Application Note

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

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

1 CAN Introduction 3
1.1 CAN Standard 3
1.2 MSPM0 MCAN Module 5
2 Sysconfig Configuration for MCAN Module 6
2.1 MCAN Clock Frequency 6
2.2 MCAN Basic Configuration 6
2.3 Advanced Configuration 16
2.4 Retention Configuration 17
2.5 Interrupts 17
2.6 Pin Configuration and PinMux 19
3 SDK Example Code Introduction 20
3.1 mcan_loopback 20
3.2 mcan_message_rx 20
3.3 mcan_message_rx_tcan114x 20
3.4 mcan_multi_message_tx 20
3.5 mcan_multi_message_tx_tcan114x 21
3.6 mcan_single_message_tx 21
4 MCAN Use Case 22
4.1 TX Buffer Element 22
4.2 RX Buffer Element 23
4.3 TX Handling Buffer Mode 24
4.4 TX Handling FIFO Mode 24
4.5 RX Handling Buffer Mode 26
4.6 RX Handling FIFO Mode 28
4.7 Using MSPM0 MCAN in Low Power Mode 30
4.8 Bus Off Handling 31

Copyright © 2026 Texas Instruments Incorporated

5 Design Test and Debug Tips....33
5.1 Minimum Number of Nodes Required....33
5.2 Why a Transceiver is Needed....33
5.3 Debug Checklist....33
6 Summary....34
7 References....34
8 Revision History....35

List of Figures

Figure 1-1. Typical CAN Bus....3
Figure 1-2. CAN FD Frame....3
Figure 1-3. Partition of Bit Time....4
Figure 2-1. MCAN Clock Frequency....6
Figure 2-2. MCAN Basic Configuration....6
Figure 2-3. Transmitter Delay Compensation (TDC)....7
Figure 2-4. Bit Timing Parameters....8
Figure 2-5. Standard and Extended ID Filter Configuration....11
Figure 2-6. TX MSG RAM....14
Figure 2-7. RX MSG RAM....15
Figure 2-8. Advanced Configuration....16
Figure 2-9. Retention Configuration....17
Figure 2-10. MCAN Integration....17
Figure 2-11. Interrupts....18
Figure 2-12. Pin Configuration and PinMux....19
Figure 3-1. MCAN Loopback Message....20
Figure 3-2. Output of Bus-Monitoring Tool for mcan_multi_message_tx....20
Figure 3-3. Output of Bus-Monitoring Tool for mcan_multi_message_tx_tcan114x....21
Figure 3-4. Output of Bus-Monitoring Tool for mcan_single_message_tx....21

Trademarks

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

Copyright © 2026 Texas Instruments Incorporated

1 CAN Introduction

This section introduces CAN standard and MSPM0 MCAN modules.

1.1 CAN Standard

This section introduces CAN bus and frame structure, and CAN bit timing.

For more CAN standard specification, see official documentation.

1.1.1 CAN Bus

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.

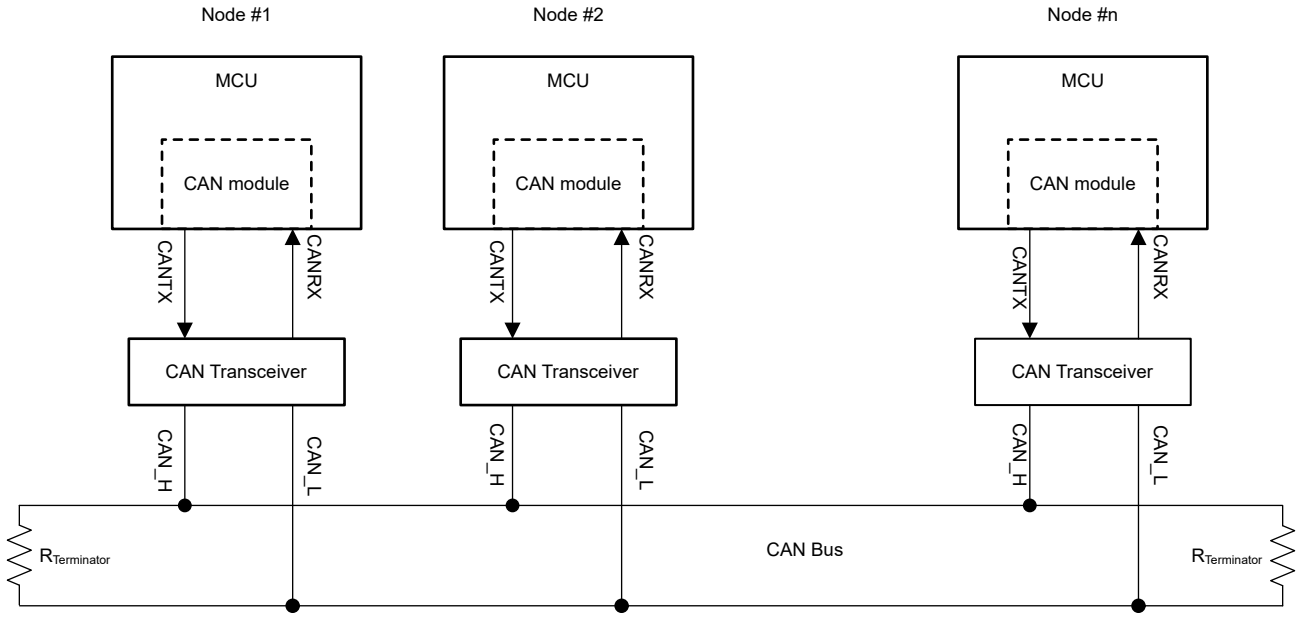


Figure 1-1. Typical CAN Bus

1.1.2 CAN Frame

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 eight 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.

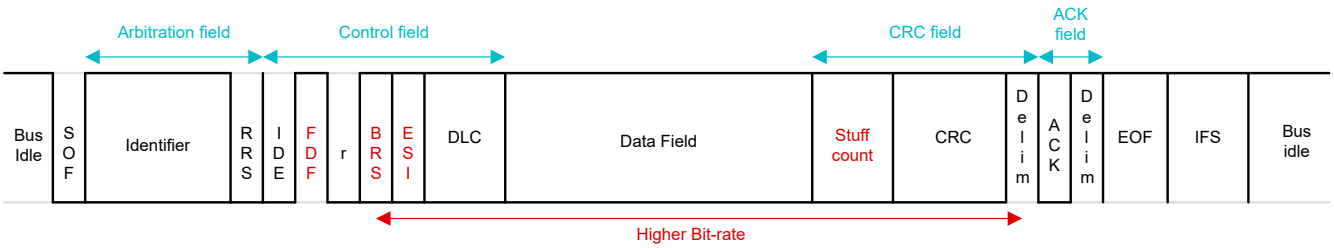


Figure 1-2. CAN FD Frame

Copyright © 2026 Texas Instruments Incorporated

1.1.3 CAN Bit Timing

CAN bit rate is the number of bits per second that was transmitted in the absence of resynchronization by an ideal transmitter.

In CAN-FD protocol, two bit rates is used, arbitration bit rate is used in low speed - arbitration phase and the data bit rate are used in high speed - data phase.

Both arbitration and data bit rate can be divided into separate non-overlapping time segments.

- Synchronization segment (Sync_Seg): Sync_Seg is used to synchronize the various nodes on the bus. An edge is expected to lie within this segment.
- Propagation segment (Prop_Seg): Prop_Seg is used to compensate for the physical delay times within the network. Prog_Seg = 2*(signal's propagation + input comparator delay + output driver delay).
- Phase buffer segment 1 (Phase_Seg1)
- Phase buffer segment 2 (Phase_Seg2): Phase_Seg1 and Phase_Seg2 are used to compensate for edge phase errors. These segments can be lengthened or shortened by resynchronization.

These segments form the bit time as shown in Figure 1-3.

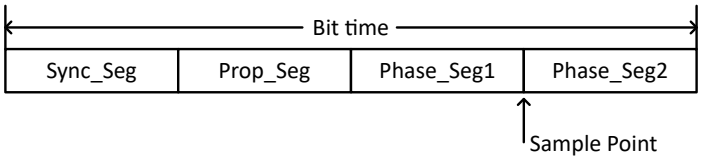


Figure 1-3. Partition of Bit Time

Other related CAN bit timing shown below.

- Sample point: Sample point is the point of time at which the bus level is read and interpreted as the value of that respective bit. It's location is at the end of Phase_Seg1.
- Time quantum (tq): The *tq* is a fixed unit of time derived from the oscillator period. There exists a programmable prescaler, with integral values, ranging at least from 1 to 32. Starting with the Minimum Time Quantum, the Time quantum can have a length of *tq = BRP * Minimum Time Quantum* with BRP the value of bit rate prescaler and Minimum Time Quantum usually the CAN module clock source.

Copyright © 2026 Texas Instruments Incorporated

1.2 MSPM0 MCAN Module

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

Copyright © 2026 Texas Instruments Incorporated

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 of LP-MSPM0G3507, based on Sysconfig tools 1.25.0.

2.1 MCAN Clock Frequency

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

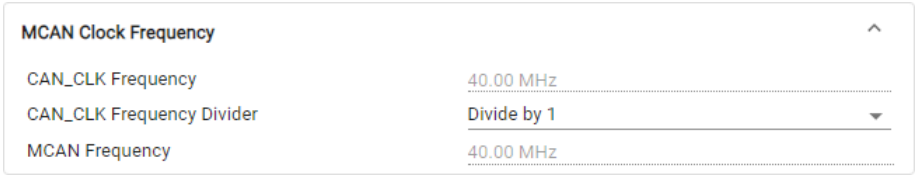


Figure 2-1. MCAN Clock Frequency

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.

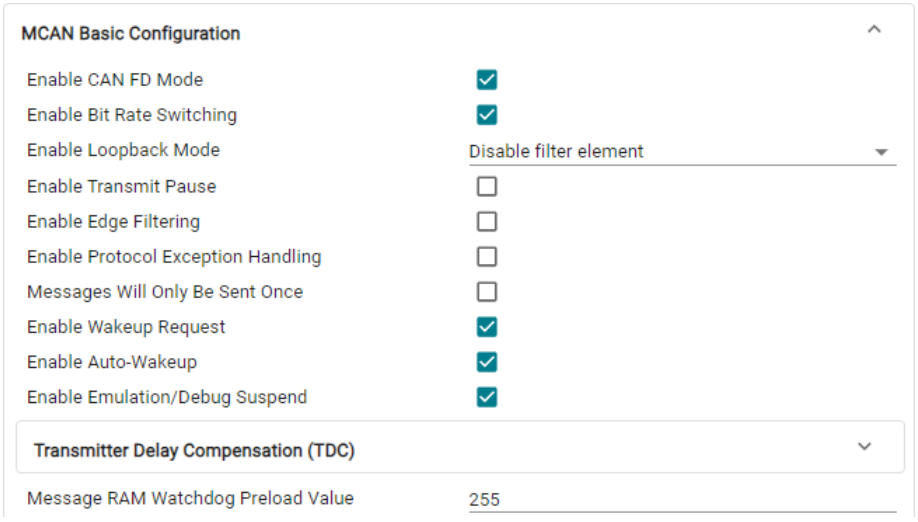


Figure 2-2. MCAN Basic Configuration

- Enable CAN FD Mode: where CAN flexible data mode needs to be enabled.

Copyright © 2026 Texas Instruments Incorporated

- 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. For Tx message of MSPM0 mcan module, user can control each message's bit rate switching in source code.
- 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.

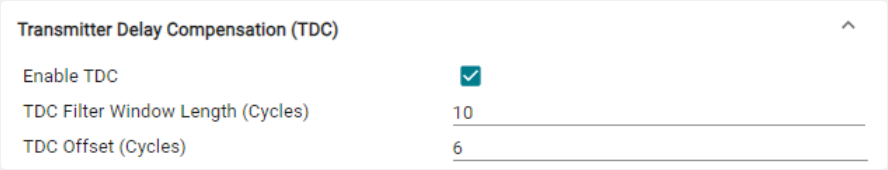


Figure 2-3. Transmitter Delay Compensation (TDC)

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.

Copyright © 2026 Texas Instruments Incorporated

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, see 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.

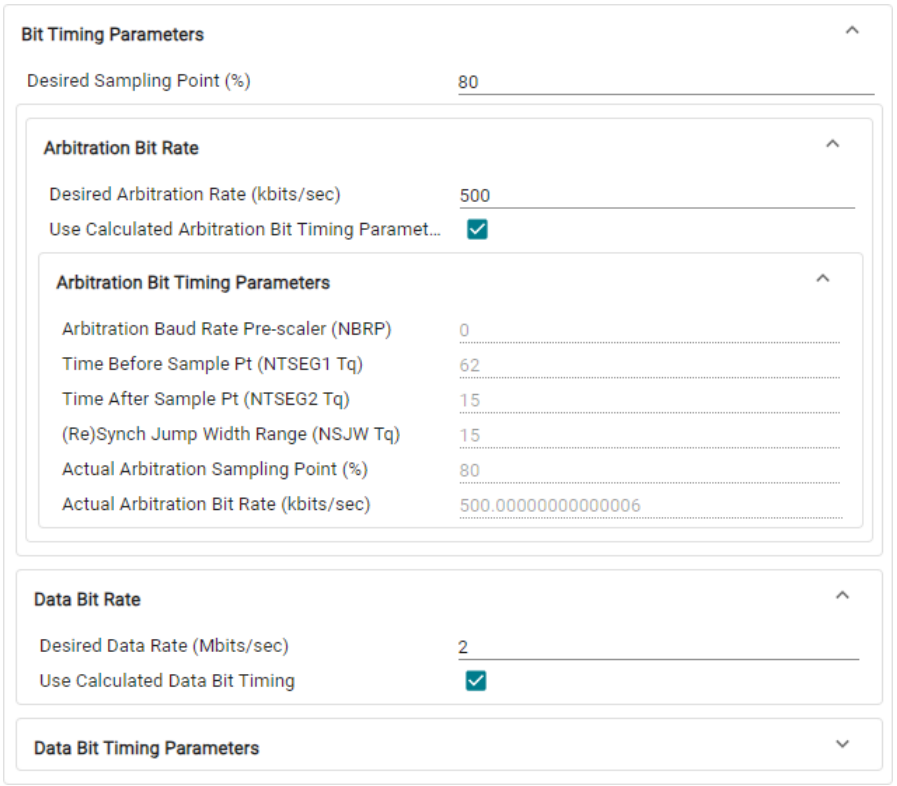


Figure 2-4. Bit Timing Parameters

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 quantum (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), Synchronization segment (Sync_Seg), Propagation segment (Prop_Seg), Phase buffer segment 1 (Phase_Seg1), Phase buffer segment 2 (Phase_Seg2), 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. For the detailed description of these segments, refer to CAN Bit Timing.

Here are parameters introduction in Figure 2-4,

- 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.

Copyright © 2026 Texas Instruments Incorporated

- 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.

Naming of the CAN bit timing always changes between different system, for example, host computer CAN Analyzer software, EB tresos in MCAL, and MCU's documents. User can refer to Table 2-1 to quickly understand the MSPM0 CAN bit timing naming difference and mapping between MSPM0 system configuration tools, and common CAN specification documents. In Table 2-1, names with x means x can be N or D, for example, xBRP can be NBRP (Arbitration Baud Rate Pre-scaler) or DBRP (Data Baud Rate Pre-scaler). For detailed MSPM0 register description,

Table 2-1. Bit Timing Naming

Common CAN	SysConfig	MSPM0 Register
tq	MCAN Frequency/xBRP	MCAN_FCLK/xBRP
Sync_Seg	N/A	Fix at 1
Prop_Seg	xTSEG1	
Phase_Seg1		
Phase_Seg2	xTSEG2	
SJW		

In MSPM0 SysConfig and register level, Sync_Seg in MSPM0 is fixed at 1 tq on hardware, xTSEG1 is combined with two segments Prop_Seg and Phase_Seg1 in common CAN. Total tq of a bit time in Sysconfig is *tq count = 1 + xTSEG1 + xTSEG2*. Sampling point is between xTSEG1 and xTSEG2.

Syscfg Bit Timing Configuration Example show a simple comparison/mapping table of CAN bit timing configuration between common CAN, and SysConfig, and MSPM0 register. MSPM0 MCAN module's bit timing is set to CAN-FD with bit rate switching enable, 500kbit/s + 2Mbit/s and sampling at 80%. Clock source MCAN_FCLK is 40MHz without divide.

Table 2-2. Syscfg Bit Timing Configuration Example

	Arbitration		Data		
Bit Length	80tq = 2us		20tq = 0.5us		
Bit Rate	0.5MBit/s		2MBit/s		
Clock Source	40MHz				
Time Quantum (tq)	25ns				
Common CAN Parameter			Common CAN Parameter		
Bit Time Parameter	Arbitration	Data	Bit Time Parameter	Arbitration	Data
N/A			xBRP	0	0
Sync_Seg	1	1	Hardware fixed at 1		
Prop_Seg	63	15	xTSEG1	62	14
Phase_Seg1					
Phase_Seg2	16	4	xTSEG2	15	3
SJW	16	4	xSJW	15	3

Copyright © 2026 Texas Instruments Incorporated

2.2.3 Message RAM Configuration

The MCAN module has a Message RAM. The main purpose of the Message RAM is to store multiple filter, transmit, and received message, several section can be configured in MCAN message RAM, and make sure that none of the sections are overlapping by comparing corresponding start address and end address with each other.

1. Message ID filter elements: standard 11-bit filter, and extended 29-bit filter.
2. Transmit message: TX Buffers, TX FIFO, and TX Event FIFO.
3. Received message: Rx FIFO0, Rx FIFO1, and Rx Buffers.

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.

MCAN Message RAM Configuration Example give an example of how to config a MCAN message RAM, refer to Sysconfig Configuration for MCAN Module for example project, detailed information of each configuration is introduced in this section.

Table 2-3. MCAN Message RAM Configuration Example

Section	Size	Number	Start Address	End Address
Standard ID Filter	4	4	0	16
Extended ID Filter	8	4	16	48
TX Buffers	72	0	N/A	
TX FIFO	72	4	48	336
TX Event FIFO	8	4	336	368
RX FIFO0	72	6	368	800
RX FIFO1	72	2	800	944
RX Buffer	72	1	944	1016

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.

Copyright © 2026 Texas Instruments Incorporated

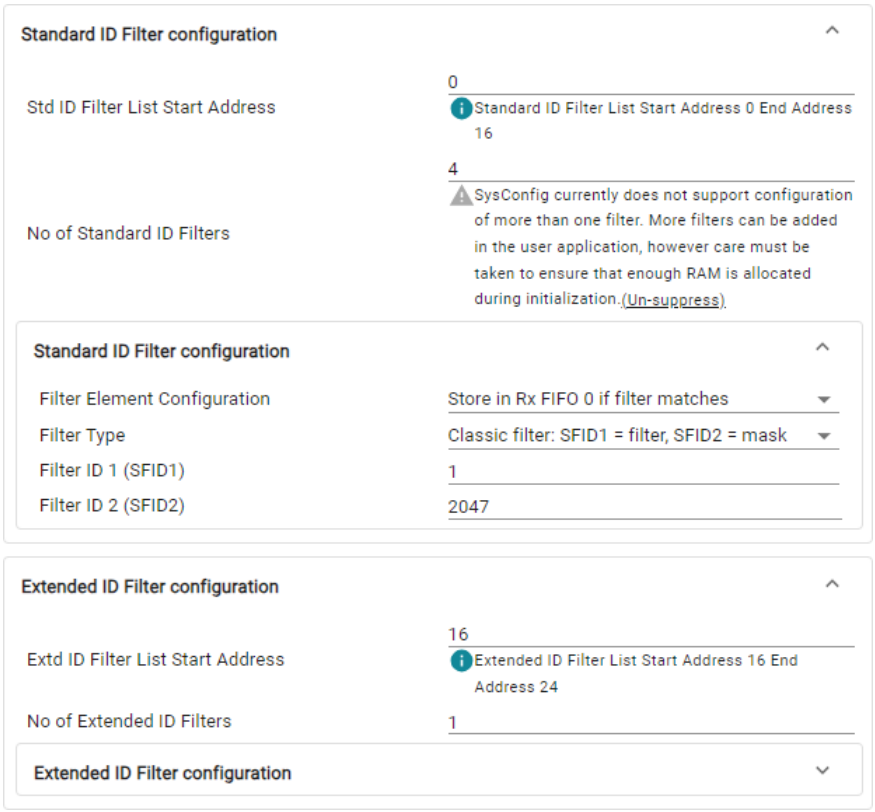


Figure 2-5. Standard and Extended ID Filter Configuration

- 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, and there is no address overlapping issue.
- 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

Copyright © 2026 Texas Instruments Incorporated

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 Config More than One Filter

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.

Example below shows how to add more standard filters in the application code. *gMCAN0StdFltWr* filter element is configured to a strandard filter *sfid1*, with mask *sfid2*, and store the matched message to FIFO0. Then use *DL_MCAN_addStdMsgIDFilter* to write *gMCAN0StdFltWr* to second filter's RAM address (*filterIndex* = 1) (Usually, first filter (*filterIndex* = 0) can be configured in Sysconfig).

```
uint8_t filterIndex = 1U;
static const DL_MCAN_StdMsgIDFilterElement gMCAN0StdFltWr = {
    .sfec  = 0x1,    // Save to FIFO0
    .sft   = 0x2,    // Filter sfid1 with mask sfid2
    .sfid1 = 3,
    .sfid2 = 0x7FF,
};
/* Configure Standard ID filter element */
DL_MCAN_addStdMsgIDFilter(MCAN0_INST, filterIndex, &gMCAN0StdFltWr);
```

Example below shows how to add more extended filters in the application code. *gMCAN0ExtFltWr* filter element is configured to match a range of filter id between efid1 and efid2, 0x10 - 0x20, for example, extended message with id = 0x15, and save the matched message to FIFO0. Then use *DL_MCAN_addExtMsgIDFilter* to write *gMCAN0ExtFltWr* to fourth filter's RAM address (*filterIndex* = 3).

```
uint8_t filterIndex = 3U;
static const DL_MCAN_ExtMsgIDFilterElement gMCAN0ExtFltWr = {
    .efec = 0x1,    // Save to FIFO0
```


Copyright © 2026 Texas Instruments Incorporated

```
    .eft  = 0x0,    // Range filter from efid1 to efid2
    .efid1 = 0x10,
    .efid2 = 0x20,
};
/* Configure Extended ID filter element */
DL_MCAN_addExtMsgIDFilter(MCAN0_INST, filterIndex, &gMCAN0ExtFltWr);
```

In order to read and verify the configured filter, user can add below API to read the filter from message RAM back to MSPM0 RAM variables, below code includes both standard and extended filter read function.

By calling the *DL_MCAN_readStdMsgIDFilter* API, CPU reads the message RAM filter element with index *filtNum*, and copy the filter value to *gMCAN0StdFltRd*.

```
uint32_t filtNum = 2;
DL_MCAN_StdMsgIDFilterElement gMCAN0StdFltRd;
void DL_MCAN_readStdMsgIDFilter(MCAN_Regs *mcan, uint32_t filtNum,
    DL_MCAN_StdMsgIDFilterElement *elem)
{
    uint32_t startAddr, elemAddr, regVal;

    //startAddr = HW_RD_FIELD32(&mcan->MCANSS.MCAN.MCAN_SIDFC, MCAN_SIDFC_FLSSA);
    startAddr = *(volatile uint32_t *)(&mcan->MCANSS.MCAN.MCAN_SIDFC);
    startAddr = (startAddr & 0x0000FFFC) >> 2;
    startAddr = (uint32_t)(startAddr << 2U);
    elemAddr  = startAddr + (filtNum * 1U * 4U);
    elemAddr += 0x0U;

    //regVal = HW_RD_REG32(((uint32_t) mcan + (uint32_t) elemAddr));
    regVal = *(volatile uint32_t *)( (uint32_t) mcan + (uint32_t) elemAddr );
    elem->sfid2 = (regVal >> 0U)  & 0x000007FF;
    elem->sfid1 = (regVal >> 16U) & 0x000007FF;
    elem->sfec  = (regVal >> 27U) & 0x00000007;
    elem->sft   = (regVal >> 30U) & 0x00000003;
};
DL_MCAN_readStdMsgIDFilter(MCAN0_INST, filtNum, &gMCAN0StdFltRd);
```

By calling the *DL_MCAN_readExtMsgIDFilter* API, CPU reads the message RAM filter element with index *filtNum*, and copy the filter value to *gMCAN0ExtFltRd*.

```
uint32_t filtNum = 2;
DL_MCAN_ExtMsgIDFilterElement gMCAN0ExtFltRd[CANEXTFLTNUM] = {0};
void DL_MCAN_readExtMsgIDFilter(MCAN_Regs *mcan, uint32_t filtNum,
    DL_MCAN_ExtMsgIDFilterElement *elem)
{
    uint32_t startAddr, elemAddr, regVal;

    //startAddr = HW_RD_FIELD32(&mcan->MCANSS.MCAN.MCAN_XIDFC, MCAN_XIDFC_FLESA);
    startAddr = *(volatile uint32_t *)(&mcan->MCANSS.MCAN.MCAN_XIDFC);
    startAddr = (startAddr & 0x0000FFFC) >> 2;
    startAddr = (uint32_t)(startAddr << 2U);
    elemAddr  = startAddr + (filtNum * 2U * 4U);
    elemAddr += 0x0U;

    //regVal = HW_RD_REG32(((uint32_t) mcan + (uint32_t) elemAddr));
    regVal = *(volatile uint32_t *)( (uint32_t) mcan + (uint32_t) elemAddr );
    elem->efid1 = (regVal >> 0U)  & 0x1FFFFFFF;
    elem->efec  = (regVal >> 29U) & 0x00000007;

    elemAddr += 4U;
    //regVal = HW_RD_REG32(((uint32_t) mcan + (uint32_t) elemAddr));
    regVal = *(volatile uint32_t *)( (uint32_t) mcan + (uint32_t) elemAddr );
    elem->efid2 = (regVal >> 0U)  & 0x1FFFFFFF;
    elem->eft   = (regVal >> 30U) & 0x00000003;
};
DL_MCAN_readExtMsgIDFilter(MCAN0_INST, filtNum, &gMCAN0ExtFltRd);
```

2.2.3.2 TX MSG RAM

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

Copyright © 2026 Texas Instruments Incorporated

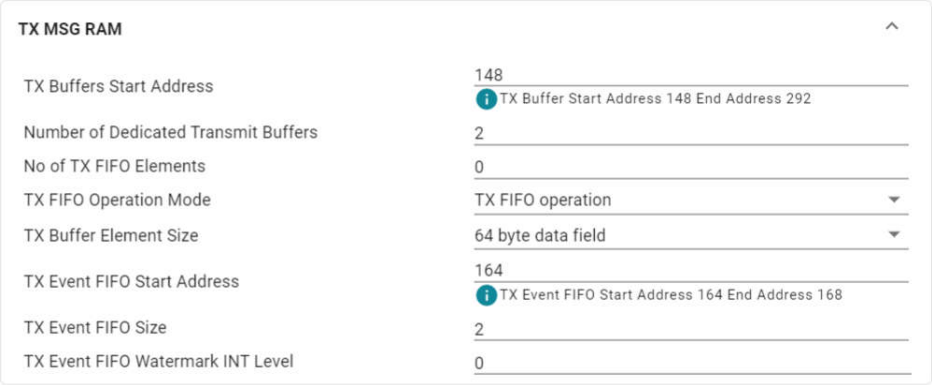


Figure 2-6. TX MSG RAM

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.

- 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.

Table 2-4 shows the differences between Tx buffer mode, Tx FIFO mode and Tx queue mode.

Table 2-4. 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).

2.2.3.3 RX MSG RAM

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

Copyright © 2026 Texas Instruments Incorporated

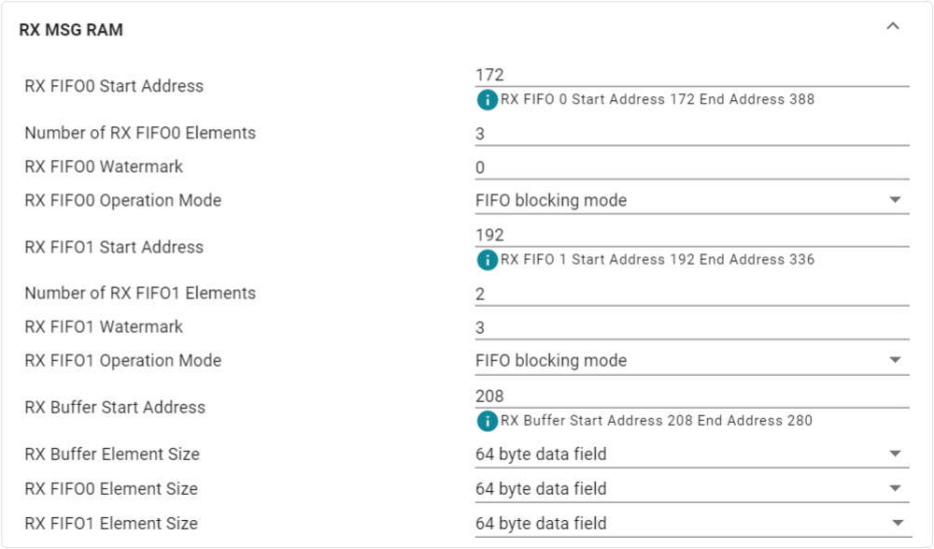


Figure 2-7. RX MSG RAM

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, RX FIFO0, and RX FIFO1 Element Size: defines the Rx buffer or FIFO element size.

The buffer, FIFO0, and FIFO1 use independent starting addresses in message RAM, and can't overlap. Users can freely control the length of each section to meet application requirements.

There is no *Number of RX Buffer Elements* section, user can control buffer write index in filter elements sfid2 or efid2 when sfec or efec set to 0x7. Or refer to the exmaple code this application used in Sysconfig Configuration for MCAN Module. Maxium index need to be handled by user, based on the free space left for RX Buffer, in this example, only one RX Buffer space size is left for Rx Buffer.

Copyright © 2026 Texas Instruments Incorporated

2.3 Advanced Configuration

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

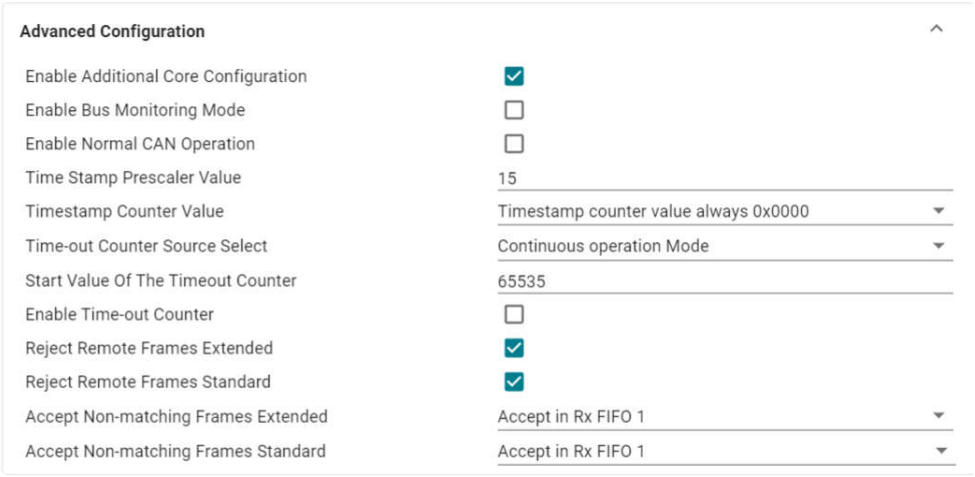


Figure 2-8. Advanced Configuration

- 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.

Copyright © 2026 Texas Instruments Incorporated

- 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 the Retention Configuration block.

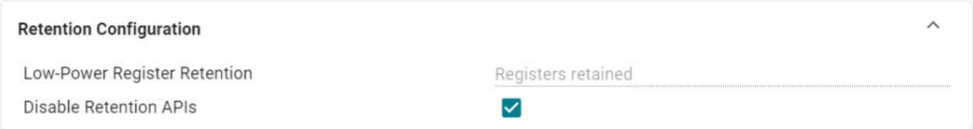


Figure 2-9. Retention Configuration

- 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

The MCAN module contains one event publisher (CPU_INT) that manages MCAN interrupt requests (IRQs) to the CPU NVIC by a static event route. Figure 2-10 shows the integration of the MCAN module in the device.

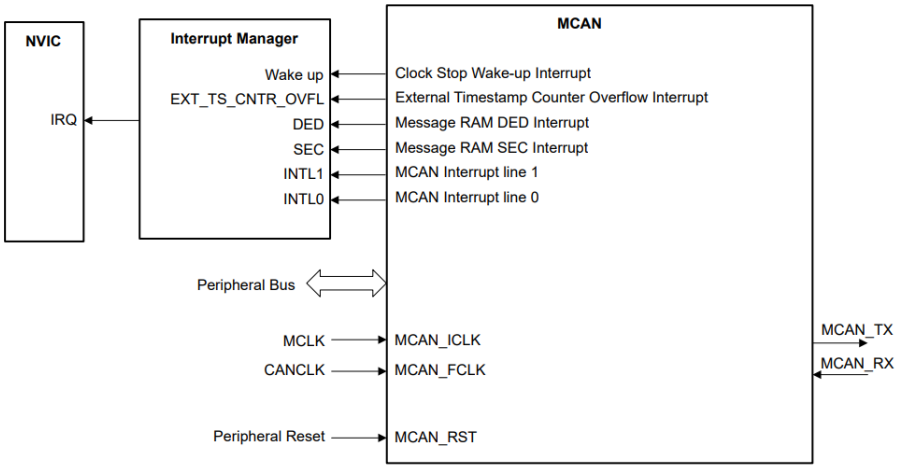


Figure 2-10. MCAN Integration

Figure 2-11 shows what parameters are included in Interrupts block.

Copyright © 2026 Texas Instruments Incorporated

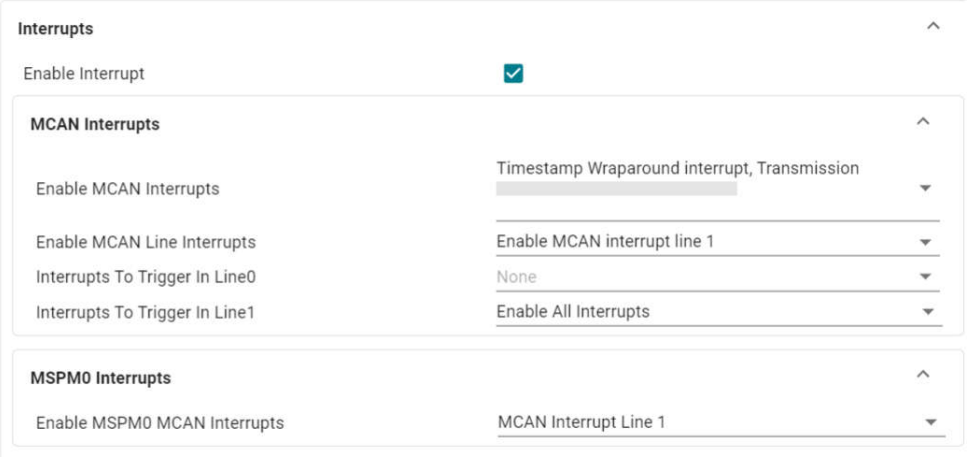


Figure 2-11. Interrupts

- MCAN Interrupts - MCAN block in Figure 2-10
  - 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 - Interrupt Manager in Figure 2-10
  - 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.

User can freely assign MCAN interrupt to Line0 or Line1. In example Figure 2-11, All interrupts are assigned to Line1, and no interrupt is assigned to Line0.

Copyright © 2026 Texas Instruments Incorporated

2.6 Pin Configuration and PinMux

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

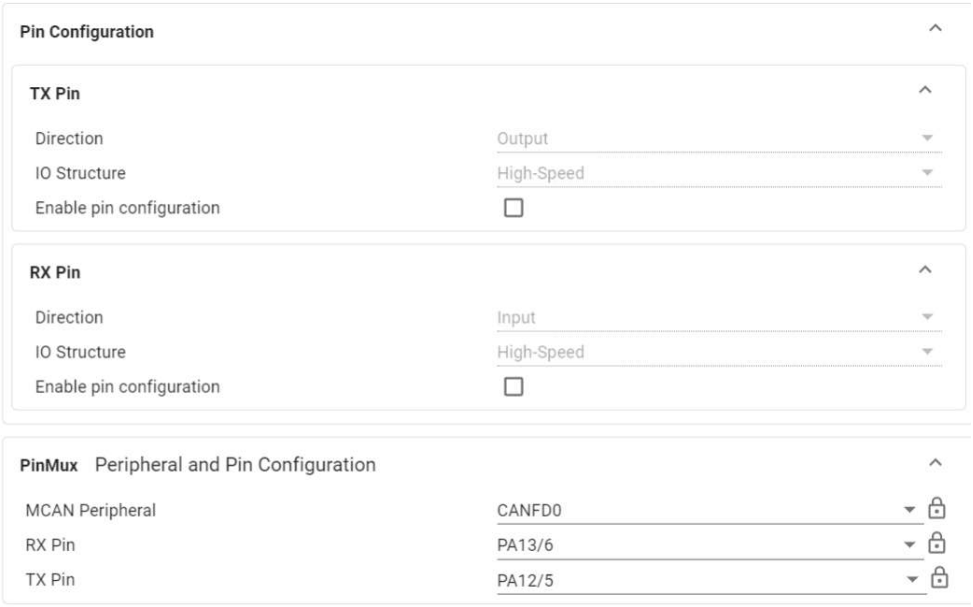


Figure 2-12. Pin Configuration and PinMux

- 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.

Copyright © 2026 Texas Instruments Incorporated

3 SDK Example Code Introduction

SDK Example Code Introduction introduces six MSPM0 MCAN code examples in driverlib folder. User can access the source in folder: {mspm0_root_folder}\examples\nortos\LP_MSPM0G3507\driverlib\mcan_*. For the MSPM0 SDK documents, refer to the online documents.

3.1 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.

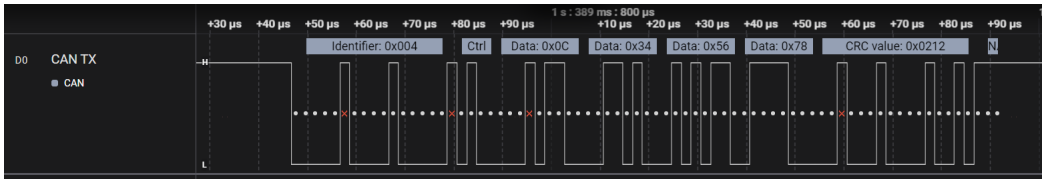


Figure 3-1. MCAN Loopback Message

3.2 mcan_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.

3.3 mcan_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.

3.4 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
					ALL	ALL	ALL		
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

Copyright © 2026 Texas Instruments Incorporated

3.5 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 initiailzes 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
					ALL	ALL	ALL		
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

3.6 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
					ALL	ALL	ALL		
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

Copyright © 2026 Texas Instruments Incorporated

4 MCAN Use Case

Besides the Sysconfig configuration guide and SDK example introduction, MCAN Use Case further introduces several useful MSPM0 MCAN TX/RX handling mechanism and how to control MCAN module in low power mode and CAN bus off status. To help users better understand MSPM0 MCAN module and accelerate the development. Users can download the example code in Sysconfig Configuration for MCAN Module for reference.

4.1 TX Buffer Element

MCAN TX buffer element is a MSPM0 RAM variable to store the message ID and data with CAN control field information that need to be transmitted. TX buffer element can be wirte into MCAN message RAM via API DL_MCAN_writeMsgRam.

Below code shows how to define a TX buffer element variable and initialize the element, and use driverlib API to write to MCAN message RAM.

```
DL_MCAN_TxBufElement txMsg;
txMsg.id = ((uint32_t)(0x4)) << 18U;    // ID = 0x4
txMsg.rtr = 0U;     // Transmit data frame
txMsg.xtd = 0U;     // 11-bit standard identifier
txMsg.esi = 0U;     // ESI bit in CAN FD format depends only on error passive flag
txMsg.dlc = 1U;     // Transmitting 1 bytes
txMsg.brs = 1U;     // CAN FD frames transmitted with bit rate switching
txMsg.fdf = 1U;     // Frame transmitted in CAN FD format
txMsg.efc = 1U;     // Store Tx events
txMsg.mm = 0xAAU;   // Message marker = 0xAA
txMsg.data[0] = 0x01;   // Data bytes
/* Write Tx Message to the Message RAM. */
DL_MCAN_writeMsgRam(MCAN0_INST, DL_MCAN_MEM_TYPE, Index, &txMsg);
```

Table 4-1 shows the description of each member of Tx buffer element.

Table 4-1. Structure for MCAN Tx Buffer Element

Member	Name	Description
id	Identifier	Identifier, 11-bit standard identifiers need to be left-shifted by 18 bits before being assigned a value.
rtr	Remote Transmission Request	0 = Transmit data frame 1 = Transmit remote frame
xtd	Extended Identifier	0 = 11-bit standard identifier 1 = 29-bit extended identifier
esi	Error State Indicator	0 = ESI bit in CAN FD format depends only on error passive flag 1 = ESI bit in CAN FD format transmitted recessive
dlc	Data Length Code	0-8 = CAN + CAN FD: transmit frame has 0-8 data bytes 9-15 = CAN: transmit frame has 8 data bytes 9-15 = CAN FD: transmit frame has 12/16/20/24/32/48/64 data bytes
brs	Bit Rat Switching	0 = CAN FD frames transmitted without bit rate switching 1 = CAN FD frames transmitted with bit rate switching
fdf	FD Format	0 = Frame transmitted in Classic CAN format 1 = Frame transmitted in CAN FD format
efc	Event FIFO Control	0 = Don't store Tx events 1 = Store Tx events
mm	Message Marker	Message marker to put into TX Event FIFO

Copyright © 2026 Texas Instruments Incorporated

Table 4-1. Structure for MCAN Tx Buffer Element (continued)

Member	Name	Description
data	Data bytes	Only first dlc number of bytes are valid.

4.2 RX Buffer Element

MCAN RX buffer element is a MSPM0 RAM variable to store the message ID and data with CAN control field information that received from CAN bus. RX buffer element can be read from MCAN message RAM via API DL_MCAN_readMsgRam.

Below code shows how to initialize the element and use driverlib API to read CAN message from MCAN message RAM to a RX buffer element.

```
DL_MCAN_RxBufElement rxMsg;
/* Read from Message RAM to Rx Buffer Element. */
DL_MCAN_readMsgRam(MCAN0_INST, DL_MCAN_MEM_TYPE, bufIndex, fifoIndex, &rxMsg);
rxMsgId = rxMsg.id;
rxMsgDlc = rxMsg.dlc;
for(int i = 0; i <= rxMsgDlc; i++) {
    rxMsgData[i] = rxMsg.data[i];
}
```

Structure for MCAN Rx Buffer Element shows the description of each member of RX buffer element.

Table 4-2. Structure for MCAN Rx Buffer Element

Member	Name	Description
id	Identifier	Identifier, 11-bit standard identifiers need to be right-shifted by 18 bits after being read from message RAM.
rtr	Remote Transmission Request	0 = Received frame is a data frame 1 = Received frame is a remote frame
xtd	Extended Identifier	0 = 11-bit standard identifier 1 = 29-bit extended identifier
esi	Error State Indicator	0 = Transmitting node is error active 1 = Transmitting node is error passive
rxts	Rx Timestamp	Timestamp Value
dlc	Data Length Code	0-8 = CAN + CAN FD: received frame has 0-8 data bytes 9-15 = CAN: received frame has 8 data bytes 9-15 = CAN FD: received frame has 12/16/20/24/32/48/64 data bytes
brs	Bit Rat Switching	0 = Frame received without bit rate switching 1 = Frame received with bit rate switching
fdf	FD Format	0 = Standard frame format 1 = CAN FD frame format (new DLC-coding and CRC)
fidx	Filter Index	Index of the filter
anmf	Accepted Non-matching Frame	0 = Received frame matching filter index FIDX 1 = Received frame did not match any Rx filter element
data	Data bytes	Only first dlc number of bytes are valid.

Copyright © 2026 Texas Instruments Incorporated

4.3 TX Handling Buffer Mode

To transmit CAN message in TX buffer mode, need to reserve message RAM for Tx buffer in SysConfig - Number of Dedicated Transmit Buffers.

Here is an example sequence of TX buffer handling mechanism, polling the Transmission Request Pending register to find the avaiable buffer, then write the CAN message to buffer:

- Find empty TX buffer in message RAM to write the TX buffer element, or check the target TX buffer empty status.
- Write message RAM with API DL_MCAN_writeMsgRam, in DL_MCAN_MEM_TYPE_BUF mode.
- Add transmission request to CANTxBufferPutIndex.
- Optional, wait for transmission finished.

```
uint8_t MCAN_TxBufPoll(MCAN_Regs *mcan, DL_MCAN_TxBufElement *txMsg, uint8_t length)
{
    uint8_t CANTxBufferPutIndex = 0;
    uint8_t loop_i = 0;

    for(loop_i = 0; loop_i < length; loop_i++) {
        /* Find the empty Tx buffer to start transfer. */
        do {
            CANTxBufferPutIndex = (__builtin_ctz(~DL_MCAN_getTxBufReqPend(mcan)));
        }
        while(CANTxBufferPutIndex >= MCAN0_INST_MCAN_TX_BUFF_SIZE);
        /* Write Tx Message to the Message RAM. */
        DL_MCAN_writeMsgRam(mcan, DL_MCAN_MEM_TYPE_BUF, CANTxBufferPutIndex, &txMsg[loop_i]);
        /* Add request for transmission. */
        DL_MCAN_TXBufAddReq(mcan, CANTxBufferPutIndex);
    }

    /* Wait until all pending transmission requirements are proceed */
    while(DL_MCAN_getTxBufReqPend(mcan) != 0x00);

    return loop_i;
}
```

TX buffer transmission transmiting CAN message fully controlled by CPU, CPU can control which index of buffer send firstly by add request API DL_MCAN_TXBufAddReq. If TX Buffer is written with several message and add transmission request at the same time, message with the lowest id number is transmitted first. If TX buffer is written with several message with same id, message with lowest buffer number is transmitted first.

4.4 TX Handling FIFO Mode

To transmit CAN message in TX FIFO mode, need to reserve message RAM for Tx FIFO in SysConfig - No of TX FIFO Elements.

Here is an example sequence of TX FIFO handling mechanism, polling the TX FIFO status to wait for TX FIFO not full status, then write the CAN message to FIFO:

- Check and wait for TX FIFO to be empty.
- Write message RAM with API DL_MCAN_writeMsgRam, in DL_MCAN_MEM_TYPE_FIFO mode.
- Add transmission request to txFS.putIdx.
- Optimal, wait for transmission finished.

```
uint8_t MCAN_TxFIFOPoll(MCAN_Regs *mcan, DL_MCAN_TxBufElement *txMsg, uint8_t length)
{
    uint8_t loop_i = 0;

    while(loop_i < length) {
        do{
            /* Update the Tx FIFO Status. */
            DL_MCAN_getTxFIFOQueStatus(mcan, &txFS);
            /* Add timeout handling here */
        }
        while(txFS.fifoFull);
        /* Write Tx Message to the Message RAM. */
        DL_MCAN_writeMsgRam(mcan, DL_MCAN_MEM_TYPE_FIFO, txFS.putIdx, &txMsg[loop_i++]);
        /* Add request for transmission. */
        DL_MCAN_TXBufAddReq(mcan, txFS.putIdx);
```


Copyright © 2026 Texas Instruments Incorporated

```
    }

    /* Wait until all pending transmission requirements are proceed */
    while(DL_MCAN_getTxBufReqPend(mcan) != 0x00);

    return loop_i;
}
```

The data stored in the Tx FIFO messages are transmitted starting with the message referenced by the Get Index MCAN_TXFQS.TFGI field. After each transmission the Get Index is incremented until the Tx FIFO is empty.

For both TX buffer and FIFO mode, MCAN interrupt supports to indicate transmission is done, use below sequence to handle the TX interrupt. User can refer to above TX buffer and FIFO polling handling example to write a interrupt handler *MCAN_TxFIFOInt_Service* or refer to the demo code mentioned in Sysconfig Configuration for MCAN Module.

- Make sure config MCAN and MSPM0 Interrupts correctly in Syscfg - MCAN.
- Enable corresponding TX buffer interrupts in MCAN_TXBTIE.
- Enable TC transmission completed interrupt for MCAN.
- Start transmission with buffer or FIFO mode.
- Process the TC interrupt in IRQHandler.
- Optimal, use TX event FIFO interrupt.

```
    /* Enable all Tx Buffer's Interrupt */
    mcan->MCANSS.MCAN.MCAN_TXBTIE = 0xFFFFFFFF;
    /* Enable MCAN Transmission Completed interrupt. */
    DL_MCAN_enableIntr(mcan, DL_MCAN_INTERRUPT_TC, 1U);

void MCAN0_INST_IRQHandler(void)
{
    switch (DL_MCAN_getPendingInterrupt(MCAN0_INST)) {
        case DL_MCAN_IIDX_LINE1:
            /* Check MCAN interrupts fired during TX/RX of CAN package */
            gInterruptLine1Status = DL_MCAN_getIntrStatus(MCAN0_INST);

            /* Transmission Complete from one interrupt enabled buffer/FIFO */
            if(gInterruptLine1Status&DL_MCAN_INTERRUPT_TC) {
                MCAN_TxFIFOInt_Service(MCAN0_INST);
            }

        default:
            break;
    }
}
```


Copyright © 2026 Texas Instruments Incorporated

4.5 RX Handling Buffer Mode

To receive CAN message in RX buffer mode, need to reserve message RAM for **Dedicated Rx Buffers** in SysConfig - RX Buffer Start Address, set a valid start address and there is enough free space for user RX buffer number. This part is calculated mannully and user can refer to MCAN Message RAM Configuration Example, and Technical Reference Manual - Rx Buffer and FIFO Element, any free space after start address is used as RX buffer and can't be used for other purpose.

ID filter must be configured correctly to correctly receive CAN message into the CAN RAM, Filter configuration guide can be found Standard and Extended ID Filter Configuration. Here is an example of a strandard ID filter to store ID = 3 message to index = 0 RX buffer, and use API DL_MCAN_addStdMsgIDFilter to save configuration to CAN RAM.

```
static const DL_MCAN_StdMsgIDFilterElement gMCAN0StdFiltelem = {
    .sfec = 0x7,  // Save message to DRX
    .sft  = 0x0,  // Ignored when sfec = 0x7
    .sfid1 = 3,   // Filter ID, message must match this ID
    .sfid2 = 0x0, // RX buffer index when sfec = 0x7
};
/* Configure Standard ID filter element */
DL_MCAN_addStdMsgIDFilter(MCAN0_INST, 0U, (DL_MCAN_StdMsgIDFilterElement *) &gMCAN0StdFiltelem);
```

Here is an example sequence of RX buffer handling machanism, polling the MCAN New Data register (MCAN_NDATx) to find the buffer that received the message, then read message from the buffer:

1. Set NDATIdx and canBufStatus based on the given required bufIdx.
2. Polling MCAN_NDATx to wair for valid CAN message recevied into CAN RX buffer.
3. Read message RAM with API DL_MCAN_readMsgRam, in DL_MCAN_MEM_TYPE_BUF mode.
4. Write 1 to corresponding status bit to clear the status.

```
uint8_t MCAN_RxBufPoll(MCAN_Regs *mcan, DL_MCAN_RxBufElement *rxMsg, uint8_t bufIdx)
{
    uint8_t NDATIdx = 0;
    volatile uint32_t *canBufStatus = NULL;

    if(bufIdx <= 31) {
        NDATIdx = bufIdx;
        canBufStatus = &mcan->MCANSS.MCAN.MCAN_NDAT1;
    }
    else {
        NDATIdx = bufIdx - 32;
        canBufStatus = &mcan->MCANSS.MCAN.MCAN_NDAT2;
    }

    /* Wait for receiving */
    while( !((*canBufStatus) & (0x00000001 << NDATIdx)) );

    if( (*canBufStatus) & (0x00000001 << NDATIdx) ) {
        /* Read message to rxMsg from CAN buffer */
        DL_MCAN_readMsgRam(mcan, DL_MCAN_MEM_TYPE_BUF, bufIdx, 0, &rxMsg[0]);
        /* Clear corresponding new data status */
        *canBufStatus = 0x00000001 << NDATIdx;

        return 1;
    }

    return 0;
}
```

RX buffer supports interrupt handling, and here is an exmaple of RX buffer interrupt handling, RX buffer uses a dedicated interrupt status DL_MCAN_INTERRUPT_DRX seperated from RX FIFO, and can use API DL_MCAN_enableIntr to enable. User can refer to above RX buffer polling handling example to write a interrupt handler *MCAN_RxBufInt_Service* or refer to the demo code mentioned in Sysconfig Configuration for MCAN Module.

```
DL_MCAN_enableIntr(mcan, DL_MCAN_INTERRUPT_DRX, 1U);

void MCAN0_INST_IRQHandler(void)
```


Copyright © 2026 Texas Instruments Incorporated

```
{
    switch (DL_MCAN_getPendingInterrupt(MCAN0_INST)) {
        case DL_MCAN_IIDX_LINE1:
            /* Check MCAN interrupts fired during TX/RX of CAN package */
            gInterruptLine1Status = DL_MCAN_getIntrStatus(MCAN0_INST);

            /* Message Stored to Dedicated Rx Buffer interrupt */
            if(gInterruptLine1Status&DL_MCAN_INTERRUPT_DRX) {
                MCAN_RxBufInt_Service(MCAN0_INST);
            }

        default:
            break;
    }
}
```

Note

In common use case, configuring a Dedicated RX Buffer for a specific CAN ID is a useful way to implement hardware-filtered message routing. Unlike an RX FIFO, where messages are queued sequentially regardless of ID, a Dedicated RX Buffer is used that a specific CAN ID is always written to the exact same memory location. This prevents high-priority or high-frequency messages from being overwritten or delayed by other traffic, making message processing possible for gateway routing, critical diagnostic tracking, and low-latency command processing. One or several ID Fitlers can be assigned to one RX buffer, and each RX buffer can be routed to difference callback function to get proceed based on NDAT0 and NDAT1 (buffer index).

Copyright © 2026 Texas Instruments Incorporated

4.6 RX Handling FIFO Mode

To receive CAN message in RX FIFO mode, need to reserve message RAM for RX FIFO in SysConfig - Number of RX FIFOx Elements. Different from TX FIFO, MSPM0 MCAN module support two RX FIFOs, user can assign one or multiple ID filter to one RX FIFO.

ID filter must be configured correctly to receive CAN message into the CAN RAM, Filter configuration guide can be found Standard and Extended ID Filter Configuration. Here is an example of a strandard ID filter, to store ID = 3 message to RX FIFO0, and use API DL_MCAN_addStdMsgIDFilter to save configuration to CAN RAM. Because sfid2 = 0x7FF (ID is checked for each bit that correspoding mask bit = 1), message's ID must exactly be same as sfid1.

```
static const DL_MCAN_StdMsgIDFilterElement gMCAN0StdFiltelem = {
    .sfec = 0x1,  // Save to FIFO0
    .sft  = 0x2,  // sfid1 is id, sfid2 is mask
    .sfid1 = 3,   // Filter ID, message must match this ID
    .sfid2 = 0x7FF, // RX buffer index when sfec = 0x7
};
/* Configure Standard ID filter element */
DL_MCAN_addStdMsgIDFilter(MCAN0_INST, 0, &gMCAN0StdFiltelem);
```

Here is an example sequence of RX FIFO handling machanism. After polling and waiting for RX FIFO not empty, read from RX FIFO and write RX FIFO ACK to CAN:

1. Polling the RX FIFO interrupt status and RX FIFO empty status to wait for avaiable message saved into RX FIFO0.
2. Read message RAM with API DL_MCAN_readMsgRam, in DL_MCAN_MEM_TYPE_FIFO mode.
3. Write acknowledge to RX FIFO.

```
uint8_t MCAN_RxFIFOPoll(MCAN_Regs *mcan, DL_MCAN_RxBufElement *rxMsg, uint8_t length)
{
    uint8_t loop_i = 0;

    for(loop_i = 0; loop_i < length; loop_i++) {
        rxFS.num = DL_MCAN_RX_FIFO_NUM_0;
        rxFS.fillLvl = 0;

        /* Waiting for New CAN Message Stored into FIFO 0 */
        while( !(DL_MCAN_getIntrStatus(mcan)&(DL_MCAN_INTERRUPT_RF0N|DL_MCAN_INTERRUPT_RF0F)) );
        /* Clear Receive FIFO 0 New Message Interrupt Flag */
        DL_MCAN_clearIntrStatus(mcan, DL_MCAN_INTERRUPT_RF0N|DL_MCAN_INTERRUPT_RF0F,
DL_MCAN_INTR_SRC_MCAN_LINE_1);

        /* Waiting for FIFO 0 Fill Level Not 0 */
        while ((rxFS.fillLvl) == 0) {
            DL_MCAN_getRxFIFOStatus(mcan, &rxFS);
        }
        DL_MCAN_readMsgRam(mcan, DL_MCAN_MEM_TYPE_FIFO, 0U, rxFS.num, &rxMsg[loop_i]);
        DL_MCAN_writeRxFIFOAck(mcan, rxFS.num, rxFS.getIdx);
    }

    return loop_i;
}
```

RX FIFO supports interrupt handling, and here is an example of RX FIFO handling. both RX FIFO has own interrupt status listed below:

- DL_MCAN_INTERRUPT_RF0N, DL_MCAN_INTERRUPT_RF1N: RX FIFO new message stored.
- DL_MCAN_INTERRUPT_RF0F, DL_MCAN_INTERRUPT_RF1F: RX FIFO new message stored and becomes full.
- DL_MCAN_INTERRUPT_RF0L, DL_MCAN_INTERRUPT_RF1L: RX FIFO message lost occurred.

Copyright © 2026 Texas Instruments Incorporated

- **Note**

  RX FIFO new and full need to use together to cover all the scenario that RX FIFO is receiving message normally. Because of RX FIFO new is not triggered when the FIFO becomes full after a message is written. RX FIFO lost is used as a error handling interrupt to indicate after RX FIFO full status, there is no read operation and new meesage is blocked and dropped or old message is overwritten.

```
/* Enable MCAN receive FIFO 0 new message interrupt. */
DL_MCAN_enableIntr(mcan, DL_MCAN_INTERRUPT_RF0N|DL_MCAN_INTERRUPT_RF0F, 1U);

void MCAN0_INST_IRQHandler(void)
{
    switch (DL_MCAN_getPendingInterrupt(MCAN0_INST)) {
        case DL_MCAN_IIDX_LINE1:
            /* Check MCAN interrupts fired during TX/RX of CAN package */
            gInterruptLine1Status = DL_MCAN_getIntrStatus(MCAN0_INST);

            /* New message received by Rx FIFO 0 */
            if( (gInterruptLine1Status & DL_MCAN_INTERRUPT_RF0N) ||
                (gInterruptLine1Status & DL_MCAN_INTERRUPT_RF0F)
              ) {
                MCAN_RxFIFOInt_Service(MCAN0_INST);
            }

        default:
            break;
    }
}
```

User can refer to above RX FIFO polling handling example to write a interrupt handler *MCAN_RxFIFOInt_Service* or refer to the demo code mentioned in Sysconfig Configuration for MCAN Module.

Copyright © 2026 Texas Instruments Incorporated

4.7 Using MSPM0 MCAN in Low Power Mode

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

Note

The first MCAN message is used as a wake up signal for the MSP device. Use a test message for this wake up function and never use a key message as a wake up message.

An example code is shown below. Users must call API *MCAN_LowPowerMode* to enter the low power mode.

```
void MCAN0_Restart(void)
{
    /* Restart the MCAN */
    DL_MCAN_reset(CANFD0);
    delay_cycles(16);
    DL_MCAN_disablePower(CANFD0);
    delay_cycles(32);
    DL_MCAN_enablePower(CANFD0);
    /* Wait until MCAN module is ready */
    while(!(DL_SYSCTL_getStatus()&DL_SYSCTL_STATUS_MCAN0_READY));
    /* Init MCAN */
    SYSCFG_DL_MCAN0_init();
}

/**
 * Demo for enter and exit STANBY0 mode steps while using MCAN
 * Switch CAN Rx to GPIO input capture in STANDBY mode
 */
void MCAN_LowPowerMode(void)
{
    /* Switch CAN Rx pin to GPIO input mode */
    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);
    /* Enable GPIO interrupt */
    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);

    /* Disable SYSPLL */
    DL_SYSCTL_switchMCLKfromHSCLKtoSYSOSC();
    /* Set default configuration */
    DL_SYSCTL_disableSYSPLL();
    /* Switch low power mode to STANDBY0 */
    DL_SYSCTL_setPowerPolicySTANDBY0();
    /* Enter low power mode here */
    __WFI();
    /* Switch low power mode to SLEEP0 */
    DL_SYSCTL_setPowerPolicyRUN0SLEEP0();
    /* Init the SYSCTL (Clock and power policy) */
    SYSCFG_DL_SYSCTL_init();

    /* Switch GPIO input mode to CAN Rx pin */
    DL_GPIO_initPeripheralInputFunction(GPIO_MCAN0_IOMUX_CAN_RX, GPIO_MCAN0_IOMUX_CAN_RX_FUNC);

    MCAN0_Restart();
}
```

Note

After existing the low power mode, both CAN filter and CAN interrupt need to re-config back to normal operation mode manually.

Copyright © 2026 Texas Instruments Incorporated

4.8 Bus Off Handling

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. This method can recover CAN controller quickly.
- 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). This method meet CAN requirements.

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_LINE1:
            /* Check MCAN interrupts fired during TX/RX of CAN package */
            gInterruptLine1Status = DL_MCAN_getIntrStatus(MCAN0_INST);

            /* MCAN bus off status changed */
            if(gInterruptLine1Status&DL_MCAN_INTERRUPT_BO) {
                DL_MCAN_getProtocolStatus(MCAN0_INST, &gProtStatus);
                if(gProtStatus.busOffStatus == 1) { // Busoff entered
                    CANBusOff = 1;
                }
                else {  // Busoff exited
                    CANBusOff = 0;
                }
            }

        default:
            break;
    }
}
```

Then, check the CANBusOff flag in the main loop. When the bus off state is detected, restart the MCAN module. Choose either Passive recovery or Force recovery menthod.

```
main loop:
{
        /* CAN bus off recovery */
        if(CANBusOff == 1) {
            /* Passive recovery: detected 128 consecutive inst/ of 11 recessive bits */
            MCAN_BusoffRecovery();  // CANBusOff reset to 0 in interrupt
                                    // This API can be directly put into CAN interrupt
```


Copyright © 2026 Texas Instruments Incorporated

```
            /* Force recovery: reset CAN to restore from bus off */
            // MCAN0_Restart();
            // CANBusOff = 0;
        }
}

void MCAN0_Restart(void)
{
    /* Restart the MCAN */
    DL_MCAN_reset(CANFD0);
    delay_cycles(16);
    DL_MCAN_disablePower(CANFD0);
    delay_cycles(32);
    DL_MCAN_enablePower(CANFD0);
    /* Wait until MCAN module is ready */
    while(!(DL_SYSCTL_getStatus()&DL_SYSCTL_STATUS_MCAN0_READY));
    /* Init MCAN */
    SYSCFG_DL_MCAN0_init();
}
```


Copyright © 2026 Texas Instruments Incorporated

5 Design Test and Debug Tips

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.

5.1 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.

5.2 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.

5.3 Debug Checklist

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

5.3.1 Programming Issues

- Is the clock to the MCAN module enabled? Check for CANCLK configuration in the Sysconfig. TI recommends using 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.

Copyright © 2026 Texas Instruments Incorporated

5.3.2 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, is there 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).

5.3.3 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.

6 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. Also, based on production experience, introduce some common use case with user defined example code. Finally, some common debugging issues and relevant debugging suggestions from the software and hardware aspects are introduced.

7 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

Copyright © 2026 Texas Instruments Incorporated

8 Revision History

Changes from Revision * (April 2025) to Revision A (October 2026)	Page
• Added CAN basic standard and MSP MCAN module introduction sections	3
• Updated the numbering format for tables, figures, and cross-references throughout the document	3
• Added CAN bit timing introduction	4
• Changed MCAN Features to MSPM0 MCAN Module	5
• Optimize all figures	6
• Added Syscfg's CAN bit timing table	8
• Added Table 2-3	10
• Added a standalone section to introduce MSP SDK example code	20
• Optimized existing buffer/FIFO section	22
• Added SYSPLL low power mode handling code to CAN low power mode function	30
• Used passive recovery as default method to recovery from CAN bus off	31

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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 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, TI’s General Quality Guidelines, or other applicable terms available either on 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025