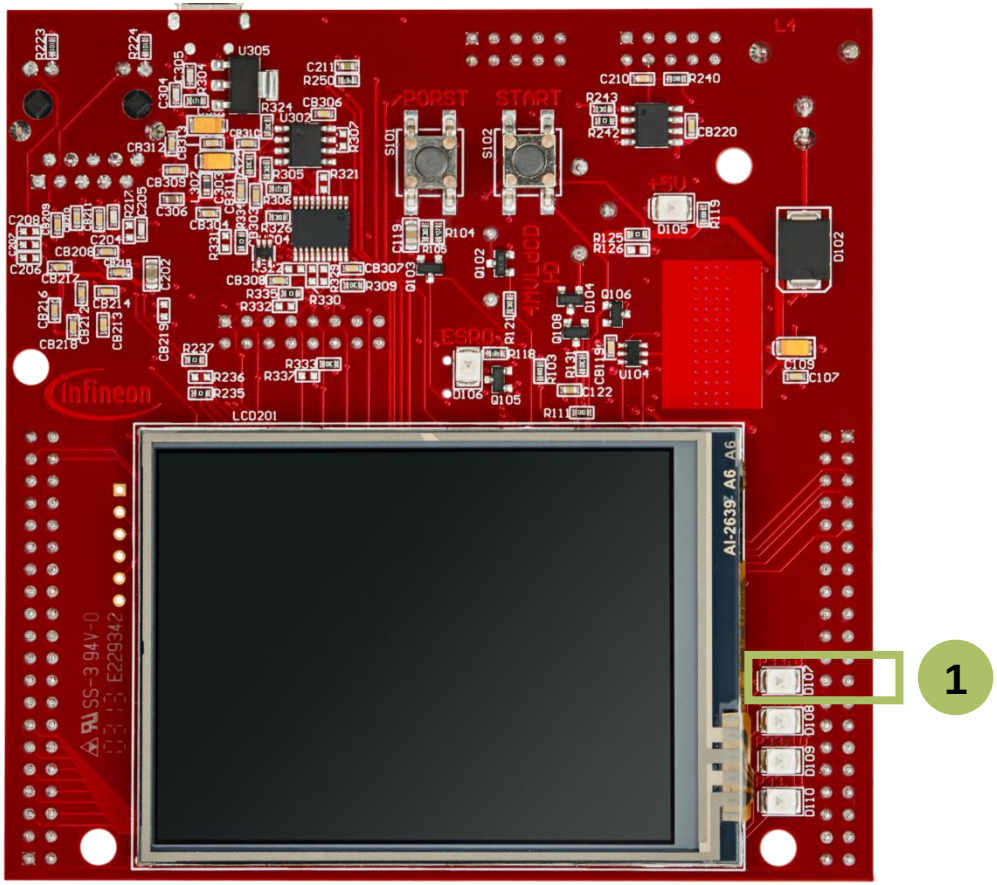
- The RX ISR reads the received CAN message with the ***IfxMultican_Can_MsgObj_readMessage()*** function. Based on the return status, the code execution can end up in an infinite loop due to the erroneous return status. If a non-erroneous return status is present, then a global variable ***g_isrRxCount*** is incremented. This variable is used as a counter to indicate the number of successfully received CAN messages.

The function is declared in the iLLD header ***IfxMultican_Can.h***.

Run and Test

After code compilation and flashing the device, observe the following behavior:

- Check that the LED (1) is turned on (correct CAN messages' content has been received, valid CUR pointers in both gateway source and destination message object have been observed)



References

› AURIX™ Development Studio is available online:
› https://www.infineon.com/aurixdevelopmentstudio
› Use the *„Import...“* function to get access to more code examples.

› More code examples can be found on the GIT repository:
› https://github.com/Infineon/AURIX_code_examples

› For additional trainings, visit our webpage:
› https://www.infineon.com/aurix-expert-training

› For questions and support, use the AURIX™ Forum:
› https://www.infineonforums.com/forums/13-Aurix-Forum

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2020-02
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Email: erratum@infineon.com

Document reference
MULTICAN_GW_TX_FIFO_1_KIT_TC297_TFT

IMPORTANT NOTICE
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”) .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.