

# FreescalE MQX RTOS Example Guide

## CAN example

This document explains the CAN example, what to expect from the example and a brief introduction to the API used.

Currently there are two different CAN examples. These two examples are based on two different CAN drivers.

For boards `twrk70f120m` and `twrk60n512` the new `fsl_flexcan` driver is used whereas for other boards including `twrk20d72m`, `twrk21f120m`, `twrk40d100m`, `twrk40x256`, `twrk60d100m`, `twrk60f120m`, `twrmcf52259`, `twrmcf54418`, `twrpxd10`, `twrpxn20`, `twrpxs30` the old `kflexcan` driver is used. This document includes two sections. The first section explains the CAN example with old `kflexcan` driver. The second section gives explanation of the CAN example with new `fsl_flexcan` driver.

## SECTION 1: kflexcan driver

### The example

The example uses the CAN driver in the MQX RTOS to exchange data between two MCU boards using the FlexCAN module inside the MCU. The `kflexcan` driver is used. User can verify the data communication using the CAN protocol in the example by comparing the transmit data displayed in one terminal and the received data displayed in the other terminal.

### Running the example

To run the example the corresponding IDE, compiler, debugger and a terminal program are needed.

For this example application the CAN system consists of two nodes connected through the 2-wire CAN bus.

The external CAN transceiver is used to convert CAN signal from CAN module of MCU into the CAN signal transmitted over the CAN bus. As a result for each node in a CAN system user needs one MCU board and one board containing the CAN transceiver connected together. The default macro setting of MQX is sufficient to run this example application.

### Explaining the example

The example application creates three tasks `Main_Task`, `Tx_Task` and `Rx_Task`. The brief description of the tasks is following.

- `Main_Task` firstly calls function `_int_install_unexpected_isr()` to display all source of exceptions without installed handler to the terminal. After that it configures the input output interface by setting appropriate IO pins for CAN module. The CAN module is reset as function `FLEXCAN_Softreset()` from CAN driver is invoked which put the FlexCAN module in Freeze state. The program calls `FLEXCAN_Initialize()` to sets up the clock source, the desired baudrate and to refresh the mailboxes and mask registers for CAN module. The module is then set up to run in normal mode with global mask register for matching process. Different interrupt handlers are

installed to handle common errors with CAN operation by functions FLEXCAN\_Install\_isr\_err\_int(), FLEXCAN\_Install\_isr\_boff\_int(), FLEXCAN\_Error\_int\_enable(). An event is created to alert Tx\_Task and Rx\_Task about the status of data transmission and reception. Main\_Task then creates Tx\_Task and Rx\_Task and starts the FlexCan module before entering an endless loop to enabling the data exchanges using CAN protocol.

```
*****FLEXCAN TEST PROGRAM.*****
Message format: Standard (11 bit id)
Message buffer 0 used for Tx and Rx.
Interrupt Mode: Enabled
Operation Mode: TX and RX --> Normal
*****

selected frequency (Kbps) is: 125
Data length: 1
FLEXCAN reset. result: 0x0
FLEXCAN initialization. result: 0x0
FLEXCAN mode selected. result: 0x0
FLEXCAN global mask. result: 0x0
FLEXCAN Error ISR install. result: 0x0
FLEXCAN Bus off ISR install. result: 0x0
FLEXCAN error interrupt enable. result: 0x0
FLEXCAN rx remote mailbox initialization. result: 0x0
FLEXCAN rx mailbox initialization. result: 0x0
FLEXCAN mailbox activation. result: 0x0
FLEXCAN RX ISR install. result: 0x0
FLEXCAN RX remote ISR install. result: 0x0
FLEXCAN tx remote mailbox initialization. result: 0x0
FLEXCAN tx mailbox initialization. result: 0x0
FLEXCAN tx mailbox activation. result: 0x0
FLEXCAN TX ISR install. result: 0x0
FLEXCAN TX remote ISR install. result: 0x0
FLEXCAN started. result: 0x0
```

- The Tx\_Task initializes two different mailboxes for transmitting CAN message by calling function FLEXCAN\_Initialize\_mailbox() in which each mailbox is assigned an ID with corresponding type of ID number. The ID number is used in the transmitting CAN frame. After initialization the mailboxes are in inactive state. Therefore they are activated by function FLEXCAN\_Activate\_mailbox(). Tx\_Task then installs the interrupt handler to signify it when the transmission of CAN message is done. To do this the function FLEXCAN\_Install\_isr() is used with the input parameter including the mailbox used and interrupt handler. From this point onwards, Tx\_Task resides in an endless loop of transmitting data message by calling FLEXCAN\_Tx\_mailbox() function and responding to remote frame request from another node in the CAN system by calling FLEXCAN\_Update\_message(). The CAN messages are transmitted every 1 second.
- The Rx\_Task initializes two different mailboxes for receiving CAN message by calling function FLEXCAN\_Initialize\_mailbox() in which each mailbox is assigned an ID with corresponding type of ID number. The ID number is used in the matching process in which CAN message with the same ID is received into the mailbox. After initialization

the mailboxes are in inactive state. Therefore they are activated by function `FLEXCAN_Activate_mailbox()`. Similar to `Tx_Task()`, `Rx_Task` calls `FLEXCAN_Install_isr()` to install interrupt handler for signaling the `Rx_Task` about the availability of CAN message in the receive mailbox.

The reception process is implemented as follow:

- o wait for the event bit set when the receive mailbox has any message.
- o lock the receive mailbox to make sure the received CAN message is not corrupted during the reading. `FLEXCAN_Lock_mailbox()` is used.
- o read the received message by calling function `FLEXCAN_Rx_message()`.
- o unlock the receive mailbox to enabling reception of new CAN message. `FLEXCAN_Unlock_mailbox()` is used.
- o transmit a remote request frame using `FLEXCAN_Request_message()` function.

The `Rx_Task` resides in an endless loop of reception of CAN message.

```
Data transmit: 1
FLEXCAN tx update message. result: 0x0
Received data: 0x6
ID is: 0x321
DLC is: 0x1

Data transmit: 2
FLEXCAN tx update message. result: 0x0
Received data: 0x7
ID is: 0x321
DLC is: 0x1

Data transmit: 3
FLEXCAN tx update message. result: 0x0
Received data: 0x8
ID is: 0x321
DLC is: 0x1

Data transmit: 4
FLEXCAN tx update message. result: 0x0
Received data: 0x9
ID is: 0x321
DLC is: 0x1

Data transmit: 5
FLEXCAN tx update message. result: 0x0
Received data: 0xa
ID is: 0x321
DLC is: 0x1

Data transmit: 6
FLEXCAN tx update message. result: 0x0
Received data: 0xb
ID is: 0x321
DLC is: 0x1
```

## **SECTION 2: fsl\_flexcan driver**

### **The example**

The example shows the data transmission between 2 MCUs using the CAN protocol. Each MCU is a node in the CAN system. The CAN bus consists of 2 wires CAN\_H and CAN\_L which are the can transceiver data lines. The CAN message data and the ID number of the received CAN message exchanged are displayed on terminal.

### **Running the example**

To run the example the corresponding IDE, compiler, debugger and a terminal program are needed.

The external CAN transceiver is used to convert CAN signal from CAN module of MCU into the CAN signal transmitted over the CAN bus. As a result for each node in a CAN system user needs one MCU board and one board containing the CAN transceiver connected together. The default macro setting of MQX is sufficient to run this example application.

### **Explaining the example**

The example application creates three tasks Main\_Task, Tx\_Task and Rx\_Task. The brief description of the tasks is following.

- Main\_Task configures the MCU as a node in CAN system including initialization of the CAN module, setting the desired baudrate, configuring specific mailboxes of the CAN module for transmission and reception of CAN messages. Because the CAN reception process uses the masking method to filter out the message from unwanted node in the CAN system the Main\_Task chooses the masking type and sets the masking criteria for the CAN module. In this example the reception using FIFO feature of CAN module is disabled. The mailbox is used for reception instead. Main\_Task creates Rx\_Task and Tx\_Task before entering an endless loop letting these newly created tasks to run.
- Tx\_Task calls flexcan\_tx\_mb\_config() function in the CAN driver to configure one mailbox as a transmit mailbox. After this it enters an endless loop in which it transmits CAN message using function flexcan\_send() from the CAN driver.
- Rx\_Task sets up one mailbox to receive CAN message. This includes setting the ID of CAN message that this mailbox accepts and the type of ID number this mailbox uses - standard ID or extended ID. Rx\_Task then enters an endless loop of reception of CAN message using function flexcan\_start\_receive() in the CAN driver. The attributes of the CAN message is displayed over the terminal.

In this example application the Tx\_Task and Rx\_Task are blocked waiting for transmission and reception to finish. The interrupt handler in the CAN driver is used to reschedule these two tasks as the transmission or the reception of CAN message is done.

The following picture shows one possible output of the example.

```
*****FLEXCAN TEST PROGRAM.*****  
Message format: Standard (11 bit id)  
Message buffer 8 used for Rx.  
Message buffer 9 used for Tx.  
Interrupt Mode: Enabled  
Operation Mode: TX and RX --> Normal  
*****
```

```
FLEXCAN get bitrate: 1000000 Hz  
FlexCAN receive config  
FlexCAN send config  
DLC=1, mb_idx=8  
RX MB data: 0x01  
ID: 0x123  
Data transmit: 0x01  
DLC=1, mb_idx=8  
RX MB data: 0x02  
ID: 0x123  
Data transmit: 0x02  
DLC=1, mb_idx=8  
RX MB data: 0x03  
ID: 0x123  
Data transmit: 0x03  
DLC=1, mb_idx=8  
RX MB data: 0x04  
ID: 0x123  
Data transmit: 0x04  
DLC=1, mb_idx=8  
RX MB data: 0x05  
ID: 0x123  
Data transmit: 0x05  
DLC=1, mb_idx=8  
RX MB data: 0x06  
ID: 0x123  
Data transmit: 0x06  
DLC=1, mb_idx=8  
RX MB data: 0x07  
ID: 0x123
```