Programming Examples and Debug Strategies for the DCAN Module

Hareesh Janakiraman Applications Engineering - C2000 Microcontroller

ABSTRACT
The DCAN peripheral is a CAN implementation on a subset of devices within the C2000 family. Some devices that have a DCAN peripheral include the TMS320F2837xD, TMS320F2837xS, TMS320F2807x and TMS320F28004x devices. The examples are meant to be run in any C2000 MCU with a DCAN module. For a complete list of devices that contain the DCAN module, see the C2000 Real-Time Control Peripherals Reference Guide. This application report describes several programming examples to illustrate how the DCAN module is set up for different modes of operation. The objective is to help you come up to speed quickly in programming the DCAN. All programs have been extensively commented to aid easy understanding. The examples will not run on the eCAN module on the older C2000 devices. For eCAN examples, see Programming Examples for the TMS320x28xx eCAN.

The code examples were tested on a TMS320F28379D device; however, the examples can be easily adapted to run on any C2000 device that features the DCAN 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.

The project files and examples described in this document are available for download as part of C2000Ware.

Contents
1 Introduction ................................................................................................................... 3
2 Program Descriptions ................................................................................................. 3
3 Debug and Design Tips to Resolve/Avoid CAN Communication Issues ...................... 8
4 Helpful Migration and Project Execution Tips ............................................................ 10
5 References .................................................................................................................. 15

List of Figures
1 Typical Implementation of a CAN Bus ........................................................................ 3
2 can_ex1_loopback........................................................................................................ 4
3 can_ex4_simple_transmit........................................................................................... 4
4 Waveform at the CANRXA Pin ................................................................................. 5
5 can_ex8_Remote_Tx-PEAK........................................................................................ 6
6 can_ex8_Remote_Tx-28x............................................................................................ 6
7 can_ex9_Remote_Answer......................................................................................... 7
8 Directory Containing the .projectspec Files ............................................................ 11
9 Project Directories in CCS Workspace ..................................................................... 11
10 CCS Error Message ............................................................................................... 12

List of Tables
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>Extended ID Frame (passing filter criterion)</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>Standard ID Frame (exact match)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Standard ID Frame (passing filter criterion)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>GPIO Pin Mapping Used for the Examples</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>eCAN-DCAN Registers and Bits Equivalence</td>
<td>13</td>
</tr>
</tbody>
</table>

**Trademarks**

Code Composer Studio is a trademark of Texas Instruments.  
All other trademarks are the property of their respective owners.
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](image)

1.1 TMS320F28xx CAN 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
  - Employs a programmable interrupt scheme with two interrupt levels
- Automatic reply to a remote request message
- Automatic retransmission of a frame in case of loss of arbitration or error

2 Program Descriptions

This section provides a brief description of the example projects, along with applicable waveforms captured with an oscilloscope. Note that the examples are within C2000Ware.

- can_ex1_loopback.c

This example illustrates the use of self-test mode. A message is transmitted once per second, using a simple delay loop for timing. The message that is sent is a 2 byte message that contains an incrementing pattern. This example sets up the CAN controller in "External" Loopback test mode. Data transmitted is visible on the CANTXA pin and is received internally back to the CAN Core. It is important that the GPIO mapping in device.h file in this project is edited to reflect the GPIO pins that are used for CAN function in your hardware. Otherwise, the transmitted data will not be seen on CANTXA pin.
Figure 2 shows the activity on the CANTX pin.

![Figure 2](image)

- **can_ex2_loopback_interruptions.c**
  Similar to can_ex1_loopback.c, but uses interrupts.
- **can_ex3_external_transmit.c**
  This example shows basic setup of CAN in order to transmit and receive messages. It sets up CAN-A as the transmitter and CAN-B as the receiver. A receive interrupt is asserted on CAN-B to verify the received data.
- **can_ex4_simple_transmit.c**
  This example is illustrates how to setup the CAN module for transmission. It could prove very useful to check the hardware connections of the CAN circuit. Figure 3 depicts the waveform on the CANTXA pin.

![Figure 3](image)
Figure 4 shows the waveform at the CANRXA pin. Note that during the ACK phase, the transmitting node transmits a recessive, but it is driven low by the receiver.

Figure 4. Waveform at the CANRXA Pin

- **can_ex5_simple_receive.c**
  This example is a simple illustration of how to setup the CAN module for reception. This example could prove very useful to generate an ACK for another CAN node.
- **can_ex8_Remote_Tx.c**
  This example demonstrates the ability of the CAN-A module to transmit a Remote-frame and receive a response in the same mailbox. CAN-B node is configured to respond to the Remote frame. If CAN-B is not available, a CAN bus analyzer may be used to provide a response. Note that the response time from such equipment may be more, because it involves some overhead due to the application running on the PC.
Figure 5 shows the response from a bus analyzer. Note that it takes about 13 ms for the response to show up on the bus.

![Figure 5. can_ex8_R..._Tx-PEAK](image)

Figure 6 shows the response from CAN-B. Response is in microseconds in this case.

![Figure 6. can_ex8_R..._Tx-28x](image)
• can_ex9_Remote_Answer.c

Figure 7 demonstrates the ability of the module to respond to a Remote-frame. A remote frame is transmitted from the CAN bus analyzer and the module responds.

![Figure 7. can_ex9_Remote_Answer](image)

• can_ex10_Mask.c

This example demonstrates acceptance mask filtering. It can be used to evaluate the effects of MXtd & MDir bits. The tables below shows the various scenarios and the outcomes. Mailbox direction was set to Receive (Dir = 0), An extended ID was written to the mailbox (Xtd = 1) and filtering enabled (UMask = 1). MXtd and MDir bits were cycled through the four possible combinations.

• An Ext ID frame, satisfying the filtering criterion, is transmitted. MXtd & Mdir bits have no bearing on the frame reception in all four cases.

<table>
<thead>
<tr>
<th>MXtd</th>
<th>Mdir</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Frame received</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Frame received</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Frame received</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Frame received</td>
</tr>
</tbody>
</table>

• An STD ID frame, with exact match for the 11 applicable bits (bits 28:18), is transmitted. In the case where MXtd is 0, the Xtd bit of the mailbox was not used for filtering. Rather, the 11 applicable bits were found to be matching and, hence, the frame was received.

<table>
<thead>
<tr>
<th>MXtd</th>
<th>Mdir</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Frame not received</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Frame not received</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Frame received</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Frame received</td>
</tr>
</tbody>
</table>
An STD ID frame, with passing filter criterion for the 11 applicable bits (bits 28:18), is transmitted. In the case where MXtd is 0, the Xtd bit of the mailbox was not used for filtering. Rather, the 11 applicable bits were found to be matching and hence the frame was received.

### Table 3. Standard ID Frame (passing filter criterion)

<table>
<thead>
<tr>
<th>MXtd</th>
<th>Mdir</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Frame not received</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Frame not received</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Frame received</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Frame received</td>
</tr>
</tbody>
</table>

# 3 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.

## 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 “silent mode”, where a node receives the frame, but does not provide an ACK; the DCAN module 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 TxRqst 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.
- 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 and the mailbox direction correctly configured?
- When attempting to initiate communication on the bus for the very first time, ensure that the 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 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.
- 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.
- 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.
4 Helpful Migration and Project Execution Tips

This section provides some tips for GPIO reconfiguration, project cloning, migrating from eCAN and bit-timing configuration.

4.1 GPIO Reconfiguration

The example programs run under the Driverlib framework. They were tested with the following GPIO configuration for the CAN pins (see Table 4).

Table 4. GPIO Pin Mapping Used for the Examples

<table>
<thead>
<tr>
<th>CAN Function</th>
<th>GPIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANTXA</td>
<td>GPIO71</td>
</tr>
<tr>
<td>CANRXA</td>
<td>GPIO70</td>
</tr>
<tr>
<td>CANTXB</td>
<td>GPIO72</td>
</tr>
<tr>
<td>CANRXB</td>
<td>GPIO73</td>
</tr>
</tbody>
</table>

The C2000Ware examples, by default, have the following GPIO configuration for CAN operation:

```
#define DEVICE_GPIO_CFG_CANRXA GPIO_30_CANRXA // "pinConfig" for CANA RX
#define DEVICE_GPIO_CFG_CANTXA GPIO_31_CANTXA // "pinConfig" for CANA TX
#define DEVICE_GPIO_CFG_CANRXB GPIO_10_CANRXB // "pinConfig" for CANB RX
#define DEVICE_GPIO_CFG_CANTXB GPIO_8_CANTXB // "pinConfig" for CANB TX
```

This may have to be modified depending on which GPIO pins are used for CAN operation in your hardware. This is a very important step and, if applicable, must be done for every example project. Use the procedure discussed in Section 4.1.1 to change the GPIO configuration.

4.1.1 How to Change the GPIO Assignment for the CAN Pins

The example C file has the following statements:

```
GPIO_setPinConfig(DEVICE_GPIO_CFG_CANRXA);
GPIO_setPinConfig(DEVICE_GPIO_CFG_CANTXA);
GPIO_setPinConfig(DEVICE_GPIO_CFG_CANRXB);
GPIO_setPinConfig(DEVICE_GPIO_CFG_CANTXB);
```

- The function `GPIO_setPinConfig(uint32_t pinConfig)` is in `gpio.c` file.
- The argument `DEVICE_GPIO_CFG_CANRXA` is in `device.h` file.

```
#define DEVICE_GPIO_CFG_CANRXA GPIO_30_CANRXA // "pinConfig" for CANA RX
```

The constant `#define GPIO_30_CANRXA 0x00081C01U` is defined in `pin_map.h` file.

To change the GPIO mapping, edit the `device.h` file as follows: (Table 4 was used as an example).

```
// Modified configuration
#define DEVICE_GPIO_CFG_CANRXA GPIO_70_CANRXA // "pinConfig" for CANA RX
#define DEVICE_GPIO_CFG_CANTXA GPIO_71_CANTXA // "pinConfig" for CANA TX
#define DEVICE_GPIO_CFG_CANRXB GPIO_73_CANRXB // "pinConfig" for CANB RX
#define DEVICE_GPIO_CFG_CANTXB GPIO_72_CANTXB // "pinConfig" for CANB TX
```

**NOTE:** `device.h` file is unique to every example directory in the workspace. The edits you make for one project will not get carried over across other projects.

For the ControlCARD, CANTXA is on GPIO31 and CANRXA is on GPIO30. CANTXB is on GPIO8 and CANRXB is on GPIO10. GPIOs will be different on Launchpad. For more information, see the board schematics in the `/boards` directory in C2000ware.
4.2 How to Duplicate (clone) an Existing Project

1. All projects start their life as a .projectspec file. They exist in
   C:\ti\c2000\C2000Ware_1_00_05_00\driverlib\f2837xd\examples\cpu1\can\CCS directory. Note that the
   exact path would depend on the version of C2000ware installed in your computer.

Figure 8. Directory Containing the .projectspec Files

2. Make a copy of an existing .projectspec file. For example, suppose you want to create a new project
called can_ex4_simple_transmit. Start by making a copy of can_ex1_loopback.projectspec and
   rename it as can_ex4_simple_transmit.projectspec.

3. Open can_ex4_simple_transmit.projectspec and replace the two instances of can_ex1_loopback with
   can_ex4_simple_transmit, the name of the new testcase.

4. In the C:\ti\c2000\C2000Ware_1_00_05_00\driverlib\f2837xd\examples\cpu1\can directory, make a
   copy of the can_ex7_loopback.c file and rename it as can_ex4_simple_transmit.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="../can_ex4_simple_transmit.c" targetDirectory=""/>".

5. Import the can_ex4_simple_transmit.projectspec file into CCS. Note that the project directories are
   created under C:\Users\Your_name\workspace_v8, not in:
   ti\c2000\C2000Ware_1_00_05_00\driverlib\f2837xd\examples\cpu1\can.

Figure 9. Project Directories in CCS Workspace
4.3 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 10.

![Figure 10. CCS Error Message](image)

If this happens, click on “Locate File” and point to the Driverlib directory. For example, `C:\Users\Your_name\workspace_v7\can_ex2_loopback_interrupts\device\driverlib`.

4.4 Migrating From eCAN

This section provides some helpful hints if you are migrating from the eCAN module to the DCAN module. DCAN follows a very different register structure compared to eCAN and hence code written for one module cannot be migrated to another. The following section highlights the differences, but also illustrates the functional similarities between the two modules.

The following features are available in DCAN that are not available in eCAN:
- Parity check mechanism for all RAM modules.
- Automatic Retransmission (upon loss of arbitration) can be disabled.
- Silent mode (Node listens passively).
- Mailbox RAM may be combined to form FIFO buffers.
- Data can be monitored on CANTX pin in self-test mode.

The following are the features that are available in eCAN that are not available in DCAN:
- Timestamping of messages.
- Transmission priority configuration (TPL).
- Data-byte order configuration (DBO).
Table 5 shows the equivalent registers and bit-fields in eCAN and DCAN and also some functional differences.

Table 5. eCAN-DCAN Registers and Bits Equivalence

<table>
<thead>
<tr>
<th>Function</th>
<th>eCAN</th>
<th>DCAN</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN module software reset bit</td>
<td>CANC.SRES</td>
<td>CAN_CTL.SWR</td>
<td></td>
</tr>
<tr>
<td>Automatic bus-on (after bus-off)</td>
<td>CANC.ABO</td>
<td>CAN_CTL.ABO</td>
<td></td>
</tr>
<tr>
<td>Self-test mode</td>
<td>CANC.STM</td>
<td>CAN_CTL.Test</td>
<td></td>
</tr>
<tr>
<td>Configuration mode</td>
<td>CANC.CCR</td>
<td>CAN_CTL.Init</td>
<td></td>
</tr>
<tr>
<td>Configuration mode enabled</td>
<td>CANES.CCE</td>
<td>CAN_CTL.CCE</td>
<td></td>
</tr>
<tr>
<td>Mailbox interrupt source</td>
<td>CANGIFx.MVy</td>
<td>CAN_INT.INTnID</td>
<td></td>
</tr>
<tr>
<td>Interrupt line selection (for mailbox)</td>
<td>CANMIL</td>
<td>CAN_IP_MUX21</td>
<td></td>
</tr>
<tr>
<td>Interrupt line selection (for error &amp; status)</td>
<td>CANGIM.GIL</td>
<td>Hardwired to CANINT0</td>
<td></td>
</tr>
<tr>
<td>CAN to PIE interrupt line0 enable</td>
<td>CANGIM.IOEN</td>
<td>CAN_CTL.IE0</td>
<td></td>
</tr>
<tr>
<td>CAN to PIE interrupt line1 enable</td>
<td>CANGIM.IIEN</td>
<td>CAN_CTL.IE1</td>
<td></td>
</tr>
<tr>
<td>Transmit Error Counter</td>
<td>CANTEC.TEC</td>
<td>CAN_ERRC.TEC</td>
<td></td>
</tr>
<tr>
<td>Receive Error Counter</td>
<td>CANREC.REC</td>
<td>CAN_ERRC.REC</td>
<td></td>
</tr>
<tr>
<td>Bus-off status</td>
<td>CANES.BO</td>
<td>CANES.Boff</td>
<td></td>
</tr>
<tr>
<td>Early warning (TEC or REC = 96)</td>
<td>CANES.EW</td>
<td>CANES.EWarn</td>
<td></td>
</tr>
<tr>
<td>Error passive</td>
<td>CANES.EP</td>
<td>CANES.EPass</td>
<td></td>
</tr>
<tr>
<td>Acceptance mask register</td>
<td>LAM</td>
<td>CAN_IFxMSK</td>
<td></td>
</tr>
<tr>
<td>Acceptance mask filter for a Mailbox</td>
<td>LAM.LAM</td>
<td>CAN_IFnMSK.Msk</td>
<td>Mask bit behavior is opposite</td>
</tr>
<tr>
<td>Mailbox message-ID register</td>
<td>MSGID</td>
<td>CAN_IF1ARB</td>
<td></td>
</tr>
<tr>
<td>Mailbox enable or disable</td>
<td>CANME.MEn</td>
<td>CAN_IFnARB.MsgVal</td>
<td>In DCAN, MBX can remain “enabled” while configuring the MSGID.</td>
</tr>
<tr>
<td>Extended Identifier</td>
<td>MSGID.ID</td>
<td>CAN_IFxARB.Xtd</td>
<td></td>
</tr>
<tr>
<td>Mailbox direction</td>
<td>CANMD.MDn</td>
<td>CAN_IFxARB.Dir</td>
<td></td>
</tr>
<tr>
<td>Message ID</td>
<td>MSGID.ID</td>
<td>CAN_IFxARB.Id</td>
<td></td>
</tr>
<tr>
<td>Lost message indication</td>
<td>CANRML.RMLn</td>
<td>CAN_IFxMCTL.MsgLst</td>
<td></td>
</tr>
<tr>
<td>Transmission request</td>
<td>CANTRS.TRSn</td>
<td>CAN_IFxMCTL.TxRqst</td>
<td></td>
</tr>
<tr>
<td># of bytes in a frame</td>
<td>MSGCTRL.DLC</td>
<td>CAN_IFxMCTL.DLC</td>
<td></td>
</tr>
<tr>
<td>Transmit or Receive priority</td>
<td>Higher numbered MBX has priority</td>
<td>Lower numbered MBX has priority</td>
<td></td>
</tr>
</tbody>
</table>

4.5 Configuring the CANBTR Register

In this section, the usage of the various bit-fields in the CANBTR register is illustrated.

The formula for bit-rate is:

\[ \text{Bit – rate} = \frac{\text{CAN module input clock}}{\text{BRP} \times \text{Bit – time}} \]  

(1)

Where BRP is the value of \((\text{BRP}_{\text{reg}} + 1)\) and \(\text{Bit-time} = (\text{TSEG1}_{\text{reg}} + 1) + (\text{TSEG2}_{\text{reg}} + 1) + 1\)

In the above equations \(\text{BRP}_{\text{reg}}, \text{TSEG1}_{\text{reg}}\) and \(\text{TSEG2}_{\text{reg}}\) represent the actual values written in the corresponding fields in the CANBTR register. The parameters \(\text{TSEG1}_{\text{reg}}, \text{TSEG2}_{\text{reg}}, \text{SJW}_{\text{reg}},\) and \(\text{BRP}_{\text{reg}}\) are automatically enhanced by 1 when the CAN module accesses these parameters. If the BRPE field is used, it should be concatenated with the BRP field.
Example 1.

Assume the following parameters are desired with a CAN module clock of 200 MHz:

Bit-rate = 500 kbps, SJW = 4, TSEG1 = 9 & TSEG2 = 6.

This is achieved by writing a value of 0x000058D8 in the CANBTR register. Here, TSEG1_{reg} = 8, TSEG2_{reg} = 5, SJW_{reg} = 3, and BRP_{reg} = 24

Effective BRP value = (BRP_{reg} + 1) = 25
Effective TSEG1 value = (TSEG1_{reg} + 1) = 9
Effective TSEG2 value = (TSEG2_{reg} + 1) = 6
Effective SJW value = (SJW_{reg} + 1) = 4

Bit-time = (TSEG1_{reg} + 1) + (TSEG2_{reg} + 1) + 1 = 16

Plugging these numbers into the bit-rate equation:

\[
\text{Bit} – \text{rate} = \frac{\text{200 MHz}}{25 \times 16} = 500 \text{ kbps}
\]

(2)

Example 2.

Assume the following parameters are desired with a CAN module clock of 200 MHz:

Bit-rate = 50 kbps, SJW = 4, TSEG1 = 9 & TSEG2 = 6.

This is achieved by writing a value of 0x000358F9 in the CANBTR register. Here, TSEG1_{reg} = 8, TSEG2_{reg} = 5, SJW_{reg} = 3, BRPE = 11b, BRP = 111001b. Effective BRP_{reg} = 11111001b (249). Note that BRPE field is concatenated with the BRP field.

Effective BRP value = (BRP_{reg} + 1) = 250
Effective TSEG1 value = (TSEG1_{reg} + 1) = 9
Effective TSEG2 value = (TSEG2_{reg} + 1) = 6
Effective SJW value = (SJW_{reg} + 1) = 4

Bit-time = (TSEG1_{reg} + 1) + (TSEG2_{reg} + 1) + 1 = 16

Plugging these numbers into the bit-rate equation:

\[
\text{Bit} – \text{rate} = \frac{\text{200 MHz}}{250 \times 16} = 50 \text{ kbps}
\]

(3)
5 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: *Improved CAN Network Security with TI’s SN65HVD1050 Transceiver*
- Texas Instruments: *Message Priority Inversion on a CAN Bus*
- Texas Instruments: *Piccolo MCU CAN Module Operation Using the On-Chip Zero-Pin Oscillator*
- Texas Instruments: *Programming Examples for the TMS320x28xx eCAN*