Application Report
SPRA876B–January 2003–Revised September 2017

Programming Examples for the TMS320x28xx eCAN

Hareesh Janakiraman — Applications Engineering - C2000 Microcontroller

ABSTRACT

The TMS320x28xx series of Microcontrollers/Digital Signal Controllers feature an on-chip enhanced Controller Area Network (eCAN) module. This module is a full-CAN controller, compliant with CAN specification 2.0B. This application report contains several programming examples, supplemental to those in C2000Ware/controlSUITE, to illustrate how the eCAN module is set up for different modes of operation. The objective is to help you come up to speed quickly in programming the eCAN. All programs have been extensively commented to aid easy understanding.

The code examples were tested on a TMS320F28335 device; however, the examples can be easily adapted to run on any C2000 device that features the eCAN module. Most of the examples need CAN-B (the second CAN node) for operation. For parts that have only one CAN module (CAN-A), a second (external) CAN node is needed to emulate the function of CAN-B. This requirement can be met by any CAN bus analysis tool. Many inexpensive USB-bus based CAN bus analysis tools are currently available. These tools provide visibility to the CAN bus traffic and are also capable of generating CAN bus frames and are an invaluable aid in debugging CAN issues. An oscilloscope with built-in CAN bus triggering/decoding is a vital debugging aid as well.

Project files are available for download from the following URL: http://www.ti.com/lit/zip/spra876.

Contents

Trademarks

1 Introduction

CAN is a multi-master serial protocol that was originally developed for automotive applications. Due to its robustness and reliability, it now finds applications in diverse areas such as Industrial automation, appliances, medical electronics, maritime electronics, and so forth. CAN protocol features sophisticated error detection (and isolation) mechanisms and lends itself to simple wiring at the physical level.

Figure 1 shows the typical implementation of the CAN bus.

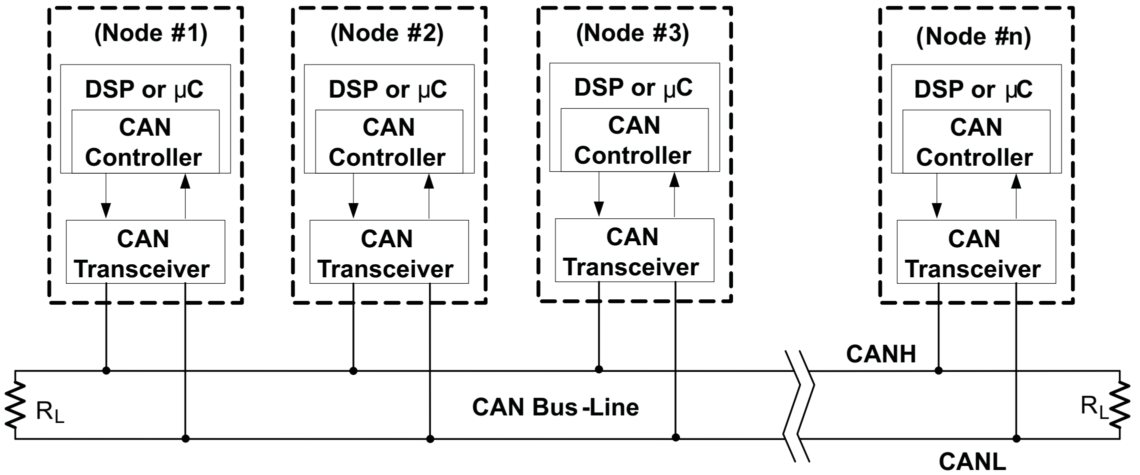

Figure 1. Typical Implementation of a CAN Bus

1.1 *TMS320F28x eCAN Features*

- Full implementation of CAN protocol, version 2.0B
- 32 mailboxes, each with the following properties:
  - Configurable as receive or transmit
  - Configurable with standard or extended identifier
  - Has a programmable receive mask (every mailbox has its own mask)
  - Supports data and remote frame
  - Composed of 0 to 8 bytes of data
  - Uses a 32-bit time stamp on receive and transmit message
  - Protects against reception of new message
  - Holds the dynamically programmable priority of transmit message
  - Employs a programmable interrupt scheme with two interrupt levels
  - Employs a programmable alarm on transmission or reception time-out
- Low-power mode
- Programmable wake-up on bus activity
- Automatic reply to a remote request message
- Automatic retransmission of a frame in case of loss of arbitration or error
- 32-bit Time Stamp Counter (TSC), which can be synchronized by a specific mailbox

2 Programs

The example projects are meant to run seamlessly when installed in the Examples directory of C2000ware. If running the examples on a 2833x/2832x target, unzip the files in C:\ti\c2000\C2000Ware_1_00_01_00\device_support\f2833x\examples directory. For other target devices, the target directory and support files need to be modified.

- CAN_MBXRAMRW

  This example performs repeated writes and reads to the mailbox RAM. Since it exercises the mailbox RAM heavily, it may also be used to check the correct functionality of the mailbox RAM bits.
- CAN_TXLOOP

  This program transmits data to another CAN module. The transmit loop can be executed a predetermined number of times or infinite times. Useful to check the transmit functionality.

- CAN_RXLOOP

  This is an example of how data may be received using polling.
- CAN_RXINT

  This is a simple example of how data may be received using interrupts. It also illustrates the ability of the CAN module to service multiple interrupts automatically.
- CAN_ECHO-AB

  CAN-A transmits to CAN-B, which then echoes the data back to CAN-A. CAN-A then verifies the transmitted and received data. This example can be used to check communication between CAN-A and CAN-B. Both CAN ports of the DSP need to be connected to each other (via CAN transceivers).
- CAN_DBOTX

  Illustrates the operation of DBO field for a Transmit mailbox.
- CAN_DLCTX

  Illustrates the operation of DLC field for a Transmit mailbox.
- CAN_MBXWDIF

  This code illustrates the functionality of the WDIFn bit (WDIF- Write Denied Interrupt Flag).
- CAN_MOTO

  This example illustrates the ”Message Object Time Out (MOTO)” feature.
- CAN_ MULTINT

  This example illustrates the ability of the CAN module to service multiple interrupts automatically. Specifically, this example shows how when an interrupt flag is set while another interrupt flag is already set, the most recent interrupt flag automatically generates a core level interrupt upon exiting the ISR of the previous interrupt.
- CAN_RXMSGLST

  This example illustrates how contents of a mailbox can be protected from being overwritten by using the “Overwrite Protection Control (OPC)” bit.
- CAN_TCOF

  This example illustrates the functionality of the “Timer Counter Overflow Flag (TCOF)” bit.
- CAN_TRPRTSTP

  This program illustrates the programmable transmit-priority and time stamping feature.
- CAN_TXABORT Checks the transmit abort operation using the TRR bit. RXMSGLST.c

  Checks the transmit abort operation using the TRR bit.
- CAN_LPMwakeup

  This example illustrates the ability of the CAN module to enter and exit low-power mode (LPM). Note that this low-power-mode is local to the CAN module and should not be confused with the device-level low-power-modes like HALT, STANDBY and IDLE.
- CAN_REMOTE_fr

  This example illustrates the ability of the CAN module to SEND remote frames from (and receive data frames in) the same Mailbox.
- CAN_LAM

  Illustrates how Acceptance Mask Filtering works using CANLAM registers.
- CAN_jig

  A useful test-jig for debugging CAN bus issues. Can Transmit/Receive Standard and Extended frames at various bit-rates. Also capable of transmitting (and responding to) Remote frames. All options are selected using GPIO0-GPIO3 or through a hard-coded value in the program.

3 Debug and Design Tips to Resolve/Avoid CAN Communication Issues

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

3.1 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 (some CAN implementations feature a so-called "silent mode", where a node receives the frame, but does not provide an ACK; the DCAN module in TMS320F2837xx series MCUs from TI has this feature). 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. In this situation, the TA bit for the transmitting mailbox does not get set. 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.

3.2 Why a Transceiver is Needed

One cannot directly connect CANTX of node-A to CANRX 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.

3.3 Debug Checklist

This section highlights some common mistakes in the design and implementation of a CAN bus network.

3.3.1 Programming Issues

- Is clock to the CAN module enabled? Check for this if writes to CAN registers are not going through. Clock is enabled through a bit in the PCLKCRn register.
- Are 32-bit R/W operations used while accessing the eCAN registers? For an example, see the device-specific TRM.
- 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.
- If a specific mailbox is not working, have you attempted to use a different mailbox? Have the mailboxes been enabled in the CANME register and the mailbox direction correctly configured in the CANMD register?
- Was the MSGGID register initialized to zero, before the individual bits/Bit-fields were configured? Watch out for bits that may come up undefined upon reset.
- Do not use Acceptance Mask Filtering initially. Transmit the same MSGID. Filtering could be added later once it is confirmed there are no hardware issues with the bus.

3.3.2 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/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 a node 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 CANBTR register has the programmed value.
- For devices with an on-chip zero-pin oscillator (INTOSCn) , are you using an external clock source or the INTOSCn? If you are using INTOSCn, do you perform temperature compensation of INTOSCn (for devices where temperature compensation is allowed)?
- 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)

3.3.3 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 CANRX pin of the MCU and it is of the expected bit-rate.
- If using an oscilloscope with a built-in CAN 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.

4 References

- *Introduction to the Controller Area Network (CAN)*
- *Controller Area Network Physical Layer Requirements*
- *Basics of Debugging the Controller Area Network (CAN) Physical Layer*
- *Calculator for CAN Bit Timing Parameters*
- *Overview of 3.3V CAN (Controller Area Network) Transceivers*
- *Simplify CAN Bus Implementations With Chokeless Transceivers*
- *Critical Spacing of CAN Bus Connections*
- Improved CAN Network Security with TI’s SN65HVD1050 Transceiver
- *Message Priority Inversion on a CAN Bus*
- *Piccolo MCU CAN Module Operation Using the On-Chip Zero-Pin Oscillator*

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from A Revision (April 2003) to B Revision	Page
• Title of the document has been updated.	1
• Abstract has been updated.	1
• Update was made in Section 1.	1
• '3.3-V CAN Transceivers' section was removed.	2
• Updates were made to Section 2.	2
• Updates were made to Section 4.	5