

# Getting Started with the MCAN (CAN FD) Module on MSPM0 MCUs



Hao Mengzhen

## ABSTRACT

The Modular Controller Area Network (MCAN) peripheral is a CAN Flexible Data-rate (CAN FD) implementation on select devices within the MSPM0™ real-time microcontroller (MCU) family. Some devices that have an MCAN peripheral include the MSPM0G350x, MSPM0G351x devices. The examples described are to be run in any MSPM0 MCU with an MCAN module. This application note describes several programming examples to illustrate how the MCAN module is set up for different modes of operation. The objective is to help users understand programming the MCAN peripheral.

The code examples were tested on MSPM0G3507. However, the examples can be easily adapted to run on any MSPM0 device that features the MCAN module. Most of the examples need a second CAN FD node for operation. This requirement can be met by another MCU featuring CAN FD or any CAN bus analysis tool that is capable of working with both classic CAN and CAN FD protocols. Many USB-bus based tools are currently available. In addition to providing visibility to the bus traffic, these tools are also capable of generating frames and are an invaluable aid in debugging. An oscilloscope with built-in CAN FD triggering or decoding is essential for debugging.

Throughout this document, the terms MCAN and CAN FD are used interchangeably. While *CAN FD* refers to the protocol, *MCAN* refers to the peripheral in the MCU that implements the protocol. The project files for examples described in this document are available for download as part of [MSPM0-SDK](#).

## Table of Contents

<b>1 Introduction</b> .....	3
1.1 MCAN Features.....	4
<b>2 Sysconfig Configuration for MCAN Module</b> .....	5
2.1 MCAN Clock Frequency.....	5
2.2 MCAN Basic Configuration.....	5
2.3 Advanced Configuration.....	12
2.4 Retention Configuration.....	13
2.5 Interrupts.....	13
2.6 Pin Configuration and PinMux.....	15
<b>3 Demo Project Descriptions</b> .....	16
3.1 TX Buffer Mode.....	17
3.2 TX FIFO Mode.....	18
3.3 RX Buffer Mode.....	19
3.4 RX FIFO Mode.....	20
<b>4 Debug and Design Tips to Resolve/Avoid CAN Communication Issues</b> .....	21
4.1 Minimum Number of Nodes Required.....	21
4.2 Why a Transceiver is Needed.....	21
4.3 Bus Off Status.....	21
4.4 Using MCAN in Low Power Mode.....	23
4.5 Debug Checklist.....	23
<b>5 Summary</b> .....	24
<b>6 References</b> .....	24

## List of Figures

Figure 1-1. Typical CAN Bus.....	3
Figure 1-2. CAN FD Frame.....	3
Figure 2-1. MCAN Clock Frequency.....	5
Figure 2-2. MCAN Basic Configuration.....	5
Figure 2-3. Transmitter Delay Compensation (TDC).....	6
Figure 2-4. Bit Timing Parameters.....	7
Figure 2-5. Standard and Extended ID Filter Configuration.....	8
Figure 2-6. TX MSG RAM.....	10
Figure 2-7. RX MSG RAM.....	11
Figure 2-8. Advanced Configuration.....	12
Figure 2-9. Retention Configuration.....	13
Figure 2-10. Interrupts.....	13
Figure 2-11. MCAN Integration.....	14
Figure 2-12. Pin Configuration and PinMux.....	15
Figure 3-1. MCAN Loopback Message.....	16
Figure 3-2. Output of Bus-Monitoring Tool for mcan_multi_message_tx.....	16
Figure 3-3. Output of Bus-Monitoring Tool for mcan_multi_message_tx_tcan114x.....	17
Figure 3-4. Output of Bus-Monitoring Tool for mcan_single_message_tx.....	17

### Trademarks

MSPM0™, Code Composer Studio™, LaunchPad™, and BoosterPack™ are trademarks of Texas Instruments. All trademarks are the property of their respective owners.

# 1 Introduction

CAN is a serial protocol that was originally developed for automotive applications. Due to robustness and reliability, CAN is available in diverse applications such as industrial equipment, medical electronics, trains, aircraft, and so forth. CAN protocol features sophisticated error detection and confinement mechanisms and is capable of simple wiring at the physical level. The original CAN protocol standard is now referred to as *classical* CAN to distinguish from the more recent CAN FD standard. [Figure 1-1](#) shows the typical wiring for CAN bus.

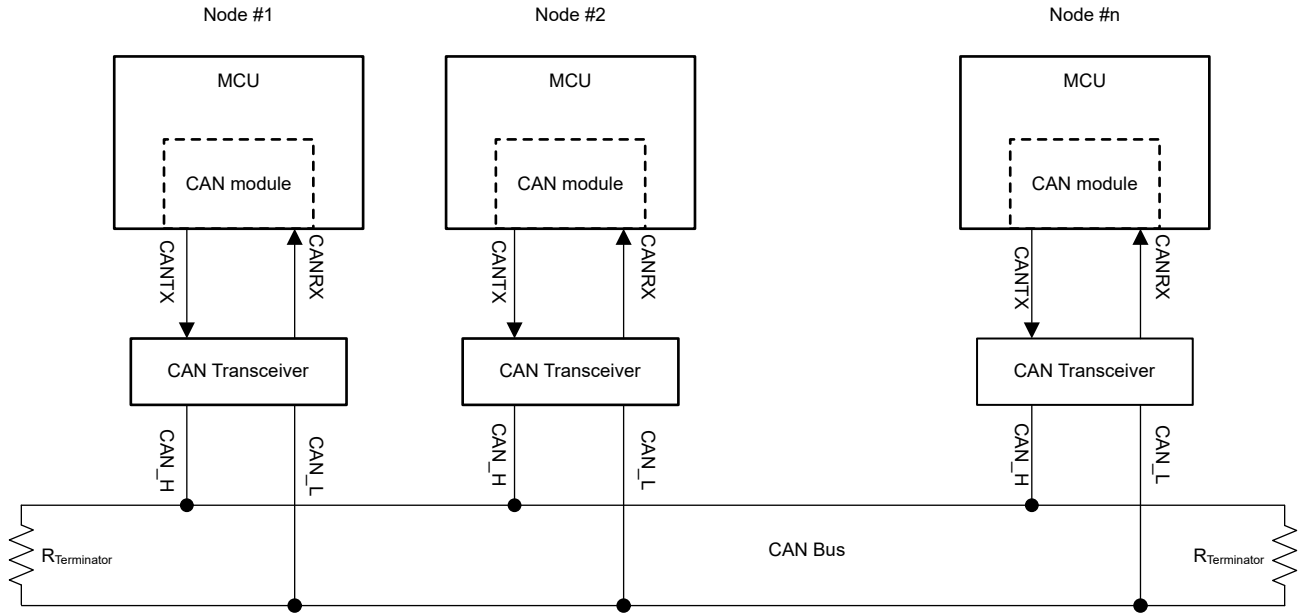


Figure 1-1. Typical CAN Bus

CAN Flexible Data Rate (CAN FD) is an enhancement to the classical CAN in terms of higher bit rates and the number of bytes transferred in one frame, thus increasing the effective throughput of communication. While classical CAN supports bit rates up to 1Mbps and a payload size of 8 bytes per frame, CAN FD supports bit-rates up to 5Mbps and a payload size of up to 64 bytes per frame. [Figure 1-2](#) shows the frame structure for CAN FD frames.

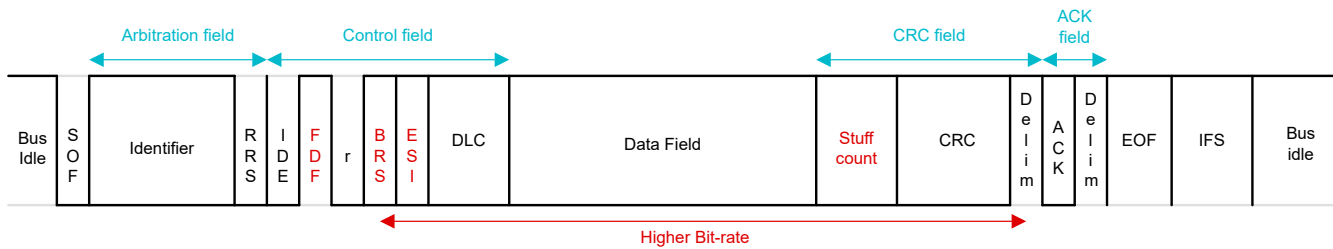


Figure 1-2. CAN FD Frame

## 1.1 MCAN Features

Salient features of the MCAN module are as follows:

- Conforms with CAN Protocol 2.0 A, B and ISO 11898-1:2015
- Full CAN FD support (up to 64 data bytes)
- AUTOSAR and SAE J1939 support
- Up to 32 dedicated transmit buffers
- Configurable transmit FIFO up to 32 elements
- Configurable transmit queue up to 32 elements
- Configurable transmit Event FIFO up to 32 elements
- Up to 14 dedicated receive buffers
- Two configurable receive FIFOs up to 14 elements each with 1kB message RAM
- Up to 128 filter elements
- Loop-back mode for self-test
- Maskable interrupt (two configurable interrupt lines, correctable ECC, counter overflow, and clock stop or wakeup)
- Non-maskable interrupt (uncorrectable ECC)
- Two clock domains (CAN clock and host clock)
- ECC check for Message RAM
- Clock stop and wakeup support
- Timestamp counter
- 1Mbps nominal bit rate; 8Mbps data bit rate

Non-supported features:

- Host bus firewall
- Clock calibration
- Debug over CAN

## 2 Sysconfig Configuration for MCAN Module

SysConfig is a configuration tool designed to simplify hardware and software configuration challenges to accelerate software development. SysConfig is available as part of the Code Composer Studio™ integrated development environment and is a standalone application. Additionally, SysConfig can be run in the cloud by visiting the [TI developer zone](#). SysConfig provides an intuitive graphical user interface for configuring pins, peripherals, radios, software stacks, RTOS, clock tree and other components. SysConfig automatically detects, exposes and resolves conflicts to speed software development.

TI recommends users to use Sysconfig to configure the MCAN module in applications. Sysconfig helps users quickly set up the MCAN module as desired with little effort. The next sections are used to introduce how to use Sysconfig to configure MCAN with the example `mcan_loopback_LP_MSPM0G3507` inside MSPM0-SDK.

### 2.1 MCAN Clock Frequency

Figure 2-1 shows what parameters are included in MCAN Clock Frequency block.

MCAN Clock Frequency	
CAN_CLK Frequency	40.00 MHz
CAN_CLK Frequency Divider	Divide by 1
MCAN Frequency	40.00 MHz

Figure 2-1. MCAN Clock Frequency

Peripheral asynchronous clock (CAN\_CLK) for MCAN can be clocked either through HFXT or SYSPLL. This clock configuration has to be done through the SYSCTL section. TI recommends for the MCAN frequency is to be configured as 40MHz.

#### Note

TI recommends to use HFXT as the clock source for CAN\_CLK. The accuracy of SYSOSC does not meet the requirements of applications in special scenarios and cause higher error frames. The recommendation frequency of the CAN\_CLK is 20MHz, 40MHz and 80MHz.

### 2.2 MCAN Basic Configuration

Figure 2-2 shows what parameters are included in MCAN Basic Frequency block.

MCAN Basic Configuration	
Enable CAN FD Mode	<input checked="" type="checkbox"/>
Enable Bit Rate Switching	<input checked="" type="checkbox"/>
Enable Loopback Mode	Enabled and internal
Enable Transmit Pause	<input type="checkbox"/>
Enable Edge Filtering	<input type="checkbox"/>
Enable Protocol Exception Handling	<input type="checkbox"/>
Messages Will Only Be Sent Once	<input type="checkbox"/>
Enable Wakeup Request	<input checked="" type="checkbox"/>
Enable Auto-Wakeup	<input checked="" type="checkbox"/>
Enable Emulation/Debug Suspend	<input checked="" type="checkbox"/>
Transmitter Delay Compensation (TDC)	
Message RAM Watchdog Preload Value	255
Bit Timing Parameters	
Message RAM Configuration	

Figure 2-2. MCAN Basic Configuration

- Enable CAN FD Mode: where CAN flexible data mode needs to be enabled.
- Enable Bit Rate Switching: by enabling this feature, MCAN sends data at higher rate instead of the arbitration rate. This function only works when CAN FD mode is enabled.
- Enable Loopback Mode: transmitted messages become received messages. This allows users to monitor the CAN messages on the CAN\_TX pin without a CAN transceiver.
- Enable Transmit Pause: pauses for two CAN bit times before the next transmission. The transmit pause feature is intended for use in CAN networks where the CAN Message IDs are specific and cannot easily be changed. These Message IDs can have a higher priority than other defined Message IDs, while in a specific application, the relative priority is inverse. This allows for when one ECU sends a burst of CAN messages that causes CAN messages from another ECU to be delayed (paused).
- Enable Edge Filtering: two consecutive dominant time quanta required to detect an edge for hard synchronization. Enabling this function is to make sure that nodes can accurately detect signal edges and perform hard synchronization under the condition of longer data bit times, thereby improving the stability and reliability of CAN bus communication.
- Enable Protocol Exception Handling: detection of bits that are reserved for future protocol expansion.
- Messages Will Only Be Sent Once: if automatic retransmission is disabled, then the MCAN module no longer retransmits when there is a transmission error, NACK, or the MCAN module loses arbitration.
- Enable Wakeup Request: enables the MCAN module to wakeup on CAN RXD activity.
- Enable Auto-Wakeup: enables the MCAN module to automatically clear the MCAN CCCR.INIT bit, fully waking the MCAN up on an enabled wakeup request. Issuing a clock stop request puts the MCAN module into powerdown mode (sleep mode). During transition from IDLE to ACTIVE, if the enable wakeup request and enable auto-wakeup functions are enabled, then a read-modify-write is issued to clear the MCAN\_CCCR.INIT bit. The MCAN core resumes operation after the MCAN Core responds to the removal of the clock stop request with removing the clock stop acknowledge.

#### Note

Note that after a clock stop request has been removed by the hardware, the first frame (wakeup frame) is not received. This is because after the clock stop is issued, there are no active clocks running into the IP. Therefore, after removing the clock stop request, the wakeup frame that enables the clock has to be retransmitted.

- Enable Emulation or Debug Suspend: MCAN module can be suspended for emulation or debug.
- Message RAM Watchdog Preload Value: the RAM Watchdog monitors the READY output of the Message RAM. A Message RAM access by the Generic Master Interface of the MCAN starts the Message RAM Watchdog Counter with the value configured. The counter is reloaded when the Message RAM signals successful completion by activating the READY output. In case there is no response from the Message RAM until the counter has counted down to zero, the counter stops and the interrupt flag MCAN\_IR.WDI is set. The RAM Watchdog Counter is clocked by the host (system) clock.

### 2.2.1 Transmitter Delay Compensation (TDC)

Figure 2-3 shows what parameters are included in Transmitter Delay Compensation (TDC) block.



Transmitter Delay Compensation (TDC)	
Enable TDC	<input checked="" type="checkbox"/>
TDC Filter Window Length (Cycles)	10
TDC Offset (Cycles)	6

**Figure 2-3. Transmitter Delay Compensation (TDC)**

TDC is a mechanism used to compensate for the delay caused by the loop delay of the transceiver in CAN FD systems. This delay can prevent nodes from performing meaningful bit error checks at the sample point during high-bit-rate data transmission. Specifically, TDC introduces a secondary sample point (Secondary Sample Point or SSP) in the data phase, where the transmitted bit is compared with the received bit after accounting for the delay. This makes sure that the bit errors are correctly detected and handled.

TDC is necessary when the bit rate is high, leading to short data bits and significant loop delays. These delays can cause the node to miss the correct sample point for bit error detection. TDC is only active during the data phase and does not affect the arbitration phase.

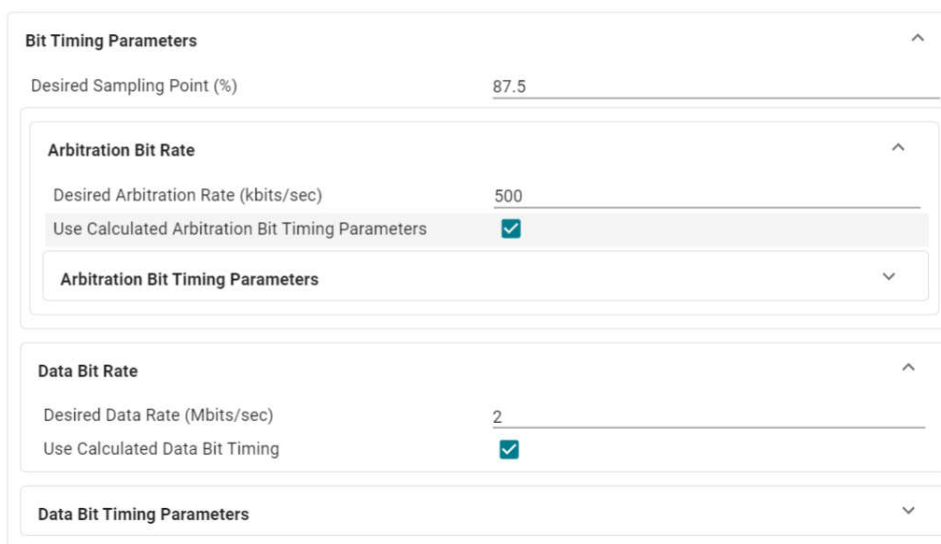
By using TDC, the data phase can have a shorter bit time than the nominal bit time, enabling higher data rates without compromising error detection.

- TDC Filter Window Length (Cycles): this filter feature defines a minimum value for the SSP position to avoid the case in which a dominant glitch inside the received FDF bit ends the delay compensation measurement before the falling edge of the received res bit, resulting in an early taken SSP position.
- TDC Offset (Cycles): this offset is used to adjust the position of the SSP inside the received bit (for example, half of the bit time in the data phase).

For more information about how to measure TDC, refer to the [MSPM0 G-Series 80MHz Microcontrollers technical reference manual](#).

### 2.2.2 Bit Timing Parameters

Figure 2-4 shows what parameters are included in Bit Timing Parameters block.



**Figure 2-4. Bit Timing Parameters**

Bit timing in the CAN bus refers to the critical parameters used for synchronization and control of data transmission in CAN communication. This divides the time of each bit into multiple time periods (called Time Quanta or TQ) to make sure that all nodes on the network can accurately receive and transmit data. The setting of bit timing includes several key components: Time Quantum (TQ), Sync Segment (SyncSeg), Propagation Segment (PropSeg), Phase Buffer Segment 1 (PBSeg1), Phase Buffer Segment 2 (PBSeg2), and Sample Point. The role of bit timing in data transmission is reflected in aspects such as synchronization and consistency, data integrity, fault tolerance, data rate and network performance, and anti-interference capability.

- Desired Sampling Point (%): for CAN FD, the required sampling point percentage is from 15% to 95%. This parameter always matches the other CAN nodes on the bus.
- Arbitration Bit Rate Configuration: desired arbitration rate (kbits/sec): Defines the arbitration rate.
- Use Calculated Arbitration Bit Timing Parameters: with this configuration enabled, Sysconfig automatically calculates the arbitration baud rate pre-scaler, time before sample Pt, time after sample Pt and (Re) synch jump width range based on desired sampling point and arbitration rate.
- Data Bit Rate Configuration: desired data rate (kbits/sec): Defines the data rate.
- Use Calculated Data Bit Timing Parameters: with this configuration enabled, Sysconfig automatically calculates data baud rate pre-scaler, time before sample Pt, time after sample Pt and (Re) synch jump width range based on desired sampling point and data rate.

### 2.2.3 Message RAM Configuration

The MCAN module has a Message RAM. The main purpose of the Message RAM is to store:

1. Message ID filter elements
2. Transmit messages
3. Tx event elements
4. Received messages

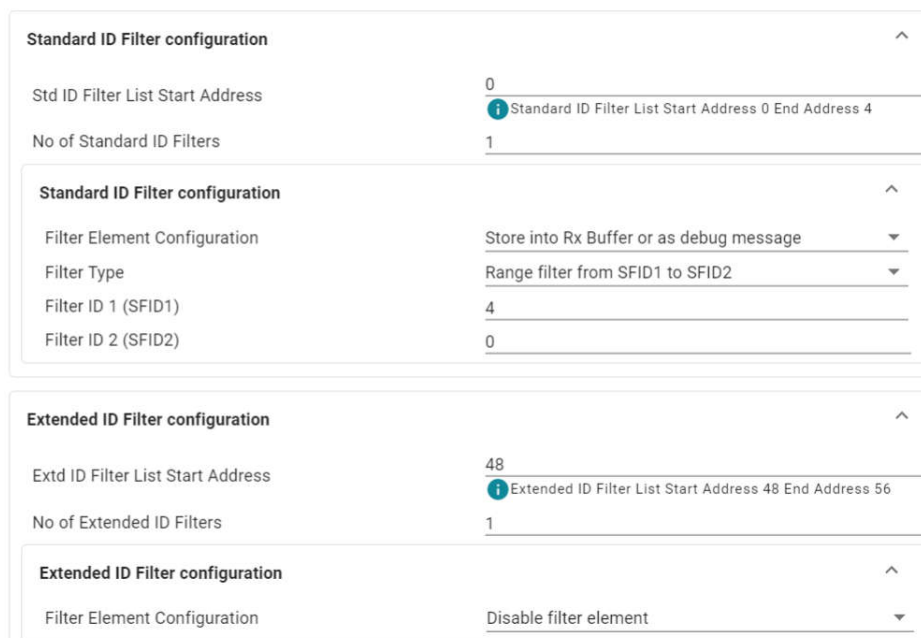
The Message RAM size is configured for 1kB size with a width of 32 bits is described in this application note.

#### Note

TI recommends to configure the Message RAM through Sysconfig. Sysconfig informs user of the address space occupied by the current configuration. This helps users to avoid an address overlapping issue.

#### 2.2.3.1 Standard and Extended ID Filter Configuration

Figure 2-5 shows what parameters are included in Standard and Extended ID Filter configuration block.



The screenshot shows the Sysconfig configuration interface for the MCAN module. It is divided into two main sections: Standard ID Filter configuration and Extended ID Filter configuration.

**Standard ID Filter configuration:**

- Std ID Filter List Start Address: 0 (Info: Standard ID Filter List Start Address 0 End Address 4)
- No of Standard ID Filters: 1

**Standard ID Filter configuration (Expanded):**

- Filter Element Configuration: Store into Rx Buffer or as debug message
- Filter Type: Range filter from SFID1 to SFID2
- Filter ID 1 (SFID1): 4
- Filter ID 2 (SFID2): 0

**Extended ID Filter configuration:**

- Extd ID Filter List Start Address: 48 (Info: Extended ID Filter List Start Address 48 End Address 56)
- No of Extended ID Filters: 1

**Extended ID Filter configuration (Expanded):**

- Filter Element Configuration: Disable filter element

**Figure 2-5. Standard and Extended ID Filter Configuration**

- Std ID Filter List Start Address: each standard ID filter takes 4 message RAM address.
- Number of Standard ID Filters: up to 128 filter elements can be configured for 11-bit standard IDs. SysConfig currently does not support configuration of more than one filter. More filters can be added in the user application, however make sure that enough RAM is allocated during initialization
- Standard ID Filter configuration → Filter Element Configuration: all enabled filter elements are used for acceptance filtering of standard frames. Acceptance filtering stops at the first matching enabled filter element or when the end of the filter list is reached. Options for this parameter are shown below.
  - 0x0: disable filter element
  - 0x1: store in Rx FIFO 0 if filter matches
  - 0x2: store in Rx FIFO 1 if filter matches
  - 0x3: reject ID if filter matches
  - 0x4: set priority if filter matches
  - 0x5: set priority and store in FIFO 0 if filter matches
  - 0x6: set priority and store in FIFO 1 if filter matches
  - 0x7: store into Rx Buffer, configuration of standard filter type ignored



- Standard ID Filter configuration → Filter Type: standard filter type configuration. Options for this parameter are shown below.
  - 0x0: range filter from SFID1 to SFID2 (SFID2 ≥ SFID1)
  - 0x1: dual ID filter for SFID1 or SFID2
  - 0x2: classic filter: SFID1 = filter; SFID2 = mask
  - 0x3: filter element disabled
- Standard ID Filter configuration → Filter ID 1 (SFID1): Standard Filter ID 1. When filtering for Rx buffers, this field defines the ID of a standard message to be stored. The received identifiers must match exactly, and no masking mechanism is used.
- Standard ID Filter configuration → Filter ID 2 (SFID2): Standard Filter ID 2. This ID has different definitions depending on the filter element configuration. If the filter element configuration is from 0x1 to 0x6, then SFID2 is the second ID of standard ID filter element. If the filter element configuration is 0x7, then SFID2 is a filter for Rx buffers.

The configuration for the Extended ID filter is shown below.

- Extd ID Filter List Start Address: each extended ID filter takes 8 message RAM addresses.
- Number of Extended ID Filters: up to 64 filter elements can be configured for 29-bit extended IDs. SysConfig currently does not support configuration of more than one filter. More filters can be added in the user application, however make sure that enough RAM is allocated during initialization.
- Extended ID Filter configuration → Filter Element Configuration: all enabled filter elements are used for acceptance filtering of extended frames. Acceptance filtering stops at the first matching enabled filter element or when the end of the filter list is reached.
  - 0x0: disable filter element
  - 0x1: store in Rx FIFO 0 if filter matches
  - 0x2: store in Rx FIFO 1 if filter matches
  - 0x3: reject ID if filter matches
  - 0x4: set priority if filter matches
  - 0x5: set priority and store in FIFO 0 if filter matches
  - 0x6: set priority and store in FIFO 1 if filter matches
  - 0x7: store into Rx Buffer or as debug message, configuration of extended filter type ignored
- Extended ID Filter configuration → Filter Type: extended filter type configuration. Options for this parameter are shown below.
  - 0x0: range filter from EFID1 to EFID2 (EFID2 ≥ EFID1)
  - 0x1: dual ID filter for EFID1 or EFID2
  - 0x2: classic filter: EFID1 = filter, EFID2 = mask
  - 0x3: range filter from EFID1 to EFID2 (EFID2 ≥ EFID1), Extended ID and Mask not applied
- Extended ID Filter configuration → Filter ID 1 (EFID1): extended Filter ID 1. First ID of the extended ID filter element. When filtering for Rx buffers, this field defines the ID of an extended message to be stored.
- Extended ID Filter configuration → Filter ID 2 (EFID2): Extended Filter ID 2. This ID has different definitions depending on the extended filter element configuration. If the extended filter element configuration is from 0x1 to 0x6, then EFID2 is the second ID of extended ID filter element. If the extended filter element configuration is 0x7, then EFID2 is a filter for Rx buffers.

#### **2.2.3.1.1 How to Add More Filters**

The Sysconfig does not currently support the configuration of more than one filter. More filters can be added in the user application, however, make sure that enough RAM is allocated during initialization. Remember to configure the number of filters in Sysconfig before configuring the filters in application code.

An example of how to add more filters in the application code is shown below. The start address is configured as 0x0 and the number of filters is configured as 2 in Sysconfig.

```

static const DL_MCAN_StdMsgIDFilterElement gMCAN0StdFiltelem_0 = {
    .sfec = 0x1,
    .sft = 0x0,
    .sfid1 = 3,
    .sfid2 = 4,
};

static const DL_MCAN_StdMsgIDFilterElement gMCAN0StdFiltelem_1 = {
    .sfec = 0x10,
    .sft = 0x0,
    .sfid1 = 13,
    .sfid2 = 14,
};
/* Configure Standard ID filter element */
DL_MCAN_addStdMsgIDFilter(MCAN0_INST, 0U, (DL_MCAN_StdMsgIDFilterElement *) &gMCAN0StdFiltelem_0);
DL_MCAN_addStdMsgIDFilter(MCAN0_INST, 1U, (DL_MCAN_StdMsgIDFilterElement *) &gMCAN0StdFiltelem_1);
    
```

The example below shows how to add more extended filters.

```

static const DL_MCAN_ExtMsgIDFilterElement gMCAN0ExtFiltelem_0 = {
    .efec = 0x1,
    .eft = 0x2,
    .efid1 = 0x3,
    .efid2 = 0x1FFFFFFF,
};
/* Configure Extended ID filter element */
DL_MCAN_addExtMsgIDFilter(MCAN0_INST, 0U, (DL_MCAN_ExtMsgIDFilterElement *) &gMCAN0ExtFiltelem_0);
    
```

### 2.2.3.2 TX MSG RAM

Figure 2-6 shows what parameters are included in TX MSG RAM block.

**TX MSG RAM** ^

TX Buffers Start Address	148 <small>TX Buffer Start Address 148 End Address 292</small>
Number of Dedicated Transmit Buffers	2
No of TX FIFO Elements	0
TX FIFO Operation Mode	TX FIFO operation <span style="float: right;">▼</span>
TX Buffer Element Size	64 byte data field <span style="float: right;">▼</span>
TX Event FIFO Start Address	164 <small>TX Event FIFO Start Address 164 End Address 168</small>
TX Event FIFO Size	2
TX Event FIFO Watermark INT Level	0

**Figure 2-6. TX MSG RAM**

The Tx buffers section can be configured to hold dedicated Tx buffers as well as a Tx FIFO and Tx Queue. When the Tx buffers section is shared by dedicated Tx buffers and a Tx FIFO and Tx Queue, the dedicated Tx buffers start at the beginning of the Tx buffers section followed by the buffers assigned to the Tx FIFO or Tx Queue. [Table 2-1](#) shows the differences between Tx buffer mode, Tx FIFO mode and Tx queue mode.

**Table 2-1. Differences Between Tx Mode**

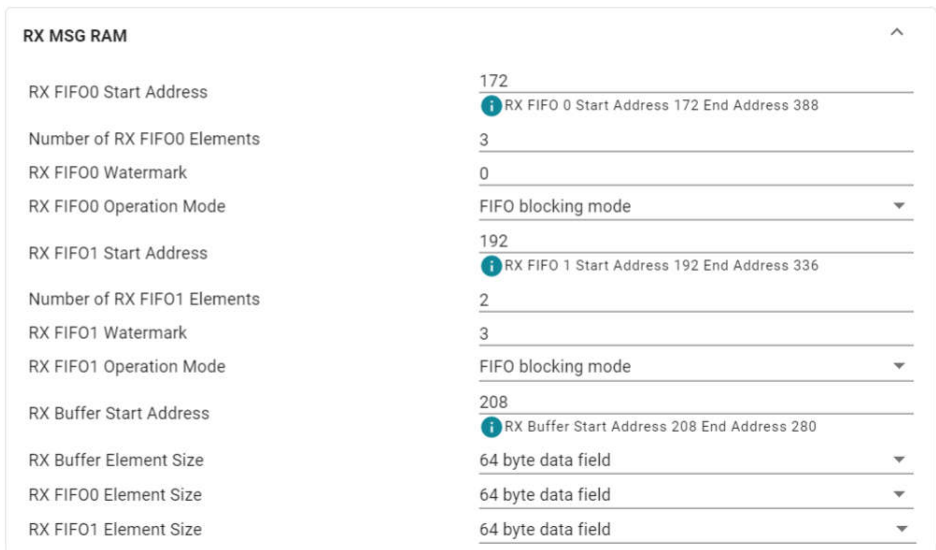
Tx Mode	Description
Tx buffer mode	Dedicated Tx buffers are intended for message transmission under complete control of the host CPU.
Tx FIFO mode	Tx FIFO allows transmission of messages with the same Message ID from different Tx buffers in the order these messages have been written to the Tx FIFO.
Tx queue mode	The stored in the Tx Queue messages are transmitted starting with the highest priority message (lowest Message ID).

- TX Buffers Start Address: the start address of Tx buffers in message RAM.
- Number of Dedicated Transmit Buffers: defines how many elements are configured as dedicated Tx buffers.
- No of TX FIFO Elements: defines how many elements are configured as Tx FIFO or Tx queue.
- TX FIFO Operation Mode: defines the Tx FIFO mode or Tx queue mode.

- TX Buffer Element Size: defines the Tx buffer data field size. In case the data length code DLC of a Tx Buffer element is configured to a value higher than the Tx Buffer data field size, the bytes not defined by the Tx Buffer are transmitted as 0xCC (padding bytes).
- TX Event FIFO Start Address: the Tx Event FIFO stores information about transmitted messages. To support Tx event handling, the Message RAM has implemented a Tx Event FIFO section. By reading the Tx Event FIFO, the Host CPU gets this information in the order the messages were transmitted. After message transmission on the CAN bus, Message ID and Timestamp are stored in a Tx Event FIFO element. To link a Tx Event to a Tx Event FIFO element, the Message Marker from the transmitted Tx Buffer is copied into the Tx Event FIFO element.
- TX Event FIFO Size: up to 32 Tx Event FIFO elements can be configured.
- TX Event FIFO Watermark INT Level: defines Tx Event FIFO fill level threshold. The Tx Event FIFO watermark can be configured to avoid a Tx Event FIFO overflow.

**2.2.3.3 RX MSG RAM**

Figure 2-7 shows what parameters are included in RX MSG RAM block.



**Figure 2-7. RX MSG RAM**

Up to 64 Rx buffers and two Rx FIFOs can be configured in the Message RAM. Each Rx FIFO section can be configured to store up to 64 received messages. The element size can be configured for storage of CAN FD messages with up to 64 bytes data field.

- RX FIFO0 and RX FIFO1 Start Address: defines the start address of Rx FIFOs in message RAM.
- Number of RX FIFO0 and RX FIFO1 Elements: each Rx FIFO can be configured to store up to 64 received messages.
- RX FIFO0 and RX FIFO1 Watermark: the Rx FIFO watermark can be used to prevent an Rx FIFO overflow. If the Rx FIFO fill level reaches the Rx FIFO watermark, then an interrupt flag MCAN\_IR.RF0W/ MCAN\_IR.RF1W is set.
- RX FIFO0 and RX FIFO1 Operation Mode:
  - Rx FIFO Blocking Mode: the Rx FIFO blocking mode is the default operation mode for the Rx FIFOs. If an Rx FIFO full condition is reached, then no further messages are written to the corresponding Rx FIFO until at least one message is read out and the Rx FIFO Get Index is incremented.
  - Rx FIFO Overwrite Mode: when an Rx FIFO full condition is reached, the next accepted message for the FIFO overwrites the oldest FIFO message.
- RX FIFO0 and RX FIFO1 Element Size: defines the Rx FIFO element size.
- RX Buffer Start Address: defines the start address of Rx buffer in message RAM.
- RX Buffer Element Size: defines the Rx buffer element size.

## 2.3 Advanced Configuration

Figure 2-8 shows what parameters are included in Advanced Configuration block.

Advanced Configuration	
Enable Additional Core Configuration	<input checked="" type="checkbox"/>
Enable Bus Monitoring Mode	<input type="checkbox"/>
Enable Normal CAN Operation	<input type="checkbox"/>
Time Stamp Prescaler Value	15
Timestamp Counter Value	Timestamp counter value always 0x0000
Time-out Counter Source Select	Continuous operation Mode
Start Value Of The Timeout Counter	65535
Enable Time-out Counter	<input type="checkbox"/>
Reject Remote Frames Extended	<input checked="" type="checkbox"/>
Reject Remote Frames Standard	<input checked="" type="checkbox"/>
Accept Non-matching Frames Extended	Accept in Rx FIFO 1
Accept Non-matching Frames Standard	Accept in Rx FIFO 1

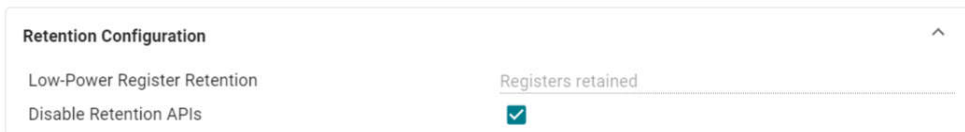
**Figure 2-8. Advanced Configuration**

- **Enable Additional Core Configuration:** enable or disable additional functions like timestamp and process nonmatching frames.
- **Enable Bus Monitoring Mode:** in bus monitoring mode (see ISO 11898-1:2015, Bus Monitoring section), the MCAN module is able to receive valid data and remote frames, but cannot start a transmission. The MCAN module sends only recessive bits on the CAN bus. If the MCAN module is required to send a dominant bit (ACK bit, overload flag, active error flag), then the bit is rerouted internally so that the MCAN module monitors this dominant bit. The CAN bus can remain in recessive state. The bus monitoring mode can be used to analyze the traffic on a CAN bus without affecting the bus by the transmission of dominant bits.
- **Enable Normal CAN Operation:** defines normal CAN operation mode to restricted operation mode. In restricted operation mode, the CAN node is able to receive data and remote frames and to give acknowledgment to valid frames, but the node does not send data frames, remote frames, active error frames, or overload frames. In case of an error condition or overload condition, the node does not send dominant bits; instead, the node waits for the occurrence of bus idle condition to resynchronize to the CAN communication. The restricted operation mode is automatically entered when the Tx Handler is not able to read data from the Message RAM in time. To leave restricted operation mode, the host CPU has to reset the MCAN\_CCCR.ASM bit. This mode can be used in applications that adapt themselves to different CAN bit rates. In this case, the application tests different bit rates and leaves the restricted operation mode after the node has received a valid frame.
- **Time Stamp Prescaler Value:** the MCAN module has integrated a 16-bit wrap-around counter for timestamp generation. The timestamp counter prescaler MCAN\_TSCC.TCP field can be configured to clock the counter in multiples of CAN bit times (1-16). On start of a frame reception or transmission, the counter value is captured and stored into the timestamp section of an Rx Buffer, Rx FIFO or Tx Event FIFO element.
- **Timestamp Counter Value:** configures the timestamp counter value to 0x0, from the internal 16-bit counter or from an external timestamp.
- **Time-out Counter Source Select:** the MCAN module has an integrated a 16-bit timeout counter. The timeout counter is used to signal timeout conditions for the Rx FIFO 0, Rx FIFO 1, and Tx Event FIFO Message RAM elements. In continuous mode, the counter is immediately restarted at the value configured by the MCAN\_TOCC.TOP field. In case the timeout counter is controlled by one of the FIFOs, an empty FIFO presets the counter to the value configured by the MCAN\_TOCC.TOP field. Down-counting is started when the first FIFO element is stored.
- **Start Value Of The Timeout Counter:** defines the timeout duration.
- **Enable Time-out Counter:** enables the timeout function.
- **Reject Remote Frames Extended:** filter or reject all remote frames with 29-bit extended IDs.
- **Reject Remote Frames Standard:** filter or reject all remote frames with 11-bit standard IDs.

- Accept Non-matching Frames Extended: defines how received messages with 29-bit IDs that do not match any element of the filter list are treated.
- Accept Non-matching Frames Standard: defines how received messages with 11-bit IDs that do not match any element of the filter list are treated.

## 2.4 Retention Configuration

Figure 2-9 shows what parameters are included in Retention Configuration block.

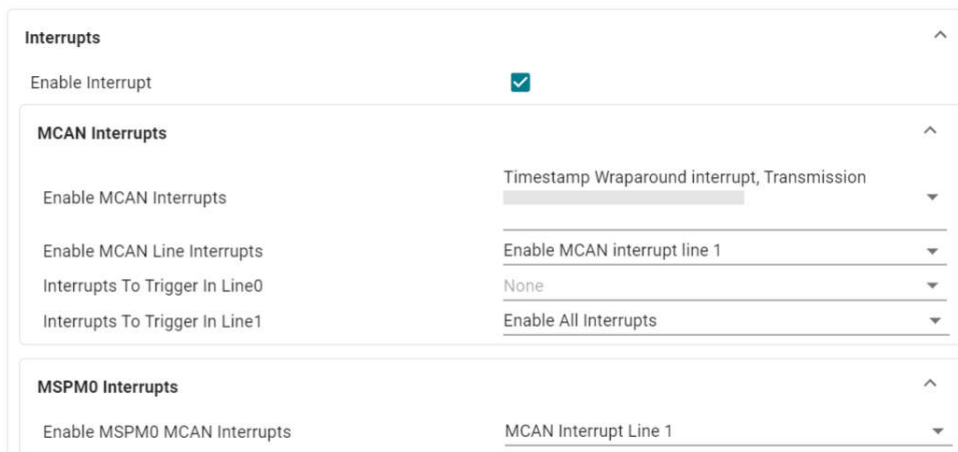


**Figure 2-9. Retention Configuration**

- Low-Power Register Retention: some MSPM0G peripherals residing in PD1 domain do not retain register contents when entering STOP or STANDBY modes. Developers can decide to reinitialize the peripheral using the default initialization from SysConfig in the application. This approach is more memory-efficient. Alternatively, the user can also use the provided DriverLib APIs to save and restore the register configuration of the peripheral before and after entering low-power mode. This approach is recommended if the peripheral configuration is modified at runtime.
- Disable Retention APIs: when selected, the retention APIs are not generated regardless of selected peripheral.

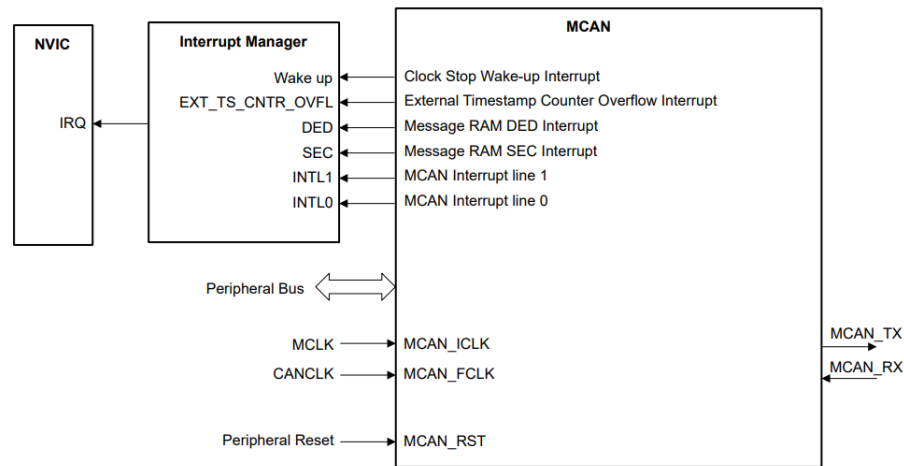
## 2.5 Interrupts

Figure 2-10 shows what parameters are included in Interrupts block.



**Figure 2-10. Interrupts**

The MCAN module contains one event publisher (CPU\_INT) that manages MCAN interrupt requests (IRQs) to the CPU subsystem by a static event route. Figure 2-11 shows the integration of the MCAN module in the device.

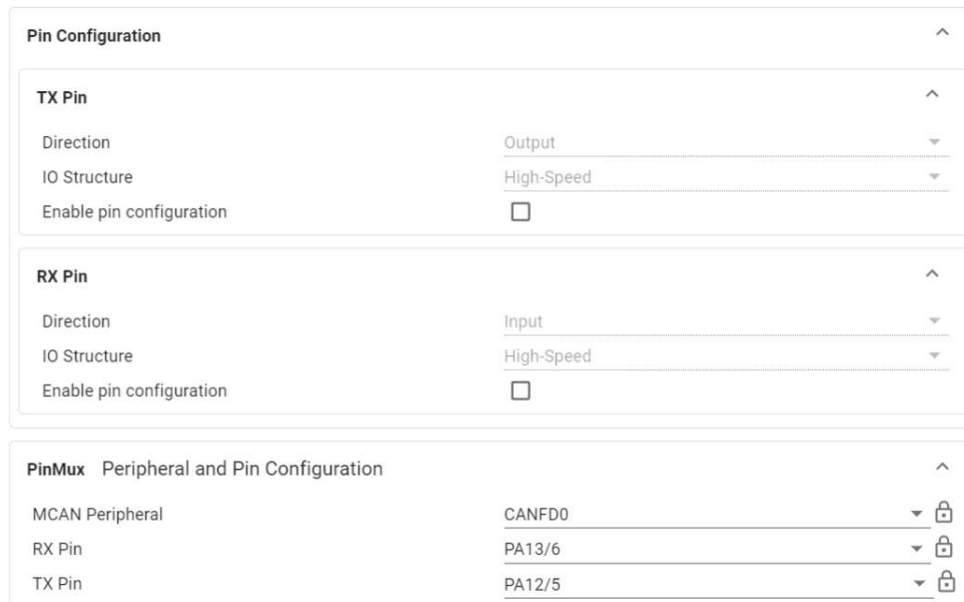


**Figure 2-11. MCAN Integration**

- MCAN Interrupts
  - Enable MCAN Interrupts: the MCAN core has two interrupt lines and 30 internal interrupt sources. Each source can be configured to drive one of the two interrupt lines. The MCAN core provides two interrupt requests Interrupt Line0 and Interrupt Line1.
  - Enable MCAN Line Interrupts: defines which interrupt lines are used in the application.
  - Interrupts To Trigger In Line0: defines which interrupt sources are assigned to Interrupt Line0.
  - Interrupts To Trigger In Line1: defines which interrupt sources are assigned to Interrupt Line1.
- MSPM0 Interrupts
  - Enable MSPM0 MCAN Interrupts: the MCAN module provides different interrupt sources which can be configured to source a CPU interrupt event. In order of decreasing interrupt priority, the CPU interrupt events from the MCAN are configured here.

## 2.6 Pin Configuration and PinMux

Figure 2-12 shows what parameters are included in Pin Configuration and PinMux block.



Pin Configuration	
<b>TX Pin</b>	
Direction	Output
IO Structure	High-Speed
Enable pin configuration	<input type="checkbox"/>
<b>RX Pin</b>	
Direction	Input
IO Structure	High-Speed
Enable pin configuration	<input type="checkbox"/>
<b>PinMux</b> Peripheral and Pin Configuration	
MCAN Peripheral	CANFD0
RX Pin	PA13/6
TX Pin	PA12/5

**Figure 2-12. Pin Configuration and PinMux**

- Pin Configuration
  - TX Pin and RX Pin: configures digital IOMUX features on a dedicated pin. Such as internal pull-up or pull-down resistor, invert output, drive strength and high impedance settings. TI recommends to keep default configuration on the IOMUC configurations for CAN applications.
- Pin Mux
  - MCAN Peripheral: some MSPM0 devices are integrated with multiple MCAN modules. Users can choose which MCAN module is configured here.
  - TX/RX Pin: configures the GPIO pins for MCAN TX and RX function.



### 3 Demo Project Descriptions

- mcan\_loopback*

This example illustrates the loopback functionality of the MCAN module. The loopback operation is completely internal to the module. However, the transmitted data is visible in the MCANTX pin. An advantage of this test case is that the transceiver is not needed, so the loopback operation can run on the LaunchPad™ boards. To facilitate easy analyzing of data on a logic analyzer, only four bytes of data are transmitted. However, data is transmitted as a CAN frame with bit-rate switching disabled.

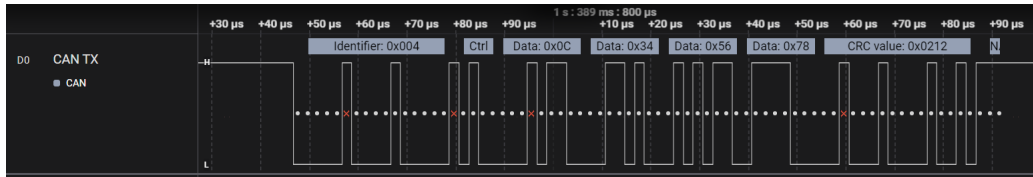


Figure 3-1. MCAN Loopback Message

- mcan\_multi\_message\_tx*

This example demonstrates the MCAN External Transmit function to send multiple messages. External communication is done between two CAN nodes. The receiving node can be another MCU or a CAN bus analysis tool capable of receiving or acknowledging the transmitted frames. Connect both CAN nodes through a CAN transceiver. This example can be used with the *mcan\_multi\_message\_rx* example project. A nominal bit rate of 250kbps and data bit rate of are used.

The TX messages are stored in CAN message RAM as buffer mode. Then, use a software call add request for the transmission API to transmit message in the desired TX buffer.

Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
0	0.000000	Devic...	0	0x3	StandardFrame	CANFD Accelerate	Rx	1	00
1	0.000200	Devic...	0	0x4	StandardFrame	CANFD Accelerate	Rx	1	00

Figure 3-2. Output of Bus-Monitoring Tool for *mcan\_multi\_message\_tx*

- mcan\_multi\_message\_rx*

This example demonstrates the MCAN receive function. The transmitting node can be another MCU or a CAN bus analysis tool capable of transmitting CAN FD frames. A nominal bit rate of 250kbps and data bit rate of 2Mbps are used. Only frames with a standard message ID of 0x3 and 0x4 are received. If another MCU with MCAN module is used as the transmitter, then *mcan\_multi\_message\_tx* example project can be run for the transmit function.

- mcan\_multi\_message\_tx\_tcan114x*

This example demonstrates the MCAN external transmit function to send multiple messages using the BOOSTXL-TCAN1145 BoosterPack™. External communication is done between two CAN nodes. The receiving node can be another MCU or a CAN bus analysis tool capable of receiving or acknowledging the transmitted frames. Connect both CAN nodes through a CAN transceiver. This example can be used with the *mcan\_multi\_message\_rx\_tcan114x* example project. A nominal bit rate of 250kbps and data bit rate of 2Mbps are used.

The software first initializes the TCAN114x module through SPI. Meanwhile, TX messages are stored in the CAN message RAM as buffer mode. Then, use a software call add request for the transmission API to transmit messages in the desired TX buffer.



Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
0	0.000000	Devic...	0	0x4	StandardFrame	CANFD Accelerate	Rx	1	00
1	2.122000	Devic...	0	0x3	StandardFrame	CANFD Accelerate	Rx	1	00

**Figure 3-3. Output of Bus-Monitoring Tool for mcan\_multi\_message\_tx\_tcan114x**

- *mcan\_multi\_message\_rx\_tcan114x*

This example demonstrates the MCAN receive function using a BOOSTXL-TCAN1145 BoosterPack. The transmitting node can be another MCU or a CAN bus analysis tool capable of transmitting CAN FD frames. A nominal bit rate of 250kbps and data bit rate of 2Mbps are used. Only frames with a standard message ID of 0x3 and 0x4 are received. If another MCU with MCAN module is used as the transmitter, then mcan\_multi\_message\_tx\_tcan114x example project can be run for the transmit function.

- *mcan\_single\_message\_tx*

This example demonstrates the MCAN external transmit function to send signal message. External communication is done between two CAN nodes. The receiving node can be another MCU or a CAN bus analysis tool capable of receiving or acknowledging the transmitted frames. Connect both CAN nodes through a CAN transceiver. This example can be used with the mcan\_multi\_message\_rx example project. A nominal bit rate of 250kbps and data bit rate of 2Mbps are used.

The TX message is stored in CAN message RAM as buffer mode. Then, use a software call add request for the transmission API to transmit messages in the TX buffer.

Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
0	0.000000	Devic...	0	0x4	StandardFrame	CANFD Accelerate	Rx	1	00

**Figure 3-4. Output of Bus-Monitoring Tool for mcan\_single\_message\_tx**

### 3.1 TX Buffer Mode

The TX is configured as buffer mode in the current demo projects in SDK. Below content discusses how to use TX in buffer mode.

```
DL_MCAN_TxBufElement txMsg;
/* Initialize message to transmit. */
/* Identifier Value. */
txMsg.id = ((uint32_t)(0x4)) << 18U;
/* Transmit data frame. */
txMsg.rtr = 0U;
/* 11-bit standard identifier. */
txMsg.xtd = 0U;
/* ESI bit in CAN FD format depends only on error passive flag. */
txMsg.esi = 0U;
/* Transmitting 4 bytes. */
txMsg.dlc = 1U;
/* CAN FD frames transmitted with bit rate switching. */
txMsg.brs = 1U;
/* Frame transmitted in CAN FD format. */
txMsg.fdf = 1U;
/* Store Tx events. */
txMsg.efc = 1U;
/* Message Marker. */
txMsg.mm = 0xA AU;
/* Data bytes. */
txMsg.data[0] = LED_STATUS_ON;

/* Write Tx Message to the Message RAM. */
DL_MCAN_writeMsgRam(MCAN0_INST, DL_MCAN_MEM_TYPE_BUF, 0U, &txMsg);

/* Add request for transmission. */
DL_MCAN_TXBufAddReq(MCAN0_INST, 0U);
```

First, the TX message is saved in to Message RAM in Buffer type with the buffer number by calling `DL_MCAN_writeMsgRam()`. Then, use the add request to transmit with the buffer number information of this message by calling `DL_MCAN_TXBufAddReq()`.

### 3.2 TX FIFO Mode

The TX is configured as buffer mode in the current demo projects in SDK. The section below describes how to use TX in FIFO mode.

```
uint8_t MCAN_send_frame(uint32_t id, uint8_t* data, uint16_t len)
{
    frame_send_success = false;
    DL_MCAN_TxBufElement txMsg;
    DL_MCAN_TxFIFOStatus txfifoStatus;

    /* Initialize message to transmit. */
    /* Identifier Value. */
    txMsg.id = id;
    /* Transmit data frame. */
    txMsg.rtr = 0U;
    /* 11-bit standard identifier. */
    txMsg.xtd = 0U;
    /* ESI bit in CAN FD format depends only on error passive flag. */
    txMsg.esi = 0U;
    /* Transmitting 4 bytes. */
    txMsg.dlc = encode_dlc(len);
    /* CAN FD frames transmitted with bit rate switching. */
    txMsg.brs = protocol_mode;
    /* Frame transmitted in CAN FD format. */
    txMsg.fdf = protocol_mode; //protocol_mode;
    /* Store Tx events. */
    txMsg.efc = 1U;
    /* Message Marker. */
    txMsg.mm = 0xAAU;
    /* Data bytes. */
    for (int i = 0; i < len; i++) txMsg.data[i] = data[i];
    while (DL_MCAN_OPERATION_MODE_NORMAL != DL_MCAN_getOpMode(CANFD0))
        ;

    /* Write Tx Message to the Message RAM (FIFO). */
    DL_MCAN_writeMsgRam(CANFD0, DL_MCAN_MEM_TYPE_FIFO, 0, &txMsg);

    /* Get put index and other TXFIFO details in txfifoStatus*/
    DL_MCAN_getTXFIFOQueueStatus(CANFD0, &txfifoStatus);

    /* Enable Transmission interrupt.*/
    DL_MCAN_TXBufTransIntrEnable(CANFD0, txfifoStatus.putIdx, 1U);

    /* Add request for transmission. */
    DL_MCAN_TXBufAddReq(CANFD0, txfifoStatus.putIdx);
    if (txMsg.id == MCAN_HOST_ID) {
        while (frame_send_success == false) {
            ;
        }
    }
    return 0;
}
```

First, the TX message is saved in to Message RAM in FIFO type by calling `DL_MCAN_writeMsgRam()`. Then, get the index of this message, which indicates the location of the saved message in FIFO, by calling `DL_MCAN_getTXFIFOQueueStatus()`. After, enable the transmission interrupt with the put index information of this message by calling `DL_MCAN_TXBufTransIntrEnable()`. Finally, call the add request for transmission with put index information of this message to transmit by calling `DL_MCAN_TXBufAddReq()`.

### 3.3 RX Buffer Mode

The RX is configured as FIFO mode in the current demo projects in SDK. The section below describes how to use RX in buffer mode.

```
static const DL_MCAN_StdMsgIDFilterElement gMCAN0StdFiltelem = {
    .sfec = 0x7,
    .sft = 0x0,
    .sfid1 = 3,
    .sfid2 = 0x0,
};
/* Configure Standard ID filter element */
DL_MCAN_addStdMsgIDFilter(MCAN0_INST, 0U, (DL_MCAN_StdMsgIDFilterElement *) &gMCAN0StdFiltelem);
```

The RX buffer mode is enabled when the filter element is configured as *Store into Rx buffer or as debug message*. First, add a filter with .sfec as 0x7. The .sft does not care when .sfec is set as 0x7. The .sfid1 is the filter ID, the receive message ID must match this ID to receive. The .sfid2 configures which Rx buffer is used to store receive messages.

```
/* New message received by Rx Buffer (Filter matched) */
if(gInterruptLineStatus & DL_MCAN_INTR_SRC_DEDICATED_RX_BUFF_MSG){
    gInterruptLine1Status |= DL_MCAN_INTR_SRC_DEDICATED_RX_BUFF_MSG;
    DL_MCAN_getNewDataStatus(MCAN0_INST, &newDataStatus);

    if(newDataStatus.statusLow) {
        /* Check Rx Buffer0 status */
        if(newDataStatus.statusLow & 0x1){
            DL_MCAN_readMsgRam(MCAN0_INST, DL_MCAN_MEM_TYPE_BUF, MCAN_RX_MSG_BUFFER_INDEX, 0U,
&rxMsg);
            ProcessCanRxMsg(&rxMsg);
        }
        /* Check remaining Rx Buffer */
    }
    if(newDataStatus.statusHigh) {
        ; /* 32-63, not applicable in demo */
    }
    /* Clear all new data status */
    DL_MCAN_clearNewDataStatus(MCAN0_INST, &newDataStatus);
}
```

When receiving a message which matches the filter configuration, the RX message is stored into the RX buffer from 0 to 63. First, get a new data status. If there are new messages received in RX buffer 0 to RX buffer 31, then the corresponding bit of the newDataStatus.statusLow is set to 1. If there are new messages received in RX buffer 32 to RX buffer 63, then the corresponding bit of the newDataStatus.statusHigh is set to 1. Then, check if there is new message received in RX buffer 0. If so, then read the RX message out by calling DL\_MCAN\_readMsgRam() with MCAN\_RX\_MSG\_BUFFER\_INDEX value. In this case, the index value of RX buffer 0 is 0. Finally, after processing the message, clear the new data status by calling DL\_MCAN\_clearNewDataStatus().

### 3.4 RX FIFO Mode

The RX is configured as FIFO mode in the current demo projects in SDK. The section below introduces the structure of `DL_MCAN_RxFIFOStatus`, which is commonly used in RX FIFO mode. TI recommends for the users to check this structure when pulling the message from RX FIFO.

```

/**
 * @brief Structure for MCAN Rx FIFO Status.
 */
typedef struct {
    /*! Rx FIFO number
     * One of @ref DL_MCAN_RX_FIFO_NUM
     */
    uint32_t num;
    /*! Rx FIFO Fill Level */
    uint32_t fillLvl;
    /*! Rx FIFO Get Index */
    uint32_t getIdx;
    /*! Rx FIFO Put Index */
    uint32_t putIdx;
    /*! Rx FIFO Full
     * 0 = Rx FIFO not full
     * 1 = Rx FIFO full
     */
    uint32_t fifoFull;
    /*! Rx FIFO Message Lost */
    uint32_t msgLost;
} DL_MCAN_RxFIFOStatus;

```

- The *num* indicates the Rx FIFO instance number of the current operation. MCAN can support multiple Rx FIFOs (such as Rx FIFO 0 and Rx FIFO 1), and the specific instance must be specified. For example, the `DL_MCAN_RX_FIFO_NUM` enumeration contains `DL_MCAN_RX_FIFO_0` and `DL_MCAN_RX_FIFO_1`.
- The *fillLvl* indicates the number of valid messages currently stored in the Rx FIFO. For example, when *fillLvl* is 5, this means there are 5 unread messages in the FIFO. When *fillLvl* reaches the depth of the Rx FIFO (such as 6 messages), the *fifoFull* field is set to 1, indicating that the FIFO is full.
- The *getIdx* points to the next message to be read. The CPU reads the messages in the FIFO sequentially by incrementing *getIdx*. When the Rx FIFO is full and in overwrite mode, new messages overwrite the oldest messages. At this time, user starts reading from *getIdx* + 1 to avoid reading old data that is being overwritten.
- The *putIdx* points to the next writable message location. When a new message is received, the message is written to the location pointed to by *putIdx*, and then *putIdx* is incremented.
- The *fifoFull* indicates the Rx FIFO is full or not. When Rx FIFO is full, try the following options:
  - Blocking mode: new messages are discarded and the *msgLost* count is triggered.
  - Overwrite mode: the new message overwrites the oldest message, *putIdx* and *getIdx* are incremented at the same time.
- The *msgLost* counts the number of messages discarded due to Rx FIFO full. This is triggered in the following situations:
  - Blocking mode: when the Rx FIFO is full, a new message is received and the message is rejected and discarded.
  - Overwrite mode: when overwriting old messages, the overwritten messages are counted as lost.

## 4 Debug and Design Tips to Resolve/Avoid CAN Communication Issues

This section illustrates some of the common mistakes and oversights while implementing a CAN bus. This is followed by some debugging tips useful to troubleshoot bus issues.

### 4.1 Minimum Number of Nodes Required

Unless working in self-test mode, a minimum of two nodes are needed on the CAN bus for the following reason. When a node transmits a frame on the CAN bus, the node expects an acknowledgment (ACK) from at least one other node on the network. Anytime a CAN node successfully receives a message, the node automatically transmits an ACK, unless that feature has been turned off (*silent mode*. Silent node is where a node receives the frame, but does not provide an ACK; the bus monitoring mode in MCAN). The node that provides the ACK does not need to be the intended recipient of the frame, although this can happen. (All active nodes on the bus provide an ACK, regardless of whether the nodes are the intended recipients of that frame).

When the transmitting node does not receive an ACK, this results in an ACK error and the transmitting node keeps retransmitting the frame forever. The Transmit Error Counter (TEC) increments to 128 and stops there. REC stays at 0. The node does not go bus-off. No interrupts are generated either. If another node is brought into the network, then TEC starts decrementing (all the way to 0) with every successful transmit.

### 4.2 Why a Transceiver is Needed

Users cannot directly connect MCAN\_TX of node-A to MCAN\_RX of node-B and vice versa and expect successful CAN communication. In this case, CAN is unlike other serial interfaces like UART or SPI. For example, UART can work with a RS232 transceiver or through a direct connection (UART\_TX of one node to UART\_RX of another node and vice versa). However, CAN bus needs a CAN transceiver for the following reason. In addition to converting the single-ended CAN signal for differential transmission, the transceiver also loops back the CAN\_TX pin to the CAN\_RX pin of a node. This is because a CAN node needs to be able to monitor the transmission.

- This has to do with the ACK requirement mandated by the CAN protocol. When a node transmits a frame on the CAN bus, the node expects an ACK from at least one other node on the network. For the ACK phase, the transmitter puts out a 1 and expects to read back a 0.
- During arbitration, a node with a higher-priority Message ID needs to be able to override a 1 with a 0. Here, the transmitter needs to be able to read back the transmitted data. When a node puts out a 1 and reads back a 0 during the arbitration phase, the node loses arbitration.

To save the cost of a transceiver, some applications (where all the nodes are on the same PCB and in close proximity) use discrete components like diodes to meet the requirement that a CAN node to monitor the transmission.

### 4.3 Bus Off Status

The Bus Off state occurs when a CAN node is forcibly isolated due to severe bus errors. In this state, the node stops transmitting and receiving data, effectively disconnecting from the bus. This mechanism protects the bus from persistent errors and prevents fault propagation.

The primary trigger is when the TEC exceeds a threshold (typically 255). Common scenarios include:

- Hardware failures: damaged CAN controllers or transceivers.
- Physical layer issues: open or short circuits, incorrect termination resistors, or signal interference (for example, strong EMI).
- Configuration errors: mismatched baud rates causing arbitration failures or bit errors.
- Software errors: repeated transmission of malformed frames.

For example, each transmission error increments the TEC by 1. If errors accumulate rapidly and TEC exceeds 255, then the node enters bus off state.

The device can recover from bus off state by performing one of the following actions:

- Hardware reset: reset the CAN controller (by software reset or power cycling) to clear error counters.
- Reception of recovery sequence: detect 129 consecutive sequences of 11 recessive bits (bus idle signals), which automatically restores the node to an *active error* state. This requires sufficient bus idle time (total duration = 129 × 11 bits).

An example is shown below of how to detect bus off status by on enabling Bus\_Off Status interrupt in Sysconfig first. When there is a change in the Bus Off status, the MCAN triggers a Bus\_Off Status interrupt to notify users of this situation. Users need to check if the bus off state is in the interrupt routine.

```

/**
 * CAN protocol status
 *   Bus off
 */
DL_MCAN_ProtocolStatus gProtStatus;
volatile uint8_t CANBusOff = 0;
void MCAN0_INST_IRQHandler(void)
{
    switch (DL_MCAN_getPendingInterrupt(MCAN0_INST)) {
        case DL_MCAN_IIDX_LINE0:
            break;
        case DL_MCAN_IIDX_LINE1:
            /* MCAN bus off status changed */
            if(gInterruptLine1Status&DL_MCAN_INTERRUPT_BO) {
                DL_MCAN_getProtocolStatus(MCAN0_INST, &gProtStatus);
                if(gProtStatus.busOffStatus == 1) {
                    CANBusOff = true;
                }
                else {
                    CANBusOff = false;
                }
            }
            /* Clear all MCAN interrupt status */
            DL_MCAN_clearIntrStatus(MCAN0_INST, gInterruptLine1Status,
DL_MCAN_INTR_SRC_MCAN_LINE_1);
            break;
        default:
            break;
    }
}

```

Then, check the CANBusOff flag in the main loop. When the bus off state is detected, restart the MCAN module.

```

main loop:
{
    if(CANBusOff == 1) {
        /* Re-start MCAN */
        MCAN0_Restart();
        CANBusOff = 0;
    }
}
/*MCAN restart function*/
void MCAN0_Restart(void)
{
    DL_MCAN_reset(CANFD0);
    delay_cycles(16);
    DL_MCAN_disablePower(CANFD0);
    delay_cycles(32);
    DL_MCAN_enablePower(CANFD0);
    // MCAN RAM need at least 50us to finish init
    // 1600 CPU cycles@CPU32MHZ
    // 4000 CPU cycles@CPU80MHZ
    delay_cycles(4000);
    SYSCFG_DL_MCAN0_init();
}

```

## 4.4 Using MCAN in Low Power Mode

Users need to put the MCU in low power mode to meet the application requirements. However, the MCAN module is disabled in low power mode. As a workaround, users can configure the MCAN RX pin as an input pin before entering low power mode. Enable a edge fail interrupt on that pin. When there is one message received on the MCAN RX pin, the MCU is woken up by the edge fail interrupt. Then, reconfigure the MCAN RX pin as an MCAN function and reconfigure the MCAN module. The MCAN restores normal functions this way.

---

### Note

The first MCAN message is used as a wake up signal for the MCU. Use a test message for this wake up function.

---

An example code is shown below.

```
void MCAN_LowPowerMode(void)
{
    DL_GPIO_initDigitalInputFeatures(GPIO_MCAN0_IOMUX_CAN_RX,
        DL_GPIO_INVERSION_DISABLE, DL_GPIO_RESISTOR_PULL_UP,
        DL_GPIO_HYSTERESIS_DISABLE, DL_GPIO_WAKEUP_DISABLE);
    DL_GPIO_setLowerPinsPolarity(GPIO_MCAN0_CAN_RX_PORT, DL_GPIO_PIN_13_EDGE_FALL);
    DL_GPIO_clearInterruptStatus(GPIO_MCAN0_CAN_RX_PORT, GPIO_MCAN0_CAN_RX_PIN);
    DL_GPIO_enableInterrupt(GPIO_MCAN0_CAN_RX_PORT, GPIO_MCAN0_CAN_RX_PIN);

    DL_SYSCTL_setPowerPolicySTANDBY0();
    __WFI();
    DL_SYSCTL_setPowerPolicyRUN0SLEEP0();

    DL_GPIO_initPeripheralInputFunction(
        GPIO_MCAN0_IOMUX_CAN_RX, GPIO_MCAN0_IOMUX_CAN_RX_FUNC);

    MCAN0_Restart();
}
/*MCAN restart function*/
void MCAN0_Restart(void)
{
    DL_MCAN_reset(CANFD0);
    delay_cycles(16);
    DL_MCAN_disablePower(CANFD0);
    delay_cycles(32);
    DL_MCAN_enablePower(CANFD0);
    // MCAN RAM need at least 50us to finish init
    // 1600 CPU cycles@CPU32MHZ
    // 4000 CPU cycles@CPU80MHZ
    delay_cycles(4000);
    SYSCFG_DL_MCAN0_init();
}
}
```

## 4.5 Debug Checklist

This section highlights some common mistakes and provides useful debug tips.

### 4.5.1 Programming Issues

- Is the clock to the MCAN module enabled? Check for CANCLK configuration in the Sysconfig. TI recommends to use an external crystal as the CANCLK source to get a low error rate.
- Try the code without interrupts first. Use polling instead. Once polling works, users can add interrupts later.
- When attempting to initiate communication on the bus for the very first time, make sure that the mailbox in the transmitting node and the receiving node are programmed with the same Message ID. Do not use filter function initially. Filtering can be added later once there are no hardware issues confirmed.



### 4.5.2 Physical Layer Issues

- Has the bus been terminated correctly (with 120Ω) at either ends only? The bus must be terminated only at either ends and with a 120Ω resistor. No more than two terminator resistors can be present on the bus, unless split termination is followed, in which there are two resistors on either ends. While designing a CAN bus system, the termination resistors can be enabled or disabled from outside the system enclosure. This scheme is easy for when nodes have to be added or removed to and from the network.
- Are all CAN nodes configured for the same bit-rate? Mismatched node bit rates repeatedly introduce error frames on the bus. Capture the output of the CAN\_TX pin on the oscilloscope to physically verify the bit-time.
- Have users tried a lower bit-rate? For example, 50kbps. Timing issues concerning propagation delays can be caught trying a lower bit-rate. Make sure that the bit timing parameters are configured correctly in Sysconfig.
- Have users tried to reduce the bus length and number of nodes?
- Before the error condition occurs, were any error-frames seen on the bus? This can be timing violations or noise issues.
- How many nodes are there in the bus? (In non-self-test mode, there must be at least two nodes on the network, due to the acknowledge (ACK) requirement mandated by the CAN protocol).

### 4.5.3 Hardware Debug Tips

- To see the waveform until the ACK phase, a transceiver must be connected to the node. Without a transceiver, the node immediately goes into an error state.
- Check if the CAN frame is correctly seen at the MCAN\_TX pin of the transmitting MCU and is of the expected bit-rate. If the expected data is seen at the MCAN\_TX pin, then check the data at the MCAN\_RX pin. If the same data is seen at the MCAN\_RX pin, then the transceiver is correctly looping back the data.
- If using an oscilloscope with a built-in CAN FD trigger, then make sure that the signal configured for triggering matches the signal being probed on the board. Many oscilloscopes are capable of triggering on CAN-transmit (CANTX), CAN-receive (CANRX), CAN\_H and CAN\_L signals, in addition to Start-of\_Frame (SOF), Remote frames, Error frames and specific Message IDs.
- If the scope does not decode the waveform, then make sure input threshold value for the channel is correct. This is similar to the *trigger level* that is normally used for signals.
- Make sure the bit-rate for both nominal and data phases are correctly configured in the oscilloscope. Otherwise, this shows incorrect data.
- CAN bus analyzer tool: make sure the bit-rate for both nominal and data phases are correctly configured.

## 5 Summary

This document begins with an overview of the CAN bus and the features of MCAN module integrated in the MSPM0 MCU. The application note details how to use Sysconfig to configure the CAN module, so that users can adjust related parameters to meet application requirements. This document briefly describes the demo example related to MCAN module provided in the MSPM0 SDK. Finally, some common debugging issues and relevant debugging suggestions from the software and hardware aspects are introduced.

## 6 References

- Texas Instruments, [MSPM0 G-Series 80MHz Microcontrollers](#), technical reference manual
- Texas Instruments, [MSPM0-SDK](#)
- Texas Instruments, [Introduction to the Controller Area Network \(CAN\)](#), application note
- Texas Instruments, [Controller Area Network Physical Layer Requirements](#), application note
- Texas Instruments, [Basics of debugging the controller area network \(CAN\) physical layer](#), analog design journal
- Texas Instruments, [Calculator for CAN Bit Timing Parameters](#), application note
- Texas Instruments, [Overview of 3.3V CAN \(Controller Area Network\) Transceivers](#), application note
- Texas Instruments, [Simplify CAN bus implementations with chokeless transceivers](#), marketing white paper
- Texas Instruments, [Critical Spacing of CAN Bus Connections](#), application note
- Texas Instruments, [Message priority inversion on a CAN bus](#), analog design journal



## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#) or other applicable terms available either on [ti.com](https://www.ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Copyright © 2025, Texas Instruments Incorporated