

# Getting Started with the MCAN (CAN FD) Module



Yashwant Temburu and Hareesh Janakiraman

C2000 Microcontroller Group

## ABSTRACT

The Modular Controller Area Network (MCAN) peripheral is a CAN Flexible Data-rate (CAN FD) implementation on select devices within the C2000™ real-time Microcontroller (MCU) family. Some devices that have a MCAN peripheral include the TMS320F2838xD, TMS320F2838xS and TMS320F28003x devices. The examples described are meant to be run in any C2000 real-time MCU with a MCAN module. For a complete list of devices that contain the MCAN module, see the [C2000 Real-Time Control Peripherals Reference Guide](#). This application report describes several programming examples to illustrate how the MCAN module is set up for different modes of operation. The objective is to help you come up to speed quickly in programming the MCAN peripheral.

The code examples were tested on CPU1 (28x) of a TMS320F28388D MCU; however, the examples can be easily adapted to run on any C2000 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/decoding is essential for debugging.

Throughout this document, the terms MCAN and CAN FD may be 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 and an Excel spreadsheet to calculate bit-timing parameters are all available for download as part of [C2000Ware](#).

## Table of Contents

<b>1 Introduction</b>	<b>2</b>
1.1 MCAN Features	3
<b>2 Useful tips to run the C2000ware examples</b>	<b>3</b>
<b>3 Program Descriptions</b>	<b>4</b>
<b>4 Debug and Design Tips to Resolve/Avoid CAN Communication Issues</b>	<b>7</b>
4.1 Minimum Number of Nodes Required	7
4.2 Why a Transceiver is Needed	7
4.3 Debug Checklist	7
<b>5 How to Duplicate (clone) an Existing Project</b>	<b>8</b>
<b>6 How to Get Visibility Into Driverlib Files</b>	<b>9</b>
<b>7 References</b>	<b>9</b>

## List of Figures

Figure 1-1. Typical CAN bus	2
Figure 1-2. CAN FD Frame	2
Figure 2-1. Adding the Predefined Symbol in Project Properties	3
Figure 3-1. Output From MCANTX Pin for mcan_ex1_loopback.c	4
Figure 3-2. Output of Bus-Monitoring Tool for mcan_ex1_loopback.c	5
Figure 3-3. Output of Bus-Monitoring Tool for mcan_ex3_transmit.c	5
Figure 3-4. Output From MCANTX Pin for mcan_ex3_transmit.c	6
Figure 6-1. CCS Error Message	9

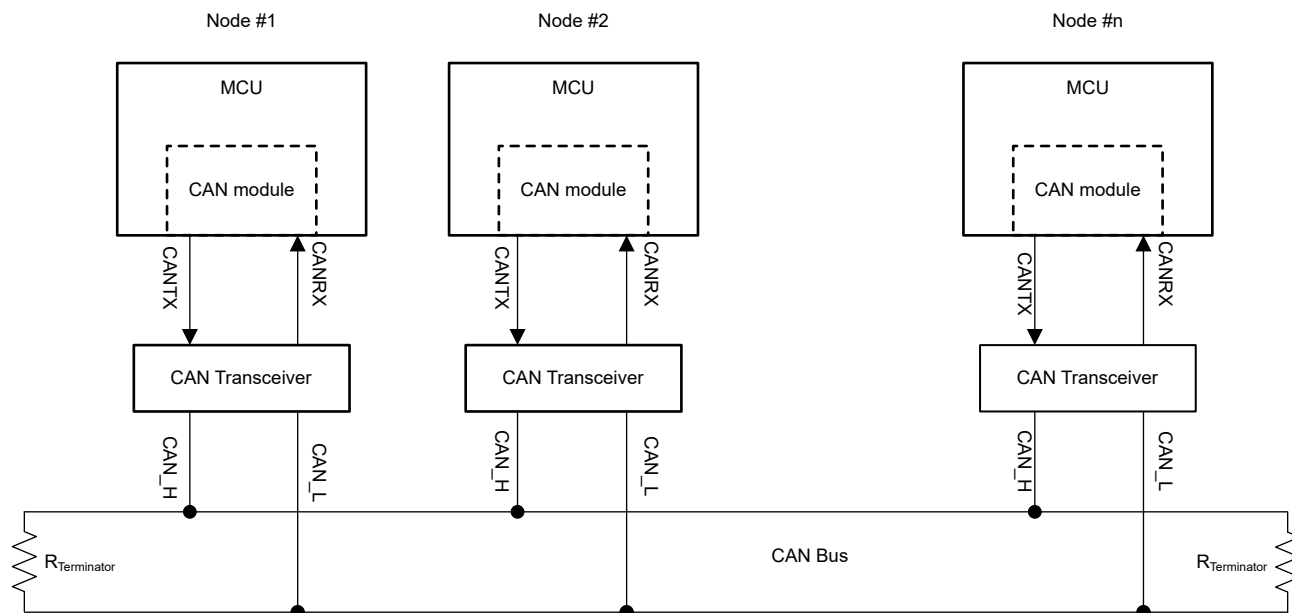
## Trademarks

C2000™ and Code Composer Studio™ are trademarks of Texas Instruments.

All trademarks are the property of their respective owners.

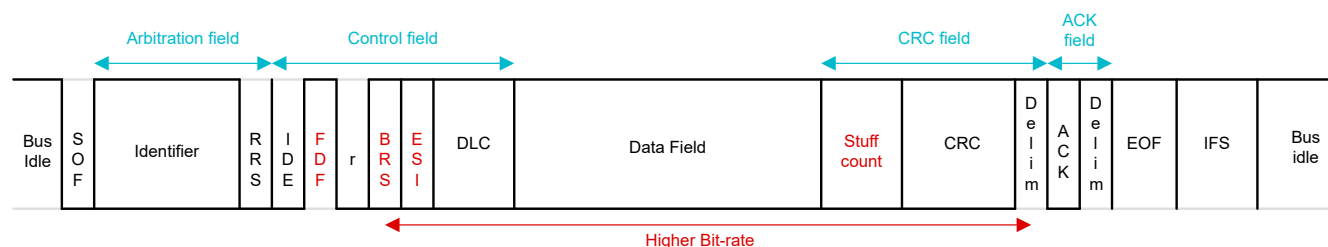
## 1 Introduction

CAN is a serial protocol that was originally developed for automotive applications. Due to its robustness and reliability, it now finds application in diverse areas such as industrial equipment, medical electronics, trains, aircraft, and so forth. CAN protocol features sophisticated error detection and confinement mechanisms and lends itself to simple wiring at the physical level. The original CAN protocol standard is now referred to as "classical" CAN to distinguish it 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 1 Mbps and a payload size of 8 bytes per frame, CAN FD supports bit-rates up to 5 Mbps 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:

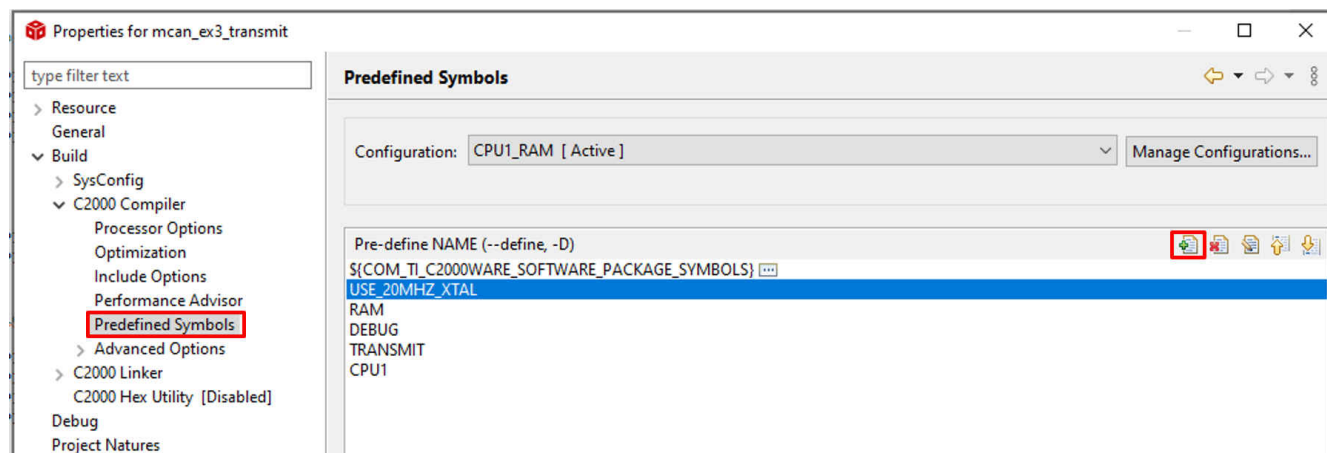
- Conforms with CAN Protocol (ISO 11898-1:2015)
- Full CAN FD support (up to 64 data bytes)
- Up to 32 dedicated transmit buffers
- Configurable transmit FIFO, up to 32 elements
- Configurable transmit queue, up to 32 elements
- Configurable transmit Event FIFO, up to 32 elements
- Up to 64 dedicated receive buffers
- Two configurable receive FIFOs, up to 64 elements each
- Up to 128 filter elements
- Loop-back mode for self-test
- ECC check for Message RAM
- Clock stop and wakeup support
- Timestamp counter

## 2 Useful tips to run the C2000ware examples

On devices like TMS320F2838xD and TMS320F2838xS, the MCAN module may be controlled either by CPU1 (28x) or the Connectivity Manager (M4). This is not applicable for devices with only a 28x CPU. Below are some useful tips for a successful run of the examples:

- For TMS320F2838xD and TMS320F2838xS devices, there may be two versions of some examples, a version that runs on CPU1 (28x) and another that runs on CM (M4). For example, in C2000ware version 3.04, CPU1 examples reside in `C:\ti\c2000\C2000Ware_3_04_00_00\driverlib\2838x\examples\c28x\mcan` and the CM examples reside in `C:\ti\c2000\C2000Ware_3_04_00_00\driverlib\2838x\examples\cm\mcan`. The projects have the same name, so they cannot exist together in the same CCS workspace. Before running any code on the M4 processor, the file `cm_common_config_c28x.c` should first be executed.
- For the examples to run successfully, it is very important to ensure the clock settings are correct. This is based on the input clock to the device. For details, see below note in the device.c file.

"By default, the `Device_init` function assumes 25 MHz XTAL. If a 20 MHz crystal is used, add a predefined symbol "`USE_20MHZ_XTAL`" in your CCS project. If a different XTAL is used, please update the `DEVICE_SETCLOCK_CFG` macro accordingly. For example, if you are running the examples on a board with a 20 MHz input clock to the MCU, make sure `USE_20MHZ_XTAL` is added to the predefined symbols". For details, see [Figure 2-1](#).



**Figure 2-1. Adding the Predefined Symbol in Project Properties**

If you convert a RAM-based example to a Flash based example, it is important to once again add the `USE_20MHZ_XTAL` symbol for the flash-based project as well.

- GPIO configuration: Considering the MCAN signals can be routed to multiple pins, it is important to ensure the correct options are chosen based on your board design. By default, GPIO30 and GPIO31 are chosen for the MCAN function. The following settings are in the `device.h` file :

```
#define DEVICE_GPIO_CFG_MCANRXA    GPIO_30_MCAN_RX // "pinConfig" for MCAN RX
#define DEVICE_GPIO_CFG_MCANTXA    GPIO_31_MCAN_TX // "pinConfig" for MCAN TX
```

GPIO\_30\_MCAN\_RX & GPIO\_31\_MCAN\_TX are defined in `pin_map.h`

- It is possible to communicate between MCAN and eCAN or DCAN, if FDF and BRS bits are disabled. In this case, only classic CAN frames can be used.

### 3 Program Descriptions

`mcan_ex1_loopback.c` – 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 testcase is that it does not need the transceiver, so it can run on boards like the controlCARD. To facilitate easy capturing of data on an oscilloscope, only four bytes of data are transmitted. However, it is transmitted as a CAN FD frame with Bit-rate switching. For more information, see [Figure 3-1](#).

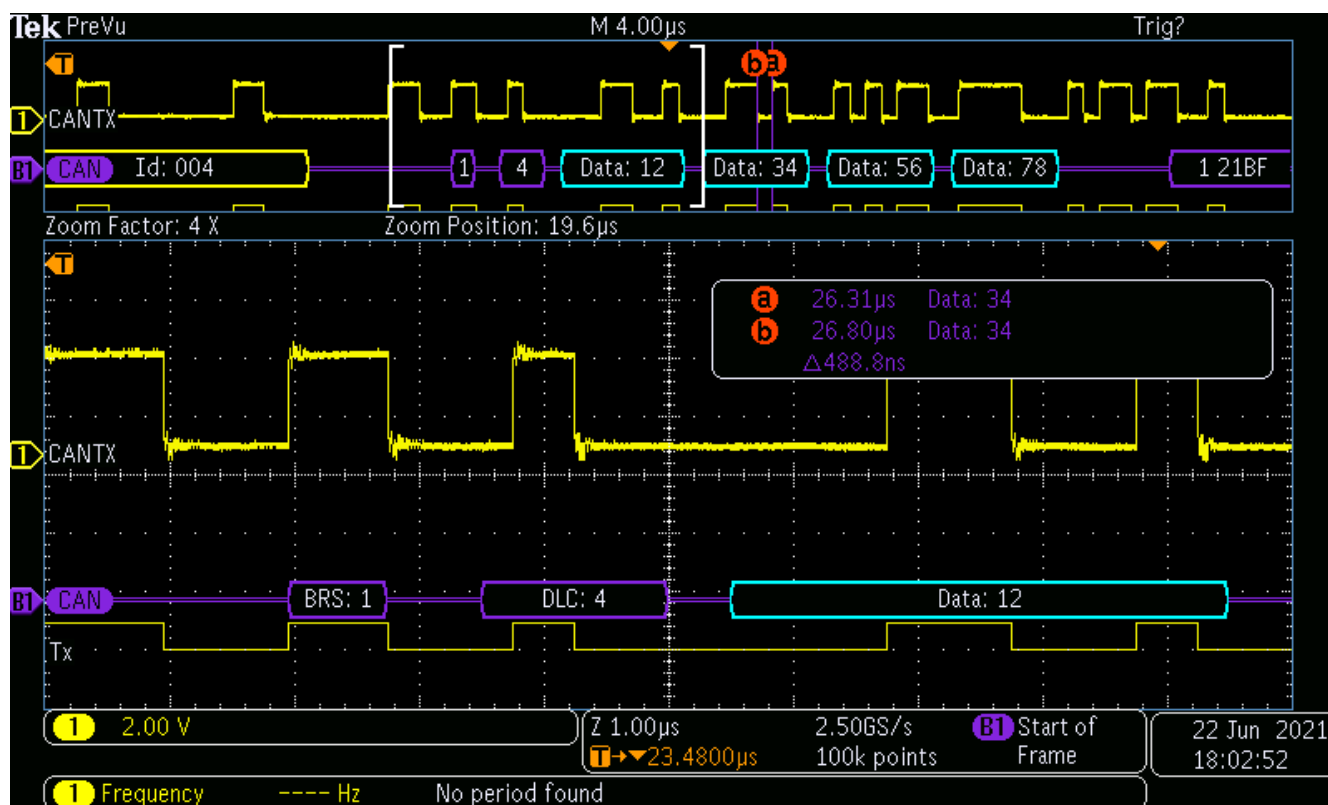


Figure 3-1. Output From MCANTX Pin for `mcan_ex1_loopback.c`

Figure 3-2 shows the output of the CAN bus monitoring tool. Note that both FDF and BRS bits are flagged in the output. The program terminates after transmitting four bytes.

The screenshot shows the 'Receive / Transmit' tab in the CAN bus monitoring tool. A single frame is listed with the following details:

CAN-ID	Type	Length	Data
004h	F0 BRS	4	12 34 56 78

**Figure 3-2. Output of Bus-Monitoring Tool for mcan\_ex1\_loopback.c**

*mcan\_ex3\_transmit.c* - This example demonstrates the MCAN External Transmit function. External communication is done between two CAN nodes. The receiving node could be another MCU or a CAN bus analysis tool capable of Receiving/ACKnowledging the transmitted frames. Both CAN nodes should be connected through a CAN Transceiver. Nominal Bit Rate of 500 kbps and Data bit rate of 1 Mbps is used.

Standard Identifier (STD ID) 0x4 is transmitted with 64 data bytes one time. #defines that are not required for this test case have been commented out. However, they have been left in the code should the scope of this code be expanded to include Receive and FIFO functions.

The screenshot shows the 'Receive / Transmit' tab in the CAN bus monitoring tool. A single frame is listed with the following details:

CAN-ID	Type	Length	Data	Cycle Time	Count
004h	F0 BRS	64	01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40		1

**Figure 3-3. Output of Bus-Monitoring Tool for mcan\_ex3\_transmit.c**

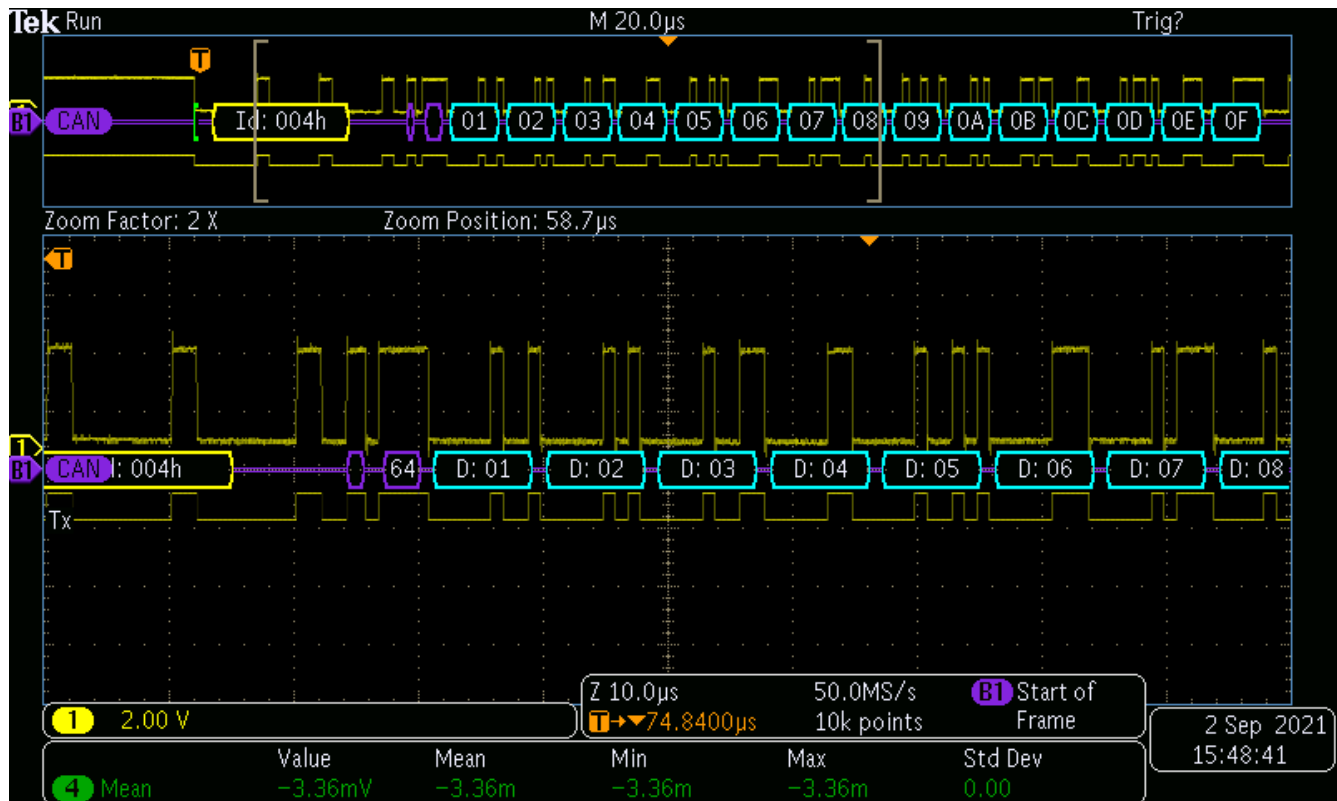


Figure 3-4. Output From MCANTX Pin for `mcan_ex3_transmit.c`

*mcan\_ex4\_receive.c* - This example demonstrates the MCAN receive function. The transmitting node could be another MCU or a CAN bus analysis tool capable of transmitting CAN FD frames. Nominal Bit Rate of 500 kbps & Data bit rate of 1 Mbps is used. Only frames with a standard message ID of 0x4 is received. If another C2000 MCU with MCAN module is used as the transmitter, `mcan_ex3_transmit.c` can be run on it for the transmit function.

*mcan\_ex5\_mask\_filter\_receive.c* - This example demonstrates Reception (with mask filter) into RX-FIFO1. The transmitting node could be a CAN FD capable controller or a CAN bus analysis tool capable of transmitting CAN FD frames. Bits 0, 1 & 3 of the identifier are masked. So, these bits can have any value. This is achieved by using `stdFiltelim.sfid1 = 00000001111b` and `stdFiltelim.sfid2 = 1111110100b` (0 for "don't care"), which means any frame with an ID of 0000000x1xxb are received and stored into the FIFO. That is, frames with the following standard IDs are received: 0x004, 0x005, 0x006, 0x007, 0x00C, 0x00D, 0x00E, 0x00F. All other IDs are not received. Classic bit-mask filter is employed. This example may be used in conjunction with `mcan_ex3_transmit`. Nominal Bit Rate of 500 kbps and Data bit rate of 1 Mbps is used.

*mcan\_ex7\_classic\_transmit.c* - This simple example demonstrates how to transmit classic CAN frames from the MCAN module. This example is very useful when the other node is only capable of processing classic CAN frames. This example is useful to ascertain the external communication capability of the MCAN node. Bit Rate configured is 500 kbps. A frame with an extended identifier of 0x15A5A5A5 is transmitted with 8 data bytes.

*mcan\_ex8\_range\_filter.c* - This example demonstrates Reception (with range filter) into RX-FIFO1. The transmitting node could be a CAN FD capable controller or a CAN bus analysis tool capable of transmitting CAN FD frames. Only frames with extended IDs from 0x1FFFFFF23 to 0x1FFFFFF46 are received. Other IDs are not received. RXFIFO1 starts at an offset of 748 (2EC). MCAN Message RAM starts at 0x58000, so received messages will be copied starting at address 0x582EC. Note that as long as the respective ID bits match, "classic" CAN frames will also be received. Nominal Bit Rate of 500 kbps and Data bit rate of 1 Mbps is used.

## 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 the 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, it expects an acknowledgment (ACK) from at least one other node on the network. Any time a CAN node successfully receives a message it will automatically transmit an ACK, unless that feature has been turned off ("silent mode", 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 it could very well be. (All active nodes on the bus will provide an ACK, regardless of whether they are the intended recipients of that frame or not).

When the transmitting node does not receive an ACK, it results in an ACK error and the transmitting node keeps re-transmitting the frame forever. The Transmit Error Counter (TEC) will increment to 128 and stop there. REC stays at 0. Node will not go bus-off. No interrupts will be generated either. If another node is brought into the network, the TEC will start decrementing (all the way to 0) with every successful transmit.

### 4.2 Why a Transceiver is Needed

One cannot directly connect MCANTX of node-A to MCANRX of node-B and vice versa and expect successful CAN communication. In this case, CAN is unlike other serial interfaces like SCI or SPI. For example, SCI can be made to work with a RS232 transceiver or through a direct connection (SCITX of one node to SCIRX 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 CANTX pin to the CANRX pin of a node. This is because a CAN node needs to be able to monitor its own transmission. Why?

- This has to do with the ACK requirement mandated by the CAN protocol. When a node transmits a frame on the CAN bus, it 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 MSGID needs to be able to override a 1 with a 0. Here again, 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, it loses arbitration.

### 4.3 Debug Checklist

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

#### 4.3.1 Programming Issues

- Is clock to the MCAN module enabled? Check for this if writes to MCAN registers are not going through. Clock is enabled through a bit in the PCLKCRn register.
- Comment all EDIS from your code until you get it to work. You could add it later. Many registers and bits are EALLOW protected and a write may not go through if EALLOW is not active.
- Try your code without interrupts first. Use polling instead. Once polling works, you can add interrupts later.
- When attempting to initiate communication on the bus for the very first time, ensure that the mailbox in the transmitting node and the receiving node are programmed with the same MSGID. Do not use Acceptance Mask Filtering initially. Filtering could be added later once it is confirmed there are no hardware issues.



### 4.3.2 Physical Layer Issues

- Has the bus been terminated correctly (with 120-Ω) at either ends (only)? The bus must be terminated only at either ends and with a 120-Ω resistor. In other words, no more than two terminator resistors may be present on the bus, unless split termination is followed, in which case there will be two resistors on either ends. While designing a CAN bus system, it is important that the termination resistors can be enabled/disabled from outside the system enclosure. This scheme makes it easy when nodes have to be added or removed to/from the network.
- Are all CAN nodes configured for the same bit-rate? Mis-matched node bit rates would repeatedly introduce error frames on the bus. Capture the output of the CANTX pin on the oscilloscope to physically verify the bit-time.
- Have you tried a lower bit-rate? Say, 50 kbps, for example? Timing issues concerning propagation delays may be caught trying a lower bit-rate. Ensure that the NBTP and DBTP registers have the programmed value.
- Have you tried to reduce the bus length and number of nodes?
- Before the occurrence of the error condition, were any error-frames seen on the bus? This could point to 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).

### 4.3.3 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 MCANTX pin of the transmitting MCU and it is of the expected bit-rate. If the expected data is seen at the MCANTX pin, check the data at the MCANRX pin. If the same data is seen at the MCANRX pin, the transceiver is correctly looping back the data.
- If using an oscilloscope with a built-in CAN FD trigger, 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 MSGIDs.
- If the scope does not decode the waveform, 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, it will show incorrect data.
- CAN bus analyzer tool: Make sure bit-rate for both nominal and data phases are correctly configured.

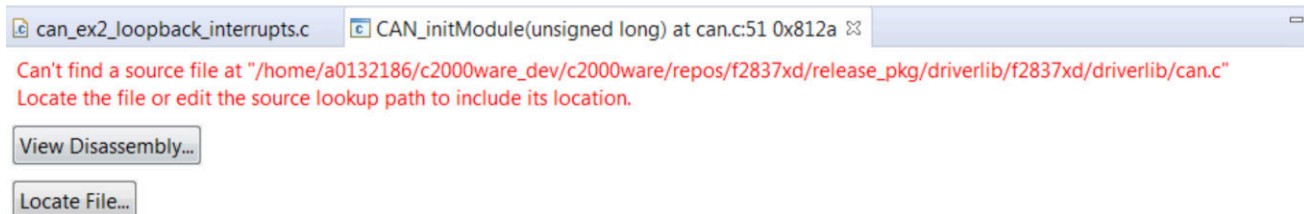
## 5 How to Duplicate (clone) an Existing Project

1. All projects start their life as a *.projectspec* file. They exist in `C:\ti\c2000\C2000Ware_3_04_00_00\driverlib\f2838x\examples\c28x\mcan` directory. Note that the exact path would depend on the version of C2000ware installed in your computer.
2. Make a copy of an existing *.projectspec* file. For example, suppose you want to create a new project called *mcan\_ex6\_transmit\_timestamp*. Start by making a copy of *mcan\_ex1\_loopback.projectspec* and rename it as *mcan\_ex6\_transmit\_timestamp.projectspec*.
3. Open *mcan\_ex6\_transmit\_timestamp* and replace the two instances of *mcan\_ex1\_loopback* with *mcan\_ex6\_transmit\_timestamp*, the name of the new testcase.
4. In the `C:\ti\c2000\C2000Ware_3_04_00_00\driverlib\f2838x\examples\c28x\mcan` directory, make a copy of the *mcan\_ex1\_loopback.c* file and rename it as *mcan\_ex6\_transmit\_timestamp.c*. This is very important because when the *.projectspec* file is imported into Code Composer Studio™, it copies the new file into the target directory when it executes the following statement: `<file action="copy" path="..../mcan_ex6_transmit_timestamp.c" targetDirectory="" />`.
5. Import the *mcan\_ex6\_transmit\_timestamp.projectspec* file into CCS. Note that the project directories are created under `C:\Users\Your_name\your_workspace`, not in: `C:\ti\c2000\C2000Ware_3_04_00_00\driverlib\f2838x\examples\c28x\mcan`.



## 6 How to Get Visibility Into Driverlib Files

While single-stepping through the test cases, if the code calls a function in a Driverlib file, CCS may display an error message as shown in [Figure 6-1](#).



**Figure 6-1. CCS Error Message**

If this happens, click on “Locate File” and point to the Driverlib directory. For example, C:\Users\Your\_name\your\_workspace\mcan\_ex1\_loopback\device\driverlib.

## 7 References

- Texas Instruments: [Introduction to the Controller Area Network \(CAN\)](#)
- Texas Instruments: [Controller Area Network Physical Layer Requirements](#)
- Texas Instruments: [Basics of debugging the controller area network \(CAN\) physical layer](#)
- Texas Instruments: [Calculator for CAN Bit Timing Parameters](#)
- Texas Instruments: [Overview of 3.3V CAN \(Controller Area Network\) Transceivers](#)
- Texas Instruments: [Simplify CAN bus implementations with chokeless transceivers](#)
- Texas Instruments: [Critical Spacing of CAN Bus Connections](#)
- Texas Instruments: [Message priority inversion on a CAN bus](#)
- Texas Instruments: [Programming Examples for the TMS320x28xx eCAN](#)
- Texas Instruments: [Programming Examples for the DCAN Module](#)

## 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](http://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 © 2022, Texas Instruments Incorporated